Differential program semantics

Thibaut Girka

To cite this version:

Thibaut Girka. Differential program semantics. Programming Languages [cs.PL]. Université Paris Diderot, 2018. English. NNT : . tel-01890508v1

HAL Id: tel-01890508
https://inria.hal.science/tel-01890508v1

Submitted on 8 Oct 2018 (v1), last revised 14 Jan 2020 (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.

Thèse de doctorat
de l'Université Sorbonne Paris Cité
Préparée à l'Université Paris Diderot
Ecole Doctorale 386 — Sciences Mathématiques de Paris Centre
Institut de recherche en informatique fondamentale (I.R.I.F.),
équipe Preuves Programmes Systèmes

Differential program semantics

Par Thibaut Girka

Thèse de doctorat d'Informatique

Dirigée par

Roberto Di Cosmo	Directeur de thèse
David Mentré	Co-directeur de thèse
Yann Régis-Gianas	Co-directeur de thèse

Présentée et soutenue publiquement à Paris le 3 juillet 2018

Présidente du jury :	Lawall, Julia	Directrice de Recherche	Inria
Rapporteurs :	Barthe, Gilles	Research Professor	IMDEA Software Institute
	Pichardie, David	Professeur	ENS Rennes
Examinateurs :	Cohen, Julien	Maître de Conférences	Université de Nantes
Directeur de thèse :	Di Cosmo, Roberto	Professeur	Université Paris Diderot
Co-directeurs de thèse :	Mentré, David	Research Manager	Mitsubishi Electric R&D Centre Europe
	Régis-Gianas, Yann	Maître de Conférences	Université Paris Diderot

Except where otherwise noted, this work is licensed under
http://creativecommons.org/licenses/by-sa/4.0/

Acknowledgments & Remerciements

Comme il est de coutume, il convient de commencer ces remerciements par mes directeurs de thèse. Merci, donc, tout d'abord, à Roberto Di Cosmo, pour avoir accepté de diriger ma thèse, et pour avoir suivi de loin mon travail. Merci aussi à mon codirecteur David Mentré, pour son regard extérieur sur mes travaux, pour avoir supporté mon librisme quotidien, et grâce à qui les développements logiciels réalisés lors de cette thèse peuvent être distribués sous licence libre. Merci enfin et surtout à Yann Régis-Gianas, mon directeur de thèse officieux, pour son encadrement scientifique inestimable et pour avoir su faire face, pendant ces longues années, à mon pessimisme régulier.

Merci à David Pichardie et à Gilles Barthe pour avoir accepté de rapporter ma thèse. Merci également à ce second pour ses travaux qui ont été une importante source d'inspiration, en témoigne le Chapitre 2 de cette thèse. Merci à Julien Cohen pour avoir accepté de faire partie du jury, et à Julia Lawall pour avoir accepté de le présider.

Thanks also to J Strother Moore for helping me keep my stress to manageable levels during my first participation to an international conference.

La majorité de mes travaux ayant été menés à ~~PPS~~ l'IRIF, il serait mal venu de ne pas remercier les nombreux collègues que j'y ai côtoyés. Merci donc, tout d'abord, aux collègues partageant mon bureau, en commençant par celui dont la ponctualité est plus que discutable et qui entretient une fascination quelque peu morbide pour toutes sortes de dictateurs. Merci également à son acolyte à peine plus ponctuelle, avec qui les discussions politiques ou historiques ne manquent pas. Merci aussi à l'aîné du 3033, qui n'y vient plus troller qu'à temps partiel.

Il convient ensuite de remercier les doctorants un peu plus lointains, tels que ceux qui ont aménagé leur bureau d'un canapé ma foi fort confortable, ou celles et ceux qui ont orchestré l'emprunt d'un manuscrit extraordinaire ainsi que la numérisation de ses 777 pages…

Des remerciements sont également dûs aux courageux étudiants d'un directeur de thèse particulièrement insaisissable, reclus ensembles au fond d'un même bureau, qu'il s'agisse d'un doctorant extrêmement bavard, d'un autre qui semble parfois être sous le coup d'une malédiction, ou de ce dernier dont la foi ne semble pas affecter l'ouverture d'esprit et qui donne souvent l'impression d'être monté sur ressorts.

Il me faut aussi remercier mes anciens collègues, tels que le fameux « surfer australien » germanique, ou la ribambelle de docteurs tous plus brillants les uns que les autres, comme la personne m'ayant fait découvrir un des pires groupes de chanteuses qu'il m'ait été donné d'écouter[1], l'homme en short à l'humour redoutable, la *workaholic* qui emmenait ses collègues boire pour perfectionner son Français, ou la championne du syndrome de l'imposteur qui a finalement soutenu l'équivalent d'au moins deux thèses.

Merci également aux stagiaires qui nous ont rendu visite ainsi qu'à mes autres collègues de l'IRIF, que j'ai moins connus ou qu'un moment d'égarement m'aurait fait ne pas citer.

En dehors du monde académique, je tiens également à remercier un ami de longue date dont la radicalisation s'est grandement accélérée suite à son séjour outre manche, l'ex-tartine cubique, l'homme que tout le monde connaît[2] et qui a joué un rôle certain dans mon choix de me diriger vers la recherche, l'administrateur réseau dont l'addiction au café en a même affecté son pseudonyme, le grand curieux adepte du *shitpost* et des arcs-en-ciel, le barbu reconnu pour ses jeux de mots et par sa façon systématique de partager le moindre *selfie*, l'esperluette abattant une quantité de travail redoutable et dont l'enthousiasme est plus que communicatif, ainsi que le petit curieux dont le cœur balance entre le serpent et la pierre précieuse et à qui je dois beaucoup de soutien.

Thanks to codemom, for her invaluable work and her support. Thanks to the multiple owners of cuddly dinosaurs for sharing their cuteness online. Thanks also to gargamel, whose work has played a significant role in me meeting or keeping in touch with many of the people I have already thanked in those last two paragraphs.

Merci à toutes les personnes qui participent et font vivre des *hackerspaces* inclusifs, à toutes les personnes qui se battent pour préserver les libertés individuelles sur Internet ou hors ligne, qui militent activement pour davantage de justice sociale, et pour rendre le monde un peu meilleur à leur façon.

Merci à Jany Belluz pour la police Fantasque Sans Mono[3] utilisée pour les extraits de code, ainsi qu'à Black[Foundry] pour la police Inria Serif utilisée pour le reste de cette thèse[4].

Merci, enfin, à toutes les autres personnes formidables qui sont malheureusement parvenues, d'une manière ou d'une autre, à échapper jusqu'ici à ces remerciements.

[1] Aussi bien musicalement qu'au niveau de l'idéologie véhiculée.

[2] Et qui a tendance à abuser des notes de bas de page.

[3] Maladroitement augmentée de quelques glyphes par l'auteur de cette thèse.

[4] Il n'est d'ailleurs pas impossible que le titre à rallonge de ce chapitre ne soit en fait qu'une excuse pour utiliser une esperluette et de multiples ornements.

Abstract

Computer programs are rarely written in one fell swoop. Instead, they are written in a series of incremental changes. It is also frequent for software to get updated after its initial release. Such changes can occur for various reasons, such as adding features, fixing bugs, or improving performances for instance. It is therefore important to be able to represent and reason about those changes, making sure that they indeed implement the intended modifications.

In practice, program differences are very commonly represented as *textual differences* between a pair of source files, listing text lines that have been deleted, inserted or modified. This representation, while exact, does not address the semantic implications of those textual changes. Therefore, there is a need for better representations of the *semantics* of program differences.

Our first contribution is an algorithm for the construction of a *correlating program*, that is, a program interleaving the instructions of two input programs in such a way that it simulates their semantics. Further static analysis can be performed on such correlating programs to compute an over-approximation of the semantic differences between the two input programs. This work draws direct inspiration from an article by Partush and Yahav[32], that describes a correlating program construction algorithm which we show to be unsound on loops that include `break` or `continue` statements. To guarantee its soundness, our alternative algorithm is formalized and mechanically checked within the Coq proof assistant.

Our second and most important contribution is a formal framework allowing to precisely describe and formally verify semantic changes. This framework, fully formalized in Coq, represents the difference between two programs by a third program called an *oracle*. Unlike a *correlating program*, such an *oracle* is not required to interleave instructions of the programs under comparison, and may “skip” intermediate computation steps. In fact, such an oracle is typically written in a different programming language than the programs it relates, which allows designing *correlating oracle languages* specific to certain classes of program differences, and capable of relating crashing programs with non-crashing ones.

We design such *oracle languages* to cover a wide range of program differences on a toy imperative language. We also prove that our framework is at least as expressive as Relational Hoare Logic by encoding several variants as correlating oracle languages, proving their soundness in the process.

Résumé

La comparaison de programmes informatiques est au cœur des pratiques actuelles de développement logiciel : en effet, les programmes informatiques n'étant pas écrits d'un seul coup, mais par des modifications successives souvent effectuées par plusieurs programmeurs, il est essentiel de pouvoir comprendre chacune de ces modifications.

Des outils, tels que `diff` et `patch` existent pour traiter des différences entre programmes. Cependant, ces outils représentent les différences comme la liste des lignes modifiées, ajoutées ou supprimées du code source d'un programme, et ne traitent pas de sa sémantique.

Il convient donc de trouver des façons de représenter et raisonner à propos des différences entre programmes au niveau sémantique, ce à quoi cette thèse s'attelle.

Programmes de corrélation

La première partie de notre travail se concentre sur l'inférence automatique de différences sémantiques entre deux programmes. Dans cette optique, nous avons repris un article d'Eran Yahav et Nimrod Partush[32] décrivant une technique pour calculer une sur-approximation des variables ayant une valeur différente à l'issue de l'exécution des deux programmes. Pour ce faire, l'approche décrite par Eran Yahav et Nimrod Partush consiste à produire un *programme de corrélation* entrelaçant statiquement les instructions des deux programmes de façon à ce que le programme de corrélation simule les deux programmes à comparer. Ce programme de corrélation est alors analysé par un outil d'interprétation abstraite utilisant un domaine relationnel pour tenter de préserver des relations—si possible d'égalité—entre les variables des deux programmes.

Malheureusement, l'algorithme décrit pour calculer les *programmes de corrélation* est incorrect. En effet, nous montrons qu'en présence d'instructions `goto`, `break` ou `continue`, le programme de corrélation engendré par l'algorithme proposé ne simule pas toujours les programmes à comparer.

Pour remédier à cela, nous proposons un nouvel algorithme de calcul de programmes de corrélation. Contrairement à l'approche très textuelle de l'algorithme de Nimrod Partush et Eran Yahav, cet algorithme repose sur une notion de différence syntaxique structurée, qui représente la différence entre deux arbres de syntaxe abstraite. Cette représentation très syntaxique et structurée des différences facilite grandement

l'écriture de notre algorithme, qui prend la forme d'une fonction définie récursivement sur la structure de la différence syntaxique. Cette définition récursive permet à son tour une preuve par induction structurelle.

En effet, contrairement à l'article d'origine, nos travaux sont formalisés et prouvés corrects grâce à l'assistant de preuve Coq. Cette formalisation nous paraît nécessaire compte tenu de la facilité avec laquelle il est possible de faire des erreurs lors de la conception de ce type d'algorithmes, comme en témoigne l'erreur présente dans l'article d'origine.

Oracles de corrélation

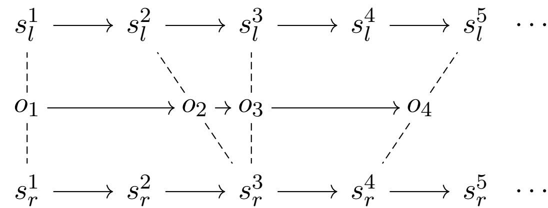
Pour la suite de notre travail, nous nous sommes intéressés à la spécification et à la vérification de différences sémantiques, plutôt qu'à leur inférence automatique. Dans ce contexte, nous proposons un cadre formel permettant de décrire et représenter des différences sémantiques entre programmes. En effet, nous partons du principe que la personne qui effectue une modification le fait avec une intention précise, qu'il pourrait décrire si un formalisme le lui permettait. Une fois cette intention formellement décrite, il serait possible de prouver que la modification effectuée y correspond, que ce soit de manière automatique ou en écrivant une preuve manuellement.

À la manière des *programmes de corrélation*, nous avons décidé de représenter la différence entre deux programmes par un troisième programme. Cependant, contrairement à un programme corrélé, ce troisième programme—que nous appelons *oracle de corrélation*—n'est pas forcément un entrelacement des instructions des deux autres programmes, et est autorisé à « sauter » des instructions des programmes à comparer. Un tel oracle de corrélation n'est d'ailleurs généralement pas écrit dans le même langage de programmation que les programmes qu'il met en relation, mais dans un langage spécialisé. Un tel langage spécialisé, ou *langage d'oracles*, caractérise un type de différences spécifique.

Plus précisément, afin de pouvoir mettre en relation des programmes qui ne terminent pas, ainsi que des programmes susceptibles de *planter*, nous nous plaçons dans le cadre de la sémantique opérationnelle à petits pas : un langage de programmation est défini entre autres par des *règles de réduction* décrivant comment les programmes s'exécutent. L'exécution d'un programme donné est donc une succession d'états obtenus par des réductions—ou petits pas—successives.

Dans ce cadre, un langage d'oracles est aussi défini par une sémantique opérationnelle à petit pas. Un état de l'oracle induit une relation entre les états des programmes *gauche* et *droit* mis en relation. Chaque pas de l'oracle correspond alors à un certain nombre de pas dans les programmes à mettre en relation.

Le schéma suivant illustre l'exécution d'un exemple d'oracle—dont les états successifs sont o_1, o_2 etc.—ainsi que des programmes gauche—dont les états successifs sont s_l^1, s_l^2, etc.—et droite—s_r^1, s_r^2, etc.—mis en relation. Les flèches représentent des réductions, et les lignes pointillées les relations entre états induites par un état d'oracle :



En plus de sa sémantique opérationnelle à petits pas, un langage d'oracles est défini par une propriété définie sur les états de l'oracle, par exemple que les états mis en relation assignent bien la même valeur à certaines variables. Cette propriété, appelée *invariant*, sert à la fois à caractériser la relation que l'on veut montrer entre les deux programmes, et comme outil de preuve qu'un oracle simule bien deux programmes donnés, c'est-à-dire que chaque pas de l'oracle correspond effectivement à un nombre non-nul de pas des programmes gauche et droite.

Afin d'évaluer le cadre formel esquissé plus haut, nous avons écrit un ensemble de langages d'oracles pour un langage impératif jouet appelé Imp. Ces langages d'oracles couvrent aussi bien des modifications qui n'influent pas sur le résultat final de l'exécution des deux programmes—comme l'échange de deux affectations successives indépendantes ou l'échange de deux branches d'une conditionnelle—que des modifications qui changent profondément la sémantique des deux programmes—comme la correction d'une erreur fatale en rajoutant une condition défensive, ou la modification de la valeur d'une variable qui n'influe pas sur le flot de contrôle du programme.

Tout comme pour la première partie de notre travail, ce cadre formel ainsi que les différents langages d'oracles sur Imp ont été formellement définis à l'aide de l'assistant de preuve Coq et toutes les preuves de correction ont également été vérifiées grâce à cet outil.

Langages de différences

La finalité de notre travail étant de permettre à des programmeurs de décrire leurs modifications et de vérifier qu'elles correspondent bien à leurs intentions, il convient de leur proposer un langage homogène pour le faire. Nous avons donc défini une notion de *langage de différences*, permettant de composer des différences primitives pour décrire les différences sémantiques entre deux programmes.

Un langage de différences fournit une syntaxe pour décrire des relations entre traces d'exécution. En pratique, nous utilisons des *oracles* pour caractériser ces traces et vérifier la correction des différences décrites.

Nous avons défini un langage de différence jouet sur Imp en donnant une syntaxe aux langages d'oracles évoqués dans la section précédente.

Contributions

Les contributions principales de cette thèse sont :

- Un exemple montrant que l'algorithme de calcul de *programme de corrélation* proposé par Eran Yahav et Nimrod Partush[32] est incorrect en présence d'instructions `goto`, `break` ou `continue` (Chapitre 1).
- Un algorithme formellement prouvé correct pour calculer un *programme de corrélation* similaire à ceux d'Eran Yahav et Nimrod Partush (Chapitre 1).
- Le cadre formel d'*oracles de corrélation*, formellement défini en Coq, pour décrire précisément des différences sémantiques entre programmes à l'aide d'un troisième programme appelé *oracle de corrélation* (Chapitre 3).
- Un ensemble de *langages d'oracles* utilisant ce cadre formel pour décrire une douzaine de classes de différences sur le langage impératif jouet Imp (Chapitre 4).
- De nouvelles preuves de correction de plusieurs variantes de la Logique Relationnelle de Hoare obtenues en les encodant sous forme de langages d'oracles, donnant une sémantique opérationnelle à petits pas aux preuves de Logique Relationnelle de Hoare (Chapitre 5).
- Un exemple de *langage de différences* sur Imp, faisant usage des langages d'oracles définis précédemment (Chapitre 6).

Contents

Acknowledgments & Remerciements 1

Abstract 3

Résumé 5

Contents 9

Introduction 11
- Textual differences and informal messages . . . 12
- Inferring properties from a textual change . . . 14
- Towards formal descriptions of program changes . . . 16
- Contributions and plan . . . 17

1 Correlating Programs 21
- 1.1 Unsound handling of `goto` statements . . . 23
- 1.2 Formal definition of the Imp_{br} language . . . 25
- 1.3 Formal definition of the Guarded Language . . . 27
- 1.4 Translation to guarded form . . . 28
- 1.5 Structured program differences . . . 31
- 1.6 Generation algorithm directed by structured differences . . . 33
- 1.7 Implementation and experiments . . . 38

2 Product programs 41
- 2.1 Imp . . . 42
- 2.2 Hoare Logic . . . 44
- 2.3 Relational Hoare Logic . . . 45
- 2.4 Product programs . . . 49
- 2.5 Shortcomings of RHL and Product Programs . . . 51

3 Correlating oracles 53
- 3.1 General definition of a programming language . . . 55
- 3.2 Correlating oracles: operational semantics for differences . . . 56

4 Oracle languages for Imp **63**
4.1 Renaming . . . 65
4.2 Control-flow-preserving value changes . . . 66
4.3 Branches swapping . . . 69
4.4 Refactoring branches with common code . . . 72
4.5 Sequence Associativity . . . 74
4.6 Independent assignments swapping . . . 76
4.7 Contextual equivalence . . . 78
4.8 Crash-avoiding conditionals . . . 80
4.9 Off-by-one crash fixes . . . 82

5 Encoding Relational Hoare Logic **85**
5.1 Minimal RHL . . . 85
5.2 Core RHL . . . 89
5.3 Core RHL extended with self-composition . . . 91

6 Difference languages **93**
6.1 Trace relations . . . 93
6.2 Difference languages . . . 95

7 Coq implementation **101**
7.1 Code metrics and layout . . . 101
7.2 Walk through the code . . . 103

8 Related work **115**
8.1 Coccinelle and semantic patches . . . 116
8.2 Correlating programs and speculative correlation . . . 116
8.3 Other automatic inference of program differences . . . 117
8.4 Product programs . . . 117
8.5 Bisimulation . . . 118
8.6 Refinement mappings . . . 119

Conclusion **121**
Contributions . . . 121
Limitations . . . 121
Other perspectives . . . 123

Bibliography **125**

Introduction

Much like other kinds of creative or engineering activities, software development is a process: as one can imagine, programs are rarely written in one fell swoop, but rather developed over an extended period of time. Indeed, large programs are always written in a series of incremental changes. In fact, common programming methodologies encourage starting from minimal programs and progressively adding and generalizing functionality with consistent, separate and minimal changes. Many software projects also have long life cycles, with increasingly frequent updates after an initial release: newer versions get released for various reasons, such as fixing bugs, improving performances, adding requested features, maintaining compatibility with other software, or addressing security issues. All those updates constitute program changes, possibly written by multiple developers.

All of the aforementioned changes are made of modifications to the source code of the program, that is the human-readable set of instructions that is later processed by a computer to execute the program. Those changes to the source code may consequently change its semantics—that is, the program's meaning, how it actually behaves and what it computes. Each one of those changes is made with a precise intent in mind, such as fixing a bug, adding a feature, or simply making the code more readable to other programmers. It is common practice for programmers to explain that intent and the reasoning behind the change—also called a *patch*, from when programs were written on paper tapes or punched cards, and modifying an existing program required physically cutting and patching it—to other programmers.

In fact, this notion of program differences is so central to the practice of programming that nearly all software development is done using a *version control system*[20, 38] such as `git`[15] or `mercurial`[28], a tool that stores successive versions of given documents—typically source code—and is often structured around *changesets*, sets of consistent differences between versions accompanied by a *commit message*, an informal description of the change given by the person who wrote it.

As discussed before, changes made to a program are made with a specific intent which is typically described in a free-form informal text message accompanying the change, but the patch may have unintended consequences: it may add new bugs—also called *regressions*—or it may be downright incorrect, not matching the intent of the programmer. Verifying that the patch indeed corresponds to the programmer's intent and does not introduce new issues[37] is the main motivation behind the common practice of *code review* in which at least one other programmer reviews the proposed

patch before it gets included in the project.

To summarize, *patches* are pervasive and essential to software development, and as such, understanding them and reasoning about them is an important topic of software development[21]. Characterizing patches, and in particular their impact on the *semantics* of programs, is essential. An appropriate characterization of program differences may help programmers understand their changes, assist them with code review, or limit the number of *regression tests*—tests written in order to spot unwanted changes in program behavior when updating it—to run[8].

Textual differences and informal messages

By far the most common characterization of program differences are textual differences—as produced and processed by the `diff`[18, 27] and `patch`[27] utilities—together with informal free-form textual messages from the programmer. Both of those representations are typically displayed together in version control systems and code review platforms.

A textual difference, or *diff* for short, is an unambiguous, formal—if low-level—representation of the change on the program's source code seen as an unstructured piece of text. Such a *diff* lists modified portions of the document, presenting lines that are either removed from the original version, added in the modified version or kept unchanged between the two versions. Lines containing modifications but otherwise present in both versions are considered removed from the original version and added to the modified version. Those removed and added lines are accompanied by the location at which they appear in both versions of the program, so that the *patch* can be mechanically *applied* to the old program to get the new one, as does the `patch` utility. They are also commonly surrounded by some unmodified lines, partly in order to provide some context to a person reading the patch, and partly as a means for the `patch` utility to check that it is being applied to the intended file—it will fail if it cannot find a list of lines matching the context.

For instance, let us consider two versions of a sample C program:

Left (original) version

```
sum = 0;
x = -x;
y = 0;
while (sum < x) {
  y = y + 1;
  sum = sum + 1 + 2 * y;
}
sum = 0;
```

Right (modified) version

```
sum = 0;
x = -x;
count = 0;
while (sum < x) {
  count = count + 1;
  sum = sum + 1 + 2 * count;
}
sum = 0;
```

In the example above, the right program has been obtained by *renaming* every occurrence of the variable “`y`” in the left program to “`count`”. The intent behind this change would typically have been to make the code more readable by using more

meaningful variable names. In any case, this difference would typically be represented as the following *diff*:

```
 sum = 0;
 x = -x;
-y = 0;
+count = 0;
 while (sum < x) {
-  y = y + 1;
-  sum = sum + 1 + 2 * y;
+  count = count + 1;
+  sum = sum + 1 + 2 * count;
 }
 sum = 0;
```

On the one hand, as stated above, textual differences describe the changes between two programs' source code precisely and unambiguously. They are formal representations of those changes, so they can be automatically manipulated by computer programs, and, to some extent, reasoned about. However, they are also purely textual. This means the `diff` and `patch` utilities work for any text-based programming language, but they cannot exploit the programming language's syntax or semantics in any way. A *diff* thus cannot efficiently convey the meaning of the programmer nor provide any insight on the way the change actually affects the program's execution. Depending on the nature of the described change, they can also be needlessly verbose: for instance, in the example above, almost every line is affected by the patch, even though it can be summarized as renaming the variable "`y`" to "`count`".

On the other hand, *commit messages* are free-form textual messages written by and meant for programmers. They usually explain the intent the programmer had when making that change, make claims about the expected program semantics, and explain the reasoning behind it. Being free-form, though, commit message are neither exact nor complete, and cannot be automatically manipulated by computer programs in any meaningful way. Furthermore, since this message is informal, it can also be understood differently by different programmers, and it can be difficult to verify that it actually corresponds to the exact program difference. In general, such change descriptions provide valuable insight from the programmer, but cannot be used in a formal or automated setting.

Those two very common kinds of change descriptions are thus complementary. In particular, during code review, reviewers analyze the *patch* in light of the *commit message*, but as explained earlier, this has to be a manual—hence error-prone—operation.

For this reason, we are interested in providing ways to reason about the semantic implications of program changes. While there are several existing theoretical constructs to semantically relate pairs of programs[25], we are especially interested in practical tools that could be used by programmers to formally specify *relational properties*—that is, properties referencing multiple programs—and, ultimately, mechanically

verify them. To this end, we have initially investigated automatically inferring relational properties from textual changes. However, while that approach may in some cases be successful at proving or disproving that two programs are equivalent, the automatically-inferred properties may not actually be of any interest to the programmer. Indeed, the result of such an analysis is purely extensional, making it difficult to reason about intensional properties such as optimizations. Furthermore, it is also somewhat low-level, as it describes relations between the final values of both programs' variables with linear equations. Finally, it is often imprecise: due to the nature of the relational analysis, some relations cannot be described precisely. In other cases, achieving enough precision to prove expected properties is possible, but requires the programmer to manually instruct the tool with very specific low-level hints, which should not be required from them to formally *specify* their intent. For these reasons, our focus moved to developing new structured and semantics-focused descriptions of program changes that allow programmers to *specify* the properties they are interested in and allowing them to subsequently prove them.

Inferring properties from a textual change

The first approach we have explored to understand program differences is to automatically infer properties relating both programs from a textual change. For instance, one can imagine a tool providing a sound approximation of the set of variables that have different values at the end of the execution of both programs, thus highlighting their similarities and differences.

We first attempted building such a tool. In fact, as we started working on that idea, we found very similar existing work by Eran Yahav and Nimrod Partush[32]. Their approach consisted in building a single *correlating program* simulating the parallel execution of two successive versions of a same program. The purpose of this *correlating program* is to *interleave* instructions of both programs' source code in a way such that "corresponding" instructions of both programs would appear next to each other while maintaining the meaning of both programs. Doing so enables further static analysis of the correlating program with existing—if tweaked for this case—techniques.

For instance, a very simplified *correlating program* of the two C example programs defined earlier could be the following program, in which variables prefixed by "T_" refer to the right program while those not prefixed by "T_" refer to the left program:

Correlating program

```
sum = 0;                  T_sum = 0;
x = -x;                   T_x = -T_x;
y = 0;                    T_count = 0;
guard = (sum < x);        T_guard = (T_sum < T_x);
while (guard || T_guard) {
  if (guard) y = y + 1;
  if (T_guard) T_count = T_count + 1;
```

```
  if (guard) sum = sum + 1 + 2 * y;
  if (T_guard) T_sum = T_sum + 1 + 2 * T_count;

  if (guard) guard = (sum < x);
  if (T_guard) T_guard = (T_sum < T_x);
}
```

Notice how this correlating program closely interleaves instructions of both programs, factorizing left and right loops in the process, while allowing them to have different numbers of iterations thanks to the use of the *guard variables* `guard` and `T_guard`.

However, as we studied Yahav and Partush's article and the accompanying tool, we found issues with their generation algorithm for the *correlating program*: there were cases where the generated program would incorrectly simulate the two programs under consideration. This lead us to design an alternative algorithm for building that *correlating program*. As illustrated by the issues in the original paper[32], the kind of *program transformations* required to build a *correlating program* is particularly error prone. For this reason, we implemented our algorithm and proved it correct using the Coq proof assistant, thus ensuring that it would correctly simulate the two programs under consideration in all cases, while we also experimentally verified that it achieved an interleaving of the two programs which is compatible with the further static analysis techniques described in the original paper[32].

This notion of having mathematical proofs—mechanically verified by computer software such as the Coq proof assistant—is a cornerstone of this thesis: each of our contributions is indeed formalized in Coq and any stated theorem is also proved within this formal framework.

While formalizing and proving our alternative algorithm for building correlating programs, it became apparent that difference representations needed to be better *structured*: indeed, the original flawed algorithm worked at a very textual level, making it especially difficult to reason about. Instead, we re-interpreted textual differences as differences on the *abstract syntax trees* of programs, that is, a structured representation of their source code in the form of a tree. This more *structured* representation of differences made it possible to design our algorithm and prove it correct.

While a possible follow-up to this work would have been to move from a *static interleaving* of two programs to an analysis that would *dynamically* interleave them, in the manner Partush and Yahav did in the meantime[33], such an approach would still rely on *ad hoc* heuristics, and the classes of program differences it would accurately describe would be difficult to characterize. Instead, we decided to pursue the road of more *structured*, *expressive* and *semantic* representations of program changes, that would ideally allow for mechanical verification of any well-specified semantic change.

Towards formal descriptions of program changes

Our approach thus shifted from automatically inferring relational properties between programs to providing programmers with new, more semantic ways of describing their changes. Indeed, programmers already routinely describe their changes in the *commit message*, but those descriptions are informal and may not actually correspond to the performed modifications. Therefore, the main motivation behind the second part of this thesis is to provide programmers with a formal unambiguous *difference language*[5] to describe program changes. This language would be focused on the *semantics* of the change, and presented in a more structured way than textual changes.

Writing such a difference language requires a tremendous amount of language design, but first and foremost, it requires a proper formalism for change semantics: how can we associate meaning to a software *patch*? The majority of this thesis attempts to answer this question by proposing a formal framework to describe changes between two programs—written in arbitrary programming languages. This proposed framework draws inspiration from correlating programs[32] in that the change between two programs is represented by a third program—called *correlating oracle*—simulating the two others. However, unlike *correlating programs*, this third program directly embeds the expected semantic relation: from the ground up, it is built to exhibit—or verify—a relation known to the programmer, in contrast to building a *correlating program* without semantics insight and then trying to recover semantic information from it. In that regard, it is also very similar to the notion of *product programs* as presented by G. Barthe et al.[2, 3], which we will explain in details in Chapter 2.

At their heart, *correlating oracles* are simply programs simulating two other programs, in such a way that precise properties can be asserted on each related states. In the case of our previous example, if the left program executes in a series of computation steps $s_l^1 \longrightarrow s_l^2 \longrightarrow s_l^3 \longrightarrow s_l^4 \longrightarrow \ldots$ and the right program executes in a series of computation steps $s_r^1 \longrightarrow s_r^2 \longrightarrow s_r^3 \longrightarrow s_r^4 \longrightarrow \ldots$, then a useful *correlating oracle* would simulate those two programs in lockstep, asserting that each pair of synchronized states feature equivalent states modulo renaming, that is, that the programs behave exactly the same way, but refer to different variable names:

$$
\begin{array}{ccccccccc}
s_l^1 & \longrightarrow & s_l^2 & \longrightarrow & s_l^3 & \longrightarrow & s_l^4 & & \cdots \\
\vdots & & \vdots & & \vdots & & \vdots & & \\
o_1 & \longrightarrow & o_2 & \longrightarrow & o_3 & \longrightarrow & o_4 & & \cdots \\
\vdots & & \vdots & & \vdots & & \vdots & & \\
s_r^1 & \longrightarrow & s_r^2 & \longrightarrow & s_r^3 & \longrightarrow & s_r^4 & & \cdots
\end{array}
$$

Using our framework, we have defined specialized *oracle languages* in which *correlating oracles* corresponding to some common program changes can be written. Those common program changes include *renaming variables*, *swapping independent*

[5] We avoid the term "semantic patches" so as to not create confusion with Coccinelle[30, 29], which takes a different approach, as described in Chapter 8

assignments, and *fixing off-by-one errors causing crashes*, amongst others, and will be detailed in Chapter 4.

Those oracle languages can then be put together in a *difference language*, making use of the aforementioned framework to give formal semantics to change descriptions written by programmers. For instance, the oracle informally described above can be written "**rename** $[y \leftrightarrow count]$".

A more elaborate example of a difference that can be expressed in such a *difference language* is the difference between a *crashing* and a *fixed* program such as the following pair of programs:

Left (original) version

```
1 s = 0;
2 d = x;
3 while (0 <= d) {
4   if (x % d == 0)
5     s = s + d;
6   d = d - 1;
7 }
```

Right (modified) version

```
1 s = 0;
2 d = y;
3 while (0 < d) {
4   if (y % d == 0)
5     s = s + d;
6   d = d - 1;
7 }
```

In the example above, the right program is obtained by renaming `x` to `y` in the left program, and changing the condition `0 <= d` of line 3 to `0 < d`, hence avoiding a division by zero error when evaluating `x % d == 0` with `d` yields 0. This change can be described in our *difference language* along the lines of "**rename** $[x \leftrightarrow y]$; **fix off-by-one at line 3**". The actual formal difference description is slightly more complex, as we will see in Chapter 6.

As with the previously-highlighted contribution on correlating programs, this formal framework, along with several examples on a toy imperative programming language, have been formalized in the Coq proof assistant. The expressiveness of this framework has also been proven by encoding several *Relational Hoare Logic*[7, 3] variants presented in the aforementioned paper[3].

Contributions and plan

The main contributions presented in this thesis can be summarized as follows:

1. An example exhibiting that Eran Yahav and Nimrod Partush's[32] *correlating program* generation algorithm is unsound in the presence of `goto` statements or early loop exit statements such as `break` or `goto`.
2. An alternative, formally-proven algorithm addressing the aforementioned issue by handling the difference between two programs as a *structured* difference between their abstract syntax trees.
3. A novel formal framework for describing differences in program semantics, formalized as a Coq library with mechanically-checked theorems. In this framework, a difference between two programs is represented by a third program

called a *correlating oracle*. The small-step execution of that third program describes the relation between the execution traces of the two programs under consideration by pairing some of their intermediate states, asserting a formal *invariant* on each such pair. The framework guarantees that this *invariant* holds for every such pair if it holds on the initial state of the programs. Compared to previous work such as product programs, this approach enables reasoning about non-terminating and crashing programs.

4. A collection of fully-formalized *oracle languages* making use of this framework to describe a dozen different classes of differences on a toy imperative language. Each such *oracle language* allows writing concise *correlating oracles* specific to a given class of semantic differences. These oracle languages cover the following set of common changes: renaming variables, changing some values not affecting the programs' control flow, swapping two conditional branches by negating the corresponding condition, refactoring conditionals containing common code, handling reparenthesizing of sequences, commutation of independent assignments, avoiding crashes with defensive conditionals, fixing off-by-one errors inducing crashes, or replacing a sub-program with an equivalent one.
5. An independent proof of some *Relational Hoare Logic* variants, by encoding them within the aforementioned framework, giving small-step semantics to *Relational Hoare Logic* proofs.
6. A toy *difference language* on an idealized imperative programming language, giving a syntax for the previously-defined oracle languages.

The contents of this thesis are presented as follows:

1. Chapter 1 presents our first two contributions: it exhibits the soundness issue found in Partush and Yahav's paper[32] and proposes an alternative, formally-proven algorithm addressing it.
2. Chapter 2 reviews the notions of *product programs* and *Relational Hoare Logic* as presented by Barthe et al.[3]. While this chapter contains no new contribution, it lays out definitions that are necessary to the remaining of this thesis while presenting existing work that is a major inspiration for the work described in the following chapters.
3. Chapter 3 presents a novel formal framework for program differences.
4. Chapter 4 defines multiple *correlating oracle languages* on a small imperative language, exhibiting various capabilities of our framework.
5. Chapter 5 provides new proofs of soundness of several *Relational Hoare Logic* variants presented in chapter 2 by encoding them in our framework.
6. Chapter 6 presents a toy *difference language* leveraging from the previously-defined *correlating oracle languages*.
7. Chapter 7 presents the Coq library implementing chapters 3 to 6.
8. The conclusion presents the limitations of our approach and discusses alternatives as well as possible future work.

Chapter 1 is essentially an extended version of an article[13] we published previously, while Chapters 3 to 7 are an updated and much more extensive presentation of a work we already presented in another article[14].

Chapter 1

Correlating Programs

In this chapter, which is essentially an extended version of an article[13] we published previously, we present existing work by Eran Yahav and Nimrod Partush[32], demonstrate why it is flawed, and set out to fix it by proposing an alternative algorithm—formally proved using the Coq proof assistant.

The aforementioned work describes a tool to characterize differences between two programs by performing automatic static analysis on textual differences without requiring additional input from the programmer. This approach can be decomposed in two main components: the first one is a set of program transformations—which are implemented in a tool called `ccc`—used to build a *correlating program*; the second, and main component is a static analysis using abstract interpretation[11] on this generated *correlating program*—implemented in a tool called `dizy`.

In this chapter, we focus on the main component, that is the algorithm for generating *correlating programs*. A *correlating program* is a program tightly interleaving the instructions of two other programs in such a way that it correctly simulates some parallel execution of those two programs. The particular way the instructions are interleaved follows an heuristic aimed at increasing the likelihood of factorizing the two program's execution paths, keeping supposedly related instructions close to increase the precision of further analysis. This interleaving effectively defines a static scheduling of the two programs' parallel execution.

Yahav and Partush's original correlating program generation algorithm works in two steps: the first step is to transform each of the two programs to be compared to an equivalent *guarded form* in which every code block is broken into atomic statements, each of which is prefixed by a *guard condition* encoding the code path leading to the code block it was in. The second step is to actually interleave those guarded instructions in a common correlating program. The order in which those instructions are interleaved follows a textual diff on a "cleaned-up" version of the guarded forms, as we will explain more precisely.

To illustrate those steps, let us consider the following pair of C programs which differ by a moved line:

Left (original) version

```
void f(int a, int b) {
  if (a < b) {
    a = a + 10; // moved line
    b = b - 10;
  } else {
    b = b + 1;
  }
}
```

Right (modified) version

```
void f(int a, int b) {
  if (a < b) {
    b = b - 10;
  } else {
    a = a + 10; // moved line
    b = b + 1;
  }
}
```

First, both of those programs are transformed to semantically-equivalent guarded forms where every imperative command is guarded by a set of *guard variables* encoding the control flow:

Left version (guarded)

```
void f(int a, int b) {
  Guard G0 = (a < b);
  if (G0) a = a + 10; // moved line
  if (G0) b = b - 10;
  if (!G0) b = b + 1;
}
```

Right version (guarded)

```
void f(int a, int b) {
  Guard G0 = (a < b);
  if (G0) b = b - 10;
  if (!G0) a = a + 10; // moved line
  if (!G0) b = b + 1;
}
```

Then, a vector of "imperative commands" is extracted from each program in order to perform the *diff* operation that will drive the interleaving of the two guarded forms above. Those "imperative commands" are the actual variable assignments and `goto` statements, without any reference to the *guard variables*. In the case of this example, the left and right imperative commands are the following:

Left imperative commands

```
a = a + 10;
b = b - 10;
b = b + 1;
```

Right imperative commands

```
b = b - 10;
a = a + 10;
b = b + 1;
```

A *diff* is then performed on those two vectors of "imperative commands" to determine the order in which to insert each guarded form's instructions into the correlating program. The *diff* consists in annotating lines of both "imperative command" vectors with "-" for lines present in the first program but not the second, "+" for lines present in the second but not the first, and "=" for lines present in both:

```
- a = a + 10; // Original version
= b = b - 10; // Original version
= b = b - 10; // Modified version
+ a = a + 10; // Modified version
= b = b + 1;  // Original version
= b = b + 1;  // Modified version
```

This diff is used to determine how to interleave instructions from both guarded programs, and tagging instructions from the second program to avoid collisions, leading to the final *correlating program*:

Correlating program
```
1  void f(int a, int b) {
2    int T_a = a; int T_b = b;
3    Guard G0 = (a < b);        // if condition (Original version)
4    Guard T_G0 = (T_a < T_b);  // if condition (Modified version)
5    if (G0) a = a + 10;        // then branch  (Original version)
6    if (G0) b = b - 10;        // then branch  (Original version)
7    if (T_G0) T_b = T_b - 10;  // then branch  (Modified version)
8    if (!T_G0) T_a = T_a + 10; // else branch  (Modified version)
9    if (!G0) b = b + 1;        // else branch  (Original version)
10   if (!T_G0) T_b = T_b + 1;  // else branch  (Modified version)
11 }
```

1.1 Unsound handling of `goto` statements

In their original paper, Yahav and Partush claim to generate *correlating programs* for any pair of C programs, using the text-based algorithm presented above, as implemented in the tool "`ccc`". Unfortunately, we quickly found a bug in the way "`goto`" statements were handled, which made some generated correlating programs unsound, failing to simulate one or both of the programs under consideration. Due to how their tool translates loops into "`goto`" statements, this issue also affects loops constructions with early exit.

Example 1. Indeed, consider the following pair of programs (P_1, P_2), which differ by the inclusion of a "`break`" statement in P_2 (on the right), causing the loop to be iterated at most once in P_2:

Left program
```
void fail(int x) {
  i = 0;
  while (i <= 1) {
    i = i + 1;
    x = x + 1;

  }
}
```

Right program
```
void fail(int x) {
  i = 0;
  while (i <= 1) {
    i = i + 1;
    x = x + 1;
    break; // added break statement
  }
}
```

Given those two programs, `ccc` produces the following correlating program, in which `Guard` is a type for boolean values encoding the control flow:

Correlating program

```
void fail(int x) {
 int T_x = x;
 int i = 0; int T_i = 0;

L1: T_L1:;
 Guard G0 = (i <= 1);
 Guard T_G0 = (T_i <= 1);

 if (G0) i = i + 1;
 if (T_G0) T_i = T_i + 1;
 if (G0) x = x + 1;
 if (T_G0) T_x = T_x + 1;
 if (T_G0) goto T_L3;

 if (G0) goto L1;
 if (T_G0) goto T_L1;
L3: T_L3:
}
```

As one can see, `ccc` translates `while`-loops by defining *guard variables* holding the result of evaluating the loops' condition (lines 6 and 7), guarding the loops' body and inserting `goto` statements (lines 15 and 16) to recover the looping behavior. This translation enables close interleaving of the two programs' instructions within the loop.

Early exit constructs such as `break` are translated to additional `goto` statements (line 13). While this is fine in a single-program setting, this makes interleaving the two programs unsound. Indeed, this incorrect correlating program will return as soon as the `goto` statement on line 13 is executed, incorrectly skipping the instructions corresponding to the original program, which is supposed to run one additional iteration of the loop.

In our opinion, Partush and Yahav's approach is conceptually flawed because their generation algorithm is directed by a line-by-line *textual* difference between the two programs. This textual representation is not structured well enough to correctly perform a static scheduling of the two programs instructions. In addition, such program transformations can prove quite challenging to get right, and the insufficiently formal description of the algorithm makes reasoning about them significantly harder.

Indeed, such program transformations are disconcertingly easy to get wrong, with issues creeping in every forgotten corner case. This is why we believe such program transformations should be formally proved correct. That is what we did by writing an

alternative generating algorithm for correlating program and by *formally proving* it using the Coq proof assistant.

This alternative algorithm is directed by a *structured difference* between *abstract syntax trees* of the two programs rather than by a textual difference, which makes that algorithm significantly easier to formalize and prove correct.

1.2 Formal definition of the Imp_{br} language

Admittedly, going for a formally proven approach leads to some compromises: indeed, we have significantly restricted the scope of our generating algorithm to a small imperative language that roughly corresponds to a subset of the C programming language. This source language "Imp_{br}" is a minimalist imperative language commonly used in computer science lectures, featuring assignments, conditionals and "while" loops, extended with early exit constructs "`break`" and "`continue`".

1.2.1 Syntax of Imp_{br}

We write x for a variable identifier taken in an enumerable set of identifiers $\mathcal{I}$, n for an integer value taken in $\mathbb{Z}$, and b for a boolean value in **true**, **false**. Expressions are simple arithmetic expressions, and commands feature the standard assignments, **while**-loops and **if**-statemnts, as well as the non-local control-flow operators **break** and **continue**.

$$
\begin{aligned}
c &::= a \mid c;c \mid \textbf{while}\ (b)\ c \mid \textbf{if}\ (b)\ c\ \textbf{else}\ c && \text{(Commands)}\\
a &::= \textbf{skip} \mid x = e \mid \textbf{break} \mid \textbf{continue} && \text{(Atomic commands)}\\
b &::= \textbf{true} \mid \textbf{false} \mid b \,\&\&\, b \mid !b \mid e \le e && \text{(Boolean Expressions)}\\
e &::= x \mid \mathsf{n} \mid e + e && \text{(Arithmetic Expressions)}
\end{aligned}
$$

1.2.2 Semantics of Imp_{br}

The Imp_{br} language enjoys a standard big-step semantics: a program transforms the store by means of commands and commands make use of pure expressions to perform arithmetic and boolean computations.

A store $S : \mathcal{I} \to \mathbb{Z}$ is a partial map from variable identifiers to integer values. The empty store is written $\emptyset, \forall\, x \in dom(S)$ quantifies over the finite domain of the store S, and $S[x \mapsto \mathsf{n}]$ is the store defined on $dom(S) \cup \{x\}$ such that $S[x \mapsto \mathsf{n}](y) = \mathsf{n}$ if $x = y$ and $S(y)$ otherwise.

The judgment "$S \vdash e \Downarrow \mathsf{n}$" reads "In the store S, the arithmetic expression e evaluates into the integer n" and the judgment "$S \vdash b \Downarrow \mathsf{b}$" reads "In the store S, the boolean expression b evaluates into the boolean b".

1.2.2.1 Expressions semantics

Cst
$$\frac{}{S \vdash \mathsf{n} \Downarrow \mathsf{n}}$$

Var
$$\frac{}{S \vdash x \Downarrow S(x)}$$

Sum
$$\frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \qquad S \vdash e_2 \Downarrow \mathsf{n}_2}{S \vdash e_1 + e_2 \Downarrow \mathsf{n}_1 +_{\mathbb{Z}} \mathsf{n}_2}$$

1.2.2.2 Boolean expressions semantics

Cst
$$\frac{}{S \vdash \mathsf{b} \Downarrow \mathsf{b}}$$

Not
$$\frac{S \vdash b \Downarrow \mathsf{b}}{S \vdash !b \Downarrow \neg \mathsf{b}}$$

And
$$\frac{S \vdash b_1 \Downarrow \mathsf{b}_1 \qquad S \vdash b_2 \Downarrow \mathsf{b}_2}{S \vdash b_1 \,\&\&\, b_2 \Downarrow \mathsf{b}_1 \wedge \mathsf{b}_2}$$

LessOrEqual
$$\frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \qquad S \vdash e_2 \Downarrow \mathsf{n}_2}{S \vdash e_1 \leq e_2 \Downarrow \mathsf{n}_1 \leq_{\mathbb{Z}} \mathsf{n}_2}$$

The interpretation of a command yields a *return mode* which is either *normal* (written $\Box$), *interrupted* (written $\star$, used to handle **break** statements) or *continuing* (written $\circ$, used to handle **continue** statements). The judgment "$S_0 \vdash c \Downarrow_m S_1$" reads "The command c transforms the store S_0 into S_1 in mode m".

1.2.2.3 Commands semantics

Skip
$$\frac{}{S \vdash \textbf{skip} \Downarrow_{\Box} S}$$

Assign
$$\frac{S \vdash e \Downarrow \mathsf{n}}{S \vdash x = e \Downarrow_{\Box} S[x \mapsto \mathsf{n}]}$$

Seq
$$\frac{S \vdash c_1 \Downarrow_{\Box} S' \qquad S' \vdash c_2 \Downarrow_m S''}{S \vdash c_1; c_2 \Downarrow_m S''}$$

Seq Interrupted
$$\frac{S \vdash c_1 \Downarrow_m S' \qquad m \neq \Box}{S \vdash c_1; c_2 \Downarrow_m S'}$$

Then
$$\frac{S \vdash b \Downarrow \textbf{true} \qquad S \vdash c_1 \Downarrow_m S'}{S \vdash \textbf{if}\,(b)\; c_1 \;\textbf{else}\; c_2 \Downarrow_m S'}$$

Else
$$\frac{S \vdash b \Downarrow \textbf{false} \qquad S \vdash c_2 \Downarrow_m S'}{S \vdash \textbf{if}\,(b)\; c_1 \;\textbf{else}\; c_2 \Downarrow_m S'}$$

While False
$$\frac{S \vdash b \Downarrow \textbf{false}}{S \vdash \textbf{while}\,(b)\; c \Downarrow_{\Box} S}$$

While True
$$\frac{S \vdash b \Downarrow \textbf{true} \qquad S \vdash c \Downarrow_m S' \qquad S' \vdash \textbf{while}\,(b)\; c \Downarrow_{\Box} S'' \qquad m \neq \star}{S \vdash \textbf{while}\,(b)\; c \Downarrow_{\Box} S''}$$

While Break
$$\frac{S \vdash b \Downarrow \textbf{true} \qquad S \vdash c \Downarrow_{\star} S'}{S \vdash \textbf{while}\,(b)\; c \Downarrow_{\Box} S'}$$

Break
$$\frac{}{S \vdash \textbf{break} \Downarrow_{\star} S}$$

Continue
$$\frac{}{S \vdash \textbf{continue} \Downarrow_{\circ} S}$$

1.3 Formal definition of the Guarded Language

As mentioned earlier, Yahav and Partush's approach to generating correlating programs is to first transform each program into a corresponding *guarded form*, then interleaving the instructions of both *guarded programs* into a single *correlating program*. As we loosely follow Yahav and Partush's approach to keep compatibility with their further static analysis step, we also have to guard statements accordingly. Therefore, we have decided to formally define a guarded language derived from our input language, thus syntactically enforcing the use of a *guarded form*.

Every condition of the Guarded Language is stored into a boolean variable called a *guard variable* and every atomic instruction is guarded by a conjunction of guard variables (called a *guard* thereafter). This specific form effectively abstracts execution paths into guard variables, as the values of the guard variables uniquely determine a single block in the control flow graph of the input program. Thus, assigning specific values to these guard variables activates specific instructions of the input program. In Section 1.6, this mechanism will be at the heart of the static interleaving of programs instructions.

1.3.1 Syntax

In order to simplify the formal proof of some technical lemmas, guard variable identifiers are taken in the set $\mathcal{G}$ of words over the alphabet $\{0, 1\}$. We will later make use of the fact that a word in this alphabet can encode a path in an Imp abstract syntax tree. We write g for such guard identifiers.

The syntax of the guarded language includes assignment statements ac_G guarded by a conjunction of (positive or negative) guard variables and a **while**-loop statement guarded by a disjunction of guard conjunctions. Notice that **break** and **continue** are *not* present in the guarded language: indeed, as we will see later, such statements can be encoded using additional guard variables.

$$
\begin{array}{rcll}
c_G & ::= & c_G; c_G \mid \textbf{skip} \mid \textbf{while}\ (\vec{g_\vee})\ c_G \mid \textbf{while}\ (b)\ c \mid \textbf{if}\ (\vec{g_\wedge})\ ac_G & \text{(Commands)} \\
ac_G & ::= & x = e \mid g = b & \text{(Atomic commands)} \\
\vec{g_\vee} & ::= & \bot \mid \vec{g_\wedge} \vee \vec{g_\vee} & \text{(Guard disjunctions)} \\
\vec{g_\wedge} & ::= & \top \mid g_l \wedge \vec{g_\wedge} & \text{(Guard conjunctions)} \\
g_l & ::= & g \mid \neg g & \text{(Guard literals)} \\
g & \in & \{0, 1\}^* & \text{(Guard variables)}
\end{array}
$$

1.3.2 Semantics

Besides the store S of integer variables, a program written in the guarded language also transforms a guard store G of guard variables, which is a partial map $G : \mathcal{G} \to \mathbf{b}$ from guard identifiers to boolean values. The operations over standard stores are naturally transported to guard stores.

The judgment "$S_0, G_0 \vdash c_G \Downarrow S_1, G_1$" reads "The guarded command c_G transforms the store S_0 and the guard store G_0 into a new store S_1 and a new guard store G_1". The rules for the evaluation of guards and disjunctions of guards are straightforward and thus omitted.

Skip
$$\frac{}{S, G \vdash \textbf{skip} \Downarrow S, G}$$

Seq
$$\frac{S, G \vdash c_{G1} \Downarrow S', G' \qquad S', G' \vdash c_{G2} \Downarrow S'', G''}{S, G \vdash c_{G1}; c_{G2} \Downarrow S'', G''}$$

Ignore
$$\frac{G \vdash \vec{g_\wedge} \Downarrow \textbf{false}}{S, G \vdash \textsf{if}\ (\vec{g_\wedge})\ ac_G \Downarrow S, G}$$

Activate
$$\frac{G \vdash \vec{g_\wedge} \Downarrow \textbf{true} \qquad S, G \vdash ac_G \Downarrow S', G'}{S, G \vdash \textsf{if}\ (\vec{g_\wedge})\ ac_G \Downarrow S', G'}$$

GAssignment
$$\frac{S \vdash b \Downarrow \textsf{b}}{S, G \vdash g = b \Downarrow S, G[g \mapsto \textsf{b}]}$$

Assignment
$$\frac{S \vdash e \Downarrow \textsf{n}}{S, G \vdash x = e \Downarrow S[x \mapsto \textsf{n}], G}$$

While False
$$\frac{G \vdash \vec{g_\vee} \Downarrow \textbf{false}}{S, G \vdash \textbf{while}\ (\vec{g_\vee})\ c_G \Downarrow S, G}$$

While True
$$\frac{G \vdash \vec{g_\vee} \Downarrow \textbf{true} \qquad S, G \vdash c_G \Downarrow S', G' \qquad S', G' \vdash \textbf{while}\ (\vec{g_\vee})\ c_G \Downarrow S'', G''}{S, G \vdash \textbf{while}\ (\vec{g_\vee})\ c_G \Downarrow S'', G''}$$

1.4 Translation to guarded form

While our algorithm, unlike Yahav and Partush's, is not split in two distinct steps and directly builds a correlating program from a *structured difference* as we will see later, the proof of correctness relies on the correct transformation of individual programs into equivalent guarded forms. We define the translation into guarded forms by using a recursive translation function CI defined as follows:

c	o	$CI(\vec{g_\wedge},\ \pi,\ c,\ o)$
skip	o	**skip**
$x = e$	o	$\textsf{if}\ (\vec{g_\wedge})\ x = e$
$c_1; c_2$	o	$CI(\vec{g_\wedge},\ 0 \cdot \pi,\ c_1,\ o); CI(\vec{g_\wedge},\ 1 \cdot \pi,\ c_2,\ o)$
if $(b)\ c_1$ **else** c_2	o	$\textsf{if}\ (\vec{g_\wedge})\ \pi = b;$ $CI(\vec{g_\wedge} \wedge \pi,\ 1 \cdot \pi,\ c_1,\ o);$ $CI(\vec{g_\wedge} \wedge \neg\pi,\ 0 \cdot \pi,\ c_2,\ o)$
while $(b)\ c$	o	$\textsf{if}\ (\vec{g_\wedge})\ \pi = b;$ **while** $(\vec{g_\wedge} \wedge \pi)\ \{$ $\textsf{if}\ (\vec{g_\wedge} \wedge \pi)\ 1 \cdot \pi =$ **true**; $CI(\vec{g_\wedge} \wedge \pi \wedge (1 \cdot \pi),\ 1 \cdot 1 \cdot \pi,\ c,\ \textsf{Some}\ \pi);$ $\textsf{if}\ (\vec{g_\wedge} \wedge \pi)\ \pi = b$ $\}$
break	$\textsf{Some}\ \pi'$	$\textsf{if}\ (\vec{g_\wedge})\ \pi' =$ **false**
continue	$\textsf{Some}\ \pi'$	$\textsf{if}\ (\vec{g_\wedge})\ 1 \cdot \pi' =$ **false**

$CI(\vec{g_\wedge}, \pi, c, o)$ is a guarded program simulating a sub-program c at path π in a larger program ; $\vec{g_\wedge}$ is the guard conjunction guarding c in this larger program, and o is a *loop marker* used to indicate the innermost (if any) loop of the larger program under which c is executed. This loop marker equals "**Some** π'" if the innermost loop containing "c" is rooted at path π' and "**None**" if there is no such loop. o is used in the translation of **break** and **continue** statements by keeping track of the guard variables π' and $1 \cdot \pi'$ controlling the execution of the innermost loop.

The path π is used to name fresh guard variables, linking every guard variable back to the position of the corresponding condition in the original syntax tree, ensuring freshness, and enabling reasoning about sub-programs in isolation thanks to the notion of guard identifier independence defined below:

Definition 1 (Guard identifier independence). A guard g is independent for a path π, written $g\#\pi$, if g does not end with π.

Let us quickly go over each case of the translation function CI:

- As a **skip** has no effect, it does not matter what it is guarded by. Therefore, for simplicity's sake, the translation of a **skip** is simply a **skip**, no matter what the other parameters are.
- An assignment $x = e$ is an effectful statement that has to be guarded appropriately. In this case, the appropriate guards are simply those accumulated in the $\vec{g_\wedge}$ parameter.
- The case of a sequence $c_1; c_2$ is purely recursive. Since a sequence does not introduce any condition, the same guard list $\vec{g_\wedge}$ is passed for both recursive calls. The reasoning is the same for the *loop marker*: a sequence does not change the loop structure, each of the two subcommands are in the same inner loop (if any) as the sequence $c_1; c_2$, thus o is passed as-is. However, we have to keep track of the path of each subcommand in the larger program, which is why CI is called with $0 \cdot \pi$ in one case and $1 \cdot \pi$ in the other.
- In the case where c is "**if** (b) c_1 **else** c_2", we first create a fresh guard name "π" to which we conditionally assign the value of b under the guard conjunction $\vec{g_\wedge}$: "**if** $(\vec{g_\wedge})$ $\pi = b$". This guard represents the condition used to select either the "then" or "else" branch of the **if** statement. We then recursively call CI on c_1 (the "then" branch), guarded by the conjunction of the previous guards and the newly created one ($\vec{g_\wedge} \wedge \pi$). We do the same for c_2 (the "else" branch), negating the guard π ($\vec{g_\wedge} \wedge \neg\pi$). As for the sequence, we keep track of the path of each subcommand with $0 \cdot \pi$ and $1 \cdot \pi$.
- The case of a loop **while** (b) c is the most involved: it is recursive because it has a subcommand under a new path, and it introduces new guards and a new inner loop for this subcommand. Just as in the case of **if** statements, a new guard π is created and conditionally set to the conditional expression b by the guarded assignment "**if** $(\vec{g_\wedge})$ $\pi = b$". This new guard π represents the loop condition and will guard the whole translated loop. However, yet another new guard $1 \cdot \pi$ is

created and set to **true** (“**if** $(\vec{g_\wedge} \wedge \pi)\ 1 \cdot \pi = \textbf{true}$”) to guard the translated body of the loop, while an additional assignment is added at the end of the loop’s body to re-evaluate the loop condition. The purpose of the $1 \cdot \pi$ guard is to encode **continue** statements.

- A **break** statement can only occur inside a loop, and its effect is to break out of the innermost one. Since this subcommand is inside a loop, the loop marker is set to **Some** π' for some π', which is, by construction, a guard variable guarding the whole loop. Therefore, setting this guard variable to **false** will effectively disabling further execution of the loop’s instructions.
- A **continue** statement is handled in a way similar to **break**: the loop marker **Some** π' lets us set the guard variable $1 \cdot \pi'$ to **false**, thus disabling the execution of the loop’s body for this iteration, at the end of which the guard condition will be re-evaluated.

The soundness of CI is stated by the following technical lemma. Roughly speaking, this lemma states that $CI(\vec{g_\wedge},\ \pi,\ c,\ o)$ simulates c correctly under a guard store G if the following assumptions are met:

- The guard conjunction $\vec{g_\wedge}$ is active ($G \vdash \vec{g_\wedge} \Downarrow \textbf{true}$).
- If the execution mode yielded by the execution of c is not $\square$, then c can only be contained in a wider program involving a loop, for which o contains the guard variable.
- If $o = \pi'$ for some π', then both π' and $0 \cdot \pi'$ appear in $\vec{g_\wedge}$.
- Every guard variable of $\vec{g_\wedge}$ is independent with regards to π so that all guard variables created by CI will effectively be fresh.

Lemma 1 (CI is sound on active guards)**.** *For all stores S and S', a guard store G, return mode m, guard conjunction $\vec{g_\wedge}$, command c, path π and optional loop marker o, if the following statements hold:*

i. $S \vdash c \Downarrow_m S'$
ii. $G \vdash \vec{g_\wedge} \Downarrow \textbf{true}$
iii. $m \neq \square \implies \exists\, \pi_l,\ o = \textsf{Some}\ \pi_l$
iv. $\forall\, \pi_l,\ o = \textsf{Some}\ \pi_l \implies G(\pi_l) = G(1 \cdot \pi_l) = \textbf{true}$
v. $\forall\, g,\ g \# \pi$

then, there exists a store G' such that the following statements hold:

a. $S, G \vdash CI(\vec{g_\wedge},\ \pi,\ c,\ o) \Downarrow S', G'$
b. $\forall\, g \in dom(G),\ g \# \pi \implies G(g) = G'(g)$
c. $m = \star \implies \exists \pi_l,\ o = \textsf{Some}\ \pi_l \wedge G'(\pi_l) = \textbf{false}$
d. $m = \circ \implies \exists \pi_l,\ o = \textsf{Some}\ \pi_l \wedge G'(1 \cdot \pi_l) = \textbf{false}$

Proof. The proof is done by structural induction on $S \vdash c \Downarrow_m S'$. □

1.5 Structured program differences

As noted earlier, our approach does not directly work on textual differences, but rather on structured syntactic differences—which can be provided by the developer or computed automatically.

Such a syntactic difference between the abstract syntax trees of two syntactically correct programs is denoted by a special representation of the whole programs. This representation will be processed in a purely recursive way by the algorithm that generates the correlating program.

1.5.1 Syntax

The structured difference language is derived from the input language, in such a way that each internal node of the abstract syntax tree is associated with a local mutation Δ (where $\pm ::= - \mid +$):

$$\begin{aligned}\Delta ::= {} & a \to a' \mid \pm[c];\Delta \mid \pm\Delta;[c] \mid \Delta;\Delta \\ & \mid \textbf{if}\ (b \to b')\ \Delta\ \textbf{else}\ \Delta \mid \textbf{while}\ (b \to b')\ \Delta \\ & \mid \pm[\textbf{if}\ (b)\ c\ \textbf{else}]\ \Delta \mid \pm[\textbf{if}\ (b)]\ \Delta\ [\textbf{else}\ c] \mid \pm[\textbf{while}\ (b)]\ \Delta\end{aligned}$$

A structured difference represents the full original program along with differences leading to the modified program.

Informally, the notation "$a \to a'$" means that the leaf command a of the original program is replaced by the leaf command a' in the modified program.

Likewise, the notation "$\pm[c];\Delta$" means that the command c is removed from the orginal program ("$-[c];\Delta$") or inserted into the modified program ("$+[c];\Delta$") while the right side is kept with a local mutation Δ. The notation "$\pm\Delta;[c]$" is symmetrical. The notation "$\Delta;\Delta'$" means that the both programs are sequences, with the left side of the sequence described by the local mutation Δ while the right side is described by Δ'.

Similarly, "$+[\textbf{if}\ (b)\ c\ \textbf{else}]\ \Delta$" means that an **if**" statement is inserted in the modified program with the command c as its **then** branch and using existing code (with local mutation Δ) as its **else** branch.

The other constructs follow the same pattern, and a more formal meaning is given by the projection functions $\Pi_0()$ and $\Pi_1()$ which return the original and modified embedded programs respectively:

Δ	$\Pi_0(\Delta)$	$\Pi_1(\Delta)$
$-[c_1]; \Delta_2$	$c_1; \Pi_0(\Delta_2)$	$\Pi_1(\Delta_2)$
$-\Delta_1; [c_2]$	$\Pi_0(\Delta_1); c_2$	$\Pi_1(\Delta_1)$
$+[c_1]; \Delta_2$	$\Pi_0(\Delta_2)$	$c_1; \Pi_1(\Delta_2)$
$+\Delta_1; [c_2]$	$\Pi_0(\Delta_1)$	$\Pi_1(\Delta_1); c_2$
$\Delta_1; \Delta_2$	$\Pi_0(\Delta_1); \Pi_0(\Delta_2)$	$\Pi_1(\Delta_1); \Pi_1(\Delta_2)$
$\textbf{if}\ (b \to b')\ \Delta_1\ \textbf{else}\ \Delta_2$	$\textbf{if}\ (b)\ \Pi_0(\Delta_1)\ \textbf{else}\ \Pi_1(\Delta_2)$	$\textbf{if}\ (b')\ \Pi_0(\Delta_2)\ \textbf{else}\ \Pi_1(\Delta_2)$
$\textbf{while}\ (b \to b')\ \Delta'$	$\textbf{while}\ (b)\ \Pi_0(\Delta')$	$\textbf{while}\ (b')\ \Pi_1(\Delta')$
$+[\textbf{if}\ (b)\ c_1\ \textbf{else}]\ \Delta_2$	$\Pi_0(\Delta_2)$	$\textbf{if}\ (b)\ c_1\ \textbf{else}\ \Pi_1(\Delta_2)$
$+[\textbf{if}\ (b)]\ \Delta_1\ [\textbf{else}\ c_2]$	$\Pi_0(\Delta_1)$	$\textbf{if}\ (b)\ \Pi_1(\Delta_1)\ \textbf{else}\ c_2$
$-[\textbf{if}\ (b)\ c_1\ \textbf{else}]\ \Delta_2$	$\textbf{if}\ (b)\ c_1\ \textbf{else}\ \Pi_0(\Delta_2)$	$\Pi_1(\Delta_2)$
$-[\textbf{if}\ (b)]\ \Delta_1\ [\textbf{else}\ c_2]$	$\textbf{if}\ (b)\ \Pi_0(\Delta_1)\ \textbf{else}\ c_2$	$\Pi_1(\Delta_1)$
$+[\textbf{while}\ (b)]\ \Delta'$	$\Pi_0(\Delta')$	$\textbf{while}\ (b)\ \Pi_1(\Delta')$
$-[\textbf{while}\ (b)]\ \Delta'$	$\textbf{while}\ (b)\ \Pi_0(\Delta')$	$\Pi_1(\Delta')$
$a \to a'$	a	a'

As stated by the following theorem, a difference between any two programs can always be found.

Theorem 1 (Completeness of the diff. language)**.** *For all pairs of programs (c, c'), there exists a difference Δ such that $\Pi_0(\Delta) = c$ and $\Pi_1(\Delta) = c'$.*

Proof. As we are only interested in the existence of a difference between every pair of programs, and not in any measure of conciseness or quality of that difference, we define a recursive total function $\underset{\text{naive}}{\Delta}$ for computing differences. Informally, that function builds a diff that "removes" every construct of the left program and relies on a second recursive total function $\underset{\text{naive}}{\Delta^+}$ to "add" the second program:

$$
\begin{aligned}
\underset{\text{naive}}{\Delta^+}(a, a') &= a \to a' \\
\underset{\text{naive}}{\Delta^+}(a, c_1; c_2) &= +\underset{\text{naive}}{\Delta^+}(a, c_1); [c_2] \\
\underset{\text{naive}}{\Delta^+}(a, \textbf{while}\ (b)\ c) &= +[\textbf{while}\ (b)]\ \underset{\text{naive}}{\Delta^+}(a, c) \\
\underset{\text{naive}}{\Delta^+}(a, \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2) &= +[\textbf{if}\ (b)\ c_1\ \textbf{else}]\ \underset{\text{naive}}{\Delta^+}(a, c_1)
\end{aligned}
$$

$$
\begin{aligned}
\underset{\text{naive}}{\Delta}(a, c) &= \underset{\text{naive}}{\Delta^+}(a, c) \\
\underset{\text{naive}}{\Delta}(c_1; c_2, c') &= -\underset{\text{naive}}{\Delta}(c_1, c'); [c_2] \\
\underset{\text{naive}}{\Delta}(\textbf{while}\ (b)\ c, c') &= -[\textbf{while}\ (b)]\ \underset{\text{naive}}{\Delta}(c, c') \\
\underset{\text{naive}}{\Delta}(\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2, c') &= -[\textbf{if}\ (b)]\ \underset{\text{naive}}{\Delta}(c_1, c')\ [\textbf{else}\ c_2]
\end{aligned}
$$

That function is proved to produce correct differences, by induction on the structure of the left program. The base case of a leaf command as the left program is proved correct by induction on the right program. ◻

1.6 Generation algorithm directed by structured differences

As instructions from both programs will be interleaved in a single program, variable identifiers of both programs have to be differentiated. To keep things simple and compatible with the "dizy" abstract interpertation tool, we adopt the following convention: variable identifiers of the original (left) program start with "$O_$", while variable identifiers of the modified (right) program start with "$T_O_$".

Definition 2 (Difference tagging). Difference tagging, denoted $(\Delta)^{T,T'}$, is defined as follows:

$$
\begin{array}{rcl}
(-[c_1];\Delta_2)^{T,T'} & = & -[T(c_1)];(\Delta_2)^{T,T'} \\
(-\Delta_1;[c_2])^{T,T'} & = & -(\Delta_1)^{T,T'};[T(c_2)] \\
(+[c_1];\Delta_2)^{T,T'} & = & +[T'(c_1)];(\Delta_2)^{T,T'} \\
(+\Delta_1;[c_2])^{T,T'} & = & +(\Delta_1)^{T,T'};[T'(c_2)] \\
(\Delta_1;\Delta_2)^{T,T'} & = & (\Delta_1)^{T,T'};(\Delta_2)^{T,T'} \\
(\textbf{if}\,(b \rightarrow b')\ \Delta_1\ \textbf{else}\ \Delta_2)^{T,T'} & = & \textbf{if}\,(T(b) \rightarrow T'(b'))\ (\Delta_1)^{T,T'}\ \textbf{else}\ (\Delta_2)^{T,T'} \\
(\textbf{while}\,(b \rightarrow b')\ \Delta')^{T,T'} & = & \textbf{while}\,(T(b) \rightarrow T'(b'))\ (\Delta')^{T,T'} \\
(+[\textbf{if}\,(b)\ c_1\ \textbf{else}]\ \Delta_2)^{T,T'} & = & +[\textbf{if}\,(T'(b))\ T'(c_1)\ \textbf{else}]\ (\Delta_2)^{T,T'} \\
(+[\textbf{if}\,(b)]\ \Delta_1\ [\textbf{else}\ c_2])^{T,T'} & = & +[\textbf{if}\,(T'(b))]\ (\Delta_1)^{T,T'}\ [\textbf{else}\ T'(c_2)] \\
(-[\textbf{if}\,(b)\ c_1\ \textbf{else}]\ \Delta_2)^{T,T'} & = & -[\textbf{if}\,(T(b))\ T(c_1)\ \textbf{else}]\ (\Delta_2)^{T,T'} \\
(-[\textbf{if}\,(b)]\ \Delta_1\ [\textbf{else}\ c_2])^{T,T'} & = & -[\textbf{if}\,(T(b))]\ (\Delta_1)^{T,T'}\ [\textbf{else}\ T(c_2)] \\
(+[\textbf{while}\,(b)]\ \Delta')^{T,T'} & = & +[\textbf{while}\,(T'(b))]\ (\Delta')^{T,T'} \\
(-[\textbf{while}\,(b)]\ \Delta')^{T,T'} & = & -[\textbf{while}\,(T(b))]\ (\Delta')^{T,T'} \\
(a \rightarrow a')^{T,T'} & = & T(a) \rightarrow T'(a')
\end{array}
$$

We define a correlating program generation algorithm directed by structured differences as a recursive function CP, which works much like CI, but handling both left and right programs at the same time by following structured differences. The program $CP(\Delta,\ \pi_0,\ \pi_1,\ \vec{g_{\wedge 0}},\ \vec{g_{\wedge 1}},\ o_0,\ o_1)$ is a correlating program of $\Pi_0(\Delta)$ and $\Pi_1(\Delta)$, corresponding to an interleaving of the guarded forms $CI(\vec{g_{\wedge 0}},\ \pi_0,\ \Pi_0(\Delta),\ o_0)$ and $CI(\vec{g_{\wedge 1}},\ \pi_1,\ \Pi_1(\Delta),\ o_1)$. The paths π_0 and π_1 are the position of $\Pi_0(\Delta)$ and $\Pi_1(\Delta)$ in the larger correlating program. Just as with CI, they are used as fresh names for new guard variables, which also eases Coq proofs by encoding path information in the name of the guards. $\vec{g_{\wedge 0}}$ and $\vec{g_{\wedge 1}}$ are the guard conjunctions guarding $\Pi_0(\Delta)$ and $\Pi_1(\Delta)$ in the whole correlating program, while o_0 and o_1 represent the innermost loop (if any) under which $\Pi_0(\Delta)$ and $\Pi_1(\Delta)$ are executed, and are used to translate **break** and **continue** statements.

One key reason for using structured syntactic differences is that they are well-suited for factoring the control structures of both programs. Indeed, under the assumption that the control structure of the modified program is kept unchanged, then it is possible to represent the difference as a structured syntactic difference having the same structure itself. By being directed by such a structured syntactic difference, the correlating program generation algorithm can closely interleave instructions that are likely to be executed under the same conditions.

Δ	$CP(\Delta, \pi_0, \pi_1, \vec{g}_{\wedge 0}, \vec{g}_{\wedge 1}, o_0, o_1)$
$-[c]; \Delta$	$CI(\vec{g}_{\wedge 0}, 0 \cdot \pi_0, c, o_0)$; $CP(\Delta, 1 \cdot \pi_0, \pi_1, \vec{g}_{\wedge 0}, \vec{g}_{\wedge 1}, o_0, o_1)$
$-\Delta; [c]$	$CP(\Delta, 0 \cdot \pi_0, \pi_1, \vec{g}_{\wedge 0}, \vec{g}_{\wedge 1}, o_0, o_1)$; $CI(\vec{g}_{\wedge 0}, 1 \cdot \pi_0, c, o_0)$
$+[c]; \Delta$	$CI(\vec{g}_{\wedge 1}, 0 \cdot \pi_1, c, o_1)$; $CP(\Delta, \pi_0, 1 \cdot \pi_1, \vec{g}_{\wedge 0}, \vec{g}_{\wedge 1}, o_0, o_1)$
$+\Delta; [c]$	$CP(\Delta, \pi_0, 0 \cdot \pi_1, \vec{g}_{\wedge 0}, \vec{g}_{\wedge 1}, o_0, o_1)$; $CI(\vec{g}_{\wedge 1}, 1 \cdot \pi_1, c, o_1)$
$\Delta_0; \Delta_1$	$CP(\Delta_0, 0 \cdot \pi_0, 0 \cdot \pi_1, \vec{g}_{\wedge 0}, \vec{g}_{\wedge 1}, o_0, o_1)$ $CP(\Delta_1, 1 \cdot \pi_0, 1 \cdot \pi_1, \vec{g}_{\wedge 0}, \vec{g}_{\wedge 1}, o_0, o_1)$
if $(b_0 \to b_1)$ Δ_0 **else** Δ_1	**if** $(\vec{g}_{\wedge 0})\ \pi_0 = b_0$; **if** $(\vec{g}_{\wedge 1})\ \pi_1 = b_1$; $CP(\Delta_0, 1 \cdot \pi_0, 1 \cdot \pi_1, \vec{g}_{\wedge 0} \wedge \pi_0, \vec{g}_{\wedge 1} \wedge \pi_1, o_0, o_1)$; $CP(\Delta_1, 0 \cdot \pi_0, 0 \cdot \pi_1, \vec{g}_{\wedge 0} \wedge \neg\pi_0, \vec{g}_{\wedge 1} \wedge \neg\pi_1, o_0, o_1)$

while $(b_0 \to b_1)$ Δ	**if** $(\vec{g}_{\wedge 0})\ \pi_0 = b_0$;
	if $(\vec{g}_{\wedge 1})\ \pi_1 = b_1$;
	while $((\vec{g}_{\wedge 0} \wedge \pi_0) \vee (\vec{g}_{\wedge 1} \wedge \pi_1))$ {
	 if $(\vec{g}_{\wedge 0} \wedge \pi_0)\ 1 \cdot \pi_0 =$ **true**;
	 if $(\vec{g}_{\wedge 1} \wedge \pi_1)\ 1 \cdot \pi_1 =$ **true**;
	 $CP(\Delta, 1 \cdot 1 \cdot \pi_0, 1 \cdot 1 \cdot \pi_1, \vec{g}_{\wedge 0} \wedge \pi_0 \wedge (1 \cdot \pi_0),$
	    $\vec{g}_{\wedge 1} \wedge \pi_1 \wedge (1 \cdot \pi_1),$ **Some** π_0, **Some** $\pi_1)$;
	 if $(\vec{g}_{\wedge 0} \wedge \pi_0)\ \pi_0 = b_0$;
	 if $(\vec{g}_{\wedge 1} \wedge \pi_1)\ \pi_1 = b_1$;
	}

$c_{G0} \to c_{G1}$	$CI(\vec{g}_{\wedge 0}, \pi_0, c_{G0}, o_0)$; $CI(\vec{g}_{\wedge 1}, \pi_1, c_{G1}, o_1)$
$-$[**if** (b) c **else**] Δ	**if** $(\vec{g}_{\wedge 0})\ \pi_0 = b$; $CI(\vec{g}_{\wedge 0} \wedge \pi_0, 1 \cdot \pi_0, c, o_0)$; $CP(\Delta, 0 \cdot \pi_0, \pi_1, \vec{g}_{\wedge 0} \wedge \neg\pi_0, \vec{g}_{\wedge 1}, o_0, o_1)$
$+$[**if** (b) c **else**] Δ	**if** $(\vec{g}_{\wedge 1})\ \pi_1 = b$; $CI(\vec{g}_{\wedge 1} \wedge \pi_1, 1 \cdot \pi_1, c, o_1)$; $CP(\Delta, \pi_0, 0 \cdot \pi_1, \vec{g}_{\wedge 0}, \vec{g}_{\wedge 1} \wedge \neg\pi_1, o_0, o_1)$
$-$[**if** (b)] Δ [**else** c]	**if** $(\vec{g}_{\wedge 0})\ \pi_0 = b$; $CP(\Delta, 1 \cdot \pi_0, \pi_1, \vec{g}_{\wedge 0} \wedge \pi_0, \vec{g}_{\wedge 1}, o_0, o_1)$; $CI(\vec{g}_{\wedge 0} \wedge \neg\pi_0, 0 \cdot \pi_0, c, o_0)$
$+$[**if** (b)] Δ [**else** c]	**if** $(\vec{g}_{\wedge 1})\ \pi_1 = b$; $CP(\Delta, \pi_0, 1 \cdot \pi_1, \vec{g}_{\wedge 0}, \vec{g}_{\wedge 1} \wedge \pi_1, o_0, o_1)$; $CI(\vec{g}_{\wedge 1} \wedge \neg\pi_1, 0 \cdot \pi_1, c, o_1)$

Figure 1.1: Difference-directed correlating program generation function CP

Δ	$CP(\Delta,\, \pi_0,\, \pi_1,\, \vec{g_\wedge}_0,\, \vec{g_\wedge}_1,\, o_0,\, o_1)$

$-$[**while** (b)] Δ	**if** $(\vec{g_\wedge}_0)\ \pi_0 = b$;
	if $(\vec{g_\wedge}_0 \wedge \pi_0)\ 1 \cdot \pi_0 = \textbf{true}$;
	$CP(\Delta,\, 1 \cdot 1 \cdot \pi_0,\, \pi_1,\, \vec{g_\wedge}_0 \wedge \pi_0 \wedge (1 \cdot \pi_0),\, \vec{g_\wedge}_1,\, \textsf{Some}\ \pi_0,\, o_1)$;
	if $(\vec{g_\wedge}_0 \wedge \pi_0)\ \pi_0 = b$;
	while $(\vec{g_\wedge}_0 \wedge \pi_0)$ {
	if $(\vec{g_\wedge}_0 \wedge \pi_0)\ 1 \cdot \pi_0 = \textbf{true}$;
	$CI(\vec{g_\wedge}_0 \wedge \pi_0 \wedge (1 \cdot \pi_0),\, 1 \cdot 1 \cdot \pi_0,\, \Pi_0(\Delta),\, \textsf{Some}\ \pi_0)$;
	if $(\vec{g_\wedge}_0 \wedge \pi_0)\ \pi_0 = b$;
	}

$+$[**while** (b)] Δ	**if** $(\vec{g_\wedge}_1)\ \pi_1 = b$;
	if $(\vec{g_\wedge}_1 \wedge \pi_1)\ 1 \cdot \pi_1 = \textbf{true}$;
	$CP(\Delta,\, \pi_0,\, 1 \cdot 1 \cdot \pi_1,\, \vec{g_\wedge}_0,\, \vec{g_\wedge}_1 \wedge \pi_1 \wedge (1 \cdot \pi_1),\, o_0,\, \textsf{Some}\ \pi_1)$;
	if $(\vec{g_\wedge}_1 \wedge \pi_1)\ \pi_1 = b$;
	while $(\vec{g_\wedge}_1 \wedge \pi_1)$ {
	if $(\vec{g_\wedge}_1 \wedge \pi_1)\ 1 \cdot \pi_1 = \textbf{true}$;
	$CI(\vec{g_\wedge}_1 \wedge \pi_1 \wedge (1 \cdot \pi_1),\, 1 \cdot 1 \cdot \pi_1,\, \Pi_1(\Delta),\, \textsf{Some}\ \pi_1)$;
	if $(\vec{g_\wedge}_1 \wedge \pi_1)\ \pi_1 = b$;
	}

Figure 1.1: Difference-directed correlating program generation function CP (cont.)

Overall, the definition of CP (Figure 1.1) is very similar to that of CI—and in fact often *calls* CI—but has to handle many more cases due to the nature of structured differences.

For example, consider the definition for $-$[**if** (b) c **else**] Δ corresponding to the removal of an **if** statement and its "then" part while keeping its "else" part. We first create a new guard π_0 to which we conditionally assign the value b under the conjunction $\vec{g_\wedge}_0$ of the original program ("**if** $(\vec{g_\wedge}_0)\ \pi_0 = b$") because the **if** statement is only executed in the first program. We then call CI to output the guarded form of the statement c under the removed "then" part: "$CI(\vec{g_\wedge}_0 \wedge \pi_0,\ 1 \cdot \pi_0,\ c,\ o_0)$". This will be conditionally executed under the conjunction of $\vec{g_\wedge}_0$ and the new guard π_0, under a new unique path "$1 \cdot \pi_0$". We then continue the translation of the remaining structured difference Δ by recursively calling CP on it: "$CP(\Delta,\, 0 \cdot \pi_0,\, \pi_1,\, \vec{g_\wedge}_0 \wedge \neg\pi_0,\, \vec{g_\wedge}_1,\, o_0,\, o_1)$". For the original program, we create a new unique path "$0 \cdot \pi_0$". This part is guarded by "$\vec{g_\wedge}_0 \wedge \neg\pi_0$" as it is executed under the "else" part of the original program. For the modified program, we keep the guard $\vec{g_\wedge}_1$ and the path π_1 which is still unique. o_0 and o_1 are reused unmodified as we are not translating a loop.

Example 2. The result of calling CP on the pair of programs from Example 1 is the following correlating program (printed as C code):

Correlating program

```
void fail(int O_x) {
 int T_O_x = O_x;
 int O_i = 0; int T_O_i = 0;

 Guard G1 = 1; Guard T_G1 = 1;
 Guard G0 = (O_i <= 1);
 Guard T_G0 = (T_O_i <= 1);
 while (G0 || T_G0) {
  if (G0) G1 = 1;                    // unused here (otherwise
  if (T_G0) T_G1 = 1;                // used to encode “continue”)
  if (G0) if (G1) O_i = O_i + 1;
  if (T_G0) if (T_G1) T_O_i = T_O_i + 1;
  if (G0) if (G1) O_x = O_x + 1;
  if (T_G0) if (T_G1) T_O_x = T_O_x + 1;
  if (T_G0) if (T_G1) T_G0 = 0;     // encodes “break”
  if (G0) G0 = (O_i <= 1);          // update left loop's condition
  if (T_G0) T_G0 = (T_O_i <= 1);    // update right loop's condition
 }
}
```

Even though correlating programs include instructions from two different programs, manipulating two sets of variables, they execute in a single store. Thanks to tagging, those variables are independent, and the store of a correlating program can be *split* into two stores corresponding to each of the two programs under comparison. The same can be said about guard stores.

Definition 3 (Store splitting)**.** Two stores S_0 and S_1 are said to split a store S if $\forall n \in \{0,1\}$, $\forall x \in dom(S_n)$, $S(x) = S_n(x)$; and S_0 contains only variables starting with "$O_$", and S_1 with "$T_O_$".

Definition 4 (Guard store splitting)**.** Two guard stores G_0 and G_1 are said to split a guard store G if $\forall n \in \{0,1\}$, $\forall x \in dom(G_n)$, $G(x) = G_n(x)$; and G_0 contains only variables ending with 0, and G_1 with "$1\cdot$".

Lemma 2 (CP is sound under context)**.** *For all differences Δ, stores S_0, S'_0, S_1, S'_1, and S, guard stores G_0, G'_0, G_1, G'_1, and G, paths π_0 and π_1, guard conjunctions $\vec{g_\wedge}_0$ and $\vec{g_\wedge}_1$, and optional loop markers o_0 and o_1, if the following statements hold:*

i. $\forall n \in \{0,1\},\ S_n, G_n \vdash CI(\vec{g_\wedge}_n,\ \pi_n,\ \Pi_n(\Delta),\ o_n) \Downarrow S'_n, G'_n$
ii. *S_0 and S_1 split S*
iii. *G_0 and G_1 split G*

iv. *Variable identifiers appearing in* $\Pi_0(\Delta)$ *start with "O_", and those appearing in* $\Pi_1(\Delta)$ *start with "T_O_"*

v. $\vec{g_{\wedge 0}}$ *contains only variables ending with* 0 *and* $\vec{g_{\wedge 1}}$ *contains only variables ending with* 1

then there exists a store S' *and a guard store* G' *such that:*

a. $S, G \vdash CP(\Delta,\ \pi_0,\ \pi_1,\ \vec{g_{\wedge 0}},\ \vec{g_{\wedge 1}},\ o_0,\ o_1) \Downarrow S', G'$

b. S'_0 *and* S'_1 *split* S'

c. G'_0 *and* G'_0 *split* G'

Proof. The proof is primarily done by induction on the structure of the difference Δ. Most of the resulting cases are tedious but rather straightforward, as they consist in proving that leaf commands from either program execute without changing any of the variables or guards corresponding to the other program. It may be noted that 'break' and 'continue' statements are no exception and do not pose any particular difficulty, as such considerations are taken care of when proving CI.

However, there are some more complicated cases: indeed, the induction on the structure of Δ is insufficient for reasoning about loops. Instead, when Δ describes a pair of **while** loops, we need to reason by induction on their big-step execution. This turns out to be somewhat tricky as we cannot reason about a single loop but on a particular interleaving of the two loops. This is done by introducing a new judgment `two_loops` with rules corresponding to the interleaved execution of two loops. This judgment is proved to be equivalent to describing the separate execution of the two loops, so that it can be used to replace hypothesis (i) and be used for induction. □

While the above key lemma mentions the invariants used in the induction, CP is typically used with fixed initial values for most of its arguments, hence the function $correlated_program(\Delta) = CP(\Delta,\ 0,\ 1,\ \textbf{true},\ \textbf{true},\ \texttt{None},\ \texttt{None})$ and the associated theorem:

Theorem 2 (correlating_program is sound). *For all stores* S_0, S_1, S'_0, *and* S'_1, *guard store* G *and difference* Δ, *if the following statements hold:*

i. $S_0 \vdash \Pi_0((\Delta)^{T,T'}) \Downarrow_{\square} S'_0$

ii. $S_1 \vdash \Pi_1((\Delta)^{T,T'}) \Downarrow_{\square} S'_1$

then there exists a store S' *and a guard store* G' *such that:*

a. $S_0 \uplus S_1, G \vdash correlated_program((\Delta)^{T,T'}) \Downarrow S', G'$

b. S'_0 *and* S'_1 *split* S'

Proof. As for the other proofs, the details can be found in the Coq development. The proof can however be outlined as follows:

By lemma 1, we have that $CI(\textbf{true},\ 0,\ \Pi_0((\Delta)^{T,T'}),\ \texttt{None})$ simulates $\Pi_0((\Delta)^{T,T'})$ and $CI(\textbf{true},\ 1,\ \Pi_1((\Delta)^{T,T'}),\ \texttt{None})$ simulates $\Pi_1((\Delta)^{T,T'})$.

By lemma 2, we have that $correlated_program((\Delta)^{T,T'})$ simulates both $CI(\textbf{true}, 0, \Pi_0((\Delta)^{T,T'}), \texttt{None})$ and $CI(\textbf{true}, 1, \Pi_1((\Delta)^{T,T'}), \texttt{None})$ simulates $\Pi_1((\Delta)^{T,T'})$, thus simulating $\Pi_0((\Delta)^{T,T'})$ and $\Pi_1((\Delta)^{T,T'})$. □

1.7 Implementation and experiments

As stated earlier, the algorithm described above has been proved within the Coq proof assistant. This amounts to about $3,800$ lines of Coq, of which 10% are definitions of the Imp_{br} and guarded languages, as well of the definition of what a correct correlating program is. The remaining lines are used for the algorithm itself and its soundness proof.

This Coq code is then extracted to OCaml. In addition to this extracted code, we also wrote about $1,000$ lines of OCaml to parse the input language, construct a syntactic difference and print the correlating program in C syntax. One should notice that the semantics of the generated C program does not exactly match the semantics of the formalized development: for instance, our input language manipulates mathematical integers while the generated C program uses fixed-length machine integers. Albeit it was not done because outside the core of our work, we do not consider it would be conceptually difficult to integrate into our input language the semantics of the generated C code (e.g. 32 bits integers), as the generation algorithm manipulates the syntax of expressions abstractly. Moreover, the semantics of our language is compatible with the semantics of input C language expected by "`dizy`". It should also be noted that the language presented in this paper has been slightly simplified for readability, while the actual tool and its formalization in Coq handle additional operators as well as a limited form of arrays.

To compute the structural syntactic differences, we aim at finding a minimal difference by an exploration of the space of mappings between abstract syntax trees, starting from the root nodes of the abstract syntax trees of the two programs. We recursively descend along those trees, comparing at each level all possible differences and keeping the minimal one (using an heuristics that tries to minimize insertion or deletion of loops). We use some memoization to implement a weak form of dynamic programming. While this computation of the syntactic difference is not proven correct nor optimal in Coq, a mechanically verified checker dynamically ensures that the projections of the chosen structural difference are indeed the two input programs.

To compare the quality of our correlating programs with the ones produced by "`ccc`", a series of 23 examples (most between 10 and 20 lines long, with the exception of one around 140 lines long) were analyzed by "`dizy`". While doing so, we found no instance where the correlating program produced by "`ccc`" enabled a more precise analysis than that permitted by the correlating program generated by `correlating_program`. On the contrary, we found several examples where `correlating_program` outperformed "`ccc`" (examples 6, 7 and 23). One of the examples is a binary search algorithm in a sorted array, with the modified version introducing early loop exits. We can generate the correlating program and analyze it with "`dizy`" (by slightly modifying it to correctly

handle read access to arrays). We also attempted to test more complex examples but were limited by `dizy`'s capabilities (e.g., no handling of C's bit-wise logical operations). All those tests are available online at https://www.irif.fr/thib/atva15/examples/ and can be reproduced. In practice, the computation of structural differences and the generation of the correlating program was almost instantaneous on our examples. Most of the computation time is spent in `dizy`.

Chapter 2

Product programs

In this chapter—which only presents prior work and contains no new contribution—we review *product programs*, an existing notion that relates both to the previous notion of *correlating programs* and to the remaining of our work.

The *correlating programs* presented in the previous chapter are programs built by soundly interleaving the instructions of pairs of programs in order to be processed for further static analysis. This interleaving process is driven by heuristics which aim at factorizing the execution paths of the two programs with the hope that better factorized code paths will lead to better abstract analysis results. In that sense, programs are interleaved based on their syntactic proximity. In a nutshell, correlating programs are a generic way to present a pair of programs as a single program for a subsequent semantic analysis.

Much like correlating programs, *product programs*, as presented by Barthe et al.[3], are built by soundly interleaving the instructions of given pairs of programs. But in contrast with correlating programs, the very way they are built follows some semantics insight: indeed, each product program is constructed in order to prove a given specific semantic property, and every assumption made when factorizing code paths from both programs—for instance, that both programs always execute the same branch of a conditional—is made explicit in the product program by the addition of an **assert** statement. In that sense, unlike correlating programs, product programs are inherently semantic.

Again, we want to stress that this chapter contains no significant original contribution: it is merely a reframing of the aforementioned paper[3] featuring a few minor fixes, with the intent to bridge the work presented in the previous chapter with the work that will be presented next. This chapter also serves to present some definitions necessary to the subsequent chapters. These definitions include the formal definition of the simple imperative language Imp which will be used throughout the remaining of this manuscript, as well as that of several program logics, and of product programs.

2.1 Imp

The formal logic systems and product program rules that we will present in this chapter are tailored for the idealized imperative language Imp, which is not to be confused with Imp_{br} presented earlier in Section 1.2. Indeed, while there are other reasons—which will be addressed in more details in the next chapter—for this change, the Imp language is also closer to the core imperative language used by Barthe et al.[3], easing comparison.

2.1.1 Syntax of Imp

The syntax of Imp is pretty standard, being composed of arithmetic expressions, boolean expressions, assignments, **while**-loops, conditionals and **assert** statements:

Definition 5 (Imp syntax).

$$\boxed{\text{Statements}}$$

$$\begin{array}{rcl} c & ::= & \textbf{skip} \mid x = e \mid c; c \mid \textbf{if}\,(b)\; c\,\textbf{else}\; c \\ & & \mid \textbf{while}\;(b)\;c \mid \textbf{assert}\;(b) \end{array}$$

$$\boxed{\text{Arithmetic expressions}}$$

$$\begin{array}{rcl} e & ::= & \mathsf{n} \mid x \mid e \oplus e \\ \oplus & ::= & + \mid - \mid * \mid / \mid \% \end{array}$$

$$\boxed{\text{Boolean expressions}}$$

$$\begin{array}{rcl} b & ::= & \textbf{true} \mid \textbf{false} \\ & & \mid !b \mid b \,\&\&\, b \mid b \mid\mid b \mid b == b \\ & & \mid e == e \mid e < e \mid e \leq e \end{array}$$

2.1.2 Semantics of Imp

As with Imp_{br}, the semantics of arithmetic and boolean expressions are defined on stores, that is, partial maps from variable identifiers to integer values.

The judgment "$S \vdash e \Downarrow \mathsf{n}$" reads "In store S, the arithmetic expression e evaluates into the integer n", and the judgment "$S \vdash b \Downarrow \mathsf{b}$" reads "In the store S, the boolean expression b evaluates into the boolean b".

Definition 6 (Imp expression semantics)**.** The following rules define the operational semantics for Imp expressions:

$$\frac{}{S \vdash \mathsf{n} \Downarrow \mathsf{n}}\ \text{Constant} \qquad \frac{}{S \vdash x \Downarrow S(x)}\ \text{Variable} \qquad \frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \quad S \vdash e_2 \Downarrow \mathsf{n}_2}{S \vdash e_1 + e_2 \Downarrow \mathsf{n}_1 +_{\mathbb{Z}} \mathsf{n}_2}\ \text{Sum}$$

$$\frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \quad S \vdash e_2 \Downarrow \mathsf{n}_2}{S \vdash e_1 - e_2 \Downarrow \mathsf{n}_1 -_{\mathbb{Z}} \mathsf{n}_2}\ \text{Subtraction} \qquad \frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \quad S \vdash e_2 \Downarrow \mathsf{n}_2}{S \vdash e_1 * e_2 \Downarrow \mathsf{n}_1 \times_{\mathbb{Z}} \mathsf{n}_2}\ \text{Multiplication}$$

$$\frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \quad S \vdash e_2 \Downarrow \mathsf{n}_2 \quad \mathsf{n}_2 \neq 0}{S \vdash e_1 / e_2 \Downarrow \mathsf{n}_1 /_{\mathbb{Z}} \mathsf{n}_2}\ \text{Division} \qquad \frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \quad S \vdash e_2 \Downarrow \mathsf{n}_2}{S \vdash e_1 \% e_2 \Downarrow \mathsf{n}_1 \%_{\mathbb{Z}} \mathsf{n}_2}\ \text{Modulo}$$

Definition 7 (Imp boolean expression semantics)**.** The following rules define the operational semantics for Imp boolean expressions:

$$\frac{}{S \vdash \mathsf{b} \Downarrow \mathsf{b}}\ \text{Constant} \qquad \frac{S \vdash b \Downarrow \mathsf{b}}{S \vdash !b \Downarrow \neg \mathsf{b}}\ \text{Not}$$

$$\frac{S \vdash b_1 \Downarrow \mathsf{b}_1 \quad S \vdash b_2 \Downarrow \mathsf{b}_2}{S \vdash b_1 \mathbin{\&\&} b_2 \Downarrow \mathsf{b}_1 \wedge \mathsf{b}_2}\ \text{And} \qquad \frac{S \vdash b_1 \Downarrow \mathsf{b}_1 \quad S \vdash b_2 \Downarrow \mathsf{b}_2}{S \vdash b_1 \mid\mid b_2 \Downarrow \mathsf{b}_1 \vee \mathsf{b}_2}\ \text{Or}$$

$$\frac{S \vdash b_1 \Downarrow \mathsf{b}_1 \quad S \vdash b_2 \Downarrow \mathsf{b}_2}{S \vdash b_1 == b_2 \Downarrow \mathsf{b}_1 = \mathsf{b}_2}\ \text{BoolEqual} \qquad \frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \quad S \vdash e_2 \Downarrow \mathsf{n}_2}{S \vdash e_1 == e_2 \Downarrow \mathsf{n}_1 =_{\mathbb{Z}} \mathsf{n}_2}\ \text{Equal}$$

$$\frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \quad S \vdash e_2 \Downarrow \mathsf{n}_2}{S \vdash e_1 < e_2 \Downarrow \mathsf{n}_1 <_{\mathbb{Z}} \mathsf{n}_2}\ \text{LessThan} \qquad \frac{S \vdash e_1 \Downarrow \mathsf{n}_1 \quad S \vdash e_2 \Downarrow \mathsf{n}_2}{S \vdash e_1 \leq e_2 \Downarrow \mathsf{n}_1 \leq_{\mathbb{Z}} \mathsf{n}_2}\ \text{LessOrEqual}$$

The interpretation of statements transforms a store into another store: the judgment $S \vdash c \Downarrow S'$ states that the statement c transforms the store S into the store S'.

Definition 8 (Imp statements semantics)**.** The following rules define the operational semantics for Imp statements:

Skip
$$\frac{}{S \vdash \textbf{skip} \Downarrow S}$$

Assign
$$\frac{S \vdash e \Downarrow \mathsf{n}}{S \vdash x = e \Downarrow S[x \mapsto \mathsf{n}]}$$

Assert
$$\frac{S \vdash b \Downarrow \textbf{true}}{S \vdash \textbf{assert}\ (b) \Downarrow S}$$

Seq
$$\frac{S \vdash c_1 \Downarrow S' \qquad S' \vdash c_2 \Downarrow S''}{S \vdash c_1; c_2 \Downarrow S''}$$

If-Then
$$\frac{S \vdash b \Downarrow \textbf{true} \qquad S \vdash c_1 \Downarrow S'}{S \vdash \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \Downarrow S'}$$

If-Else
$$\frac{S \vdash b \Downarrow \textbf{false} \qquad S \vdash c_2 \Downarrow S'}{S \vdash \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \Downarrow S'}$$

While False
$$\frac{S \vdash b \Downarrow \textbf{false}}{S \vdash \textbf{while}\ (b)\ c \Downarrow S}$$

While True
$$\frac{S \vdash b \Downarrow \textbf{true} \qquad S \vdash c \Downarrow S' \qquad S' \vdash \textbf{while}\ (b)\ c \Downarrow S''}{S \vdash \textbf{while}\ (b)\ c \Downarrow S''}$$

For simplicity's sake, unlike the language defined by Barthe et al.[3], Imp does not feature arrays. Therefore we will omit the array update rules defined in the original article. This omission is not a fundamental change, as the array update rules are almost identical to the assignment rules.

2.2 Hoare Logic

Hoare Logic[16] is a well-known program logic used to prove that programs are correct with regards to a given specification. While it only considers the execution of a single program—as opposed to the execution of multiple programs—it is both an inspiration and a building block for the proof systems presented thereafter.

In Hoare Logic, the correct execution of a program is described by a Hoare Triple of the form $\{P\}\, c\, \{Q\}$ where P and Q are logical assertions over states. Informally, such a triple states that for every store S satisfying the precondition P, no error arises when executing c in S and the resulting store S'—if any—satisfies the postcondition Q.

Hoare Logic is made of a set of axioms and inference rules to prove Hoare Triples. A Hoare Triple is valid if there exist a derivation using the Hoare Logic rules defined below that reaches its conclusion.

Hoare Logic provides a set of rules to prove such triples valid:

$$\frac{}{\vdash \{P\}\,\textbf{skip}\,\{P\}}\ \text{Skip} \qquad \frac{P \Rightarrow b}{\vdash \{P\}\,\textbf{assert}\,(b)\,\{P\}}\ \text{Assert}$$

$$\frac{}{\vdash \{P[e/x]\}\, x = e\,\{P\}}\ \text{Assign} \qquad \frac{\vdash \{P\}\,c_1\,\{Q\} \qquad \vdash \{Q\}\,c_2\,\{R\}}{\vdash \{P\}\,c_1; c_2\,\{R\}}\ \text{Seq}$$

$$\frac{\vdash \{P \wedge b\}\,c_1\,\{Q\} \qquad \vdash \{P \wedge \neg b\}\,c_2\,\{Q\}}{\vdash \{P\}\,\textbf{if}\,(b)\ c_1\ \textbf{else}\ c_2\,\{Q\}}\ \text{If}$$

$$\frac{\vdash \{P \wedge b\}\,c\,\{P\}}{\vdash \{P\}\,\textbf{while}\,(b)\ c\,\{P \wedge \neg b\}}\ \text{While} \qquad \frac{\vdash \{P\}\,c\,\{Q\} \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\vdash \{P'\}\,c\,\{Q'\}}\ \text{Sub}$$

Note that a Hoare Triple can be valid even if the specified program does not actually terminate. Indeed, the `While` rule does not ensure termination, as it does not require the repeated execution of the loop's body c in a store satisfying P to eventually lead to a store in which b evaluates to **false**. This notion of correctness is called partial correctness: a valid Hoare Triple does not mean the program terminates on any input matching the precondition, but that whenever it does, the result satisfies the given postcondition. If termination is expected, it has to be proven separately.

In addition, Hoare Logic is incomplete, in the sense that not every true Hoare Triple is provable. This incompleteness stems from the incompleteness of the logic used for assertions: indeed, the `Sub` and `Assert` rules require determining the validity of certain logical assertions. However, Gödel's Incompleteness Theorem states that there exist no consistent proof system containing basic arithmetic in which every true assertion can be proven. Hoare Logic is, however, said to have *relative completeness* in the sense that supposing every true assertion has a proof, then every true Hoare Triple has a Hoare Logic proof.

In our Coq implementation, assertions are Coq propositions on stores, thus using Coq's own logic to prove those assertions.

2.3 Relational Hoare Logic

Relational Hoare Logic[7] (RHL) is a program logic for reasoning about pairs of programs. As the name suggests, it is an adaptation of Hoare Logic to pairs of programs and relational properties. Its judgments are of the form $\vdash \{P\}\,c_1 \sim c_2\,\{Q\}$ where c_1 and c_2 are the two statements under consideration, while the precondition P and the postcondition Q are assertions on pairs of program states. For simplicity's sake we assume that the program states have separate sets of variable identifiers, that is, variable identifiers occurring in one program are absent from the other. Intuitively, such a judgment is valid if all executions of c_1 and c_2 from a pair of states satisfying P either diverge or successfully result in a pair of states satisfying Q.

Example 3. For instance, consider the two following programs:

$$\textbf{if}\,(x < 4)\; x = x + 2\;\textbf{else skip} \tag{2.1}$$

$$\textbf{if}\,(y < 2)\; y = y + 1\;\textbf{else skip} \tag{2.2}$$

The following judgment relates those two programs, asserting that executing them on a pair of states satisfying the precondition $x = 2y$ results in a pair of states satisfying the postcondition $x = 2y$:

$\vdash \{x = 2y\}\,\textbf{if}\,(x < 4)\; x = x + 2\;\textbf{else skip} \;\sim\; \textbf{if}\,(y < 2)\; y = y + 1\;\textbf{else skip}\,\{x = 2y\}$

The remaining of this section presents increasingly expressive variants of Relational Hoare Logic.

2.3.1 Minimal Relational Hoare Logic

A first RHL variant is Minimal Relational Hoare Logic, which is akin to using traditional Hoare Logic by applying the same rules to both programs under consideration at the same time. Because of this, Minimal Relational Hoare Logic considers only pairs of programs with the same syntactic structure, requiring them to execute in lockstep, taking the same branches and performing the same number of loop iterations.

Definition 9 (Minimal RHL)**.** Minimal Relational Hoare Logic is defined by the following set of inference rules:

$$\frac{}{\vdash \{P\}\,\textbf{skip} \;\sim\; \textbf{skip}\,\{P\}}\ \text{R-Skip} \qquad \frac{P \Rightarrow b_1 \wedge b_2}{\vdash \{P\}\,\textbf{assert}\,(b_1) \;\sim\; \textbf{assert}\,(b_2)\,\{P\}}\ \text{R-Assert}$$

$$\frac{}{\vdash \{P[e_2/x_2][e_1/x_1]\}\, x_1 = e_1 \;\sim\; x_2 = e_2\,\{P\}}\ \text{R-Assign}$$

$$\frac{\vdash \{P\}\,c_1^1 \;\sim\; c_1^2\,\{Q\} \qquad \vdash \{Q\}\,c_2^1 \;\sim\; c_2^2\,\{R\}}{\vdash \{P\}\,c_1^1; c_2^1 \;\sim\; c_1^2; c_2^2\,\{R\}}\ \text{R-Seq}$$

$$\frac{\vdash \{P \wedge b_1\}\,c_1^1 \;\sim\; c_1^2\,\{Q\} \qquad \vdash \{P \wedge \neg b_1\}\,c_2^1 \;\sim\; c_2^2\,\{Q\} \qquad P \Rightarrow b_1 = b_2}{\vdash \{P\}\,\textbf{if}\,(b_1)\; c_1^1\;\textbf{else}\;c_2^1 \;\sim\; \textbf{if}\,(b_2)\; c_1^2\;\textbf{else}\;c_2^2\,\{Q\}}\ \text{R-If}$$

$$\frac{\vdash \{P \wedge b_1\}\,c_1 \;\sim\; c_2\,\{P\} \qquad P \Rightarrow b_1 = b_2}{\vdash \{P\}\,\textbf{while}\,(b_1)\; c_1 \;\sim\; \textbf{while}\,(b_2)\; c_2\,\{P \wedge \neg b_1\}}\ \text{R-While}$$

$$\frac{\vdash \{P\}\,c_1 \;\sim\; c_2\,\{Q\} \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\vdash \{P'\}\,c_1 \;\sim\; c_2\,\{Q'\}}\ \text{R-Sub}$$

Minimal Relational Hoare Logic is sound, that is, if $\vdash \{P\}\,c_1 \;\sim\; c_2\,\{Q\}$ is derivable using the above rules, then all non-diverging executions of c_1 and c_2 from pairs of states satisfying P successfully reach pairs of states satisfying Q. We will provide a novel, mechanically-checked proof of this assertion in Chapter 5.

2.3.2 Core Relational Hoare Logic

The requirement imposed by Minimal RHL that both programs must share the same syntactic structure and dynamic execution paths makes it unsuitable for relating all but the most similar pairs of programs.

For instance, Minimal RHL cannot be used to relate the program **skip**; $x = e$ to the program $x' = e'$. Indeed, none of the rules defined above applies to statements with different syntactic structures.

Thankfully, Minimal RHL can easily be extended to such pairs of programs featuring different syntactic structure by adding rules for reasoning separately about each program.

Definition 10 (Core RHL). Core Relational Hoare Logic is the logic defined by extending Minimal Relational Hoare Logic with the following rules for reasoning about left programs:

$$\frac{\vdash \{P\}\,\textbf{skip} \sim c\,\{Q\} \qquad P \Rightarrow b}{\vdash \{P\}\,\textbf{assert}\,(b) \sim c\,\{Q\}}\ \text{R-Assert-L}$$

$$\frac{\vdash \{P\}\,\textbf{skip} \sim c\,\{Q\}}{\vdash \{P[e/x]\}\,x = e \sim c\,\{Q\}}\ \text{R-Assign-L}$$

$$\frac{\vdash \{P \wedge b\}\,c_1 \sim c\,\{Q\} \qquad \vdash \{P \wedge \neg b\}\,c_2 \sim c\,\{Q\}}{\vdash \{P\}\,\textbf{if}\,(b)\ c_1\ \textbf{else}\ c_2 \sim c\,\{Q\}}\ \text{R-If-L}$$

$$\frac{\vdash \{P\}\,c_1 \sim \textbf{skip}\,\{Q\} \qquad \vdash \{Q\}\,c_2 \sim c\,\{R\}}{\vdash \{P\}\,c_1; c_2 \sim c\,\{R\}}\ \text{R-Seq-Skip-L}$$

And the following rules for reasoning about right programs:

$$\frac{\vdash \{P\}\,c \sim \textbf{skip}\,\{Q\} \qquad P \Rightarrow b}{\vdash \{P\}\,c \sim \textbf{assert}\,(b)\,\{Q\}}\ \text{R-Assert-R}$$

$$\frac{\vdash \{P\}\,c \sim \textbf{skip}\,\{Q\}}{\vdash \{P[e/x]\}\,c \sim x = e\,\{Q\}}\ \text{R-Assign-R}$$

$$\frac{\vdash \{P \wedge b\}\,c \sim c_1\,\{Q\} \qquad \vdash \{P \wedge \neg b\}\,c \sim c_2\,\{Q\}}{\vdash \{P\}\,c \sim \textbf{if}\,(b)\ c_1\ \textbf{else}\ c_2\,\{Q\}}\ \text{R-If-R}$$

$$\frac{\vdash \{P\}\,\textbf{skip} \sim c_1\,\{Q\} \qquad \vdash \{Q\}\,c \sim c_2\,\{R\}}{\vdash \{P\}\,c \sim c_1; c_2\,\{R\}}\ \text{R-Seq-Skip-R}$$

Readers familiar with the original article may notice that our R-Assign-L rule is slightly different from the version presented by Barthe et. al: indeed, when formalizing

Core RHL in Coq, we found that this rule had the preconditions swapped between the premises and conclusions. The R-Update-L rule from the original paper, which is omitted here, presents a similar error. In addition, we added the R-Seq-Skip-L rule, which we believe was improperly omitted in the original paper.

Example 4. To take our earlier example, with Core Relational Hoare Logic, comparing **skip**; c to c becomes possible:

$$\dfrac{\dfrac{}{\vdash \{P\}\,\textbf{skip} \sim \textbf{skip}\,\{P\}}\,\text{R-Skip} \qquad \dfrac{\vdots}{\vdash \{P\}\,c \sim c\,\{Q\}}}{\vdash \{P\}\,\textbf{skip}; c \sim c\,\{Q\}}\,\text{R-Seq-Skip-L}$$

As with Minimal Relational Hoare Logic, we will provide a new, mechanically-checked proof of soundness for Core Relational Hoare Logic in the next chapter.

2.3.3 Extended Relational Hoare Logic

The final variant presented in the original paper, Extended Relational Hoare Logic, is obtained by extending Core Relational Hoare Logic with the single rule `R-Unfold`:

$$\dfrac{\vdash \{P\}\,c_1' \sim c_2'\,\{Q\} \qquad \vdash [c_1] \succcurlyeq [c_1'] \qquad \vdash [c_2] \succcurlyeq [c_2']}{\vdash \{P\}\,c_1 \sim c_2\,\{Q\}}\,\text{R-Unfold}$$

where $\vdash [c] \succcurlyeq [c']$ holds if and only if c' crashes whenever c crashes and finite executions of c and c' lead to extensionally-equivalent results.

This rule allows replacing the two statements under consideration by two other statements whose non-crashing extensional behaviors are included in those of the two statements under consideration. This enables reasoning about pairs of programs with non-synchronized loops.

2.3.4 Self-composition

While the Relational Hoare Logic variants presented above are increasingly expressive, they are still incomplete, as they can only be used to relate pairs of programs with similar structures. Relative completeness can be recovered by adding the following *self-composition rule* to any of the Relational Hoare Logic variants previously defined:

$$\dfrac{\vdash \{P\}\,c_1; c_2\,\{Q\}}{\vdash \{P\}\,c_1 \sim c_2\,\{Q\}}\,\text{R-SelfComp}$$

This rule reduces the proof of a relational property to a proof in regular Hoare Logic, enjoying the relative completeness of the latter. Proofs in this style will however not exploit the possible similarity in the control structure of both programs and will therefore be much more involved.

2.4 Product programs

A product program, much like the correlated programs presented in the previous chapter, is a single program simulating the behaviors of two other programs. In contrast to correlated programs, however, product programs are built following syntactic rules based on semantics insights. Barthe et al.[3] present several set of rules to build such programs, each rule set corresponding to one of the RHL variants presented above. Informally, product programs interleave statements in the same “order” they are considered in RHL proofs, while materializing side conditions by inserting corresponding **assert** statements. This leads Hoare Logic proofs on a product program c' to follow the same structure as Relational Hoare Logic proofs on the corresponding pair of programs (c_1, c_2), such that the correction of a Hoare Triple $\{P\}\, c'\, \{Q\}$ implies that of the Relational Hoare Quadruple $\{P\}\, c_1 \sim c_2\, \{Q\}$. One benefit of doing so is that off-the-shelf non-relational verification tools can then be used on the product program.

By comparison with the previous approach of *correlating programs*, while *product programs* can also be used in a two-stage analysis—first building a product program and then performing non-relational program analysis on it—, the relation between control paths of both programs is statically assumed when building the product program, and those assumptions are made explicit by the addition of **assert** statements. Alternatively, product program generation and verification can be done simultaneously, the later driving the construction of the product program. This is explained in more details in the original paper, and not repeated here as it is out of the scope of this thesis.

2.4.1 Minimal product programs

The rules driving the construction of minimal product programs mirror that of Relational Hoare Logic, deferring the verification of side conditions to subsequent proofs by pushing them into **assert** statements, in such a way that a regular Hoare Logic proof on a Minimal product program will have roughly the same shape as a Minimal Relational Hoare Logic proof of the same property. For instance, whenever a Minimal RHL proof involves the rule `R-Skip`, the Hoare Logic proof on the corresponding minimal product program will use the Hoare Logic rule `Skip`. Likewise, as a pair of **assert** statements is translated to a sequence of **assert** statements, Minimal RHL proofs involving `R-Assert` will lead to Hoare Logic proofs involving the Hoare Logic rules `Seq` and `Assert`.

Valid minimal product programs are defined by the judgment $c_1 \times c_2 \longrightarrow c$ stating that c is a valid product program of c_1 and c_2. As with Minimal RHL, we assume that left and right programs refer to separate variable identifiers, so that instructions from one program do not change the other program’s behavior.

$$\frac{}{\textbf{skip} \times \textbf{skip} \longrightarrow \textbf{skip}}\ \text{P-Skip} \qquad \frac{}{x_1 = e_1 \times x_2 = e_2 \longrightarrow x_1 = e_1; x_2 = e_2}\ \text{P-Assign}$$

$$\frac{}{\textbf{assert}\ (b_1) \times \textbf{assert}\ (b_2) \longrightarrow \textbf{assert}\ (b_1); \textbf{assert}\ (b_2)}\ \text{P-Assert}$$

$$\frac{c_1^1 \times c_1^2 \longrightarrow c_1 \qquad c_2^1 \times c_2^2 \longrightarrow c_2}{c_1^1; c_2^1 \times c_1^2; c_2^2 \longrightarrow c_1; c_2}\ \text{P-Seq}$$

$$\frac{c_1^1 \times c_1^2 \longrightarrow c_1 \qquad c_2^1 \times c_2^2 \longrightarrow c_2}{\textbf{if}\ (b_1)\ c_1^1\ \textbf{else}\ c_2^1 \times \textbf{if}\ (b_2)\ c_1^2\ \textbf{else}\ c_2^2 \longrightarrow \textbf{assert}\ (b_1 == b_2); \textbf{if}\ (b_1)\ c_1\ \textbf{else}\ c_2}\ \text{P-If}$$

$$\frac{c_1 \times c_2 \longrightarrow c}{\textbf{while}\ (b_1)\ c_1 \times \textbf{while}\ (b_2)\ c_2 \longrightarrow \textbf{assert}\ (b_1 == b_2); \textbf{while}\ (b_1)\ c}\ \text{P-While}$$

Each of the rules defined above corresponds to a Minimal RHL rule. For instance, `P-Skip` corresponds to `R-Skip` and relates a pair of **skip** statements by replacing them with a single **skip**. A sequence of **skip** could have been used instead, but it would have had the same extensional semantics. Likewise, `P-Assign` corresponds to `R-Assign` and transforms a pair of assignments into a sequence of those same two assignments, while `P-Assert` corresponds to `R-Assert` and transforms a pair of assertions into a sequence.

`P-Seq` is purely recursive and corresponds to `R-Seq`.

`P-If`, which corresponds to `R-If` is more interesting: instead of requiring a proof that the two programs' conditions are equivalent, this condition is made explicit by inserting a corresponding assertion. This allows deferring the proof to a subsequent Hoare Logic proof. `P-While` is to `R-While` what `P-If` is to `R-If`, encoding the side-condition by inserting an assertion.

2.4.2 Core products

The core product program construction extends minimal product program rules with rules mimicking Core Relational Hoare Logic, with, much like the former, rules for reasoning about part of the left program:

$$\frac{}{x = e \times \textbf{skip} \longrightarrow x = e}\ \text{P-Assign-L} \qquad \frac{}{\textbf{assert}\ (b) \times \textbf{skip} \longrightarrow \textbf{assert}\ (b)}\ \text{P-Assert-L}$$

$$\frac{c_1 \times \textbf{skip} \longrightarrow c_1' \qquad c_2 \times c \longrightarrow c_2'}{c_1; c_2 \times c \longrightarrow c_1'; c_2'}\ \text{P-Seq-Skip-L}$$

$$\frac{c_1 \times c \longrightarrow c_1' \qquad c_2 \times c \longrightarrow c_2'}{\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \times c \longrightarrow \textbf{if}\ (b)\ c_1'\ \textbf{else}\ c_2'}\ \text{P-If-L}$$

… and rules for reasoning about part of the right program:

$$\frac{}{\textbf{skip} \times x = e \longrightarrow x = e}\ \text{P-Assign-R} \qquad \frac{}{\textbf{skip} \times \textbf{assert}\ (b) \longrightarrow \textbf{assert}\ (b)}\ \text{P-Assert-R}$$

$$\frac{\textbf{skip} \times c_1 \longrightarrow c_1' \qquad c \times c_2 \longrightarrow c_2'}{c \times c_1; c_2 \longrightarrow c_1'; c_2'}\ \text{P-Seq-Skip-R}$$

$$\frac{c \times c_1 \longrightarrow c_1' \qquad c \times c_2 \longrightarrow c_2'}{c \times \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \longrightarrow \textbf{if}\ (b)\ c_1'\ \textbf{else}\ c_2'}\ \text{P-If-R}$$

2.5 Shortcomings of RHL and Product Programs

Both Relational Hoare Logic and the closely-related product program construction rules mentioned above enable formal relational reasoning on pairs of programs. However, those logics only ensure partial correctness, and as such cannot be used for non-terminating or crashing programs.

In particular, *Relational Hoare Logic* and *product programs* cannot express relations between a crashing program and a non-crashing one, even though such changes—known as bugfixes—are frequent in the day-to-day practice of software development.

This is easily explained on product programs: as the instructions of both programs are re-used in the product program with the same meaning, any crashing instruction of either program gets included in the product program, which will also crash. For instance, consider the two following programs:

```
m = a % b;
if (m == 0)
  is_multiple = 1;
else
  is_multiple = 0;
```

```
if (b == 0) {
  is_multiple = 0;
} else {
  m = a % b;
  if (m == 0)
    is_multiple = 1;
  else
    is_multiple = 0;
}
```

In the above example, the left program will crash if “`b`” is set to 0, because it will attempt to divide “`a`” by 0. The right program avoids this issue by adding a conditional, setting the resulting value “`is_multiple`” to 0 if “`b`” is set to 0. This change could be described as fixing a known crash, and asserting that “is_multiple” is set to 0 for every input that makes the left program crash. This property cannot be stated in Relational Hoare Logic. After tagging left and right variables to avoid conflicts by respectively prefixing them with “`l_`” and “`r_`”, one possible product program would be:

```
1  if (r_b == 0) {
2    r_is_multiple = 0;
3    l_m = l_a % l_b;
4    if (l_m == 0)
5      l_is_multiple = 1;
6    else
7      l_is_multiple = 0;
8  } else {
9    l_m = l_a % l_b;
10   r_m = r_a % r_b;
11   assert ((l_m == 0) == (r_m == 0));
12   if (l_m == 0) {
13     l_is_multiple = 1;
14     r_is_multiple = 1;
15   } else {
16     l_is_multiple = 0;
17     r_is_multiple = 0;
18   }
19 }
```

One can see that in the example above, if "`l_b = 0`", even though the right program is fixed, the product program will execute code from the left program, and will crash at line 3, when attempting to divide by zero.

The issue illustrated above is the main shortcoming we will attempt to address in the remaining of this thesis.

Chapter 3

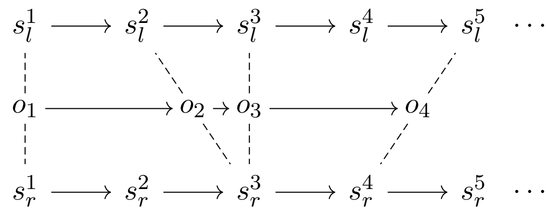
Correlating oracles

In this chapter, we present our main contribution, that is, the formal framework of *correlating oracles*, which purpose is to express and verify relations between the executions of a pair of programs. It is specifically designed to address the shortcomings of *product programs* and *correlating programs* by allowing reasoning on *crashing* and *non-terminating* programs.

This work has been summarily presented in an earlier paper[14] we published, but this chapter presents an updated version of this work, and presents it more extensively.

Our framework is based on an idea similar to that of *product programs* and *correlating programs*: indeed, much like them, *correlating oracles* are programs specifically constructed to relate pairs of programs by soundly simulating them. However, because we are interested in reasoning about crashing and non-terminating programs, the relations exhibited by *correlating oracles* cannot be characterized solely on the programs' final states, as those may not exist. Instead, correlating oracles relate the *execution* of both programs, a notion we will define more formally within this chapter.

In the meantime, we can informally describe a correlating oracle between a *left* program and a *right* program as a third program which successive states relate states of the left and right programs, as illustrated by the following diagram, in which the top line presents successive states of the *left* program, the bottom line represents those of the *right* program, the middle one that of the *correlating oracle*, arrows represent computation steps, and dashed lines the "relation" between states:



In the above diagram, the correlating oracle state o_1 relates the left program state s_l^1 with the right program state s_r^1. A single oracle computation step from o_1 then leads to the oracle state o_2, which relates the left program state s_l^2 to the right

program state s_r^3. Note that the oracle "skipped" the left program state s_r^2, not putting it in relation with any left program state. This ability is especially useful to "ignore" intermediary steps and highlight locally-equivalent ones whenever two programs compute the same result in two different ways.

Back to the example, a single computation step from oracle state o_2 leads to oracle state o_3, relating program states s_l^3 and s_r^3. In this case, the oracle did not simulate any right program step, and s_r^3 is related both to s_l^2 and s_l^3. This is especially useful when a program performs a computation that is not significant with regards to the relation we are interested in: for instance, we may be interested in showing that some variables take the same successive values in both programs, but one of the program performs extra operations on a variable we are not interested in.

In a sense, *correlating oracles* are *bi-interpreters*, as they *interpret* two programs at the same time: as illustrated above, an execution step of a correlating oracle corresponds to a finite number of steps of the two programs it relates. The choice of the verb "interpret" here rather than "simulate" is no accident: indeed, as we will see with most of our oracle languages, oracles will in practice often maintain an internal copy of the program states under consideration, inspect them, and *interpret* them if needed. This is not a requirement, though, and oracles may very well use different techniques, for instance maintaining an internal copy of just one of the two related programs and transform it to the other program's on-the-fly. Nevertheless, we want to place emphasis on the ability of *correlating oracles* to inspect and manipulate program states.

Indeed, while *correlating programs* simulate left and right programs by interleaving the two programs' statements based on syntactic or textual criteria, and *product programs* do so following an explicit semantic reasoning provided upfront, *correlating oracles* are typically written in a different, specific programming language and do not syntactically interleave actual instructions from left and right programs.

Much like a *program* is written in a *programming language*, *correlating oracles* in our framework are written in *oracle languages*. Our framework is generic in the sense that the definition of oracle languages is parameterized by two programming languages: indeed, each oracle language is specific to a pair of programming languages, and any oracle written in such a language is specific to a pair of programs written in the corresponding programming languages. This does not mean that an oracle language has to be able to relate every single pair of programs or that there should be only one oracle language per pair of programming languages. In fact, the approach we have taken is to write an oracle language for each "kind of relation" we wanted to exhibit: an oracle language for renaming variables, one oracle language for swapping independent assignments, etc.

This *correlating oracle* framework is formally defined within the Coq proof assistant: every definition in this section roughly corresponds to a formal Coq definition, and each of the instantiated oracle languages is proved correct within Coq as well. For every definition, lemma or theorem present in this section, the corresponding Coq definitions will be either linked or included.

3.1 General definition of a programming language

As stated earlier, our framework is generic with regards to programming languages. Those programming languages need to conform to a defined formalism, however, in order to be manipulated consistently. To this effect, we have chosen deterministic small-step operational semantics as our formalism. The reason for this choice is that we are interested in relating programs that may crash or not terminate, stating for instance that two non-terminating programs have infinitely many "locally-equivalent" program states, or that a crashing program's last non-crashed state somehow relates to a non-crashing program's execution.

Definition 11 (Programming language definition)**.** A programming language definition is a 7-uple $(\mathcal{P}, \mathcal{G}, \mathcal{S}, \mathcal{I}, \mathcal{E}, \mathcal{R}, \mathcal{F})$ such that:

- $\mathcal{P}$ is the type of *program abstract syntax trees*.
- $\mathcal{G}$ is the type of *static evaluation environments* or *global environments*, e.g. a table of class definitions in the case of object-oriented languages such as Java.
- $\mathcal{S}$ is the type of *dynamic evaluation environments* or *state*, e.g. heap, stack and program counter or continuation in Java.
- $\mathcal{I}$ is the *initialisation function* of type $\mathcal{P} \to \mathcal{G} \times \mathcal{S}$ to produce an initial program state (or configuration) given its source code as an abstract syntax tree.
- $\mathcal{E}$ is the partial *evaluation function* of type $\mathcal{G} \to \mathcal{S} \nrightarrow \mathcal{S}$ to execute program configurations step by step.
- $\mathcal{R}$ is the *program return type*, e.g. an integer for C programs.
- $\mathcal{F}$ is the *result extraction* partial function of type $\mathcal{S} \nrightarrow \mathcal{R}$ to extract the result of a successful final state.

In addition, these definitions must also fulfill the following requirements:

1. $\mathcal{F}$ only characterizes stuck configurations, that is: $\forall g : \mathcal{G}, \forall s : \mathcal{S}$, $\mathcal{E}\ g\ s$ is undefined if $\mathcal{F}\ s$ is defined.
2. $\mathcal{S}$ and $\mathcal{G}$ must be equipped with a decidable equality preserved by $\mathcal{E}$.

This presentation is relatively standard, and it is close to the one used in the CompCert certified compiler[26]. The main difference between our presentation and that of CompCert is that $\mathcal{E}$ is a transition function, and not a transition relation, thus requiring programming languages to be deterministic, a restriction that we will explain in more details in the conclusion of this thesis.

Definition 12 (Program configurations)**.** We call *configuration* a pair of a static evaluation environment and of a dynamic evaluation environment. The set of configurations of a programming language $\mathcal{L}$ is $\mathcal{C}_\mathcal{L} = \mathcal{G}_\mathcal{L} \times \mathcal{S}_\mathcal{L}$.

We say that a configuration is *stuck* (or, abusively, that a state is stuck) if $\mathcal{E}$ is not defined on it. In the remaining of this section, we may write $s \to s'$ for $\mathcal{E}\ g\ s = s'$ when there is no ambiguity about the static environment g. Likewise, we may write $s \not\to$ for $(g,\ s) \notin \mathrm{dom}(\mathcal{E})$.

Definition 13 (Terminating configurations)**.** A configuration $c \in \mathcal{C}$ is said to terminate (written $c \rightarrow^* \not\rightarrow$) if it reaches a *stuck* state.

This last state is called a *final state* if a result value can be extracted using $\mathcal{F}$. An execution that reaches a stuck state which is not a final state *crashes*.

3.2 Correlating oracles: operational semantics for differences

While they do not share the above definition, oracle languages are, at their heart, programming languages defined by a deterministic operational semantics. Those languages are, in addition, equipped with a few functions to *project* oracle states to *left* and *right* program states, as well as properties to assert some *invariant* property on oracle states and to prove the aforementioned projections to be *valid*.

Definition 14 (Correlating oracle language definition)**.** Given two language definitions $\mathcal{L}_1 = (\mathcal{P}_1, \mathcal{G}_1, \mathcal{S}_1, \mathcal{I}_1, \mathcal{E}_1, \mathcal{R}_1, \mathcal{F}_1)$ and $\mathcal{L}_2 = (\mathcal{P}_2, \mathcal{G}_2, \mathcal{S}_2, \mathcal{I}_2, \mathcal{E}_2, \mathcal{R}_2, \mathcal{F}_2)$, an *oracle language definition between $\mathcal{L}_1$ and $\mathcal{L}_2$* is a 8-uple $(\mathbb{G}, \mathbb{O}, \mathbb{S}, \pi'_L, \pi'_R, \pi_L, \pi_R, \mathbb{I})$ such that:

- $\mathbb{G}$ is the type of static evaluation environments.
- $\mathbb{O}$ is the type of dynamic evaluation environments.
- $\mathbb{S}$ is the interpretation function of type $\mathbb{G} \times \mathbb{O} \rightarrow\!\!\!\!\!\rightarrow \mathbb{O}$.
- π'_L (resp. π'_R) is a projection function of type $\mathbb{G} \rightarrow \mathcal{G}_1$ (resp. $\mathbb{G} \rightarrow \mathcal{G}_2$).
- π_L (resp. π_R) is a projection function of type $\mathbb{O} \rightarrow \mathcal{S}_1$ (resp. $\mathbb{O} \rightarrow \mathcal{S}_2$).
- $\mathbb{I}$ is an invariant of type $\mathbb{G} \times \mathbb{O} \rightarrow \mathbb{P}$.

with the following additional requirements ensuring its soundness:

1. $\mathbb{I}$ is preserved by $\mathbb{S}$ i.e. $\forall g : \mathbb{G},\ \forall o : \mathbb{O},\ \mathbb{I}(g, o) \Rightarrow \mathbb{I}(g, \mathbb{S}(g, o))$
2. $\mathbb{S}$ leads to *correct* and *productive* predictions i.e. $\forall g : \mathbb{G},\ \forall o\, o' : \mathbb{O}$,

$$\mathbb{I}(g, o) \Rightarrow \mathbb{S}(g, o) = o' \Rightarrow \exists\, n_1\, n_2, \begin{cases} \pi_L(\mathbb{S}(g, o)) & = & \mathcal{E}_1^{n_1}(\pi'_L(g), \pi_L(o)) \\ \pi_R(\mathbb{S}(g, o)) & = & \mathcal{E}_2^{n_2}(\pi'_R(g), \pi_R(o)) \\ n_1 + n_2 > 0 \end{cases}$$

3. the oracle is complete, in the sense that it only terminates when both underlying programs themselves terminate, i.e. $\forall g : \mathbb{G},\ \forall o : \mathbb{O}$,

$$(g,\ o) \notin \mathrm{dom}(\mathbb{S}) \Rightarrow (\pi'_L(g), \pi_L(o)) \notin \mathrm{dom}(\mathcal{E}_1) \wedge (\pi'_R(g), \pi_R(o)) \notin \mathrm{dom}(\mathcal{E}_2)$$

Each oracle configuration thus relates a pair of program configurations that can be recovered using the appropriate projection functions. The oracle asserts some property on those pairs of configurations through the invariant $\mathbb{I}$, which is defined on oracle configurations rather than on projected program configurations: indeed, additional runtime information may be needed for the oracle to execute properly, as

well as to prove its soundness, as we will see with the oracle languages defined in Chapters 4 and 5.

The *productivity* condition, that is, the fact that each oracle step performed by $\mathbb{S}$ corresponds to at least one reduction step in either program ($n_1 + n_2 > 0$ in requirement 2.) is essential to ensuring that the oracle does not “silently stop relating programs” by performing non-productive reductions.

Remark. As one may notice, the definition of *oracle languages* resembles that of *programming languages*: both have static and dynamic evaluation environments, as well as a step function. Therefore, one may wonder why *oracle languages* are not *programming languages*. The reasons for this are mainly practical: indeed, as it is reasonable for oracle to be able to verify equality between two program states, we require in our Coq development that two languages have decidable equality for their states. Unless we are interested in having oracles relating other oracles,—which we cannot find a use-case for—such a requirement is unneeded and would significantly clutter formal oracle language definitions for no clear benefit. Furthermore, a *result extraction* function on oracle configurations would be of limited use, and syntactic representations of oracles will be covered in the next chapter.

To illustrate the execution of an oracle, we will often present its execution trace along that of the programs it relates, as in the following diagram:

$$
\begin{array}{ccccccccccc}
s_l^1 & \xrightarrow{\mathcal{E}_l} & s_l^2 & \xrightarrow{\mathcal{E}_l} & s_l^3 & \xrightarrow{\mathcal{E}_l} & s_l^4 & \xrightarrow{\mathcal{E}_l} & s_l^5 & \cdots \\
\uparrow \pi_L & & \pi_L \nwarrow & & \uparrow \pi_L & & & \pi_L \nearrow & & \\
o_1 & \xrightarrow{\mathbb{S}} & & o_2 \xrightarrow{\mathbb{S}} & o_3 & \xrightarrow{\mathbb{S}} & & o_4 & & \\
\downarrow \pi_R & & & \pi_R \searrow & \downarrow \pi_R & & \pi_R \swarrow & & & \\
s_r^1 & \xrightarrow[\mathcal{E}_r]{} & s_r^2 & \xrightarrow[\mathcal{E}_r]{} & s_r^3 & \xrightarrow[\mathcal{E}_r]{} & s_r^4 & \xrightarrow[\mathcal{E}_r]{} & s_r^5 & \cdots
\end{array}
$$

In the diagram above, s_l^1 to s_l^5 are successive dynamic states of the left program while s_r^1 to s_r^5 are successive dynamic states of the right program, o_1 to o_4 are successive dynamic states of the oracle, $\mathcal{E}_l$ and $\mathcal{E}_r$ are the step functions of the languages the left and right program are respectively written in, and every static state is omitted for conciseness.

The main advantage of our approach over product programs and correlating programs is that *correlating oracles*, by not actually interleaving instructions of a pair of programs, but rather *interpreting* them and *inspecting* program states, are able to relate them even if one of them gets stuck. This is essential to characterizing some bugfixes, which are a very common type of changes. Some correlating oracle languages dedicated to these kind of changes will be defined in Chapter 4. When relating a crashed program with a non-crashed one, such oracles will only step in the non-crashing one.

The reason for which those small-step operational semantics are deterministic is mainly historic and will be covered in more details in the conclusion of this thesis.

3.2.1 Identity and universal oracles

Two really simple language-agnostic oracle languages can be defined within our framework: the identity oracle and the universal oracle. Those two examples are language-agnostic in the sense that they are parameterized by one (in the case of the identity oracle) or two (in the universal oracle's case) programming languages on which they pose no restriction. Of course, those two oracle languages are of very limited practical interest, as they either convey trivial information (in the identity oracle's case, that the two programs are syntactically strictly equal) or no information at all (in the universal oracle's case) but they are suitable for a decent practical introduction to our framework's formal definitions.

3.2.1.1 Identity oracle

The identity oracle relates pairs of programs that are exactly the same, evaluating them in lock-step and asserting that for each such step, both programs' configurations are identical.

Definition 15 (Identity oracle language)**.** The identity oracle language is defined for every language definition $\mathcal{L} = (\mathcal{P}, \mathcal{G}, \mathcal{S}, \mathcal{I}, \mathcal{E}, \mathcal{R}, \mathcal{F})$ as the oracle language definition $(\mathcal{G}, \mathcal{S}, \mathcal{E}, id, id, id, id, \mathbb{I})$ where:

- $id(x) = x$ is the polymorphic identity function
- $\mathbb{I}(g,\ s) = \top$ is the universally-valid invariant

The identity oracle language is thus a thin wrapper around the underlying programming language, executing a single instance of a program using its standard semantics. Since the related programs are identical, their configurations are recovered by the projection functions, which simply returns the oracle's internal configuration.

$$\begin{array}{ccccccccc}
s^1 & \xrightarrow{\mathcal{E}} & s^2 & \xrightarrow{\mathcal{E}} & s^3 & \xrightarrow{\mathcal{E}} & s^4 & & \cdots \\
{\scriptstyle id}\uparrow & & {\scriptstyle id}\uparrow & & {\scriptstyle id}\uparrow & & {\scriptstyle id}\uparrow & & \\
s^1 & \xrightarrow{\mathcal{E}} & s^2 & \xrightarrow{\mathcal{E}} & s^3 & \xrightarrow{\mathcal{E}} & s^4 & & \cdots \\
\downarrow{\scriptstyle id} & & \downarrow{\scriptstyle id} & & \downarrow{\scriptstyle id} & & \downarrow{\scriptstyle id} & & \\
s^1 & \xrightarrow[\mathcal{E}]{} & s^2 & \xrightarrow[\mathcal{E}]{} & s^3 & \xrightarrow[\mathcal{E}]{} & s^4 & & \cdots
\end{array}$$

Lemma 3 (Invariant preservation of the identity oracle)**.** *Invariant preservation is immediate, as the invariant is always satisfied no matter the internal state.*

Lemma 4 (Prediction soundness of the identity oracle)**.** *Prediction soundness is immediate, as each step of the identity oracle performs exactly one step of the underlying program.*

Lemma 5 (Prediction completeness of the identity oracle)**.** *As the program configurations under consideration are the same as the oracle configuration, the oracle only terminates when the two underlying (identical) programs terminate.*

3.2.1.2 Universal oracle

The universal oracle can relate any pair of programs without asserting any precise relation between them, by relating their successive configurations arbitrarily.

Definition 16 (Universal oracle language)**.** The universal oracle language is defined for every pair of language definitions $\mathcal{L}_1 = (\mathcal{P}_1, \mathcal{G}_1, \mathcal{S}_1, \mathcal{I}_1, \mathcal{E}_1, \mathcal{R}_1, \mathcal{F}_1)$ and $\mathcal{L}_2 = (\mathcal{P}_2, \mathcal{G}_2, \mathcal{S}_2, \mathcal{I}_2, \mathcal{E}_2, \mathcal{R}_2, \mathcal{F}_2)$ as the oracle language definition $(\mathcal{G}_1 \times \mathcal{G}_2,\ \mathcal{S}_1 \times \mathcal{S}_2,\ \mathbb{S},\ fst,\ snd,\ fst,\ snd,\ \mathbb{I})$ where:

- $fst((x, y)) = x$
- $snd((x, y)) = y$
- $\mathbb{S}((g_1,\ g_2),\ (s_1,\ s_2)) = \begin{cases} (\mathcal{E}_1(g_1, s_1), \mathcal{E}_2(g_2, s_2)) & \text{if } (g_1, s_1) \in \text{dom}(\mathcal{E}_1) \wedge (g_2, s_2) \in \text{dom}(\mathcal{E}_2) \\ (s_1, \mathcal{E}_2(g_2, s_2)) & \text{if } (g_1, s_1) \notin \text{dom}(\mathcal{E}_1) \wedge (g_2, s_2) \in \text{dom}(\mathcal{E}_2) \\ (\mathcal{E}_1(g_1, s_1), s_2) & \text{if } (g_1, s_1) \in \text{dom}(\mathcal{E}_1) \wedge (g_2, s_2) \notin \text{dom}(\mathcal{E}_2) \\ \text{undefined otherwise} & \end{cases}$
- $\mathbb{I}(g,\ s) = \top$

A universal oracle configuration consists of the configurations of the two "related" programs, and their evaluation is delegated to the evaluation function of the corresponding languages.

$$\begin{array}{ccccccccc}
s_l^1 & \xrightarrow{\mathcal{E}} & s_l^2 & \xrightarrow{\mathcal{E}} & s_l^3 & \xrightarrow{\mathcal{E}} & s_l^4 & \cdots \\
{\scriptstyle fst}\uparrow & & {\scriptstyle fst}\uparrow & & {\scriptstyle fst}\uparrow & & {\scriptstyle fst}\uparrow & \\
(s_l^1,\ s_r^1) & \xrightarrow{\mathbb{S}} & (s_l^2,\ s_r^2) & \xrightarrow{\mathbb{S}} & (s_l^3,\ s_r^3) & \xrightarrow{\mathbb{S}} & (s_l^4,\ s_r^4) & \cdots \\
\downarrow{\scriptstyle snd} & & \downarrow{\scriptstyle snd} & & \downarrow{\scriptstyle snd} & & \downarrow{\scriptstyle snd} & \\
s_r^1 & \xrightarrow[\mathcal{E}]{} & s_r^2 & \xrightarrow[\mathcal{E}]{} & s_r^3 & \xrightarrow[\mathcal{E}]{} & s_r^4 & \cdots
\end{array}$$

In case one of the programs gets stuck before the other, the universal oracle will continue to interpret the remaining one:

$$\begin{array}{ccccccccc}
s_l^1 & \xrightarrow{\mathcal{E}} & s_l^2 & \xrightarrow{\mathcal{E}} & s_l^3 & & & \\
{\scriptstyle fst}\uparrow & & {\scriptstyle fst}\uparrow & & {\scriptstyle fst}\uparrow & \nwarrow{\scriptstyle fst} & & \\
(s_l^1,\ s_r^1) & \xrightarrow{\mathbb{S}} & (s_l^2,\ s_r^2) & \xrightarrow{\mathbb{S}} & (s_l^3,\ s_r^3) & \xrightarrow{\mathbb{S}} & (s_l^4,\ s_r^4) & \cdots \\
\downarrow{\scriptstyle snd} & & \downarrow{\scriptstyle snd} & & \downarrow{\scriptstyle snd} & & \downarrow{\scriptstyle snd} & \\
s_r^1 & \xrightarrow[\mathcal{E}]{} & s_r^2 & \xrightarrow[\mathcal{E}]{} & s_r^3 & \xrightarrow[\mathcal{E}]{} & s_r^4 & \cdots
\end{array}$$

Lemma 6 (Invariant preservation of the universal oracle)**.** *Invariant preservation is immediate, as in the Identity oracle's case, since the oracle's invariant is always satisfied.*

Lemma 7 (Prediction soundness of the universal oracle). *Soundness of the universal oracle is guaranteed by each oracle step corresponding to exactly one step of at least one of the two underlying programs.*

Lemma 8 (Prediction completeness of the universal oracle). *Prediction completeness of the universal oracle is immediate as it keeps executing as long as at least one of the underlying programs keep executing.*

3.2.2 General theorems

Our main generic theoretical result on correlating oracles is the following: given two programs whose executions terminate in a finite number of steps, any sound oracle relating them will reach those programs' stuck states in a finite number of states. This is a somewhat direct consequence of the productivity condition of oracle languages, and is particularly useful to prove extensional properties, as we will see in Chapter 5.

Theorem 3 (Oracles between terminating programs reach terminating states). *Given an oracle language* $(\mathbb{G}, \mathbb{O}, \mathbb{S}, \pi'_L, \pi'_R, \pi_L, \pi_R, \mathbb{I})$ *between programs written in languages* $\mathcal{L}_1 = (\mathcal{P}_1, \mathcal{G}_1, \mathcal{S}_1, \mathcal{I}_1, \mathcal{E}_1, \mathcal{R}_1, \mathcal{F}_1)$ *and* $\mathcal{L}_2 = (\mathcal{P}_2, \mathcal{G}_2, \mathcal{S}_2, \mathcal{I}_2, \mathcal{E}_2, \mathcal{R}_2, \mathcal{F}_2)$*,*

- *for every oracle configuration* (g, o) *such that* $\mathbb{I}(g, o)$*,*
- *for every natural numbers* n*,* n_1*, and* n_2 *such that* $n \leq n_1 + n_2$*,*
- *for every left program state* s'_1 *such that* $\pi_L(o) \rightarrow^{n_1} s'_1 \not\rightarrow$*,*
- *for every right program state* s'_2 *such that* $\pi_R(o) \rightarrow^{n_2} s'_2 \not\rightarrow$*, and,*
- *for every oracle state* o' *such that* $\mathbb{S}'^n(g, o) = o'$

$$\text{where } \mathbb{S}'(g, o) = \begin{cases} \mathbb{S}(g, o) & \text{if } (g, o) \in dom(\mathbb{S}) \\ (g, o) & \text{if } (g, o) \notin dom(\mathbb{S}) \end{cases}$$

then

- $\pi_L(o') = s'_1$
- $\pi_R(o') = s'_2$
- $\mathbb{I}(g, o')$

That is, for every oracle global environment g *and every oracle state* o *on which the oracle's invariant* $\mathbb{I}$ *holds and that can be projected to left and right configurations that reduce to stuck states* s'_1 *in* n_1 *steps and* s'_2 *in* n_2 *steps respectively, then the oracle* o *reaches in at most* $n_1 + n_2$ *steps an oracle state* o' *that projects to the final configurations* s'_1 *and* s'_2*.*

Proof. The proof is done by induction on n.

In the base case, we have $n = 0$. Since $n \leq n_1 + n_2$, $n_1 = n_2 = 0$, so $s'_1 = \pi_L(o)$, and $s'_2 = \pi_R(o)$, and $o' = o$.

In the other case, with $n > 0$, there are two subcases:

- Either $(g, o) \notin \mathrm{dom}(\mathbb{S})$, in which case $\pi_L(o)$ and $\pi_R(o)$ are, by the oracle's completeness condition, stuck states, and thus are equal to s'_1 and s'_2 respectively, and $\mathbb{S}'^n(g, o) = o$.
- Or $(g, o) \in \mathrm{dom}(\mathbb{S})$, which is a bit more involved. Indeed, in this case, we have $\mathbb{S}'^n(g, o) = \mathbb{S}^{n-1}(g, \mathbb{S}(g, o))$ and, before being able to use the induction hypothesis, we need to handle this first oracle step, which generates three more subcases as the oracle may simulate steps in the left program, in the right program, or in both programs in a single step. Each of these subcases involves subtracting to n_1 and n_2 the number of steps predicted by the oracle in the corresponding programs, then use the induction hypothesis accordingly.

□

Theorem 4 (Oracles between terminating programs terminate)**.** *Given an oracle language* $(\mathbb{G}, \mathbb{O}, \mathbb{S}, \pi'_L, \pi'_R, \pi_L, \pi_R, \mathbb{I})$ *,*

$$\begin{array}{l}\forall g : \mathbb{G},\ \forall o : \mathbb{O},\ \mathbb{I}(g, o) \Rightarrow \\ (\pi'_L(g), \pi_L(o)) \to^* \not\to \Rightarrow \\ (\pi'_R(g), \pi_R(o)) \to^* \not\to \Rightarrow \\ \exists\, n\, o',\ \mathbb{S}^n(g, o) = o' \ \wedge\ \mathbb{I}(g, o') \ \wedge\ \pi_L(o') \not\to \ \wedge \pi_R(o') \not\to\end{array}$$

That is, for every oracle global environment g and every oracle state o on which the oracle's invariant $\mathbb{I}$ holds and that can be projected to left and right configurations resulting both in converging executions, o reduces to a final oracle state o' after a finite number of reduction steps.

Proof. This theorem is a direct consequence of Theorem 3.

□

Chapter 4

Oracle languages for Imp

While we have defined two generic oracle languages in the previous chapter, most informative oracle languages are specific to a pair of programming languages. For this reason, we have designed a collection of oracle languages on the idealized imperative language Imp presented in the previous chapter.

The choice of Imp over Imp_{br} as a target language can be explained by our focus moving from studying extensional relational properties to intensional ones: indeed, such a change in focus also implied a change from a big-step presentation of the semantics to a small-step operational semantics. Since the early exit constructs of Imp_{br} would have made the small-step semantics noticeably more complex for no fundamental benefit, we dropped them in the process for simplicity's sake.

The different oracle languages we have defined on Imp constitute a first practical exploration of the language design questions arising from our framework, as well as a first informal evaluation of its expressivity. As such, each of those oracle languages corresponds to a specific class of differences, demonstrating different capabilities of the framework. For instance, some of those oracle languages describe pairs of programs that perform exactly the same number of reduction steps, with each pair of configurations tightly related. This would be the case of an oracle language relating programs that are equal modulo renaming of variable identifiers, for instance. Other oracle languages abstract several computation steps in order to reach a related pair of program configurations, which may be needed to describe more involved code refactoring or optimizations. Finally, some of those oracle languages will describe pairs of extensionally-equivalent programs while others will relate pairs of programs presenting different outcomes.

It must be noted that, when designing those oracle languages, our main preoccupation was to figure out what kind of changes could be represented and how. Furthermore, as each of those oracle languages has been fully formalized in the Coq proof assistant, some compromises had to be made to keep the quantity and difficulty of proofs manageable when exploring different possibilities for oracle languages. As a result, the oracle languages presented in the chapter may not be the most *practical* or *generic* possible for the kinds of changes they describe.

While we have already defined Imp's big-step semantics, our framework requires small-step semantics. The small-step semantics of Imp is defined on *states*,—that is, pairs of a *store* S and a *continuation* κ—using big-step evaluation for expressions and boolean expressions:

Definition 17 (Imp states).

$$\begin{array}{rcl} S_{\text{Imp}} & ::= & (\kappa, S) \\ \kappa & ::= & \textbf{halt} \mid c \cdot \kappa \end{array}$$

Just as with the big-step semantics defined earlier, a store S is a partial map from variable identifiers to integer values. In addition, a program state S_{Imp} also includes a continuation κ, that is a stack of statements to be executed next, with the very next statement on the top of the stack.

Definition 18 (Imp transition function). The semantics is then defined on states by the following transition function $\mathcal{E}_{Imp}$, interpreting the statement on the top of the continuation stack:

- When the statement at the top of the stack is a **skip**, it is discarded without modifying the store:

$$\mathcal{E}_{Imp}(\textbf{skip} \cdot \kappa, S) = (\kappa, S)$$

- When that statement is an assignment, it is removed from the stack, the right-hand-side expression is evaluated in the current store and the result is stored in the variable x. If the right-hand-side expression cannot be evaluated in the current store, the program crashes:

$$\mathcal{E}_{Imp}(x = e \cdot \kappa, S) = (\kappa, S[x \mapsto n]) \qquad \text{where } S \vdash e \Downarrow n$$

- If the head statement is a sequence, it is unfolded. That is, it is removed from the stack, and the two sub-statements are pushed in its place:

$$\mathcal{E}_{Imp}((c_1; c_2) \cdot \kappa, S) = (c_1 \cdot (c_2 \cdot \kappa), S)$$

- When the head statement is a conditional, it is removed from the continuation, and the condition expression is evaluated in the current store. If it cannot be evaluated, the program crashes. Otherwise, the sub-statement corresponding to the appropriate branch is pushed on the stack:

$$\begin{array}{ll} \mathcal{E}_{Imp}(\textbf{if}\,(b)\ c_1\ \textbf{else}\ c_2 \cdot \kappa, S) = (c_1 \cdot \kappa, S) & \quad \text{where } S \vdash b \Downarrow \textbf{true} \\ \mathcal{E}_{Imp}(\textbf{if}\,(b)\ c_1\ \textbf{else}\ c_2 \cdot \kappa, S) = (c_2 \cdot \kappa, S) & \quad \text{where } S \vdash b \Downarrow \textbf{false} \end{array}$$

- Upon encountering a **while** statement, its condition is evaluated in the current store. If the condition cannot be evaluated, the program crashes. If it evaluates to **true**, the loop's body is pushed onto the stack. If it evaluates to **false**, the loop is removed from the stack:

$$\begin{array}{ll} \mathcal{E}_{Imp}(\textbf{while}\,(b)\ c \cdot \kappa, S) = (c \cdot (\textbf{while}\,(b)\ c \cdot \kappa), S) & \quad \text{where } S \vdash b \Downarrow \textbf{true} \\ \mathcal{E}_{Imp}(\textbf{while}\,(b)\ c \cdot \kappa, S) = (\kappa, S) & \quad \text{where } S \vdash b \Downarrow \textbf{false} \end{array}$$

- Finally, when an **assert** statement is encountered, its boolean expression is evaluated. If it does not evaluate to **true**, the program crashes. Otherwise, the statement is simply popped from the stack:

$$\mathcal{E}_{Imp}(\textbf{assert}\ (b) \cdot \kappa, S) = (\kappa, S) \qquad \text{where } S \vdash b \Downarrow \textbf{true}$$

Lemma 9 (Small-step and big-step semantics for Imp are equivalent)**.** *This small-step definition is equivalent to the big-step presentation defined earlier, that is, every finite execution of a single statement in a given store S results in the same store S' whether it is considered through the small-step semantics or the big-step one:*

$$\forall S\ S'\ c,\ S \vdash c \Downarrow S' \Leftrightarrow \exists n,\ \mathcal{E}^n_{Imp}(c \cdot \textbf{halt}, S) = (\textbf{halt}, S')$$

Now that we have given Imp a deterministic small-step semantics, we can use it in our framework.

Definition 19 (Imp programming language definition)**.** The language definition for Imp is given by the tuple $\mathcal{L}_{Imp} = (c,\ unit,\ S_{\text{Imp}},\ \mathcal{I}_{Imp},\ \mathcal{E}'_{Imp},\ unit,\ \mathcal{F}_{Imp})$ where

- $unit$ is the *unit* type with a single inhabitant tt
- c is the type of Imp abstract syntax trees as defined in Chapter 2
- $\mathcal{E}'_{Imp}(_, S_{\text{Imp}}) = \mathcal{E}_{Imp}(S_{\text{Imp}})$
- $\mathcal{F}_{Imp}((\textbf{halt}, S)) = tt$
- $\mathcal{I}_{Imp}(c) = (c \cdot \textbf{halt}, \emptyset)$

Notice that the semantics of Imp does not involve any global environment, as everything needed is included in its dynamic state. For that reason, the definition $\mathcal{L}_{Imp}$ makes use of the type $unit$ as its global environment type, as it has only one inhabitant tt. Its step function $\mathcal{E}'_{Imp}$ is a light wrapper around $\mathcal{E}_{Imp}$ in order to accomodate for the global environment, which is discarded as it does not hold information. An Imp program is considered to be terminated (without errors) if it has reached a configuration with an empty continuation. As Imp programs do not have proper return values, their return type is also $unit$.

In the following sections, we will go through all of the Imp-specific oracle languages we have defined in our Coq development. While those different oracles tackle various challenges and are only loosely related to each other, we attempt to maintain a progression in our presentation. Indeed, we will start from the conceptually simpler oracles, and we will try to present them in an order such that each presented language will only introduce a minimal number of new difficulties and proof techniques.

The Coq development itself will be presented in a more hands-on way in Chapter 7.

4.1 Renaming

One simple yet informative Imp-specific class of differences is that of valid renamings (akin to α-renaming in λ-calculus). Informally, a renaming is a syntactic operation

replacing all occurrences of some variable identifier in a program by another variable identifier.

Example 5. The following pair of programs, in which the variable "a" from the left program has been renamed to "sum" in the second one, illustrates such a change:

`a = 0;`
`d = x;`
`while (0 < d) {`
`  if (x % d == 0)`
`    a = a + d;`
`  d = d - 1;`
`}`
	`sum = 0;`
	`d = x;`
	`while (0 < d) {`
	`  if (x % d == 0)`
	`    sum = sum + d;`
	`  d = d - 1;`
	`}`

In order for the modified program to preserve the semantics of the original program,—in the sense that every step should yield the same results modulo variable renaming— renamed variables must not conflict with other variables: if two variables were distinct before the renaming, they have to be distinct after the renaming. We decided to represent renaming operations by a bijection between variable identifiers, as bijections avoid such conflicts.

Definition 20 (`Renaming` language on Imp)**.** The renaming oracle language `Renaming`[1] on Imp is defined by the oracle language definition

$$(\mathbb{G}_{ren},\ S_{\text{Imp}} \times S_{\text{Imp}},\ \mathbb{S}_{ren},\ 1_{tt},\ 1_{tt},\ fst,\ snd,\ \mathbb{I}_{ren})$$

where:

- $1_{tt}(_) = tt$ is the constant function returning tt
- $\mathbb{G}_{ren}$ is the set of bijections from variable identifiers to variable identifiers
- $\mathbb{I}_{ren}(\phi,\ (s_1,\ s_2))$ holds iff $\phi(s_1) = s_2$, where $\phi(s_1)$ is the term obtained by replacing every variable identifier x by $\phi(x)$ in s_1
- $\mathbb{S}_{ren}(tt,\ (s_1,\ s_2)) = (s_1',\ s_2')$ if $\mathcal{E}_{Imp}(s_1) = s_1'$ and $\mathcal{E}_{Imp}(s_2) = s_2'$

The oracles written in this language thus relate pairs of programs that are renamings of one another, having exactly the same number of execution steps, yielding at each point pairs of configurations that are equivalent modulo renaming.

4.2 Control-flow-preserving value changes

Another useful class of differences on Imp is the class of pairs of programs which share the exact same dynamic control-flow while allowing differing values for a given set of variable identifiers. The oracle language `ValueChange` describes a subset of such differences, restricted to pairs of programs matching an easily-checkable syntactic criterion. Informally, this criterion requires that:

[1]from the name of the Coq module implementing this oracle language.

1. Potentially modified variables cannot appear in conditionals.
2. Programs can only differ on assignments of modified variables.
3. In order to avoid both programs having different crashing behaviors, modified assignments must feature the same set of variable identifiers and divisor expressions.
4. Whenever a potentially modified variable appears in the right-hand side of an assignment, that assignment's left-hand side must also be considered as a potentially modified variable.

Example 6. One example pair of programs that can be related using this oracle language is the following, where the right program is syntactically identical to the left one except for the initial assignment of the variable `x`:

Left	Right

`x = 42;`
`count = 5;`
`pow = 1;`
`while (0 < count) {`
`  count = count - 1;`
`  pow = pow * x;`
`}`
	`x = 10;`
	`count = 5;`
	`pow = 1;`
	`while (0 < count) {`
	`  count = count - 1;`
	`  pow = pow * x;`
	`}`

In that example, the variables dynamically affected by this change are `x` and `pow`, and the control flow is provably preserved, since the only conditional expression (`0 < count`) does not depend on `x` nor on `pow`.

The exact criterion defined on continuations is captured by the judgment "$c_1 \equiv_{\overline{x}} c_2$" stating that c_1 and c_2 have the same control flow and may only differ in the value of variable identifiers included in $\overline{x}$. This judgment is formally defined in Figure 4.1.

Definition 21 (`ValueChange` language on Imp)**.** The oracle language `ValueChange` of control-flow-preserving value changes in Imp programs is defined by the oracle language definition

$$(\mathbb{G}_{VC},\ S_{\text{Imp}} \times S_{\text{Imp}},\ \mathbb{S}_{VC},\ 1_{tt},\ 1_{tt},\ fst,\ snd,\ \mathbb{I}_{VC})$$

where:

- $\mathbb{G}_{VC}$ is the set of lists of variable identifiers
- $\mathbb{I}_{VC}(\overline{x},\ ((\kappa_1, S_1),\ (\kappa_2, S_2))) =$

$$\kappa_1\ \equiv_{\overline{x}}\ \kappa_2 \land dom(S_1) = dom(S_2) \land \forall x \notin \overline{x} \Rightarrow S_1(x) = S_2(x)$$

- $\mathbb{S}_{VC}(tt,\ (s_1,\ s_2)) = (s'_1,\ s'_2)$ if $\mathcal{E}_{Imp}(s_1) = s'_1$ and $\mathcal{E}_{Imp}(s_2) = s'_2$

$$\textsc{Control-Skip}\quad \frac{}{\textbf{skip} \equiv_{\overline{x}} \textbf{skip}}$$

$$\textsc{Control-Seq}\quad \frac{c_1^1 \equiv_{\overline{x}} c_1^2 \qquad c_2^1 \equiv_{\overline{x}} c_2^2}{c_1^1; c_2^1 \equiv_{\overline{x}} c_1^2; c_2^2}$$

$$\textsc{Control-If}\quad \frac{c_1^1 \equiv_{\overline{x}} c_1^2 \qquad c_2^1 \equiv_{\overline{x}} c_2^2 \qquad \forall y,\, y \in vars(b) \Rightarrow y \notin \overline{x}}{\textbf{if}\,(b)\; c_1^1 \;\textbf{else}\; c_2^1 \equiv_{\overline{x}} \textbf{if}\,(b)\; c_1^2 \;\textbf{else}\; c_2^2}$$

$$\textsc{Control-While}\quad \frac{c^1 \equiv_{\overline{x}} c^2 \qquad \forall y,\, y \in vars(b) \Rightarrow y \notin \overline{x}}{\textbf{while}\,(b)\; c^1 \equiv_{\overline{x}} \textbf{while}\,(b)\; c^2}$$

$$\textsc{Control-Assert}\quad \frac{\forall y,\, y \in vars(b) \Rightarrow y \notin \overline{x}}{\textbf{assert}\,(b) \equiv_{\overline{x}} \textbf{assert}\,(b)}$$

$$\textsc{Control-Same-Assign}\quad \frac{\forall y,\, y \in vars(e) \Rightarrow y \notin \overline{x}}{x = e \equiv_{\overline{x}} x = e}$$

$$\textsc{Control-Diff-Assign}\quad \frac{y \in \overline{x} \qquad vars(e_1) = vars(e_2) \qquad divs(e_1) = divs(e_2)}{y = e_1 \equiv_{\overline{x}} y = e_2}$$

$$\textsc{Control-Halt}\quad \frac{}{\textbf{halt} \equiv_{\overline{x}} \textbf{halt}}$$

$$\textsc{Control-Cmd}\quad \frac{c_1 \equiv_{\overline{x}} c_2}{c_1 \cdot \kappa_1 \equiv_{\overline{x}} c_2 \cdot \kappa_2}$$

where, informally, $vars(e)$ is the set of variable identifiers appearing in e and $divs(e)$ is the set of divisor expressions appearing in e. The formal definition of $vars$ and $divs$ can be found in the Coq development.

Figure 4.1: Syntactic criterion implying preserved control-flow

As one can see, the definition of `ValueChange`'s step function is straightforward, as it simply interprets the two programs in lockstep, without maintaining any additional information. Its invariant, however, is slightly more involved: in addition to stating that the two programs' continuations are related by the criterion discussed above, it also requires the two programs' stores to contain exactly the same variable identifiers, in order to ensure that, should an expression contain an unknown variable identifier, its evaluation will cause both programs to crash. In addition, variables are only allowed to have different values if they are listed as potentially modified variables, in order to ensure possible effects on the control flow are properly tracked.

It must be noted that the judgment $\kappa_1 \equiv_{\overline{x}} \kappa_2$ is very restrictive. In particular, the list of variables allowed to change is global, meaning that dependence propagation between variables does not only go forward but also *backward*.

The reason for this unusual restriction boils down to a compromise between capturing interesting changes and keeping the Coq proof manageable.

4.3 Branches swapping

The oracle language `SwapBranches` for *branches swapping* describes pairs of programs identical except for swapped "then" and "else" branches of conditionals which have their condition negated. The affected conditional statements are identified by their path in the abstract syntax tree of the programs under consideration. Such oracles may seem exceedingly simple, as they execute both programs in lock-step, yielding identical stores for both programs at each step. However, unlike every other oracle language we have presented so far, `SwapBranches` describes changes that are local in nature. Indeed, branch swaps only affect a handful of reduction steps: those where the head term is the modified conditional. As a consequence, in order to track when the those reductions take place, `SwapBranches` maintains some dynamic information in addition to a copy of both programs' configurations. This pattern of dynamically tracking syntactic changes will be reused in many of our other oracle languages.

Example 7. One example pair of programs that can be related using this oracle language is:

`if (x < y)`
`  result = x + 42;`
`else`
`  result = y;`
	`if (!(x < y))`
	`  result = y;`
	`else`
	`  result = x + 42;`

As hinted earlier, in order to perform the branch swap at the appropriate time, the oracle needs to do some bookkeeping to find out when the affected conditionals are encountered. This is done by maintaining a data structure mimicking the shape of the continuations of the programs under comparison: that data structure is a list of changes to be performed on the continuation or on statements at a given point of the execution. Because this pattern will be common to several other oracle languages, this structure is parametric in the actual type of changes to be applied to statements and continuations:

Definition 22 (Dynamic modifier structure on Imp). For a given type α of actual statement-level changes and a type β of actual continuation-level changes, Imp *dynamic modifiers* are defined by the following syntax:

$$\begin{array}{rcl} \delta c & ::= & \Delta_{\texttt{Leaf}}(\alpha) \mid \texttt{Id} \mid \Delta_{\texttt{Rec}}(\delta c,\, \delta c) \\ \delta\kappa & ::= & \Delta_c(\delta c) \mid \Delta_\kappa(\beta) \end{array}$$

That is, a statement modifier δc defines a tree where leaves are either the identity modification `Id` or an actual statement-level modifier α, while a continuation modifier $\delta\kappa$ defines either a continuation-level modifier β or a statement modifier δc that applies to the statement on top of a continuation.

This continuation modifier structure is manipulated by the $\mathbb{S}_{\texttt{Helper}}$ helper function of type $S_{\text{Imp}} \to S_{\text{Imp}} \to \delta c \to \overline{\delta\kappa} \nrightarrow \mathit{step_helper_res}$, which executes the left

and right programs in lockstep whenever the continuation modifier indicates a local identity, maintaining the continuation modifier in the process. Whenever the head term of the continuation modifier is an actual statement-level or continuation-level change, a special value is returned to signify that local change has to be handled specifically. This helper function is a partial function which is not defined when the two programs under comparison do not reduce using the same reduction rule although they are expected to be locally identical. As with the continuation modifier structure, $\mathbb{S}_{\texttt{Helper}}$ will be reused in different oracle languages and is therefore parametric in the type of actual changes to be applied to statements and continuations.

Definition 23 ($\mathbb{S}_{\texttt{Helper}}$ return type)**.** For a given type α of actual statement-level changes and a type β of actual continuation-level changes, the result type of the $\mathbb{S}_{\texttt{Helper}}$ function is the following algebraic abstract data type:

$$\begin{array}{rcl} \mathit{step_helper_res} & ::= & \textbf{GStep}\ S_{\text{Imp}}\ S_{\text{Imp}}\ \overline{\delta\kappa} \\ & | & \textbf{SCmdStep}\ \alpha\ \overline{\delta\kappa} \\ & | & \textbf{SContStep}\ \beta\ \overline{\delta\kappa} \\ & | & \textbf{Stuck} \end{array}$$

Those four constructors represent different possible outcomes of executing the two programs according to a continuation modifier list: **GStep** $s_1\ s_2\ \overline{\delta\kappa}$ represents the result of a successful lock-step execution step leading to the pair of configurations (s_1, s_2) and a remaining continuation modifier $\overline{\delta\kappa}$, **Stuck** is returned when the two programs are both stuck, and the other cases are returned when an actual statement-level or continuation-level change is to be handled, requiring special action from the caller.

Definition 24 ($\mathbb{S}_{\texttt{Helper}}$ step helper)**.** We do not give here the full formal definition of $\mathbb{S}_{\texttt{Helper}}$ as to not clutter this manuscript with its many cases, but we will only give a partial, semi-formal definition to convey how this function works. As always, the full definition is available in the Coq development.

$$\begin{array}{l} \mathbb{S}_{\texttt{Helper}}((\textbf{skip} \cdot \kappa_1, S_1),\ (\textbf{skip} \cdot \kappa_2, S_2),\ \Delta_c(\texttt{Id}),\ \overline{\delta\kappa}) \\ \qquad = \textbf{GStep}\ (\kappa_1, S_1)\ (\kappa_2, S_2)\ \overline{\delta\kappa} \\ \mathbb{S}_{\texttt{Helper}}((c_1^1; c_1^2 \cdot \kappa_1, S_1),\ (c_2^1; c_2^2 \cdot \kappa_2, S_2),\ \Delta_c((\Delta_{\texttt{Rec}}(\alpha,\ \alpha'))),\ \overline{\delta\kappa}) \\ \qquad = \textbf{GStep}\ (c_1^1 \cdot c_1^2 \cdot \kappa_1, S_1)\ (c_2^1 \cdot c_2^2 \cdot \kappa_2, S_2)\ (\alpha :: \alpha' :: \overline{\delta\kappa}) \\ \qquad\quad \vdots \\ \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ \Delta_c((\Delta_{\texttt{Leaf}}(\alpha))),\ \overline{\delta\kappa}) = \textbf{SCmdStep}\ \alpha\ \overline{\delta\kappa} \\ \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ \Delta_\kappa(\beta),\ \overline{\delta\kappa}) = \textbf{SContStep}\ \beta\ \overline{\delta\kappa} \end{array}$$

The first two cases above illustrate the lockstep execution of the two programs when they are locally identical: $\mathbb{S}_{\texttt{Helper}}$ takes care of stepping in both programs and updating the continuation modifier structure. In the last two cases, the head term of the continuation modifier structure indicates a change that must be handled by the oracle language.

Note that the $\mathbb{S}_{\texttt{Helper}}$ is undefined whenever the continuation modifier indicates a lockstep execution but the programs do not actually perform the same reduction step. For instance, neither of the following cases is defined:

$$\begin{aligned}&\mathbb{S}_{\texttt{Helper}}((\textbf{skip} \cdot \textbf{halt}, \emptyset),\ (x = e \cdot \textbf{halt}, \emptyset),\\ &\qquad \Delta_c(\texttt{Id}),\ _)\\ &\mathbb{S}_{\texttt{Helper}}((\textbf{if}\,(x \leq 42)\ \textbf{skip else skip} \cdot \textbf{halt}, \emptyset[x \mapsto 0]),\\ &\qquad (\textbf{if}\,(x \leq 42)\ \textbf{skip else skip} \cdot \textbf{halt}, \emptyset[x \mapsto 50]),\\ &\qquad \Delta_c(\texttt{Id}),\ _)\end{aligned}$$

Since `SwapBranches` executes programs in lockstep, continuation-level changes are not needed, and the `SwapBranches` oracle language definitions thus make use of $\mathbb{S}_{\texttt{Helper}}$ with the empty type for the type β of actual continuation-level changes. The type α of actual statement-level changes is defined as a sum type with two constructors: `Negate` and `UnNegate`. While they do not actually appear in the semantics of `SwapBranches`, they are used in its invariant property and its soundness proof.

Definition 25 (`SwapBranches` language on Imp)**.** The oracle language `SwapBranches` of branches swapping is defined by the oracle language definition

$$(unit,\ S_{\text{Imp}} \times S_{\text{Imp}} \times \overline{\delta\kappa},\ \mathbb{S}_{SB},\ 1_{tt},\ 1_{tt},\ fst,\ snd,\ \mathbb{I}_{SB})$$

where:

- $\mathbb{I}_{SB}(tt,\ (s_1,\ s_2,\ km))$ holds if s_1 and s_2's continuations are indeed related by km (the details are omitted in this already lengthy description but can be found in the Coq development)
- $\mathbb{S}_{SB}(tt,\ (s_1,\ s_2,\ m \cdot km))$

$$= \begin{cases} (s_1',\ s_2',\ km') & \text{if}\ \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{GStep}\ s_1'\ s_2'\ km' \\ (s_1',\ s_2',\ km') & \text{if}\ \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SCmdStep}\ \alpha\ km' \\ & \quad \text{and}\ \mathcal{E}_{Imp}(s_1) = s_1' \\ & \quad \text{and}\ \mathcal{E}_{Imp}(s_2) = s_2' \end{cases}$$

Truth be told, the branch swapping oracle's step function $\mathbb{S}_{SB}$ could be much simpler, since it simulates both programs in lockstep. Indeed, $\mathbb{S}_{SB}$ could be defined exactly as the renaming oracle's step function $\mathbb{S}_{ren}$. Instead, we opted for storing additional information in the oracle states, thus requiring a more complex step function to maintain it. Indeed, the additional information stored in the oracle state allows for a more direct and precise definition of the invariant, as doing otherwise would have required the invariant to rely on existentially-quantified variables holding similar information.

One may wonder why we have not defined a "generic invariant" to be used with the $\mathbb{S}_{\texttt{Helper}}$ function. The reason for that is that while using $\mathbb{S}_{\texttt{Helper}}$ avoids a lot of boilerplate code, the generic part of related invariants is exceedingly simple, and

using a generic construct would only add overhead. As for invariant preservation and correctness proof, they are very much dependent on the specifics of each oracle language.

4.4 Refactoring branches with common code

Another class of syntactic changes between extensionally-equivalent programs is the class of changes consisting in deduplicating code that is ending both the **then** and **else** branches of a conditional by moving it after that conditional. This class of changes is encoded by the `RefactorCommonBranchRemainder` oracle language, the second oracle language to make use of the $\mathbb{S}_{\texttt{Helper}}$ function defined in the previous section, and the first occasionally simulating multiple steps of both programs within a single oracle step.

Example 8. One example of such a pair of programs that can be related by `RefactorCommonBranchRemainder` could be:

```
if (x < y) {
  a = x + 42;
  a = a * 2;
  result = a + 6;
} else {
  a = y;
  a = a * 2;
  result = a + 6;
}
```

```
if (x < y) {
  a = x + 42;
} else {
  a = y;
};
a = a * 2;
result = a + 6;
```

The relation between the two programs' execution traces is a bit trickier than the ones we have defined so far: indeed, instead of always simulating a single step of each program, it will occasionally perform two steps at once in both programs in order to skip locally-different execution steps and synchronize to locally-related ones. This occurs when the left configuration's head term is the **if** statement that gets refactored and the right configuration's head term is a sequence of the **if** statement and the common remainder. Simulating multiple steps of each program in a single oracle step is made possible thanks to the existentially-quantified variables n_1 and n_2 in the oracle soundness property from definition 14: $\forall g : \mathbb{G}$, $\forall s s' : \mathbb{O}$,

$$\mathbb{I}(g, s) \Rightarrow \mathbb{S}(s) = s' \Rightarrow \exists\, n_1\, n_2, \begin{cases} \pi_L(\mathbb{S}(s)) & = & \mathcal{E}^{n_1}_{\mathcal{L}_1}(\pi'_L(g), \pi_L(s)) \\ \pi_R(\mathbb{S}(s)) & = & \mathcal{E}^{n_2}_{\mathcal{L}_2}(\pi'_R(g), \pi_R(s)) \\ n_1 + n_2 > 0 \end{cases}$$

Example 9. To illustrate this, let us consider the following execution—where, for the

sake of clarity, stores are omitted and only the continuation is displayed:

$$
\begin{array}{ccccc}
\textbf{if}\,(b)\ \{c_1; c'\}\ \textbf{else}\ \{c_2; c'\} \cdot \kappa_1 & \longrightarrow & c_1; c' \cdot \kappa_1 & \longrightarrow & c_1 \cdot c' \cdot \kappa_1 \\
{\scriptstyle \pi_L}\uparrow & & & & {\scriptstyle \pi_L}\uparrow \\
\mathbb{O}_1 & & \longrightarrow & & \mathbb{O}_2 \\
{\scriptstyle \pi_R}\downarrow & & & & {\scriptstyle \pi_R}\downarrow \\
\textbf{if}\,(b)\ c_1\ \textbf{else}\ c_2; c' \cdot \kappa_2 & \longrightarrow & \textbf{if}\,(b)\ c_1\ \textbf{else}\ c_2 \cdot c' \cdot \kappa_2 & \longrightarrow & c_1 \cdot c' \cdot \kappa_2
\end{array}
$$

In this case, a single oracle step from $\mathbb{O}_1$ simulates two steps in both the left and right programs, so that the states projected from $\mathbb{O}_2$ have the same shape and lock-step execution can be resumed.

Lastly, there is a corner case that must be addressed: when performing those two steps, both programs may crash after a different number of steps when evaluating the `if` statement's condition: the left program will be immediately stuck while the right program will be stuck after a single step. Since a correct oracle is required to step both programs to their end whenever those programs are finite, the `RefactorCommonBranchRemainder` oracle language must handle this situation by returning both programs' stuck state. That means that, not only does the oracle step function not always perform exactly one step of each program, it may also perform a different number of steps in both programs whenever they crash, as in the following example diagram, where b cannot be evaluated, causing the left and right programs to be stuck after a different number of reduction steps:

$$
\begin{array}{ccccc}
\textbf{if}\,(b)\ \{c_1; c'\}\ \textbf{else}\ \{c_2; c'\} \cdot \kappa_1 & & \longrightarrow & & \not\to \\
{\scriptstyle \pi_L}\uparrow & & \nwarrow {\scriptstyle \pi_L} & & \\
\mathbb{O}_1 & & \longrightarrow & & \mathbb{O}_2 \\
{\scriptstyle \pi_R}\downarrow & & & & {\scriptstyle \pi_R}\downarrow \\
\textbf{if}\,(b)\ c_1\ \textbf{else}\ c_2; c' \cdot \kappa_2 & \longrightarrow & \textbf{if}\,(b)\ c_1\ \textbf{else}\ c_2 \cdot c' \cdot \kappa_2 & \longrightarrow & \not\to
\end{array}
$$

Like the `SwapBranches` oracle language, `RefactorCommonBranchRemainder` makes use of the $\mathbb{S}_{\texttt{Helper}}$ function with the empty type as the type β of actual continuation-level changes. The type α of actual statement-level changes is the singleton $unit$, as `RefactorCommonBranchRemainder` only handles one kind of change.

Definition 26 (`RefactorCommonBranchRemainder` language on Imp)**.** The oracle language `RefactorCommonBranchRemainder` on Imp is defined by the oracle language definition

$$(unit,\ S_{\text{Imp}} \times S_{\text{Imp}} \times \overline{\delta\kappa} + S_{\text{Imp}} \times S_{\text{Imp}},\ \mathbb{S}_{RCBR},\ 1_{tt},\ 1_{tt},\ fst,\ snd,\ \mathbb{I}_{RCBR})$$

where:

- $\mathbb{I}_{RCBR}(tt,\ (s_1,\ s_2,\ km))$ holds if s_1 and s_2's continuations are indeed related by km (the details are omitted in this already lengthy description but can be found in the Coq development)
- $\mathbb{I}_{RCBR}(tt,\ (s_1,\ s_2))$ holds if $s_1 \not\to$ and $s_2 \not\to$
- $\mathbb{S}_{RCBR}(tt,\ (s_1,\ s_2,\ m \cdot km)) =$

$$\begin{cases} (s'_1,\ s'_2,\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{GStep}\ s'_1\ s'_2\ km' \\ (s'_1,\ s'_2,\ (\Delta_c(\texttt{Id})) :: (\Delta_c(\texttt{Id})) :: km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SCmdStep}\ \alpha\ km' \\ & \quad \text{and } \mathcal{E}_{Imp}(\mathcal{E}_{Imp}(s_1)) = s'_1 \\ & \quad \text{and } \mathcal{E}_{Imp}(\mathcal{E}_{Imp}(s_2)) = s'_2 \\ (s_1,\ s'_2) & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SCmdStep}\ \alpha\ km' \\ & \quad \text{and } s_1 \not\to \\ & \quad \text{and } \mathcal{E}_{Imp}(s_2) = s'_2 \end{cases}$$

Remark. The reader may have noticed an unusual and unexplained limitation imposed by this oracle language: indeed, each branch of the original program must be a sequence of a particular shape, with the code specific to that branch on the left side of the sequence, and the code common to both branches on the right side. These restrictions on the programs' shape are common in our oracle languages as they significantly simplify their presentation. Those restrictions do not reduce the expressive power of such languages, thanks to the `SeqAssoc` oracle language presented next.

4.5 Sequence Associativity

It must be noted that in our definition of Imp, programs are trees of statements, rather than lists of statements: $(c_1; c_2); c_3$ and $c_1; (c_2; c_3)$ are not syntactically equal, but they are equivalent, as sequence in Imp is associative. While this class of differences is not directly useful to programmers, it is required nonetheless for formal reasoning. Indeed, other oracle languages, such as the `RefactorCommonBranchRemainder` oracle language defined in the previous section, will often require specific ways of parenthesizing sequences. Therefore, having oracles to transform programs that do not conform to those parenthesizing schemes is useful. The oracle language representing reparenthesizing of sequences, `SeqAssoc`, is also, with `RefactorCommonBranchRemainder`, one of the simplest that does not feature lock-step execution. Indeed, as unfolding a sequence is a small execution step on its own, two identical programs modulo sequence associativity will not necessarily execute their effectful statements after the same number of steps. For instance, the assignment statement will not be executed after the same number of steps in the two following programs: $(x = e; c_2); c_3$ and $x = e; (c_2; c_3)$. Indeed, in the first case, the assignment is performed in the third reduction step, while it is performed in the second reduction step in the second program.

To address this, the `SeqAssoc` oracles will simulate a different number of steps in the left and right programs when encountering this pattern.

$$
\begin{array}{ccccc}
(c_1; c_2); c_3 \cdot \kappa_1 & \longrightarrow & c_1; c_2 \cdot c_3 \cdot \kappa_1 & \longrightarrow & c_1 \cdot c_2 \cdot c_3 \cdot \kappa_1 \\
\uparrow \pi_L & & & \nearrow \pi_L & \\
\mathbb{O}_1 & \longrightarrow & & \mathbb{O}_2 & \\
\downarrow \pi_R & & \swarrow \pi_R & & \\
c_1; (c_2; c_3) \cdot \kappa_2 & \longrightarrow & c_1 \cdot c_2; c_3 \cdot \kappa_2 & &
\end{array}
$$

To this end, the `SeqAssoc` oracle makes use of the $\mathbb{S}_{\texttt{Helper}}$ function to keep track of the execution points where those changes have to be applied. This is a bit more complex as in the previous oracles since the program that has been "stepped once" needs to be "stepped twice" at a later point to perform the matching sequence unfolding.

$$
\begin{array}{ccccccccc}
(c_1; c_2); c_3 \cdot \kappa_1 & \to & c_1; c_2 \cdot c_3 \cdot \kappa_1 & \to & c_1 \cdot c_2 \cdot c_3 \cdot \kappa_1 & \longrightarrow \cdots \longrightarrow & & & c_2 \cdot c_3 \cdot \kappa_1 \\
\uparrow \pi_L & & & \nearrow \pi_L & & & & \nearrow \pi_L & \uparrow \pi_L \\
\mathbb{O}_1 & \longrightarrow & & \mathbb{O}_2 & \longrightarrow \cdots \longrightarrow & & \mathbb{O}_n & \to & \mathbb{O}_{n+1} \\
\downarrow \pi_R & & \swarrow \pi_R & & & \swarrow \pi_R & & & \downarrow \pi_R \\
c_1; (c_2; c_3) \cdot \kappa_2 & \to & c_1 \cdot c_2; c_3 \cdot \kappa_2 & \longrightarrow \cdots \longrightarrow & & c_2; c_3 \cdot \kappa_2 & \to & & c_2 \cdot c_3 \cdot \kappa_2
\end{array}
$$

To do so, the `SeqAssoc` oracle languages defines a type of actual statement-level changes for the "sequence unfolding" operation and the "sequence folding" operation along with two corresponding continuation-level changes:

- The statement-level change type is defined as $\alpha = \mathsf{U_{Seq}} \mid \mathsf{F_{Seq}}$. $\mathsf{U_{Seq}}$ describes the transformation from $(c_1; c_2); c_3$ to $c_1; (c_2; c_3)$ while $\mathsf{F_{Seq}}$ represents the opposite transformation.
- The continuation-level change type is defined as $\beta = \mathsf{U_L} \mid \mathsf{U_R}$. When $\mathsf{U_L}$ is on the top of the continuation modifier list, then the oracle have to interpret a single step in the left program while not stepping in the right program. Conversely, if $\mathsf{U_R}$ is on the top, it means that the oracle has to step in the right program without stepping in the left one.

To illustrate how this modifier structure is used by the `SeqAssoc` oracle language, consider this version of the previous diagram, in which oracle states are replaced with their modifier structure—abusively writing x for $\Delta_\kappa(x)$ and y for $\Delta_c(y)$ in order to make the diagram more readable:

$$
\begin{array}{ccccccccc}
(c_1; c_2); c_3 \cdot \kappa_1 & \longrightarrow & c_1; c_2 \cdot c_3 \cdot \kappa_1 & \longrightarrow & c_1 \cdot c_2 \cdot c_3 \cdot \kappa_1 & \longrightarrow \cdots \longrightarrow & & & c_2 \cdot c_3 \cdot \kappa_1 \\
\uparrow \pi_L & & & \nearrow \pi_L & & & \nearrow \pi_L & & \uparrow \pi_L \\
\Delta_{\mathsf{Leaf}}(\mathsf{U_{Seq}}) :: km' & \longrightarrow & & \mathtt{Id} :: \mathsf{U_R} :: km' & \longrightarrow \cdots \longrightarrow & \mathsf{U_R} :: km' & \to & & \mathtt{Id} :: \mathtt{Id} :: km' \\
\downarrow \pi_R & & \swarrow \pi_R & & \swarrow \pi_R & & & & \downarrow \pi_R \\
c_1; (c_2; c_3) \cdot \kappa_2 & \longrightarrow & c_1 \cdot c_2; c_3 \cdot \kappa_2 & - \cdots \to & c_2; c_3 \cdot \kappa_2 & \longrightarrow & & & c_2 \cdot c_3 \cdot \kappa_2
\end{array}
$$

Definition 27 (`SeqAssoc` language on Imp)**.** The oracle language `SeqAssoc` on Imp is defined by the oracle language definition

$$(unit,\ S_{\text{Imp}} \times S_{\text{Imp}} \times \overline{\delta\kappa},\ \mathbb{S}_{SA},\ 1_{tt},\ 1_{tt},\ fst,\ snd,\ \mathbb{I}_{SA})$$

where:

- $\mathbb{I}_{SA}(tt,\ (s_1,\ s_2,\ km))$ holds if s_1 and s_2's continuations are indeed related by km (the details are omitted in this already lengthy description but can be found in the Coq development)
- $\mathbb{S}_{SA}(tt,\ (s_1,\ s_2,\ m \cdot km)) =$

$$\begin{cases} (s_1',\ s_2',\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{GStep}\ s_1'\ s_2'\ km' \\ (s_1',\ s_2',\ \Delta_c(\texttt{Id}) :: \Delta_\kappa(\texttt{U}_\texttt{R}) :: km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SCmdStep}\ \texttt{U}_{\texttt{Seq}}\ km' \\ & \quad \text{and } \mathcal{E}_{Imp}(\mathcal{E}_{Imp}(s_1)) = s_1' \\ & \quad \text{and } \mathcal{E}_{Imp}(s_2) = s_2' \\ (s_1',\ s_2',\ \Delta_c(\texttt{Id}) :: \Delta_\kappa(\texttt{U}_\texttt{L}) :: km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SCmdStep}\ \texttt{F}_{\texttt{Seq}}\ km' \\ & \quad \text{and } \mathcal{E}_{Imp}(s_1) = s_1' \\ & \quad \text{and } \mathcal{E}_{Imp}(\mathcal{E}_{Imp}(s_2)) = s_2' \\ (s_1',\ s_2,\ \Delta_c(\texttt{Id}) :: \Delta_c(\texttt{Id}) :: km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SContStep}\ \texttt{U}_\texttt{L}\ km' \\ & \quad \text{and } \mathcal{E}_{Imp}(s_1) = s_1' \\ (s_1,\ s_2',\ \Delta_c(\texttt{Id}) :: \Delta_c(\texttt{Id}) :: km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SContStep}\ \texttt{U}_\texttt{R}\ km' \\ & \quad \text{and } \mathcal{E}_{Imp}(s_2) = s_2' \end{cases}$$

4.6 Independent assignments swapping

The oracle language `SwapAssign` for *independent assignments swapping* describes the pairs of programs obtained by swapping two adjacent independent assignments. This is yet another oracle language for extensionally-equivalent programs, simulating them mostly in lock-step except for the precise execution points where the two assignments are to be swapped. `SwapAssign` does not make use of any new technique in its definition or proofs.

Example 10. An example of programs that can be related by a `SwapAssign` oracle is the following pair of programs, in which the two assignments in the loop's body have been swapped:

```
a = 10;
x = 2;
while (0 <= a) {
  x = 42 * x;
  a = a - 1;
}
```

```
a = 10;
x = 2;
while (0 <= a) {
  a = a - 1;
  x = 42 * x;
}
```

Oracles of the `SwapAssign` oracle language behave by executing the two programs in lockstep until they reach a sequence of two independent assignments that are swapped in both programs. To do so, they rely on the $\mathbb{S}_{\texttt{Helper}}$ function with the singleton type *unit* for both the type α of actual statement-level changes and the type β of actual continuation-level changes. Indeed, there is only one way to swap two adjacent independent assignments. The statement-level modifier is meant to syntactically track the sequence containing the two assignments to swap. When that point is reached, `SwapAssign` oracles will perform a single step of both programs—unfolding the head sequence—and pushing a *continuation modifier* on top of the continuation modifier structure to indicate that the two assignments at the head of the continuations are to be swapped. The next oracle step will then simulate two steps of both the left and right programs. This is illustrated by the following diagram:

$$\begin{array}{ccccccc}
x = e; y = e' \cdot \kappa_1 & \longrightarrow & x = e \cdot y = e' \cdot \kappa_1 & \longrightarrow & y = e' \cdot \kappa_1 & \longrightarrow & \kappa_1 \\
\pi_L \uparrow & & \pi_L \uparrow & & & & \pi_L \uparrow \\
\mathbb{O}_1 & \longrightarrow & \mathbb{O}_2 & & \longrightarrow & & \mathbb{O}_3 \\
\pi_R \downarrow & & \pi_R \downarrow & & & & \pi_R \downarrow \\
y = e'; x = e \cdot \kappa_2 & \longrightarrow & y = e' \cdot x = e \cdot \kappa_2 & \longrightarrow & x = e \cdot \kappa_2 & \longrightarrow & \kappa_2
\end{array}$$

As with `RefactorCommonBranchRemainder`, left and right programs might crash in the middle of those two simulated steps, as they are performing assignments. As with `RefactorCommonBranchRemainder`, this needs to be taken care of and is handled using an extra state constructor encoding crashed executions. As an illustration, consider the following diagram describing an execution of the oracle in which the expression e cannot be evaluated:

$$\begin{array}{ccccccc}
x = e; y = e' \cdot \kappa_1 & \longrightarrow & x = e \cdot y = e' \cdot \kappa_1 & \longrightarrow & \not\to & & \\
\pi_L \uparrow & & \pi_L \uparrow & \nwarrow^{\pi_L} & & & \\
\mathbb{O}_1 & \longrightarrow & \mathbb{O}_2 & \longrightarrow & \mathbb{O}_3 & & \\
\pi_R \downarrow & & \pi_R \downarrow & & & \searrow^{\pi_R} & \\
y = e'; x = e \cdot \kappa_2 & \longrightarrow & y = e' \cdot x = e \cdot \kappa_2 & \longrightarrow & x = e \cdot \kappa_2 & \longrightarrow & \not\to
\end{array}$$

Since the assignments are swapped, e is evaluated first in the left program, and second in the right program, causing them to get stuck after a different number of steps.

Much like with `RefactorCommonBranchRemainder`, the type for `SwapAssign` dynamic states is a sum type with two constructors: one for regular states and one for crashed states.

Definition 28 (`SwapAssign` language on Imp). The oracle language `SwapAssign` on Imp is defined by the oracle language definition

$$(unit,\ S_{\text{Imp}} \times S_{\text{Imp}} \times \overline{\delta\kappa} + S_{\text{Imp}} \times S_{\text{Imp}},\ \mathbb{S}_{SwA},\ 1_{tt},\ 1_{tt},\ fst,\ snd,\ \mathbb{I}_{SwA})$$

where:

- $\mathbb{I}_{SwA}(tt,\ (s_1,\ s_2,\ km))$ holds if s_1 and s_2's continuations are indeed related by km (the details are omitted in this already lengthy description but can be found in the Coq development)
- $\mathbb{I}_{SwA}(tt,\ (s_1,\ s_2))$ holds if $s_1 \not\to$ and $s_2 \not\to$
- $\mathbb{S}_{SwA}(tt,\ (s_1,\ s_2,\ m \cdot km)) =$

$$\begin{cases} (s'_1,\ s'_2,\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{GStep}\ s'_1\ s'_2\ km' \\ (s'_1,\ s'_2,\ \Delta_\kappa(tt) :: km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SCmdStep}\ tt\ km' \\ & \quad \text{and } \mathcal{E}_{Imp}(s_1) = s'_1 \\ & \quad \text{and } \mathcal{E}_{Imp}(s_2) = s'_2 \\ (s'_1,\ s'_2,\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SContStep}\ tt\ km' \\ & \quad \text{and } \mathcal{E}_{Imp}(s_1) = s'_1 \\ & \quad \text{and } \mathcal{E}_{Imp}(s_2) = s'_2 \\ (s_1,\ s'_2) & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SContStep}\ tt\ km' \\ & \quad \text{and } s_1 \not\to \\ & \quad \text{and } \mathcal{E}_{Imp}(s_2) = s'_2 \\ (s'_1,\ s_2) & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{SContStep}\ tt\ km' \\ & \quad \text{and } \mathcal{E}_{Imp}(s_1) = s'_1 \\ & \quad \text{and } s_2 \not\to \end{cases}$$

In the above definition, lockstep execution is handled by $\mathbb{S}_{\texttt{Helper}}$. A statement-level modifier causes the oracle to perform a single step—unfolding the head sequence—and add a marker at the top of the continuation modifier structure. Such a continuation-level marker causes the oracle to perform two steps at once in both programs. If evaluating those steps fails, the oracle switches to the second oracle state constructor signifying a stuck state.

4.7 Contextual equivalence

The oracle language `AbstractEquiv` of *abstract equivalences* describes pairs of programs that are syntactically equal except for two terminating extensionally-equivalent sub-programs. While this oracle language is very powerful, it offloads the termination and equivalence proofs of the sub-programs to external proofs. Although this somewhat reduces the usefulness of such an oracle language, `AbstractEquiv` provides a first illustration of how to integrate external extensional properties within our framework.

Example 11. For instance, consider the following pair of programs, which only differ by the last two assignments in the loop's body:

```
1  y = 1; x = 0;
2  n = 0;
3  while (n < v) {
4    n = n + 1;
5    z = x + y;
6    y = z;
7    x = z - x;
8  }
```

```
1  y = 1; x = 0;
2  n = 0;
3  while (n < v) {
4    n = n + 1;
5    z = x + y;
6    x = y;
7    y = z;
8  }
```

In this case, the two loops' bodies are two extensionally-equivalent sub-programs enclosed in syntactically-equal contexts.

As with the last few oracle languages presented, `AbstractEquiv` uses the $\mathbb{S}_{\texttt{Helper}}$ function to simulate the syntactically-equal portion of the two equivalent programs in a lockstep fashion. Then, whenever it encounters the two extensionally-equivalent but syntactically distinct subprograms, it invokes the (constructive) proof of termination of both subprograms to recover the resulting store (in case of a successful execution) or a bound on the number of steps needed to reach a crashing state. In the former case, that is, when the subprograms execute without errors, the oracle effectively skips the intermediate steps thanks to the provided external proof. In the case the subprograms do crash, the oracle performs as many steps as needed to reach the crashing configuration and switches to a distinguished stuck state as with `SwapAssign` or `RefactorCommonBranchRemainder`.

More specifically, `AbstractEquiv` uses $\mathbb{S}_{\texttt{Helper}}$ with the empty type as the type β of actual continuation-level changes, and the type α of actual statement-level changes is the product type $c \times c \times (S \to S + \mathbb{N}) \times (S \to S + \mathbb{N})$ of quadruples $(c_1,\ c_2,\ f_1,\ f_2)$ such that f_1 (respectively f_2) describes the execution of c_1 (respectively f_2) in a given store, by returning the resulting store in case of successful execution, or a bound on the number of steps required to reach a stuck state otherwise. Furthermore, f_1 and f_2 must return equivalent results on equivalent input: whenever one returns a store, the other must return an equivalent store, and whenever one indicates a stuck state, the other one must also indicate a stuck state. For conciseness, those additional restrictions are omitted from the quadruples in this presentation, but they are in fact part of the data type used in the Coq development.

The following diagram illustrates how two non-crashing programs are related by an `AbstractEquiv` oracle. In order to make the diagram more readable, stores are omitted, and oracle states are replaced with their modifier structure:

$$
\begin{array}{ccccc}
x = e \cdot c_1 \cdot \kappa_1 & \longrightarrow & c_1 \cdot \kappa_1 & \longrightarrow \cdots \longrightarrow & \kappa_1 \\
\uparrow \pi_L & & \uparrow \pi_L & & \uparrow \pi_L \\
\texttt{Id} :: \Delta_c(c_1, c_2, f_1, f_2) :: km & \longrightarrow & \Delta_c(c_1, c_2, f_1, f_2) :: km & \longrightarrow & km \\
\downarrow \pi_R & & \downarrow \pi_R & & \downarrow \pi_R \\
x = e \cdot c_2 \cdot \kappa_2 & \longrightarrow & c_2 \cdot \kappa_2 & \longrightarrow \cdots \longrightarrow & \kappa_2
\end{array}
$$

In the above diagram, once the oracle reaches a state where the statement-level change (c_1, c_2, f_1, f_2) is on the top of the continuation modifier structure, it simulates multiple steps of both programs by calling f_1 and f_2.

The type of `AbstractEquiv` oracle states is actually a sum type to allow for crashed states, much like `SwapAssign` and other oracle languages.

Definition 29 (`AbstractEquiv` language on Imp)**.** The oracle language `AbstractEquiv` on Imp is defined by the oracle language definition

$$(unit,\ S_{\text{Imp}} \times S_{\text{Imp}} \times \overline{\delta\kappa} + S_{\text{Imp}} \times S_{\text{Imp}},\ \mathbb{S}_{AE},\ 1_{tt},\ 1_{tt},\ fst,\ snd,\ \mathbb{I}_{AE})$$

where:

- $\mathbb{I}_{AE}(tt,\ (s_1,\ s_2,\ km))$ holds if s_1 and s_2's continuations are indeed related by km (the details are omitted in this already lengthy description but can be found in the Coq development)
- $\mathbb{I}_{AE}(tt,\ (s_1,\ s_2))$ holds if $s_1 \not\to$ and $s_2 \not\to$
- $\mathbb{S}_{AE}(tt,\ (s_1,\ s_2,\ m \cdot km)) =$

$$\begin{cases} (s'_1,\ s'_2,\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{GStep}\ s'_1\ s'_2\ km' \\ (s'_1,\ s'_2,\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}((c_1 \cdot \kappa_1, S_1),\ (c_2 \cdot \kappa_2, S_2),\ m,\ km) = \textbf{SCmdStep}\ (c_1, c_2, f_1, f_2)\ km' \\ & \quad \text{and } s'_1 = (\kappa_1, f_1(S_1)) \\ & \quad \text{and } s'_2 = (\kappa_2, f_2(S_2)) \\ (s'_1,\ s'_2) & \text{if } \mathbb{S}_{\texttt{Helper}}((c_1 \cdot \kappa_1, S_1),\ (c_2 \cdot \kappa_2, S_2),\ m,\ km) = \textbf{SCmdStep}\ (c_1, c_2, f_1, f_2)\ km' \\ & \quad \text{and } f_1(S_1) = n \\ & \quad \text{and } f_2(S_2) = m \\ & \quad \text{and } \exists n' \leq n,\ \mathcal{E}^{n'}_{Imp}((c_1 \cdot \kappa_1, S_1)) = s'_1 \wedge s'_1 \notin dom(\mathcal{E}_{Imp}) \\ & \quad \text{and } \exists m' \leq m,\ \mathcal{E}^{m'}_{Imp}((c_2 \cdot \kappa_2, S_2)) = s'_2 \wedge s'_2 \notin dom(\mathcal{E}_{Imp}) \end{cases}$$

4.8 Crash-avoiding conditionals

As discussed multiple times in this thesis, without illustration so far, our framework enables reasoning about programs that *crash*. The `CrashFix` oracle language performs this kind of reasoning by describing pairs of programs where one is known to crash under certain conditions and the other avoids such conditions by adding a “defensive” **if** statement.

Example 12. The two following programs illustrate such a situation, with the assignment `y = 42 % x` causing the left program to crash if `v` is 0, while this assignment is guarded by a conditional statement in the right program:

<table>
<tr>
<td>

```
x = v;

y = 42 % x;


z = x + y;
```

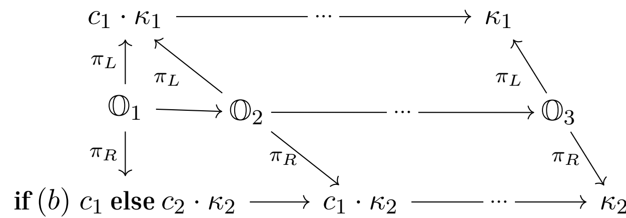
</td>
<td>

```
x = v;
if (x != 0)
  y = 42 % x;
else
  y = 0;
z = x + y;
```

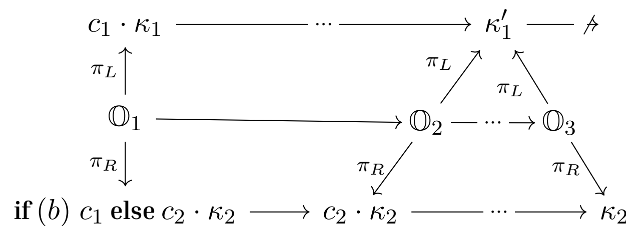
</td>
</tr>
</table>

Informally, a `CrashFix` oracle simulates the two programs in lockstep until the potentially crash-inducing statement is reached. Then, it evaluates the newly-introduced guard condition. If evaluating the guard fails, then *both* programs crash. If the guard evaluates to **true**, then, the oracle simulates a single step of the "fixed" program to unfold the conditional so that lockstep evaluation of the two programs can be resumed. Finally, if the guard evaluates to **false**, the oracle simulates a single step of the "fixed" program to unfold the conditional, and as many steps of the left program as needed for it to reach a crashed state. As the oracle step function is a Coq function, it is required to provably terminate, and therefore requires a bound on the number of steps needed to reach a stuck state to be provided by an external proof. Once a crashed state for the left program has been reached, subsequent reduction steps of the oracle will only step in the right program.

For instance, consider the following diagram—in which stores are omitted and oracle states' details are not shown—illustrating the successful execution of a pair of programs related by a `CrashFix` oracle:



The following diagram illustrates the case of the right program effectively avoiding a crash present in the left program by guarding the affected instruction c_1 under a guard condition b:



To perform as described, the `CrashFix` oracle language makes use of the $\mathbb{S}_{\texttt{Helper}}$ function with the empty type as the type β of actual continuation-level changes, and with a product type $b \times c \times c \times (S \to \mathbb{N})$ as the type α of actual statement-level

changes. Each statement-level change is a quadruple $(b,\ c_1,\ c_2, f)$ such that whenever the boolean expression b does not evaluate to true in a store S, $f(S)$ gives a bound on the number of steps required for c_1 to crash. This last requirement is omitted from the quadruple in this presentation, but is part of the statement-level change type in the Coq definition.

Definition 30 (`CrashFix` language on Imp)**.** The oracle language `CrashFix` on Imp is defined by the oracle language definition

$$(unit,\ S_{\text{Imp}} \times S_{\text{Imp}} \times \overline{\delta\kappa} + S_{\text{Imp}} \times S_{\text{Imp}},\ \mathbb{S}_{CF},\ 1_{tt},\ 1_{tt},\ fst,\ snd,\ \mathbb{I}_{CF})$$

where:

- $\mathbb{I}_{CF}(tt,\ (s_1,\ s_2,\ km))$ holds if s_1 and s_2's continuations are indeed related by km (the details are omitted in this already lengthy description but can be found in the Coq development)
- $\mathbb{I}_{CF}(tt,\ (s_1,\ s_2))$ holds if $s_1 \not\to$
- $\mathbb{S}_{CF}(tt,\ (s_1,\ s_2)) = (tt,\ (s_1,\ \mathcal{E}_{Imp}(s_2)))$
- $\mathbb{S}_{CF}(tt,\ (s_1,\ s_2,\ m \cdot km)) =$

$$\begin{cases} (s_1',\ s_2',\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{GStep}\ s_1'\ s_2'\ km' \\ (s_1,\ s_2',\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ (c_2 \cdot \kappa_2, S_2),\ m,\ km) = \textbf{SCmdStep}\ (b, c_1, c_2, f)\ km' \\ & \quad \text{and } S_2 \vdash b \Downarrow \textbf{true} \\ & \quad \text{and } \mathcal{E}_{Imp}((c_2 \cdot \kappa_2, S_2)) = s_2' \\ (s_1',\ s_2') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ (c_2 \cdot \kappa_2, S_2),\ m,\ km) = \textbf{SCmdStep}\ (b, c_1, c_2, f)\ km' \\ & \quad \text{and } S_2 \vdash b \Downarrow \textbf{false} \\ & \quad \text{and } \mathcal{E}_{Imp}((c_2 \cdot \kappa_2, S_2)) = s_2' \\ & \quad \text{and } \exists n \leq f(S_2),\ \mathcal{E}^n_{Imp}(s_1) = s_1' \wedge s_1' \notin dom(\mathcal{E}_{Imp}) \\ (s_1',\ s_2) & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ (c_2 \cdot \kappa_2, S_2),\ m,\ km) = \textbf{SCmdStep}\ (b, c_1, c_2, f)\ km' \\ & \quad \text{and } b \text{ does not evaluate in } S_2 \\ & \quad \text{and } \exists 1 \leq n \leq f(S_2),\ \mathcal{E}^n_{Imp}(s_1) = s_1' \wedge s_1' \notin dom(\mathcal{E}_{Imp}) \end{cases}$$

It is important to note that this oracle language does not guarantee that the "fall-back" statement does not crash, only that it avoids a code path that was *known* to crash. The rationale behind this decision is that proving the new code path does not crash would require traditional techniques and is not made substantially easier by a change-based approach: after all, if the previous version of the program crashed, it probably lacked any safety proof that could be reused.

4.9 Off-by-one crash fixes

A second example of oracle language specifically handling *crashing programs* is the `CrashFixWhile` oracle language, describing pairs of programs in which the left program

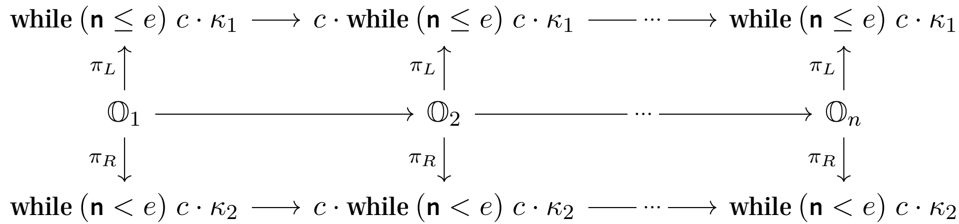
crashes during the last iteration of a **while** loop which condition is of the shape $\mathsf{n} \leq e$ and in which the right program avoids this by replacing the loop condition with $\mathsf{n} < e$.

Example 13. The following pair of programs, in which the right program is obtained by changing the condition `0 <= d` of line 4 of the left program to `0 < d`, is an example of programs related by `CrashFixWhile`:

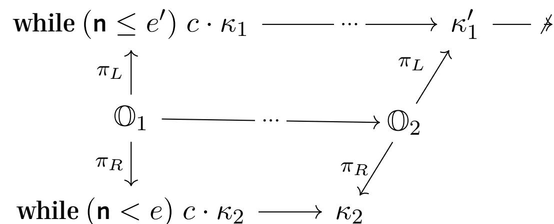
	Left	Right
1	`x = v + 1;`	`x = v + 1;`
2	`s = 0;`	`s = 0;`
3	`d = x;`	`d = x;`
4	`while (0 <= d) {`	`while (0 < d) {`
5	`  if (x % d == 0)`	`  if (x % d == 0)`
6	`    s = s + d;`	`    s = s + d;`
7	`  d = d - 1;`	`  d = d - 1;`
8	`}`	`}`

The `CrashFixWhile` oracle language is defined in a way very similar to the `CrashFix` oracle language, performing lock-step execution of both programs until the last iteration of the loop is reached, in which case the oracle performs as many steps as needed in the left program to reach the crashed state, performs a single step in the right program, and switches to an oracle state asserting that the left program has crashed.

As illustrated by the following diagram, as long as the last iteration of the loop isn't reached, the oracle simulates the two programs in lockstep:



Things get more interesting when reaching the last iteration, where $\mathsf{n} \leq e$ evaluates to **true** but $\mathsf{n} < e$ evaluates to **false**. As stated earlier, the oracle will pop the **while** from the right program's continuation and step as many times as needed in the left program to reach a crashed state. This is illustrated by the following diagram:



To perform as illustrated, `CrashFixWhile` uses $\mathbb{S}_{\texttt{Helper}}$ in a way very similar to `CrashFix`, using an empty type for the type β of actual continuation-level changes, and a product type $\mathbf{n} \times e \times c \times (S \to \mathbb{N})$ for the type α of statement-level changes, where every actual statement-level change is a quadruple $(\mathbf{n},\ e,\ c,\ f)$ such that for every store S in which e evaluates to $\mathbf{n}$, c crashes in at most $f(S)$ steps. As with previous oracles, this last condition is enforced in the Coq development but not formalized in this presentation.

Definition 31 (`CrashFixWhile` language on Imp)**.** The oracle language `CrashFixWhile` on Imp is defined by the oracle language definition

$$(unit,\ S_{\text{Imp}} \times S_{\text{Imp}} \times \overline{\delta\kappa} + S_{\text{Imp}} \times S_{\text{Imp}},\ \mathbb{S}_{CFW},\ 1_{tt},\ 1_{tt},\ fst,\ snd,\ \mathbb{I}_{CFW})$$

where:

- $\mathbb{I}_{CFW}(tt,\ (s_1,\ s_2,\ km))$ holds if s_1 and s_2's continuations are indeed related by km (the details are omitted in this already lengthy description but can be found in the Coq development)
- $\mathbb{I}_{CFW}(tt,\ (s_1,\ s_2))$ holds if $s_1 \not\to$
- $\mathbb{S}_{CFW}(tt,\ (s_1,\ s_2)) = (tt,\ (s_1,\ \mathcal{E}_{Imp}(s_2)))$
- $\mathbb{S}_{CFW}(tt,\ (s_1,\ s_2,\ m \cdot km)) =$

$$\begin{cases}
(s_1',\ s_2',\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ s_2,\ m,\ km) = \textbf{GStep}\ s_1'\ s_2'\ km' \\
(s_1',\ s_2',\ km') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ (c_2 \cdot \kappa_2, S_2),\ m,\ km) = \textbf{SCmdStep}\ (\mathbf{n}, e, c, f)\ km' \\
 & \quad \text{and } S_2 \vdash e < \mathbf{n} \Downarrow \textbf{true} \\
 & \quad \text{and } \mathcal{E}_{Imp}(s_1) = s_1' \\
 & \quad \text{and } \mathcal{E}_{Imp}((c_2 \cdot \kappa_2, S_2)) = s_2' \\
(s_1',\ s_2',\ km'') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ (c_2 \cdot \kappa_2, S_2),\ m,\ km) = \textbf{SCmdStep}\ (\mathbf{n}, e, c, f)\ km' \\
 & \quad \text{and } S_2 \vdash \mathbf{n} < e \Downarrow \textbf{true} \\
 & \quad \text{and } \mathcal{E}_{Imp}(s_1) = s_1' \\
 & \quad \text{and } \mathcal{E}_{Imp}((c_2 \cdot \kappa_2, S_2)) = s_2' \\
 & \quad \text{and where } km'' = \Delta_c(\texttt{Id}) :: \Delta_c(\Delta_{\textsf{Leaf}}(\mathbf{n}, e, c, f)) :: km' \\
(s_1',\ s_2') & \text{if } \mathbb{S}_{\texttt{Helper}}(s_1,\ (c_2 \cdot \kappa_2, S_2),\ m,\ km) = \textbf{SCmdStep}\ (\mathbf{n}, b, c, f)\ km' \\
 & \quad \text{and } S_2 \vdash e \Downarrow \mathbf{n} \\
 & \quad \text{and } \mathcal{E}_{Imp}((c_2 \cdot \kappa_2, S_2)) = s_2' \\
 & \quad \text{and } \exists n \leq f(S_2),\ \mathcal{E}^n_{Imp}(s_1) = s_1' \wedge s_1' \notin dom(\mathcal{E}_{Imp})
\end{cases}$$

Chapter 5

Encoding Relational Hoare Logic

As seen in Chapter 2, Relational Hoare Logic is a program logic to reason about relational properties on pairs of terminating programs. Encoding Relational Hoare Logic proofs with oracles demonstrates the expressivity of our framework, provides a small-step semantics interpretation of Relational Hoare Logic proofs, and forms the basis of a certified proof of soundness of the considered Hoare Logic variants.

We wrote three oracle languages to encode three different variants of Relational Hoare Logic (RHL): Minimal RHL, Core RHL and Core RHL extended with self-composition. The basic idea is to see correlating oracles as small-step interpreters of RHL proof terms.

Extended RHL extends Core RHL by allowing to replace some sub-programs by other extensionnally-equivalent sub-programs within a RHL proof. While we have no reason to doubt it can also be encoded using our framework, the insufficiently formal presentation of the original paper[3], together with the fact the added rule is inherently extensional, made us decide not to formally prove it. Intuitively, encoding it would probably involve the same kind of reasoning as in the `AbstractEquiv` oracle language, with the added complexity of having to unfold the RHL proof at the same time.

5.1 Minimal RHL

The oracle language `MinimalRHL` of programs related by a Minimal Relational Hoare Logic quadruple evaluates the two programs in lockstep and maintains at all times a formula describing the relation between the stores of the projected program configurations, in such a way that upon completion of both programs, this formula implies the postcondition of the original Minimal Relational Hoare Logic quadruple.

To do this, a `MinimalRHL` oracle maintains a list of Minimal Relational Hoare Logic proof terms of the same length as the two programs' continuation, in such a way that:

i. projecting the left (respectively right) statement of each proof term's conclusion results in the left (respectively right) program's continuation;

ii. the current formula implies the first proof term's precondition,
iii. each proof term's postcondition implies the next one's precondition
iv. the last proof term's postcondition implies the original quadruple's postcondition

A `MinimalRHL` dynamic oracle state is thus a quintuple $(s_1, s_2, S, P, \overline{\Pi})$ where s_1 and s_2 are the two related Imp program configurations as defined in Chapter 4, S is a disjoint union of those states' stores, P is a valid assertion on the store S, and $\overline{\Pi}$ is a list of Minimal RHL proof terms corresponding to the program states' continuations and consistent in the sense that the assertion P implies the precondition of the first Minimal RHL proof term, and every proof term's postcondition implies the precondition of the next proof term.

For instance, consider the following dynamic `MinimalRHL` oracle state:

$$(s_1, s_2, S, P, \frac{\vdots}{\vdash \{A\}\, c_1 \sim c_2 \,\{B\}} \cdot \frac{\vdots}{\vdash \{C\}\, c_3 \sim c_4 \,\{D\}} \cdots \frac{\vdots}{\vdash \{E\}\, c_i \sim c_j \,\{F\}})$$

This state is valid if (i) the store S is indeed the disjoint union of the two program states' store, the list of proof terms actually corresponds to (ii) the proof left continuation and (iii) the right continuation, and (iv) the list of proof terms is consistent, i.e.:

i. $S = snd(s_1) \uplus snd(s_2)$
ii. $fst(s_1) = c_1 \cdot c_3 \cdots c_i \cdot \textbf{halt}$
iii. $fst(s_2) = c_2 \cdot c_4 \cdots c_j \cdot \textbf{halt}$
iv. $P \Rightarrow A, B \Rightarrow C, \ldots$

In order to maintain the aforementioned properties on oracle states, `MinimalRHL` "interprets" the list of proof terms in the union store, effectively defining a small-step semantics on Minimal RHL proof terms. This interpretation is equivalent to interpreting the two programs and "unfolding" the proof terms according to the execution paths taken. This is done with a recursive step function ψ defined below.

Definition 32 (ψ). The ψ function takes a store, a distinguished RHL proof term, and a list of RHL proof terms, and returns a triple of a store, an assertion, and a list of RHL proof terms. It is defined inductively on its second argument by the following cases:

- When the distinguished proof tree starts with a R-Skip rule, it is discarded without modifying the store. The assertion returned is the postcondition P of the proof term, since it has been evaluated in its entirety:

$$\psi\left(S, \frac{}{\vdash \{P\}\,\textbf{skip} \sim \textbf{skip}\,\{P\}}\text{R-Skip}, \overline{\Pi}\right) = (S,\, P,\, \overline{\Pi})$$

- When the distinguished proof term is built with a R-Assign rule, the right-hand-sides of both left and right assignments are evaluated, and their result stored in the returned store. The returned assertion is the postcondition P of the proof term since it has been handled entirely:

$$\psi\left(S, \frac{}{\vdash \{P'\}\, x_1 = e_1 \;\sim\; x_2 = e_2\, \{P\}}\,\text{R-Assign}, \overline{\Pi}\right) \quad = \quad (S[x_1 \mapsto n_1][x_2 \mapsto n_2],\ P,\ \overline{\Pi}) \quad \text{where } S \vdash e_1 \Downarrow n_1 \text{ and } S \vdash e_2 \Downarrow n_2$$

- For simplicity's sake, the R-Assert rule is abusively handled like the R-Skip rule. This is abusive as it should only be defined when b_1 and b_2 both evaluate to **true** in S. Fortunately, as we only use ψ in contexts in which left and right programs both execute properly, b_1 and b_2 are guaranteed to evaluate to **true**:

$$\psi\left(S, \frac{P \Rightarrow b_1 \wedge b_2}{\vdash \{P\}\, \textbf{assert}\,(b_1) \;\sim\; \textbf{assert}\,(b_2)\, \{P\}}\,\text{R-Assert}, \overline{\Pi}\right) \quad = \quad (S,\ P,\ \overline{\Pi})$$

- The rule R-Seq is handled by "unfolding" the tree, pushing the two underlying proof trees on the stack of proof terms in a way that mirrors unfolding a sequence. The returned assertion is the original term's precondition, as that implies the new head term's precondition:

$$\psi\left(S, \frac{\Pi_1 \qquad \Pi_2}{\vdash \{P\}\, c_1^1; c_2^1 \;\sim\; c_1^2; c_2^2\, \{Q\}}\,\text{R-Seq}, \overline{\Pi}\right) \quad = \quad (S,\ P,\ \Pi_1 \cdot \Pi_2 \cdot \overline{\Pi})$$

- The R-If rule covers the two branches, but it needs to get unfolded differently depending on the execution path, in a fashion very similar to that of the small-step execution of an **if** statement. The proof of $P \Rightarrow b_1 = b_2$ ensures that both programs execute the same branch, but the proof itself is not manipulated by ψ. The returned assertion is the precondition of the subtree corresponding to the selected branch:

$$\psi\left(S, \frac{\Pi_1 \qquad \Pi_2 \qquad P \Rightarrow b_1 = b_2}{\vdash \{P\}\, \textbf{if}\,(b_1)\; c_1^1\; \textbf{else}\; c_2^1 \;\sim\; \textbf{if}\,(b_2)\; c_1^2\; \textbf{else}\; c_2^2\, \{Q\}}\,\text{R-If}, \overline{\Pi}\right) \quad = \quad (S,\ P \wedge b_1,\ \Pi_1 \cdot \overline{\Pi}) \quad \text{where } S \vdash b_1 \Downarrow \textbf{true}$$

$$\psi\left(S, \frac{\Pi_1 \qquad \Pi_2 \qquad P \Rightarrow b_1 = b_2}{\vdash \{P\}\, \textbf{if}\,(b_1)\; c_1^1\; \textbf{else}\; c_2^1 \;\sim\; \textbf{if}\,(b_2)\; c_1^2\; \textbf{else}\; c_2^2\, \{Q\}}\,\text{R-If}, \overline{\Pi}\right) \quad = \quad (S,\ P \wedge \neg b_1,\ \Pi_2 \cdot \overline{\Pi}) \quad \text{where } S \vdash b_1 \Downarrow \textbf{false}$$

- The R-While rule is handled by either duplicating the proof tree Π_1 corresponding to the loop body, in which case the returned assertion is the precondition of Π_1, or drop the proof term Π of the loop, in which case the returned assertion is the postcondition of Π:

$$\psi\left(S, \frac{\Pi_1 \qquad P \Rightarrow b_1 = b_2}{\vdash \{P\}\, \textbf{while}\, (b_1)\; c_1 \;\sim\; \textbf{while}\, (b_2)\; c_2\, \{P \wedge \neg b_1\}}\text{R-While}, \overline{\Pi}\right) \quad = \quad (S,\; P \wedge b_1,\; \Pi_1 \cdot \Pi \cdot \overline{\Pi}) \quad \text{where } S \vdash b_1 \Downarrow \textbf{true}$$

$$\psi\left(S, \frac{\Pi_1 \qquad P \Rightarrow b_1 = b_2}{\vdash \{P\}\, \textbf{while}\, (b_1)\; c_1 \;\sim\; \textbf{while}\, (b_2)\; c_2\, \{P \wedge \neg b_1\}}\text{R-While}, \overline{\Pi}\right) \quad = \quad (S,\; P \wedge \neg b_1,\; \overline{\Pi}) \quad \text{where } S \vdash b_1 \Downarrow \textbf{false}$$

- Finally, the R-Sub rule does not correspond to any small-step computation. It is thus handled by recursively calling ψ in order to unfold the proof tree until reaching a proof term corresponding to a small-step computation:

$$\psi\left(S, \frac{\Pi_1 \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\vdash \{P'\}\, c_1 \;\sim\; c_2\, \{Q'\}}\text{R-Sub}, \overline{\Pi}\right) \quad = \quad \psi(S, \Pi_1, \overline{\Pi})$$

Definition 33 (`MinimalRHL` oracle language)**.** The oracle language `MinimalRHL` on Imp is defined by the oracle language definition

$$(p,\; S_{\text{Imp}} \times S_{\text{Imp}} \times S \times P \times \overline{\Pi},\; \mathbb{S}_{MRHL},\; 1_{tt},\; 1_{tt},\; fst,\; snd,\; \mathbb{I}_{MRHL})$$

where:

- $\mathbb{I}_{MRHL}(Q,\; ((\kappa_1, S_1),\; (\kappa_2, S_2),\; S,\; P,\; \overline{\Pi}))$ holds iff:
  - P holds on S
  - $S = S_1 \uplus S_2$
  - κ_1 can be recovered by taking the left term of the conclusion of each proof term in $\overline{\Pi}$
  - κ_2 can be recovered by taking the right term of the conclusion of each proof term in $\overline{\Pi}$
  - P implies the precondition of the first Relational Hoare quadruple of $\overline{\Pi}$, each quadruple's postcondition implies the next one's precondition, and the last postcondition implies Q
- $\mathbb{S}_{MRHL}(Q,\; (s_1,\; s_2,\; S, P,\; \Pi \cdot \overline{\Pi})) = (\mathcal{E}_{Imp}(s_1),\; \mathcal{E}_{Imp}(s_2),\; S',\; P',\; \overline{\Pi'})$ when $\psi(S,\; \Pi,\; \overline{\Pi}) = (S',\; P',\; \overline{\Pi'})$

Given a Minimal RHL proof term Π with the quadruple $\vdash\; \{P\}\, c_1 \;\sim\; c_2\, \{Q\}$ as a conclusion, and a pair of initial stores S_1 and S_2 such that P holds on $S_1 \uplus S_2$, a `MinimalRHL` oracle configuration "interpreting" the proof can be constructed as $(Q,\; ((c_1 \cdot \textbf{halt}, S_1),\; (c_2 \cdot \textbf{halt}, S_2),\; S_1 \uplus S_2,\; \Pi \cdot \textbf{halt}))$. This encoding of Minimal RHL leads to a new proof of correction of the Minimal RHL rules.

Theorem 5 (Minimal RHL is sound)**.** *For every Minimal RHL proof term Π with conclusion $\vdash\; \{P\}\, c_1 \;\sim\; c_2\, \{Q\}$ and every pair of disjoint stores S_1 and S_2, if P holds on $S_1 \uplus S_2$ and c_1 and c_2 both terminate when executed in S_1 and S_2 respectively, then c_1 and c_2 successfully execute leading to stores S_1' and S_2' such that Q holds on $S_1' \uplus S_2'$.*

Proof. As always, the proof has been formalized in the Coq proof assistant. Nevertheless, we provide here a rough outline of the proof:

- By theorem 4, since c_1 and c_2 have terminating executions, so does the oracle, leading to a final oracle state $((\kappa'_1, S'_1),\ (\kappa'_2, S'_2),\ S',\ P',\ \overline{\Pi}')$
- It can then be proven the oracle cannot lead to *crashed* executions. (κ'_1, S'_1) and (κ'_2, S'_2) are thus successful final states, with **halt** as continuation, thus $\kappa'_1 = \kappa'_2 = \textbf{halt}$.
- Since the oracle invariant holds on $((\textbf{halt}, S'_1),\ (\textbf{halt}, S'_2),\ S',\ P',\ \overline{\Pi}')$, $\overline{\Pi}'$ is empty, and P' implies the global postcondition Q.
- We also have $S' = S'_1 \uplus S'_2$ and P' holds on S', so Q holds on $S'_1 \uplus S'_2$.

□

5.2 Core RHL

As described in Chapter 2, Core Relational Hoare Logic extends Minimal Relational Hoare Logic with rules corresponding to the execution of either the left or right program. Extending `MinimalRHL` to support those new rules is mostly straightforward, if a bit tedious, save for the fact that the disparity between the big-step presentation of the Core RHL rules and the small-step formalisation of our framework requires some special care.

Indeed, in the big-step presentation used for Relational Hoare Logic, empty programs are represented as **skip** statements; thus, some of the rules specific to Core RHL introduce **skip** statements that do not syntactically appear in the programs under consideration and that do not appear in the small-step reductions either. As a consequence, some rules can apply to two actual sub-statements or to only one actual sub-statement and a "fictitious" **skip** statement.

Indeed, the R-Assign-L, R-Assert-L and R-Seq-Skip-L rules reproduced below—as well as their symmetric counterparts—introduce a "fictitious" **skip** statement—highlighted—to encode an empty program—which does not correspond to any statement in the continuation:

$$\frac{\vdash \{P\}\ \textbf{skip}\ \sim\ c\,\{Q\} \qquad P \Rightarrow b}{\vdash \{P\}\ \textbf{assert}\,(b)\ \sim\ c\,\{Q\}}\ \text{R-Assert-L}$$

$$\frac{\vdash \{P\}\ \textbf{skip}\ \sim\ c\,\{Q\}}{\vdash \{P[e/x]\}\ x = e\ \sim\ c\,\{Q\}}\ \text{R-Assign-L}$$

$$\frac{\vdash \{P\}\ c_1\ \sim\ \textbf{skip}\ \{Q\} \qquad \vdash \{Q\}\ c_2\ \sim\ c\,\{R\}}{\vdash \{P\}\ c_1; c_2\ \sim\ c\,\{R\}}\ \text{R-Seq-Skip-L}$$

Therefore, any rule applicable to a **skip** statement on either side may refer to either a real or a “fictitious” **skip**. Those rules—reproduced below with a highlight on possible “fictitious” **skip** statements appearing in the conclusion, color-coded to keep track of their propagation in the proof tree—are R-Skip, R-Sub, R-Assign-L, R-Assign-R, R-Assert-L, R-Assert-R, R-If-L, R-If-R, R-Seq-Skip-L and R-Seq-Skip-R.

$$\frac{}{\vdash \{P\}\, \textbf{skip} \;\sim\; \textbf{skip}\, \{P\}}\ \text{R-Skip}$$

$$\frac{\vdash \{P\}\, c_1 \;\sim\; c_2\, \{Q\} \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\vdash \{P'\}\, c_1 \;\sim\; c_2\, \{Q'\}}\ \text{R-Sub}$$

$$\frac{\vdash \{P\}\, \textbf{skip} \;\sim\; c\, \{Q\} \qquad P \Rightarrow b}{\vdash \{P\}\, \textbf{assert}\,(b) \;\sim\; c\, \{Q\}}\ \text{R-Assert-L}$$

$$\frac{\vdash \{P\}\, \textbf{skip} \;\sim\; c\, \{Q\}}{\vdash \{P[e/x]\}\, x = e \;\sim\; c\, \{Q\}}\ \text{R-Assign-L}$$

$$\frac{\vdash \{P\}\, c_1 \;\sim\; \textbf{skip}\, \{Q\} \qquad \vdash \{Q\}\, c_2 \;\sim\; c\, \{R\}}{\vdash \{P\}\, c_1; c_2 \;\sim\; c\, \{R\}}\ \text{R-Seq-Skip-L}$$

$$\frac{\vdash \{P \wedge b\}\, c_1 \;\sim\; c\, \{Q\} \qquad \vdash \{P \wedge \neg b\}\, c_2 \;\sim\; c\, \{Q\}}{\vdash \{P\}\, \textbf{if}\,(b)\; c_1\; \textbf{else}\; c_2 \;\sim\; c\, \{Q\}}\ \text{R-If-L}$$

Let us illustrate how “fictitious” **skip** statements occur with a Core RHL proof introducing a **skip** with R-Assert-L and “consuming” it with R-Skip:

$$\frac{\dfrac{}{\vdash \{x = 1\}\, \textbf{skip} \;\sim\; \textbf{skip}\, \{x = 1\}}\ \text{R-Skip} \qquad x = 1 \Rightarrow x \leq 2}{\vdash \{x = 1\}\, \textbf{assert}\,(x \leq 2) \;\sim\; \textbf{skip}\, \{x = 1\}}\ \text{R-Assert-L}$$

One way to interpret the above proof as a series of reduction steps is to consider the R-Assert-L rule to describe exactly one reduction of the left program. With that interpretation, the R-Skip rule above then describes exactly one reduction of the right program. But there are other uses of the R-Skip rule that describe reductions in both programs at once. For this reason, a `CoreRHL` oracle needs to handle such rules differently based on whether they apply to “fictitious” **skip** statements or not.

In addition, the R-Sub and R-Skip rules may be applied to “fictitious” **skip** statements *on both sides* at the same time, in which case they do not represent any computation. Such a tree must not appear in the state of an oracle: it should be “unfolded” within a single oracle step instead, as to guarantee oracle productiveness.

This is illustrated by the following example, in which the R-Assert-L can be interpreted as a step of the left program and R-Assert-R as a step of the second program, but the R-Skip rule does not correspond to any computation:

$$\dfrac{\dfrac{\dfrac{}{\vdash \{x = 2\}\ \textbf{skip} \ \sim\ \textbf{skip}\ \{x = 2\}}\text{R-Skip} \qquad x = 2 \Rightarrow \textbf{true}}{\vdash \{x = 2\}\ \textbf{skip} \ \sim\ \textbf{assert}\,(\textbf{true})\ \{x = 2\}}\text{R-Assert-R} \qquad x = 2 \Rightarrow x \leq 2}{\vdash \{x = 2\}\ \textbf{assert}\,(x \leq 2) \ \sim\ \textbf{assert}\,(\textbf{true})\ \{x = 2\}}\text{R-Assert-L}$$

With all that being said, the general structure of the `CoreRHL` oracle is very similar to the `MinimalRHL` one except each term of the "Relational Hoare Logic continuation" can be of three types: proof terms relating two actual sub-programs, proof terms applying only to the left program and featuring only fictitious **skip** statements as the right program, and proof terms applying only to the right program and featuring only fictitious **skip** statements as the left program.

The step function is accordingly extended to support the additional rules, and also uses variants handling proof terms applying specifically to the left or right program.

The invariant is also a bit less straightforward as the "Relational Hoare Logic continuation" may not be the same length as the left and right continuations: when projecting to the left (resp. right) program, it has to be filtered to omit any proof term specific to the right (resp. left) program.

Theorem 6 (Core RHL is sound). *For every Core RHL proof term Π with conclusion $\vdash \{P\}\, c_1 \ \sim\ c_2\, \{Q\}$ and every pair of disjoint stores S_1 and S_2, if P holds on $S_1 \uplus S_2$ and c_1 and c_2 both terminate when executed in S_1 and S_2 respectively, then c_1 and c_2 successfully execute leading to stores S_1' and S_2' such that Q holds on $S_1' \uplus S_2'$.*

Proof. As with Minimal RHL, the proof has been formalized in the Coq proof assistant. The proof has the same outline as that of Minimal RHL:

- By theorem 4, since c_1 and c_2 have terminating executions, so does the oracle, leading to a final oracle state $((\kappa_1', S_1'),\ (\kappa_2', S_2'),\ S',\ P',\ \overline{\Pi}')$
- It can then be proven the oracle cannot lead to *crashed* executions. (κ_1', S_1') and (κ_2', S_2') are thus successful final states, with **halt** as continuation, thus $\kappa_1' = \kappa_2' = \textbf{halt}$.
- Since the oracle invariant holds on $((\textbf{halt}, S_1'),\ (\textbf{halt}, S_2'),\ S',\ P',\ \overline{\Pi}'),\ \overline{\Pi}'$ is empty, and P' implies the global postcondition Q.
- We also have $S' = S_1' \uplus S_2'$ and P' holds on S', so Q holds on $S_1' \uplus S_2'$.

□

5.3 Core RHL extended with self-composition

The final oracle language presented in this thesis is a simple extension to the `CoreRHL` oracle language to incorporate self-composition, that is, the ability to reduce a relational proof to a single-program proof on the sequential composition of the two related programs.

The additional rule R-SelfComp consists in proving the postcondition on the sequential composition of the two programs under scrutiny using regular Hoare Logic, thus not imposing any condition on the relation between the two programs' structure:

$$\frac{\vdash \{P\}\, c_1 ; c_2\, \{Q\}}{\vdash \{P\}\, c_1 \sim c_2\, \{Q\}}\ \text{R-SelfComp}$$

In our case, it means that when the head proof term begins with a R-SelfComp rule, the oracle will need to only predict steps of the left sub-program until it ends (if it does) before switching to the right sub-program. Intuitively, this is achievable by unfolding the head proof term until we get a Seq rule, and deconstructing it to recover a Hoare proof term for the left program and a Hoare proof term for the right program. Indeed, the only (regular) Hoare Logic rules admitting a sequence as a conclusion are the Sub and Seq rules, causing Core RHL proof terms with an application of the R-SelfComp rule at their root to always start with a finite—possibly null—number of Sub rules followed by a Seq rule.

There is a tricky case to consider, however: much like some Core RHL rules might apply to "fictitious" **skip** statements, this is also the case of the R-SelfComp rule, as illustrated:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \{\top\}\, \mathbf{skip}\, \{\top\}} \quad \dfrac{}{\vdash \{\top\}\, \mathbf{skip}\, \{\top\}}}{\vdash \{\top\}\, \mathbf{skip} ; \mathbf{skip}\, \{\top\}}\ \text{Seq}}{\vdash \{\top\}\, \mathbf{skip} \sim \mathbf{skip}\, \{\top\}}\ \text{R-SelfComp} \quad \top \Rightarrow \mathbf{true}}{\vdash \{\top\}\, \mathbf{assert}\,(\mathbf{true}) \sim \mathbf{skip}\, \{\top\}}\ \text{R-Assert-L}$$

One last trick to consider is that the R-SelfComp, much like R-Sub, R-SelfComp does not represent progress in the computation of either program and must thus be handled within a single oracle step, in addition to a productive computation.

In the end, the `CoreRHLSC` oracle language extends the `CoreRHL` oracle language by adding two new variants for regular Hoare Logic terms to the type of "Relational Hoare Logic continuations" elements, as well as the associated cases in the step function.

Chapter 6

Difference languages

The various oracle languages presented in the previous chapters serve the role of proof schema for relational program properties: each oracle language corresponds to a given class of program differences and can be instantiated as an oracle for a specific pair of programs, reducing the constructive proof of a given meaningful relation between their execution traces to a proof that the oracle's invariant holds on its initial state. Indeed, someone interested in proving a particular relation between two programs can pick an oracle language describing that relation, and write the initial oracle state relating the two program states of interest. Proving that the result oracle indeed relates the two programs—and thus that there exists such a relation between the two programs—then reduces to proving the oracle's invariant on the initial oracle state.

This process can be streamlined by the use of a *difference language*: as stated in this thesis' introduction, the main purpose of those oracle languages is to form the building blocks of a descriptive difference language that could be used by programmers to specify the intent behind their changes. Ideally, such change descriptions should be as concise and intuitively readable to programmers as ordinary informative *commit messages* while having unambiguous semantics, being machine-readable and with the ability to be mechanically *verified*.

In this chapter, we take a step back from oracles to formally state what we expect from such a difference language, defining *trace relations*, *correlations* and *difference languages*. Then we show how the previously-defined oracle languages can be used within this setting, in order to build a simple difference language for Imp.

6.1 Trace relations

As we are interested in *semantic differences*, we study the relation between the *semantics* of two programs. One traditional representation of program semantics is their execution trace. As stated in Chapter 3, small-step semantics and execution traces are indeed a formalism enabling precise reasoning about programs, including non-termination and crashing behaviors as well as any intensional property. Therefore, we are interested in relations between reduction traces.

Definition 34 (Reduction trace)**.** Given a programming language $(\mathcal{P}, \mathcal{G}, \mathcal{S}, \mathcal{I}, \mathcal{E}, \mathcal{R}, \mathcal{F})$, the reduction trace of a program $p \in \mathcal{P}$ is the potentially infinite nonempty list $\mathcal{T}(p)$ of program configurations coinductively defined by:

$$
\begin{aligned}
\mathcal{T}(p) &= \mathcal{T}^\star\ (\mathcal{I}(p)) \\
\mathcal{T}^\star\ (g, s) &= \begin{cases} (g, s) & \text{if } \mathcal{E}\ g\ s \text{ is undefined} \\ (g, s) \cdot \mathcal{T}^\star\ (g, \mathcal{E}\ g\ s) & \text{otherwise} \end{cases}
\end{aligned}
$$

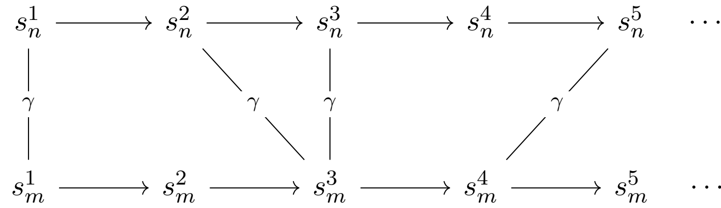
We write T the set of reduction traces of $\mathcal{L}$, i.e. $\mathrm{T} = \{\mathcal{T}(p) \mid p \in \mathcal{P}\}$.

We write $\lceil \ell \rceil$ for the first element of a nonempty potentially-infinite list ℓ.

Definition 35 (Trace relation)**.** The set R of *relations over reduction traces* is defined as the set of binary predicates over T, i.e. the inhabitants of $\mathrm{T} \to \mathrm{T} \to \mathbb{P}$.

Amongst the relations between program reduction traces, we are particularly interested in γ-correlations, trace relations defined by a pairwise relation between program configurations, as several interesting differences are naturally specified by this kind of relations.

Informally, a γ-correlation between two execution traces states that (some) states from one trace are related to (some) states of the other trace by a relation γ:



Definition 36 (γ-correlations)**.** Let γ be a relation over program configurations. A relation ρ is a γ-*correlation* if it relates pairs of traces from which two sub-sequences of configurations are pointwise related by γ i.e. $\mathcal{T}_1 \rho \mathcal{T}_2$ implies that $\mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2$, where $\overset{\gamma}{\sim}$ is coinductively defined as:

$$
\frac{|\overline{C}_1| + |\overline{C}_2| > 0 \qquad \lceil \overline{C}_1 \cdot \mathcal{T}_1 \rceil \gamma \lceil \overline{C}_2 \cdot \mathcal{T}_2 \rceil \qquad \mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2}{\overline{C}_1 \cdot \mathcal{T}_1 \overset{\gamma}{\sim} \overline{C}_2 \cdot \mathcal{T}_2}\ \text{Step}
\qquad
\frac{C_1 \gamma C_2}{C_1 \overset{\gamma}{\sim} C_2}\ \text{Stop}
$$

This definition of γ-correlation deserves several remarks.

First, for $\mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2$ to hold, the head configurations of $\mathcal{T}_1$ and $\mathcal{T}_2$ must be related by γ. If each trace contains only one configuration, the γ-correlation holds. Otherwise, there must exist two strict finite prefixes $\overline{C}_1$ and $\overline{C}_2$ of $\mathcal{T}_1$ and $\mathcal{T}_2$ that can be skipped to get to another pair of γ-correlated traces.

Second, the hypothesis $|\overline{C}_1| + |\overline{C}_2| > 0$ forces one of the prefixes to not be empty. This condition ensures that a correlation between two *finite* traces covers both traces.

It might be surprising that we do not force both prefixes not to be empty. This is actually necessary to be able to express Lamport's and Abadi's *stuttering*[25]. In a word, the stuttering mechanism allows the γ-correlation to relate a configuration of one trace to several configurations of the other trace. That way, a γ-correlation can denote a local desynchronization between the two compared traces. A γ-correlation is said to be (locally) *lockstep* if $|\overline{C}_1| = 1$ and $|\overline{C}_2| = 1$ for a given instantiation of the rule (Step).

As a final remark, one may notice that our notion of γ-correlation does not require the relation to cover infinitely many configurations from infinite traces: if $\mathcal{T}_2$ is infinite, a γ-correlation may choose not to progress in $\mathcal{T}_1$ by consuming an infinite number of configurations from $\mathcal{T}_2$ (taking $|\overline{C}_1| = 0$ and $|\overline{C}_2| > 0$ to satisfy the first hypothesis of the rule (Step)). An additional *fairness* condition could be added in the definition of γ-correlations to avoid that case: typically, we could force the hypotheses $|\overline{C}_1| > 0$ and $|\overline{C}_2| > 0$ to alternate infinitely often. Yet, as we will discuss in the conclusion of this thesis, such a fairness condition may not always be appropriate for our purpose.

6.2 Difference languages

Trace relations as defined above are the basis for difference languages' *semantics*. A difference language itself is a descriptive language in which each word is interpreted as such a trace relation.

Definition 37 (Difference language)**.** A *difference language definition* for a programming language $\mathcal{L}$ is a pair $(\mathcal{D}, [\![\bullet]\!])$ where:

- $\mathcal{D}$ is the type of difference abstract syntax trees;
- $[\![\bullet]\!]$ is the interpretation function of type $\mathcal{D} \rightarrow \mathrm{R}$

6.2.1 Oracles and difference languages

Oracles effectively describe the relation between two programs' execution traces in a constructive fashion. In fact, trace relations realized by correlating oracles fit closely with the concept of γ-correlations defined above.

This is especially apparent when comparing the shape of an oracle execution trace and its projections with the shape intended for γ-correlations:

$$
\begin{array}{ccccccccccc}
s_n^1 & \xrightarrow{\mathcal{E}_n} & s_n^2 & \xrightarrow{\mathcal{E}_n} & s_n^3 & \xrightarrow{\mathcal{E}_n} & s_n^4 & \xrightarrow{\mathcal{E}_n} & s_n^5 & \cdots \\
\uparrow \pi_L & & \pi_L \nwarrow & & \uparrow \pi_L & & & \pi_L \nearrow & & \\
o_1 & \xrightarrow{\quad\mathbb{S}\quad} & & o_2 \xrightarrow{\mathbb{S}} & o_3 & \xrightarrow{\quad\mathbb{S}\quad} & o_4 & & & \\
\downarrow \pi_R & & & \pi_R \searrow & \downarrow \pi_R & & \pi_R \swarrow & & & \\
s_m^1 & \xrightarrow[\mathcal{E}_m]{} & s_m^2 & \xrightarrow[\mathcal{E}_m]{} & s_m^3 & \xrightarrow[\mathcal{E}_m]{} & s_m^4 & \xrightarrow[\mathcal{E}_m]{} & s_m^5 & \cdots
\end{array}
$$

Definition 38 ($\gamma_{\mathcal{O},g}$). Any oracle language $\mathcal{O} = (\mathbb{G}, \mathbb{O}, \mathbb{S}, \pi'_L, \pi'_R, \pi_L, \pi_R, \mathbb{I})$ along with a fixed global environment $g \in \mathbb{G}$ defines a γ-relation where γ is defined as follows:

$$\gamma_{\mathcal{O},g} = \lambda C_1 C_2.\, \exists\, o, \mathbb{I}(o) \;\wedge\; C_1 = (\pi'_L(g),\, \pi_L(o)) \;\wedge\; C_2 = (\pi'_R(g),\, \pi_R(o))$$

In the above definition, the dynamic information contained in the oracle state is pushed behind an existentially-quantified variable, which is forced to correspond to the program configurations under consideration through the projection functions of the oracle.

The oracle then realizes the γ-relation is the sense that each oracle step yields a new oracle state which can be projected left and right to recover the next related states in the γ-correlation. γ is then guaranteed to hold on this new pair of states thanks to the invariant preservation and prediction soundness requirements of Definition 14, and the productiveness side-condition of prediction soundness matches that of the Step rule of γ-correlations.

Since the dynamic oracle state is bound by an existential quantifier, such a γ-correlation may be less precise than the actual relation defined by the oracle. Indeed, while it ensures the two reduction traces are related by a succession of oracle states on which the oracle invariant holds, it does not ensure that each of those successive oracle states is the result of performing a single oracle step from the previous oracle state.

Definition 39 ($\rho_{\mathcal{O},g,o}$). An oracle written in a language $\mathcal{O} = (\mathbb{G}, \mathbb{O}, \mathbb{S}, \pi'_L, \pi'_R, \pi_L, \pi_R, \mathbb{I})$ and with a fixed global environment $g \in \mathbb{G}$ and an initial dynamic state $o \in \mathbb{O}$ defines a trace relation as follows:

$$\rho_{\mathcal{O},g,o} = \lambda \mathcal{T}_1 \mathcal{T}_2.\, \lceil\, \mathcal{T}_1 \,\rceil = (\pi'_L(g),\, \pi_L(o)) \wedge \lceil\, \mathcal{T}_2 \,\rceil = (\pi'_R(g),\, \pi_R(o)) \wedge \mathbb{I}(g,\, o)$$

This relation simply asserts that the two initial programs states are related by a sound oracle. By the oracle's soundness, this relation implies the γ-correlation defined above.

6.2.2 A difference language on Imp

We now introduce a toy difference language ΔImp for Imp, effectively giving a syntax for each oracle language defined in our earlier chapter. A difference δ can be either a primitive difference δ_p, or a composition $\delta; \delta$ of two differences, in order to relate two programs by a succession of primitive differences (necessitating intermediary programs between each pair of primitive differences).

6.2.2.1 Syntax

			Statement contexts
$\mathbb{C}$	$::=$	$[]$	Context hole
	$\mid$	$\mathbb{C}; c$	On the left of a sequence
	$\mid$	$c; \mathbb{C}$	On the right of a sequence
	$\mid$	**if** (b) $\mathbb{C}$ **else** c	In the **then** branch
	$\mid$	**if** (b) c **else** $\mathbb{C}$	In the **else** branch
	$\mid$	**while** (b) $\mathbb{C}$	In the loop body

			Composite differences
δ	$::=$	δ_p	Primitive
	$\mid$	$\delta; \delta$	Composition
	$\mid$	$\delta \,\&\, \delta$	Superposition

		Primitive differences
δ_p	$::=$	*Syntactic refactorings*
	$\mid$	**rename** $\overline{x \leftrightarrow y}$
	$\mid$	**swap assign at** $\mathbb{C}$
	$\mid$	**swap branches at** $\mathbb{C}$
	$\mid$	**refactor common trailing code at** $\mathbb{C}$
	$\mid$	**reparenthesize sequence at** $\mathbb{C}$
		Syntactic changes
	$\mid$	**fix off-by-one at** $\mathbb{C}$
	$\mid$	**fix with defensive condition at** $\mathbb{C}$
	$\mid$	**change values of** $\overline{x}$
		Extensional changes
	$\mid$	**ensure equivalence at** $\mathbb{C}$
	$\mid$	**assume** $\{P\}$**ensure**$\{Q\}$
		Abstract changes
	$\mid$	**refactor with respect to** γ
	$\mid$	**crash fix**
	$\mid$	**optimize**

Figure 6.1: Syntax of ΔImp.

The syntax for primitive differences enumerates a collection of *builtin* differences.

This choice of primitives is *ad hoc* and there is no guarantee that they match all the software changes that can happen in a real software development. Finding a more complete set of primitives is left as future work and discussed in the conclusion of this thesis.

Nevertheless, we have defined three categories of changes, depending on their level of abstraction. The two categories named *syntactic refactorings* and *syntactic changes* contain primitive differences that can be expressed as program transformations. The next category, *extensional changes*, relates two programs by exploiting a proof that a specific relation holds between their configurations during the evaluation.

In fact, each of these primitive differences is implemented by an oracle language defined in the previous chapter.

Syntactic refactorings are program transformations that preserve the semantics of the source program for a given notion of program equivalence. They include:

1. Any renaming with respect to a bijection between their variable identifiers
2. Any swap between two consecutive assignments found at a program point characterized by a context $\mathbb{C}$, provided those assignments are syntactically independent
3. Any swap between the branches of a conditional statement provided that the condition of this statement is negated in the target program
4. Any refactoring consisting in moving after a conditional code that was present at the end of both of the conditional's branches
5. Any reparenthesizing (sequence associativity)

Syntactic changes are program transformations that modify the meaning of the source program:

1. The difference **fix off-by-one at** $\mathbb{C}$ is a program transformation that applies to a **while**-loop whose last iteration crashes and that modifies its condition to avoid this last buggy iteration.
2. The difference **fix with defensive condition at** $\mathbb{C}$ is a local program transformation that inserts a conditional statement at a source program location characterized by $\mathbb{C}$ which precedes a statement c that triggers a crash. This conditional statement makes the evaluation of the target program avoid the evaluation of the statement c.
3. The difference **change values of** $\overline{x}$ is a program transformation which modifies the assignments of any variable in $\overline{x}$ in the source program provided that this variable has no influence on the control flow. This category could be extended by any program transformation whose impact on the program behaviors can be characterized statically.

Finally, extensional changes are modifications of the source program that are not necessarily instances of a program transformation but for which a proof can be exploited to show that a specific relation holds between the two programs' traces:

1. The difference **ensure equivalence at** $\mathbb{C}$ relates two equivalent subprograms located at $\mathbb{C}$ in the two programs. This difference exploits a proof of extensional equivalence to show that the target program is a refactoring of the source program.
2. The difference **assume** $\{P\}$ **ensure** $\{Q\}$ relates two programs for which a relational Hoare logic proof validates the precondition P and the postcondition Q. As said in the introduction, by referring to the variables of both programs in P and Q, such a proof establishes that a specific relation holds between the reduction traces of two *a priori* inequivalent programs.

6.2.2.2 Semantics

Now that we have given a syntax for our difference language, we need to give it an interpretation. The interpretation of composition and superposition is straightforward:

$$\begin{array}{rcl} [\![\delta]\!] & \in & \mathrm{R} \\ [\![\delta_1;\delta_2]\!] & = & \lambda \mathcal{T}_1 \mathcal{T}_2.\ \exists \mathcal{T}_3.\ \mathcal{T}_1 [\![\delta_1]\!] \mathcal{T}_3 \wedge \mathcal{T}_3 [\![\delta_2]\!] \mathcal{T}_2 \\ [\![\delta_1 \,\&\, \delta_2]\!] & = & \lambda \mathcal{T}_1 \mathcal{T}_2.\ \mathcal{T}_1 [\![\delta_1]\!] \mathcal{T}_2 \wedge \mathcal{T}_1 [\![\delta_2]\!] \mathcal{T}_2 \end{array}$$

That is, the composition of two differences relates two traces if there exists an intermediate trace such that the first composed difference relates the left trace to that intermediate trace and the second composed difference relates the intermediate trace to the right trace. The superposition of two differences relates two traces if both of the superposed differences relate those traces.

Every primitive difference can be interpreted by the γ-correlation induced by a correlating oracle language. For instance, the interpretation for a renaming could be:

$$[\![\textbf{rename}\ \phi]\!] = \lambda \mathcal{T}_1 \mathcal{T}_2.\ \mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2 \text{ where } \gamma = \gamma_{\texttt{Renaming},\phi}$$

However, that relation is not extremely precise, as the oracle state is hidden behind an existential and thus cannot be specified. This is made apparent with other primitive differences, such as assignments swapping:

$$[\![\textbf{swap assign at}\ \mathbb{C}]\!] = \lambda \mathcal{T}_1 \mathcal{T}_2.\ \mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2 \text{ where } \gamma = \gamma_{\texttt{SwapAssign},tt}$$

Indeed, the context $\mathbb{C}$ is part of the dynamic state of an oracle and thus cannot be specified in a γ-correlation, as explained above. As a result, $\gamma_{\texttt{SwapAssign},tt}$ relates more traces than those obtained by executing programs in which the assignments at context $\mathbb{C}$ have been swapped. For this reason, we have opted for a more precise interpretation of those differences, in the sense that it relates fewer pairs of traces: this is done by using $\rho_{\mathcal{O},g,s}$, which requires constructing a dynamic oracle state from the primitive difference, instead of $\gamma_{\texttt{SwapAssign},tt}$.

Chapter 7

Coq implementation

As stated throughout this thesis, the framework of *correlating oracles* along with its instantiation on Imp, as presented in the last four chapters, has been formalized in Coq. This formalization is available online[1] as a Coq library called `OracleLanguages` and licensed under the BSD license[2].

This chapter presents that implementation, describing its structure and providing Coq snippets on how to actually use it to define oracle languages, specify and verify differences. In this chapter, prior knowledge of Coq is expected from the reader.

7.1 Code metrics and layout

The `OracleLanguages` library is written for Coq 8.6 and depends on no additional library. It is composed of meta-theoretical definitions and proofs on programming languages and oracle languages, generic oracle languages (the identity and universal oracle language), a formal definition of Imp and several oracle languages specific to that language.

The following table gives a more detailed overview of the source code, listing source files along with a short description and the chapter of the corresponding chapter of the thesis:

Table 7.1: source file layout

Path in the archive	Chapter	Description
Common.v	N/A	Common definitions, notations and helpers
ProgrammingLanguage.v	3	Meta-definition of programming languages and helper lemmas

[1] https://www.irif.fr/~thib/oracles/oracles.tar.xz
[2] https://opensource.org/licenses/BSD-3-Clause

Path in the archive	Chapter	Description
OracleLanguage.v	3	Meta-definition of oracle languages and termination proofs (Theorems 3 and 4)
ReductionTrace.v	6	Definition of reduction traces and helper lemmas
ReductionTracesRelation.v	6	Definition of trace relations
DifferenceLanguage.v	6	Meta-definition of a difference language
Oracles/*.v	3	Universal and identity oracle languages
Paper.v	7	Hands-on introduction to the framework
Imp/Syntax.v	2	Imp abstract syntax
Imp/SmallStep.v	4	Imp small-step semantics and helpers
Imp/Lang.v	4	Imp definition as a programming language
Imp/BigStep.v	2	Imp big-step semantics and proofs
Imp/Hoare.v	2	Hoare Logic rules for Imp
Imp/DisjointStates.v	N/A	Helper definitions and lemmas for reasoning about disjoint states
Imp/RHL.v	2	Relational Hoare Logic rules for Imp
Imp/Oracles/*.v	4	Oracle languages for Imp
Imp/Oracles/MinimalRHL.v	5	Oracle language encoding Minimal Relational Hoare Logic (RHL)
Imp/Oracles/CoreRHL.v	5	Oracle language encoding Core RHL
Imp/Oracles/CoreRHLSC.v	5	Oracle language encoding Core RHL with self-composition
Imp/ImpDifferenceLanguage.v	6	Difference language for Imp leveraging aforementioned oracle languages

Overall, the whole Coq development is about 5000 lines of definitions and 8000 lines of proofs. About 6000 lines of those proofs are about individual oracle languages for Imp, and are not particularly well factorized. The individual oracle languages also concentrate a fair portion of the specifications. A more precise overview of the lines

of code in the project is presented in the following table, representing the output of `coqwc`, grouped by component:

Table 7.2: `coqwc` output for `OracleLanguages`

Path in the archive	Lines of specification	Lines of proof
coq/	575	594
coq/Oracles/	79	32
coq/Imp/	588	897
coq/Imp/Hoare.v	68	38
coq/Imp/RHL.v	183	0
coq/Imp/Oracles/	2640	6077
coq/Imp/ImpDifferenceLanguage.v	222	0
coq/Paper.v	475	348
Total	4830	7986

7.2 Walk through the code

This section largely follows a similar approach to that of the `Paper.v` source file, presenting the framework and how to use it through Coq snippets.

7.2.1 Programming languages

In chapter 3, we represent a programming language as 7-uple $(\mathcal{P}, \mathcal{G}, \mathcal{S}, \mathcal{I}, \mathcal{E}, \mathcal{R}, \mathcal{F})$ equipped with additional proofs. This is encoded in our Coq library as a record with fields for both the different components of the tuple and for the accompanying proofs. The definition of that record type lies in `OracleLanguages.ProgrammingLanguage`.

```
Require Import OracleLanguages.Common.
Require Import OracleLanguages.ProgrammingLanguage.
```

In order to provide a detailed example of a programming language definition, let us consider a very simple programming language for a stack machine that can only perform additions.

```
Section PLExample.
  Inductive instr :=
  | PUSH : nat → instr (* Push a number to the stack *)
  | ADD : instr.       (* Sum the two numbers on top of the stack *)

  Definition ex_ret   : Type := nat.
  Definition ex_prog  : Type := list instr.
  Definition ex_genv  : Type := unit.
  Definition ex_state : Type := list nat * list instr.
```

```
Definition ex_step (_ : unit) (S : ex_state) :=
  match S with
  | (stack, (PUSH n)::cont) => Some (n::stack, cont)
  | (n::m::stack, ADD::cont) => Some ((n + m)::stack, cont)
  | _ =>
    None
  end.

Definition ex_state_final (S : ex_state) :=
  match S with
  | ([n], []) => Some n
  | _ => None
  end.

Definition ex_initial_state (p : ex_prog) := (tt, ([0], p)).
```

By the above definition, a program is a list of instructions, its dynamic state is a stack of natural numbers paired with a list of instructions, its global static state is empty, and the step function performs as expected. A program has properly terminated when its stack contains a single value and its continuation is empty. These definitions would suffice to define the 7-uple (`ex_state_prog`, `ex_genv`, `ex_state`, `ex_initial_state`, `ex_step`, `ex_ret`, `ex_state_final`), but we also need to prove the accompanying properties, that is, `ex_state_final` actually characterizes final states, and both `ex_genv` and `ex_state` are equipped with a decidable equality preserved by `ex_step`.

This would translate to the following Coq code (proof scripts omitted):

```
Lemma ex_state_final_no_step :
  ∀ genv s v, ex_state_final s = Some v → ex_step genv s = None.

Definition ex_genv_eq := @eq unit.
Definition ex_state_eq := @eq ex_state.

Property ex_genv_eq_dec:
  ∀ (x y : unit), { x = y } + { x ≠ y }.

Property ex_state_eq_dec:
  ∀ (x y : ex_state), { x = y } + { x ≠ y }.

Lemma ex_state_final_eq:
  ∀ S S', ex_state_eq S S' → ex_state_final S = ex_state_final S'.

Lemma ex_step_eq:
  ∀ genv₁ genv₂ S₁ S₂ S₁' S₂',
    ex_genv_eq genv₁ genv₂ →
    ex_state_eq S₁ S₂ →
    ex_step genv₁ S₁ = Some S₁' →
    ex_step genv₂ S₂ = Some S₂' →
```

```
    ex_state_eq S₁' S₂'.

Lemma ex_step_eq':
  ∀ genv₁ genv₂ S₁ S₂,
    ex_genv_eq genv₁ genv₂ →
    ex_state_eq S₁ S₂ →
    ex_step genv₁ S₁ = None ↔ ex_step genv₂ S₂ = None.
```

In this case, we just use Coq's equality as our decidable equality. We still had to prove that it is decidable, though, and that `ex_step` does preserve equality. With all of that done, we can now define the `language` record corresponding to our example.

```
Definition ex_language : language :=
  {|
    return_type        := ex_ret;
    state              := ex_state;
    genv_type          := unit;
    step               := ex_step;
    state_final        := ex_state_final;
    state_final_no_step := ex_state_final_no_step;
    prog               := ex_prog;
    initial_state      := ex_initial_state;
    genv_eq            := ex_genv_eq;
    genv_eq_refl       := @eq_refl unit;
    genv_eq_sym        := @eq_sym unit;
    genv_eq_trans      := @eq_trans unit;
    genv_eq_dec        := ex_genv_eq_dec;
    state_eq           := ex_state_eq;
    state_eq_refl      := @eq_refl ex_state;
    state_eq_sym       := @eq_sym ex_state;
    state_eq_trans     := @eq_trans ex_state;
    state_eq_dec       := ex_state_eq_dec;
    state_final_eq     := ex_state_final_eq;
    step_eq            := ex_step_eq;
    step_eq'           := ex_step_eq';
  |}.
End PLExample.
```

The record `ex_language` now holds a complete definition of our example language, and we could now write *oracle languages* for this language. However we will instead use the definitions for Imp from our library, as it will allow us to present our development.

In our library, Imp is provided as a `language` record by a functor of the module `OracleLanguages.Imp.Lang`:

```
Require Import OracleLanguages.Imp.Lang.
```

In order to get a `language` record representing Imp, we need to provide a type for the variable identifiers. In this section, this will be a simple finite inductive type:

```
Inductive varid :=
| x | y | z | n | a | b | c | d | s | count | pow.

Lemma varid_eq_dec :
  ∀ x y : varid, { x = y } + {x ≠ y}.
Proof.
  decide equality.
Defined.

Module Var <: MiniDecidableType.
  Definition t := varid.
  Definition eq_dec := varid_eq_dec.
End Var.
Module Var_as_DT := Make_UDT Var.

Module Imp := Imp.Lang.Make Var_as_DT.
```

The record of type `language` representing Imp is then `Imp.imp_language`. Before moving on to the next session, let us quickly review Imp's definition:

- Imp's global environments do not hold any information. Therefore, `genv_type imp_lang = unit`, with `υ` used as an alias for `tt`.
- Imp's dynamic environments are pairs of a continuation and a store. Hence, `state imp_language = (cont * store)`.
- An Imp program is a statement, `prog imp_language = cmd`.
- The initial state of an Imp program is the pair of a continuation made of that program and of the empty memory: `initial_state imp_language p = (υ, ([p], M.empty Z))`.
- A successful final state is a state which continuation is empty.
- Equality on Imp is defined extensionally, relying on `M.Equal` for extensional store equality.

7.2.2 Oracle languages

In Chapter 3, we represent an oracle language as a 8-uple $(\mathbb{G}, \mathbb{O}, \mathbb{S}, \pi'_L, \pi'_R, \pi_L, \pi_R, \mathbb{I})$ with some extra requirements. In our Coq library, this is encoded as a record type parameterized by two programming languages: the one left programs are written in, and the one right programs are written in. The definition of that family of record types is in `OracleLanguages.OracleLanguage`.

```
Require Import OracleLanguages.OracleLanguage.
```

In order to provide a detailed example of an oracle language, let us redefine the identity oracle language:

```
Definition id_step {L : language} (genv : genv_type L) (os : state L) :=
  step L genv os.
```

Note that, as every definition below, the step function of that oracle language is parameterized by a language record. Indeed, the identity oracle language is generic in that it may be used for any single programming language.

```
Lemma id_step_soundness {L : language}:
  ∀ (genv : genv_type L) (os os' : state L),
    True →
    id_step genv os = Some os' →
    ∃ n₁ n₂,
      opt_state_eq (nsteps (id genv) n₁ (id os))
                   (Some (id os'))

      ∧ opt_state_eq (nsteps (id genv) n₂ (id os))
                     (Some (id os'))
      ∧ n₁ + n₂ > 0.

Lemma id_step_completeness {L : language}:
  ∀ (genv : genv_type L) (os : state L),
    Logic.True →
    id_step genv os = None →
    step L genv os = None
    ∧ step L genv os = None.

Definition identity_oracle {L : language} : oracle_language L L :=
  {|
    oracle_genv  := genv_type L;
    oracle_state := state L;
    oracle_step  := id_step;
    left_state   := id;
    right_state  := id;
    left_genv    := id;
    right_genv   := id;
    invariant    := λ genv os, True;
    invariant_1  := λ genv os os' Hinv Hstep, I;
    prediction_soundness := id_step_soundness;
    prediction_completeness := id_step_completeness;
  |}.
```

The identity oracle is thus defined for any language L as an oracle language relating left programs written in L with right programs written in that same language L.

Another generic oracle language is defined in our Coq development: the universal oracle language, which relates any two programs written in any two programming languages. It is provided by `OracleLanguages.Oracles.UniversalOracle` and parameterized by two programming language records.

Finally, our library defines a range of oracle languages between pairs of programs written in Imp, as presented in chapters 4 and 5. Those oracle languages are defined in modules `OracleLanguages.Imp.Oracles.SwapAssign`,

`OracleLanguages.Imp.Oracles.Renaming` and so on. Most of them are written using the helper functions described in chapter 4 and implemented in `OracleLanguages.Imp.Oracles.OracleHelpers`.

7.2.3 Using an oracle language

In this section, we present a couple of instantiations of oracles on Imp.

7.2.3.1 Renaming

First, let us require and instantiate the appropriate module:

```
Require OracleLanguages.Imp.Oracles.Renaming.
Module Renaming :=
  OracleLanguages.Imp.Oracles.Renaming.Make Var_as_DT Imp.
```

As one may recall, our Coq definition of Imp is parameterized by the type of variable identifiers. Oracles on Imp follow the same pattern: in our case, we instantiated it with `Var_as_DT` and the `Imp` module instantiated in a previous section.

We can now move on to importing `Imp` and `Renaming` definitions and notations to define the left and right programs to be related by a renaming oracle:

```
Section RenamingExample.
  Import Imp.
  Import Imp.SmallStep.Syntax.Notations.
  Import ZArith.
  Import Renaming.

  Example P₀ (v : nat) : prog imp_language :=
    {
      x ← Int (Z.of_nat (S v));
      a ← Int 0;
      d ← Var x;
      WHILE Int 0 ≤ Var d DO {
        IF (Var x) % (Var d) == Int 0 THEN
          a ← Var a + Var d
        ELSE Skip;
        d ← Var d - Int 1
      };
      WHILE Int 0 ≤ Var a DO
        d ← Var d + Int 1
    }%AST.

  Example P₁ (v : nat) : prog imp_language :=
    {
      x ← Int (Z.of_nat (S v));
      s ← Int 0;
      d ← Var x;
```

```
    WHILE Int 0 ≤ Var d DO {
      IF (Var x) % (Var d) == Int 0 THEN
        s ← Var s + Var d
      ELSE Skip;
      d ← Var d - Int 1
    };
    WHILE Int 0 ≤ Var s DO
      d ← Var d + Int 1
  }%AST.
```

The definitions of P_0 and P_1 above are parameterized by a natural number v. Indeed, our definition only handle closed programs. Therefore, parameterizing the definition is a way to bypass this limitation and generalize input values.

Now that we have defined the two programs to be related, we also need to write the actual oracle relating them. In the case of the `Renaming` language, it is a matter of exhibiting a bijection between the variable identifiers of both programs.

```
Definition φ (var : varid) : varid :=
  match var with
  | a => s
  | s => a
  | var => var
  end.

Program Definition φ' : bijection varid varid :=
  exist _ φ _.
Next Obligation.
  exists φ. intros x y. unfold φ.
  split ; intro H ; subst ; simpl.
  - destruct x ; reflexivity.
  - destruct y ; reflexivity.
Qed.

Example renaming_ex v : oracle_genv renaming_oracle * oracle_state renaming_oracle :=
  let '(_, S₁) := initial_state imp_language (P₀ v) in
  let '(_, S₂) := initial_state imp_language (P₁ v) in
  (φ', Renaming.State S₁ S₂).
```

In the Coq snippet above, ϕ is a total function mapping a to s and vice versa, while behaving as the identity for any other variable identifier. ϕ' is the same function, wrapped with a proof that it is indeed a bijection. Finally, given an initial value v, `renaming_ex` provides the initial configuration of the renaming oracle relating P_0 and P_1.

Now, this doesn't prove `renaming_ex` is a *sound* oracle relating P_0 and P_1. In order to prove it, we have yet to prove the oracle language's invariant holds on that initial state.

```
Example renaming_ex_1 v:
  invariant renaming_oracle (fst (renaming_ex v)) (snd (renaming_ex v)).
```

```
  Proof.
    simpl. split ; auto.
    intro x. symmetry. rewrite (rename_Equal φ) ; auto.
    destruct φ' as [φ'' H] eqn:Hφ'.
    unfold φ' in Hφ'. inversion Hφ'. subst φ''. auto.
  Qed.
End RenamingExample.
```

7.2.3.2 Independent assignment swapping

Another simple oracle is the `SwapAssign` oracle. Compared to `Renaming`, it is simpler to use, but it makes use of the `OracleHelpers` module internally. For this reason, we also have to provide `OracleHelpers` when instantiating `SwapAssign`.

```
Require OracleLanguages.Imp.Oracles.OracleHelpers.
Module OracleHelpers :=
  OracleLanguages.Imp.Oracles.OracleHelpers.Make Var_as_DT Imp.
Require OracleLanguages.Imp.Oracles.SwapAssign.
Module SwapAssign :=
  OracleLanguages.Imp.Oracles.SwapAssign.Make Var_as_DT Imp OracleHelpers.
```

We can now import the relevant modules and write the programs that will be related by a `SwapAssign` oracle:

```
Section SwapAssignExample.
  Import Imp.
  Import Imp.SmallStep.Syntax.Notations.
  Import SwapAssign.
  Import OracleHelpers.

  Example swap_example : cmd :=
    {
      a ← Int 10;
      x ← Int 2;
      (WHILE Int 0 ≤ Var a DO
             Seq (x ← Int 42 * Var x)
                 (a ← (Var a - Int 1)))
    }%AST.

  Example swap_example' : cmd :=
    {
      a ← Int 10;
      x ← Int 2;
      (WHILE Int 0 ≤ Var a DO
             Seq (a ← (Var a - Int 1))
                 (x ← Int 42 * Var x))
    }%AST.
```

We can now define the initial state of the correlating oracle and prove it correct. To do so, we use the statement modifier structures from `OracleHelpers` to precisely locate the position of the assignments to be swapped.

```
Example swap_assign_or : oracle_genv swapassign_oracle * oracle_state swapassign_oracle :=
    let '(_, S₁) := initial_state imp_language swap_example in
    let '(_, S₂) := initial_state imp_language swap_example' in
    (tt, State S₁ S₂ [CmdMod
                        (RecMod Id (RecMod Id (RecMod (RecMod (LeafMod Swap) Id) Id)))]).

  Lemma swap_assign_or_1:
    invariant swapassign_oracle (fst swap_assign_or) (snd swap_assign_or).
  Proof.
    simpl. split ; auto using MFacts.Equal_refl.
    apply cmd_cont ; auto using empty_cont_mod.
  Qed.
End SwapAssignExample.
```

7.2.4 Difference language on Imp

All the oracles presented so far in this chapter have been manually instantiated. However, as described in chapter 6, the underlying oracle languages can be used in a somewhat higher-level *difference language*, giving syntax to those oracle languages and allowing to compose them.

What constitutes a difference language is formally defined in the module `OracleLanguages.DifferenceLanguage`, and a toy difference language for Imp is defined by `OracleLanguages.Imp.ImpDifferenceLanguage`.

To demonstrate how to use this difference language, we will describe the difference between the program P_0 from earlier and a program P_2 which is obtained by renaming some variable and fixing a crash:

```
Section DifferenceLanguage.
  Import Imp.
  Import Imp.SmallStep.Syntax.Notations.
  Import ZArith.

  Example P₂ (v : nat) : prog imp_language :=
  {
    x ← Int (Z.of_nat (S v));
    s ← Int 0;
    d ← Var x;
    WHILE Int 0 < Var d DO {
      IF (Var x) % (Var d) == Int 0 THEN
        s ← Var s + Var d
      ELSE Skip;
      d ← Var d - Int 1
    };
    WHILE Int 0 ≤ Var s DO
```

```
    d ← Var d + Int 1
  }%AST.
```

The difference between P_0 and P_2 is actually composed of two successive differences. We have already described the first one in the previous section, so we will just re-use the bijection ϕ' we had defined back then.

```
Import ImpDifferenceLanguage.

Example renaming := Renaming φ'.
```

The second difference is one that fixes a crash caused by a off-by-one error. Unfortunately, describing it is a bit more involved than we would hope, as one needs to prove that the loop does crash under given conditions.

```
Example P₁_loop_body := {
    IF (Var x) % (Var d) == Int 0 THEN
      s ← Var s + Var d
    ELSE
      Skip;
    d ← Var d - Int 1
  }%AST.

  Lemma P₁_loop_body_crashes:
    ∀ s,
      eval_exp s (Var d) = Some 0%Z →
      ∀ k',
        nsteps υ 3 (P₁_loop_body::k', s) = None.
```

The `CrashFixWhile` difference between P_1 and P_2 can then be defined by giving the location of the loop in the program, a bound on the number of steps needed for the loop to crash, and the proof that it does indeed crash.

```
Definition P₁_loop_ctx ν :=
    CtxSeqR (x ← Int (Z.of_nat (S ν)))%AST
   (CtxSeqR (s ← Int 0)%AST
   (CtxSeqR (d ← Var x)%AST
   (CtxSeqL CtxHole
            (Seq (WHILE Int 0 ≤ Var s DO d ← Var d + Int 1) Skip)%AST))).

  Example crash_fix ν :=
    let 'exp := (Var d) in
    let 'f := (λ _, 3) in
    CrashFixWhile (P₁_loop_ctx ν)
                  P₁_loop_body
                  exp 0 f P₁_loop_body_crashes.
```

We can now describe the difference between P_0 and P_2 as a composition—using the `Compose` constructor—of the two previously-defined primitive differences—using the `Compare` constructor:

```
Example δ v : ImpDifferenceLanguage.t :=
  Compose (Compare renaming) (Compare (crash_fix v)).
```

Once defined, we yet have to prove that the difference δ is correct. Unlike previous proof scripts, this one is displayed in its entirety to give a sense of how involved the proof is, even though it is not expected from the reader to understand them completely. Basically, `sound_program_difference` asserts that the interpretation of the given difference ($\delta\ v$) is indeed a relation between the execution traces of the left ($\mathsf{P}_0\ v$) and right ($\mathsf{P}_2\ v$) programs.

```
Import OracleLanguages.DifferenceLanguage.
Import OracleLanguages.ReductionTrace.
Import OracleLanguages.ReductionTracesRelation.

Lemma δ_sound:
  ∀ v, sound_program_difference _ _ imp_difference_language (δ v) (P₀ v) (P₂ v).
Proof.
  intro v. simpl.
  (* We first have to provide the intermediate execution trace: *)
  unfold compose_trace_relations_externally.
  exists (program_trace imp_language (P₁ v)).
  (* We then have to prove the two differences separately: *)
  split ; auto.
  + exists (P₀ v), (P₁ v).
    simpl. unfold oracle_from_primitive_diff. simpl.
    match goal with |- ∃ _, Some ?x = _ ∧ _ => exists x end.
    split ; auto. simpl.
    unfold OracleLanguage.oracle_correlation_relation.
    simpl. unfold program_trace. simpl.
    repeat split ; try apply traceFrom_trace.
  + exists (P₁ v), (P₂ v).
    simpl. unfold oracle_from_primitive_diff. simpl.
    rewrite beq_cmd_reflx. simpl.
    match goal with |- ∃ _, Some ?x = _ ∧ _ => exists x end.
    split ; auto. simpl.
    unfold OracleLanguage.oracle_correlation_relation.
    simpl. unfold program_trace. simpl.
    repeat split ; try apply traceFrom_trace.
    apply CrashFixWhile.cmd_cont ; auto using CrashFixWhile.km_invariant.
    do 4 (apply CrashFixWhile.mod_rec_seq ; auto using CrashFixWhile.mod_id).
    apply CrashFixWhile.mod_special'.
Qed.
```

Since we are *composing* two differences, we have to provide the intermediate execution trace. That intermediate execution trace is that of $\mathsf{P}_1\ v$, the program obtained by renaming $\mathsf{P}_0\ v$. We then have to separately prove that the first difference relates the execution traces of $\mathsf{P}_0\ v$ and $\mathsf{P}_1\ v$ and that the second difference relates $\mathsf{P}_1\ v$ and $\mathsf{P}_2\ v$. Each one of this proof then follow the same pattern, mainly relying on the oracle's soundness.

Finally, we can *superpose* this difference with a more abstract one that is not backed by an oracle: let us state, in addition to the precisely-defined difference above, that the difference is between a program that *does* crash and one that *does not*. This last assertion is stronger than what `CrashFixWhile` provides, as `CrashFixWhile` merely states that some particular crash has been avoided, not that there is no crash whatsoever.

```
Example δ' v : ImpDifferenceLanguage.t :=
  Superpose
    (Compare FixCrash)
    (Compose
       (Compare renaming)
       (Compare (crash_fix v))).
```

To prove this difference correct, we have to prove that $\mathtt{P}_0\ v$ does crash and $\mathtt{P}_2\ v$ does not. Unsurprisingly, this is much more involved than the previous proofs, as the `FixCrash` difference is more abstract and is not backed by an oracle language providing proof principles. In the following Coq code snippet, we omit the—quite long—proofs of `P`$_2$`_does_not_crash` and `P`$_0$`_crashes`, but not that of δ'`_sound`, which simply leverage those two proofs and the previous proof on δ.

```
  Lemma P₀_crashes:
    ∀ v, ∃ n, nsteps_crashing_trace n (program_trace imp_language (P₀ v)).

  Lemma P₂_does_not_crash:
    ∀ v, ¬ (∃ n, nsteps_crashing_trace n (program_trace imp_language (P₂ v))).

  Lemma δ'_sound:
    ∀ v, sound_program_difference _ _ imp_difference_language (δ' v) (P₀ v) (P₂ v).
  Proof.
    intro v. simpl. split.
  - destruct (P₀_crashes v) as [n₁ H₁].
    exists n₁.
    unfold program_trace.
    repeat split ; try apply traceFrom_trace ; auto.
    apply (P₂_does_not_crash v).
  - apply δ_sound.
  Qed.
End DifferenceLanguage.
```

Chapter 8

Related work

The formal study of equivalence between programs dates back at least to the sixties[19], while textual comparison of programs' source code dates at least as far as the seventies with the `diff` algorithm[18]. However, to the best of our knowledge, the earliest study of semantic differences between programs only dates back to 1990 and is due to Susan Horwitz[17] with an article in which she describes a static analysis technique to compare two programs. This comparison is performed on a structured intermediate representation—called Program Representation Graph—of those two programs. Our first contribution, described in Chapter 1, descends from this line of work, defining automatic static analysis which computes an over-approximations of the semantic differences of two programs.

However, for the most part, our thesis is focused on formally describing, reasoning about, and proving program semantic differences rather than automatically inferring them. Such formal comparisons of program semantics and reasoning about relational properties are the subject of various previous works, such as the formal framework of refinement mappings[25] allowing to prove that a program implements a given higher-level specification by translating states of the lower-level program to states of the higher-level specification, the formal proof system of relational Hoare logic[7] adapting Hoare Logic to relational properties, and that of product programs[4, 2, 3] reducing proofs of relational properties to proofs on single programs. This last line of work is a major source of inspiration for our framework of *correlating oracles*, which we will compare more closely to the aforementioned works in the following sections.

Another close topic is that of tools allowing to describe or automate refactoring and program evolution. Such tools include RedBaron[35], an AST-level code transformation engine for Python, and Coccinelle[31, 30], a program matching and transformation tool that defines *semantic patches*, source-level patches generalizing standard patches by abstracting details so that they can be applied to perform program transformations across a large code base, for instance transforming each call site when a function's interface changes. As we will see in the next section, though, *semantic patches* have a somewhat misleading name as they do not actually formally describe the semantic implications of those changes. Various Integrated Development Environments also

include automated refactoring tools, but to the best of our knowledge, none of them is mechanically certified, save for a renaming refactoring tool[10] based on CompCert[26]. Furthermore, most of those refactoring tools have been shown[36] to actually contain bugs.

All the works mentioned above are focused on comparing source-level changes, but there are also attempts at finding semantic differences between binary programs, such as the work of Gao et al.[12], matching the control-flow and call graphs of two binary programs by using a graph isomorphism algorithm and resorting to symbolic execution and automatic theorem proving to detect functionally-equivalent blocks of code.

8.1 Coccinelle and semantic patches

One line of work that frequently comes up when discussing semantic differences between programs is that of Coccinelle[31, 30]'s *semantic patches*. Unlike what that name may suggest, though, this line of work is not directly related to our framework of *correlating oracles* or our *difference languages*. Indeed, Coccinelle's *semantic patches* are a generalization of textual differences, expressed on abstract syntax trees rather than lines of text. In that sense, *semantic patches* are indeed more focused on the semantics of a change than regular textual diffs, as they abstract away irrelevant details such as some variations in coding style, irrelevant code—thanks to an ellipsis construct—or choice of variable names—thanks to the use of *metavariables*. This is particularly useful to describe *collateral evolution*, that is, changes made to a codebase to accommodate some changes in the interface of a library. Furthermore, such generic patches can be inferred[1] from collections of standard *patches*. Despite their name, though, *semantic patches* do not formally characterize the semantics of the changes they describe.

For those reasons, Coccinelle's semantic patches cannot be directly compared to our work, as our work is focused on *formal semantics* while Coccinelle is focused on matching and transforming abstract syntax trees. That being said, the work presented throughout this thesis is itself split into the semantics-oriented framework of *correlating oracles* and descriptive *difference languages*. It might be possible to define a *difference language* based on Coccinelle's semantic patches, or at the very least draw inspiration from it, but it would require a formal semantic interpretation of those patches. Doing so would enable certifying and verifying the correctness of such patches applied to a given program.

8.2 Correlating programs and speculative correlation

As repeatedly stated throughout this thesis, one major inspiration behind our work is that of Eran Yahav and Nimrod Partush[32], describing a two-stages static analysis of pairs of programs consisting in first building a *correlating program* by statically

interleaving instructions of two programs and then using abstract interpretation on the result to infer relational properties.

We discussed this paper in Chapter 1, exhibiting an issue making the interleaving of instructions unsound in some cases, and proposed an alternative algorithm for generating sound correlating programs.

In the meantime, Eran Yahav and Nimrod Partush moved away from correlating programs and the static scheduling they define: indeed, they published a follow-up paper[33] in which the abstract interpretation is not performed on a *correlating program* but directly on the pair of source programs. This analysis is still performed on an interleaving of both programs' instructions, but that interleaving is chosen dynamically by repeatedly comparing several interleavings and choosing the "most promising one" according to heuristics based on the state of the abstract interpretation.

It is difficult to directly compare this line of work with our framework of *correlating oracles*, as our focus is on specifying and verifying precise differences while Eran Yahav and Nimrod Partush's approach is to automatically infer relational properties. Nevertheless, it should be possible for the abstract interpretation tool to output its results as an oracle "certifying" the inferred properties by replaying the interleaving choices made during the speculative abstract interpretation and linking concrete executions to the abstract results.

8.3 Other automatic inference of program differences

Suzette Person et al. designed *Differential Symbolic Execution*[34] to automatically infer the semantic differences between programs using *Symbolic Execution*, generating function summaries and comparing them. Their technique exploits the similarity between programs by replacing common chunks of code with uninterpreted functions, thus avoiding costly symbolic execution of those chunks of code whenever they are executed in equivalent contexts. In some cases, it also allows bypassing some limitations of Symbolic Execution, for instance if those chunks of code contain loops which bounds are not statically known. Differential Symbolic Execution's results can be refined on demand if those abstracted code chunks are called in non-equivalent contexts.

Shuvendu K. Lahiri et al. designed `SymDiff`[22], a tool that pairs supposedly-equivalent functions of two programs and uses Satisfiability Modulo Theories (SMT) solvers to attempt to prove them equivalent. Failing that, it produces counter-examples of execution paths leading to different results.

8.4 Product programs

The other major inspiration to our work, that we have covered in Chapter 2, is Barthe et al.'s work on Relational Hoare Logic and *Product Programs*[3]. Product programs are similar to correlating programs in that they statically interleave instructions of a pair of programs to reduce relational analysis to non-relational analysis. However,

they enjoy a very syntactic formal definition strongly related to the rules of Relational Hoare Logic, rules which are described in the original article as well as in Chapter 2. In particular, product programs, like correlating programs, define a static scheduling, but that scheduling is directed by the relational semantic property to be proved.

Chapter 5 is dedicated to encoding Relational Hoare Logic in our framework of *correlating oracles*. In that chapter, we described how to encode Minimal Relational Hoare Logic, Core Relational Hoare Logic and Core Relational Hoare Logic extended with self-composition in our framework, proving the soundness of those logics through proofs on the oracle languages encoding them. However, we have not described how *product programs* relate to *correlating oracles*. Indeed, *product programs* cannot be directly encoded in our framework. Intuitively, the reason for that is that product programs may still crash without executing both programs to completion whenever one of the underlying program crashes or one of the assumption is broken. In other words, product programs can only be seen as valid oracles if they do not crash. It would surely be possible to encode product programs in our framework under that assumption, but we decided against due to the amount of work it would require and due to the fact that it would be almost functionally equivalent to the oracles encoding Relational Hoare Logic itself.

In other works[5, 6], Barthe et al. have moved to reasoning about pairs of probabilistic programs. To do so, they designed probabilistic Relational Hoare Logic in order to reason on *distributions* rather than memories. This is particularly useful to prove security properties on cryptographic code.

Unfortunately, our framework was not designed with probabilistic programs in mind. Furthermore, it is not completely obvious to us how to handle the probabilistic case. It seems reasonable for a correlating oracle between probabilistic programs to be probabilistic itself, but we also need ways to reason about the distributions.

8.5 Bisimulation

In the field of *concurrency*, that is, the study of programs containing parts that can be executed independently in different orders, equivalent programs are often characterized by the notion of bisimulation relations. A bisimulation is a binary relation between *labelled state transition systems* (LTS), relating systems that have identical observable behavior. A LTS is a system made of a set of states, a set of labels, and a set of labelled transitions from a state to another.

More formally, a labelled transition system is defined by a triple $(S, \Sigma, \rightarrow)$, where S is the state of states, Σ is the set of labels, and $\rightarrow$ the set of labelled transitions, that is, a subset of $S \times \Sigma \times S$. A transition from a state s to a state s' with label α is written $s \xrightarrow{\alpha} s'$.

Comparing bisimulations to our work is especially hard as our setting does not fit that of concurrent programming. Indeed, the programming languages that can currently be described in our framework are *deterministic* and do not define *observable transitions*—program transitions are between two states and are not labelled.

Nonetheless, we could imagine encoding Labelled state Transition Systems within our framework.

A possible encoding of *deterministic* LTS would involve defining program configurations as pairs $(s,\ \rho) \in S \times \Sigma^\omega$ of an LTS state s and a potentially-infinite list ρ of upcoming transition labels. The transition function $\mathcal{E}$ of such programs would consume a label from the potentially-infinite list and select a transition accordingly, that is:

$$\mathcal{E}(_,\ (s, \alpha \cdot \rho)) = (s',\ \rho) \Leftrightarrow s \xrightarrow{\alpha} s'$$

Finitely branching nondeterminism could then be recovered by using a powerset construction, with each program configuration being a pair $(\overline{s},\ \rho) \in 2^S \times \Sigma^\omega$ of a set of LTS states $\overline{s}$ and a potentially-infinite list ρ of upcoming transition labels. The transition function $\mathcal{E}$ would then be defined such that:

$$\mathcal{E}(_,\ (\overline{s},\ \alpha \cdot \rho)) = (\{q \mid p \in \overline{s} \wedge p \xrightarrow{\alpha} q\},\ \rho)$$

In that setting, an oracle encoding bisimulations would execute left and right programs in lockstep, maintaining a single list of upcoming labels as well as a pair of LTS states—or, if using the powerset construction, a set of pairs of LTS states. That is, the type of oracle states would be $S \times S \times \Sigma^\omega$—or, if using the powerset construction, $2^S \times 2^S \times \Sigma^\omega$. The oracle's invariant would simply assert that every pair of left and right states are bisimilar. In this sense, our framework is expressive enough to express bisimulations, but the encoding relies on the very definition of bisimularity and does not provide any direct insight.

In [9], the notion of bisimilarity is generalized by quantifying the difference between Metric Labelled Transition Systems as a non-negative real number, such that the distance between two states is zero if and only if those two states are bisimilar. We cannot think of any direct relation between their approach and our own, as their approach is *quantitative* whereas ours is *qualitative*.

8.6 Refinement mappings

Another way of describing semantic relations between close programs is by projecting or mapping the states of a first program to states of a second program[23, 24]. More precisely, Abadi and Lamport[25] define a *refinement mapping* as a function transforming a program's states into another program's such that every behavior—that is, possible sequence of states—of the first program translates to a behavior of the second program. This can be an efficient technique to prove that a low-level program *implements* a higher-level program, as the existence of a refinement mapping which preserves external—or observable–states across the translation suffices to prove inclusion of observable behaviour.

However, this technique may not be directly applicable to some pairs of programs, as the lower-level and higher-level programs may have “incompatible” internal states,

with the higher-level specification holding more information, or performing its computations in a more granular way. To address this limitation, Abadi and Lamport[25] propose adding auxiliary variables—known as *history variables* and *prophecy variables*—to the low-level program so that a refinement mapping can be found. They then state sufficient properties to the existence of a refinement mapping between a low-level program extended with auxiliary variables and a high-level program.

This work is a bit difficult to compare to our framework of correlating oracles for a few reasons: on the one hand, refinement mappings deal with non-deterministic programs whereas our framework only handles deterministic programs—we will elaborate on this limitation in the conclusion of this thesis. On the other hand, the framework of refinement mappings only deal with behavior inclusions while correlating oracles may describe all sorts of relations, including between programs with different behaviors. Another significant difference between both frameworks is the fact that refinement mappings separate program states in *external* and *internal* program states, a distinction that our framework does not do. This, however, can be encoded in the oracle's invariant by explicitly partitioning the variables into *external* and *internal* variables.

With these considerations out of the way, a refinement mapping—without resorting to auxiliary variables—between two *deterministic* programs can probably be encoded as an oracle in the following manner: a single step of the oracle would perform a single step of the low-level program and the second program's state would be retrieved by applying the refinement mapping to the low-level program's state. It may cause stutter in the second program, but that is allowed in our framework.

However, as stated earlier, there doesn't always exist a direct mapping from a low-level program to a higher-level program. Abadi and Lamport deal with this limitation by adding history and prophecy variables to the low-level program, which results in another program with the same external behavior but a larger state space. Encoding this as an oracle between the unmodified low-level program and the higher-level program can then be performed by stepping in the annotated low-level program in the same way as described earlier, and simply erasing the history and prophecy variables when projecting back to the unmodified low-level program.

Conclusion

Contributions

Throughout this thesis, we studied the semantics of program changes from different angles. We first approached the problem of automatically inferring simple relational properties from a pair of programs. In doing so, we identified a flaw in a previous publication by Eran Yahav and Nimrod Partush[32] and proposed an alternative, mechanically-checked algorithm addressing this issue.

We then shifted our focus to specifying and verifying possibly elaborate relational properties, which lead to our main contribution: a formal framework to precisely characterize the semantics of program changes. Drawing inspiration from various publications[32, 3], this framework of *correlating oracles* gives *patches* a *small-step semantics* which enables constructively building a relation between the two program's traces. This framework is meant to be used as a building block for writing descriptive and verifiable difference languages for programmers. We formalized this framework in the Coq proof assistant.

We also wrote a set of fully-formalized *correlating oracle languages* for the toy imperative language Imp covering various classes of semantic differences. Those difference classes range from minor syntactic changes preserving the overall semantics of the programs—like swapping two independent assignments—to fixing crashes by adding defensive conditions.

In order to evaluate the expressivity of our framework, we successfuly encoded several variants of Relational Hoare Logic as oracle languages. In doing so, we gave small-step semantics to Relational Hoare Logic proofs and provided new mechanically-checked proofs of soundness of those Relational Hoare Logic variants.

Finally, we designed a toy difference language for Imp illustrating how to make use of the aforementioned oracles.

Limitations

Completeness of a difference language

A first limitation of our *difference language* approach is that such difference languages are not typically complete. Indeed, only differences fitting already-identified difference classes implemented into that difference language can be represented this way. For

instance, the toy difference language for Imp presented in Chapter 6 allows precisely describing some changes preserving the program's control-flow, but only if those changes conform to the very strict syntactic criterion given in Chapter_4.

This limitation can be worked around by providing more generic difference primitives. However, such difference primitives are usually either less precise, or require more input from the programmer. This is for instance the case of the `AbstractEquiv` oracle language—and the associated **ensure equivalence at** $\mathbb{C}$ primitive of the difference language—defined in Chapter 4, which allows relating two equivalent programs made of syntactically-different subprograms embedded in syntactically-equal contexts, provided that they come with a proof that the two sub-programs are extensionally equivalent.

Scaling to realistic languages

A related limitation is the possibly prohibitive amount of upfront work needed to *design* a difference language. Indeed, designing a few oracle languages for Imp—and proving them sound—has proven to be a significant undertaking.

Designing such a difference language involved identifying interesting classes of program modifications, characterizing their semantics through an oracle language, and designing a proper syntax for use by programmers.

Unfortunately, each of these steps is bound to get significantly more challenging and time-consuming when considering realistic programming languages. Indeed, categorizing and identifying common and useful classes of program differences is in itself a substantial task, and the number of such classes is likely to depend on the complexity of the programming language. Furthermore, the amount of work needed to define and prove each oracle language is heavily dependent on the number of reduction rules of the programming language, with in places a number of cases that is quadratic in the number of reduction rules, as oracles relate pairs of programs.

For our approach to scale to real-world usages, more efficient ways of writing oracle languages are needed. Our experience designing oracle languages for Imp suggests that some common patterns could be factored out, as we have done to some extent with the helper function $\mathbb{S}_{\texttt{Helper}}$ defined in Chapter 4. Continuing on this direction, it might be possible to write useful "oracle language combinators", allowing to refactor more common patterns in oracle languages.

Non-determinism

Our framework only considers languages with *deterministic* small-step semantics. However, many real-world languages have *non-deterministic* semantics, and are thus not directly supported by our framework.

While the reasons for this restriction are mostly historic, there are a few fundamental questions worth considering when dealing with non-deterministic correlating oracles. Indeed, should left and right projections of an oracle cover every possible left

and right program behaviors? Should it cover every possible behavior on one side and only a subset of the other's? Should it cover *every pair* of program behaviors?

The answer to those questions is probably going to depend on the particular use case. For instance, if we are interested in showing that two programs have the same possible outcomes, an oracle should cover every possible left and right program behaviors. However, if we are interested in proving that a program is an optimized version of an original program, the oracle should probably cover every possible behavior of the optimized program but not necessarily those of the original one.

Since different answers are possible for those questions, more work is needed in order to figure out which of the above combinations make sense and how they can be integrated in our framework.

That being said, bounded non-determinism can be encoded within our framework using a powerset construction, such as described in Chapter 8 when comparing our work to the notion of bisimulations.

Other perspectives

Automatic inference

As stated repeatedly throughout this manuscript, the issue of automatically inferring differences has been set aside when designing correlating oracles. Therefore, they may not be well-suited for automatically analyzing program differences without human intervention.

Inferring differences between two close programs may be possible, though. In particular, *Integrated Development Environments* could use heuristics to figure out which changes a programmer is performing as they type. Furthermore, such *Integrated Development Environments* often include refactoring tools, which could be extended to produce correlating oracles certifying the refactoring operations performed.

Version Control Systems

Another possible use-case for semantic patches such as those proposed in this manuscript is merging different branches in a version control system. Indeed, a common operation in version control systems is *merging* two different changes—for instance, the addition of two different features—back into a single program. The *merge* operation is usually done on a textual level and will, if possible, apply both changes to the common ancestor. In case that isn't possible, the programmer is requested to manually *resolve merge conflicts*, that is, manually edit the portions of source code modified by both changes.

However, even when two different changes do not modify the same lines, and are thus automatically *merged* by the version control system, those changes may have deep semantic consequences, leading to inconsistent semantics.

Therefore, there is an opportunity for *semantic merges*, and having semantic description of program changes is a first step towards handling them.

Bibliography

[1] Jesper Andersen and Julia L Lawall. "Generic patch inference". In: *Automated software engineering* 17.2 (2010), pp. 119–148.

[2] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Beyond 2-safety: Asymmetric product programs for relational program verification". In: *International Symposium on Logical Foundations of Computer Science*. Springer. 2013, pp. 29–43.

[3] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Product programs and relational program logics". In: *J. Log. Algebr. Meth. Program.* 85.5 (2016), pp. 847–859. doi: 10.1016/j.jlamp.2016.05.004. url: http://dx.doi.org/10.1016/j.jlamp.2016.05.004.

[4] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Relational verification using product programs". In: *International Symposium on Formal Methods*. Springer. 2011, pp. 200–214.

[5] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. "Formal certification of code-based cryptographic proofs". In: *ACM SIGPLAN Notices* 44.1 (2009), pp. 90–101.

[6] Gilles Barthe et al. "Coupling proofs are probabilistic product programs". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. 2017, pp. 161–174.

[7] Nick Benton. "Simple Relational Correctness Proofs for Static Analyses and Program Transformations (Revised, Long Version)". In: (2005).

[8] D. Binkley. "Using semantic differencing to reduce the cost of regression testing". In: *Software Maintenance, 1992. Proceerdings., Conference on*. Nov. 1992, pp. 41–50. doi: 10.1109/ICSM.1992.242560.

[9] Franck van Breugel. "A behavioural pseudometric for metric labelled transition systems". In: *International Conference on Concurrency Theory*. Springer. 2005, pp. 141–155.

[10] Julien Cohen. "Renaming Global Variables in C Mechanically Proved Correct". In: *Proceedings of the Fourth International Workshop on Verification and Program Transformation, VPT@ETAPS 2016, Eindhoven, The Netherlands, 2nd April 2016.* 2016, pp. 50–64. doi: 10.4204/EPTCS.216.3. url: https://doi.org/10.4204/EPTCS.216.3.

[11] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1977, pp. 238–252.

[12] Debin Gao, Michael K. Reiter, and Dawn Song. "BinHunt: Automatically Finding Semantic Differences in Binary Programs". English. In: *Information and Communications Security*. Ed. by Liqun Chen, MarkD. Ryan, and Guilin Wang. Vol. 5308. LNCS. Springer, 2008, pp. 238–255. isbn: 978-3-540-88624-2. doi: 10.1007/978-3-540-88625-9_16. url: http://dx.doi.org/10.1007/978-3-540-88625-9_16.

[13] Thibaut Girka, David Mentré, and Yann Régis-Gianas. "A Mechanically Checked Generation of Correlating Programs Directed by Structured Syntactic Differences". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2015, pp. 64–79.

[14] Thibaut Girka, David Mentré, and Yann Régis-Gianas. "Verifiable semantic difference languages". In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*. ACM. 2017, pp. 73–84.

[15] *Git*. url: https://git-scm.com/.

[16] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[17] Susan Horwitz. "Identifying the Semantic and Textual Differences Between Two Versions of a Program". In: *SIGPLAN Not.* 25.6 (June 1990), pp. 234–245. issn: 0362-1340. doi: 10.1145/93548.93574. url: http://doi.acm.org/10.1145/93548.93574.

[18] James Wayne Hunt. *An algorithm for differential file comparison*.

[19] Shigeru Igarashi. "An axiomatic approach to equivalence problems of algorithms with applications". In: *PhD. Thesis* (1964).

[20] Ivar Jacobson et al. *The unified software development process*. Vol. 1. Addison-wesley Reading, 1999.

[21] Shuvendu K. Lahiri, Kapil Vaswani, and C A. R. Hoare. "Differential Static Analysis: Opportunities, Applications, and Challenges". In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 201–204. isbn: 978-1-4503-0427-6. doi: 10.1145/1882362.1882405. url: http://doi.acm.org/10.1145/1882362.1882405.

[22] Shuvendu K. Lahiri et al. "SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs". In: *Proceedings of the 24th International Conference on Computer Aided Verification*. CAV'12. Berkeley, CA: Springer-Verlag, 2012, pp. 712–717. isbn: 978-3-642-31423-0. doi: 10.1007/978-3-642-31424-7_54. url: http://dx.doi.org/10.1007/978-3-642-31424-7_54.

[23] Simon S Lam and A Udaya Shankar. "Protocol verification via projections". In: *IEEE transactions on software engineering* 4 (1984), pp. 325–342.

[24] Leslie Lamport. "Specifying concurrent program modules". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5.2 (1983), pp. 190–222.

[25] Leslie Lamport and Martin Abadi. "The existence of refinement mappings". In: *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*. July 1988, pp. 165–175.

[26] Xavier Leroy et al. "The CompCert verified compiler". In: *Documentation and user's manual. INRIA Paris-Rocquencourt* (2012).

[27] D MacKenzie, P Eggert, and R Stallman. *Comparing and Merging Files With Gnu Diff and Patch. Network Theory Ltd, Jan. 2003. Unified Format section*. url: http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.

[28] *Mercurial*. url: https://www.mercurial-scm.org/.

[29] Mads Chr. Olesen et al. "Clang and Coccinelle: Synergising program analysis tools for CERT C Secure Coding Standard certification". In: *4th International Workshop on Foundations and Techniques for Open Source Software Certification*. Pisa, Italy, Sept. 2010.

[30] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. "Semantic Patches, Documenting and Automating Collateral Evolutions in Linux Device Drivers". In: *Ottawa Linux Symposium (OLS 2007)*. Ottawa, Canada, June 2007.

[31] Yoann Padioleau, Julia L Lawall, and Gilles Muller. "SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers". In: *Electronic Notes in Theoretical Computer Science* 166 (2007), pp. 47–62.

[32] Nimrod Partush and Eran Yahav. "Abstract Semantic Differencing for Numerical Programs". English. In: *Static Analysis Symposium*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 238–258. isbn: 978-3-642-38855-2. doi: 10.1007/978-3-642-38856-9_14. url: http://dx.doi.org/10.1007/978-3-642-38856-9_14.

[33] Nimrod Partush and Eran Yahav. "Abstract Semantic Differencing via Speculative Correlation". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 811–828. isbn: 978-1-4503-2585-1. doi: 10.1145/2660193.2660245. url: http://doi.acm.org/10.1145/2660193.2660245.

[34] Suzette Person et al. “Differential Symbolic Execution”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT ’08/FSE-16. Atlanta, Georgia: ACM, 2008, pp. 226–237. isbn: 978-1-59593-995-1. doi: 10.1145/1453101.1453131. url: http://doi.acm.org/10.1145/1453101.1453131.

[35] Laurent Peuch. *RedBaron documentation*. 2014. url: https://redbaron.readthedocs.io/en/latest/.

[36] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. “Automated behavioral testing of refactoring engines”. In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 147–162.

[37] Ofer Strichman and Benny Godlin. “Regression Verification - A Practical Way to Verify Programs”. English. In: *Verified Software: Theories, Tools, Experiments*. Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. LNCS. Springer, 2008, pp. 496–501. isbn: 978-3-540-69147-1. doi: 10.1007/978-3-540-69149-5_54. url: http://dx.doi.org/10.1007/978-3-540-69149-5_54.

[38] Walter F Tichy. “RCS—a system for version control”. In: *Software: Practice and Experience* 15.7 (1985), pp. 637–654.