Université de Paris
École Doctorale 386 – Sciences Mathématiques de Paris Centre
Institut de Recherche en Informatique Fondamentale

Verification of Shell Scripts Performing File Hierarchy Transformations

Thèse de doctorat en informatique

Présentée et soutenue par

Nicolas Jeannerod

le 30 mars 2021

devant le jury composé de

Directeur de thèse	Ralf Treinen	Professeur	Université de Paris
Directeur de thèse	Yann Régis-Gianas	Maître de Conférences	Université de Paris
Rapporteur	Andreas Podelski	Professor	University of Freiburg
Rapporteur	Stéphane Demri	Directeur de Recherche	CNRS
Examinatrice	Julia Lawall	Directrice de Recherche	Inria
Examinateur	Roberto Di Cosmo	Professeur	Université de Paris
Examinateur	Michael Greenberg	Assistant Professor	Pomona College

Abstract

Title: Verification of Shell Scripts Performing File Hierarchy Transformations

Keywords: POSIX Shell · Unix Filesystems · Unix Utilities · Modelisation · Feature Tree Logics · Decidability of First-Order Logic · Efficient Constraints Solving · Debian · Software Quality Assurance · Software Package Installation

This thesis aims at applying techniques from deductive program verification and analysis of tree transformations to the problem of analysing Shell scripts. In particular, we aim at analysing Shell scripts that are used in software installation in the Debian GNU/Linux distribution. The final goal is to build a proof-of-concept tool able to read Debian packages – the way Debian has to distribute software – and report on their quality and on the potential bugs they might have.

Shell is a scripting language providing control structures around Unix utility calls. Unix utilities are objects that can perform all kind of transformation on Unix filesystems. We model Unix filesystems using feature trees and transformations of Unix filesystems using formulas in a feature tree logic named FTS. We describe these modelisations extensively and discuss their validity. The control structures of Shell scripts are converted to control structures in an intermediary language that has clearly defined semantics. This involves the definition of this intermediary language, the design of a static parser for Shell scripts and of a conversion that respects the semantics of both languages. The semantics of Shell scripts is then computed using symbolic execution of the aforementioned intermediary language, using a database of specifications of Unix utility calls as formulas of FTS. The result is, for each potential trace of execution of a Shell script, a formula of FTS describing the filesystem transformation this trace performs.

The main part of the thesis then focuses on decidability of formulas of FTS. The goal is to be able to detect traces of execution of Shell scripts that cannot happen and to check properties on the Shell scripts, such as “if the script fails, then it must not have performed any transformation”. A first, theoretical, part aims at showing that the full first-order theory of FTS is decidable. This goes by first reasoning only on Σ_1-formulas of FTS and defining a system of rules $\mathcal{R}_1$ that transforms Σ_1-formulas. We show that we can use $\mathcal{R}_1$ to decide the satisfiability of Σ_1-formulas as well as some other properties. We then extend the reasoning from Σ_1-formulas to first-order formulas of FTS using properties of $\mathcal{R}_1$ and weak quantifier eliminations. We conclude by stating that the first-order theory of FTS is indeed decidable. A second, more practical, part aims at designing efficient decision procedures for a subset of FTS rich enough to express the semantics of Unix utilities and Shell scripts. This goes by focusing on conjunctive formulas and improving on $\mathcal{R}_1$. This results in a system $\mathcal{R}_2$ which is more efficient on conjunctive formulas but would not have the required properties to prove decidability of the first-order. We then show how $\mathcal{R}_2$ can be implemented efficiently and that it can be extended without loss of efficiency to support specific forms of Σ_1-formulas.

Finally, this thesis describes the applications of the theoretical work to the implementation of a toolchain able to analyse all software packages in the Debian distribution and report on them. We describe our analysis and the bugs that we have found during the whole project. This thesis takes place within the CoLiS project, ANR-15-CE25-0001, taking place from October 2015 to March 2021.

Résumé

Titre : Vérification de scripts Shell effectuant des transformations de système de fichiers hiérarchiques

Mots-clefs : POSIX Shell · Système de fichiers · Utilitaires Unix · Modélisation · Logiques d'arbres de traits · Décidabilité d'une logique du premier ordre · Résolution efficace de contraintes · Debian · Qualité du logiciel · Installation de paquet logiciel

Cette thèse vise à appliquer des techniques de vérification déductive de programmes et d'analyse de transformations d'arbres au problème de l'analyse de scripts Shell. En particulier, nous visons à analyser les scripts Shell utilisés pendant l'installation de logiciels de la distribution Debian GNU/Linux. Le but final est de démontrer la faisabilité de notre analyse en développant un outil capable de lire des paquets Debian – le format dans lequel Debian distribue des logiciels – et de fournir un rapport sur leur qualité et sur les bogues potentiels qu'on pourrait y trouver.

Shell est un langage de script qui fournit des structures de contrôle autour d'appels d'utilitaires Unix. Les utilitaires Unix sont des objets qui peuvent effectuer toutes sortes de transformations sur des systèmes de fichiers Unix. Nous modélisons les systèmes de fichiers Unix à l'aide d'arbres de traits et les transformations de tels systèmes de fichiers à l'aide de formules dans une logique d'arbres de traits nommée FTS. Nous décrivons ces modélisations en détail et discutons leur validité. Les structures de contrôle des scripts Shell sont converties en des structures de contrôle d'un langage intermédiaire avec une sémantique clairement définie. Cela implique la définition du langage intermédiaire en question et le développement d'un parseur statique pour les scripts Shell et d'une conversion qui respecte les sémantiques des deux langages. La sémantique d'un script Shell est ensuite calculée par exécution symbolique sur le langage intermédiaire susmentionné en utilisant une base de spécifications des utilitaires Unix comme des formules de FTS. Pour chaque trace d'exécution d'un script Shell, le résultat est une formule de FTS décrivant la transformation de système de fichiers qu'effectue cette trace.

La partie principale de cette thèse s'intéresse à la décidabilité de formules de FTS. L'objectif est d'être capable de détecter des traces d'exécution de scripts Shell qui ne peuvent pas arriver et de vérifier des propriétés sur les scripts Shell, comme par exemple le fait que « si le script échoue, alors il ne doit pas avoir effectué de transformation. » Une première approche théorique vise à montrer que la théorie du premier ordre de FTS est décidable. Cela se fait par un raisonnement sur les formules Σ_1 de FTS et par la définition d'un système de règles $\mathcal{R}_1$ qui transforme les formules Σ_1. Nous montrons que nous pouvons utiliser $\mathcal{R}_1$ pour décider de la satisfiabilité de formules Σ_1, parmi d'autres propriétés. Nous étendons ensuite le raisonnement des formules Σ_1 vers les formules du premier ordre de FTS à l'aide de propriétés de $\mathcal{R}_1$ et d'une élimination faible des quantificateurs. Nous concluons en disant que la théorie du premier ordre de FTS est bien décidable. Une seconde approche, plus pratique, vise à créer des procédures de décision efficaces pour un sous-ensemble de FTS assez riche pour exprimer les sémantiques des utilitaires Unix et des scripts Shell. Nous faisons cela en nous intéressant en particulier aux formules conjonctives et en améliorant $\mathcal{R}_1$. Le résultat est un système $\mathcal{R}_2$ qui est plus efficace sur les formules conjonctives mais qui n'a pas les propriétés nécessaires pour prouver la décidabilité du premier ordre. Nous montrons ensuite comment $\mathcal{R}_2$ peut être implémenté efficacement et comment il peut être étendu pour supporter des formes spécifiques de formules de Σ_1 sans pertes d'efficacité.

Enfin, cette thèse décrit les applications de ce travail théorique à l'implémentation d'un groupe d'outils capable d'analyser tous les paquets logiciels de la distribution Debian et de fournir un rapport. Nous décrivons notre analyse et les bogues que nous avons trouvés au long du projet. Cette thèse s'inscrit dans le projet CoLiS, ANR-15-CE25-0001, se déroulant entre octobre 2015 et mars 2021.

Remerciements & Acknowledgements

I would like to start these acknowledgements by warmly thanking my reviewers, Andreas Podelski and Stéphane Demri, for accepting to read this manuscript in detail in such a short delay and my examiners, Julia Lawall, Roberto Di Cosmo and Michael Greenberg, for accepting to be part of my jury. I am honoured by the time and the attention that you all have put in my work.

Merci à vous deux, Ralf et Yann, pour tellement de choses. C'était un très grand plaisir de vous avoir comme directeurs de thèse. Merci pour la recherche passionnante que vous m'avez confié. Moi qui pensais ne pas faire de stage de recherche en M2, je me suis retrouvé à faire une thèse entière avec vous, et j'ai adoré. Vous avez été et êtes toujours des modèles, des partenaires de discussions, des soutiens précieux, des amis.

Je me considère très chanceux parce qu'en sus de mes deux directeurs de thèse, j'ai eu trois collègues très présents pour moi dans ma vie quotidienne. Je pense bien sûr à vous, Mihaela, Claude et Benedikt. Travailler avec vous a été extrêmement intéressant et d'une grande émulation. Merci !

Faire une thèse à l'IRIF aura été un grand plaisir. C'est extrêmement rassurant de savoir que l'on peut passer au bureau d'à côté poser une question scientifique et qu'on repartira avec des questionnements sur le nombre de parts de pizza qu'on peut faire rentrer dans un micro-ondes trop petit[1]. Merci à la foule de thésard·e·s des troisième et quatrième étages qui contribuent à faire de ce lieu un cadre de travail accueillant.

Tout d'abord, merci à toi, Victor, d'avoir été le meilleur cobureau imaginable. Notre rencontre dans le placard des stagiaires[2], nos séances de jeux vidéo[3], nos longues discussions sur toutes sortes de sujets, et tellement d'autres souvenirs, sont et resteront gravées dans ma mémoire.

Merci à toi, Jovana, pour ces regards noirs quand les clics souris commençaient à indiquer que je n'étais plus tout à fait en train de travailler. Merci pour tous ces moments passés ensemble, au bureau, à la cantine ou au bar. All these classes we skipped together did not mean a thing!

Merci à tous mes autres cobureaux[4] de m'avoir supporté également. Étienne, merci pour ce goût partagé des problèmes importants, de LaTeX aux palindromes en passant par `bash`. Merci Anne[5] pour tous ces moments passés ensemble à jouer de la musique et à parler de tout et de rien. S'il y avait une seule personne que je laisserais épouser Victor à ma place, c'est bien toi ! Adrien, j'ai hâte de te revoir pour une autre session de jeux de société. Merci enfin à toi, Pierre, dernier en date mais déjà si grand !

Merci à tou·te·s les autres thésard·e·s, que vous soyez arrivé·e·s avant ou après moi. Merci à la génération

[1] Problème pour lequel nous avons, si je ne m'abuse, une conjecture mais pas de preuve.

[2] « Tiens, salut, je ne crois pas t'avoir déjà vu, tu viens d'où ? Le MPRI ? Ah bah moi aussi. »

[3] Mais jamais sur les horaires de bureau.

[4] Qu'il est moche, ce mot.

[5] Franchement, ça compte.

précédente – Cyrille, Pierre, Maxime, Ludovic, Amina, Marie, Léonard, Thibaut, Clément, Yann, Théo – pour votre accueil chaleureux et pour les longs débats sur le meilleurs CROUS. Merci à ma génération – Zeinab, Léo, Rémi, Cédric et tou·te·s les autres que j'oublie – d'avoir pris la relève à tous les niveaux et d'avoir écrasé le débat en mangeant à la cantine du ministère.

Merci à vous, les non-thésards[6], Juliusz, Laurent, Daniel, qui passez me rendre visite[7] et me traînez au bar quand l'heure se fait tardive. Merci à vous, l'équipe de Dans la cuisine, Mika, Pétro, Gigi, qui rendez la visite de votre établissement plus plaisante à chaque fois.

Merci à l'équipe administrative de choc qui m'a débloqué de nombreuses situations. Odile, Dieneba, Natalia, Nawel, votre attention et votre efficacité ne sont pas passées inaperçues !

Merci enfin à tou·te·s mes collègues d'enseignement et mes étudiant·e·s qui ont rendu ces heures de TD et TP amusantes et enrichissantes. Merci en particulier à toi, Étienne, ami rencontré au hasard pendant ton stage de L2 !

Dans ma vie personnelle également, je me considère extrêmement chanceux. J'ai une famille merveilleuse. Claire, Adrien, Papa, Maman, merci infiniment d'être qui vous êtes. Je vous aime. Merci à vous, mes grand-parents, mes oncles et tantes, mes cousins et cousines. Vous me manquez et j'ai hâte que cette pandémie soit derrière nous et qu'on puisse faire de grandes cousinades à nouveau ! Merci aussi à vous, toutes les gens de Maranatha. Chiche de refaire une réunion quand on pourra ?

Merci enfin à mes ami·e·s. À toutes les étapes de ma vie, vous m'avez fait rire, pleurer, vous m'avez soutenu et remotivé.

Merci à vous, Nicolas, Clément, Adrien, Margot, Erwan, mes plus vieux·eilles et fidèles ami·e·s !

Merci à Florence pour m'avoir transmis l'amour de la clarinette.

Merci à mes belles rencontres de prépa. Aymeric, merci d'être toujours là pour moi après toutes ces années pendant lesquelles nous avons tous deux beaucoup changé. Romain[8], merci pour tout ce que tu m'as apporté ; tu as eu une influence sur moi bien plus grande que tu ne le penses. Nina, merci pour nos discussions, nos réflexions. Tu as contribué énormément à la personne que je suis aujourd'hui.

Merci à mes ami·e·s de l'ENS et notamment du B6. Vous êtes trop nombreux·ses pour être tou·te·s cité·e·s ici, mais je vous porte dans mon cœur ! Merci à toi, Julie, de m'avoir ramassé parfois à la cuiller mais toujours avec délicatesse, et pour partager mon amour des blagues douteuses[9]. Merci à toi, Marion, pour ces heures de débats autour de camemberts bon marché au fond d'un bar.

Merci également à mes ami·e·s de la danse, à Paris, à Lyon et partout ailleurs. J'ai hâte de pouvoir enfin danser de nouveau avec vous ! Merci en particulier au comité d'organisation de la branche de Paris pour toutes ces réunions dans la bonne humeur, merci au Paris Band pour toutes ces heures de musique. Merci à toi, Claire, pour tout ce temps passé ensemble. Merci à toi, Shannon, pour ces longues discussions et balades dans les bois. Merci enfin à toi, Pascaline, pour toutes ces affreuses batailles de sophismes.

Merci à mes colocataires pour m'avoir supporté même en voyant ma tête tous les jours. Merci à toi, Catalina, pour cet art des questions intimes, abruptes et délicates à la fois. Merci à toi, Martin, pour ces (très, trop ?) nombreuses heures de TFA et pour ces longues discussions au moments où nous en avions le plus besoin. Quand est-ce qu'on se remet en colocation ?

[6] On parle de permanents et non-permanents. C'est pas juste.
[7] Probablement plus pour le canapé que pour moi, cela dit !
[8] À qui l'on doit la genèse du nom « Niols ».
[9] Petite pensée pour notre PEB parti trop tôt.

Last but not least, thank you, Jana. Meeting you was— well, it could have been so much worse.

Contents

Page

Chapter 1

Introduction

1.1 History & Motivation

Humankind has built and used devices to aid computation for thousands of years. It is only in the 20th century, however, that *modern computers* got invented. Contrary to most previous devices, modern computers store *programs* in their memory. A person who would want to modify the behaviour of such a computer would then only need to update its memory, and not rebuild a whole machine. The idea of a general-purpose computer whose behaviour is specified by its memory goes back to 1837 and Charles Babbage and Ada Lovelace's Analytical Engine[1]. The idea of *stored programs* in the modern sense was introduced a century later by Alan Turing [Turing 1937] and the first digital computer running such *stored programs* was the Manchester Baby[2] which ran for the first time in 1948.

This idea of stored programs makes modern computers very versatile objects capable of solving a wide range of problems, as long as one or several *programmers* take the time to update the program. In the beginning, computers were monolithic devices only operated by experts and programs were short. Quickly, however, as computers became more powerful, there was a huge increase in the complexity of the problems that they could solve. The size of programs followed and easily reached thousands of lines. For instance, the code of the guidance computer for the command module of Apollo 11, which was developed in the 1960s and helped to safely land a human on the moon in 1969, contains a bit over 25,000 lines of code.[3]

Bigger code and more lines of code meant more problems, which people started to be aware of. By 1968, the term *software crisis*[4] was coined. It encompasses all the problems related to the difficulty of writing good quality computer programs in the required time. This covers problems with project, time and budget management but also problems of efficiency and quality of software: an awareness for quality assurance was born.

Coincidingly – starting in the early 1960s[5] –, in order to speed up processing, operating systems were

[1] See `https://en.wikipedia.org/wiki/Analytical_Engine`.

[2] Or "Small-Scale Experimental Machine". See `https://en.wikipedia.org/wiki/Manchester_Baby`.

[3] For the Apollo guidance computer, see `https://en.wikipedia.org/wiki/Apollo_Guidance_Computer`. For its program, see `https://github.com/chrislgarry/Apollo-11/`. That same repository also contains the code of the lunar module, which also counts a bit more than 25,000 lines of code.

[4] See `https://en.wikipedia.org/wiki/Software_crisis`.

[5] In the 1950s, some basic operating system features such as *resident monitor* functions were developed. Such functions could automatically run several programs one after the other in order to save time. The modern form of operating systems only came in the early 1960s and therefore we chose to keep that date.

developed. They are programs that manage resources – and, in particular, time – for other programs to live together.

One family of operating systems will be of particular interest to us. Derived from the original AT&T Unix, whose development started in the 1970s, Unix operating systems inherently support multitasking and multiusers. This means that a unique computer with one operating system could support several users doing several different tasks at the same time.[6]

With the rise of affordable personal computers[7] from the mid-1970s onwards, the model of operating systems changed from a few big computers with several users and an expert team of system administrators to a lot of microcomputers with one user each. This was a big change as it meant that operating systems, programs and their updates had to be distributed to a lot of non-expert users. The expertise of installation, updating, etc. could not rely on such users and therefore had to move from the system administrators to the software providers, under the form of automated setup scripts.

During the late 1980s and the early 1990s, several things happened that are of relevance for this story. Firstly, Internet[8] became more and more widespread in academia in particular and, because of its commercialisation in the early 1990s, everywhere. Secondly, the *GNU project*[9] took off and the development of the *Linux kernel*[10] started. Together, they form one of the most widespread basis for *Linux distributions*. Finally, the *Debian GNU/Linux*[11] distribution, one of the biggest and oldest operating systems based on the Linux kernel, launched its first release in September 1993.[12]

Linux distributions are a way for users to have access to a rich and consistent ecosystem of programs from which they can pick those that are of interest to them. In order to eliminate the need for manual installations and updates, distributions often feature a *package manager*[13]. These are software tools that automate the process of installing, upgrading, configuring, or removing programs on an operating system. The package manager at the base of the package management system of Debian – and its numerous derivatives – is named `dpkg`[14].

Package managers handle packages as a way to distribute software as well as the instructions for the package manager to install, update, remove, etc. the software in question. A Debian package is made of several elements [22, Chapter 3], some of which are detailed in the following list.

- The package contains metadata about the software, its version, its dependencies – the packages that need to be installed first[15] –, etc. Figure 1.1 shows an excerpt of a file containing such metadata for the package `rancid-cgi`.
- The package must also contain the *static content* of the software: an archive of files to be placed on the target machine when installing the package. These are the files of the software itself: its binaries – the executable part –, its configuration files, etc.

[6] For instance, one user could compute things while editing a document and, at the same time, another user could read another document while sending it to a printer.

[7] See `https://en.wikipedia.org/wiki/Personal_computer`.

[8] See `https://en.wikipedia.org/wiki/Internet`.

[9] See `https://en.wikipedia.org/wiki/GNU`.

[10] See `https://en.wikipedia.org/wiki/Linux`.

[11] See `https://en.wikipedia.org/wiki/Debian`.

[12] My big brother by a few months!

[13] See `https://en.wikipedia.org/wiki/Package_manager`.

[14] **D**ebian **P**ac**k**a**g**e. See `https://en.wikipedia.org/wiki/Dpkg`.

[15] Actually, these would be the pre-dependencies. The pre-dependencies are necessary at the time of installation of the package; they must therefore be installed before the package itself. The dependencies are only necessary when the software contained in the package is used. Technically, they can be installed at any time before the use of the software, and that can be after its installation.

- Finally, the package may come with a number of so-called *maintainer scripts* which are executed when installing, upgrading, or removing the package. The name of *maintainer* scripts comes from the fact that these scripts are not given by the provider of the software, but by the maintainer of the package, which is the person (or team) in charge of the package in Debian.

For the sake of the example, Figure 1.2 shows how an installation of `rancid-cgi` could look like in Debian. There are various ways to install software in Debian, via the command-line, terminal user interface (TUI), or graphical user interface (GUI). We chose here to show a way which we believe to be widespread among system administrators. This is the installation of `rancid-cgi` using the `apt` utility[16] in a terminal. `apt` is a *meta* package manager. It handles resolution of dependencies and downloading of packages but leaves the actual operations on packages to `dpkg`, the only real package manager of Debian. Let us now detail all the steps of installation.

1. Line 1 shows the prompt `root@debian~#`, which is simply the system awaiting a command, as well as the command `apt install rancid-cgi` asking the utility `apt` to install the package `rancid-cgi`. All the following lines are then written by the `apt` utility.

2. Lines 2 to 9 show the resolution of dependencies. This phase aims at deciding which packages need to be installed before or at the same time as the requested package.

3. Lines 10 to 15 show the downloading of the actual packages. Since there are 60,000 packages in Debian, having them all on one's computer would be a waste of space.[17] The packages are therefore kept on an external storage medium or even the network and downloaded from an archive when required.[18]

4. Lines 16 to 18 show the pre-configuration phase. The goal of this phase is to prepare the system to receive the software. This can mean cleaning up previous versions, or checking that certain files are stored at the right location, etc. The instructions are described by the `preinst` maintainer script.

5. Lines 19 to 23 show the unpacking phase. This is the part where the static content of the package is actually put on the system.

6. Lines 24 to 25 show the configuration phase. This can mean launching utilities to register the software in databases, running configuration scripts that adapt the behaviour of the software to this specific machine, etc. The instructions are described by the `postinst` maintainer script.

7. Line 26 shows the processing of a trigger. This is a phase of configuration of other packages to take into account the arrival of the new package. In our example, the trigger is for `man-db` which is a package handling the user manuals of all the installed packages.

8. Line 27 shows the return of the prompt, which simply means that the command is done running and that the system awaits for further commands.

The interesting steps for us are Steps 4, 5 and 6. In particular, Steps 4 and 6 require running the `preinst` and `postinst` maintainer scripts with full privileges on the machine. This is potentially problematic as this means trusting that such scripts will not contain bugs or malicious code.

The problem becomes increasingly complex when the number of packages grow bigger. Of course, more users means more different needs and therefore more software, packages and maintainer scripts. As of

[16] **A**dvanced **P**ackage **T**ool. See `https://en.wikipedia.org/wiki/APT_(software)`.

[17] Moreover, some of them would be incompatible. There exists some research on the co-installability of packages but this is off-topic here.

[18] Of course, this is fairly simple now that most people have a reasonable internet connection, which has not always been the case and is still not the case for everyone on the planet.

```
Package: rancid-cgi
Version: 3.9-1
Priority: optional
Section: net
Source: rancid
Maintainer: [...]
Installed-Size: 152 kB
Depends: liblockfile-simple-perl, rancid, perl:any
Suggests: apache2 | httpd-cgi
Homepage: https://www.shrubbery.net/rancid/
Tag: implemented-in::perl, interface::web, role::program, scope::utility,
 use::monitor, web::cgi
Download-Size: 76.6 kB
APT-Sources: http://debian.mirrors.ovh.net/debian buster/main amd64 Packages
Description: looking glass CGI based on rancid tools
 The looking glass is a web interface for [...]
```

Figure 1.1: Metadata of the rancid-cgi package (excerpt)

```
root@debian:~# apt install rancid-cgi
Reading package lists...
Building dependency tree...
Reading state information...
The following additional packages will be installed:
  rancid
The following NEW packages will be installed:
  rancid rancid-cgi
0 upgraded, 2 newly installed, 0 to remove and 0 not upgraded.
Need to get 311 kB of archives.
After this operation, 2,103 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://debian.mirrors.ovh.net/debian buster/main amd64 rancid amd64 3.9-1 [234 kB]
Get:2 http://debian.mirrors.ovh.net/debian buster/main amd64 rancid-cgi all 3.9-1 [76.6 kB]
Fetched 311 kB in 0s (6,601 kB/s)
Preconfiguring packages ...
Preparing to unpack .../rancid_3.9-1_amd64.deb ...
Preparing to unpack .../rancid-cgi_3.9-1_all.deb ...
Selecting previously unselected package rancid.
(Reading database ... 123451 files and directories currently installed.)
Unpacking rancid (3.9-1) ...
Selecting previously unselected package rancid-cgi.
Unpacking rancid-cgi (3.9-1) ...
Setting up rancid (3.9-1) ...
Setting up rancid-cgi (3.9-1) ...
Processing triggers for man-db (2.8.5-2) ...
root@debian:~#
```

Figure 1.2: Installation of rancid-cgi with APT on Debian (excerpt)

```
 1  [...]
 2
 3  Package: cmigrep
 4  Version: 1.3-1
 5  Severity: critical
 6  Justification: breaks unrelated software
 7
 8  cmigrep's emacsen-install script is overzealous; specifically, it
 9  inappropriately attempts to compile all .el files in
10  /usr/share/emacs/site-lisp even if they don't work with the current
11  emacsen flavor (for instance, remembrance-agent's remem.el
12  vs. xemacs), and compounds the problem by removing [...]
```

Figure 1.3: A bug report on the package `cmigrep` (excerpt)

October 6, 2019, the Debian distribution[19] contains 28,814 maintainer scripts in 12,592 different packages (out of a total of 60,000), 9,771 of which are completely or partially written by hand. Moreover, the big user base of Debian brings about a big number of different devices, architectures, installed combinations of software, etc. It is therefore impossible for the package maintainers to plan everything in advance and it is inevitable to meet corner cases that will behave differently than expected.

The consequences of a faulty script can go from mild – eg. the package fails to install – to critical – eg. the system does not start anymore, or user data are lost. For instance, Figure 1.3 shows a bug report [1]. Lines 3 to 6 contain metadata about the bug, namely the package and version it applies to (here, `cmigrep` in version `1.3-1`), the severity of the bug (here, critical) and a short description of the bug. Lines 8 to 12 contain an excerpt of the longer, more detailed description of the bug. In this case, the installation or upgrade of the package `cmigrep` breaks other software, which can happen in the middle of a global update of the system, without the user noticing before they actually try to use the other broken software.

The bug in question came from a change in the package that the maintainer carried without noticing that it would imply the removal of files belonging to – and necessary for the proper functioning of – other packages. Since they may have to perform any kind of action on the target machine, the maintainer scripts are almost exclusively written in POSIX Shell, a general-purpose scripting language that allows for invoking any Unix command. Unfortunately, this language is full of pitfalls that make it easy for anyone – including package maintainers that are used to it – to make tiny mistakes leading to grave consequences.

The situation is therefore the following. The critical process of installing, upgrading and removing software in Debian – and many other distributions – relies on thousands of scripts written by voluntary package maintainers in a language in which it is easy to make mistakes. Hence, there is a cruel need of work on quality assurance of those packages.

1.2 Approaches & Feature Tree Logics

1.2.1 Other Approaches – Related Works

When facing such a problem of quality assurance, many approaches are possible and the Debian community did not wait for the third millennium to try and improve the quality of Debian packages on a global scale. There is even a team of Debian specialised in the quality assurance of packages.[20]

One product of the work to improve the quality of Debian packages is the Debian Policy [22]. It is a

[19] And, more exactly, *sid* for *amd64*, including *contrib* and *non-free*.

[20] Named… the Debian Quality Assurance Team! See `https://qa.debian.org/`.

document, written in natural language, that aims to normalise important technical aspects of packages. It prescribes the control flow of the different stages of the package installation process, including attempts of error recovery and defines how dpkg invokes maintainer scripts. It also states requirements on the syntax and on the execution behaviour of scripts.

Some automation of the checking of such properties – and therefore of the quality of Debian packages – has been added to Debian over the years. Lintian[21], for instance, automatically checks syntactic properties of the packages of the Debian distribution – and not only in their scripts but also in the other files that compose a package. Another tool, the piuparts suite[22], checks if packages can be installed, upgraded and removed successfully in a clean environment. The piuparts suite does not check whether the installed software is actually usable. Package maintainers are however encouraged to provide such tests in their packages via the autopkgtest facility.[23] The continuous integration[24] of Debian runs these tests systematically on all the packages of the Debian archive. Finally, the process of acceptance of packages in the Debian distribution is trying to give time to voluntary users to stumble upon bugs before they reach more critical users by providing an unstable distribution – named *sid* – receiving all the updates and a stable distribution receiving the updates only after they have been validated by the users of *sid*.

No solution is ever perfect, and we believe that a lot of formal verification techniques can apply to this problem and provide ways to check more properties, or new ways to check the same properties. Some work has already been done in this line, most notably by the EDOS and Mancoosi projects[25] [Mancinelli et al. 2006; Abate et al. 2012] which aimed at checking properties of the dependencies of packages and at providing efficient solvers for those dependencies. The content of maintainer scripts was however never their concern.

None of these approaches would have found the aforementioned bug in cmigrep, version 1.3-1. Indeed, this bug only occurs during the installation of cmigrep on a machine that already has other related packages installed – in that case, packages belonging to the Emacs ecosystem. In a real-world use case, this will happen easily because users are susceptible to indeed have these other packages. In the testing performed by the piuparts suite, packages are installed and removed in a clean environment, in which the bug would therefore not occur.

1.2.2 Our Approach

In the CoLiS[26] project, we believe that we can contribute to the quality assessment of Debian packages by applying already existing formal method techniques to maintainer scripts. Such techniques could allow to verify properties and find bugs that tools like Lintian or the piuparts suite cannot uncover. The bugs in Debian maintainer scripts that we attempt to find may come at different levels: simple syntax errors (which may very well go unnoticed due to the unsafe design of the POSIX Shell language), non-compliance with the requirements of the Debian Policy, usage of unofficial or undocumented features, or failure of a script in a situation where it is supposed to succeed.

In this context, the Debian Policy [22] is a very useful tool for us as it specifies all the properties that can be expected from maintainer scripts. This document has been extensively enriched over the years by the very users and maintainers of the packages. We therefore do not need to define the problem nor what it is we aim at verifying. There is however a necessary phase of formalisation of the problem from the

[21] A **lint**er for Deb**ian**. See https://lintian.debian.org/.

[22] **P**ackage **i**nstallation, **up**grading **a**nd **r**emoval **t**esting **s**uite. See https://piuparts.debian.org/.

[23] See https://salsa.debian.org/ci-team/autopkgtest/raw/master/doc/README.package-tests.rst.

[24] See https://ci.debian.net/doc/.

[25] See https://www.mancoosi.org/ and https://www.mancoosi.org/edos/ – the EDOS website seems to be dead now.

[26] CoLiS stands for Correctness of Linux Scripts.

explanations of the Debian Policy, as it is written in an informal natural language, leading to ambiguities and incompleteness.

To the information given by the Debian Policy, we add a information of what appears in the maintainer scripts which we obtain by running statistical analysis on the corpus of all maintainer scripts. This analysis allows us to highlight what needs to be handled in priority, what can be postponed and what can be ignored. This analysis is described in Section 7.1.

Our project is oriented towards finding bugs and not fully certifying that scripts respect all the requirements they are supposed to. This allows us to do as many approximations as necessary in the process of finding such bugs, as long as we can report clearly on the bugs and can show that they actually happen. Of course, these approximations are not random and are guided by the aforementioned statistical analysis.

The particular setting of Debian packages and the freedom to approximate when required are two key elements that made this analysis even possible.

We are still facing two important challenges. The first one lies in the language used for these maintainer scripts we are interested in. The Debian Policy states that the standard Shell interpreter is POSIX Shell, with the consequence that 99% of all maintainer scripts are written in this language. This language has a surprising specification, is full of pitfalls, highly dynamic and recalcitrant to static analysis, both on a syntactic and semantic level. The modelisation of POSIX Shell is the topic of Chapter 6.

The second challenge comes from the nature of Unix systems on which the maintainer scripts are run. A Unix filesystem implementation contains many features that are difficult to model, eg. ownership, permissions, timestamps, symbolic links, and multiple hard links to regular files. There is an immense variety of Unix utilities that may be invoked from scripts, all of which have to be modelled in order to be treated by our tools. To address properties of scripts required by the Debian Policy, we need to capture the transformation done by the script on a filesystem hierarchy. For this, we need some kind of logic that is expressive enough, and still allows for automated reasoning methods. In this work, we use feature tree logics.

1.2.3 Feature Tree Logics

Although we contributed to all the parts of the toolchain described in Section 1.3, our main contribution – and the main topic of this thesis – is the design of decision procedures for *feature tree logics*. This leads to theoretical decidability results for such logics and to the design of an efficient backend for the symbolic execution engine mentioned in Section 1.3.1.

Feature trees are trees where nodes have an unbounded number of children, and where edges from nodes to their children carry names – called *features* – such that no node has two different outgoing edges with the same name. Hence, the names on the edges can be used to select the different children of a node. Feature trees have been used in constraint-based formalisms in the field of computational linguistics [Smolka 1992] and constrained logic programming [Aït-Kaci et al. 1994; Smolka & Treinen 1994]. The work presented in this thesis is motivated by a different application of feature trees: we find them to be a quite accurate model of Unix filesystems. We will discuss this in Section 3.1 and discuss why abstracting Unix filesystems as feature trees makes sense in this work.

Feature tree logics have at their core basic constraints like $x[f]y$, expressing that y is a subtree of x accessible from the root of x via a feature f and $x[f]\uparrow$, expressing that the tree x does not have a feature f at its root node.

This is already sufficient to describe some tree languages that are useful in our context. For instance, the command `mkdir /etc/rancid/apache.conf` which creates the directory `/etc/rancid/apache.conf` (mkdir stands for make directory), succeeds on a tree that satisfies Formula 1.1.

$$\exists x, y, z \cdot (r[\texttt{etc}]x \wedge x[\texttt{rancid}]y \wedge y[\texttt{apache.conf}]\uparrow) \tag{1.1}$$

Formula 1.1 expresses that etc is a subdirectory of the root r represented by the variable x, which has itself a subdirectory rancid represented by the variable y, which itself *does not* have a subdirectory apache.conf. We ignore here the difference between directories and regular files, as well as file permissions.

In order to express the relation between the input and output trees of this example, we need more expressivity. A first idea is to introduce an update predicate $y = x\{f \mapsto z\}$, which states that the tree y is obtained from the tree x by changing the child reachable through f to z, and creating the child when it does not exist. Using this, the semantics of `mkdir /etc/rancid/lg.conf` could be described by Formula 1.2, of the input root r and output root r'.

$$\exists x, y, z, x', y' \cdot \begin{pmatrix} r[\texttt{etc}]x \wedge x[\texttt{rancid}]y \wedge y[\texttt{apache.conf}]\uparrow \\ \wedge\, r' = r\{\texttt{etc} \mapsto x'\} \wedge x' = x\{\texttt{rancid} \mapsto y'\} \\ \wedge\, y' = y\{\texttt{apache.conf} \mapsto z'\} \wedge z'[\star]\uparrow \end{pmatrix} \tag{1.2}$$

Here, $z'[\star]\uparrow$ expresses that all features $(\star)$ are absent from z', which means that z' is an empty directory. Note that this formula, by virtue of the update constraint, expresses that any existing file under etc different from rancid is not touched. Similarly, any existing file under rancid different from apache.conf is not touched.

The difficulty in solving such update predicates stems from the fact that they involve three trees: the original tree, the final tree and the sub-tree that gets grafted onto the original tree. There are no symmetries between these three arguments, and a conjunction of several update predicates may become quite involved. Our approach to handle this rather complex predicate is to replace it by a more elementary predicate system based on the classical $x[f]y$ and a new *similarity predicate* $x =_F y$, where F is a set of features. The latter expresses that x and y have the same children (or absence of children) in all the names of F. In particular, applied to a feature f, the predicate $x =_{^c\{f\}} y$ – where ^cF is the complement of F – expresses that x and y are the same everywhere, except possibly in f where they may differ.

The similarity predicates of the form $x =_{^c\{f\}} y$ have the same expressive power as the update predicates since, on the one hand, $x' = x\{f \mapsto y\}$ is equivalent to $x =_{^c\{f\}} x' \wedge x'[f]y$ and, on the other hand, $x =_{^c\{f\}} x'$ is equivalent to $\exists y, z \cdot (z = x\{f \mapsto y\} \wedge z = x'\{f \mapsto y\})$. Moreover, for each set of features F, similarity predicates $=_F$ are equivalence relations, which is very useful when designing simplification rules, and these relations have useful properties, such as $(x =_F y \wedge x =_G y) \leftrightarrow x =_{F \cup G} y$ and $(x =_F y \wedge y =_G z) \rightarrow x =_{F \cap G} z$.

The feature tree logic augmented with this similarity predicate, named FTS, is presented in Section 3.2. This thesis then presents two directions of work on FTS. The first line of work establishes theoretical decidability results for the full first-order theory. The case of a feature tree logic with update constraints was open up to now. This line of work is described in details in Chapter 4. In a second line of work, we aim at designing efficient decision procedures for a restricted set of formulas with implementation in mind. It is described in detail in Chapter 5.

1.3 Overview of the Toolchain

Let us take the package rancid-cgi [32] as a running example. It comes with only two maintainer scripts: preinst and postinst. The goal of the CoLiS project is to build a toolchain able to take this

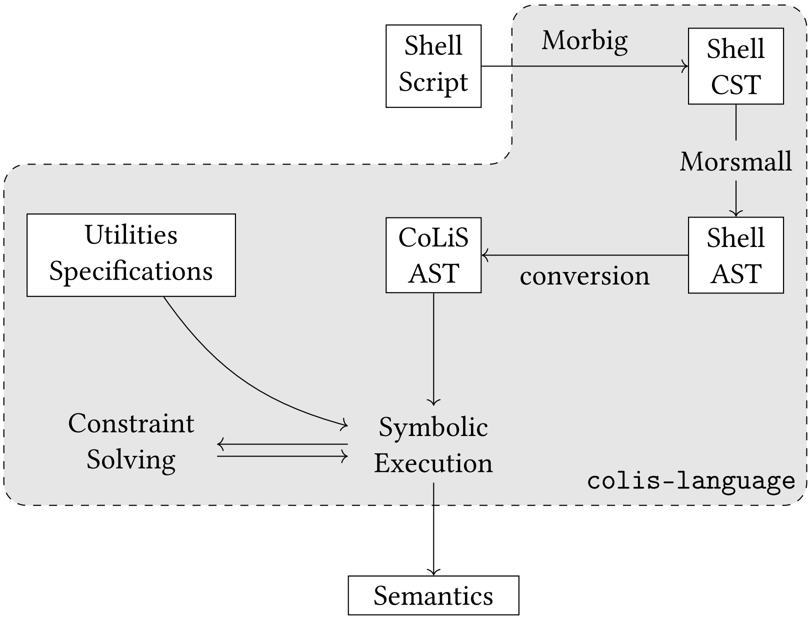


Figure 1.4: `colis-language`: toolchain for the analysis of one Shell script

```
if [ -h /etc/rancid/lg.conf ]; then
  rm /etc/rancid/lg.conf
fi
if [ -e /etc/rancid/apache.conf ]; then
  rm /etc/rancid/apache.conf
fi
```

Figure 1.5: `preinst` script of the `rancid-cgi` package

package as input and to automatically compute and output a report explaining the status of this package and potentially showing the presence of bugs.

1.3.1 First Layer – One Script

Since this work focuses on the Shell scripts present in packages, a first layer of our toolchain will need to comprise tools to read Shell scripts, reason about them and compute their semantics in some formalism. The first layer of our toolchain, `colis-language` [11], analyses one Shell script and computes its semantics. It is summarised in Figure 1.4.

As we have said, the huge majority of maintainer scripts is written in Shell. Shell is a scripting language that provides control flow structures around calls to Unix utilities. As an example, let us consider the `preinst` script of `rancid-cgi`, presented in Figure 1.5. It contains four utility calls, the first one being `[ -h /etc/rancid/lg.conf ]`. The script reads as follows: if the symbolic link `/etc/rancid/lg.conf` exists then it is removed; if the file `/etc/rancid/apache.conf` exists, no matter its type, it is also removed. Both removal operations use the POSIX utility `rm` which, without options, cannot remove directories. Hence, if `/etc/rancid/apache.conf` is a directory, this script fails while trying to remove it.

The Shell scripts are first being parsed using Morbig and Morsmall. These two tools have been developed as part of the CoLiS project. The former, Morbig, is a static parser for POSIX Shell providing *concrete syntax trees* for Shell scripts. The latter, Morsmall, is a wrapper around Morbig that provides *abstract*

```
if test [ '-h'; '/etc/rancid/lg.conf' ] then
  rm [ '/etc/rancid/lg.conf' ]
fi
if test [ '-e'; '/etc/rancid/apache.conf' ] then
  rm [ '/etc/rancid/apache.conf' ]
fi
```

Figure 1.6: `preinst` script of the `rancid-cgi` package in CoLiS

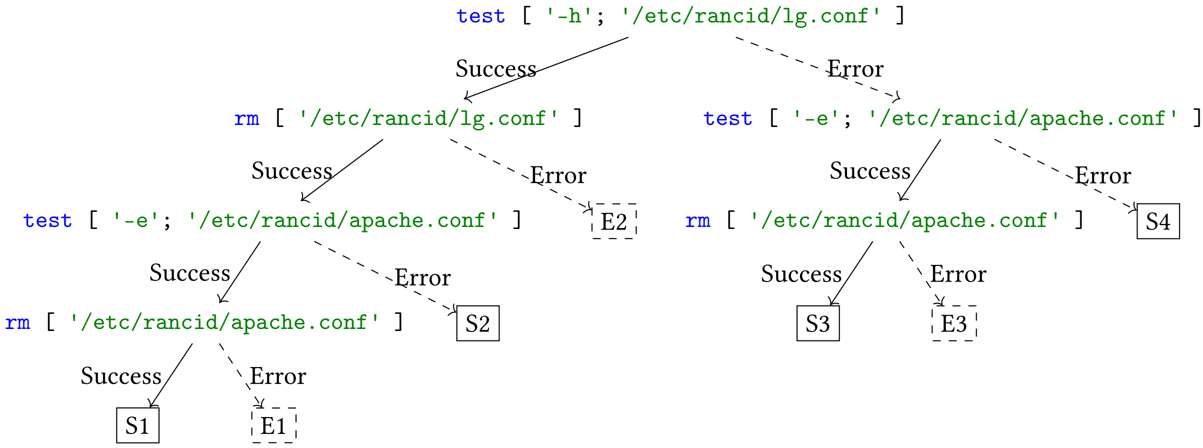

test ['-h'; '/etc/rancid/lg.conf']
Success
Error
rm ['/etc/rancid/lg.conf']
test ['-e'; '/etc/rancid/apache.conf']
Success
Error
Success
Error
test ['-e'; '/etc/rancid/apache.conf']
E2
rm ['/etc/rancid/apache.conf']
S4
Success
Error
Success
Error
rm ['/etc/rancid/apache.conf']
S2
S3
E3
Success
Error
S1
E1

Figure 1.7: Traces of execution of the `preinst` script of the `rancid-cgi` package

syntax trees for Shell scripts. They are described in Chapter 6.

The Shell scripts are then converted to an intermediary language named “CoLiS”. The CoLiS version of this `preinst` script of `rancid-cgi` is shown in Figure 1.6. The Shell and CoLiS versions of the script look very similar, which comes from the fact that CoLiS aims at being the target of an automated conversion from Shell. However, some key pitfalls of Shell have been dealt with and eliminated from CoLiS. In the example of Figures 1.5 and 1.6, one can for instance see that the CoLiS language features more structure than Shell, and contains for instance delimiters of strings.

The semantics of CoLiS scripts is then computed by symbolic execution. Basically, this explores all the possible traces of execution of the scripts in terms of success and error of the utility calls they contain. For instance, for the script of Figure 1.6, the traces would be that of Figure 1.7. This script has a total of seven traces, four of which lead to a successful exit (S1, S2, S3 and S4) and three of which lead to an error exit (E1, E2 and E3).

For each of these traces, we want to compute on which filesystems they may happen and what transformation they perform in that case. For instance, the trace leading to S4 performs no transformation and happens when `/etc/rancid/lg.conf` is not a symbolic link (or does not exist) and `/etc/rancid/apache.conf` does not exist.

By computing on which filesystems the trace can happen, we can also hope to detect traces that are not reachable. In the example, for instance, the trace leading to E2 is not reachable because `rm [ '/etc/rancid/lg.conf' ]` cannot fail if we know for sure that `/etc/rancid/lg.conf` is a symbolic link, which we do because we are in a success case of `test [ '-h'; '/etc/rancid/lg.conf' ]`. The detection of unreachable traces can also be done on the fly in order to stop exploring them as soon as possible.

In order to do that, we need three ingredients. Firstly, we need a formalism in which to express transfor-

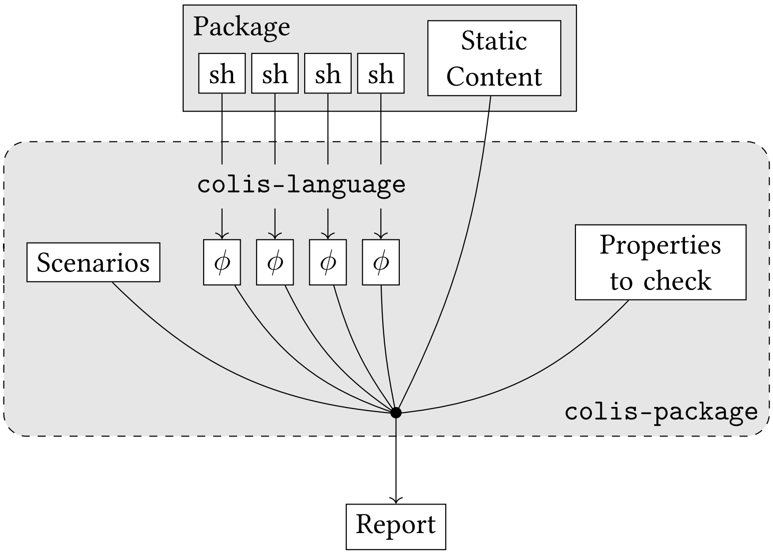


Figure 1.8: `colis-package`: toolchain for the analysis of a Debian package

mations of trees. This will be FTS, the logic of feature trees mentioned in Section 1.2.3.[27] Secondly, we need a database of the transformations that are associated to every success of error cases of a command. Thirdly, we need a solver able to take a list of transformations associated to commands and to compute if the composed transformation can actually happen.

The parsing phase is described in details in Section 6.1. The CoLiS language and the conversion are described in details in Section 6.2. The symbolic execution engine is detailed in Section 7.2. The model used to represent trees is defined in Section 3.1 and FTS is defined in Section 3.2. The modelisation of utilities is described in Section 3.3 and their specification is developed in Section 3.4. The constraint solving is the main topic of this thesis and is developed in Chapter 4 for the theoretical aspect and Chapter 5 for efficiency considerations.

1.3.2 Second Layer – One Package

Once we are able to compute the semantics of a Shell script, we can extend that to computing the semantics of various *scenarios* – installation, update, removal, purge, etc. – for a given package. The second layer of our toolchain is based on the first one and is able to analyse packages and report on them. It is summarised in Figure 1.8.

In Debian, the installation of a package[28] is handled by the `dpkg` utility. Roughly speaking, for installation, `dpkg` calls the `preinst` script, then unpacks the static content, and finally calls the `postinst` scripts.[29] The precise sequence of script invocations and the actual script parameters are defined formally in the Debian Policy and described informally by flowcharts [22, Appendix 9]. The flowchart for the installation of a package is shown in Figure 1.9.

The tool `colis-package` [23] uses `colis-language` on the maintainer scripts and then composes their semantics to determine in which conditions the scenarios outputs can be reached and what transformation is performed on the filesystem in that case. In each of these possible output states, `colis-package` can

[27] An other team in the CoLiS project explores the use of tree transducers to provide this same formalism. We did not contribute to this line of work and it is therefore not going to be presented in this thesis.

[28] As well as all the other common operations: update, removal, purge, etc.

[29] For removal of a package, `dpkg` calls the `prerm` script, then removes the static content and finally calls the `postrm` script.

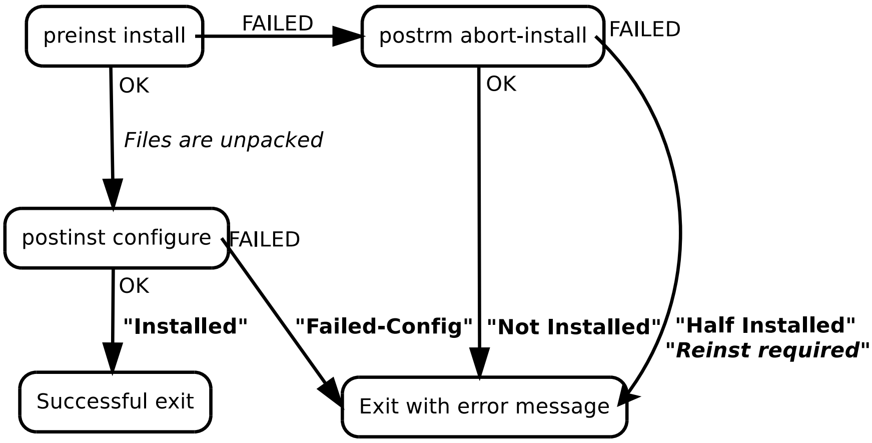


Figure 1.9: Flowchart for the installation of a package. Published in the Debian Policy [22, Appendix 9]

then check that the requirements given by the Debian Policy [22, Chapters 6 and 10] are satisfied. Finally, `colis-package` outputs a report on the given package.

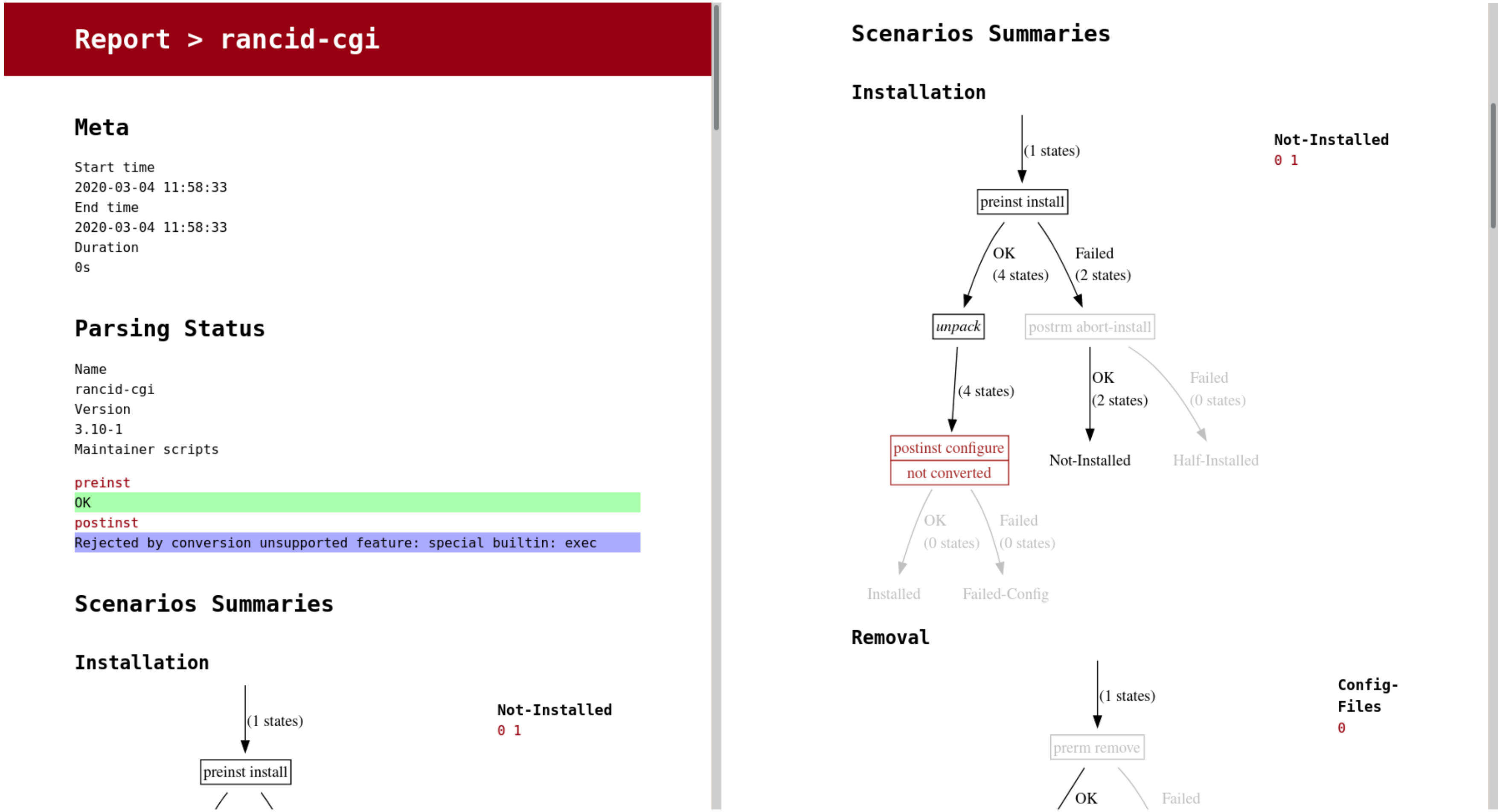
The results of `colis-package` running scenarios are presented as a set of HTML pages, including an index page giving data on the execution of the toolchain as well as a summary of all scenarios and a quick access to execution paths, a page for each maintainer script presenting its original Shell version and its converted CoLiS version if parsing and conversion were successful and a page for each execution path of each scenario, presenting the constraint corresponding to that path as well as debug traces to help us follow the execution path in the Shell or CoLiS script.

Let us go back to our running example. The HTML page reporting on `rancid-cgi` contains metadata about the time of the analysis (less than 1 second for such a simple package), the parsing status of maintainer scripts (two maintainer scripts, `preinst` and `posting`, the latter rejected by conversion because it uses an unsupported feature of the utility `exec`) and the list of scenarios as well as a quick access to their execution paths. A screenshot is available in Figure 1.10.

For the installation scenario, the HTML page includes the flowchart of Figure 1.9 with extra information about our specific package. We can read that the execution of the `preinst` script returned 6 states comprising 4 successes and 2 errors. The analysis of the 4 success states stops rapidly as the `postinst` script could not be converted and can therefore not be analysed. Since there is no `postrm` script, the 2 error cases go directly to the "Not-Installed" output state.

Reaching this "Not-Installed" output state is not per se a bug: it can be reasonable for a `preinst` script to cancel the installation before it takes place if some precondition is not satisfied. In such a case, the script should report on the error and, of course, leave the filesystem untouched.

In this case, one of the "Not-Installed" output state corresponds to the screenshot presented in Figure 1.11. The diagram represents the constraints on the input filesystem on the left and the resulting output filesystem on the right. We can read that this case happens when `/etc/rancid/lg.conf` exists in the input filesystem and is a symbolic link and `/etc/rancid/apache.conf` exists in the input filesystem and is a directory. In this case, the output filesystem is similar to the input one except for `/etc/rancid/lg.conf` that does not exist. All the rest, including `/etc/rancid/apache.conf`, remains the same. This error happens because the `rm` utility cannot remove directories if its `-r` argument is not specified. It is important to notice, here that, although we reach the "Not-Installed" case, there has been a modification of the filesystem. This can therefore be considered to be a bug. This bug has been reported to Debian in October



Report > rancid-cgi

Meta

```
Start time
2020-03-04 11:58:33
End time
2020-03-04 11:58:33
Duration
0s
```

Parsing Status

```
Name
rancid-cgi
Version
3.10-1
Maintainer scripts

preinst
OK
postinst
Rejected by conversion unsupported feature: special builtin: exec
```

Scenarios Summaries

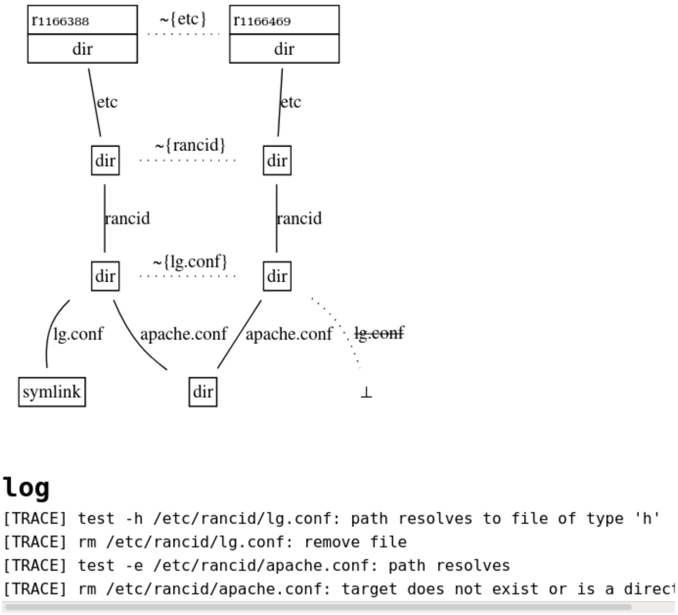
Installation

Scenarios Summaries

Installation

Removal

Figure 1.10: Report of colis-package on rancid-cgi – Index



```
[TRACE] test -h /etc/rancid/lg.conf: path resolves to file of type 'h'
[TRACE] rm /etc/rancid/lg.conf: remove file
[TRACE] test -e /etc/rancid/apache.conf: path resolves
[TRACE] rm /etc/rancid/apache.conf: target does not exist or is a direc
```

Figure 1.11: Report of `colis-package` on `rancid-cgi` – One output state

2019 [9] and, since then, fixed.

1.3.3 Third Layer – Several Packages

As of October 6, 2019, there are 12,592 packages that contain at least one maintainer script. Analysing them all by hand is therefore unreasonable and we need a tool helping us here. This is where the third layer of our toolchain, `colis-batch` [23], comes to play. It runs `colis-package` on several packages in parallel and outputs a report containing all the individual reports of the packages as well as a summary report. It is summarised in Figure 1.12. An example report is publicly available as a Zenodo archive [30].

The main page of the report contains information about the time taken for the full analysis, statistics about the number of scripts and their status with respect to the conversion, statistics about the number of scenarios ran in total and how many could be run without issues. The report then contains one page per scenario listing the packages that can reach each of the possible outcomes. A screenshot is shown in Figure 1.13. For instance, the page for the installation scenario of a corpus containing our example package `rancid-cgi` would say that at least one package reaches the “Not-Installed” state. A screenshot is shown in Figure 1.14. `rancid-cgi` would then show up in the list of non-installed packages and a link would lead to its individual report, described in Section 1.3.2.

1.4 Contributions & Plan of the Thesis

The main contributions of this thesis can be stated in the following list. The thesis itself is broadly organised according to these contributions.

- Our first contribution lies in the modelisation of Unix systems. This includes the modelisation of Unix filesystems as feature trees and transformation of Unix filesystems as formulas in FTS. This also includes the modelisation of Unix utilities as objects performing transformations of Unix filesystems using FTS. This modelisation is common work with other members of the CoLiS projects, but we

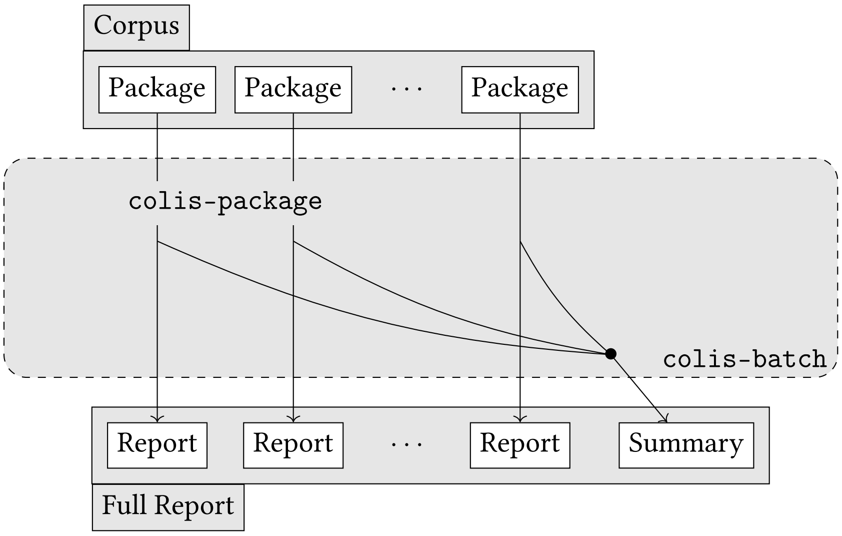
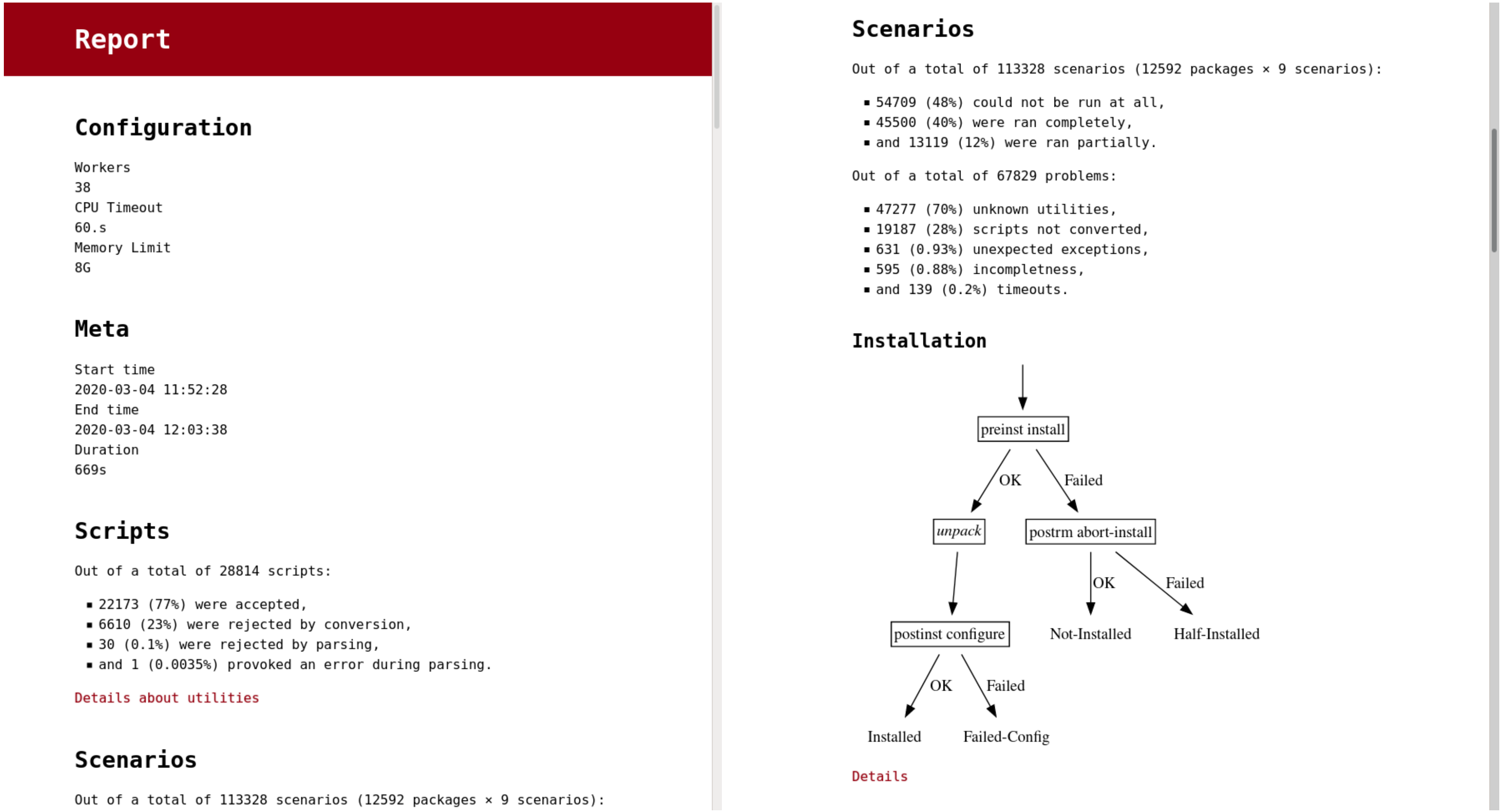


Figure 1.12: `colis-batch`: toolchain for the analysis of several Debian packages

contributed by providing the format as well as the specification of some utilities. This is described in Chapter 3.

- The main contribution of this thesis lies in decision procedures for FTS. This includes a theoretical dimension and more practical considerations. The theoretical approach results in a decision procedure for the first-order theory of FTS. The decidability of a feature tree logic with update constraints was open up to now. This decision procedure gives a result on the decidability of formulas of FTS, but its prohibitive complexity makes it unusable in practice. This theoretical dimension is the subject of Chapter 4. The practical considerations focus on designing a solver able to reason efficiently about a chosen restricted subset of formulas of FTS which are necessary for the rest of the project. This is described in Chapter 5.
- Another contribution lies in the modelisation of the Shell language. This results in the introduction of an intermediary language whose semantics is close to that of Shell but which avoids a lot of its pitfalls. This language comes with an automated conversion from a subset of Shell. This is described in Chapter 6.
- Finally, our contributions are scattered in a lot of places in the CoLiS project. They range from a formal interpretation of the POSIX standard [18] to contributions to the implementation of the toolchain mentioned in Section 1.3, while also spending time on ensuring quality on the software in the toolchain. This is described in Chapter 7.

We conclude with a word on the perspectives that remain open after this work in Chapter 8.



Report

Configuration

Workers
38
CPU Timeout
60.s
Memory Limit
8G

Meta

Start time
2020-03-04 11:52:28
End time
2020-03-04 12:03:38
Duration
669s

Scripts

Out of a total of 28814 scripts:

- 22173 (77%) were accepted,
- 6610 (23%) were rejected by conversion,
- 30 (0.1%) were rejected by parsing,
- and 1 (0.0035%) provoked an error during parsing.

Details about utilities

Scenarios

Out of a total of 113328 scenarios (12592 packages × 9 scenarios):

Scenarios

Out of a total of 113328 scenarios (12592 packages × 9 scenarios):

- 54709 (48%) could not be run at all,
- 45500 (40%) were ran completely,
- and 13119 (12%) were ran partially.

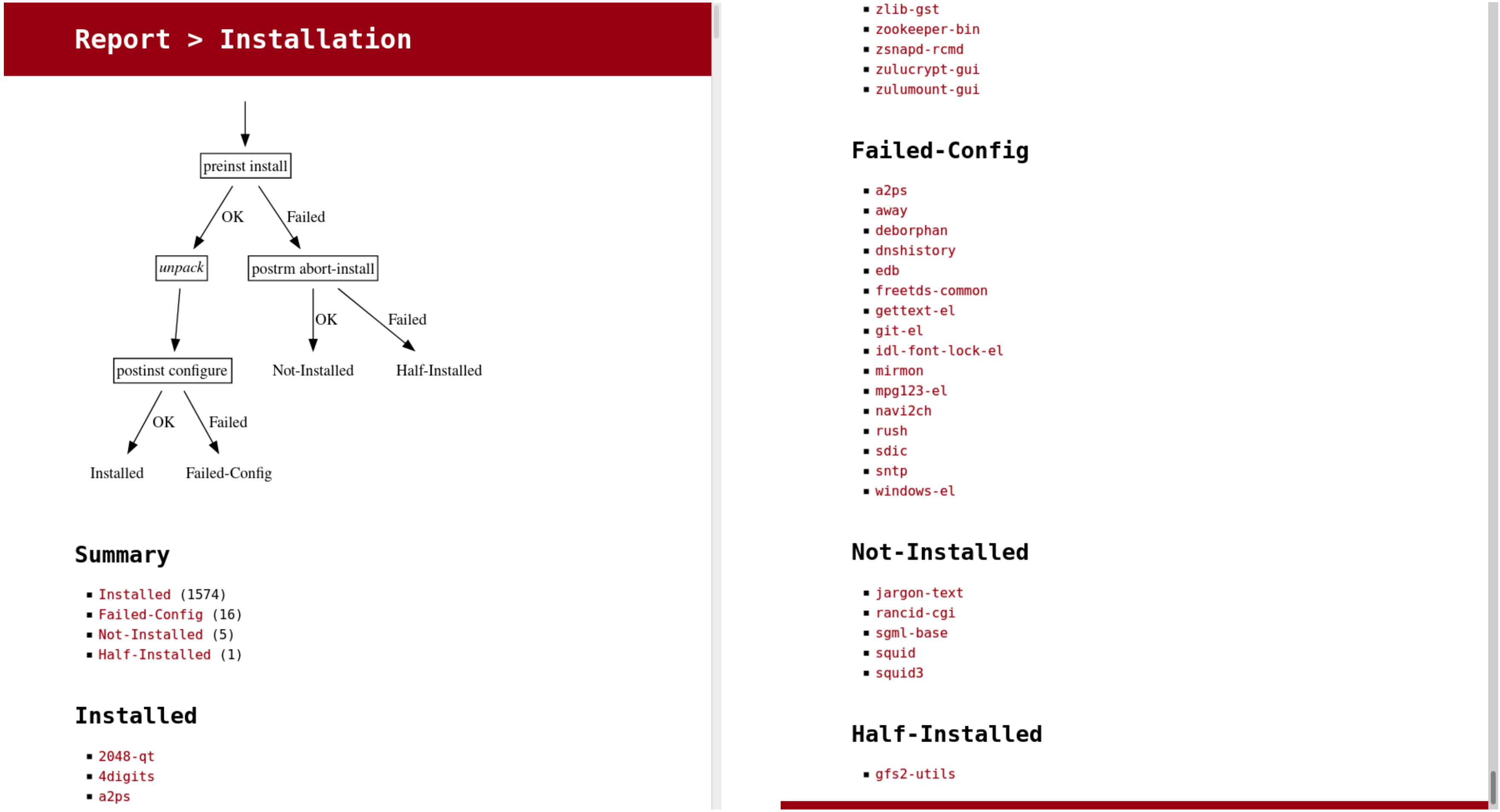
Out of a total of 67829 problems:

- 47277 (70%) unknown utilities,
- 19187 (28%) scripts not converted,
- 631 (0.93%) unexpected exceptions,
- 595 (0.88%) incompleteness,
- and 139 (0.2%) timeouts.

Installation

Details

Figure 1.13: Summary report by `colis-batch` – Index



Report > Installation

Summary

- Installed (1574)
- Failed-Config (16)
- Not-Installed (5)
- Half-Installed (1)

Installed

- 2048-qt
- 4digits
- a2ps
- zlib-gst
- zookeeper-bin
- zsnapd-rcmd
- zulucrypt-gui
- zulumount-gui

Failed-Config

- a2ps
- away
- deborphan
- dnshistory
- edb
- freetds-common
- gettext-el
- git-el
- idl-font-lock-el
- mirmon
- mpg123-el
- navi2ch
- rush
- sdic
- sntp
- windows-el

Not-Installed

- jargon-text
- rancid-cgi
- sgml-base
- squid
- squid3

Half-Installed

- gfs2-utils

Figure 1.14: Summary report by colis-batch – Page of the installation scenario

Chapter 2

Notations

This chapter describes the notations that we use in the whole document.

Sets

The empty set is noted $\varnothing$. If the universe is obvious from context, then we use $\star$ to represent the full set containing all the elements of the universe.

If A and B are two sets, then $A \cup B$ represents their union and $A \cap B$ their intersection.

If $\mathcal{A}$ is a finite set of sets, then $\bigcup \mathcal{A}$ represents the finite union of all the sets of $\mathcal{A}$ and $\bigcap \mathcal{A}$ their finite intersection. If $\mathcal{A}$ is empty, then $\bigcup \mathcal{A} = \varnothing$ and $\bigcap \mathcal{A} = \star$.

If `constr` is an element and A is a set, we note $\texttt{constr}(A)$ the set $\{(\texttt{constr}, e) \mid e \in A\}$.

Lists & Stacks

We use lists and stacks with the same notations. Lists are typically named l while stacks are typically named π.

For presentation purposes, we alternatively use lists from the left and from the right. Either way, the empty list is noted ε. If x is an element and l is a list, then $x \rangle l$ (resp. $l \langle x$) represents the list containing all the elements of l in the same order and starting (resp. ending) by x. For instance, if l is a list from the right containing the elements 1, 2 and 3, 1 being at the head of the list, we have $l = \varepsilon \langle 3 \langle 2 \langle 1$. If l_1 and l_2 are two lists, then $l_1 \rangle\rangle l_2$ (resp. $l_2 \langle\langle l_1$) represents the list containing all the elements of l_1 in the same order followed (resp. preceded) by all the elements of l_2 in the same order.

If A is a set, then A^+ (resp. A^*) represents non-empty (resp. potentially empty) lists of elements of A.

Functions

The domain of a function f is noted $\texttt{dom}(f)$.

If A and B are sets, $A \to B$ represents the set of functions from A to B, that is the set of functions whose domain is A and whose image is included in B. $A \rightsquigarrow B$ represents the set of partial functions from A to B, that is the set of functions from a subset of A to B. The subset in question can be infinite.

```
1  function sort(l : list of integers) : list of integers
2    match l
3    | ε -> return ε
4    | x › l′ ->
5      let l″ = sort(l′)
6      return insert(x, l″)
7
8  function insert(x : integer, l : list of integers) : list of integers
9    match l
10   | ε -> return x › ε
11   | x′ › l′ ->
12     if x < x′
13       return x › x′ › l′
14     else
15       return x′› insert(x, l′)
```

Figure 2.1: An example program in pseudo code

If x_1, x_2, v_1, v_2, etc. are elements, then $[x_1 \mapsto v_1, x_2 \mapsto v_2, \dots]$ represents the function whose domain is $\{x_1, x_2, \dots\}$ and that associates v_1 to x_1, v_2 to x_2, etc. $f[x_1 \mapsto v_1, x_2 \mapsto v_2, \dots]$ represents the function whose domain is $\texttt{dom}(f) \cup \{x_1, x_2, \dots\}$ and that associates v_1 to x_1, v_2 to x_2, etc. and $f(x)$ to every x that is not x_1, x_2, etc.

Two functions f and g are equal, noted $f = g$, if $\texttt{dom}(f) = \texttt{dom}(g)$ and for all $x \in \texttt{dom}(f)$, $f(x) = g(x)$. This is the intentional equality of functions. Two functions f and g are equal on a set A, noted $f =_A g$, if $\texttt{dom}(f) \cap A = \texttt{dom}(g) \cap A$ and for all $x \in \texttt{dom}(f) \cap A$, $f(x) = g(x)$.

Pseudo Code

We often present code as a way to support an explanation or describe an algorithm. For this purpose, we use pseudo-code presenting a functional language strongly influenced by OCaml. An example can be found Figure 2.1. Our pseudo code features:

- functions, including mutually recursive functions (Line 1),
- types (Line 1), including option types of the form `type or ⊥`,
- pattern-matching (Line 2),
- let bindings (Line 5), although we avoid the use of shadowing,
- and conditional statements (Line 12).

We use syntax colouring as a way to make our code more readable. In no way is this colouring necessary for the understanding of the semantics of our programs.

- `keywords` are red,
- `functions` are blue,
- `types` are green.[1]

We sometimes show programs in other (existing) languages. They share the same syntax colouring except they do not feature types. They sometimes show `strings` in green and `variables` in gold.

[1] ...and so should you? [31]

Chapter 3

Modelisation of Unix Filesystems and Utilities

Let us dive into the modelisation of Unix filesystems and utilities. In Section 3.1, we first describe Unix filesystems and feature trees and explain how and why we can model the former using the latter. We then introduce, in Section 3.2, the logic FTS which allows us to represent relations between these feature trees, and therefore model filesystem transformations. In Section 3.3, we describe Unix utilities and how we can see them as objects performing filesystem transformations, and therefore how we can model them using FTS. Finally, in Section 3.4, we discuss some properties that we can expect from the formal specifications of Unix utilities and state what we need from decision procedures in order to reason about such specifications.

3.1 Modelisation of Filesystems

Let us start by describing Unix filesystems and how we model them in this work. In Section 3.1.1, we describe various aspects Unix filesystems. In Section 3.1.2, we explain which aspects can be abstracted away in our work, and why. Finally, in Section 3.1.3, we define feature trees and discuss their pertinence as models of filesystems.

3.1.1 Filesystems

A *filesystem* is a hierarchical structure used to store and find data efficiently. Filesystems can be found in storage devices: hard disk drives, solid-state drives, magnetic tapes, optical discs, USB sticks, etc. Some can also be found outside of permanent storage devices. This is in particular the case of the temporary filesystems stored directly in the computer's main memory (RAM) or of the ones accessible via a network protocol. In this work, we are interested in Unix filesystems and we will only describe these in the rest of this subsection.

In such filesystems, everything is a *file*[1]. The most common kind of files are called *regular files*. They are used to store all kind of permanent data. There are many other kinds of files: *pipes*, *sockets*, *block* and *character device* files, *symbolic links*, etc. All of these are stored inside *directories*[2] that are also considered to be files. A whole filesystem is thus one *root* directory containing other directories and files recursively.

[1] See `https://en.wikipedia.org/wiki/Everything_is_a_file` for the "everything is a file" quote.

[2] Directories are also often called *folders*. We will only use the former.

Although hierarchical, a filesystem is not necessarily a tree but rather a directed acyclic graph. This will be explained in details later.

From a directory, one accesses other files by their *file name*. File names can be any string that does not contain / and that is neither `.` or `..`. We will note by $\mathcal{F}$ the set of allowed file names. One dot, `.`, denotes the *current directory* itself. Two dots, `..`, denotes the parent directory, that is the directory that contains the current directory. The parent of the root is the root itself.

The list of file names that lead to a file is called the *path* of that file. Paths can also include the current directory and the parent directory. They can be absolute and relative. Absolute paths are interpreted from the root of the filesystem. Relative paths are interpreted relatively to a *current working directory*.

Definition 3.1 (Abstract Syntax Path)**.** Given a set $\mathcal{F}$ of allowed file names, the set of *path components* is $\mathcal{PC} = \mathcal{F} \cup \{\texttt{.}, \texttt{..}\}$. The two path kinds are `abs` and `rel`. The set of *paths* is defined as

$$\mathcal{P} = \texttt{abs}(\mathcal{PC}^*) \cup \texttt{rel}(\mathcal{PC}^+)$$

where $\mathcal{PC}^+$ is a non-empty list of path components and $\mathcal{PC}^*$ is a possibly empty list of path components.

A path is *absolute* if it is in $\texttt{abs}(\mathcal{PC}^*)$. A path is *relative* if it is in $\texttt{rel}(\mathcal{PC}^+)$. A path is *linear* if it does not contain `.` or `..`. A path is *normal* if it is absolute and linear. □

Note that relative paths cannot be empty while absolute paths can. The empty absolute path represents the root. Paths are written as the list of their path components separated by /. Absolute paths start with /, relative paths do not. For instance, the normal path of the file `ocaml` in the directory `lib` in the directory `usr` at the root is noted `/usr/lib/ocaml`. The absolute path `/usr/../usr/lib/./ocaml` would point to the same file. The relative path `../lib/ocaml`, when considered from the directory `/usr/share`, would also point to the same file.

As we mentioned earlier, filesystems are not Herbrand-style trees. This comes from two reasons. Firstly, the file names belong to the directory that contains the files and not to the files themselves. In other words, they are on the *edges* of the filesystem and not on the nodes. This means that the same file can be present with different names in the same directory. It can also be present in a different directory anywhere else in the filesystem. This introduces sharing in filesystem structures: the same file can have several normal paths. Such alternative accesses for the same file are called *hard links*. They are forbidden on directories so as to avoid creating cycles.[3]

Secondly, filesystems can contain symbolic links. They are files that contain a path, absolute or relative. One can then access a symbolic link and must follow the path it contains to get to the file it refers to. For instance, if `/usr/local/lib` is a symbolic link containing `../lib`, then `/usr/local/lib/ocaml` would point to the same file as `/usr/lib/ocaml`. These symbolic links differ from hard links in that they may point to directories, thus creating cycles. They may also point to non-existing files: nothing guarantees that the path they contain is valid in the filesystem.

In order to better understand the notion of path and its interpretation, let us consider Figure 3.1. It contains a pseudo-code version of the resolution of a path in a filesystem, for explanatory purposes. It defines a function `resolve`, Line 24, which takes a filesystem `fs`, a normal path containing the current directory `cwd` and a path p to resolve. It returns either a normal path corresponding to the resource p points to, or an error if the path is not valid. `resolve` only matches on the given path p to get its list of path components q

[3] Historically, some systems have tried allowing hard linking to directories. This has been abandoned in most current systems to prevent loops in filesystems and to keep the interpretation of the parent directory consistent. It still has some infrequent uses, for instance in *Mac OSX*'s *Time Machine* backup mechanism.

```
function file-kind (fs : filesystem, cwd : normal path, $f$ : file name)

function resolve-pc
    (fs : filesystem, cwd : normal path,
     $q$ : list of path components) : normal path or error
  match $q$
  | $\varepsilon$ -> return cwd
  | . › $q'$ -> return resolve-pc(fs, cwd, $q'$)
  | .. › $q'$ ->
    match cwd
    | / -> return resolve-pc(fs, /, $q'$)
    | cwd' ‹ _ -> return resolve-pc(fs, cwd, $q'$)
  | $f$ › $q'$ ->
    match file-kind(fs, cwd, $f$)
    | directory -> return resolve-pc(fs, cwd ‹ $f$, $q'$)
    | symlink to abs($q''$) -> return resolve-pc(fs,  /, $q''$ » $q'$)
    | symlink to rel($q''$) -> return resolve-pc(fs, cwd, $q''$ » $q'$)
    | other kind ->
      match $q'$
      | $\varepsilon$ -> return cwd ‹ $f$
      | _ -> return error
    | no such file -> return error

function resolve (fs : filesystem, cwd : normal path, $p$ : path)
    : normal path or error
  match $p$
  | abs($q$) -> return resolve-pc(fs,  /, $q$)
  | rel($q$) -> return resolve-pc(fs, cwd, $q$)
```

Figure 3.1: Resolution of a path in a filesystem

and to know whether resolution should start from the root – if p is absolute – or from the current directory – if p is relative. In both cases, the core of the resolution is left to `resolve-pc`. `resolve-pc`, Line 3 takes a filesystem `fs`, a current directory `cwd` and a list of path components q and returns a normal path or an error. It works as follows:

- If the given list of path components q is empty (Line 7), there is nothing to resolve and the current directory `cwd` is the resource we have been resolving. Otherwise, we consider the first path component of the list. The rest of the list will be named q'.
- If the first path component of q is `.` (Line 8), we do not move and we continue resolving the rest of the path from the same directory.
- If the first path component of q is `..` (Line 9), we have to remove the last path component of `cwd`. If `cwd` is the root, there is nothing to remove and we continue resolving from the same place (Line 11). Otherwise, we remove the last path component and continue resolving from there (Line 12).
- Otherwise, if the first path component of q is a file name f (Line 13), we query the filesystem to see what is to be found in `cwd` at the file name f. For that, we assume the existence of a function `file-kind` that returns the kind of our file if it exists.[4] We leave the function `file-kind` unspecified in this example.
- If the file in question exists and is a directory (Line 15), we can continue resolving from this directory: the name f is added at the end of `cwd`.
- If the file exists and is a symbolic link (Lines 16 and 17), we append its path components q'' in front of the list of path components that remain to be resolved q'. We start the new resolution from the root or the current directory depending on whether the link is symbolic or not.
- If the file exists but is another kind of file (Line 18), we have two possibilities. If there does not remain anything to resolve after this, then the file in question is the resource we have been resolving and we can return `cwd` to which we add the file name (Line 20).
- If the file exists but there remains things to resolve, or if the file does not exist, then the resolution fails (Lines 21 and 22).

Finally, all files carry additional metadata. In other words, the *nodes* of a filesystem carry various bookkeeping information. The minimal information that must be carried by a node in a POSIX filesystem [19, 13. Headers, `<sys/stat.h>`] is the following:

- the *user* and *group* owning the file;
- the *mode*, containing the *permissions* stating who can read, write or execute the files;
- the *inode* and the *device id* containing the file;
- three *timestamps* describing when they have been accessed or modified last, or when their status has been changed last;
- in case of regular files, the *size*: the number of bits they occupy on the storage medium;
- the number of hard links – incoming edges – to them;

Some extensions are already described in the POSIX standard and, technically, any other is possible.

[4] This function is not present in that way in Unix. A similar functionality is provided by the `stat()` system call [21, 3. System Interfaces, `stat()`].

3.1.2 Abstracting Away from the Filesystem

Filesystems contain a lot of information. However, we are not necessarily interested in modelling all of them for our work. As mentioned in Section 1.2, we are guided in our modelisation by (with increasing order of importance:

- the POSIX standard [18], a document that describes what can be expected from Unix systems in general and filesystems and Shell in particular;
- the Debian Policy [22], a document that describes what can be expected on Debian systems in general and what should and should not be in maintainer scripts in particular;
- and a statistical analysis that we did ourselves (see Section 7.1), that allows us to measure what is actually used in Debian maintainer scripts.

The Debian Policy, for instance, states that maintainer scripts will be executed as the *root* user with maximum privileges. This level of privileges allows to ignore information of owner, group and permissions. We can thus safely abstract away from these in our model. Our statistical analysis [25; Jeannerod et al. 2017b] shows us that the other metadata – the timestamps[5], the file size, etc. – are not used in maintainer scripts. It is thus safe to remove them too from our abstraction.

For similar reasons, we do not consider the content of regular files. This restriction allows us in particular to ignore any problem related to sharing – i.e. to files being reachable from different places in the filesystem – and therefore to ignore hard links. Indeed, hard links only matter because the modification of a file will affect all the places in the filesystem where it is located. Since we do not model the content of files, this problem goes away and we are allowed to see filesystems as trees.

The statistical analysis also shows that the file kind does not matter: we are only interested in the difference between directories and other kind of files. In fact, handling several file kinds does not add any complexity to the modelisation. For presentation purposes, in all the rest of this chapter as well as in Chapters 4 and 5, we will ignore file kinds altogether, as if all files were directories.

Finally, the handling of symbolic links is known to be a complicated problem. There exists research focusing on this topic [Ntzik & Gardner 2015]. Fortunately for us, they are rarely present in maintainer scripts. We have thus decided to ignore them and to put our focus on other aspects of the modelisation of filesystem relations instead. Technically, this means that there are some scripts that are going to be modelled incorrectly; the potential bugs we find therefore can only be false positives and have to be validated a posteriori. The fact that they are rarely present in maintainer scripts means that we will only encounter a low number of such false positives.

3.1.3 Feature Trees

Feature trees are trees of unbounded depth where nodes have an unbounded number of children and where edges from nodes to their children carry names such that no node has two different outgoing edges with the same name. Hence, the names on the edges can be used to select the different children of a node. In an abstraction like ours where a filesystem can be seen as a tree, feature trees happen to be adequate representations of such trees.

We assume given an infinite set of *features* $\mathcal{F}$. It is used to model the allowed file names in a filesystem. In the rest of this document, we use the letters f, g, h to denote features. We can now give the definition of feature trees in Definition 3.2.

[5] Besides, on a lot of current systems, the default is to disable the update of the access time of files. Some others update it but not systematically.

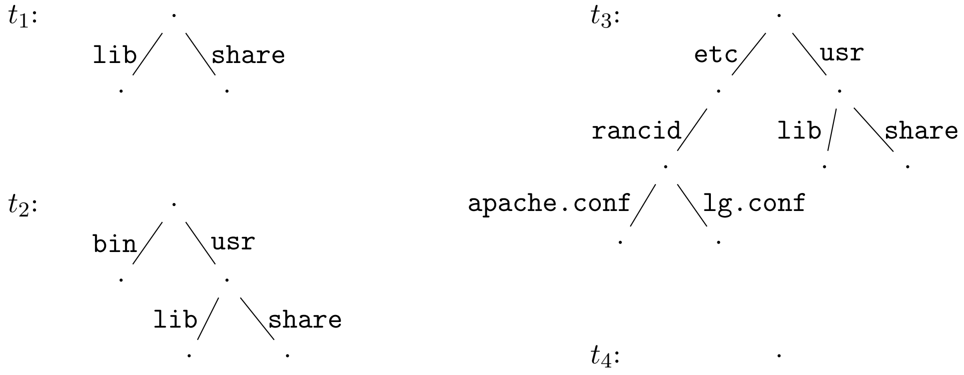


Figure 3.2: Examples of feature trees

Definition 3.2 (Feature Trees)**.** The set $\mathcal{FT}_\infty$ of *feature trees* is coinductively defined as:

$$\mathcal{FT}_\infty = \mathcal{F} \rightsquigarrow \mathcal{FT}_\infty$$

A *finite feature tree* is an inhabitant of $\mathcal{FT}_\infty$ that is finite. We will note by $\mathcal{FT}$ the set of finite feature trees. In other words, it is *inductively* defined as:

$$\mathcal{FT} = \mathcal{F} \rightsquigarrow \mathcal{FT}$$

where all the partial functions have finite domain. Here, the case of a partial function with empty domain serves as base case of the induction. □

We believe it is important to note that our intuition is of a persistent, extensional world which we manipulate with the intentional equality only (see Chapter 2). The intentional equality gives us equivalence classes containing all the various ways[6] to represent the same objects, as trees or as directed graphs. Since the world is persistent, and since we only use the intentional equality, we can allow ourselves to see all the objects as trees[7]. A reader whose intuition is of an intentional world might be disturbed by Definition 3.2 because they will see the objects of $\mathcal{FT}$ as directed acyclic graphs, while we systematically call them trees in this thesis. These two intuitions, however, give the same results: the latter simply manipulates as objects what the former sees as equivalence classes of all the ways these objects can be represented.

Note that, even if all the trees in $\mathcal{FT}$ are finite, their depths and their widths are unbounded[8]. In this work, we are going to consider most of the time finite feature trees. Unless explicitly mentioned, all the feature trees in the rest of this document are finite, that is unless explicitly mentioned, we place ourselves in the model $\mathcal{FT}$.

Figure 3.2 shows examples of feature trees. t_1 is a tree whose root has two outgoing edges lib and share both leading to a tree whose root has no outgoing edges. Formally, t_1 is a partial function of domain $\{\texttt{lib}, \texttt{share}\}$ and such that $t_1(\texttt{lib}) = t_1(\texttt{share}) = t_4$, where t_4 is the empty feature tree. One can notice that $t_1 = t_2(\texttt{usr})$, that $t_2(\texttt{usr}) = t_3(\texttt{usr})$ and that $t_1(\texttt{usr}) = t_4$[9].

[6] And there is always an infinity.

[7] This intuition comes directly from programming and persistent data structures: we know that there is an infinity of ways to organise them on the heap, yet we abstract away from that by using structural equality. The equality is not able to make the difference between directed graphs and trees and we therefore can allow ourselves to think of everything as trees.

[8] If $\mathcal{F}$ is a finite set, then the width of trees in $\mathcal{FT}$ is of course trivially bounded.

[9] Among plenty other seemingly interesting facts, but our goal here is not to list them all.

Feature trees have been used previously in constraint-based formalisms in the field of computational linguistics [Smolka 1992]. They have then been introduced as record-like data structures in constraint logic programming [Aït-Kaci et al. 1994; Smolka & Treinen 1994] The use of record-like structures in logic programming languages, under the form of ψ-terms [Aït-Kaci 1986], was introduced by the languages LOGIN [Aït-Kaci & Nasr 1986] and LIFE [Aït-Kaci & Podelski 1993]. Shortly later, the language Oz [Smolka et al. 1993; Smolka 1995] used a feature constraint system whose semantics was directly based on feature trees.

3.2 Logic Over Feature Trees – FTS

We now have a model of filesystems as feature trees. What we do want to describe in this work, however, is filesystem transformations, as they are what Unix utilities and Shell scripts perform. We thus need a formalism able to describe transformations of feature trees. In particular, we need to be able to describe an update from one tree to another one and to express facts resembling "y is an update of x where f now leads to an empty directory". Feature tree logics have been studied before and seem quite appropriate to describe the trees we are interested in; they just need to be extended to express updates of trees.

In this section, we define a feature tree logic FTS[10] that corresponds to these needs. In Section 3.2.1, we define the syntax of the formulas of FTS and give examples. In Section 3.2.2, we give the semantic interpretation of these formulas and define basic semantic notions. In Section 3.2.3, we discuss the expressivity of FTS in comparison to related works. Finally, in Section 3.2.4, we define classes of formulas that will be used in this document.

3.2.1 Syntax of FTS

We assume given an *algebra of sets over* $\mathcal{F}$, noted $\mathcal{FS}$. This is, by definition, a subset of $\mathcal{P}(\mathcal{F})$ containing the empty set, all the singletons of features, and stable by union, intersection and complement. It thus contains all the finite and cofinite[11] sets of features, including $\mathcal{F}$ itself. We will note $\star$ for the full set, that is $\mathcal{F}$. We require a test of finiteness, a test of membership ($\in$) and a test of inclusion ($\subseteq$). In particular, we can test whether a set F is empty ($F \subseteq \varnothing$), or full ($\star \subseteq F$). Typically, we could restrict ourselves to only finite and cofinite sets, but we could also consider regular languages of features, for instance.

We consider a first-order logic over feature trees. We assume given an infinite supply $\mathcal{V}$ of variables all distinct from features of $\mathcal{F}$. We will use x, y, z to denote these variables. Let us give the syntax of logic formulas of FTS in Definition 3.3.

Definition 3.3 (Syntax of FTS)**.** The logical formulas ϕ, ψ, etc. are defined inductively as:

[10] When I checked on the Internet whether the initials "FTS" were already in use, I discovered that they were... by a company named `FrenchTouchSeduction.com`. That is when I knew I had made the good choice.

[11] If $\mathcal{F}$ is finite, then every subset of $\mathcal{F}$ is both trivially finite and cofinite.

$$\begin{array}{rcll}
\phi, \psi, \ldots & ::= & x[f]y & \text{— Feature } f \text{ from } x \text{ to } y \\
& \mid & x[F]\uparrow & \text{— Absence of } F \text{ from } x \\
& \mid & x =_F y & \text{— Similarity of } x \text{ and } y \text{ on } F \\
& \mid & \top & \text{— True} \\
& \mid & \bot & \text{— False} \\
& \mid & \neg\phi & \text{— Negation of } \phi \\
& \mid & \phi \wedge \psi & \text{— Conjunction of } \phi \text{ and } \psi \\
& \mid & \phi \vee \psi & \text{— Disjunction of } \phi \text{ and } \psi \\
& \mid & \exists x \cdot \phi & \text{— Existential quantification of } x \text{ in } \phi \\
& \mid & \forall x \cdot \phi & \text{— Universal quantification of } x \text{ in } \phi
\end{array}$$

where f is a feature from $\mathcal{F}$ and F is a feature set from $\mathcal{FS}$.

We consider the similarity predicate to be symmetrical: we identify $x =_F y$ with $y =_F x$. We consider conjunction to be associative and commutative. Associativity means that we identify $\phi \wedge (\psi_1 \wedge \psi_2)$ with $(\phi \wedge \psi_1) \wedge \psi_2$ and we write $\phi \wedge \psi_1 \wedge \psi_2$ in that case. Commutativity means that we identify $\phi \wedge \psi$ with $\psi \wedge \phi$. Similarly, we consider disjunction to be associative and commutative. We consider existential quantification to be commutative: we identify $\exists x \cdot \exists y \cdot \phi$ with $\exists y \cdot \exists x \cdot \phi$ and we write $\exists x, y \cdot \phi$. Similarly, we consider universal quantification to be commutative.

We write $x[f_1 \ldots f_n]\uparrow$ for $x[\{f_1 \ldots f_n\}]\uparrow$. We write $x \neq_F y$ for $\neg(x =_F y)$. We use the shortcuts $\phi \rightarrow \psi$ for $\neg\phi \vee \psi$, and $\phi \leftrightarrow \psi$ for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. □

Note that FTS only has quantification over variables. It does not, in particular, allow quantification over features or sets of features. Variables that appear in an existential or universal quantifier are said to be *bound*. The others are called *free variables*. We define free variables in Definition 3.4.

Definition 3.4 (Free Variables of a Formula)**.** The *free variables of a formula* ϕ, noted $\mathcal{V}(\phi)$ are defined as follows:

$$\begin{array}{rcl}
\mathcal{V} : \text{formulas} & \rightarrow & \text{sets of variables} \\
\top,\ \bot & \mapsto & \varnothing \\
x[F]\uparrow & \mapsto & \{x\} \\
x[f]y,\ x =_F y & \mapsto & \{x,\ y\} \\
\neg\phi & \mapsto & \mathcal{V}(\phi) \\
\phi \wedge \psi,\ \phi \vee \psi & \mapsto & \mathcal{V}(\phi) \cup \mathcal{V}(\psi) \\
\exists x \cdot \phi,\ \forall x \cdot \phi & \mapsto & \mathcal{V}(\phi) \backslash \{x\}
\end{array}$$

A formula ϕ that has no free variables (that is such that $\mathcal{V}(\phi) = \varnothing$) is said to be *closed.* □

Any formula can be closed by adding as many existential or universal quantifiers as necessary. These closures are defined in Definition 3.5. Note that closed formulas are syntactically equal to both their existential and universal closures.

Definition 3.5 (Existential and Universal Closures)**.** The *existential closure* of a formula ϕ, written $\tilde{\exists} \cdot \phi$, is the formula $\exists \mathcal{V}(\phi) \cdot \phi$. The *universal closure* of a formula ϕ, written $\tilde{\forall} \cdot \phi$, is the formula $\forall \mathcal{V}(\phi) \cdot \phi$. □

Figure 3.3 shows example formulas in FTS. Let us describe informally their meaning in natural language.

- Formula 3.1 states that there is a feature `lib` from x to y and that everything is absent from y (in other words, y is empty).

$$x[\mathtt{lib}]y \wedge y[\star]\uparrow \tag{3.1}$$
$$\exists y \cdot (x =_{\{\mathtt{usr},\mathtt{var}\}} y \wedge x[\mathtt{etc}]\uparrow \wedge \neg y[\mathtt{etc}]\uparrow) \tag{3.2}$$
$$\forall x \cdot ((x =_F y \wedge x[G]\uparrow) \rightarrow y[F \cap G]\uparrow) \tag{3.3}$$
$$\exists x, y \cdot \forall z \cdot (z \neq_F x \vee z \neq_F y) \tag{3.4}$$

Figure 3.3: Examples of formulas

- Formula 3.2 states that there exists a y such that x and y are similar in $\{\mathtt{usr}, \mathtt{var}\}$, and that $\mathtt{etc}$ is absent from x and not absent from y (in other words, $\mathtt{etc}$ is present in y).
- Formula 3.3 states that, for all x, if x and y are similar in F, and all the features of G are absent from x, then all the features of $F \cap G$ are absent from y.
- Formula 3.4 states that there exists x and y such that for all z, either x and z are not similar in F, or y and z are not similar in F (in other words, z cannot be similar in F with both x and y).

The formalisation of the meaning of these formulas is the subject of Section 3.2.2.

3.2.2 Semantics of FTS

Let us now define what meaning to give to a formula of FTS defined in Section 3.2.1. We first need to define *models* in Definition 3.6 and *valuations to models* in Definition 3.7.

Definition 3.6 (Model)**.** A model is a set of feature trees, that is a subset of $\mathcal{FT}_\infty$. □

Definition 3.7 (Valuation to a model)**.** Given a model M, a *valuation* to M is a partial function from variables to M. We use ρ to denote such functions. The set of all valuations to M will be noted R_M. □

We can then define the *interpretation of a formula ϕ in a model M* in Definition 3.8. The interpretation of ϕ in M is a set of valuations to M – that is a subset of R_M – thanks to which the formula makes sense. We define this in any model although we will place ourselves in the specific model $\mathcal{FT}$ most of the time.

Definition 3.8 (Interpretation of a formula in a model)**.** The *interpretation* of a formula ϕ in a model M, noted $\mathcal{I}_M(\phi)$, is:

$$\begin{array}{rcl}
\mathcal{I}_M : \text{formulas} & \rightarrow & \mathcal{P}(R_M) \\
x[f]y & \mapsto & \{\rho \in R_M \mid x, y \in \mathtt{dom}(\rho), f \in \mathtt{dom}(\rho(x)), \rho(x)(f) = \rho(y)\} \\
x[F]\uparrow & \mapsto & \{\rho \in R_M \mid x \in \mathtt{dom}(\rho), \mathtt{dom}(\rho(x)) \cap F = \varnothing\} \\
x =_F y & \mapsto & \{\rho \in R_M \mid x, y \in \mathtt{dom}(\rho), \rho(x) =_F \rho(y)\} \\
\top & \mapsto & R_M \\
\bot & \mapsto & \varnothing \\
\neg\phi & \mapsto & R_M \backslash \mathcal{I}_M(\phi) \\
\phi \wedge \psi & \mapsto & \mathcal{I}_M(\phi) \cap \mathcal{I}_M(\psi) \\
\phi \vee \psi & \mapsto & \mathcal{I}_M(\phi) \cup \mathcal{I}_M(\psi) \\
\exists x \cdot \phi & \mapsto & \{\rho \in R_M \mid \exists t \in M \cdot \rho[x \mapsto t] \in \mathcal{I}_M(\phi)\} \\
\forall x \cdot \phi & \mapsto & \{\rho \in R_M \mid \forall t \in M \cdot \rho[x \mapsto t] \in \mathcal{I}_M(\phi)\}
\end{array}$$

We note $\rho \models_M \phi$ when $\rho \in \mathcal{I}_M(\phi)$. We say that ρ *satisfies* the formula ϕ in the model M. We note $\rho \models \phi$ when ρ satisfies ϕ in all models. □

As a reminder, the definition of the equality $=$ and partial equality $=_F$ for functions is given in Chapter 2. Considering how $\mathcal{FT}$ is defined, $=$ and $=_F$ correspond to the observational equality and the observational partial equality.

Depending on whether $\mathcal{I}_M(\phi)$ contains zero, more or all of the valuations of R_M, we say that ϕ is *unsatisfiable*, *satisfiable* or *valid* respectively. The formal definition of these terms is given in Definition 3.9.

Definition 3.9 (Satisfiability)**.** A formula ϕ is *unsatisfiable* in a model M if there does not exist any valuation $\rho \in R_M$ such that $\rho \models_M \phi$. A formula ϕ is *satisfiable* in a model M if there exists a valuation $\rho \in R_M$ such that $\rho \models_M \phi$. A formula ϕ is *valid* in a model M if, for all valuations $\rho \in R_M$, $\rho \models_M \phi$. □

Note that a valid formula is also satisfiable. Note also that, for closed formulas, the valuations do not matter. In other words, if one valuation satisfies a closed formula, then all the other valuations do too. Satisfiability and validity are then the same. If ϕ is a closed formula, we write $\models_M \phi$ and $\models \phi$ if ϕ is satisfiable/valid in M and all models respectively. Satisfiability and validity have a close relationship with existential and universal closures, as stated in Lemma 3.1.

Lemma 3.1 (Satisfiability and Validity and Existential and Universal Closures)**.** *A formula ϕ is satisfiable in a model M if and only if its existential closure is satisfiable/valid in M. In other words, ϕ is satisfiable in M if and only if $\models_M \tilde{\exists} \cdot \phi$. A formula ϕ is valid in a model M if and only if its universal closure is satisfiable/valid in M. In other words, ϕ is valid in M if and only if $\models_M \tilde{\forall} \cdot \phi$.*

Note also that these notions of satisfiability and validity differ from the standard ones in that there is here only one model being considered. The satisfiability is about the existence of a *valuation* that satisfies the formula *in that one model*. The validity is about the fact that all *valuations* satisfy the given formula *in that one model*.

Finally, let us define *implication* and *equivalence* of formulas in Definitions 3.10 and 3.11.

Definition 3.10 (Implication)**.** A formula ϕ *implies* ψ in a model M if $\models_M \tilde{\forall} \cdot (\phi \rightarrow \psi)$. In other words, ϕ implies ψ if, for all valuation ρ, if $\rho \models_M \phi$, then $\rho \models_M \psi$. □

Definition 3.11 (Equivalence)**.** Two formulas ϕ and ψ are *equivalent* in a model M if $\models_M \tilde{\forall} \cdot (\phi \leftrightarrow \psi)$. In other words, ϕ and ψ are equivalent if, for all valuation ρ, $\rho \models_M \phi$ if and only if $\rho \models_M \psi$. To put it yet another way, ϕ and ψ are equivalent if ϕ implies ψ and ψ implies ϕ. □

Although these definitions are generic with respect to the model, all our work is placed in the one model of all feature trees $\mathcal{FT}$. We will thus simply say that a valuation satisfies a formula, leaving the model of feature trees implicit. Similarly, we will say for instance that two formulas are equivalent, leaving the model of feature trees implicit. On the other hand, when required, we will make it explicit if we do not talk only of $\mathcal{FT}$.

Let us now interpret (in $\mathcal{FT}$) the formulas of Figure 3.3 by using the trees of Figure 3.2:

- Formula 3.1 is a formula with two free variables x and y. It is clearly not valid. Indeed, the valuation $[x \mapsto t_4, y \mapsto t_4]$ that gives the empty tree – t_4 – to both x and y does not satisfy it. It is however satisfiable. Indeed, the valuation $[x \mapsto t_1, y \mapsto t_4]$ that to gives t_1 to x and t_4 to y satisfies it.
- Formula 3.2 has one free variable x – y being bound by an existential quantifier. It is not a valid formula as any valuation ρ such that $x \in \mathtt{dom}(\rho)$ and $\mathtt{etc} \in \mathtt{dom}(\rho(x))$ is not in its interpretation. It

$$\begin{array}{rcl}
x \doteq y & \rightsquigarrow & x =_{\star} y \\
x[f]y & \rightsquigarrow & x[f]y \\
x[f]\uparrow & \rightsquigarrow & x[\{f\}]\uparrow \\
x[F] & \rightsquigarrow & x[{}^{c}F]\uparrow \\
x \sim_F y & \rightsquigarrow & x =_{{}^{c}F} y
\end{array}$$

Figure 3.4: Conversion from FT extended with fence and similarity to the logic presented in this work

$$\begin{array}{rlcl}
x[f]y & & \rightsquigarrow & x[f]y \\
x[F]\uparrow & (F\text{ finite}) & \rightsquigarrow & \bigwedge_{f\in F} x[f]\uparrow \\
x[F]\uparrow & (F\text{ cofinite}) & \rightsquigarrow & x[{}^{c}F] \\
x =_F y & (F\text{ finite}) & \rightsquigarrow & \bigwedge_{f\in F} \left(\begin{array}{c} (x[f]\uparrow \wedge y[f]\uparrow) \\ \vee(\exists z \cdot x[f]z \wedge y[f]z) \end{array}\right) \\
x =_F y & (F\text{ cofinite}) & \rightsquigarrow & x \sim_{{}^{c}F} y
\end{array}$$

Figure 3.5: Conversion from the logic presented in this work with the algebra of finite and cofinite sets to FT extended with fence and similarity.

is however satisfiable by the valuation $[x \mapsto t_2, y \mapsto t_3]$. Indeed, t_2 and t_3 are similar on $\{\mathtt{usr}, \mathtt{var}\}$ as they are equal in the former and do not have the latter. t_2 indeed does not have `etc` in its domain, but t_3 does.

- Formula 3.3 has one free variable y – x being bound by a universal quantifier. It is a valid formula. Indeed, let us take any valuation ρ. Let us take any t and consider $\mu = \rho[x \mapsto t]$[12]. If μ does not satisfy the left-hand side of the implication, then it satisfies the whole formula. If μ does satisfy the left-hand side, then $x, y \in \mathtt{dom}(\mu)$, $\mu(x) =_F \mu(y)$[13] and $\mathtt{dom}(\mu(x)) \cap G = \varnothing$. That does indeed imply that $\mathtt{dom}(\mu(y)) \cap F \cap G = \varnothing$, and thus μ satisfies the right-hand side and the whole formula.[14]
- Formula 3.4 is a closed formula as all its three variables are bound. Its validity depends on F. If F is the empty set, then both negated similarity atoms are false no matter which value is given to their variables. If F is not empty, however, and one considers ρ that gives to x and y trees that are not equal in F, then it is true that for any tree t, it is either different from $\rho(x)$ or from $\rho(y)$ in F.

3.2.3 Expressivity of FTS in Comparison to Related Work

FTS is strictly more expressive than FT [Aït-Kaci et al. 1994], the first first-order feature tree logic that has been introduced. FT comprises the predicates $x[f]y$ and $x[f]\uparrow$ – the absence of one feature only. Of course, these two can be encoded in our logic.

FT was later extended to CFT [Smolka & Treinen 1994] that adds an *arity* predicate $x[F]$ for any finite set of features F. This arity predicate states that the root has precisely the features that appear in F. In other words, it is satisfied by any valuation ρ such that $\mathtt{dom}(\rho) = F$. Since we can express the absence of a cofinite set of features and the presence of features[15], we can encode it in FTS as $x[{}^{c}F]\uparrow \wedge \bigwedge_{f\in F} \neg x[f]\uparrow$. This formula first states that the complement of F is absent from x. In other words, that the only features that are allowed are that of F. The formula then lists all the features in f, using the negation of the absence atom to state that they have to be present.

More recently, we extended FT to add a *fence* predicate $x[F]$ and a *similarity* predicate $x \sim_F y$, for any finite set of features F [Jeannerod & Treinen 2018]. The similarity atom states that the two variables may

[12] Note that this is the same as taking any valuation μ. We try however to follow the syntax of the formula in our proof.

[13] Note that this is correct as the symbol $=_F$ is both used for the similarity predicate and for the partial equality of functions. See Chapter 2 for the latter.

[14] Note that $F \cap G$ can be empty, in which case this formula is not so interesting. It is valid nonetheless. It will later be known as the propagation of the absence through the similarity.

[15] Actually, we can also express the absence of a finite set of features and the presence of a feature inside a finite or cofinite set, but this is not relevant for this precise point.

not differ outside F: it is satisfied by any valuation ρ such that $\rho(x) =_{^cF} \rho(y)$. The fence atom differs from the arity atom in that it is an upper bound on the domain of the valuation: it is satisfied by any valuation ρ such that $\mathtt{dom}(\rho) \subseteq F$. The reason behind that choice is that the interactions of the similarity atom with the fence atom are easier to manipulate than with the arity atom. Figures 3.4 and 3.5 sketch the relationship between FTS and FT with fence and similarity atoms: Figure 3.4 shows that FTS is at least as expressive as FT with fence and similarity predicates. Figure 3.5 shows that, if we take for $\mathcal{FS}$ exactly the finite and cofinite sets, then the two logics have exactly the same expressivity.

Finally, our previous work [Jeannerod & Treinen 2018] studied the interaction between FT with fence and similarity predicates – inherently stating properties on the edges of feature trees – with a logic of so-called *decorations*. In that case, the model of feature trees is extended to carry an information of decorations on the nodes. The full logic that was considered was then FT with fence and similarity predicates parametrised by a logic on decorations. The decorations were thoughts to be useful for the modelisation of the file metadata described in Section 3.1.1. Since then, we realised that such metadata could simply be abstracted away. Moreover, our previous work showed that introducing decorations did not add complexity. We therefore decided to leave them away in this thesis.

FT has also been extended by adding quantification over features [Treinen 1993]. The result of this work, however, is that such logics are undecidable. Later work [Treinen 1997] has shown that decidability can be recovered if one restricts the use of feature variables to express existence of features only. Quantification over features did not seem relevant for our use, but it does suggest that FTS could be extended in such a way as well, while keeping decidability results.

Finally, FTS indeed allows us to express update of trees. If we consider again the example "y is an update of x where f now leads to an empty directory", it can be expressed in our logic with Formula 3.5.

$$y =_{^c\{f\}} x \land \exists z \cdot (y[f]z \land z[\star]\uparrow) \tag{3.5}$$

In general, the update predicate of the form "y is an update of x where f now leads to z" is expressible with the formula $y =_{^c\{f\}} x \land y[f]z$. This way, we cut the notion of update in two parts, one of which is the feature atom, which has been studied numerous times before. The other part is the similarity predicate that happens to be rather flexible. For instance, for any F, $=_F$ is an equivalence relation:

- It is reflexive as, for all x and F, $x =_F x$ is valid.
- It is symmetric as, for all x, y and F, $x =_F y$ and $y =_F x$ are equivalent.
- It is transitive as, for all x, y, z and F, $x =_F y \land y =_F z$ implies $x =_F z$.

In fact, the transitivity can be made more general by recognising that for all x, y, z, F and G, $x =_F y \land y =_G z$ implies $x =_{F\cap G} z$. This idea of introducing a similarity predicate as something easier to manipulate than an update predicate can actually be found in other lines of work, and in particular in theory of arrays [Stump et al. 2001] when one wants to express updates of arrays.

3.2.4 Classes of Formulas

This subsection introduces the nomenclature of various classes of formulas that we are going to manipulate in this document. We will start with the simplest formulas: *atoms* in Definition 3.12 and *literals* in Definition 3.13.

Definition 3.12 (Atom)**.** An *atom* – or *predicate* – is a formula that does not have a proper sub-formula. In our logic, it is $x[f]y$, $x[F]\uparrow$ or $x =_F y$ for any x, y, f and F. □

Definition 3.13 (Literal)**.** A *literal* is a formula that is either an atom or the negation of an atom. An atom is said to be a *positive* literal. A negated atom is said to be a *negative* literal. □

Most of the formulas that we will manipulate in this work will be under the form of a disjunction of existentially-quantified conjuntions. We are thus going to define *constraints* in Definition 3.14, *existential constraints* in Definition 3.15 and disjunctions of existential constraints in Definition 3.16.

Definition 3.14 (Constraint)**.** A *constraint* is either $\top$ or of the form $l_1 \wedge \cdots \wedge l_n$ $(n \geq 1)$ where, for all i, l_i is a literal. It can be seen as a – possibly empty – set of literals, the empty set being $\top$. □

Definition 3.15 (Existential Constraint)**.** A *x-constraint*, short for *existential constraint* is of the form $\exists X \cdot c$ where c is a constraint. □

Definition 3.16 (Disjunction of Existential Constraints)**.** A *DXC* – short for *disjunction of x-constraints* – is either $\bot$ or of the form $c_1 \vee \cdots \vee c_m$ $(m \geq 1)$ where, for all j, c_j is an x-constraint. It can be seen as a – possibly empty – set of x-constraints, the empty set being $\bot$. □

DXC are a very particular form of formulas with existential quantifiers. It is not true that any formula built without $\forall$ is logically equivalent to a DXC, that is equivalent to every model.[16] It is true, however, if one restricts where such quantifiers can occur. Let us define what it means for a quantifier to be in a *positive* or *negative* occurrence, in Definition 3.17.

Definition 3.17 (Positive and Negative Occurrence of a Quantifier)**.** In a formula ϕ, an occurrence of a quantifier $\exists$ or $\forall$ is said to be *positive* if it appears under an even number of negations $\neg$. Other occurrences are said to be *negative*. □

For instance, in $\exists x \cdot \neg((\forall y \cdot \phi) \vee (\exists z \cdot \psi) \wedge \neg(\forall w \cdot \chi))$, $\exists x$ and $\forall w$ are positive occurrences and $\forall y$ and $\exists z$ are negative occurrences.

Some classes of formulas can be defined by restrictions on the occurrences of quantifiers. This is the case of Σ_1- and Π_1-*formulas*. These will be defined in Definition 3.18 and Definition 3.19 respectively. We will then state in Lemma 3.2 that, for every Σ_1-formula, there exists an equivalent DXC. We will assume the existence of a function `DXC` that yields an equivalent DXC for any Σ_1-formula.

Definition 3.18 (Σ_1-formula)**.** A Σ_1-*formula* is a formula in which all existential quantifiers appear in positive occurrence and all universal quantifiers appear in negative occurrence. □

Definition 3.19 (Π_1-formula)**.** A Π_1-*formula* is a formula in which all universal quantifiers appear in positive occurrence and all existential quantifiers appear in negative occurrence. □

Lemma 3.2 (Existence of DXC for Σ_1-formulas)**.** *For all Σ_1-formula, there exists an equivalent DXC.*

DXC allow us to manipulate very limited formulas. Although most of this work will focus on such formulas, we will need to sometimes talk about any formula of our logic. In order to manipulate these, we introduce in Definition 3.21 what it means for formulas to be in *prenex normal form*. We state in Lemma 3.3 that any formula has a prenex normal form. We describe more precisely in Lemma 3.4 the shape of this prenex normal form for Π_1-formulas. We will assume the existence of a function `PNF` that takes a formula and returns one of its prenex normal forms.

Definition 3.20 (Quantifier-free Formula)**.** A formula is *quantifier-free* if it does not contain $\exists$ or $\forall$. □

[16] The whole Chapter 4 will show that it is true in $\mathcal{FT}$.

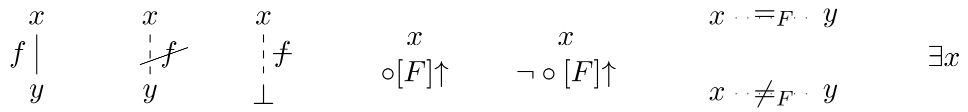


Figure 3.6: Base predicates of FTS

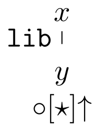


Figure 3.7: Formula 3.1

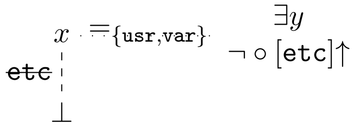


Figure 3.8: Formula 3.2

Definition 3.21 (Prenex Normal Form)**.** A formula ϕ is in *PNF* – short for *prenex normal form* – if there exists Q a string of quantifications and ψ a quantifier-free formula such that $\phi = Q \cdot \psi$. □

Lemma 3.3 (Existence of PNF for any Formula)**.** *For any formula ϕ, there exists a formula in prenex normal form that is equivalent to ϕ in every model. In other words, for any formula ϕ, there exists Q a string of quantifications and ψ a quantifier-free formula such that: $\models \tilde{\forall} \cdot (\phi \leftrightarrow Q \cdot \psi)$. We say that ψ is a prenex normal form of ϕ or that ϕ has the prenex normal form ψ.*

Lemma 3.4 (Shape of PNF for Π_1-formulas)**.** *For any Π_1-formula ϕ, there exists a prenex normal form with only universal quantifiers that is equivalent to ϕ in every model. In other words, for any Π_1-formula ϕ, there exists X a set of variables and ψ a quantifier-free formula such that $\models \tilde{\forall} \cdot (\phi \leftrightarrow \forall X \cdot \psi)$.*

Let us take as example the formulas of Figure 3.3. Formula 3.1 is a constraint of two positive literals. It is also a x-constraint (with an empty quantifier block), a DXC and a PNF. Formula 3.2 is an x-constraint of two positive and one negative literals. It is also a DXC and a PNF. Formulas 3.3 and 3.4 are in PNF.

In this work, we will focus particularly on DXC. For these formulas, we introduce a graphical representation of the base predicates in Figure 3.6. X-Constraints are simply represented by the presence of multiple base predicates in the same figure. Existential quantification is represented using the $\exists$ symbol near a variable. DXC are represented using multiple figures. The graphical representation of an x-constraint can be seen as an oriented graph whose vertices are variables. A feature $x[f]y$ is represented by an edge from x to y carrying the feature f. By convention, features will always be top to bottom and will not carry an arrow.[17] A negated feature atom $\neg x[f]y$ is represented by a dashed edge carrying the feature f from x to y. The whole edge is striked through. An absence $x[f]\uparrow$ is represented by a dashed edge carrying the feature f from the variable x to a $\perp$ node. The feature f is striked through. An absence $x[F]\uparrow$ is either represented by several absences $x[f]\uparrow$ if F is finite or by the annotation $\circ[F]\uparrow$ under the variable it concerns. A negated absence $\neg x[F]\uparrow$ is represented by the annotation $\neg\circ[F]\uparrow$ under the variable it concerns. A similarity atom $x =_F y$ and a negated similarity atom $x \neq_F y$ are both represented by a dotted edge from x to y carrying $=_F$ and $\neq_F$ respectively. Finally, a variable that is existentially quantified upon in the x-constraint will be preceded by the $\exists$ symbol.

Figures 3.7 and 3.8 show the graphical representations of Formulas 3.1 and 3.2. Formulas 3.3 and 3.4 do not have a graphical representation because they both contain universal quantifiers and are thus not DXC.

[17] This makes sense most of the time as we mostly consider the model $\mathcal{FT}$ of finite feature trees. A formula containing a cycle of features is always trivially false in this model and is therefore not so interesting to consider and represent graphically.

```
1 rm /etc/rancid/lg.conf
2 rm -R /etc/rancid/lg.conf
3 rm -R -i /etc/rancid/lg.conf
4 rm -Ri /etc/rancid/lg.conf
5 rm -Ri /etc/rancid/lg.conf /usr/lib/ocaml
```

Figure 3.9: Five example calls to the utility `rm`

3.3 Modelisation of Utilities

Section 3.1 defined feature trees as models for filesystems. Section 3.2 defined FTS, a logic over feature trees which comprises, in particular, a similarity predicate allowing to express the update. We can therefore use formulas of FTS to model transformations of filesystems. We will see in this section how we use them to model Unix utilities.

In Section 3.3.1, we define utilities and describe what we want to model in them. In Section 3.3.2, we show how we use FTS to model one call to a utility. In Section 3.3.3, we define a set of macros that help us extend such utilities to schemes that cover all similarly looking utility calls. Finally, in Section 3.3.4, we show how we model utilities.

3.3.1 Utilities

A *utility* [20, Section 4] is a program that takes as input a (potentially empty) list of arguments and a string. It produces as output a number and two strings. It can also read and perform modifications of the filesystem of the computer as side-effect. The strings as input and outputs are called *standard input*, *standard output* and *standard error*. The number is called *return code*. It is used to represent the status of the utility – success or error. *Utilities* are also often called *command* but we will avoid that name as, in the context of Shell, it can be misleading.

Some utilities do not have any effect on the filesystem. They can be used for various tasks, for instance giving the date (`date`) or modifying an input text (`sed`)[18]. Some do not perform modifications but do read the filesystem. They can for instance be used to test the state of a file (`test`, aka. the bracket `[`), or to list the contents of a directory (`ls`). Finally, some utilities do perform modifications of the filesystem. They create directories (`mkdir`), move files (`mv`), write to files, etc. Utilities do not have to perform only simple tasks. Some are very rich and complex pieces of software that can range from text editors (`nano`) to web browsers (`links`) and e-mail clients (`mutt`). Technically, bigger, graphical programs are also considered to be utilities.

In this work, we will only look into utilities that appear in maintainer scripts. They are usually simple. Also – but it really varies from one utility to another –, utilities that do not depend on the filesystem are meant to provide useful outputs for a user, and utilities that do perform modifications of the filesystem do not provide outputs except to report errors.

A *utility call* is the description of a utility name and its arguments. We will not encounter the case of utilities that take a string on its standard input in this part of the work. A utility call will typically be represented as in Figure 3.9, by analogy with Shell commands: they always start with the name of the utility (in this example, `rm`, that can remove files) and are followed by a space-separated list of arguments. The utility call on Line 5, for instance, has for arguments the three strings "`-Ri`", "`/etc/rancid/lg.conf`" and "`/usr/lib/ocaml`".

[18] Yes, `sed` can *also* modify the filesystem, but not actually the version in POSIX standard, and we will explain later that utilities can be extremely versatile and change completely their behaviour depending on their argument.

Technically, utilities arguments are not restrained to specific syntactic conventions. Most of the ones we will encounter in this work follow the *XBD Utility Syntax Guidelines* [19, Section 12.2] and expect their arguments to be one-character options starting by a dash (`-`) character (`-R`, `-i`, etc.) followed by operands. Multiple options can be merged together without changing the meaning of the command (eg. `-Ri`). Since we are interested here in utilities that perform modifications of the file system, these operands will often be paths (eg. `/etc/rancid/lg.conf`).

Options can radically change the behaviour of a utility. Let us consider the five different ways in which the utility `rm` is called in Figure 3.9:

1. Without options, `rm` removes the given file if it is not a directory. It fails on directories.
2. The `-R` options makes `rm` remove the target recursively. This only makes a difference if the target is a directory, in which case the directory and all of its content is removed.
3. The `-i` options makes `rm` ask for confirmation to the user before removing anything.
4. The `-R` and `-i` options can be merged together and written as `-Ri`. This does not change in any way the behaviour of the utility.
5. Several operands can be provided and `rm` will simply treat them one after the other. There is here a subtlety in case the handling of one of the arguments go wrong. This is described in Section 3.3.4.

Utilities that we are interested in typically work in two phases. In a first phase, they read their arguments, run sanity checks on them and decide what modification of the filesystem they will perform. In a second phase, they actually perform that modification if it is possible in the current state of the filesystem. Similarly, we will abstract utilities as programs that first read their arguments, run sanity checks and decide what modification of the filesystem they will try to perform, and then instantiate a *specification* – that is a logical formula representing this modification in our logic – for this particular case.

Two points need to be noted. Firstly, the standard input and output and the return codes will not be part of the specifications, that is they do not belong to the logical formalism. They are however still present in the model. Secondly, in our abstraction, we collapse all the 127 error return codes that a utility may return and only consider that a utility may succeed or fail. This is justified as that control structures of Shell only differentiate between these two cases and as none of maintainer scripts of Debian only seldom inspect the value of return codes.

3.3.2 Specifications of One Utility Call

Using FTS, we can give specifications to utility calls. A utility call can be seen as a transformation from an input tree to an output tree. We can specify such objects with relations between trees, which can themselves be represented by a formula of two free variables: the input and the output filesystems. Such formulas will be called *specifications*. Although formulas of two free variables represent relations, we will see in Section 3.4 that, the huge majority of the time, we write specifications as functions. For this reason, we allow ourselves the use of the word *transformation* to describe the formula represented by a specification.

In fact, specifications have to be split between several *specification cases* describing the transformations that are performed when the utility succeeds and when the utility fails. For readability of such formulas by human readers, we sometimes provide several cases for success or error. For instance, if we consider the utility call `rm /etc/rancid/lg.conf`, it can fail when the path `/etc/rancid/lg.conf` does not exist but also when it exists and points to a directory.

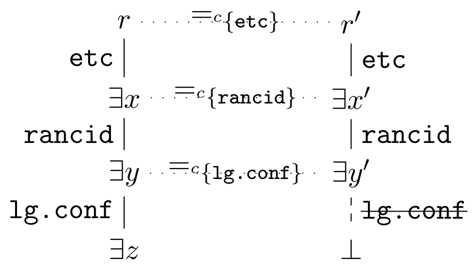


Figure 3.10: Specification of success case for rm -R /etc/rancid/lg.conf

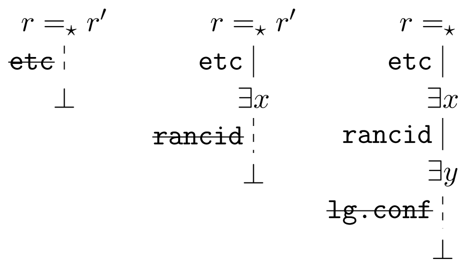


Figure 3.11: Specification of error cases for rm -R /etc/rancid/lg.conf

Each specification case is written in two parts. The first part is a formula of the input tree only. It describes a *precondition*. The second part is a formula describing the *transformation* which is performed when the precondition is met. The *specification case* is the conjunction of the two parts.

Although this suggests that specifications are written under the form of a DXC, they are not. Specification cases are conjunctions of two parts. These parts can contain existential quantifiers and also disjunctions. We are however going to make sure that our specifications can be transformed into DXC as this is what our solvers can handle efficiently (see Chapter 4). We thus need to force ourselves to write Σ_1-formulas only. By Lemma 3.2, this is sufficient.

Let us take the utility call `rm -R /etc/rancid/lg.conf` as an example. The `-R` option means that `rm` removes the path `/etc/rancid/lg.conf` as long as it exists, whether it is a directory or not. It makes for a simpler specification and avoids handling file kinds. Let us write the specification case of its success, that is two formulas $\phi_1^{(p)}(r)$ and $\phi_1^{(t)}(r, r')$, where r and r' represent the input and output trees respectively. For this utility call to succeed, the path `/etc/rancid/lg.conf` has to exist in the input tree r. The output tree r' is then very similar except in this path. In the output tree, the path `/etc/rancid/lg.conf` does not exist anymore. As a complete formula, and quantifying over the intermediary variables, this gives us Formula 3.6. A graphical representation can be found in Figure 3.10.

$$\begin{aligned}
\phi_1{}^{(p)}(r) \quad &= \exists x, y, z \cdot (r[\texttt{etc}]x \land x[\texttt{rancid}]y \land y[\texttt{lg.conf}]z) \\
\phi_1{}^{(t)}(r, r') &= \exists x, y, x', y' \cdot \begin{pmatrix} r =_{c\{\texttt{etc}\}} r' \land x =_{c\{\texttt{rancid}\}} x' \land y =_{c\{\texttt{lg.conf}\}} y' \\ \land\, r[\texttt{etc}]x \land x[\texttt{rancid}]y \land r'[\texttt{etc}]x' \land x'[\texttt{rancid}]y' \\ \land\, y'[\texttt{lg.conf}]\uparrow \end{pmatrix} \\
\phi_1 \quad (r, r') &= \phi_1{}^{(p)}(r) \land \phi_1{}^{(t)}(r, r')
\end{aligned} \tag{3.6}$$

This does not cover all the possibilities as this utility call can also fail. We also need to take into account this possible failure and the transformation that is performed then. In this case, the utility call simply does nothing and the output tree is the same as the input one. This can be described with a precondition that simply negates that of Formula 3.6 and a transformation being $\phi_2^{(t)}(r, r') = r =_\star r'$. Such a precondition is given in Formula 3.7.

$$\phi_2{}^{(p)}(r) \quad = \neg\exists x, y, z \cdot (r[\texttt{etc}]x \land x[\texttt{rancid}]y \land y[\texttt{lg.conf}]z) \tag{3.7}$$

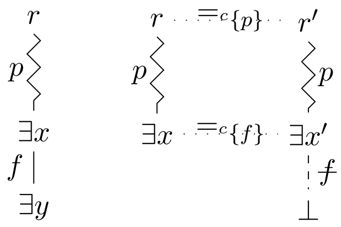
Formula 3.7, however, is a Π_1-formula as it involves a negative occurrence of an existential quantifier. This is forbidden because – as discussed at the beginning of this subsection – we want to restrict ourselves to formulas that admit a DXC. We thus prefer an alternative version that circumvents this problem by enumerating the various reasons why a path could not exist: `etc` may not exist; if it exists, then `rancid` may not exist; if it exists, then `lg.conf` does not. This is shown in Formula 3.8. A graphical representation can be found in Figure 3.11.

$$\phi_2{}^{(p)}(r) \quad = \exists x, y \cdot \begin{array}{l} r[\texttt{etc}]\uparrow \\ \lor\ (r[\texttt{etc}]x \land x[\texttt{rancid}]\uparrow) \\ \lor\ (r[\texttt{etc}]x \land x[\texttt{rancid}]y \land y[\texttt{lg.conf}]\uparrow) \end{array} \tag{3.8}$$

Formula 3.8 is equivalent to Formula 3.7 in the model $\mathcal{FT}$, but contrary to Formula 3.7, Formula 3.8 does admit a DXC. We will show in Chapter 5 that we have an efficient way to avoid generating explicitly such a disjunction.

The full specification of `rm -R /etc/rancid/lg.conf` is then the list of all its specification cases. In this example, there are only two: the specification case for the success – $\phi_1(r, r')$ – and the specification case for the error – $\phi_2(r, r')$. If we solely consider the transformation performed by the utility, then we are only interested in the disjunction of all the specification cases. It is important, however, for the purpose of symbolic execution, to be able to separate success and error cases.

3.3.3 Specifications of Utility Call Schemes



(a) Precondition (b) Transformation

Figure 3.12: Success case of `rm -R p/f`

Section 3.3.2 presents the specification of one utility call. Of course, we do not specify all the utility calls by hand – there is an infinity of possible path arguments. Instead, we generalise the specification given in Section 3.3.2 so that it covers all the similar utility calls.

As an example, let us give a specification for `rm -R` p/f, where p/f is any path ending with a feature f. Figure 3.12 presents an informal graphical representation of both the precondition and transformation of `rm -R` p/f. In this graphical representation, the zigzags labelled by p represents a chain of feature and similarity atoms following the path p.

We need to define functions that formalise these zigzags labels, that is functions that build formulas expressing the existence of a path, the similarity of two trees along a path and the non-existence of a path.

Let us start with the existence of a path. We simply have to define a function that chains feature atoms for the whole given path. We could for instance define a function `resolve`(x, q, z) which chains feature atoms from x to z following the list of features q. The definition of such a function can be found in Figure 3.13, Line 1. It works as follows. If q is empty, then x and z have to be equal and we therefore return $x =_\star z$ (Line 4). If q is not empty, then it is of the form f/q' where f is a feature. The chain of feature atoms has to start with a feature $x[f]y$ for some fresh variable y and there remains q' to crawl from y (Line 5).

The state of affairs is actually more complex as the paths can be absolute or relative, and can contain `.` and `..`. As described in Section 3.1.1, absolute paths are resolved from the root of the filesystem and relative paths are resolved from the *current working directory*. Our functions will thus take that current working directory as an argument, named `cwd`.

Let us now define the actual function `resolve`$(r,$ `cwd`$, p, z)$ that builds the resolution in the root r of a path p from the current directory `cwd` and to z. Similarly to that of Figure 3.13, it will create a new feature

```
function resolve($x$ : variable, $q$ : list of features, $z$ : variable)
    : x-constraint
  match $q$
  | $\varepsilon$ -> return $x =_\star z$
  | $f \rangle q'$ -> return $\exists y \cdot (x[f]y \wedge$ resolve$(y, q', z))$
```

Figure 3.13: Function resolve on normal paths

```
1  function resolve-s($x$ : variable, $\pi$ : stack of variables,
2      $q$ : list of path components, $z$ : variable) : x-constraint
3    match $q$
4    | $\varepsilon$ -> return $x =_\star z$
5    | $f \rangle q'$ -> return $\exists y \cdot (x[f]y \wedge$ resolve-s$(y, \pi \langle x, q, z))$
6    | . $\rangle q'$ -> return resolve-s$(x, \pi, q', z)$
7    | .. $\rangle q'$ ->
8      match $\pi$
9      | $\varepsilon$ -> return resolve-s$(x, \varepsilon, q', z)$
10     | $\pi' \langle y$ -> return resolve-s$(y, \pi', q', z)$
11
12 function resolve($r$ : variable, cwd : normal path,
13     $p$ : path, $z$ : variable) : x-constraint
14   match $p$
15   | abs($q$) -> return resolve-s$(r, \varepsilon,$     $q, z)$
16   | rel($q$) -> return resolve-s$(r, \varepsilon,$ cwd/$q, z)$
```

Figure 3.14: Function resolve for any path

atom for each path component that is not . or .. . All occurrences of . will be ignored. On occurrences of .., our function finds the parent variable (or the current one if it is the root) and starts again from there. In order to do that, we actually use an auxiliary function resolve-s(x, π, q, z) which keeps a stack π of parent variables, thanks to which it is able to interpret .. . The definitions of resolve and resolve-s can be found in Figure 3.14, Lines 12 and 1 respectively. We believe that the code is similar to that of Figure 3.1, except that we do not need to care about different file kinds. Figure 3.15 presents an example of a formula generated by resolve.

Defining a function similar for the similarity of two trees along a path brings different issues. With respect to parent directories in particular, one would be tempted to define it in the same way resolve is defined. Figure 3.16 gives an example of the formula that such a naive version could produce. The main problem here is that the pairs of variables that are visited several times by the function receive several similarity predicates. In the example, the two variables x and x' are said to be both similar outside $\{f\}$

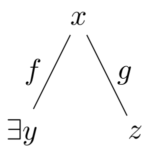


Figure 3.15:
resolve$(x, f, ../../g, z)$

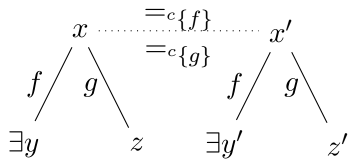


Figure 3.16:
similar$(x, x', f/../../g, z, z')$
Naive version

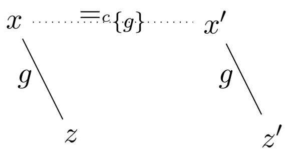


Figure 3.17:
similar$(x, x', f/../../g, z, z')$
After normalising the path

```
function similar-n($x$ : variable, $x'$ : variable,
    $q$ : list of features, $z$ : variable, $z'$ : variable) : x-constraint
  match $q$
  | $\varepsilon$ -> return $x =_\star z \wedge x' =_\star z'$
  | $f \rangle q'$ -> return $\exists y, y' \cdot (x[f]y \wedge x'[f]y' \wedge x =_{c\{f\}} x' \wedge \texttt{similar-n}(y, y', q', z, z'))$
```

Figure 3.18: Function similar-n for a normalised path

```
function normalise(cwd : normal path, $q$ : list of path components)
    : normal path
  match $q$
  | $\varepsilon$ -> return cwd
  | $f \rangle q'$ -> return normalise(cwd $\langle f, q'$)
  | . $\rangle q'$ -> return normalise(cwd, $q'$)
  | .. $\rangle q'$ ->
    match cwd
    | / -> return normalise(/, $q'$)
    | cwd' $\langle$ _ -> return normalise(cwd', $q'$)

function similar($r$ : variable, $r'$ : variable, cwd : normal path,
    $p$ : path, $z$ : variable, $z'$ : variable) : x-constraint
  match $p$
  | abs($q$) -> return similar-n($r, r'$, normalise(  /, $q$), $z, z'$)
  | rel($q$) -> return similar-n($r, r'$, normalise(cwd, $q$), $z, z'$)
```

Figure 3.19: Functions normalise and similar for any path

and outside $\{g\}$. If f and g are distinct, this means that x and x' are similar in everything, which is clearly not what we want.

Our solution is to first normalise the path syntactically, erasing all occurrences of f/.., and then to build a chain of similarity atoms on the normalised path with a function similar-n. It is to be noted that the existence of the normalised version of a path does not imply the existence of the path itself. This means that the obtained function similar-n only makes sense when used in conjunction with resolve. An example of what similar-n produces is given in Figure 3.17. The definition of similar-n on normalised path is given in Figure 3.18.

The definition of the similarity of any path, similar, can then be given in term of the path normalisation normalise and the similarity on normalised path similar-n. It can be found in Figure 3.19.

Finally, there remains to define the non-existence of a path. As mentioned in Section 3.3.2, and because we want our specifications to be DXC formulas only, we cannot define the failure to resolve a path p as in Formula 3.9 as this is a Π_1-formula.

$$\neg\exists z \cdot \texttt{resolve}(r, \texttt{cwd}, p, z) \tag{3.9}$$

Instead, we have to give a definition that lists all the possible cases of non-existence. Such a definition can be found in Figure 3.20.

Finally, let us give the scheme of specifications for rm -R p/f without options. The success and error

```
function noresolve-s($x$ : variable, $\pi$ : variable stack, $q$ : path)
    : $\Sigma_1$-formula
  match $q$
  |   /   -> return $\bot$
  |  $f/q'$ -> return $x[f]\uparrow \vee \exists y \cdot (x[f]y \wedge$ noresolve-s$(y, \pi \langle x, q'))$
  |  ./$q'$ -> return noresolve-s$(x, \pi, q')$
  | ../$q'$ ->
    match $\pi$
    | $\varepsilon$ -> return noresolve-s$(x, \varepsilon, q')$
    | $\pi' \langle y$ -> return noresolve-s$(y, \pi', q')$

function noresolve($r$ : variable, cwd : path, $p$ : path)
    : $\Sigma_1$-formula
  match $p$
  | abs($q$) -> return noresolve-s$(r, \varepsilon,$     $q)$
  | rel($q$) -> return noresolve-s$(r, \varepsilon, \texttt{cwd}/q)$
```

Figure 3.20: Function noresolve for any path

cases can be found in Formulas 3.10 and 3.11 respectively.

$$\begin{aligned}
{\phi_1}^{(p)}(r) &= \exists z \cdot \texttt{resolve}(r, \texttt{cwd}, p/f, z) \\
{\phi_1}^{(t)}(r, r') &= \exists y, y' \cdot \texttt{similar}(r, r', \texttt{cwd}, p, y, y') \wedge y =_{c\{f\}} y' \wedge y'[f]\uparrow \\
\phi_1 \quad (r, r') &= {\phi_1}^{(p)}(r) \wedge {\phi_1}^{(t)}(r, r')
\end{aligned} \tag{3.10}$$

$$\begin{aligned}
{\phi_2}^{(p)}(r) &= \texttt{noresolve}(r, \texttt{cwd}, p/f) \\
{\phi_2}^{(t)}(r, r') &= r =_\star r' \\
\phi_2 \quad (r, r') &= {\phi_2}^{(p)}(r) \wedge {\phi_2}^{(t)}(r, r')
\end{aligned} \tag{3.11}$$

One of our contributions as part of the CoLiS projects is a technical report containing such specifications for the schemes of utility calls that are most widely used in Debian maintainer scripts [Jeannerod et al. 2019].

3.3.4 Modelisation of Utilities

Once we have given specifications to the various utility call schemes, as described in Section 3.3.3, we can give an implementation to our modelled utility. This is a program that takes command line arguments as input and generates a specification as output. This program has to handle the options to decide which specification has to be generated from the given paths. It will also have to handle iteration through a list of paths if several paths are given as argument.

The program has to behave concretely and according to the standard [20, Section 4] on its arguments. In the example of rm, this means the following.

- If the last component of the path given to rm is . or .., or if the path resolves to the root directory, rm fails and skips this path.
- If the option -R (or -r) is not given, then rm fails if the path resolves to a directory. If the option is given, rm succeeds and remove the directory and all of its content.
- When several paths are given, rm iterates through all of them even if a path triggers one of the two aforementioned errors. The utility succeeds if all paths were removed successfully and fails otherwise.

```
 1 function rm-R-spec(cwd : normal path, path : path) : specification
 2   match path
 3   | / -> return ((Error, $r =_\star r'$) › $\varepsilon$)
 4   | _‹. -> return ((Error, $r =_\star r'$) › $\varepsilon$)
 5   | _‹.. -> return ((Error, $r =_\star r'$) › $\varepsilon$)
 6   | $p$‹$f$ ->
 7     return (
 8       (Success, $\exists y, y', z \cdot (\texttt{resolve}(r, \texttt{cwd}, \texttt{path}, z)$
 9                          $\wedge\, \texttt{similar}(r, r', \texttt{cwd}, p, y, y') \wedge y =_{c\{f\}} y' \wedge y'[f]\uparrow)$)
10       › (Error, $(\texttt{noresolve}(r, \texttt{cwd}, \texttt{path}) \wedge r =_\star r')$) ;
11       › $\varepsilon$
12     )
13
14 function rm-spec(cwd : normal path, p : path) : specification
15
16 function iterate-spec(spec : (normal path, path) -> specification,
17      cwd : normal path, paths : list of paths) : specification
18
19 function rm-model(stdin : string, cwd : normal path,
20     args : list of strings) : specification
21   match args
22   | "-R"›args' -> return iterate-spec(rm-R-spec, cwd, args')
23   | _ -> return iterate-spec(rm-spec, cwd, args)
```

Figure 3.21: Implementation of a modelled rm utility (excerpt; simplified)

Figure 3.21 gives an idea of what the implementation of such a modelled utility would look like. This figure purposefully leaves a lot of details aside and is therefore very approximate.

The utility rm is modelled by a function rm-model (Line 19). This function takes as argument the standard input stdin – in the case of rm, this is ignored –, the current working directory cwd and a list of arguments args. It returns a specification. The function rm-model basically only handles the parsing of the arguments. In this case, it checks whether the list of arguments starts by "-R" or not. The rest of the computation is left to three auxiliary functions, iterate-spec, rm-spec and rm-R-spec, Lines 16, 14 and 1 respectively.

- iterate-spec (Line 16) is a helper function that iterates a specification generator – that is rm-spec or rm-R-spec – on all the paths given as argument, composes the obtained specification and returns a specification of two cases. The success case comprises all the success cases of all the given paths. The error case comprises all the combinations that contain at least one error case.

- rm-spec (Line 14) will not be detailed. It is very similar to rm-R-spec except that its success case requires that the given path is not a directory and that it has one extra error cases corresponding the when the given path is a directory.

- rm-R-spec (Line 1) generates the specification for one path given to rm -R. If analyses the path that has been given to it. If the path is the root, . or .. (Lines 3 to 5), the returned specification is the trivial specification containing only one cases that is an error and no transformation. If the path is not one of these (Line 6), the returned specification corresponds to the scheme described in Section 3.3.3.

3.4 Specifications

Although we have described in details how we write the specification of a utility call, we have not taken time to discuss these objects in details. This is the topic of this section. In Section 3.4.1, we discuss properties of the specifications that we write for Unix utilities. Finally, in Section 3.4.2, we explain how such specifications are to be composed – as `iterate-specs` would do in Section 3.3.4 – and we discuss the requirements that this imposes on the solvers that we will design in subsequent chapters.

3.4.1 Properties of Specifications

Let us first note that, when writing the specification of `rm -R /etc/rancid/lg.conf` in Section 3.3.2, we have paid attention to the fact that the preconditions covered all the possible input trees. In a sense, this means that our specification was *complete*, as defined in Definition 3.22.

Definition 3.22 (Completeness of a Specification)**.** A specification ϕ is *complete* if the preconditions of its cases – success or error – cover all the possible input trees, that is if:

$$\models_{\mathcal{FT}} \forall r \cdot \bigvee_i \phi_i^{(p)}(r) \qquad \square$$

Completeness is important for us because it ensures that we do not miss behaviours from a utility. In our approach where we try to find bugs, that means that there are no bugs missed by the specifications.[19]

This is in fact not exact as it is possible to have a complete specification ϕ and an input tree t such that $[r \mapsto t]$ does not satisfy $\exists r' \cdot \phi(r, r')$. This comes from our way to write specification cases as the conjunction of a precondition and a transformation. It does not matter that the preconditions cover all the input trees if the transformations are unsatisfiable. This is something we pay attention to when writing specifications, and which we call the *coherence* of a specification. This is defined in Definition 3.23.

Definition 3.23 (Coherence of a Specification)**.** A specification case ϕ_i is *coherent* if its precondition implies the satisfiability of its associated transformation, that is if:

$$\models_{\mathcal{FT}} \forall r \cdot (\phi_i^{(p)}(r) \rightarrow \exists r' \cdot \phi_i^{(t)}(r, r'))$$

A specification is *coherent* if all its specification cases are coherent. □

Completeness and coherence together imply that a specification is *total*, that is that the relation it represents covers all the possible input trees. This is defined in Definition 3.24. The relation between completeness, coherence and totality is stated in Lemma 3.5.

Definition 3.24 (Totality of a Specification)**.** A specification ϕ is *total* if it has, for each input, at least one output tree, that is if:

$$\models_{\mathcal{FT}} \forall r \cdot \exists r' \cdot \phi(r, r') \qquad \square$$

Lemma 3.5 (Relation Between Completeness, Coherence and Totality)**.** *Any specification that is total is also complete. Any specification that is both complete and coherent is also total.*

[19] Of course, there might be plenty of bugs missed in the modelisation phase. Completeness does however guarantee that we do not miss anything in the models that we are considering.

Proof. Let us take a specification ϕ that is total. Let us show that it is then also complete. To achieve that, let us take t any feature tree. We have that $[r \mapsto t] \models_{\mathcal{FT}} \exists r' \cdot \phi(r, r')$. We have thus that $[r \mapsto t] \models_{\mathcal{FT}} \bigvee_i (\phi_i^{(p)}(r) \wedge \exists r' \cdot \phi_i^{(t)}(r, r'))$. If we consider the conjunctive normal form of this disjunction, it contains a lot of disjunctive clauses among which we find $\bigvee_i \phi_i^{(p)}(r)$. In other words, there exists ψ a formula such that $\bigvee_i (\phi_i^{(p)}(r) \wedge \exists r' \cdot \phi_i^{(t)}(r, r'))$ is equivalent to $(\bigvee_i \phi_i^{(p)}(r)) \wedge \psi$. We thus have that $[r \mapsto t]$ is a model of $\bigvee_i \phi_i^{(p)}(r)$ – ϕ is complete.

Let us now take a specification ϕ that is complete and coherent and let us show that it is total. To achieve that, let us take t any feature tree. We are going to show that there exists t' such that $[r \mapsto t, r' \mapsto t'] \models_{\mathcal{FT}} \phi(r, r')$.

Since ϕ is complete, we have that $[r \mapsto t] \models_{\mathcal{FT}} \bigvee_i \phi_i^{(p)}(r)$. There is thus a i_0 such that $[r \mapsto t] \models_{\mathcal{FT}} \phi_{i_0}^{(p)}(r)$. Since all of ϕ's cases are coherent, the i_0-th in particular is coherent. From which it follows that $[r \mapsto t] \models_{\mathcal{FT}} \exists r' \cdot \phi_{i_0}^{(t)}(r, r')$. There exists thus a t' such that $[r \mapsto t, r' \mapsto t'] \models_{\mathcal{FT}} \phi_{i_0}^{(t)}(r, r')$. $[r \mapsto t, r' \mapsto t']$ is then a model of $\phi_{i_0}^{(t)}(r, r')$ but also of $\phi_{i_0}^{(p)}(r, r')$. It is thus also a model of $\phi_{i_0}(r, r')$ and of the whole specification $\phi(r, r')$. □

Completeness, coherence and totality have to do with covering all the possible cases for the input tree. They ensure that we do not loose traces of executions – and therefore bugs – in our specifications. They do not, however, carry information on the precision of the specifications. In particular, they do not guarantee that we will not get *false positives*, that is bugs that are not actually reachable. The properties that carry such information are the *determinism* and the *functionality* of a specification. Determinism is defined in Definition 3.25.

Definition 3.25 (Determinism of a Specification)**.** A specification ϕ is *deterministic* if there is no pair of preconditions that cover the same possible input tree, that is if, for all $i \neq j$:

$$\models_{\mathcal{FT}} \neg\exists r \cdot (\phi_i^{(p)}(r) \wedge \phi_j^{(p)}(r)) \qquad \square$$

Determinism is particularly important between preconditions of different status – success or error. Indeed, if it holds, then it ensures that the input trees are classified correctly. Therefore, and even if we cannot say precisely to which output tree they are mapped, we can guarantee that the traces of execution associated to them are actually reachable. Determinism is in fact a weaker form of functionality, which is defined in Definition 3.26.

Definition 3.26 (Functionality of a Specification)**.** A specification – or a specification case – ϕ is *functional* if every input tree is related to at most one output tree, that is if:

$$\models_{\mathcal{FT}} \forall r, r'_1, r'_2 \cdot ((\phi(r, r'_1) \wedge \phi(r, r'_2)) \rightarrow r'_1 =_\star r'_2) \qquad \square$$

Determinism is weaker than functionality in the sense that it does not ensure that, inside one specification case, the same input tree can be related to two output trees. Of course, if one takes a specification that is deterministic and such that all its cases are functional, then it is a functional specification. This is stated in Lemma 3.6.

Lemma 3.6 (Relation Between Determinism and Functionality)**.** *Any specification that is functional is also deterministic. Any specification that is both deterministic and with all its cases being functional is also functional.*

In this work, all our specifications are written to be complete and coherent. Most of them are also functional. However, for some of them, our logics is not expressive enough to express exactly what the utility call does. This is for instance the case of the utility `cp`. In one particular case, `cp` produces as output a (potentially partial) interleaving of two input trees, that is a (potentially strict) subset of the union of the two input trees, which our logic can simply not express. In that case, our specification over-approximates the behaviour of the utility, allowing one input tree to lead to several output trees, giving up on functionality in the process. Such situations occur very rarely in the maintainer scripts of Debian, which justifies that we do not extend FTS – with all the complexity such an extension implies – to handle them.

An alternative way to write the same specifications could have been in an *implicative* style. In such a style, specification cases ϕ_i are written of the form $\phi_i^{(p)}(r) \rightarrow \phi_i^{(t)}(r, r')$ and the specification is the conjunction – and not the disjunction – of its cases, as in Formula 3.12.

$$\phi_{\rightarrow}(r, r') = \bigwedge_i (\phi_i^{(p)}(r) \rightarrow \phi_i^{(t)}(r, r')) \tag{3.12}$$

The two representations of specifications are in fact very close. Provided some properties of this subsection are verified, they are actually equivalent, as stated in the following lemma.

Lemma 3.7 (Equivalence of Specifications and their Implicative Form)**.** *If a specification ϕ is complete, then it is implied by its implicative version $\phi_{\rightarrow}$. If a specification ϕ is deterministic, then it implies its implicative version $\phi_{\rightarrow}$.*

Proof. Let ϕ be a specification that is complete. Let us take ρ a model of its implicative variant $\phi_{\rightarrow}$. Since ϕ is complete, then there exists i_0 such that $\rho \models_{\mathcal{FT}} \phi_{i_0}^{(p)}(r)$. Since ρ is a model of the implicative variant, then it models all the parts of its conjunction and in particular the i_0-th. From which we can conclude that $\rho \models_{\mathcal{FT}} \phi_{i_0}^{(t)}(r, r')$ and thus that $\rho \models_{\mathcal{FT}} \phi_{i_0}(r, r')$ and that $\rho \models_{\mathcal{FT}} \phi(r, r')$.

In the other direction, let ϕ be a specification that is deterministic. Let us take ρ a model of ϕ. There is i_0 such that $\rho \models_{\mathcal{FT}} \phi_{i_0}^{(p)}(r)$ and $\rho \models_{\mathcal{FT}} \phi_{i_0}^{(t)}(r, r')$. Since ϕ is deterministic, ρ cannot be a model of any other precondition: for any $j \neq i_0$, $\rho \models_{\mathcal{FT}} \neg\phi_j^{(p)}(r)$. Hence, for any j, $\rho \models_{\mathcal{FT}} \phi_j^{(p)}(r) \rightarrow \phi_j^{(t)}(r, r')$. Finally, $\rho \models_{\mathcal{FT}} \phi_{\rightarrow}(r, r')$. □

3.4.2 Composing Specifications

Each specification describes a transformation – ie. a formula mapping an input tree to an output tree. In this work, we are interested in composing transformations along the traces of execution of a script. We therefore need a way to obtain the specification of the composed transformations. It is here easy as the specification of the composition is nothing else than the composition – using simple logical constructions – of the specifications. For instance, if we have two specifications ϕ and ϕ', then the composition of these two specifications is ψ in Formula 3.13.

$$\psi(r, r') = \exists r_t \cdot (\phi(r, r_t) \land \phi'(r_t, r')) \tag{3.13}$$

As we have mentioned in Section 3.3.2, although specifications are not written this way, they are transformed immediately into DXC and processed that way. Another way to see the composition is to say that, if the first specification is a DXC of k x-constraints ϕ_1 to ϕ_k and the second specification is a DXC of l x-constraints ϕ'_1 to ϕ'_l, then the composition is a DXC of $k \times l$ x-constraints $\psi_{1,1}$ to $\psi_{k,l}$ where, for

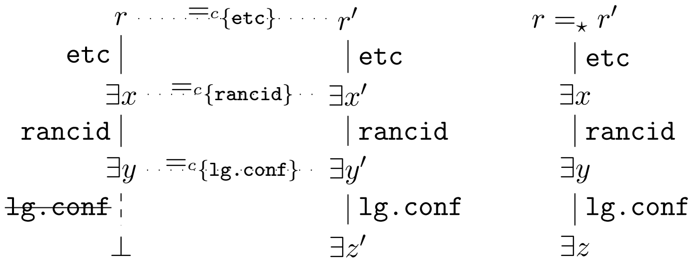


Figure 3.23: Specification of success cases for `touch` `/etc/rancid/lg.conf`

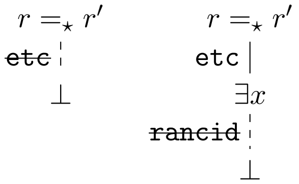


Figure 3.24: Specification of error cases for `touch` `/etc/rancid/lg.conf`

all $1 \leq i \leq k$ and $1 \leq j \leq l$:[20]

$$\psi_{i,j}(r,r') = \exists r_t \cdot (\phi_i(r,r_t) \wedge \phi'_j(r_t,t')) \tag{3.14}$$

This second formulation makes explicit the fact that specifications are DXC and that composing two specifications creates a quadratic explosion of the number of x-constraints. This is not visible per se in the first formulation.

```
touch /etc/rancid/lg.conf
rm /etc/rancid/lg.conf
```

Figure 3.22: Example script that uses `touch` and `rm`

Composing specifications leads to an explosion of cases because of the fact that we do not know how to manipulate formulas that are not under the form of a DXC. When the two specifications are not independent, however, a lot of cases of the composition are in fact unsatisfiable and can be removed.

As an example, consider the script in Figure 3.22. The `touch` utility creates the file if it does not exist and leaves it untouched[21] if it already exists. It fails when the prefix does not exist. A graphical representation of the specification for this utility call can be found in Figures 3.23 and 3.24.

The two utility calls in this script have both a specification that is a DXC of four x-constraints. Their composition should thus give us sixteen x-constraints. A third utility call would multiply again the number of cases. Computing the semantics of whole scripts would then lead to an exponential blowup in cases and to unusability in practice.

However, not all the x-constraint of the specification make sense and some are actually impossible to encounter. For instance, if `touch /etc/rancid/lg.conf` fails for non-existence of `/etc/rancid`, for instance, then `rm /etc/rancid/lg.conf` can obviously not succeed. In fact, in our example script, the composition has only four x-constraints. The two success cases are described in Figure 3.25. The two error cases are very similar to that of Figure 3.24.

Let us revisit Formula 3.14 that defines the composition of two specifications ϕ and ϕ':

$$\psi(r,r') = \bigvee_{i,j} \exists r_t \cdot (\phi_i(r,r_t) \wedge \phi'_j(r_t,r')) \tag{3.14 revisited}$$

We can see here that all the x-constraints of ϕ have been duplicated in the formula as many times as there are x-constraints in ϕ'. Similarly, if we compose the whole with a third specification ϕ'', we obtain

[20] In that case, $\psi_{i,j}(r,r')$ in Formula 3.14 is not exactly an x-constraint. It is however really close and it suffices to switch the existential quantifiers with the conjunction.

[21] In fact, it updates the access time, but this is abstracted away in this work.

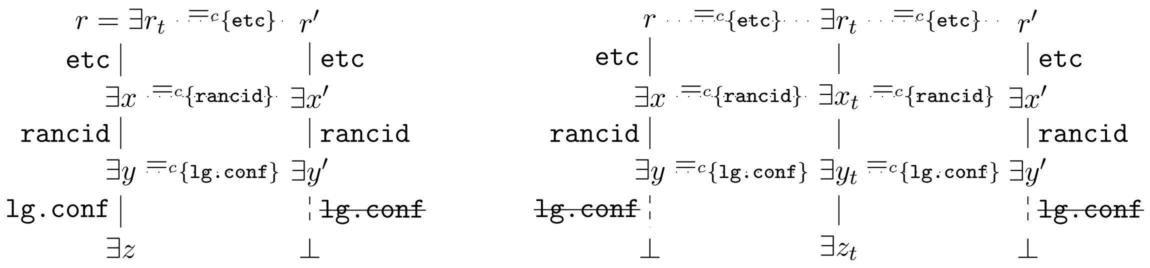


Figure 3.25: Specification of the script: `touch /etc/rancid/lg.conf; rm /etc/rancid/lg.conf`. In the right representation, the some edges are left unmarked for readability. They are of course `etc`, `rancid`, and `lg.conf`.

Formula 3.15.

$$\psi'(r, r') = \bigvee_{i,j,k} \exists r_t, r'_t \cdot (\phi_i(r, r_t) \wedge \phi'_j(r_t, r'_t) \wedge \phi''(r'_t, r')) \tag{3.15}$$

we duplicate again all the composed x-constraints. We have said however that some of these x-constraints might be unsatisfiable. In the case of our example, it was even a large majority of x-constraints: among them, twelve are unsatisfiable while only four are satisfiable. In order to speedup the process, we would want to detect unsatisfiability as soon as possible, ideally right after adding a new specification to the composition. In our example, that would mean adding a third specification to four x-constraints instead of sixteen.

Doing that has however an important drawback. If an x-constraint ϕ_i of the first utility call is decided to be satisfiable, we will then ask ourselves for all j if the x-constraints $\phi_i(r, r_t) \wedge \phi'_j(r_t, r'')$ are also satisfiable. If they are, we will then ask ourselves for all k if the x-constraints $\phi_i(r, r_t) \wedge \phi'_j(r_t, r'_t) \wedge \phi''(r'_t, r')$. Every time, when adding a new specification, we would thus compute again the satisfiability of all the previous x-constraints. In order to mitigate that, we do not only want a test of unsatisfiability. We want that it is done in an incremental way, meaning that, in case of satisfiability, we get another, simpler formula on which it is easier to restart computation.

An incremental satisfiability procedure will be introduced in Chapter 4. We will also extend it to show that the full first-order of our logic is decidable. Chapter 5 will then come back to the procedure, focusing on efficiency considerations. There, we will rework the procedure. We will also discuss improvements that can be included in the logic to reduce the number of disjunctions that are initially introduced in specifications.

Chapter 4

Decidability of the Theory of $\mathcal{FT}$

This chapter dives into the subject of expressivity of FTS and of the design of decision procedures for this logic. It also sets up the foundations for further work – in Chapter 5 – on efficiency of these procedures. The main result is a decision procedure for any closed first-order formula. This result is stated in Theorem 4.3. In all this chapter and the next, unless said otherwise, we consider ourselves in the model $\mathcal{FT}$ of finite feature trees.

The first decidability result of a full first-order theory of trees is due to Malc'ev [Malcev 1971] and concerned *Herbrand trees* (ie. based on equations $x = f(x_1, \dots, x_n)$). This result was later extended by Maher, Comon and Lescanne [Maher 1988; Comon & Lescanne 1989]. A first decidability result for the first-order theory of *feature trees* was given for the logic FT [Aït-Kaci et al. 1994] which comprises only the predicates $x[f]y$ and $x[f]\uparrow$ [Backofen & Smolka 1995]. This was later extended to the logic CFT [Smolka & Treinen 1994], which in addition to FT has an *arity predicate* $x[F]$ for any finite set F of feature symbols, expressing that the root of x has precisely the features F [Backofen 1995; Backofen & Treinen 1998]. Note that in these logics one can only quantify over trees, not over feature symbols. The generalization to a two-sorted logic which allows for quantification over features is undecidable [Treinen 1993], but decidability can be recovered if one restricts the use of feature variables to talk about existence of features only [Treinen 1997]. All these decidable logics of trees have a non-elementary lower bound [Vorobyov 1996].

This chapter will start in Section 4.1 by describing transformation procedures for DXC (Definition 3.16). These transformations allow to decide the satisfiability of DXC. More importantly, they prepare the ground for quantifier elimination of decidability of the first-order theory. They will later on be improved upon so as to be usable in practice in a symbolic engine. See Chapter 5 for the improvement and Section 7.2 for their use.

In Section 4.2, we extend on this work to handle any formula of FTS. This relies heavily on results of Section 4.1 and on *weak quantifier elimination* [Malcev 1971]. Finally, in Section 4.3, we discuss various considerations, ranging from decidability in other models than $\mathcal{FT}$ to complexity and efficiency considerations.

4.1 Transforming DXC

In a first part, let us restrict ourselves to DXC only. In Section 4.1.1, we define a way to transform DXC following equivalences in our model $\mathcal{FT}$. These transformations allow us to detect unsatisfiability of formulas. They also yield formulas on which we have more control. The properties of yield formulas are discussed in Section 4.1.2. Finally, in Section 4.1.3, we mechanise these transformations.

4.1.1 Transformation Rules for Constraints – The System $\mathcal{R}_1$

Before we start, we need to set up a few preliminaries. Let us start to define what it means for a variable and a similarity atom to be *solved*. These will be defined in Definitions 4.1 and 4.2 respectively.

Definition 4.1 (Solved Variables)**.** A variable is said to be *solved* in a constraint c if it only occurs in a full similarity atom in c, that is: x is solved in c if there exists y and c' such that c is of the form $x =_\star y \wedge c'$ with $x \notin \mathcal{V}(c')$. We will denote by $\mathcal{V}_s(c)$ the set of solved variables of c. □

Definition 4.2 (Solved Similarity Atom)**.** A similarity atom is *solved* in a constraint c if one of its variables is solved in c. □

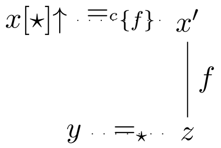


Figure 4.1: Formula 4.1

Note that the fact for a similarity atom to contain a solved variable implies that this variable appears only in this similarity atom and that this similarity atom carries the full set $\star$. As an example, consider Formula 4.1. A graphical representation can be found in Figure 4.1.

$$x[\star]\uparrow \wedge x =_{c\{f\}} x' \wedge x'[f]z \wedge y =_\star z \tag{4.1}$$

None of the variables x, x' and z are solved because they appear in other literals, namely $x[\star]\uparrow$ and $x'[f]z$. The variable y, however, is solved as it only occurs in one full similarity atom $y =_\star z$. The similarity atom $x =_{c\{f\}} x'$ is not solved because neither of its variables is, while $y =_\star z$ is.

We also need to define a notion of *subsumption of a literal by a constraint*. This is done in Definition 4.3. Subsumption is a weak, syntactic, form of implication: if a literal is subsumed by a constraint, then it is also implied by it. This is stated in Lemma 4.1.

Definition 4.3 (Subsumption)**.** A literal l is said to be *subsumed by* a constraint c, written $l \preceq c$, if $l \in c$ or if

- l is an absence atom $x[F]\uparrow$ with $F \subseteq \bigcup_{x[H]\uparrow \in c} H$,
- or l is a similarity atom $x =_F y$ with $F \subseteq \bigcup_{x =_H y \in c} H$,
- or l is a negated absence atom $\neg x[F]\uparrow$ and there exists $\neg x[H]\uparrow \in c$ with $H \subseteq F$,
- or l is a negated similarity atom $x \neq_F y$ and there exists $x \neq_H y \in c$ with $H \subseteq F$. □

Lemma 4.1 (Implication of Subsumed Literals)**.** *If a literal l is subsumed by a constraint c, then c implies l. In other words, if $l \preceq c$, then $\models_{\mathcal{FT}} \tilde{\forall} \cdot (c \rightarrow l)$.*

For instance, if we consider the constraint c defined as $x[f,g]\uparrow \wedge y =_{\{f,g\}} z \wedge y =_{\{h,k\}} z \wedge \neg z[g]\uparrow \wedge y \neq_{k,l} z$, then it subsumes $x[f]\uparrow$, $y =_{\{f,g,h\}} z$ and $\neg z[g,h]\uparrow$. c thus also implies these literals. Although c also implies $y \neq_{\{l\}} z$, it does not subsume it.

Let us now dive into the main topic of this subsection and define transformation rules of the form

$$\textsc{name} \qquad \textit{pattern} \quad \Rightarrow \quad \textit{replacement} \qquad (\textit{condition})$$

The patterns are constraints that contain meta-variables, meta-features and meta-feature sets. We show later how we lift these rules to DXC. A rule *applies* to a constraint c when

Clash Rules

C-Cycle	$x[f]y \wedge \bigwedge_{i=0}^{n-1} z_i[f_i]z_{i+1} \wedge c$	$\Rightarrow$	$\perp$	$(n \geq 1, y = z_0, x = z_n)$
C-Feat-Abs	$x[f]y \wedge x[F]\uparrow \wedge c$	$\Rightarrow$	$\perp$	$(f \in F)$
C-NAbsEmpty	$\neg x[\varnothing]\uparrow \wedge c$	$\Rightarrow$	$\perp$	
C-NSimRefl	$x \neq_F x \wedge c$	$\Rightarrow$	$\perp$	
C-NSimEmpty	$x \neq_\varnothing y \wedge c$	$\Rightarrow$	$\perp$	

Figure 4.2: Clash rules in system $\mathcal{R}_1$

(a) Its pattern matches the whole constraint c and does not mention any solved variable of c. The matching of pattern is considered modulo commutativity and associativity of $\wedge$. (In other words, we see constraints as sets of literals.)

(b) The side-condition – if there is one – is respected. Most of the side-conditions are present to ensure that rules are equivalences but some of them ensure the termination of our system of rules. Some side-conditions state that a literal to be introduced should not be *subsumed by* the whole constraint. Roughly speaking, this means that the literal does actually bring new non-trivial knowledge.

A constraint to which no rule applies is said to be *irreducible*.

Transformation rules yield other formulas that are not necessarily constraints: they can contain disjunctions and existential quantification. They can also simply be $\perp$. Thus, we cannot immediately apply another rule on them. However, these output formulas are always Σ_1-formulas. The idea basically consists in putting the output formula back into DXC and to continue on all the resulting constraints. This is explained in detail in Section 4.1.2.

Let us first define the *clash* rules that detect unsatisfiability in formulas. Consider the 5 transformation rules of Figure 4.2.

Some of them simply state that a literal is trivially false – or that its negation is trivially true. This is for instance the case of C-NAbsEmpty which simply states that $x[\varnothing]\uparrow$ is valid. The rule C-Feat-Abs detects contradictory information, namely that a feature is both present and absent in the same tree. The rule C-Cycle is necessary because our model does not allow for infinite trees.[1]

To these clash rules, let us add 5 transformation rules on positive literals. They are shown in Figure 4.3. This introduces three kinds of rules:

- *Deduction* rules are rules that create a new literal out of others of a different kind. The rule D-Feats, for instance, uses the unicity of features in the node of a feature tree to deduce full similarity of variables.
- *Propagation* rules are rules that are specific to the similarity literal. Since this literal implies that two variables behave the same, a lot of information can propagate from one to the other, allowing to later detect clashes.
- *Global* rules are rules that potentially modify and unbounded number of literals in the constraint. This is the case of G-SimFull that rewrites one variable into another, changing as many literals as necessary.

[1] This shows that the model of all of finite feature trees – that is the one we consider here – and the model of infinite feature trees do not share the same theory: $x[f]x$ is a clash in the former and satisfiable in the latter.

Deduction Rules

D-FEATS	$x[f]y \wedge x[f]z \wedge c$	$\Rightarrow$	$y =_\star z \wedge x[f]y \wedge x[f]z \wedge c$	$(y \neq z, y =_\star z \not\in c)$

Propagation Rules

P-FEAT-SIM	$x[f]y \wedge x =_G z \wedge c$	$\Rightarrow$	$z[f]y \wedge x[f]y \wedge x =_G z \wedge c$	$(f \in G, z[f]y \not\in c)$
P-ABS-SIM	$x[F]\uparrow \wedge x =_G z \wedge c$	$\Rightarrow$	$z[F \cap G]\uparrow \wedge x[F]\uparrow \wedge x =_G z \wedge c$	$(z[F \cap G]\uparrow \not\in c)$
P-SIMS	$x =_F y \wedge x =_G z \wedge c$	$\Rightarrow$	$y =_{F \cap G} z \wedge x =_F y \wedge x =_G z \wedge c$	$(y =_{F \cap G} z \not\in c)$

Global Rules

G-SIMFULL	$x =_\star y \wedge c$	$\Rightarrow$	$x =_\star y \wedge c\{x \mapsto y\}$	$(x, y \in \mathcal{V}(c))$

Figure 4.3: Transformation rules for positive literals in system $\mathcal{R}_1$

(0)			$r[f]x \wedge x[g]z \wedge r[f]y \wedge y[g,h]\uparrow$
(1)	D-FEATS	$\Rightarrow$	$r[f]x \wedge x[g]z \wedge r[f]y \wedge y[g,h]\uparrow \wedge x =_\star y$
(2)	P-ABS-SIM	$\Rightarrow$	$r[f]x \wedge x[g]z \wedge r[f]y \wedge y[g,h]\uparrow \wedge x =_\star y \wedge x[g,h]\uparrow$
(3)	C-FEAT-ABS	$\Rightarrow$	$\bot$

Figure 4.5: Transformation of Formula 4.2 using $\mathcal{R}_1$

In this whole document, we follow a convention for the naming of the rules. The name of a rule always starts by the initial letter of its kind. At this point, this can be C- for clash rules, D- for deduction rules, P- for propagation rules and G- for global rules. After that, the rule name contains the list of literals that occur in the pattern, potentially with a precision if necessary. For instance, these can be FEAT, NABS or NSIMREFL for a feature atom, negated absence atom or reflexive negated similarity atom respectively. As an example, C-NABSEMPTY reads as "the clash of a negated absence atom when it is empty" and P-FEAT-SIM reads as "the propagation of a feature atom through a similarity atom".

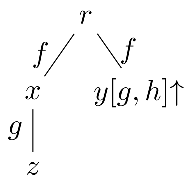


Figure 4.4: Formula 4.2

Let us now consider Formula 4.2. A graphical representation can be found in Figure 4.4.

$$r[f]x \wedge x[g]z \wedge r[f]y \wedge y[g,h]\uparrow \tag{4.2}$$

It states that the root r admits two children with the name f – x and y. The child x itself has a child via g, z. The child y, on the other hand, does not have children in either g or h. In fact, since x and y have the same name from r, they have to be identified together. Moreover, since x has a child in g but y is known to not have one, the whole formula is unsatisfiable. We can expect our system of transformation rules to detect this. Let us thus describe one possible transformation chain involving rules of Figures 4.2 and 4.3. The intermediary constraints can be found in Figure 4.5.

(0) Start from the example constraint. Notice the pattern $r[f]x \wedge r[f]y$. The rule D-FEATS can apply.

(1) Rewrite the constraint using D-FEATS. This adds a new literal $x =_\star y$ and creates the pattern $y[g,h]\uparrow \wedge x =_\star y$, meaning that the rule P-ABS-SIM can apply. In fact, we could also notice the pattern $x[g]z \wedge x =_\star y$, meaning that P-FEAT-SIM can apply. We will stay with the former for this example.

Deduction Rules

D-NFeat $\quad \neg x[f]y \wedge c \quad \Rightarrow \quad (x[f]\uparrow \vee \exists z \cdot (x[f]z \wedge y \neq_{\star} z)) \wedge c$

D-NSim-Feat $\quad x \neq_{\{f\}} y \wedge x[f]z \wedge c \quad \Rightarrow \quad \neg y[f]z \wedge x[f]z \wedge c$

D-NSim-Abs $\quad x \neq_{F} y \wedge x[G]\uparrow \wedge c \quad \Rightarrow \quad \neg y[F]\uparrow \wedge x[G]\uparrow \wedge c \qquad (F \subseteq G)$

Propagation Rules

P-NAbs-Sim $\quad \neg x[F]\uparrow \wedge x =_{G} z \wedge c \quad \Rightarrow \quad \neg z[F]\uparrow \wedge \neg x[F]\uparrow \wedge x =_{G} z \wedge c$
$(F \subseteq G, \neg z[F]\uparrow \not\preceq c)$

P-NSim-Sim $\quad x \neq_{F} y \wedge x =_{G} z \wedge c \quad \Rightarrow \quad z \neq_{F} y \wedge x \neq_{F} y \wedge x =_{G} z \wedge c$
$(F \subseteq G, z \neq_{F} y \not\preceq c)$

Refinement Rules

R-NAbs-Abs $\quad \neg x[F]\uparrow \wedge x[G]\uparrow \wedge c \quad \Rightarrow \quad \neg x[F \backslash G]\uparrow \wedge x[G]\uparrow \wedge c$

Splitting Rules

S-NAbs-Sim $\quad \neg x[F]\uparrow \wedge x =_{G} z \wedge c \quad \Rightarrow \quad (\neg x[F \cap G]\uparrow \vee \neg x[F \backslash G]\uparrow) \wedge x =_{G} z \wedge c$
$(F \not\subseteq G, F \not\subseteq {}^{c}G)$

S-NSim-Feat $\quad x \neq_{F} y \wedge x[f]z \wedge c \quad \Rightarrow \quad (x \neq_{\{f\}} y \vee x \neq_{F \backslash \{f\}} y) \wedge x[f]z \wedge c$
$(F \text{ finite}, f \in F, F \neq \{f\})$

S-NSim-Abs $\quad x \neq_{F} y \wedge x[G]\uparrow \wedge c \quad \Rightarrow \quad (x \neq_{F \cap G} y \vee x \neq_{F \backslash G} y) \wedge x[G]\uparrow \wedge c$
$(F \not\subseteq G, F \not\subseteq {}^{c}G)$

S-NSim-Sim $\quad x \neq_{F} y \wedge x =_{G} z \wedge c \quad \Rightarrow \quad (x \neq_{F \cap G} y \vee x \neq_{F \backslash G} y) \wedge x =_{G} z \wedge c$
$(F \not\subseteq G, F \not\subseteq {}^{c}G)$

Figure 4.6: Transformation rules for positive and negative literals in system $\mathcal{R}_1$

(2) Rewrite the constraint using P-Abs-Sim. This adds a new literal $x[g,h]\uparrow$ and creates the pattern $x[g]z \wedge x[g,h]\uparrow$, meaning that the rule C-Feat-Abs can apply.

(3) Rewrite the constraint using C-Feat-Abs. This leads to $\bot$ which is not matched by any pattern.

Note at this point that all the transformations that appeared in that chain were in fact equivalences. This means that Formula 4.2 is equivalent to $\bot$ in our model of feature trees. In other words, it is unsatisfiable.

Finally, let us add 10 transformation rules for negative literals. They can be found in Figure 4.6. Note that the rule S-NSim-Feat has a side-condition which requires a set to be finite. This side-condition is here solely for the purpose of termination. All the rules of Figures 4.2, 4.3 and 4.6 together form the system of transformation rules $\mathcal{R}_1$. For practicality all the rules of $\mathcal{R}_1$ can be found Figure 4.7.

Several things deserve to be noted here. Firstly, rules do not necessarily yield constraints. D-NFeat, for instance, introduces a disjunction and also a new existentially-quantified variable. Secondly, and contrary to the rules of Figure 4.3, the new propagation rules have an extra side-condition that requires the set of the negated absence or similarity atoms to be included in the set of the similarity atom. This means in particular that not all negated atoms can propagate through a similarity atom. Finally, these new rules show two new kinds of rules:

- *Refinement* rules (starting with R-) are rules that modify a literal towards something more precise. They are used here to transform the size of sets in negated absence and similarity literals. This can

Clash Rules

C-Cycle	$x[f]y \wedge \bigwedge_{i=0}^{n-1} z_i[f_i]z_{i+1} \wedge c$	$\Rightarrow$	$\bot$	$(n \geq 1, y = z_0, x = z_n)$
C-Feat-Abs	$x[f]y \wedge x[F]\uparrow \wedge c$	$\Rightarrow$	$\bot$	$(f \in F)$
C-NAbsEmpty	$\neg x[\varnothing]\uparrow \wedge c$	$\Rightarrow$	$\bot$	
C-NSimRefl	$x \neq_F x \wedge c$	$\Rightarrow$	$\bot$	
C-NSimEmpty	$x \neq_\varnothing y \wedge c$	$\Rightarrow$	$\bot$	

Deduction Rules

D-Feats	$x[f]y \wedge x[f]z \wedge c$	$\Rightarrow$	$y =_\star z \wedge x[f]y \wedge x[f]z \wedge c$	$(y \neq z, y =_\star z \npreceq c)$
D-NFeat	$\neg x[f]y \wedge c$	$\Rightarrow$	$(x[f]\uparrow \vee \exists z \cdot (x[f]z \wedge y \neq_\star z)) \wedge c$	
D-NSim-Feat	$x \neq_{\{f\}} y \wedge x[f]z \wedge c$	$\Rightarrow$	$\neg y[f]z \wedge x[f]z \wedge c$	
D-NSim-Abs	$x \neq_F y \wedge x[G]\uparrow \wedge c$	$\Rightarrow$	$\neg y[F]\uparrow \wedge x[G]\uparrow \wedge c$	$(F \subseteq G)$

Propagation Rules

P-Feat-Sim	$x[f]y \wedge x =_G z \wedge c$	$\Rightarrow$	$z[f]y \wedge x[f]y \wedge x =_G z \wedge c$	$(f \in G, z[f]y \npreceq c)$
P-Abs-Sim	$x[F]\uparrow \wedge x =_G z \wedge c$	$\Rightarrow$	$z[F \cap G]\uparrow \wedge x[F]\uparrow \wedge x =_G z \wedge c$	$(z[F \cap G]\uparrow \npreceq c)$
P-NAbs-Sim	$\neg x[F]\uparrow \wedge x =_G z \wedge c$	$\Rightarrow$	$\neg z[F]\uparrow \wedge \neg x[F]\uparrow \wedge x =_G z \wedge c$	$(F \subseteq G, \neg z[F]\uparrow \npreceq c)$
P-Sims	$x =_F y \wedge x =_G z \wedge c$	$\Rightarrow$	$y =_{F \cap G} z \wedge x =_F y \wedge x =_G z \wedge c$	$(y =_{F \cap G} z \npreceq c)$
P-NSim-Sim	$x \neq_F y \wedge x =_G z \wedge c$	$\Rightarrow$	$z \neq_F y \wedge x \neq_F y \wedge x =_G z \wedge c$	$(F \subseteq G, z \neq_F y \npreceq c)$

Refinement Rules

R-NAbs-Abs	$\neg x[F]\uparrow \wedge x[G]\uparrow \wedge c$	$\Rightarrow$	$\neg x[F \backslash G]\uparrow \wedge x[G]\uparrow \wedge c$	

Splitting Rules

S-NAbs-Sim	$\neg x[F]\uparrow \wedge x =_G z \wedge c$	$\Rightarrow$	$(\neg x[F \cap G]\uparrow \vee \neg x[F \backslash G]\uparrow) \wedge x =_G z \wedge c$	$(F \not\subseteq G, F \not\subseteq {}^cG)$
S-NSim-Feat	$x \neq_F y \wedge x[f]z \wedge c$	$\Rightarrow$	$(x \neq_{\{f\}} y \vee x \neq_{F \backslash \{f\}} y) \wedge x[f]z \wedge c$	$(F \text{ finite}, f \in F, F \neq \{f\})$
S-NSim-Abs	$x \neq_F y \wedge x[G]\uparrow \wedge c$	$\Rightarrow$	$(x \neq_{F \cap G} y \vee x \neq_{F \backslash G} y) \wedge x[G]\uparrow \wedge c$	$(F \not\subseteq G, F \not\subseteq {}^cG)$
S-NSim-Sim	$x \neq_F y \wedge x =_G z \wedge c$	$\Rightarrow$	$(x \neq_{F \cap G} y \vee x \neq_{F \backslash G} y) \wedge x =_G z \wedge c$	$(F \not\subseteq G, F \not\subseteq {}^cG)$

Global Rules

G-SimFull	$x =_\star y \wedge c$	$\Rightarrow$	$x =_\star y \wedge c\{x \mapsto y\}$	$(x, y \in \mathcal{V}(c))$

Figure 4.7: System $\mathcal{R}_1$ of Transformation Rules

$$\begin{array}{rrcl}
(0) & & & x \neq_{\{f,g,h\}} y \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow} \\
(1) & \textsc{S-NSim-Abs} & \Rightarrow & (x \neq_{\{f,g\}} y \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}) \\
 & & & \vee\, (x \neq_{\{h\}} y \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}) \\
 & & & (x \neq_{\{f\}} y \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}) \\
(2) & \textsc{S-NSim-Abs} & \Rightarrow & \vee\, (x \neq_{\{g\}} y \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}) \\
 & & & \vee\, (x \neq_{\{h\}} y \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}) \\
 & & & (\neg y[f]{\uparrow} \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}) \\
(3) & \textsc{D-NSim-Abs}^{3} & \Rightarrow & \vee\, (\neg y[g]{\uparrow} \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}) \\
 & & & \vee\, (\neg x[h]{\uparrow} \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}) \\
(4) & \textsc{R-NAbs-Abs} & \Rightarrow & (\neg y[f]{\uparrow} \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}) \\
 & +\ \textsc{C-NAbsEmpty} & & \vee\, (\neg x[h]{\uparrow} \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow})
\end{array}$$

Figure 4.8: Transformation of Formula 4.3 using $\mathcal{R}_1$

allow us in turn to detect clash with C-NAbsEmpty or C-NSimEmpty if the set turns out to be empty.

- *Splitting* rules (starting with s-) are a form of refinement rules that introduce disjunctions by splitting one predicate into two more refined versions. Refinement rules can quite often be seen as degenerated cases of splitting rules. These splitting rules are necessary to break down negated absence and similarity literals, allowing them in turn to be propagated.

As an example, let us consider Formula 4.3.

$$x \neq_{\{f,g,h\}} y \wedge x[f,g]{\uparrow} \wedge y[g,h]{\uparrow} \tag{4.3}$$

It expresses that the two variables x and y must have a difference somewhere in the names f, g or h. This is not so easy as both x and y are constrained: x cannot use the names f and g while y cannot use the names g and h. We can expect this formula to be satisfiable as long as x has a feature in h and/or y has a feature in f. Let us describe one possible transformation chain involving rules of $\mathcal{R}_1$. The intermediary constraints can be found in Figure 4.8.

(0) Start from the example constraint. Notice the pattern $x \neq_{\{f,g,h\}} y \wedge x[f,g]{\uparrow}$. The rule S-NSim-Abs can apply.

(1) Rewrite the constraint using S-NSim-Abs. This splits the negated similarity atom $x \neq_{\{f,g,h\}} y$ into $(x \neq_{\{f,g\}} y \vee x \neq_{\{h\}} y$. In DXC, this gives us two constraints. Notice the pattern $x \neq_{\{f,g\}} y \wedge y[g,h]{\uparrow}$. The rule S-NSim-Abs can apply again.

(2) Rewrite the constraint using S-NSim-Abs. This splits the negated similarity atom $x \neq_{\{f,g\}} y$ into $(x \neq_{\{f\}} y \vee x \neq_{\{g\}} y$. This gives us three constraints in total that only differ on the set carried by the negated similarity atom – $\{f\}$, $\{g\}$ or $\{h\}$. Notice the pattern $x \neq_{\{f\}} y \wedge x[f,g]{\uparrow}$. In fact, similar patterns are present in all three constraints. The rule D-NSim-Abs can apply in all of them.

(3) Rewrite each constraint using D-NSim-Abs. This replaces the negated similarity atoms by negated absence atoms $\neg y[f]{\uparrow}$, $\neg y[g]{\uparrow}$ and $\neg x[h]{\uparrow}$. The first and third ones belong to now irreducible constraints. The second one, $\neg y[g]{\uparrow}$ forms a pattern with $y[g,h]{\uparrow}$: R-NAbs-Abs can apply.

(4) Rewrite this constraint using R-NAbs-Abs. This replaces $\neg y[g]{\uparrow}$ by $\neg y[\varnothing]{\uparrow}$ which can trigger the clash C-NAbsEmpty. The resulting DXC contains two constraints. These constraints share the part $x[f,g]{\uparrow} \wedge y[g,h]{\uparrow}$ but differ on the negated absence atom that they carry, either $\neg y[f]{\uparrow}$ or $\neg x[h]{\uparrow}$.

> This disjunction expresses exactly what we had foreseen: the formula implies that either x has the feature h or y has the feature f.

As for the other example, note that all the transformations that appeared in that chain were equivalences. This is a property of all the rules of $\mathcal{R}_1$, which is stated in Lemma 4.2.

Lemma 4.2 (Rules of $\mathcal{R}_1$ perform equivalences)**.** *For any constraint c, if c transforms to ϕ via a rule of $\mathcal{R}_1$, then c and ϕ are equivalent. In other words, if $c \Rightarrow \phi$ via $\mathcal{R}_1$, then $\models_{\mathcal{FT}} \tilde{\forall} \cdot (c \leftrightarrow \phi)$.*

Proof. Let c and ϕ be a constraint and a Σ_1-formula such that $c \Rightarrow \phi$. Assume, for instance, that this transformation happened via D-Feats. Take x, y, z, f and c' such that the pattern that matched c is $x[f]y \wedge x[f]z \wedge c'$. In this situation, we have that ϕ is $y =_\star z \wedge x[f]y \wedge x[f]z \wedge c'$. Of course, c is included in ϕ and therefore ϕ implies c.

Let us show the other implication. Take ρ a valuation such that $\rho \models_{\mathcal{FT}} c$. In particular, $\rho \models_{\mathcal{FT}} x[f]y$ and therefore, by Definition 3.8, $x, y \in \mathtt{dom}(\rho)$, $f \in \mathtt{dom}(\rho(f))$ and $\rho(x)(f) = \rho(y)$. Similarly, because $\rho \models_{\mathcal{FT}} x[f]z$, $z \in \mathtt{dom}(\rho)$ and $\rho(x)(f) = \rho(z)$. These two facts together give us $\rho(y) = \rho(z)$. This here is the observational equality of functions and is therefore equivalent to $\rho(y) =_\star \rho(z)$, from which we can conclude that $\rho \models_{\mathcal{FT}} y =_\star z$ and, in turn, that $\rho \models_{\mathcal{FT}} \phi$. □

There are natural questions that arise when considering such a system: does this system of rules terminate? is it complete – for some definition of complete? is it confluent? The quick answers are: yes, the system of rules terminates; yes, it is complete; and, as a matter of fact, it is confluent, but this really does not matter in our situation because all rules perform equivalence. All theses properties will be discussed in Section 4.1.2.

Despite their presentation, the rules of $\mathcal{R}_1$ – G-SimFull aside – are local in the sense that one does not need to explore the whole constraint to decide whether they are applicable or not. To be exact, given a valuation from meta variables to variables, one does not need to explore the whole constraint to decide which rules are applicable. This is the case even if the subsumption (Definition 4.3) gives the impression to talk about the whole formula. In fact, it is local because it only requires to consider literals that share the same variables as the literal to be subsumed. This is stated in Lemma 4.3.

Lemma 4.3 (Locality of Subsumption)**.** *A literal is subsumed by a constraint if and only if it is subsumed by the part of the constraints that mentions strictly its variables.*[2]

Subsumption is also local in the sense that adding new literals does not change the literals subsumed by a constraint. If anything, the constraint only gets stronger and subsumes more literals. This is stated in Lemma 4.4.

Lemma 4.4 (Monotony of Subsumption)**.** *If a literal is subsumed by a constraint, then it is also subsumed by any extension of this constraint. In other words, for any literal l and constraints c and c', if $l \preceq c$ then $l \preceq c \wedge c'$.*

Lemmas 4.3 and 4.4 both follow immediately from Definition 4.3.

4.1.2 Properties of Irreducible Constraints of $\mathcal{R}_1$

This system of rules is interesting in that it allows to detect unsatisfiability of constraints. It is in fact stronger than that as it has the property of *garbage collection.* This property states that existentially-

[2] For a literal l and a constraint c, this will later be expressible as the global part of c with respect to the variables not in l, $\mathcal{G}_{^c\mathcal{V}(l)}(c)$.

quantified variables that are not reachable from free variables carry only redundant information and can thus safely be removed. To state it formally, let us first define in Definition 4.4 what it means for a set to be *ancestor-closed.*

Definition 4.4 (Ancestor-Closedness)**.** A set of variables X is *ancestor-closed with respect to a constraint* c if for all feature literals $x[f]y \in c$, if $y \in X$ then $x \in X$. □

When the constraint is obvious from context, it will be omitted and we will simply talk about an *ancestor-closed set of variables.*

As an example, let us consider Formula 4.4. A graphical representation is given in Figure 4.9.

$$\exists r, y, z \cdot \left(\begin{array}{c} r[f]x \wedge r[g]y \wedge y[h]z \wedge w[i]z \\ \wedge\, r =^{c}_{\{f,g,i\}} w \wedge y[^{c}\{h,i\}]\uparrow \wedge x \neq_{\{f,h\}} z \end{array}\right) \qquad (4.4)$$

In this constraint, the sets $\{r\}$, $\{r, x\}$, $\{r, y\}$, and $\{r, w\}$ are all ancestor-closed. The sets $\{r, y, z\}$ and $\{x, w\}$ are not, because z is accessible from w and x is accessible from r.

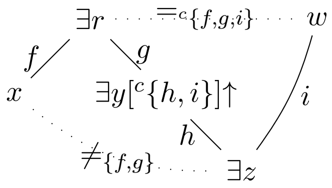


Figure 4.9: Formula 4.4

Let us now explain in Definition 4.5 how to split syntactically a constraint with respect to a set of variables, putting literals that mention variables in that set on one *local* side and literals that do not mention such variables on one *global* side.

Definition 4.5 (Global and Local Parts)**.** For any constraint c and any set of variables X, the local part of c with respect to X, written $\mathcal{L}_X(c)$, contains the literals of c that mention at least a variable of X and the global part of c with respect to X, written $\mathcal{G}_X(c)$, contains all the others. In other words:

$$\begin{aligned} \mathcal{L}_X(c) &= \{l \mid \mathcal{V}(l) \cap X \neq \varnothing, l \in c\} \\ \mathcal{G}_X(c) &= \{l \mid \mathcal{V}(l) \cap X = \varnothing, l \in c\} \end{aligned}$$

□

If we take Formula 4.4 again, and name its constraint c, we have for instance:

$$\begin{aligned} \mathcal{L}_{\{r\}}(c) &= r[f]x \wedge r[g]y \wedge r =^{c}_{\{f,g,i\}} w \\ \mathcal{G}_{\{r\}}(c) &= y[h]z \wedge w[i]z \wedge y[^{c}\{h,i\}]\uparrow \wedge x \neq_{\{f,h\}} z \\ \mathcal{L}_{\{r,y\}}(c) &= r[f]x \wedge r[g]y \wedge y[h]z \wedge r =^{c}_{\{f,g,i\}} w \wedge y[^{c}\{h,i\}]\uparrow \\ \mathcal{G}_{\{r,y\}}(c) &= w[i]z \wedge x \neq_{\{f,h\}} z \end{aligned}$$

Lemma 4.5 describes general properties of such global and local parts.

Lemma 4.5 (Properties of Global and Local Parts)**.** *For all constraint c, sets of variables X and Y:*

$$\begin{aligned} \mathcal{L}_X(c) \wedge \mathcal{G}_X(c) &= c \\ \mathcal{L}_\varnothing(c) &= \varnothing = \top \\ \mathcal{G}_\varnothing(c) &= c \\ \mathcal{L}_X(\mathcal{L}_Y(c)) &= \mathcal{L}_{X\cap Y}(c) \\ \mathcal{G}_X(\mathcal{G}_Y(c)) &= \mathcal{G}_{X\cup Y}(c) \\ \mathcal{L}_X(c) &\subseteq \mathcal{L}_Y(c) \qquad \textit{if } X \subseteq Y \\ \mathcal{G}_Y(c) &\subseteq \mathcal{G}_X(c) \qquad \textit{if } X \subseteq Y \end{aligned}$$

Lemma 4.6. *If c is irreducible with respect to $\mathcal{R}_1$, then for any set X, $\mathcal{G}_X(c)$ is irreducible with respect to $\mathcal{R}_1$.*

Proof. Let us take c a constraint irreducible with respect to $\mathcal{R}_1$ and X a set of variables of c. We are going to prove that $\mathcal{G}_X(c)$ is irreducible with respect to P-NSim-Sim. All the other cases are similar.

Assume there is a pattern in $\mathcal{G}_X(c)$ for the rule P-NSim-Sim. That means there is $x \neq_F y$ and $x =_G z$ in $\mathcal{G}_X(c)$ with $F \subseteq G$. Of course, $\mathcal{G}_X(c) \subseteq c$ so the pattern is in c as well. Since c is irreducible with respect to P-NSim-Sim, this means that the side-condition is not respected in c and therefore that $z \neq_F y \preceq c$.

By Lemma 4.3, $z \neq_F y \preceq \mathcal{G}_{^c\{y,z\}}(c)$. Obviously, $y, z \in \mathcal{V}(\mathcal{G}_X(c))$ which implies that $y, z \notin X$ and, subsequently, $X \subseteq {}^c\{y, z\}$. By Lemma 4.5, $\mathcal{G}_{^c\{y,z\}}(c) \subseteq \mathcal{G}_X(c)$. By Lemma 4.4, $z \neq_F y \subseteq \mathcal{G}_X(c)$ and, therefore, P-NSim-Sim cannot apply with this pattern.

The key arguments are basically that taking the global part of a constraint is a global operation that concerns all the literals with the same variables, while the subsumption is local and monotonous. This means that either taking the global part of a constraint removes both the subsumers and the subsumee at the same time, or none of them. □

Finally, let us state the property of garbage collection in Theorem 4.1.

Theorem 4.1 (Garbage Collection of Irreducible Constraints)**.** *Let c be a constraint*[3] *that is irreducible with respect to $\mathcal{R}_1$. Let X be a set of variables of c that is ancestor-closed. Then $\exists X \cdot c$ is equivalent to the global part of c with respect to X. In other words:*

$$\models_{\mathcal{FT}} \tilde{\forall} \cdot ((\exists X \cdot c) \leftrightarrow \mathcal{G}_X(c))$$

Our example Formula 4.4 contains a constraint that is irreducible with respect to $\mathcal{R}_1$. We can thus apply Theorem 4.1 on any set of variables that is ancestor-closed. If we consider $\{r, y\}$, for instance, we get that $\exists r, y \cdot c$ is equivalent to $\mathcal{G}_{\{r,y\}}(c)$, from which we can conclude that:

$$\exists r, y, z \cdot \begin{pmatrix} r[f]x \land r[g]y \land y[h]z \land w[i]z \\ \land\, r =_{^c\{f,g,i\}} w \land y[^c\{h,i\}]\uparrow \land x \neq_{\{f,h\}} z \end{pmatrix} \quad \leftrightarrow \quad \exists z \cdot (w[i]z \land x \neq_{\{f,h\}} z)$$

Proof of Theorem 4.1. The implication from left to right is easy to prove, has nothing to do with ancestor-closedness of sets of variables or irreducibility with respect to any system, and comes directly from Definition 4.5. Indeed, for any set of variables X and constraint c:

$$\begin{array}{rcl} \exists X \cdot c & = & \exists X \cdot (\mathcal{G}_X(c) \land \mathcal{L}_X(c)) \\ & \leftrightarrow & \mathcal{G}_X(c) \land (\exists X \cdot \mathcal{L}_X(c)) \\ & \rightarrow & \mathcal{G}_X(c) \end{array}$$

which gives us, for any set of variables X: $\models_{\mathcal{FT}} \tilde{\forall} \cdot ((\exists X \cdot c) \rightarrow \mathcal{G}_X(c))$.

The proof of the other direction goes by first proving Theorem 4.1 on only one variable, and then generalising to any set by induction for sets of any size. Lemma 4.7 states the garbage collection for one variable. □

Lemma 4.7 (Garbage Collection of one Variable in an Irreducible Constraint)**.** *Let c be a constraint that is irreducible with respect to $\mathcal{R}_1$. Let x be a variable of c such that there is no $y[f]x$ in c for any y and f. Then the global part of c with respect to $\{x\}$ implies $\exists x \cdot c$. In other words:*

$$\models_{\mathcal{FT}} \tilde{\forall} \cdot (\mathcal{G}_{\{x\}}(c) \rightarrow (\exists x \cdot c))$$

[3] Remember that constraints cannot be $\bot$ as they are defined as sets of literals.

Proof of Theorem 4.1 (continued). Let us assume Lemma 4.7 for a moment and show Theorem 4.1 by induction on the size of the set X. The property of course holds for empty sets. Let us now assume that it holds for sets of size n and take X a set of variables of size $n+1$. Let us take c any constraint such that X is ancestor-closed in c and such that c is irreducible with respect to $\mathcal{R}_1$.

Since there are no cycles (because of C-CYCLE), there is a variable $x \in X$ such that there is no feature atom pointing towards x, ie. no y and f such that $y[f]x \in c$. $\{x\}$ is thus an ancestor-closed set of variables of c. We can thus apply Lemma 4.7 and we get:

$$\models_{\mathcal{FT}} \tilde{\forall} \cdot (\mathcal{G}_{\{x\}}(c) \rightarrow \exists x \cdot c)$$

$\mathcal{G}_{\{x\}}(c)$ is a formula that is irreducible with respect to $\mathcal{R}_1$, by Lemma 4.6. $Y = X \backslash \{x\}$ is a set of variables of size n that is ancestor-closed in $\mathcal{G}_{\{x\}}(c)$. We can thus apply our induction hypothesis and we get:

$$\models_{\mathcal{FT}} \tilde{\forall} \cdot (\mathcal{G}_Y(\mathcal{G}_{\{x\}}(c)) \rightarrow \exists Y \cdot \mathcal{G}_{\{x\}}(c))$$

Finally, $Y \cup \{x\} = X$ and $\mathcal{G}_Y(\mathcal{G}_x(c)) = \mathcal{G}_{Y \cup \{x\}}(c)$ and thus:

$$\models_{\mathcal{FT}} \tilde{\forall} \cdot (\mathcal{G}_X(c) \rightarrow \exists X \cdot c)$$

□

Proof of Lemma 4.7 (sketch). The proof of Lemma 4.7 will only be sketched here. The complete proof can be found in Section 4.A. The proof goes by taking μ that satisfies $\mathcal{G}_{\{x\}}(c)$ and showing that it can be extended on x to ρ that satisfies c. The extension goes by taking $\rho(y) = \mu(y)$ for all $y \neq x$, and:

$$\rho(x)(f) = \begin{cases} \mu(y) & \text{if } x[f]y \in c & (4.8) \\ \mu(y)(f) & \text{if } x =_F y \in c \text{ with } x \neq y,\ f \in F, \text{ and } f \in \mathtt{dom}(\mu(y)) & (4.9) \\ \mathit{fresh}(f) & \text{otherwise} & (4.10) \end{cases}$$

where *fresh* is a feature tree that is different from all trees in the image of μ. In other words, we extend μ to ρ in such a way that $\rho(x)$ respects all the feature atoms from x to another variable (case 4.8) and all the similarity atoms (case 4.9). In the remaining space, we add fresh features so that negated absence and similarity atoms are respected.

It remains to show that ρ is well defined (in particular, the cases 4.8 and 4.9 are not disjoint) and that it satisfies all the literals of c. □

From Theorem 4.1 follows directly Theorem 4.2 as a corollary, stating that irreducible constraints are satisfiable.

Theorem 4.2 (Satisfiability of Irreducible Constraints). *A constraint*[4] *c that is irreducible with respect to $\mathcal{R}_1$ is satisfiable.*

Proof. Take $X = \mathcal{V}(c)$. This is a trivial ancestor-closed set of variables of c. Moreover, $\mathcal{G}_X(c)$ is empty. This gives us $\models_{\mathcal{FT}} \tilde{\forall} \cdot ((\exists X \cdot c) \leftrightarrow \top)$ and thus $\models_{\mathcal{FT}} \tilde{\exists} \cdot c$. □

```
1  function apply-rule-1 (rule, constraint) : $\Sigma_1$-formula
2
3  function choose-rule-1 ($c$ : constraint) : rule
4    return the first rule of $\mathcal{R}_1$ applicable to $c$ of
5      - all clash rules
6      - G-SimFull or D-NFeat
7      - all other rules of $\mathcal{R}_1$ but P-NAbs-Sim and P-NSim-Sim
8      - P-NAbs-Sim or P-NSim-Sim
9
10 function transform-1-xc ($\exists X \cdot c$ : x-constraint) : DXC
11   if $c$ is reducible in $\mathcal{R}_1$
12     let $r$ = choose-rule-1(c)
13     let $\phi$ = apply-rule-1($r$, $c$)
14     let $d$ = DXC($\exists X \cdot \phi$)
15     return transform-1($d$)
16   else
17     return $\exists X \cdot c$
18
19 function transform-1 ($d$ : DXC) : DXC
20   let $\bigvee_i \exists X_i \cdot c_i$ = $d$
21   return $\bigvee_i$ transform-1-xc($\exists X_i \cdot c_i$)
```

Figure 4.10: Function `transform-1`

4.1.3 Deciding the Satisfiability of DXC

Note that Theorems 4.1 and 4.2 only apply to irreducible constraints. We are now going to extend that to DXC. We are going to build a function that, from any given DXC, yields an equivalent one that is either empty – ie. $\bot$ – or satisfiable.

Consider the functions `choose-rule-1`, `transform-1-xc` and `transform-1` defined in Figure 4.10. `transform-1-xc` takes an x-constraint as input, transforms it using rules of $\mathcal{R}_1$, following a strategy defined by `choose-rule-1`, and outputs a DXC. `transform-1` takes a DXC and applies `transform-1-xc` on all x-constraints in the DXC. It returns another DXC. `transform-1-xc` works as follows:

- It first checks (Line 11) if the given constraint is reducible with respect to $\mathcal{R}_1$. If it is not, the x-constraint is returned as is (Line 17).
- If the given constraint is reducible, `transform-1-xc` finds a rule of $\mathcal{R}_1$ (Line 12) according to a strategy described in `choose-rule-1` and applies it to c (Line 13). The obtained formula is named ϕ.
- The formula ϕ is put back in DXC (Line 14). The resulting DXC is named d. It is possible to apply `DXC` on ϕ because it only contains positive occurrences of existential quantifiers. Since ϕ was possibly $\bot$, d is possibly the empty DXC.
- `transform-1-xc` then proceeds by calling `transform-1` on the DXC d (Line 15). If d is empty, `transform-1` will directly return the empty DXC. Otherwise, it will call `transform-1-xc` on all the x-constraints in d.

This function terminates on all inputs, as stated in Lemma 4.8. It returns an equivalent DXC such that all its constraints are irreducible with respect to $\mathcal{R}_1$. This is stated in Lemma 4.9. A direct corollary is that the output DXC is either empty or satisfiable. This is stated in Lemma 4.10.

[4] See footnote 3.

Lemma 4.8 (Termination of `transform-1`). *The function `transform-1` terminates on all inputs.*

Proof of Lemma 4.8 (sketch). The proof of Lemma 4.8 turns out to be highly non-trivial. It will therefore only be sketched here. The complete proof can be found in Section 4.B.

The function `transform-1`, when called on a non-empty DXC, includes one or several calls to the function `transform-1-xc` (Line 21). The function `transform-1-xc`, when called on a reducible constraint, includes a call back to `transform-1` (Line 15). Depending on the rule r that `transform-1-xc` chooses via `choose-rule-1` (Line 12), the DXC that will be passed to a subsequent call to `transform-1` will contain zero (for a clash rule), one (for most other rules) or more (for rules introducing a disjunction) x-constraints, and thus as many calls to `transform-1-xc`.

We focus on the x-constraints that are taken as argument by `transform-1-xc`. We show that if `transform-1-xc` is called on an x-constraint c, then any subsequent recursive call to `transform-1-xc` will be given an x-constraint that is "smaller" than c. We need to define what it means for an x-constraint to be "smaller" than another one. Moreover, we have to show that it is not possible for x-constraints to become smaller and smaller indefinitely, that is there is no infinite chain of constraints c_0, c_1, etc. such that c_{i+1} is smaller than c_i for all i.

In a first part of the proof (Section 4.B.1), we simplify the problem by remarking several things.

- Firstly, we can always assume that the constraints are clash-free. Indeed, clash rules have the priority in `choose-rule-1` and, therefore, any constraint that is not clash-free will be transformed to $\perp$ in the next step and is therefore not a threat to termination.
- Secondly, the system of priority of `choose-rule-1` means that `transform-1` basically runs in three phases.
  - In a first phase, only the rules G-SimFull and D-NFeat are used.
  - In a second phase, almost all the rules of $\mathcal{R}_1$ are used. This excepts P-NAbs-Sim and P-NSim-Sim. Moreover, whenever a pattern for G-SimFull or D-NFeat appears, these rules are applied immediately. They can therefore be inlined in the rules that create the pattern. The second phase can therefore be seen as an alternative system of rules, which we will name $\mathcal{R}_1^{\text{trunc}}$ and which leaves constraints irreducible with respect to the first phase.
  - Finally, in a third phase, the rules P-NAbs-Sim and P-NSim-Sim are added. These rules turn out to respect the irreducibility with respect to the first and second phase. The termination of this third phase will therefore not be an issue.

Even with these simplifications, there remains quite a lot of difficulties, detailed in Section 4.B.2. Mostly, there are two major difficulties.

- The first difficulty comes from R-NAbs-Abs and from all the splitting rules (S-NAbs-Sim, S-NSim-Feat, S-NSim-Abs and S-NSim-Sim). Indeed, these transform a negated atom (absence or similarity) into a negated atom of the same kind carrying a strictly smaller set. There is however no guarantee that these sets cannot get smaller and smaller indefinitely as they can, of course, be infinite. We will however remark that even if these sets can be infinite, there is only a finite number of sets that can appear in a transformation from a finite constraint. We can leverage this fact to consider not the set themselves by their height in the *finite* lattice of possible sets. This is discussed in Section 4.B.3.
- The second difficulty comes from the interaction between the rules D-NSim-Feat and D-NFeat. The former removes a negated similarity atom, replacing it by a negated feature atom. The latter removes a negated feature atom, replacing it by a disjunction that includes a new variable and a negated

```
function garbage-collect-1-xc ($\exists X \cdot c$ : x-constraint) : x-constraint
  let $Y$ = biggest subset of $X$ ancestor-closed in $c$
  return $\exists X \backslash Y \cdot \mathcal{G}_Y(c)$

function garbage-collect-1 ($d$ : DXC) : DXC
  let $\bigvee_i \exists X_i \cdot c_i$ = $d$
  return $\bigvee_i$ garbage-collect-1-xc($\exists X_i \cdot c_i$)
```

Figure 4.11: Functions garbage-collect-1-xc and garbage-collect-1

similarity atom. This process can repeat several times. In fact, it can repeat at least a number of times linear in the number of initial variables in the constraint. We can however remark several facts:

- In order for a negated feature atom to transform into negated similarity atom and then back into negated feature atom, the presence of an absence or a similarity atom is required.
- Except in some specific cases, these absence and similarity atoms can only appear on variables that were present in the constraint initially, that is not on new variables introduced by D-NFeat.
- The newly introduced variables cannot lead to "initial" variables. Since there is a finite number of such "initial" variables and since they cannot form cycles (because the constraints are clash-free), this gives us a notion of "depth" in constraints.
- For a negated similarity atom to transform into negated feature atom and back, it has to increase its depth. Since there is only a finite depth of "initial" variables, the process has to stop eventually.

This is discussed in details in Section 4.B.4.

Tackling these two difficulties allows us to then define a well-founded decreasing lexicographic measure on constraints from which we can conclude that transform-1 terminates. This is discussed in Section 4.B.5. □

Lemma 4.9. *Given a DXC d, transform-1 yields a DXC d' that is equivalent to d and such that all the constraints of d' are irreducible with respect to $\mathcal{R}_1$.*

Proof. This comes from the fact that DXC and apply-rule-1 both return equivalent formulas, the former by definition, and the latter by Lemma 4.2. □

Lemma 4.10. *The function transform-1 yields DXC that are either empty or satisfiable.*

In other words, transform-1 meets exactly the goals that we gave ourselves. Firstly, it can be used as an unsatisfiability check. In fact, it is a *complete* unsatisfiability check in the sense that it detects unsatisfiability if and only if the DXC is indeed unsatisfiable. Secondly, it is incremental by nature. Since the input and output DXC are equivalent, the input DXC can be thrown away to keep only the output one. If one later adds other literals to the DXC, all the computation that have been done previously are still valid and only the computation that have to do with the new literals will take place.

Finally, we can clean up the result of transform-1 by leveraging Theorem 4.1. Consider the function garbage-collect-1 defined in Figure 4.11. garbage-collect-1 takes an x-constraint $\exists X \cdot c$ and removes the biggest subset of its quantifier block X that is ancestor-closed. The existence of such a biggest subset comes from the fact that the union of two ancestor-closed sets is also ancestor-closed.

Intuitively, one can see an x-constraint as a graph whose entry points are the free variables. Intuitively, garbage-collect-1-xc removes all the variables of an x-constraint that are not *reachable*. Let us first define reachability properly in Definition 4.6.

Definition 4.6 (Reachability of a variable in an x-constraint)**.** A variable x is *reachable from* y *in a constraint* c if there exists a chain of feature atoms that leads from y to x. A variable x is *reachable in an x-constraint* $\exists X \cdot c$ if it is reachable from a free variable of $\exists X \cdot c$. □

Note that the notion of reachability is strongly tied to that of ancestor-closedness. Indeed, in an x-constraint $\exists X \cdot c$, a variable x is non-reachable if and only if there exists a subset X' of X that is ancestor-closed in c.

We can now state in Lemma 4.11 that garbage-collect-1 only leaves non-reachable variables in a DXC. This property justifies the name of the function, but it will also prove to be very important in the next section to build a function that decides first-order formulas.

Lemma 4.11. *Given a DXC* d *whose constraints are irreducible with respect to* $\mathcal{R}_1$*, garbage-collect-1 yields a DXC* d' *that is equivalent to* d *and such that, in all its x-constraints, all the local variables are reachable.*

Proof. The fact that d' is equivalent to d follows directly from Theorem 4.1. The property of reachability in an x-constraint $\exists X \cdot$ comes from the fact that a variable x is reachable if and only if there is a subset of X that contains x and that is ancestor-closed. By definition of garbage-collect-1-xc however, we applied Theorem 4.1 on the biggest ancestor-closed subset. Since ancestor-closedness is a notion that is table by union, that means that the only ancestor-closed subset of X that remains is the empty one. Therefore, all the variables of X are reachable. □

4.2 First-Order Formulas

We have defined in Section 4.1.3 the function transform-1 that takes any DXC and yields an equivalent DXC that is either empty or such that all of its constraints are irreducible with respect to $\mathcal{R}_1$. We have then defined the function garbage-collect-1 that can apply on such DXC and that removes all non-reachable variables. We are going to show that this can be extended to a function deciding satisfiability of closed first-order formulas.

Our theory of feature trees does not have the property of quantifier elimination in the strict sense [Hodges 1993], that is it is not true that any formula has an equivalent quantifier-free formula. This is already the case without the similarity literals, as we can see in the following example: $\exists x \cdot (y[f]x \wedge x[g]\uparrow)$. This formula means that there is a tree denoted by x such that y points to x through the feature f, and that x does not have the feature g. A quantifier elimination procedure would have to conserve this information about the global variable y.

This situation is not unusual when designing decision procedures. There are basically two possible remedies. The first one is to extend the logical language by new predicates which express properties which otherwise would need existential quantifiers to express. This approach of achieving the property of quantifier elimination by extension of the logical language is well-known from Presburger arithmetic, it was also successfully used for feature tree logics in the past [Backofen & Smolka 1995; Backofen 1995]. However, in the case of feature tree logics, the needed extension of the language is substantial and requires the introduction of *path constraints*. For instance, the above formula would be equivalent to the path constraint $y[f][g]\uparrow$ stating that the variable y has a feature f pointing towards a tree where there is no feature g. Unfortunately, this extension entails the need of quite complex simplification rules for these new predicates.

The alternative solution is to our knowledge due to Mal'cev [Malcev 1971] and consists in exploiting the fact that certain predicates of the logic behave like functions. This solution was also used by Comon and Lescanne [Comon & Lescanne 1989] for Herbrand trees. When switching to feature trees, this solution becomes quite elegant [Treinen 1997]. The above formula would be replaced by $\neg y[f]\uparrow \wedge \forall x \cdot (y[f]x \rightarrow x[g]\uparrow)$ stating that y has a feature f (by $\neg y[f]\uparrow$) and that for each variable x such that y points towards x via f (in fact, there is only one), x has no feature g. The price is that existential quantifiers are not completely eliminated but switched for universal ones. This is, however, sufficient, since one can now apply this transformation to any PNF, and successively reduce the number of quantifier eliminations. For instance, consider the formula $\forall y \cdot \exists x \cdot (y[f]x \wedge x[g]\uparrow)$ which has one quantifier alternation. By applying the solution above, we can transform it to $\forall y \cdot (\neg y[f]\uparrow \wedge \forall x \cdot (y[f]x \rightarrow x[f]\uparrow))$ which has no quantifier alternation. The negation of this formula is then a closed Σ_1-formula which we can decide with our satisfiability check.

We show in Section 4.2.1 a way to switch existential quantifiers for universal quantifiers in formulas of FTS. We then use that in Section 4.2.2 to build a complete decision procedure for the first-order theory of $\mathcal{FT}$.

4.2.1 Switching Existential Quantifiers from DXC

Let us start by showing how to switch a block of existential quantifiers from an x-constraint in which all the variables are reachable. This goes by iteratively applying the rule G-Exists-Feat defined as follows:

$$\text{G-EXISTS-FEAT} \qquad \exists y, Z \cdot (x[f]y \wedge c) \quad \Rightarrow \quad \neg x[f]\uparrow \wedge \forall y \cdot (x[f]y \rightarrow \exists Z \cdot c) \qquad (x, y \notin Z, x \neq y)$$

This rule leverages the fact that features are functional in our model: a tree can not have two edges leaving the root with the same feature. The fact that it performs an equivalence is stated in Lemma 4.12. The fact that all the variables of Z remain reachable in $\exists Z \cdot c$ in stated in Lemma 4.13. This will allow us to reiterate the process until there is no existential quantification anymore and it has been replaced by universal quantification, as stated in Lemma 4.14.

Lemma 4.12. *The formula yield by* G-Exists-Feat *is always equivalent to the given x-constraint.*

Proof. This is shown by the following chain of equivalences:

$$\begin{array}{lll} & \exists y, Z \cdot (x[f]y \wedge c) & \\ \leftrightarrow & \exists y \cdot (x[f]y \wedge \exists Z \cdot c) & \text{since } x, y \notin Z \\ \leftrightarrow & \neg x[f]\uparrow \wedge \forall y \cdot (x[f]y \rightarrow \exists Z \cdot c) & \text{since features are functional} \end{array}$$ □

Lemma 4.13. *If all the variables of Z are reachable in $\exists y, Z \cdot (x[f]y \wedge c)$, then they are reachable in $\exists Z \cdot c$.*

Proof. Let us take $z \in Z$ that is reachable in $\exists y, Z \cdot (x[f]y \wedge c)$. That means that there is a chain of feature atoms from a variable of ${}^cZ$ to z. If that chain of feature mentions y, then the part that goes from y to z is still present in c. Since y is not existentially quantified in $\exists Z \cdot \phi$, z is reachable there. If the chain of feature does not mention y, then it is fully included in c and is still present there. □

```
function switch-xc ($c$ : x-constraint) : $\Pi_1$-formula
  match $c$
  | $\exists y, Z \cdot (x[f]y \wedge c')$ where $x, y \notin Z$ ->
    return $\neg x[f]\uparrow \wedge \forall y \cdot (x[f]y \rightarrow ($switch-xc$(\exists Z \cdot c')))$
  | _ -> return $c$

function switch ($d$ : DXC) : $\Pi_1$-formula
  let $\bigvee_i c_i$ = $d$ where ($c_i$ : x-constraint) for all $i$
  return $\bigvee_i$ switch-xc($c_i$)
```

Figure 4.12: Functions switch-xc and switch

```
1  function decide ($\phi$ : formula) : boolean
2    match PNF($\phi$)
3    | $\exists X \cdot \psi$ where $\psi$ is quantifier-free ->
4      match transform-1(DXC($\exists X \cdot \psi$))
5      | $\perp$ -> return false
6      | _ -> return true
7
8    | $Q \cdot \exists X \cdot \psi$ where $\psi$ is quantifier-free, $Q$ does not end in $\exists$ ->
9      let $d$ = transform-1(DXC($\exists X \cdot \psi$))
10     let $d'$ = garbage-collect-1($d$)
11     let $\chi$ = switch($d'$)
12     return decide($Q \cdot \chi$)
13
14   | $Q \cdot \psi$ where $\psi$ is quantifier-free ->
15     return not(decide($\overline{Q} \cdot \neg\psi$))
```

Figure 4.14: Function decide

Consider the function switch in Figure 4.12. It uses the rule G-Exists-Feat to switch every existential quantifier of a DXC into a universal one. For instance, when given Formula 4.5, the function switch returns Formula 4.6. A graphical representation of Formula 4.5 can be found in Figure 4.13.

$$\exists y, z \cdot (x[f]y \wedge y[g]z \wedge z[g]\uparrow \wedge z[h]w \wedge x =_\star w) \tag{4.5}$$

$$\neg x[f]\uparrow \wedge \forall y \cdot (x[f]y \rightarrow (\neg y[g]\uparrow \wedge \forall z \cdot (y[g]z \rightarrow (z[g]\uparrow \wedge z[h]w \wedge x =_\star w)))) \tag{4.6}$$

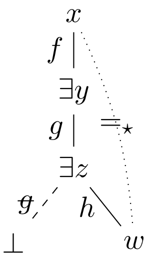


Figure 4.13: Formula 4.5

Similarly to the other functions in this section, the function switch performs equivalences. Moreover, it indeed returns a Π_1-formula when called on the right inputs. This is stated in Lemma 4.14.

Lemma 4.14. *Given a DXC d such that all its variables are reachable, the function switch terminates and yields a Π_1-formula ϕ that is equivalent to d.*

Proof. This comes directly from Lemma 4.12. □

4.2.2 Deciding the First-Order Theory of $\mathcal{FT}$

Finally, consider the function decide defined in Figure 4.14, Line 1. It accepts any formula ϕ and returns a boolean. It works as follows:

$$\begin{array}{lll}
(0) & & \texttt{decide}(\forall x \cdot ((x =_F y \wedge x[G]{\uparrow}) \rightarrow y[F \cap G]{\uparrow})) \\
(1) & \Rightarrow & \texttt{not}(\texttt{decide}(\exists x \cdot \neg((x =_F y \wedge x[G]{\uparrow}) \rightarrow y[F \cap G]{\uparrow}))) \\
(2) & \Rightarrow & \cdots\cdots\ \texttt{transform-1}(\exists x \cdot (x =_F y \wedge x[G]{\uparrow} \wedge \neg y[F \cap G]{\uparrow})) \\
(3) & \Rightarrow & \texttt{not}(\texttt{false}) \quad \Rightarrow \quad \texttt{true}
\end{array}$$

Figure 4.15: Internal steps of execution of decide on Formula 3.3

- The input formula ϕ is immediately transformed into prenex normal form – PNF – and matched upon (Line 2). In the rest of this explanation, we talk about ϕ and its PNF interchangeably.
- If ϕ is simply a Σ_1-formula, that is if it is of the form $\exists X \cdot \psi$ where ψ is quantifier-free (Line 3), we return `true` if ϕ is satisfiable and `false` otherwise, which is done using `transform-1` on the DXC of ϕ (Line 4). The algorithm is done.
- Otherwise, if ϕ is not a Σ_1-formula but its last block of quantifiers is existential, that is if ϕ is of the form $Q \cdot \exists X \cdot \psi$ where ψ is quantifier-free and Q does not end in $\exists$ (Line 8), we are exactly in the case that we described how to handle in Sections 4.1.3 and 4.2.1.
  - We first put the formula $\exists X \dot{\psi}$ in DXC and apply `transform-1` on it (Line 9). This returns an equivalent DXC d whose constraints are irreducible with respect to $\mathcal{R}_1$.
  - We can then apply the garbage collection via `garbage-collect-1` on d (Line 10). This returns an equivalent DXC d' which has all its variables reachable.
  - This is exactly the right context to switch the remaining existential quantifiers using `switch` in d' (Line 11). This returns an equivalent Π_1-formula χ.
  - We can then call `decide` recursively on $Q \cdot \chi$ (Line 12). Note that, since χ is a Π_1-formula, the number of quantifier alternations in $Q \cdot \chi$ is strictly smaller than that of $Q \cdot \exists X \cdot \psi$.
- Otherwise, if ϕ is not a Σ_1-formula and its last block of quantifiers is not existential, then it is of the form $Q \cdot \psi$ where ψ is quantifier-free (Line 14), we simply fall back on the previous case, which we know how to handle, by applying `decide` recursively on the negation of ϕ taking the negation of the result[5] (Line 15). The negation of $Q \cdot \psi$ is computed syntactically and is the formula $\overline{Q} \cdot \neg\psi$, where $\overline{Q}$ inverts the quantifiers. For instance, $\overline{\forall x \cdot \exists y \cdot \forall z} \cdot \psi = \exists x \cdot \forall y \cdot \exists z \cdot \psi$. $\overline{Q} \cdot \neg\psi$ is equivalent to $\neg\phi$, has as many quantifier alternations as ϕ, but its last block of quantifiers is now existential.

Let us first describe the execution of `decide` on two examples. As a first example, let us consider Formula 3.3. The intermediary steps are represented in Figure 4.15. Internal steps of execution are represented by the use of "$\cdots\cdots$".

(0) Start from Formula 3.3.

(1) The last block of quantification is universal. `decide` (Line 15) proceed thus recursively on the negation of this formula.

(2) Inside this call, `decide` receives a Σ_1-formula. It will therefore put it in DXC and call `transform-1` on it (Line 4). The steps of execution of `transform-1` are not detailed. In this case, `transform-1`

[5] Of course, such a recursive call can very easily be made tail-recursive and should be made tail-recursive in a real-world implementation. We do believe, however, that this makes the presentation heavier and will not be doing that in this thesis.

$$\begin{array}{lll}
(0) & & \texttt{decide}(\exists x, y \cdot \forall z \cdot (z \neq_F x \vee z \neq_F y)) \\
(1) & \Rightarrow & \texttt{not}(\texttt{decide}(\forall x, y \cdot \exists z \cdot \neg(z \neq_F x \vee z \neq_F y))) \\
(2) & \Rightarrow & \cdots\cdots\ \texttt{transform-1}(\exists z \cdot (z =_F x \wedge z =_F y)) \\
(3) & \Rightarrow & \cdots\cdots\ \texttt{garbage-collect-1}(\exists z \cdot (z =_F x \wedge z =_F y \wedge x =_F y)) \\
(4) & \Rightarrow & \cdots\cdots\ \texttt{switch}(x =_F y) \\
(5) & \Rightarrow & \texttt{not}(\texttt{decide}(\forall x, y \cdot x =_F y)) \\
(6) & \Rightarrow & \texttt{not}(\texttt{not}(\texttt{decide}(\exists x, y \cdot x \neq_F y))) \\
(7) & \Rightarrow & \cdots\cdots\ \texttt{transform-1}(\exists x, y \cdot x \neq_F y) \\
(8) & \Rightarrow & \texttt{not}(\texttt{not}(\texttt{false}))\ \text{ or }\ \texttt{not}(\texttt{not}(\texttt{true}))\ \text{ depending on } F
\end{array}$$

Figure 4.16: Application of `decide` on Formula 3.4

will propagate the absence atom through the similarity atom, and then detect a clash between these two, leading to replacing the whole formula by $\bot$.

(3) `decide` therefore returns `false`, which gets negated and the whole call returns `true`. In this case, it means that Formula 3.3, even if it is not a closed formula, is valid. This will be explained later.

As a second example, let us describe the steps of execution of `decide` on Formula 3.4. The intermediary steps are represented Figure 4.16.

(0) Start from Formula 3.4.

(1) The last block of quantification is universal. `decide` thus proceeds recursively on the negation of the formula (Line 15).

(2) Inside this call, `decide` puts the formula in DXC and calls `transform-1` (Line 9) on it. The steps of `transform-1` are not detailed. In this case, `transform-1` will propagate the similarity atoms and reach an irreducible form immediately.

(3) `decide` proceeds to call `garbage-collect-1` (Line 10) which cleans up the $\exists z$ that has become irrelevant, leaving only $x =_F y$.

(4) `decide` then calls `switch` (Line 11) on $x =_F y$. This formula has no quantifiers and thus it remains unchanged.

(5) This step of execution is done, and `decide` continues recursively (Line 12).

(6) The last block of quantification is universal again. `decide` thus proceeds recursively on the negation of the formula (Line 15).

(7) Inside this call, `decide` receives a Σ_1-formula. It calls `transform-1` (Line 4) on it. The result here depends heavily on F. If $F = \varnothing$, then `transform-1` will detect a clash and return $\bot$. Otherwise, `transform-1` will return the constraint unchanged as it is irreducible with respect to $\mathcal{R}_1$.

(8) Depending on F, `decide` will then proceed to return either `false`or `true`. That result is then negated twice. Formula 3.4 is thus valid if and only if F is not empty.

Lemma 4.15. *The function `decide` terminates on all inputs.*

Proof. All the calls from decide to other functions are safe: PNF, DXC, transform-1 (Lemma 4.8), garbage-collect-1 and switch (Lemma 4.14) all terminate. The only problem comes from the recursive calls of decide.

However, the recursive call Line 12 is on a formula with strictly less quantifier alternations. Indeed, on that line, Q is both non-empty and does not terminate on $\exists$. It must therefore terminate on $\forall$. Since ψ is the result of switch, it contains only universal quantifiers, and none of them are under a negation. The PNF of $Q \cdot \psi$ will thus have as many quantifier alternations as Q, which is one less than $Q \cdot \exists X$ on Line 8.

The recursive call Line 15 is on a formula with the same number of quantifier alternations. However, subsequent calls to decide will enter one of the cases Lines 3 and 8 which will, in turn, either return immediately or decrease the number of quantifier alternations. □

The function decide deserves its name as, if given a closed formula, it returns `true` if and only if the formula is satisfiable. This will be stated in Lemma 4.16. decide is thus a complete decision procedure for the first-order theory of $\mathcal{FT}$. This is stated in Theorem 4.3.

Lemma 4.16. *Given a closed formula ϕ, decide returns `true` if and only if ϕ is satisfiable.*

Proof. The function decide has three `return` statements that may return `true`, Lines 6, 12 and 15. Let us show by induction on the number of recursive calls that, for each formula ϕ, there is a status, either satisfiable or valid, such that decide(ϕ) is `true` if and only if ϕ has that status.

- On Line 6, decide returns `true` if and only if the given formula ϕ is satisfiable. This comes from the fact that PNF, DXC and transform-1 return equivalent formulas. Moreover, the result of transform-1 is not $\perp$ if and only if it is satisfiable (Theorem 4.2).
- On Line 12, decide(ϕ) is `true` if and only if decide($Q \cdot \chi$) is `true`. This comes from the fact that PNF, DXC, transform-1, garbage-collect-1 and switch, called in these contexts, return equivalent formulas. By induction hypothesis, there is a status – satisfiable or valid – such that decide($Q \cdot \chi$) – and thus decide(ϕ) – is `true` if and only if $Q \cdot \chi$ – and thus ϕ – has that status.
- On Line 15, decide(ϕ) is `true` if and only if decide($\overline{Q} \cdot \neg\psi$) if `false`. By induction hypothesis, there is a status, either satisfiable or valid, such that decide($\overline{Q} \cdot \neg\psi$) is `true` if and only if $\overline{Q} \cdot \neg\psi$ has that status. This means that decide(ϕ) is `true` if and only if $\overline{Q} \cdot \neg\psi$ and thus $\neg\phi$ does not have that status. We can conclude by adding that ϕ is satisfiable if and only if $\neg\phi$ is not valid, and ϕ is valid if and only if $\neg\phi$ is not satisfiable.

For any closed formula ϕ now, validity and satisfiability are the same. Therefore, decide(ϕ) is `true` if and only if ϕ is satisfiable.[6][7] □

Theorem 4.3. *The first order theory of $\mathcal{FT}$ is decidable.*

[6] An other way to prove Lemma 4.16 would have been to show that, if the given formula ϕ is closed, then the last DXC d' returned by transform-1 is closed too. In that case, satisfiability and validity are the same for d' and we can conclude immediately. This is of course true here as nothing in decide changes the free variables of a formula. This requires extra lemmas however to state this property on the rules of $\mathcal{R}_1$, transform-1, garbage-collect-1 and switch.

[7] Note that corollaries of this proof is that, on any formula, if decide returns `true`, then the given formula is satisfiable and if decide returns `false`, then the given formula is not valid.

4.3 Discussions

There are several topics from this chapter that deserve discussion. In Section 4.3.1, we discuss alternative models that could be considered in spite of $\mathcal{FT}$. Typically, this will answer the question: what about infinite trees? In Section 4.3.2, we bring in complexity considerations. Finally, in Section 4.3.3, we enumerate limitations of `transform-1` and $\mathcal{R}_1$ which make them unsuitable for our concrete use within the CoLiS project. This motivates the work of Chapter 5.

4.3.1 Alternative Models

In this whole chapter, we have been considering $\mathcal{FT}$, the model of finite feature trees of unbounded width and unbounded depth. We have shown that the first-order theory of $\mathcal{FT}$ is decidable. One might wonder if alternative models than $\mathcal{FT}$ also enjoy such a property. In particular, one might wonder if they could have the same first-order theory.

Firstly, since any finite feature tree can be described exactly by a finite formula, then any model that does not include all of $\mathcal{FT}$ has to have a different first order theory. This is in particular the case of all the feature trees of bounded depth, or of bounded width if the bound is strictly smaller than the size of the set of features $\mathcal{F}$. Indeed, it suffices to choose a tree t that is in $\mathcal{FT}$ and not in the alternative model. We can then choose a formula ϕ that has only one free variable x and that describes explicitly t in $\mathcal{FT}$, and only t. That is such that $[x \mapsto t] \models_{\mathcal{FT}} \phi$ and such that there is no t' with $[x \mapsto t'] \models_{\mathcal{FT}} \phi$. Since t is not in the alternative model, that means that ϕ is not satisfiable in that model, while it clearly is in $\mathcal{FT}$.

Let us now consider extensions of $\mathcal{FT}$ with infinite feature trees. The model $\mathcal{FT}_\infty$ of all possible feature trees (Definition 3.2) does not have the same first-order theory as $\mathcal{FT}$. Indeed, the formula $x[f]x$, for instance, is not satisfiable in $\mathcal{FT}$ but clearly is in $\mathcal{FT}_\infty$. In general, it is necessary for a model to not include trees that are subtrees of themselves if we are hoping to have the same first order theory as $\mathcal{FT}$.

Three fairly natural models that one might want to consider are thus:

1. all the feature trees of potentially infinite width but finite depth;
2. all the feature trees of finite width but potentially infinite depth as long as they have no cycles;
3. all the feature trees of potentially infinite width, and potentially infinite depth as long as they have no cycles.

In fact, these three models have the exact same first-order theory as $\mathcal{FT}$ as `decide` is a decision function in all of these models. Since the validity of `decide` as a decision function relies on the fact that all its steps are equivalences, let us list all the functions that transform formulas and hint at why they are also performing equivalences in these models.

- `DXC` and `PNF` perform an equivalence in any first-order logic. Similarly, the syntactic negation of a formula $Q \cdot \phi$ as $\overline{Q} \cdot \neg\phi$ is equivalent to the negation of $Q \cdot \phi$ in any first-order logic.
- `switch` performs an equivalence in any feature tree logic as it relies only on G-Exists-Feat that relies only on the functionality of features.
- `transform-1` performs an equivalence in the three aforementioned models as well as in $\mathcal{FT}$. This comes from the fact that all the transformation rules of $\mathcal{R}_1$ perform equivalences in all these models. In other words, Lemma 4.2 also holds in these alternative models.
- Finally, `garbage-collect-1` performs an equivalence in the three aforementioned models as well as in $\mathcal{FT}$. This comes from the fact that Theorem 4.1 also holds in these alternative models. This will not be proven formally but is discussed shortly at the end of Section 4.A.

Table 4.1: Various models and if their first order theory is the same as that of $\mathcal{FT}$

Name	Include $\mathcal{FT}$	Infinite width	Infinite depth	Cycles	Same first-order theory as $\mathcal{FT}$
$\mathcal{FT}$	obviously	no	no	obviously not	obviously
	no	irrelevant	irrelevant	irrelevant	no
	yes	yes	no	obviously not	yes
	yes	no	yes	no	yes
	yes	no	yes	yes	no
	yes	yes	yes	no	yes
$\mathcal{FT}_\infty$	yes	yes	yes	yes	no

Table 4.1 presents a summarised version of everything discussed in this subsection.

4.3.2 Complexity Considerations

We have defined two functions: `transform-1`, which can decide the satisfiability of a DXC, and `decide`, which can decide first-order formulas. One may wonder what complexity these functions have and if they are usable in practice.

The complexity of `decide` of course depends on that of `transform-1`. In any case, we know that it runs in non elementary time [Vorobyov 1996]. This comes from the interaction between the fact that we periodically negate formulas (in definition of `decide`, Line 15) and that we then put them back in DXC (Line 9). In general, putting a formula in DXC can be as long as exponential in the size of the formula. The negation of a DXC is the worst possible case and reaches precisely this exponential complexity. Since we do that for each quantifier alternation, the complexity of `decide` admits as a lower bound a tower of exponential of size the number of quantifier alternations.

The function `transform-1` has a proof of termination (Lemma 4.8) that gives a decreasing measure on the constraints that it considers. One could thus expect such a measure to give us a useful bound on the complexity of `transform-1`. This is however not the case, for three reasons[8].

Firstly, the way the proof of termination works is by showing that `transform-1-xc` applies on constraints that strictly decrease according to a measure. However, `transform-1-xc` may introduce disjunctions, in which case it gets called recursively on all the introduced constraints. This is not a problem for the proof of termination, but it does mean that the bound on the number of executions of `transform-1-xc` would be huge in comparison to that obtained from the measure.

Indeed, for each execution of `transform-1-xc`, we can build a tree of constraints such that each constraints has as children the zero, one or more constraints that are introduced by the application of the rule chosen by `choose-rule-1`. The proof of termination proves that this tree has bounded depth. Since the tree has bounded width (the rules of $\mathcal{R}_1$ introduce at most one disjunction and thus two constraints), we can deduce that it is finite. We get an over-approximation of its size as an exponential of its depth.

We can exhibit formulas on which `transform-1-xc` has to call recursively an exponential number of times. Take n any natural numbers. Take as set of features the set of all natural numbers between 0 and $2^n - 1$. Define, for all $0 \leq i < n$, the set S_i that contain the natural numbers whose i-th digit in the binary representation is 1. With intersections and set differences of such sets, one can select precisely a

[8] I cannot write that without thinking of Paul Taylor's "What The Fuck France" series.

number or another. Consider now the formula:

$$\neg x[\star]\uparrow \wedge \bigwedge_{0\leq i<n} x =_{S_i} y_i \tag{4.7}$$

This formula will trigger S-NAbs-Sim, which will refine the negated absence atom as long as its set is not either included or excluded from all of the S_i. In other words, for as long as it does not contain only one number. This will create a DXC of as many branches as there are numbers in our set of features, that is $2^n - 1$. We thus have a formula of size $n + 1$ that leads to an exponential explosion of the number of constraints in the working DXC.

Secondly, even without the introduction of disjunctions (that is if there is only one chain of constraints), the measure given by the proof of termination is a huge overapproximation of the actual behaviour of the function. The approximation is so unprecise that it could not possibly give us anything useful.

Thirdly and finally, this measure only counts the number of rule applications. It does not take into account the complexity of operations such as deciding whether a constraint is reducible or not, or finding a rule that applies to it. Since application of rules works by matching patterns on constraints, we can expect this to be at least polynomial in the size of the constraints, the exponent being the size of the patterns.

4.3.3 Limitations of transform-1 and $\mathcal{R}_1$

We can list three main limitations of transform-1. Two of them come directly from the fact that transform-1 is based on the system $\mathcal{R}_1$.

1. Firstly, $\mathcal{R}_1$ introduces disjunctions and new variables systematically when in contact with negated atoms. This leads to an explosion of constraints to handle in order to decide satisfiability. Moreover, since the goal is to use transform-1 incrementally and to compose its output with more specifications, this would easily lead to an explosion of cases there.

2. Secondly, $\mathcal{R}_1$ works by matching patterns in sets of literals. It also requires to test subsumption of literals by full constraints. These two operations are not guaranteed to be efficient. In particular, pattern can occur anywhere in a constraint.

3. On DXC, transform-1 has to handle every included x-constraint separately. Even if we change $\mathcal{R}_1$ to not introduce disjunction, some of them come directly from specifications. It would improve greatly the handling of specification if transform-1 was able to work on more expressive formulas. One could then write specifications without disjunctions. This would in particular be useful in the definition of noresolve (Figure 3.20).

We come back on Limitations 1, 2 and 3 in Chapter 5 and tackle them one by one.

Appendix 4.A Proof of Lemma 4.7

This appendix contains the full proof of Lemma 4.7. For convenience, Lemma 4.7 is repeated here:

Lemma 4.7 (Garbage Collection of one Variable in an Irreducible Constraint)**.** *Let c be a constraint that is irreducible with respect to $\mathcal{R}_1$. Let x be a variable of c such that there is no $y[f]x$ in c for any y and f. Then the global part of c with respect to $\{x\}$ implies $\exists x \cdot c$. In other words:*

$$\models_{\mathcal{FT}} \tilde{\forall} \cdot (\mathcal{G}_{\{x\}}(c) \rightarrow (\exists x \cdot c))$$

Introduction. Let c be a constraint that is irreducible with respect to $\mathcal{R}_1$. Take any variable x such that there is no $y[f]x$ in c for any y and f. Take any μ such that $\mu \models_{\mathcal{FT}} \mathcal{G}_x(c)$. We are going to extend μ to ρ such that $\rho \models_{\mathcal{FT}} c$, hence showing that $\mu \models_{\mathcal{FT}} \exists x \cdot c$ and proving Lemma 4.7.

Since c is irreducible with respect to $\mathcal{R}_1$, none of the rules of $\mathcal{R}_1$ can apply to c. This gives us 20 hypotheses of non-applicability on the shape of c. As a reminder, $\mathcal{R}_1$ can be found in Figure 4.7.

The idea is to define $\rho(x)$ by analysing the literals in the constraint.

- Feature atoms of the form $x[f]y$ impose that ρ respects the equation $\rho(x)(f) = \rho(y)$.
- Similarity atoms of the form $x =_F y$ impose that ρ respects, for all $f \in F$, either $f \notin \texttt{dom}(\rho(x))$ and $f \notin \texttt{dom}(\rho(y))$, or $\rho(x)(f) = \rho(y)(f)$.
- Negated absence atoms of the form $\neg x[F]\uparrow$ impose that there is an $f \in F$ such that f is in the domain of $\rho(x)$.
- Negated similarity atoms of the form $x \neq_F y$ impose that there is an $f \in F$ such that $\rho(x)$ is different from $\rho(y)$ in f.

Definition of $\mathcal{D}$. The two first points in this list can be immediately integrated into the definition of $\rho(x)$, as we will see later. The two last points, however, mention the existence of a feature in a certain set – the one carried by negated absence and similarity atoms. This means that there are potentially several valid choices for $\rho(x)$. Let us define a set of features $\mathcal{D}$ that implement this choice. $\mathcal{D}$ will contain one feature for each negated absence or similarity atom. These features will be part of the domain of $\rho(x)$. Let us first define a set $\mathcal{D}_0$:

1. For each $\neg x[F]\uparrow \in c$, choose $f \in F$ and add it to $\mathcal{D}_0$. Note that this is always possible because F cannot be empty, by non-applicability of C-NAbsEmpty.
2. For each $x \neq_F y \in c$, choose $f \in F$ such that there is no z with $x[f]z$ or $y[f]z$ in c, and add it to $\mathcal{D}_0$. Note that this is always possible because F cannot be empty, by non-applicability of C-NSimEmpty, and because if F is finite, then it does not include any feature from a feature atom on x. This comes from non-applicability of S-NSim-Feat which implies that either F does not include any feature from a feature atom on x or it is a singleton. If F is a singleton, then, by non-applicability of D-NSim-Feat, it does not include a feature from a feature atom on x. If F is infinite, since the constraint c contains only a finite number of feature atoms, there always exists such an f.

The definition of $\mathcal{D}_0$ might still contain features that are also covered by similarity atoms in c. We want to avoid that and to make sure that $\mathcal{D}$ has no intersection with features in feature atoms and sets of features in similarity atoms. We thus define $\mathcal{D}$ as:

$$\mathcal{D} = \mathcal{D}_0 \setminus \bigcup_{\substack{x =_F y \in c \\ x \neq y}} F$$

For instance, in the constraint $x[f,g]\uparrow \wedge \neg x[h]\uparrow \wedge \neg x[{}^c\{f,g\}]\uparrow \wedge x =_{\{h\}} y \wedge x \neq_{\{g\}} z$, the set $\{g,h\}$ is a valid choice for $\mathcal{D}_0$ as it intersects $\{h\}$, ${}^c\{f,g\}$ and $\{g\}$. The corresponding set $\mathcal{D}$ is then simply $\{g\}$.

Definition of fresh trees. For each feature f, we define a *fresh tree for f with respect to c and μ* by choosing a finite feature tree that is not in $\{\mu(y)(f) \mid y \in \mathcal{V}(c), y \neq x\}$. Since this set is finite and since the set of finite feature trees is infinite, such free trees always exist. We will denote by *fresh*(f) a fresh tree for f.

Extension of μ to ρ. We define ρ by extending μ to x. That is, we take $\rho(y) = \mu(y)$ for all $y \in \mathtt{dom}(\mu)$ different from x. For x, we define $\rho(x)$ on the domain:

$$\mathtt{dom}(\rho(x)) = \{f \mid x[f]y \in c\} \cup \mathcal{D} \cup \bigcup_{\substack{x =_F y \in c \\ x \neq y}} (\mathtt{dom}(\mu(y)) \cap F)$$

For all $f \in \mathtt{dom}(\rho(x))$, let us define

$$\rho(x)(f) = \begin{cases} \mu(y) & \text{if } x[f]y \in c \qquad (4.8) \\ \mu(y)(f) & \text{if } x =_F y \in c \text{ with } x \neq y,\ f \in F, \text{ and } f \in \mathtt{dom}(\mu(y)) \qquad (4.9) \\ \mathit{fresh}(f) & \text{otherwise} \qquad (4.10) \end{cases}$$

Verification that ρ is well-defined. Firstly, in the first two cases, $\mu(y)$ is indeed defined. This comes from the fact that, in both cases, y is different from x, either by non-applicability of C-Cycle (case 4.8) or by definition (case 4.9).

Secondly, even though the first two cases are not disjoint, this still defines a function. Indeed, if several cases apply, they can be:

- Twice case 4.8. In that case, we have $x[f]y$ and $x[f]z$ in c with $y \neq z$. In that case, by non-applicability of D-Feats, there is $y =_\star z$ in c. Because of non-applicability of C-Cycle, y and z are both distinct from x. Since $\mu \models_{\mathcal{FT}} \mathcal{G}_x(c)$, then it satisfies this full similarity atom and $\mu(y)$ and $\mu(z)$ are equal in all points, and thus equal.
- One of case 4.8 and one of case 4.9. In that case, we have $x[f]y$ and $x =_G z$ in c with $f \in G$, $x \neq y$ and $x \neq z$. In that case, by non-applicability of P-Feat-Sim, there is $z[f]y \in c$, and even in $\mathcal{G}_x(c)$ since $x \neq z$. Since $\mu \models_{\mathcal{FT}} \mathcal{G}_x(c)$, $\mu(y) = \mu(z)(f)$.
- Twice case 4.9. In that case, we have $x =_F y$ and $x =_G z$ in c with $f \in F$, $f \in G$, $x \neq y$ and $x \neq z$. In that case, by non-applicability of P-Sims and the fact that it cannot yield a formula that would be subsumed by c, $f \in F \cap G \subseteq \bigcup_{y =_H z \in c} H$. There is thus a $y =_H z$ in c such that $f \in H$. It is even in $\mathcal{G}_x(c)$ because both $x \neq y$ and $x \neq z$. Since $\mu \models_{\mathcal{FT}} \mathcal{G}_x(c)$, $\mu(y)(f) = \mu(z)(f)$.

Thirdly, ρ is indeed a valuation. That is, all the trees in its image are finite. This is true for all $\rho(y)$ with $y \neq x$ as μ was already a valuation. It is also the case for $\rho(x)$. Indeed, the constraint is finite and thus there is only a finite number of feature atoms, of negated absence and similarity atoms and of variables. This makes $\{f \mid x[f]y \in c\}$ finite, D finite, and the union of all variables distinct from x finite too. Since all the trees in μ are of finite width, all $\mathtt{dom}(\mu(y))$ are finite.

Verification that ρ satisfies c. By definition of ρ, ρ is equal to μ on all variables that are not x. It thus satisfies $\mathcal{G}_x(c)$. We only need to show that it also satisfies $\mathcal{L}_x(c)$, that is all literals that mention x. We reason by exhaustive analysis over all literal forms. This gives us six cases: feature atom (1), negated feature atom (2), absence atom (3), negated absence atom (4), similarity atom (5), and negated similarity atom (6). Since feature and similarity atoms and their negations are binary, there will be sub-cases depending on whether both their variables are x or only one.

1. Firstly, let us consider feature atoms. We have three sub-cases, as such atoms can be $x[f]x$ (1a), $x[f]y$ with $x \neq y$[9] (1b), or $y[f]x$ with $x \neq y$ (1c).

   (a) It is impossible to have a literal $x[f]x$ in c by non-applicability of C-Cycle.

[9] $x \neq y$ means that x and y are syntactically distinct, here, and not that one can find $x \neq_\star y$ in c.

(b) The literals $x[f]y$ with $x \neq y$ are satisfied by definition of ρ, case 4.8.

(c) It is impossible to have a literal $y[f]x$ with $x \neq y$ in c, because they are ruled out by the hypothesis that Lemma 4.7 makes on x.

2. Secondly, let us consider negated feature atoms. It is however impossible to have such atoms in c, by non-applicability of D-NFEAT.

3. Thirdly, let us consider absence atoms. They are of the form $x[F]\uparrow$. If F is empty, then $x[F]\uparrow$ is trivially satisfied by any valuation. Let us assume that F is not empty and consider any $f \in F$. We will show by contradiction that it is impossible that $f \in \mathtt{dom}(\rho(x))$. Assume it is the case. By definition of ρ, $\rho(x)(f)$ is

   - either (case 4.8) equal to $\rho(z)$ for some z if there is $x[f]z$ in c. Such a case cannot happen, however, because that would contradict the non-applicability of C-FEAT-ABS.
   - or (case 4.9) equal to $\rho(z)(f)$ for some $z \neq x$ if there is $x =_G z$ in c with $f \in G$ and $f \in \mathtt{dom}(\mu(z))$. By non-applicability of P-ABS-SIM, there is then $\bigwedge_i z[H_i]\uparrow$ in c with $F \cap G \subseteq \bigcup_i H_i$. Since $f \in F \cap G$, then there exists i_0 such that $f \in H_{i_0}$. The fact that μ satisfies $z[H_{i_0}]\uparrow$ contradicts the fact that $f \in \mathtt{dom}(\mu(z))$.
   - or (case 4.10) fresh. In that case, we know that $f \in \mathcal{D}$. Since $\mathcal{D} \subseteq \mathcal{D}_0$, by definition of $\mathcal{D}_0$, there exists
     - either a negated absence atom $\neg x[G]\uparrow$ such that $f \in G$. In that case, by non-applicability of R-NABS-ABS, $F \cap G = \varnothing$, which enters in contradiction with the fact that $f \in F$ and $f \in G$.
     - or a negated similarity atom $x \neq_G z$ with $z \neq x$ and such that $f \in G$. In that case, by non-applicability of S-NSIM-ABS, either $G \subseteq F$ or $G \subseteq {}^cF$. The former enters in contradiction with the non-applicability of D-NSIM-ABS. The latter enters in contradiction with the fact that $f \in F$ and $f \in G$.

4. Fourthly, let us consider negated absence atoms. They are of the form $\neg x[F]\uparrow$. By definition of $\mathcal{D}_0$, case 1, there is $f \in \mathcal{D}_0 \cap F$. We will show that there is a feature of F in $\mathtt{dom}(\rho(x))$. This will not necessarily be f. Of course, if $f \in \mathtt{dom}(\rho(x))$, then ρ indeed satisfies $\neg x[F]\uparrow$.

   If $f \notin \mathtt{dom}(\rho(x))$, however, that means that $f \notin \mathcal{D}$. In that case, by definition of $\mathcal{D}$, there is a similarity atom $x =_G z$ for some G and z with $z \neq x$ and $f \in G$. By non-applicability of S-NABS-SIM, either $F \subseteq {}^cG$ or $F \subseteq G$. Since $f \in F$ and $f \in G$, the former cannot happen. By non-applicability of P-NABS-SIM, there is then $\neg z[F]\uparrow$. μ satisfies this literal, so there is $g \in F \cap \mathtt{dom}(\mu(z))$. We then have that $g \in \mathtt{dom}(\rho(z))$.

5. Fifthly, let us consider similarity atoms. We have two sub-cases, as such atoms can be $x =_F x$ (5a) or $x =_F y$ with $x \neq y$ (5b).[10]

   (a) The similarity atoms $x =_F x$ are trivially satisfied by any valuation.

   (b) Let us consider similarity atoms of the form $x =_F y$ with $y \neq x$. Let us take any $f \in F$. If $f \in \mathtt{dom}(\mu(y))$, then, by definition of ρ, case 4.9, $f \in \mathtt{dom}(\rho(x))$ and $\rho(x)(f) = \mu(y)(f)$. Let us now assume that $f \notin \mathtt{dom}(\mu(y))$ and show by contradiction that it is not possible to have $f \in \mathtt{dom}(\rho(x))$. Assume it is the case. By definition of ρ, $\rho(x)(f)$ is

[10] There is no third sub-case for $y =_F x$ as similarity atoms are seen as symmetric.

- either (case 4.8) equal to $\rho(z)$ for some z if there is $x[f]z$ in c. By non-applicability of P-Feat-Sim, there is $y[f]z$ in c. Since μ satisfies this literal, $f \in \texttt{dom}(\mu(y))$ which is a contradiction.
- or (case 4.9) equal to $\rho(z)(f)$ for some $z \neq x$ if there is $x =_G z$ in c with $f \in G$ and $f \in \texttt{dom}(\mu(z))$. By non-applicability of P-Sims, there is $\bigwedge_i y =_{H_i} z$ in c with $F \cap G \subseteq \bigcup_i H_i$. Since $f \in F \cap G$, there is i_0 such that $f \in H_{i_0}$. μ satisfies this literal and, therefore, $f \in \texttt{dom}(\mu(y))$ which is a contradiction.
- or (case 4.10) fresh. In that case, we know that $f \in D$. By definition of $\mathcal{D}$, that enters in contradiction with the existence of our similarity atom $x =_F y$ with $f \in F$.

6. Sixthly and lastly, let us consider negated similarity atoms. We have two sub-cases, as such atoms can be $x \neq_F x$ (6a) or $x \neq_F y$ with $x \neq y$ (6b).[11]

   (a) It is impossible to have a negated similarity atom $x \neq_F x$ in c because of non-applicability of C-NSimRefl.

   (b) Let us now consider negated similarity atoms $x \neq_F y$ with $x \neq y$. By definition of $\mathcal{D}_0$, case 2, there is $f \in \mathcal{D}_0 \cap F$. We will show that there is a feature of F on which $\rho(x)$ and $\rho(y)$ are different. This will not necessarily be f.

   If $f \notin \texttt{dom}(\rho(x))$, that means that $f \notin \mathcal{D}$. In that case, by definition of $\mathcal{D}$, there is a similarity atom $x =_G z$ for some G and z with $z \neq x$ and $f \in G$. By non-applicability of S-NSim-Sim, either $F \subseteq {}^cG$ or $F \subseteq G$. Since $f \in F$ and $f \in G$, the former cannot happen. By non-applicability of P-NSim-Sim, there is then $y \neq_F z$ in c. μ satisfies this literal, so there is $g \in F$ such that $\mu(y)$ and $\mu(z)$ are different in g. g belongs also to G, and therefore $f \notin \texttt{dom}(\mu(z))$. That means that $f \in \texttt{dom}(\mu(y))$ and thus that $\rho(x)$ and $\rho(y)$ are different in g.

   If $f \in \texttt{dom}(\rho(x))$ then, by definition of ρ, $\rho(x)(f)$ is

   - either (case 4.8) equal to $\rho(z)$ for some z if there is $x[f]z$ in c. This case is however impossible, by definition of $\mathcal{D}_0$, case 2.
   - or (case 4.9) equal to $\rho(z)(f)$ for some $z \neq x$ if there is $x =_G z$ in c. In that case, by non-applicability of S-NSim-Sim, and since $f \in F \cap G$, then $F \subseteq G$. By non-applicability of P-NSim-Sim, there is then $y \neq_F z$ in c. (Note at this point that, by non-applicability C-NSimRefl, this implies that $y \neq z$, and thus that this case cannot happen). μ satisfies this literal, so there is $g \in F$ such that $\rho(y)(g) \neq \rho(z)(g) = \rho(x)(g)$.
   - or (case 4.10) fresh. In that case, by definition of fresh trees, it is different from $\rho(y)(f)$.

Note on unused rules. G-SimFull is in $\mathcal{R}_1$ but is in fact not used anywhere in this proof. It is in fact non-necessary for the good work of the system, as long as one has P-Sims. It does, however, simplify the proof of termination of `transform-1`. We believe it is a sufficient reason to keep it in $\mathcal{R}_1$.

About other models. Section 4.3.1 discusses alternative models that could be considered in place of $\mathcal{FT}$. It mentions in particular the fact that the width and depth of feature trees can be taken to be unbounded as long as there are no cycles. The first order theory then remains the same. It relies in particular in the fact that garbage collection holds in these alternative models.

The majority of this proof holds independently from these considerations. The only part that does not hold is the proof that $\rho(x)$ is a finite feature tree, because it relies on the hypotheses that the trees in μ

[11] There is no third sub-case for $y \neq_F x$ as negated similarity atoms are seen as symmetric.

```
1   function apply-rule-1 (rule, constraint) : $\Sigma_1$-formula
2
3   function choose-rule-1 ($c$ : constraint) : rule
4     return the first rule of $\mathcal{R}_1$ applicable to $c$ of
5       - all clash rules
6       - G-SimFull or D-NFeat
7       - all other rules of $\mathcal{R}_1$ but P-NAbs-Sim and P-NSim-Sim
8       - P-NAbs-Sim or P-NSim-Sim
9
10  function transform-1-xc ($\exists X \cdot c$ : x-constraint) : DXC
11    if $c$ is reducible in $\mathcal{R}_1$
12      let $r$ = choose-rule-1(c)
13      let $\phi$ = apply-rule-1($r$, $c$)
14      let $d$ = DXC($\exists X \cdot \phi$)
15      return transform-1($d$)
16    else
17      return $\exists X \cdot c$
18
19  function transform-1 ($d$ : DXC) : DXC
20    let $\bigvee_i \exists X_i \cdot c_i$ = $d$
21    return $\bigvee_i$ transform-1-xc($\exists X_i \cdot c_i$)
```

Figure 4.10: Function transform-1

are also finite. The argument can however easily adapt to other models:

- If all the trees in μ have finite width, then $\rho(x)$ has finite width.
- If all the trees in μ have finite depth, then $\rho(x)$ has finite depth.
- If all the trees in μ are cycle-free, then $\rho(x)$ is cycle-free.

These three facts come directly from the construction of ρ, as it only uses trees of μ, or fresh trees that are all finite.

Appendix 4.B Proof of Lemma 4.8

This appendix contains the proof of Lemma 4.8. The definition of transform-1 is given in Figure 4.10. Both Figure 4.10 and Lemma 4.8 are restated here for convenience.

Lemma 4.8 (Termination of transform-1)**.** *The function transform-1 terminates on all inputs.*

The function transform-1, when called on a non-empty DXC, includes one or several calls to the function transform-1-xc (Line 21). The function transform-1-xc, when called on a reducible constraint, includes a call back to transform-1 (Line 15). Depending on the rule r that transform-1-xc chooses via choose-rule-1 (Line 12), the DXC that will be passed to a subsequent call to transform-1 will contain zero (for a clash rule), one (for most other rules) or more (for rules introducing a disjunction) x-constraints, and thus as many calls to transform-1-xc.

We focus on the x-constraints that are taken as argument by transform-1-xc. We show that if transform-1-xc is called on an x-constraint c, then any subsequent recursive call to transform-1-xc will be given an x-constraint that is "smaller" than c. We need to define what it means for an x-constraint to be "smaller" than another one. Moreover, we have to show that it is not possible for x-constraints to become smaller and smaller indefinitely, that is there is no infinite chain of constraints c_0, c_1, etc. such that c_{i+1}

is smaller than c_i for all i.

We denote by $c \Rightarrow c'$ and say that c *transforms into* c', where c and c' are both constraints, when c and c' are the constraints in two subsequent calls to `transform-1-xc`. Note in that case that there must be a rule r given by `choose-rule-1` that applies to c and such that c' is a constraint in the DXC of the result. For instance, we could say that $\neg x[f]y$ transforms into $x[f]z \wedge y \neq_\star z$ via D-NFEAT. We use $\Rightarrow^\star$ to denote the reflexive and transitive closure of $\Rightarrow$.

4.B.1 Simplifying the Problem

Firstly, when writing that $c \Rightarrow c'$, we always know that c is clash-free. Indeed, if it was not, then `transform-1-xc`, via `choose-rule-1`, would have chosen to apply a clash rule to c. The result would then be the empty DXC and there would be no subsequent call to `transform-1-xc` and thus no c'. Moreover, we can always assume that c' is clash-free too, because if it is not, then we know that the call to `transform-1-xc` will be the last one, and we have no termination issue. This means that, in all the forthcoming proof, we only consider clash-free constraints. This comes in handy as, in particular, there are no cycles of features in any constraint, allowing us to define a notion of *depth of variables*.

Secondly, since the rules G-SIMFULL and D-NFEAT are chosen in priority, that means that, in a first phase, `transform-1` applies only these two rules – in addition to clash rules – until they are not applicable anymore. This phase terminates because each step reduces strictly either the number of unsolved variables or the number of negated feature atoms. After that, `transform-1` starts applying other rules of $\mathcal{R}_1$. It is possible that $\mathcal{R}_1$ introduces a full similarity atom (via D-FEATS) or a negated feature atom (via D-NSIM-FEAT). These literals will then be immediately removed by either G-SIMFULL or D-NFEAT. We can thus slightly change the rules and consider the two following ones instead:

D-NSIM-FEAT'	$x \neq_{\{f\}} y \wedge x[f]z \wedge c$	$\Rightarrow$	$(y[f]\uparrow \vee \exists z' \cdot (y[f]z' \wedge z \neq_\star z')) \wedge x[f]z \wedge c$	
D-FEATS'	$x[f]y \wedge x[f]z \wedge c$	$\Rightarrow$	$y =_\star z \wedge x[f]z \wedge c\{y \mapsto z\}$	$(y \neq z)$

The two rules are obtained by gluing together D-NSIM-FEAT and G-SIMFULL for D-NSIM-FEAT' and D-FEATS and D-NFEAT for D-FEATS'. They include both the introduction and the elimination of the full similarity and the negated feature atoms. We end up with an equivalent system of rules, $\mathcal{R}_1^{\text{trunc}}$, that never introduces negated feature atoms or full similarity atoms on unsolved variables. $\mathcal{R}_1^{\text{trunc}}$ is shown in Figure 4.17. For convenience, the rules of $\mathcal{R}_1$ that are not clash rules and not in $\mathcal{R}_1^{\text{trunc}}$ are shown in Figure 4.18. We have to prove that `transform-1`, when applying rules of this system, terminates.

Finally, after `transform-1` has terminated on $\mathcal{R}_1^{\text{trunc}}$, it starts applying the rules P-NABS-SIM or P-NSIM-SIM in a final phase. Luckily, these rules do not change irreducibility with respect to other rules. This is stated in Lemma 4.17. Knowing that, it becomes fairly easy to see that this final phase also terminates.

Lemma 4.17. *If $c \Rightarrow c'$ via* P-NABS-SIM *or* P-NSIM-SIM *and c is irreducible with respect to $\mathcal{R}'_1 = \mathcal{R}_1 \setminus \{$*P-NABS-SIM, P-NSIM-SIM*$\}$, then c' is also irreducible with respect to $\mathcal{R}'_1$.*

Proof. Let us thus take c and c' such that c is irreducible with respect to $\mathcal{R}'_1$. Assume that $c \Rightarrow c'$, where the transformation is performed by P-NSIM-SIM. Let us show that c' is irreducible with respect to $\mathcal{R}'_1$. Let us prove the case that S-NSIM-ABS cannot apply. The other cases are similar and are not detailed.

For S-NSIM-ABS to apply, one needs the interaction of a negated similarity atom with an absence atom. Since c is irreducible with respect to $\mathcal{R}'_1$, S-NSIM-ABS cannot apply on interactions that come directly from c. The interesting case is when the new negated similarity atom, propagated by P-NSIM-SIM, meets an absence atom. In that case, we have $x \neq_F y \wedge x =_G z \wedge z[H]\uparrow$ in c, with $F \subseteq G$, and $y \neq_F z$ in c'.

Deduction Rules

D-Feats’	$x[f]y \wedge x[f]z \wedge c$	$\Rightarrow$	$y =_\star z \wedge x[f]z \wedge c\{y \mapsto z\}$	$(y \neq z)$
D-NSim-Feat’	$x \neq_{\{f\}} y \wedge x[f]z \wedge c$	$\Rightarrow$	$(y[f]\uparrow \vee \exists z' \cdot (y[f]z' \wedge z \neq_\star z')) \wedge x[f]z \wedge c$	
D-NSim-Abs	$x \neq_F y \wedge x[G]\uparrow \wedge c$	$\Rightarrow$	$\neg y[F]\uparrow \wedge x[G]\uparrow \wedge c$	$(F \subseteq G)$

Propagation Rules

P-Feat-Sim	$x[f]y \wedge x =_G z \wedge c$	$\Rightarrow$	$z[f]y \wedge x[f]y \wedge x =_G z \wedge c$	$(f \in G, z[f]y \not\subseteq c)$
P-Abs-Sim	$x[F]\uparrow \wedge x =_G z \wedge c$	$\Rightarrow$	$z[F \cap G]\uparrow \wedge x[F]\uparrow \wedge x =_G z \wedge c$	$(z[F \cap G]\uparrow \not\subseteq c)$
P-Sims	$x =_F y \wedge x =_G z \wedge c$	$\Rightarrow$	$y =_{F \cap G} z \wedge x =_F y \wedge x =_G z \wedge c$	$(y =_{F \cap G} z \not\subseteq c)$

Refinement Rules

R-NAbs-Abs	$\neg x[F]\uparrow \wedge x[G]\uparrow \wedge c$	$\Rightarrow$	$\neg x[F \setminus G]\uparrow \wedge x[G]\uparrow \wedge c$	

Splitting Rules

S-NAbs-Sim	$\neg x[F]\uparrow \wedge x =_G z \wedge c$	$\Rightarrow$	$(\neg x[F \cap G]\uparrow \vee \neg x[F \setminus G]\uparrow) \wedge x =_G z \wedge c$	$(F \not\subseteq G, F \not\subseteq {}^cG)$
S-NSim-Feat	$x \neq_F y \wedge x[f]z \wedge c$	$\Rightarrow$	$(x \neq_{\{f\}} y \vee x \neq_{F \setminus \{f\}} y) \wedge x[f]z \wedge c$	$(F \text{ finite}, f \in F, F \neq \{f\})$
S-NSim-Abs	$x \neq_F y \wedge x[G]\uparrow \wedge c$	$\Rightarrow$	$(x \neq_{F \cap G} y \vee x \neq_{F \setminus G} y) \wedge x[G]\uparrow \wedge c$	$(F \not\subseteq G, F \not\subseteq {}^cG)$
S-NSim-Sim	$x \neq_F y \wedge x =_G z \wedge c$	$\Rightarrow$	$(x \neq_{F \cap G} y \vee x \neq_{F \setminus G} y) \wedge x =_G z \wedge c$	$(F \not\subseteq G, F \not\subseteq {}^cG)$

Figure 4.17: System $\mathcal{R}_1^{\text{trunc}}$ of transformation rules

Deduction Rules

D-Feats	$x[f]y \wedge x[f]z \wedge c$	$\Rightarrow$	$y =_\star z \wedge x[f]y \wedge x[f]z \wedge c$	$(y \neq z, y =_\star z \not\subseteq c)$
D-NFeat	$\neg x[f]y \wedge c$	$\Rightarrow$	$(x[f]\uparrow \vee \exists z \cdot (x[f]z \wedge y \neq_\star z)) \wedge c$	
D-NSim-Feat	$x \neq_{\{f\}} y \wedge x[f]z \wedge c$	$\Rightarrow$	$\neg y[f]z \wedge x[f]z \wedge c$	

Propagation Rules

P-NAbs-Sim	$\neg x[F]\uparrow \wedge x =_G z \wedge c$	$\Rightarrow$	$\neg z[F]\uparrow \wedge \neg x[F]\uparrow \wedge x =_G z \wedge c$	$(F \subseteq G, \neg z[F]\uparrow \not\subseteq c)$
P-NSim-Sim	$x \neq_F y \wedge x =_G z \wedge c$	$\Rightarrow$	$z \neq_F y \wedge x \neq_F y \wedge x =_G z \wedge c$	$(F \subseteq G, z \neq_F y \not\subseteq c)$

Global Rules

G-SimFull	$x =_\star y \wedge c$	$\Rightarrow$	$x =_\star y \wedge c\{x \mapsto y\}$	$(x, y \in \mathcal{V}(c))$

Figure 4.18: Rules of $\mathcal{R}_1$ that are not clash rules and not in $\mathcal{R}_1^{\text{trunc}}$

We need to prove that F cannot be a subset of H or of ${}^{c}H$. The former comes directly from non-applicability of D-NSim-Abs in c. Let us prove the latter.

By non-applicability of P-Abs-Sim in c, we know that $H \cap G \subseteq \bigcup_{x[I]\uparrow \in c} I$. By non-applicability of S-NSim-Abs and D-NSim-Abs in c, we know that F does not intersect any of these $x[I]\uparrow \in c$. Indeed, if there is a nonempty intersection and not an inclusion between these two sets, S-NSim-Abs can apply. If there is an inclusion between these two sets, D-NSim-Abs can apply. This means in particular that $I \subseteq {}^{c}F$ for all $x[I]\uparrow \in c$, which leads to $\bigcup_{x[I]\uparrow \in c} I \subseteq {}^{c}F$ and thus to $H \cap G \subseteq {}^{c}F$.

Because $F \subseteq G$, we can also say that $H \backslash G \subseteq {}^{c}F$, and thus that $H = (H \cap G) \cup (H \backslash G) \subseteq {}^{c}F$ and $F \subseteq {}^{c}H$. From this follows that S-NSim-Abs cannot apply.

This proof works for all the other cases. □

There only remains to show that `transform-1` cannot apply rules of $\mathcal{R}_1^{\text{trunc}}$ forever. In order to do that, we define a measure on constraints such that if $c \Rightarrow c'$ via $\mathcal{R}_1^{\text{trunc}}$, then the measure of c' is smaller than the measure of c. We then have to show that such measures cannot get smaller and smaller forever.

4.B.2 Overview of the Remaining Difficulties

The proof of termination of `transform-1` on $\mathcal{R}_1^{\text{trunc}}$ is not so direct. This is due to the fact that $\mathcal{R}_1^{\text{trunc}}$ has two tendencies that are a priori fighting each other. Let us consider the rules of $\mathcal{R}_1^{\text{trunc}}$ one by one.

- D-Feats' strictly decreases the number of unsolved variables. It also potentially decreases the number of literals in the constraint.
- D-NSim-Abs replaces a negated similarity atom by a negated absence atom, which is a priori simpler to handle as it is a unary predicate.
- Propagation rules (P-Feat-Sim, P-Abs-Sim and P-Sims) add a new literal to the constraint. For instance, P-Feat-Sim adds a new feature atom. There has to be a guarantee that it is not possible to add literals indefinitely. In particular, one has to make sure that it is not possible for a literal to be propagated, then transformed or removed, and then propagated again.

  This is the reason why P-NSim-Sim has a lower priority than D-NSim-Abs in `choose-rule-1`: that way, it cannot re-propagate a negated similarity atom that would then meet an absence atom.
- R-NAbs-Abs and all the splitting rules (S-NAbs-Sim, S-NSim-Feat, S-NSim-Abs and S-NSim-Sim) transform a negated atom (absence or similarity) into a negated atom of the same kind with a (strictly) smaller set. In splitting rules, the set is guaranteed to be strictly smaller thanks to the side conditions.

  Intuitively, a smaller set for such negated atoms is “better”: if $F \subseteq G$, then, for any x and y, $\neg x[F]\uparrow$ implies $\neg x[G]\uparrow$ and $x \neq_F y$ implies $x \neq_G y$. How does one quantify what it means for such negated atoms to be “better”? And since these sets are finite, is it not possible for them to get “better” indefinitely? This is the first major difficulty of this proof. It is discussed in Section 4.B.3.
- Finally, D-NSim-Feat' introduces a new variable and transforms a finite negated similarity atom into an infinite negated similarity atom. There is however no trivial guarantee that the infinite negated similarity atom is better than the finite one. In particular, nothing prevents the former to be transformed (via S-NSim-Abs or S-NSim-Sim) back into a finite negated similarity atom.

  Intuitively, two things happen here. Firstly, the new negated similarity atom is “lower” than the old one: if the old one is $x \neq_{\{f\}} y$, then the new one is on variables that are children of x and y. Secondly, the new negated similarity atom can be transformed back into a finite one, but that requires the presence of either an absence or a similarity atom. If we can show that these cannot “descend” in the constraint, we ensure that the rule D-NSim-Feat’ cannot apply indefinitely. We give a formal meaning to “lower” and “descend” and we discuss this difficulty in Section 4.B.4.

4.B.3 Quantifying Set Quality

Intuitively, a set (in a negated absence or similarity atom) is of “better quality” than another one if it is smaller. This is however tricky to define as such sets can be infinite. A priori, there can be infinite chains of sets that keep getting of better quality.

The idea is however that there is only a finite number of sets of features in a constraint and that all the transformations are computed from these sets using only union, intersection and complement. This implies in particular that such infinite chains of sets cannot exist, and thus that we can define a good notion of “quality” for our proof.

Let us first define the set of feature sets in a constraint in Definition 4.7 and the set of *possible feature sets of a constraint* in Definition 4.8.

Definition 4.7 (Feature Sets of a Constraint)**.** The *feature sets of a constraint* c, noted $\mathcal{FS}(c)$, are all the feature sets that appear in all literals of a constraint. In other words, it is:

$$\begin{aligned}\mathcal{FS}(c) = \quad & \{\{f\} \mid \exists x, y \cdot x[f]y \in c\}\\ \cup\, & \{\{f\} \mid \exists x, y \cdot \neg x[f]y \in c\}\\ \cup\, & \{F \mid \exists x \cdot x[F]\uparrow \in c\}\\ \cup\, & \{F \mid \exists x \cdot \neg x[F]\uparrow \in c\}\\ \cup\, & \{F \mid \exists x, y \cdot x =_F y \in c\}\\ \cup\, & \{F \mid \exists x, y \cdot x \neq_F y \in c\}\end{aligned}$$ □

Definition 4.8 (Possible Feature Sets of a Constraint)**.** The *possible feature sets of a constraint* c, noted $\mathcal{FS}^\star(c)$, is the set $\mathcal{FS}(c)$ augmented with $\star$ and closed by union, intersection, and complement. □

As an example, consider the constraint $c = \neg x[f, g]\uparrow \wedge x[g]\uparrow$ which can be transformed, by R-NAbs-Abs, into $c' = \neg x[f]\uparrow \wedge x[g]\uparrow$. We have $\mathcal{FS}(c) = \{\{g\}, \{f, g\}\}$ and $\mathcal{FS}(c') = \{\{f\}, \{g\}\}$. We also have:

$$\mathcal{FS}^\star(c) = \mathcal{FS}^\star(c') = \{\varnothing, \{f\}, \{g\}, \{f, g\}, {}^c\{f, g\}, {}^c\{g\}, {}^c\{f\}, \star\}$$

The idea behind the word “possible” is that, no matter what happens in a transformation, only the possible sets of a constraint can be reached. In other words, for any transformation $c \Rightarrow^\star c'$, $\mathcal{FS}(c') \subseteq \mathcal{FS}^\star(c)$. This comes in fact from a much stronger property stated in Lemma 4.18. Moreover, there is only a finite number of possible feature sets. This is stated in Lemma 4.19.

Lemma 4.18 (Stability of Possible Feature Sets By Transformation)**.** *For any two constraints c and c', if $c \Rightarrow^\star c'$ via $\mathcal{R}_1^{trunc}$, then $\mathcal{FS}^\star(c') = \mathcal{FS}^\star(c)$.*

Proof. The proof goes by induction on the number of steps of $\Rightarrow^\star$. For one step, let us take R-NAbs-Abs as an example. Assume $c \Rightarrow c'$ via R-NAbs-Abs. The only difference is that, where $\mathcal{FS}(c)$ contained an F, $\mathcal{FS}(c')$ now contains a $F \backslash G$. However, both $\mathcal{FS}(c)$ and $\mathcal{FS}(c')$ contain also G. Therefore, anything in $\mathcal{FS}^\star(c)$ that would be obtained using F is also in $\mathcal{FS}^\star(c')$ using $(F \backslash G) \cup G$, and, conversely, anything in $\mathcal{FS}^\star(c')$ that would be obtained using $F \backslash G$ is also in $\mathcal{FS}^\star(c)$ using $F \cap {}^cG$. The same method applies to every other rule. □

Lemma 4.19 (Finiteness of Possible Feature Sets By Transformation)**.** *For any constraint c, $\mathcal{FS}^\star(c)$ is finite.*

Proof. For any constraint c, $\mathcal{FS}(c)$ is finite. This comes directly from the fact that c, as any constraint, is finite.

Sets and their usual operations (union, intersection and complement) form a Boolean algebra. Therefore, since the sets of $\mathcal{FS}^\star(c)$ are obtained using only these operations, they can all be written as a disjunctive normal form of sets of $\mathcal{FS}(c)$. There is only a finite number of such DNF and, therefore, $\mathcal{FS}^\star(c)$ is finite [Davey & Priestley 2002]. □

We can now define the *quality of a set in a constraint*. This is a positive integer measure that decreases when the set gets smaller. The formal definition can be found in Definition 4.9.

Definition 4.9 (Quality of a Set in a Constraint)**.** In a constraint c, the *quality of a set* $F \in \mathcal{FS}(c)$, noted $q_c(F)$, is defined as:

$$q_c(F) = 1 + \max\{q_c(G) \mid G \subsetneq F, G \in \mathcal{FS}^\star(c)\}$$

where $\max(\varnothing) = 0$. □

This definition is valid because $\subseteq$ is a partial order. This is in fact approximately the height of F in $\mathcal{FS}^\star(c)$, seen as a finite lattice for inclusion, intersection and union. We could have taken any other monotone function for the inclusion, that is such that $q_c(G) < q_c(F)$ if $G \subsetneq F$.

In the rest of the proof, we consider the quality of all negated absence atoms of a constraint:

$$\sum_{\neg x[F]\uparrow \in c} q_c(F)$$

Such a number decreases strictly with R-NAbs-Abs, as this transformation rule either replaces a negated absence atom by one with a strictly smaller set (and thus one of strictly smaller quality) or removes the atom altogether (and thus removes one positive element of the sum).

4.B.4 Controlling Negated Similarity Atoms

Let us start by observing the mechanism of negated similarity atoms being rewritten into themselves. Consider Formula 4.11.

$$x_0 \neq_\star y_0 \wedge x_0[^c\{f_0\}]\uparrow \wedge x_0[f_0]x_1 \tag{4.11}$$

We are going to see how it can transform into Formula 4.12.

$$x_1 \neq_\star y_1 \wedge x_0[^c\{f_0\}]\uparrow \wedge x_0[f_0]x_1 \wedge y_0[f_0]y_1 \tag{4.12}$$

Graphical representations for Formula 4.11 and Formula 4.12 can be found in Figure 4.19.

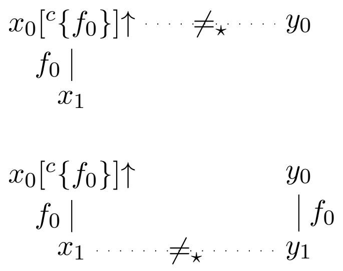


Figure 4.19: Formulas 4.11 and 4.12

Let us now describe the steps that lead from Formula 4.11 to Formula 4.12. In order to decompose a bit more, we are going to separate D-NSim-Feat and D-NFeat. Of course, in $\mathcal{R}_1^{\text{trunc}}$, they are glued together. Intermediary steps can be found in Figure 4.20.

(0) Start from Formula 4.11. Notice the pattern $x_0 \neq_\star y_0 \wedge x_0[^c\{f_0\}]\uparrow$.

$$\begin{array}{llll}
(0) & & & x_0 \neq_\star y_0 \land x_0[^c\{f_0\}]\uparrow \land x_0[f_0]x_1 \\
(1) & \text{S-NSim-Abs} & \Rightarrow & (x_0 \neq_{^c\{f_0\}} y_0 \land x_0[^c\{f_0\}]\uparrow \land x_0[f_0]x_1) \\
 & & & \lor\ (x_0 \neq_{\{f_0\}} y_0 \land x_0[^c\{f_0\}]\uparrow \land x_0[f_0]x_1) \\
(2) & \text{D-NSim-Feat} & \Rightarrow & \neg y_0[f_0]x_1 \land x_0[^c\{f_0\}]\uparrow \land x_0[f_0]x_1 \\
(3) & \text{D-NFeat} & \Rightarrow & (y_0[f_0]\uparrow \land x_0[^c\{f_0\}]\uparrow \land x_0[f_0]x_1) \\
 & & & \lor\ \exists y_1 \cdot (x_1 \neq_\star y_1 \land x_0[^c\{f_0\}]\uparrow \land x_0[f_0]x_1 \land y_0[f_0]y_1)
\end{array}$$

Figure 4.20: Transformation of Formula 4.11 into Formula 4.12 using $\mathcal{R}_1$

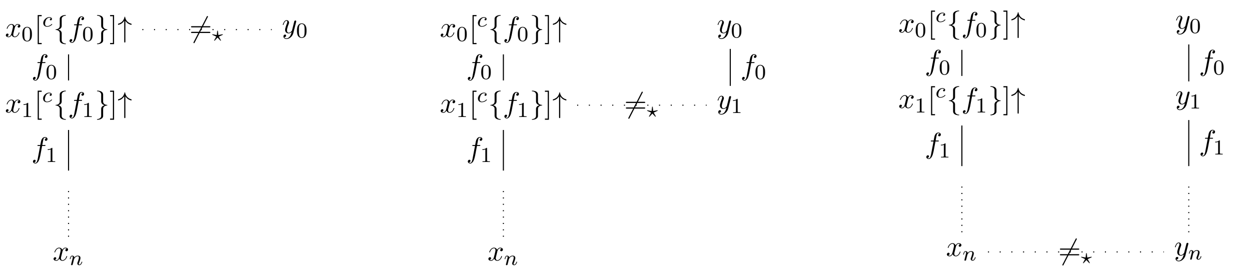


Figure 4.21: Formula 4.13, Formula 4.14, and one intermediary step

(1) Rewrite the constraint using S-NSim-Abs. A disjunction is introduced. The constraint that contains $x_0 \neq_{^c\{f_0\}} y_0$ is still reducible (by D-NSim-Abs), but is not interesting for our example. Let us focus on the other constraint, which contains the pattern $x_0 \neq_{f_0} y_0 \land x_0[f_0]x_1$.

(2) Rewrite the constraint using D-NSim-Feat. A negated feature atom appears.

(3) Rewrite the constraint using D-NFeat. A disjunction is introduced. The constraint that contains $y_0[f_0]\uparrow$ is irreducible and uninteresting for our example. The other constraint contains a new freshly introduced negated similarity atom as well as a new freshly introduced variable!

This example shows that some constraints with negated similarity atoms can be rewritten into other formulas with other negated similarity atoms. Moreover, such transformations can introduce a number of new variables linear in the size of the initial formula. Indeed, we can see how Formula 4.11 can be plugged with itself to repeat the process. For any number n, we can extend it to Formula 4.13, which can be rewritten into Formula 4.14. Graphical representations can be found in Figure 4.21.

$$x_0 \neq_\star y_0 \land \bigwedge_{0 \leq i < n} (x_i[^c\{f_i\}]\uparrow \land x_i[f_i]x_{i+1}) \tag{4.13}$$

$$x_n \neq_\star y_n \land \bigwedge_{0 \leq i < n} (x_i[^c\{f_i\}]\uparrow \land x_i[f_i]x_{i+1} \land y_i[f_i]y_{i+1}) \tag{4.14}$$

Such a transformation starts with a formula of size $2n+1$ with $n+2$ variables and ends with a formula of size $3n+1$ with $2n+2$ variables, effectively introducing a number of variables linear in the size of the initially given constraint. Moreover, it did not get rid of the negated similarity atom.

The good news here is that, for a variable to be introduced, one needs the interaction of a finite negated similarity atom with a feature atom. In that case, a variable is introduced as well as an infinite negated similarity atom. Finally, for finite negated similarity atoms to be introduced, one needs one of the rules P-NSim-Sim, S-NSim-Abs or S-NSim-Sim. All these rules require the presence of an absence or a similarity atom. Absence and similarity atoms thus behave as "fuel" that is necessary for the replication of negated similarity atoms, and we prove in the rest of this subsection that this fuel will eventually run out.

To do that, we first define a notion of *active variables*. By opposition, other variables are *passive*. Active variables are the only ones allowed to carry "fuel", that is absence and similarity atoms. They are basically the variables that were "not introduced". In order to give a formal definition to this, the *active variables* of a constraint c is defined with respect to an *original constraint* c_0 such that $c_0 \Rightarrow^\star c$. The formal definition can be found in Definition 4.10.

Definition 4.10 (Active and Passive Variables)**.** The *active variables* of constraint c in a transformation $c_0 \Rightarrow^\star c$, noted $\mathcal{V}_a(c_0 \Rightarrow^\star c)$, are defined inductively on the transformation from c_0 to c. Originally, $\mathcal{V}_a(c_0 \Rightarrow^0 c_0) = \mathcal{V}(c_0)$. Otherwise, if $c_0 \Rightarrow^\star c \Rightarrow c'$, where the last step is performed by a rule r, then:

- if $r =$ D-Feats', where y has been rewritten into z and $y \in \mathcal{V}_a(c_0 \Rightarrow^\star c)$, then

$$\mathcal{V}_a(c_0 \Rightarrow^\star c \Rightarrow c') = \mathcal{V}_a(c_0 \Rightarrow^\star c) \cup \{z\} \backslash \{y\}$$

- if $r =$ D-NSim-Feat' and c' is the left-hand side of the introduced disjunction, then

$$\mathcal{V}_a(c_0 \Rightarrow^\star c \Rightarrow c') = \mathcal{V}(c')$$

- otherwise, $\mathcal{V}_a(c_0 \Rightarrow^\star c \Rightarrow c') = \mathcal{V}_a(c_0 \Rightarrow^\star c)$.

The variables of $\mathcal{V}_a(c_0 \Rightarrow^\star c)$ are active in the constraint c in the transformation $c_0 \Rightarrow^\star c$. The other variables of c are passive in the constraint c in the transformation $c_0 \Rightarrow^\star c$. □

The active variables are basically defined as the variables of the original constraint. However, because of G-SimFull, we can have variables exchanging their active and passive status. Worse, because of D-NSim-Feat', there needs sometimes to be a "full reset" of the definition of active variables. We abuse notation and write $\mathcal{V}_a(c)$ when the transformation $c_0 \Rightarrow^\star c$ is obvious from context.

As an example, consider the transformations of Figure 4.20. The origin of the transformation is at step (0) and, at this stage, the active variables are $\{x_0, x_1, y_0\}$. Steps (1) and (2) are carried out by rules that do not change the definition of active variables. Step (3) is interesting for two reasons. On the left-hand side of the introduced disjunction, the definition of active variables is reset and includes all variables. In this case, this leaves the set of active variables unchanged. This will not always be the case; in general, the set of active variables may change. On the right-hand side of the introduced disjunction, the set of active variables is unchanged even if a new variable y_1 appears. This variable is thus passive, at this stage.

Let us now formalise right away in Lemma 4.20 our claim that only active variables can have absence and similarity atoms.

Lemma 4.20 (Control of Absence and Similarity Constraints)**.** *In a transformation $c_0 \Rightarrow^\star c$, if $x[F]\uparrow \in c$, then x is active in c and if $x =_F y \in c$ is not solved, then x and y are active in c.*

Proof. We prove this property by induction on the transformation $c_0 \Rightarrow^\star c$. The property holds for the empty transformation as then all variables are active and $c = c_0$. Assume now that $c_0 \Rightarrow^\star c \Rightarrow c'$, and that the property is true on $c_0 \Rightarrow^\star c$. Let us consider any absence and any similarity atom of c' and show that their variables are active. This is trivial if the atom being considered is already in c and if active variables have not been modified between c and c'. We thus have to consider rules that might introduce an absence atom (D-Feats', D-NSim-Feat' and P-Abs-Sim), introduce a similarity atom (D-Feats' and P-Sims) and change the definition of active variables (D-Feats' and D-NSim-Feat'). D-Feats' is listed every time because it rewrites literals and can therefore "introduce" new ones (although, when doing that, it also removes other ones).

- D-Feats' introduces a solved similarity atom and rewrites literals. The solved similarity atom is not a problem. Let us consider any other literal of c'. If it has not been rewritten, then, by Definition 4.10, its active variables remain active. If anything, and if it contains z, then it has received a newly active variable which is not a problem. If it has been rewritten, then, by Definition 4.10, it cannot have lost an active variable. Indeed, if the disappearing variable is active in c, then the appearing variable is active in c'. Since, by induction hypothesis, all absence and unsolved similarity atoms have all their variables active in c, they still have all their variables active in c'.
- D-NSim-Feat' introduces, in the left-hand side of the disjunction, an absence atom. This is however precisely the case in which the definition of active variables is reset. All variables of c' are therefore active and the property holds.
- P-Abs-Sim and P-Sims introduce an absence and a similarity atom respectively. In order to be applied, they require the presence of similarity atoms in c. By induction hypothesis, the two variables of these similarity atoms are active in c. Since these two rules do not change the definition of active variables, all the mentioned variables are therefore active in c'. □

Seen as a down-oriented graph, the shape of a constraint is the following. All the active variables are found at the top. They can form pretty much any graph, as long as it is acyclic. The passive variables are all found at the bottom of the graph, forming only strings. An informal drawing can be found in Figure 4.22. In this drawing, the downward direction corresponds to feature atoms, that is when $x[f]y$ is in the constraint, then y is below x. In order to formalise that, we will define the notion of *parents of a variable* in Definition 4.11. We will then show in Lemma 4.21 that active variables admit only active parents and that passive variables admit only one parent, unless its parents are all active.

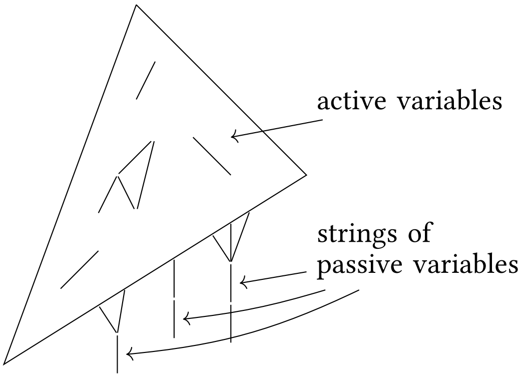


Figure 4.22: Informal drawing of the shape of constraints and their active and passive variables

Definition 4.11 (Parents of a Variable)**.** The *parents of a variable y in a constraint c*, noted $\mathtt{parents}_c(y)$, are defined as follows:

$$\mathtt{parents}_c(y) = \{x \mid x[f]y \in c\}$$ □

Lemma 4.21 (Parents of Variables)**.** *Active variables admit only active parents. Passive variables with more than one parent have only active parents. In other words, for all $y \in \mathcal{V}_a(c)$, $\mathtt{parents}_c(y) \subseteq \mathcal{V}_a(c)$, and for all $y \notin \mathcal{V}_a(c)$, if $\#\mathtt{parents}_c(y) > 1$, then $\mathtt{parents}_c(y) \subseteq \mathcal{V}_a(c)$.*

This property implies Lemma 4.22. This is mainly this (weaker) formulation that we use in the forthcoming proofs, including in the forthcoming proof of Lemma 4.21. Using Lemma 4.22 in the proof of Lemma 4.21 might seem like doing a circular reasoning. This is however fine as we are proving both of them by induction.

Lemma 4.22 (Active Parents of Variables)**.** *Any variable that has one active parent admits only active parents. In other words, for any $y \in \mathcal{V}(c)$, if $\mathtt{parents}_c(y) \cap \mathcal{V}_a(c) \neq \varnothing$, then $\mathtt{parents}_c(y) \subseteq \mathcal{V}_a(c)$.*

Proof of Lemmas 4.21 and 4.22. We prove these properties simultaneously by induction on the transformation $c_0 \Rightarrow^\star c$. The property holds for the empty transformation as then all variables are active and

$c = c_0$. Assume now that $c_0 \Rightarrow^\star c \Rightarrow c'$ and that the property holds for $c_0 \Rightarrow^\star c$. We need to consider all the rules that can either change `parents` or active variables. For the former, we need to consider all the rules that modify, remove or introduce feature atoms. These are D-Feats', D-NSim-Feat' and P-Feat-Sim. For the latter, we also need to consider D-Feats' and D-NSim-Feat'. D-Feats' is a delicate matter and we will keep it for the end.

- D-NSim-Feat', applied on a constraint c that contains $x \neq_f y \wedge x[f]y$, removes the negated similarity atom and replaces it to obtain c'.
  - On the left-hand side of the disjunction, the negated similarity atom is replaced by an absence atom $y[f]\uparrow$ in c'. This does not change `parents`: $\texttt{parents}_{c'}(u) = \texttt{parents}_c(u)$ for all u. The active variables changes, however. In c', all variables are active: $\mathcal{V}_a(c') = \mathcal{V}(c')$. The property therefore holds, because $\texttt{parents}_{c'}(u) \subseteq \mathcal{V}_a(c')$ for all u.
  - On the right-hand side of the disjunction, the negated similarity atom is replaced by $y[f]z' \wedge z \neq_\star z'$ in c', where z' is a newly-introduced variable. The active variables remains the same: $\mathcal{V}_a(c') = \mathcal{V}_a(c)$. `parents` also remains unchanged, except that it is extended to z'. We have:

    $$\begin{aligned} \texttt{parents}_{c'}(z') &= \{y\} \\ \texttt{parents}_{c'}(u) &= \texttt{parents}_c(u) && \text{for all } u \neq z' \end{aligned}$$

    By induction hypothesis, for all $u \neq z'$, $\texttt{parents}_{c'}(u) = \texttt{parents}_c(u) \subseteq \mathcal{V}_a(c) = \mathcal{V}_a(c')$. Moreover, $\#\texttt{parents}_{c'}(z') = 1$.

- P-Feat-Sim, applied to a constraint c that contains $x[f]y \wedge x =_G z$, introduces a new feature atom $z[f]y$ to obtain c'. The active variables do not change, that is $\mathcal{V}_a(c') = \mathcal{V}_a(c)$. `parents` remains unchanged, except for y. We have:

  $$\begin{aligned} \texttt{parents}_{c'}(y) &= \texttt{parents}_c(y) \cup \{z\} \\ \texttt{parents}_{c'}(u) &= \texttt{parents}_c(u) && \text{for all } u \neq y \end{aligned}$$

  By induction hypothesis, for all $u \neq y$, $\texttt{parents}_{c'}(u) = \texttt{parents}_c(u) \subseteq \mathcal{V}_a(c) = \mathcal{V}_a(c')$. Moreover, by Lemma 4.20, $x, z \in \mathcal{V}_a(c)$. Since $x \in \texttt{parents}_c(y)$, then, by induction hypothesis (under the form of Lemma 4.22), $\texttt{parents}_c(y) \subseteq \mathcal{V}_a(c)$. Therefore, $\texttt{parents}_{c'}(y) \subseteq \mathcal{V}_a(c) = \mathcal{V}_a(c')$.

- D-Feats', applied to a constraint of the form $x[f]y \wedge x[f]z \wedge c$, gives $y =_\star z \wedge x[f]z \wedge c\{y \mapsto z\}$. The application of this rule both changes the active variables and `parents`. We have:

  $$\mathcal{V}_a(c') = \begin{cases} \mathcal{V}_a(c) \cup \{z\} \backslash \{y\} & \text{if } y \in \mathcal{V}_a(c) \\ \mathcal{V}_a(c) & \text{otherwise} \end{cases}$$

  and:

  $$\begin{aligned} \texttt{parents}_{c'}(y) &= \varnothing \\ \texttt{parents}_{c'}(z) &= \texttt{parents}_c(y) \cup \texttt{parents}_c(z) \\ \texttt{parents}_{c'}(u) &= \texttt{parents}_c(u) \cup \{z\} \backslash \{y\} && \text{for all } u \neq y, z, \text{ if } y \in \texttt{parents}_c(u) \\ \texttt{parents}_{c'}(u) &= \texttt{parents}_c(u) && \text{otherwise, for all } u \neq y, z \end{aligned}$$

  Technically, in $\texttt{parents}_{c'}(z)$, we should be careful and handle the case where y is in either $\texttt{parents}_c(y)$ or $\texttt{parents}_c(z)$. However, since we consider all constraints to be clash-free, this cannot happen as it would mean that C-Cycle is applicable to either c or c'.

  Let us now prove the property for all variable u. We consider the four sub-cases given by the aforementioned $\texttt{parents}_{c'}$.

- $\texttt{parents}_{c'}(y)$ is empty and thus the property holds for y
- $\texttt{parents}_{c'}(z)$ is $\texttt{parents}_c(y) \cup \texttt{parents}_c(z)$. In c, x is a common parent of y and z. We consider two sub-cases depending on whether at least one of y and z is in $\mathcal{V}_a(c)$.
  - If at least one of y and z is in $\mathcal{V}_a(c)$, then, by induction hypothesis, $x \in \mathcal{V}_a(c)$. By induction hypothesis again (under the form of Lemma 4.22), we have $\texttt{parents}_c(y) \subseteq \mathcal{V}_a(c)$ and $\texttt{parents}_c(z) \subseteq \mathcal{V}_a(c)$. Since neither y nor z belong to these sets, we have $\texttt{parents}_{c'}(z) = \texttt{parents}_c(y) \cup \texttt{parents}_c(z) \subseteq \mathcal{V}_a(c) \backslash \{y, z\} = \mathcal{V}_a(c') \backslash \{y, z\} \subseteq \mathcal{V}_a(c')$. The property holds for z.
  - If neither y nor z is in $\mathcal{V}_a(c)$, then, by induction hypothesis, they have only one parent. Therefore, $\texttt{parents}_{c'}(z) = \texttt{parents}_c(y) \cup \texttt{parents}_c(z) = \{x\} \cup \{x\} = \{x\}$. Moreover, since $y \notin \mathcal{V}_a(c)$, then $\mathcal{V}_a(c') = \mathcal{V}_a(c)$ and $z \notin \mathcal{V}_a(c')$. The property holds for z.
- $\texttt{parents}_{c'}(u)$ is $\texttt{parents}_c(u) \cup \{z\} \backslash \{y\}$ for any $u \neq y, z$ when $y \in \texttt{parents}_c(u)$. We consider two sub-cases depending on whether $y \in \mathcal{V}_a(c)$.
  - If $y \in \mathcal{V}_a(c)$, then, by induction hypothesis (under the form of Lemma 4.22), $\texttt{parents}_c(u) \subseteq \mathcal{V}_a(c)$. We can thus remove y and add z on both side and we get $\texttt{parents}_{c'}(u) = \texttt{parents}_c(u) \cup \{z\} \backslash \{y\} \subseteq \mathcal{V}_a(c) \cup \{z\} \backslash \{y\} = \mathcal{V}_a(c')$. The property holds for u.
  - If $y \notin \mathcal{V}_a(c)$, then, by induction hypothesis, $u \notin \mathcal{V}_a(c)$ and $\texttt{parents}_c(u) = \{y\}$. We now have that $u \notin \mathcal{V}_a(c')$ and that $\texttt{parents}_c(u) = \{z\}$. The property holds for u.
- $\texttt{parents}_{c'}(u)$ is $\texttt{parents}_c(u)$ for any $u \neq y, z$ otherwise. We distinguish two sub-cases depending on whether $\texttt{parents}_c(u) \subseteq \mathcal{V}_a(c)$.
  - If $\texttt{parents}_c(u) \subseteq \mathcal{V}_a(c)$, then, since, $y \notin \texttt{parents}_c(u)$, $\texttt{parents}_c(u) \subseteq \texttt{parents}_c(u) \cup \{z\} \backslash \{y\}$. We therefore get that $\texttt{parents}_c(')u \subseteq \mathcal{V}_a(c)$ and $\texttt{parents}_{c'}(u) \subseteq \mathcal{V}_a(c) \cup \{z\} \backslash \{y\}$. In both cases, $\texttt{parents}_{c'}(u) \subseteq \mathcal{V}_a(c')$. The property holds for u.
  - Otherwise, by induction hypothesis, $u \notin \mathcal{V}_a(c)$ and $\#\texttt{parents}_c(u) = 1$. Since $u \neq y, z$, $u \notin \mathcal{V}_a(c')$ and $\#\texttt{parents}_c(u) = 1$. The property holds for u.

□

Now that we have control over the shape of constraints, we can prove the key argument. We will introduce in Definition 4.12 a notion of *depth of a variable in a constraint*. We will then use it in Definition 4.13 to define the *depth of a negated similarity atom in a constraint*. We will then prove that the depth of negated similarity atoms is bounded by the depth of active variables, which is itself bounded by their number. Since this number is mostly constant (that is, except when using D-NSim-Feat'), this will give us a tool to explain how "lower" negated similarity atoms are "better" and how then cannot get indefinitely better. The bound on the depth of active variables is stated in Lemma 4.23. The bound on the depth of negated similarity atoms is stated in Lemma 4.24 for finite ones and Lemma 4.25 for the general case.

Definition 4.12 (Depth of Variables)**.** The *depth of a variable x in a constraint c*, noted $d_c(x)$ is defined as:

$$d_c(x) = \max\{1 + d_c(y) \mid y[f]x \in c\}$$

□

This definition is valid because we only consider clash-free constraints. Because of the clash rule C-Cycle, that means that we only consider constraints that do not have cycles in the oriented graph formed by feature atoms. Since a lot depends on the definition of the depth, one can see that the fact that constraints are clash-free is not only here for convenience, but it is also a key argument for termination.

Definition 4.13 (Depth of Negated Similarity Atoms)**.** The *depth of a negated similarity atom $x \neq_F y$ in a constraint* c, noted $d_c(x \neq_F y)$, is defined as the minimum of the depth of its variables. In other words:

$$d_c(x \neq_F y) = \min(d_c(x), d_c(y)) \qquad \square$$

An alternative definition could have been obtained by using max instead of min. However, for such an alternative definition, the forthcoming Lemmas 4.24 and 4.25 do not hold.[12]

Lemma 4.23 (Depth of Active Variables)**.** *If $x \in \mathcal{V}_a(c)$, then $d_c(x) < \#\mathcal{V}_a(c)$.*

Proof. This follows from Lemma 4.21 and the absence of cycles. □

Lemma 4.24 (Depth of Negated Finite Similarity Atom)**.** *For all $x \neq_F y \in c$ with F finite, $d_c(x \neq_F y) < \#\mathcal{V}_a(c)$.*

Proof. This goes by showing that one of the two variables in a negated finite similarity atom has to be active. One can then conclude with Lemma 4.23. We prove this once again by induction on the transformation $c_0 \Rightarrow^\star c$. The property holds for the empty transformation as then all variables are active and $c = c_0$. Assume now that $c_0 \Rightarrow^\star c \Rightarrow c'$ and that the property holds for $c_0 \Rightarrow^\star c$.

We need to consider the rules that can either change the definition of active variables or modify, remove or introduce new negated finite similarity atoms. For the former, we need to consider D-Feats' and D-NSim-Feat'. For the latter, we need to consider D-Feats', S-NSim-Feat, S-NSim-Abs and S-NSim-Sim.[13]

- D-Feats', applied to a constraint of the form $x[f]y \wedge x[f]z \wedge c$, gives $y =_\star z \wedge x[f]z \wedge c\{y \mapsto z\}$. The application of this rule both changes the definition of active variables and rewrites some negated similarity atoms. Consider a finite negated similarity atom of c'. We distinguish two sub-cases depending on whether it is in c or not.
  - If it is in c, then, by induction hypothesis, one of its variables is active in c. Whether this variable is z or not[14], then, by Definition 4.10, it is still active in c.
  - If it is not in c, then it must be of the form $z \neq_H u$[15] and there is $y \neq_H u$ in c. By induction hypothesis, one of u or y is active in c. If it is u, then, by Definition 4.10, it is still active in c'. If it is y, then z is active in c'. In both cases, the property holds.
- D-NSim-Feat', applied to a constraint c that contains $x \neq_f y \wedge x[f]y$, removes the negated similarity atom and replaces it to obtain c'.
  - On the left-hand side of the disjunction, the negated similarity atom is replaced by an absence atom $y[f]\uparrow$ in c'. This changes the definition of active variables. In c', all variables are active: $\mathcal{V}_a(c') = \mathcal{V}(c')$. The property therefore holds.
  - On the right-hand side of the disjunction, the negated similarity atom is replaced by $y[f]z' \wedge z \neq_\star z'$ in c', where z' is a newly-introduced variable. The definition of active variables remains the same: $\mathcal{V}_a(c') = \mathcal{V}_a(c)$. The introduced negated similarity atom $z \neq_\star z'$ carries an infinite set[16] and, hence, this atom does not pose any problem. For any other negated similarity atom $u \neq_H u'$ in c, it is also in c'. If H is finite, then, by induction hypothesis, one

[12] It is maybe possible to find an alternative formulation of these properties to reach the same goal, but we do not know.
[13] We are not considering P-NSim-Sim because we only consider rules of $\mathcal{R}_1^{\text{trunc}}$ currently.
[14] It cannot be y as this variable is not present in any literal of c' but one solved similarity.
[15] $u \neq z$ here because constraints are clash-free.
[16] Because $\mathcal{F}$ is infinite.

of u and u' is active in c. Since active variables did not change, it is also active in c' and the property holds.

- S-NSim-Feat, applied to a constraint c that contains $x \neq_F y \wedge x[f]z$ where F is finite, removes the negated similarity atom and replaces it by either $x \neq_f y$ or $x \neq_{F\setminus\{f\}} y$ to obtain c'. The active variables and all the other negated similarity atoms are left unchanged. The new negated similarity atom in c' ($x \neq_f y$ or $x \neq_{F\setminus\{f\}} y$) has the same variables as $x \neq_F y$ in c. Since F is finite, then by induction hypothesis, one of x or y is active in c and thus in c'. The property holds.

- S-NSim-Abs and S-NSim-Sim, similarly, remove a negated similarity atom $x \neq_F y$ from c and replace it by either $x \neq_{F\cap G} y$ or $x \neq_{F\setminus G} y$ to obtain c'. G comes from an absence atom $x[G]\uparrow$ or a similarity atom $x =_G z$. The active variables and all the other negated similarity atoms are left unchanged. The new similarity in c' shares a variable x with an absence or a similarity atom. By Lemma 4.20, x is active in c'. The property holds. □

Lemma 4.25 (Depth of Negated Similarity Atom)**.** *If $x \neq_F y \in c$, then $d_c(x \neq_F y) \leq \#\mathcal{V}_a(c)$.*

Proof. This is already true for finite negated similarity atoms, by Lemma 4.24. We only have to take care of infinite negated similarity atoms. We show by induction that every negated similarity atom has either one of its variables or one of the parents of its variables being active.

We show that by induction on the transformation $c_0 \Rightarrow^\star c$. The property holds for the empty transformation as then all variables are active and $c = c_0$. Assume now that $c_0 \Rightarrow^\star c \Rightarrow c'$ and that the property holds for $c_0 \Rightarrow^\star c$.

We need to consider the rules that can either change the definition of active variables or modify, remove or introduce new negated similarity atoms. For the former, we need to consider D-Feats' and D-NSim-Feat'. For the latter, we need to consider D-Feats', S-NSim-Abs and S-NSim-Sim.[17]

- D-Feats', applied to a constraint of the form $x[f]y \wedge x[f]z \wedge c$, gives $y =_\star z \wedge x[f]z \wedge c\{y \mapsto z\}$. This case is exactly the same as in the proof of Lemma 4.24.

- D-NSim-Feat', applied to a constraint c that contains $x \neq_f y \wedge x[f]y$, removes the negated similarity atom and replaces it to obtain c'.

  - On the left-hand side of the disjunction, the negated similarity atom is replaced by an absence atom $y[f]\uparrow$ in c'. This changes the definition of active variables. In c', all variables are active: $\mathcal{V}_a(c') = \mathcal{V}(c')$. The property therefore holds.

  - On the right-hand side of the disjunction, the negated similarity atom is replaced by $y[f]z' \wedge z \neq_\star z'$ in c', where z' is a newly-introduced variable. The definition of active variables remains the same: $\mathcal{V}_a(c') = \mathcal{V}_a(c)$. By Lemma 4.24, since $\{f\}$ is finite, one of x or y is active in c and c'. Since these two variables are parents of z and z' respectively, the property holds.

- S-NSim-Abs and S-NSim-Sim remove a negated similarity atom $x \neq_F y$ from c and replace it by either $x \neq_{F\cap G} y$ or $x \neq_{F\setminus G} y$ to obtain c'. G comes from an absence atom $x[G]\uparrow$ or a similarity atom $x =_G z$. The active variables and all the other negated similarity atoms are left unchanged. The new similarity in c' shares a variable x with an absence or a similarity atom. By Lemma 4.20, x is active in c'. The property holds.

[17] We do not need to consider S-NSim-Feat as it only introduce finite negated similarity atoms.

If a negated similarity atom has one variable that is active, then its depth is bounded strictly by $\#\mathcal{V}_a(c)$. If it does not, then one of its variables has an active parent. By Lemma 4.22, all the parents of this variable are active. By Definition 4.12, the depth of this variable is thus bounded by $\#\mathcal{V}_a(c)$. □

Since we prefer nonnegative decreasing objects – because they make obvious the fact that there are no infinitely decreasing chains –, we will consider the height rather than the depth. The height of a negated similarity atom is defined in Definition 4.14.

Definition 4.14 (Height of a Negated Similarity Atom)**.** The *height of a negated similarity atom* $x \neq_F y$ *in a constraint* c, noted $h_c(x \neq_F y)$, is defined as:

$$h_c(x \neq_F y) = \#\mathcal{V}_a(c) - d_c(x \neq_F y)$$ □

4.B.5 Decreasing Measure

Let us now define a measure over constraints that strictly decreases with each transformation step. It is a tuple of 8 integer measures on which we consider the natural lexicographic order. The 8 measures are defined as follows:

1. $\#\{x \neq_F y \in c\}$, the number of negated similarity atoms in c.
2. $\sum_{x \neq_F y \in c} h_c(x \neq_F y)$, the total height of negated similarity atoms.
3. $\#\{x \mid x \in \mathcal{V}(c), x \notin \mathcal{V}_s(c)\}$, the number of unsolved variables.
4. $\sum_{x \neq_F y \in c} q_c(F)$, the total quality of feature sets in negated similarity atoms.
5. $\sum_{\neg x[F]\uparrow \in c} q_c(F)$, the total quality of feature sets in negated absence atoms.
6. $\#\{x[f]y \mid x, y \in \mathcal{V}(c), f \in \mathcal{F}(c), x[f]y \notin c\}$, the number of spare feature atoms.
7. $\#\{x[F]\uparrow \mid x \in \mathcal{V}(c), F \in \mathcal{FS}^\star(c), x[F]\uparrow \not\preceq c\}$, the number of spare and non-subsumed absence atoms.
8. $\#\{x =_F y \mid x, y \in \mathcal{V}(c), F \in \mathcal{FS}^\star(c), x \neq y, x =_F y \not\preceq c\}$, the number of spare, non-reflexive and non-subsumed similarity atoms.

All measures are positive integers. For Measures 1, 3, 6, 7 and 8, it is due to the fact that they are defined as cardinals of finite sets. The sets are finite because the constraints are always finite, which implies that they have a finite number of literals, and that the sets $\mathcal{V}(c)$, $\mathcal{F}(c)$ and (by Lemma 4.19) $\mathcal{FS}^\star(c)$ are finite. For Measures 2, 4 and 5, it is due to the fact that they are finite sums of finite integers. The sums are finite because the constraints are always finite. Moreover, the height (Definition 4.14) and the quality (Definition 4.9) of a set are always finite.

Let us now discuss the behaviour of this measure with respect to the transformation rules of $\mathcal{R}_1^{\text{trunc}}$. That is, let us take c and c' two constraints such that $c \Rightarrow c'$ and investigate the relation between the measure of c and the measure of c'. In particular, we are going to show that, no matter which rule performed the transformation, the measure is smaller on c' than it is on c. Table 4.2 contains a summary of how this measure behaves with each step of transformation.

- D-NSim-Abs makes Measure 1 decrease (noted "↓"[18] in Table 4.2), obviously. Since it removes a negated similarity atom to replace it by a negated absence atom, it decreases strictly the number of negated similarity atoms in the formula.

[18] Which has nothing to do with the absence predicate $x[F]\uparrow$.

Table 4.2: Decreasing lexicographic measure over constraints in transformation rules

		1	2	3	4	5	6	7	8
D-NSim-Abs		$\downarrow$							
D-NSim-Feat'	$\vee_l$	$\downarrow$							
	$\vee_r$	$=$	$\downarrow$						
D-Feats'		$\downarrow=$	$\downarrow=$	$\downarrow$					
S-NSim-Feat		$\downarrow=$	$\downarrow=$	$=$	$\downarrow$				
S-NSim-Abs		$\downarrow=$	$\downarrow=$	$=$	$\downarrow$				
S-NSim-Sim		$\downarrow=$	$\downarrow=$	$=$	$\downarrow$				
S-NAbs-Sim		·	·	·	·	$\downarrow$			
R-NAbs-Abs		·	·	·	·	$\downarrow$			
P-Feat-Sim		·	·	·	·	·	$\downarrow$		
P-Abs-Sim		·	·	·	·	·	·	$\downarrow$	
P-Sims		·	·	·	·	·	·	·	$\downarrow$

- D-NSim-Feat' has two sub-cases depending on whether c' is a product of the left-hand side ($\vee_l$ in the table) or right-hand side ($\vee_r$) of the disjunction:
  - On the left-hand side, a negated similarity atom is removed and replaced by an absence atom. Once again, this trivially reduces the number of negated similarity atoms in the constraint (Measure 1).
  - On the right-hand side, the number of negated similarity atoms is left unchanged (noted "$=$" in Table 4.2) as a new one is introduced to replace the old one. Measure 1 thus does not change. Measure 2, however, decreases as the newly introduced negated similarity atom has higher depth and thus lower height (Definition 4.14).

    Note in that case that most of the other measures (Measures 3, 6, 7 and 8) increase a lot because we introduced a new variable. This does not represent an issue, however, as we only consider a lexicographic order over the measures.

- D-Feats' does not add any literal except a full similarity atom. If anything, it may remove some literals if they get rewritten into a literal already present in the constraint. D-Feats' thus leaves unchanged or decreases (noted "$\downarrow=$" in Table 4.2) the number and the total height of negated similarity atoms (Measures 1 and 2). It then decreases strictly the number of unsolved variables (Measure 3).

- S-NSim-Feat, S-NSim-Abs and S-NSim-Sim replace a negated similarity atom by another one between the same two variables. This change cannot increase the number of such atoms, their total height or the number of unsolved variables (Measures 1, 2 and 3). If anything, it might decrease the first two measures if it introduces an already existing negated similarity atom. These three rules decrease the quality of sets in negated similarity atoms (Measure 4) as it replaces the sets of negated similarity atoms by a strictly smaller one and thus of strictly smaller quality.

- S-NAbs-Sim and R-NAbs-Abs replace a negated absence atom by another one on the same variable. They obviously cannot impact the number, the height or the quality of negated similarity atoms (Measures 1, 2 and 4). They also cannot increase the number of unsolved variables (Measure 3). They do however decrease the quality of sets in absence atoms (Measure 5) for similar reasons as the previous point.

- P-Feat-Sim, P-Abs-Sim and P-Sims introduce a new feature, absence or similarity atom respectively. They obviously cannot impact the number, the height or the quality of sets of negated similarity atoms (Measures 1, 2 and 4). They also cannot impact the number of unsolved variables or the quality of sets in negated absence atoms (Measures 3 and 5). They do however decrease the number of spare feature, absence or similarity atoms respectively (Measures 6, 7 and 8).

Chapter 5

Efficient Solving of Feature Tree Constraints

This chapter develops an efficient algorithm to decide satisfiability of Σ_1-formulas. Contrary to the one of Chapter 4, the new algorithm cannot be extended to decide the complete first-order theory. In particular, it does not have the property of garbage collection. It is however more efficient as a test of satisfiability. We consider the three big limitations of the system described in the previous chapter (as discussed in Section 4.3.3) and we address them one by one.

In a first part, Section 5.1, we tackle Limitation 1 by introducing the new system $\mathcal{R}_2$ that does not introduce disjunctions or new variables. We discuss its strengths and limitations compared to $\mathcal{R}_1$.

In a second part, Section 5.2, we deal with Limitation 2 by introducing a variant of $\mathcal{R}_2$ that makes explicit an efficient way to recognise patterns. This is the formal link between $\mathcal{R}_2$ and our implementation.

In a third and last part, Section 5.3, we discuss Limitation 3. We then extend $\mathcal{R}_2$ to make it able to handle more expressive formulas. We then use this extra expressivity to rewrite specifications in a more efficient way.

5.1 A System Without Disjunctions

The system $\mathcal{R}_1$ introduces disjunctions and new variables when handling negated atoms. This can make reasoning on constraints via $\mathcal{R}_1$ fairly inefficient as it increases – exponentially – the number of constraints to be processed by the system. In this section, we tackle Limitation 1.

In Section 5.1.1, we introduce a new system of transformation rules, $\mathcal{R}_2$, that does not need to introduce any disjunction or variable to handle constraints. In Section 5.1.2, we discuss properties of this system, and in particular how it can be used to decide the satisfiability of constraints. Finally, in Section 5.1.3, we introduce a function `transform-2` which decides satisfiability of constraints. We then discuss it in Section 5.1.4.

5.1.1 Transformation Rules for Constraints – The System $\mathcal{R}_2$

Let us introduce an alternative system of transformation rules, named $\mathcal{R}_2$. The goal of $\mathcal{R}_2$ is to avoid introducing disjunctions and new variables altogether, that is, the application of a rule of $\mathcal{R}_2$ on a constraint yields another constraint, and not a Σ_1-formula.

Clash Rules

C-Cycle	$x[f]y \wedge \bigwedge_{i=0}^{n-1} z_i[f_i]z_{i+1} \wedge c$	$\Rightarrow$	$\bot$	$(n \geq 1, y = z_0, x = z_n)$
C-Feat-Abs	$x[f]y \wedge x[F]\uparrow \wedge c$	$\Rightarrow$	$\bot$	$(f \in F)$
C-NAbsEmpty	$\neg x[\varnothing]\uparrow \wedge c$	$\Rightarrow$	$\bot$	
C-NSimRefl	$x \neq_F x \wedge c$	$\Rightarrow$	$\bot$	
C-NSimEmpty	$x \neq_\varnothing y \wedge c$	$\Rightarrow$	$\bot$	

Figure 5.1: Clash rules in system $\mathcal{R}_2$

Deduction Rules

D-Feats	$x[f]y \wedge x[f]z \wedge c$	$\Rightarrow$	$y =_\star z \wedge x[f]y \wedge x[f]z \wedge c$	$(y \neq z, y =_\star z \not\preceq c)$

Propagation Rules

P-Feat-Sim	$x[f]y \wedge x =_G z \wedge c$	$\Rightarrow$	$z[f]y \wedge x[f]y \wedge x =_G z \wedge c$	$(f \in G, z[f]y \not\preceq c)$
P-Abs-Sim	$x[F]\uparrow \wedge x =_G z \wedge c$	$\Rightarrow$	$z[F \cap G]\uparrow \wedge x[F]\uparrow \wedge x =_G z \wedge c$	$(z[F \cap G]\uparrow \not\preceq c)$
P-Sims	$x =_F y \wedge x =_G z \wedge c$	$\Rightarrow$	$y =_{F \cap G} z \wedge x =_F y \wedge x =_G z \wedge c$	$(y =_{F \cap G} z \not\preceq c)$

Global Rules

G-SimFull	$x =_\star y \wedge c$	$\Rightarrow$	$x =_\star y \wedge c\{x \mapsto y\}$	$(x, y \in \mathcal{V}(c))$

Figure 5.2: Transformation rules for positive literals in system $\mathcal{R}_2$

The introduction of disjunctions and new variables occurs only in the handling of negated literals. This is what $\mathcal{R}_2$ achieves in a different way than $\mathcal{R}_1$. In particular, that means that the clash rules and the rules for positive literals should remain the same. Let us recall them in Figures 5.1 and 5.2 respectively. These two figures are exactly the same as Figures 4.2 and 4.3.

The notions of subsumption of a literal by a constraint, application of a rule, irreducibility, etc. remain the same. Moreover, $\mathcal{R}_2$ presents the same kind of rules as $\mathcal{R}_1$:

- clash rules that detect unsatisfiability in formulas,
- deduction rules that create new literals out of others of a different kind,
- propagation rules that transfer information from one side of a similarity to another,
- refinement rules that modify literals to make them more precise,
- and global rules that modify the whole formula.

The key idea behind $\mathcal{R}_2$ is that it is complicated to handle negated atoms in a clean way – in $\mathcal{R}_1$, they are the source of a lot of complications, including the introduction of disjunctions and variables. Contrary to what is done in $\mathcal{R}_1$, in $\mathcal{R}_2$, instead of writing rules for negated atoms, we write more rules about their positive counterpart. These extra rules give us more guarantees on irreducible constraints with respect to $\mathcal{R}_2$. We then use these guarantees to recover a check of unsatisfiability.

The rules that allow $\mathcal{R}_2$ to handle negated literals are presented in Figure 5.3. For convenience, the full system is presented in Figure 5.4. Let us now describe how this system works.

Firstly, one can stop propagating negated atoms. Indeed, even if it is important that atoms and their

Deduction Rules

D-NFeat-Feat	$\neg x[f]y \wedge x[f]z \wedge c$	$\Rightarrow$	$y \neq_{\star} z \wedge x[f]z \wedge c$	
D-FeatsEq-Sep	$x[f]z \wedge y[f]z \wedge \mathsf{S}(x,y) \wedge c$	$\Rightarrow$	$x =_{\{f\}} y \wedge x[f]z \wedge y[f]z \wedge c$	$(x =_{\{f\}} y \not\preceq c)$
D-Feats-Sep	$x[f]z \wedge y[f]z' \wedge \mathsf{S}(x,y) \wedge c$	$\Rightarrow$	$\mathsf{S}(z,z') \wedge x[f]z \wedge y[f]z' \wedge \mathsf{S}(x,y) \wedge c$	$(z \neq z', \mathsf{S}(z,z') \not\preceq c)$
D-Abs-Sep	$x[F]\!\uparrow \wedge\, y[G]\!\uparrow \wedge\, \mathsf{S}(x,y) \wedge c$	$\Rightarrow$	$x =_{F\cap G} y \wedge x[F]\!\uparrow \wedge\, y[G]\!\uparrow \wedge\, c$	$(x =_{F\cap G} y \not\preceq c)$
D-NSim	$x \neq_F y \wedge c$	$\Rightarrow$	$\mathsf{S}(x,y) \wedge x \neq_F y \wedge c$	$(\mathsf{S}(x,y) \not\preceq c)$

Refinement Rules

R-NAbs-Abs	$\neg x[F]\!\uparrow \wedge\, x[G]\!\uparrow \wedge\, c$	$\Rightarrow$	$\neg x[F \setminus G]\!\uparrow \wedge\, x[G]\!\uparrow \wedge\, c$
R-Sims	$x =_F y \wedge x =_G y \wedge c$	$\Rightarrow$	$x =_{F\cup G} y \wedge c$
R-NSim-Sim	$x \neq_F y \wedge x =_G y \wedge c$	$\Rightarrow$	$x \neq_{F\setminus G} y \wedge x =_G y \wedge c$

Figure 5.3: Transformation rules for positive and negative literals in system $\mathcal{R}_2$

negations meet so as to detect unsatisfiability, it can be done by only propagating atoms and not their negations.

Secondly, in the case of the negated similarity atom, we have to make sure that the two variables that it relates are not forced to be equal by other means than a similarity atom. For instance, the formula $x \neq_{\{f\}} y \wedge x[f]z \wedge y[f]z$ is not satisfiable because the negated similarity atom requires a difference in f while the rest specifies that both x and y point to z through f and are thus equal there. The solution to this problem is to explicitly introduce similarity atoms when we detect such equalities. Through rules like R-NSim-Sim, they then push away negated similarity atoms and detect unsatisfiability. In the aforementioned example, one can deduce the similarity atom $x =_{\{f\}} y$. With R-NSim-Sim, the negated similarity atom $x \neq_{\{f\}} y$ is replaced by $x \neq_{\varnothing} y$ which triggers a clash.

This requires a priori to consider all the pairs of variables of the formula, in case one can detect a similarity, which might appear expensive. Moreover, this introduces rules that need to consider the whole formula. It is in fact not necessary to consider all pairs of variables but only those that might have an impact on negated similarity atoms. This means the pairs of variables that appear together in a negated similarity atom of course, but also all the pairs of their children. In order to do that, we define *separated pairs* of variables in a constraint in Definition 5.1. The rules that deduce similarity atoms from any two variables – D-FeatsEq-Sep and D-Abs-Sep – will only be triggered if they mention separated pairs of variables

Definition 5.1 (Separated Pairs of Variables)**.** A pair of variables (x, y) is *separated in the constraint* c if:

- there is a negated similarity atom $x \neq_F y$ in c for some (possibly empty) set F;
- or there are feature atoms $x'[f]x$ and $y'[f]y$ in c for some f where x' and y' are separated in c.

The set of separated pairs of variables of c is noted $\mathcal{S}(c)$. □

This definition is not local in the sense that, in order to decide whether a pair of variables is separated, one might have to follow a potentially long chain of features. In order to circumvent this problem, we explicitly add separation information to constraints. We thus consider *extended constraints* as defined in Definition 5.2.

Clash Rules

C-Cycle	$x[f]y \wedge \bigwedge_{i=0}^{n-1} z_i[f_i]z_{i+1} \wedge c$	$\Rightarrow$	$\perp$	$(n \geq 1, y = z_0, x = z_n)$
C-Feat-Abs	$x[f]y \wedge x[F]\uparrow \wedge c$	$\Rightarrow$	$\perp$	$(f \in F)$
C-NAbsEmpty	$\neg x[\varnothing]\uparrow \wedge c$	$\Rightarrow$	$\perp$	
C-NSimRefl	$x \neq_F x \wedge c$	$\Rightarrow$	$\perp$	
C-NSimEmpty	$x \neq_\varnothing y \wedge c$	$\Rightarrow$	$\perp$	

Deduction Rules

D-Feats	$x[f]y \wedge x[f]z \wedge c$	$\Rightarrow$	$y =_\star z \wedge x[f]y \wedge x[f]z \wedge c$	$(y \neq z, y =_\star z \not\preceq c)$
D-NFeat-Feat	$\neg x[f]y \wedge x[f]z \wedge c$	$\Rightarrow$	$y \neq_\star z \wedge x[f]z \wedge c$	
D-FeatsEq-Sep	$x[f]z \wedge y[f]z \wedge \mathsf{S}(x,y) \wedge c$	$\Rightarrow$	$x =_{\{f\}} y \wedge x[f]z \wedge y[f]z \wedge c$	$(x =_{\{f\}} y \not\preceq c)$
D-Feats-Sep	$x[f]z \wedge y[f]z' \wedge \mathsf{S}(x,y) \wedge c$	$\Rightarrow$	$\mathsf{S}(z,z') \wedge x[f]z \wedge y[f]z' \wedge \mathsf{S}(x,y) \wedge c$	$(z \neq z', \mathsf{S}(z,z') \not\preceq c)$
D-Abs-Sep	$x[F]\uparrow \wedge y[G]\uparrow \wedge \mathsf{S}(x,y) \wedge c$	$\Rightarrow$	$x =_{F \cap G} y \wedge x[F]\uparrow \wedge y[G]\uparrow \wedge c$	$(x =_{F \cap G} y \not\preceq c)$
D-NSim	$x \neq_F y \wedge c$	$\Rightarrow$	$\mathsf{S}(x,y) \wedge x \neq_F y \wedge c$	$(\mathsf{S}(x,y) \not\preceq c)$

Propagation Rules

P-Feat-Sim	$x[f]y \wedge x =_G z \wedge c$	$\Rightarrow$	$z[f]y \wedge x[f]y \wedge x =_G z \wedge c$	$(f \in G, z[f]y \not\preceq c)$
P-Abs-Sim	$x[F]\uparrow \wedge x =_G z \wedge c$	$\Rightarrow$	$z[F \cap G]\uparrow \wedge x[F]\uparrow \wedge x =_G z \wedge c$	$(z[F \cap G]\uparrow \not\preceq c)$
P-Sims	$x =_F y \wedge x =_G z \wedge c$	$\Rightarrow$	$y =_{F \cap G} z \wedge x =_F y \wedge x =_G z \wedge c$	$(y =_{F \cap G} z \not\preceq c)$

Refinement Rules

R-NAbs-Abs	$\neg x[F]\uparrow \wedge x[G]\uparrow \wedge c$	$\Rightarrow$	$\neg x[F \backslash G]\uparrow \wedge x[G]\uparrow \wedge c$	
R-Sims	$x =_F y \wedge x =_G y \wedge c$	$\Rightarrow$	$x =_{F \cup G} y \wedge c$	
R-NSim-Sim	$x \neq_F y \wedge x =_G y \wedge c$	$\Rightarrow$	$x \neq_{F \backslash G} y \wedge x =_G y \wedge c$	

Global Rules

G-SimFull	$x =_\star y \wedge c$	$\Rightarrow$	$x =_\star y \wedge c\{x \mapsto y\}$	$(x, y \in \mathcal{V}(c))$

Figure 5.4: System $\mathcal{R}_2$ of Transformation Rules

(0)			$r[f]x \wedge x[g]y \wedge x[^c\{g\}]\uparrow$ $\wedge\, r'[f]x' \wedge x'[g]y \wedge x'[^c\{g\}]\uparrow \wedge r \neq_{\{f,h\}} r'$
(1)	D-NSim + D-Feats-Sep	$\Rightarrow$	$r[f]x \wedge x[g]y \wedge x[^c\{g\}]\uparrow \wedge \mathrm{S}(r,r') \wedge \mathrm{S}(x,x')$ $\wedge\, r'[f]x' \wedge x'[g]y \wedge x'[^c\{g\}]\uparrow \wedge r \neq_{\{f,h\}} r'$
(2)	D-FeatsEq-Sep + D-Abs-Sep	$\Rightarrow$	$r[f]x \wedge x[g]y \wedge x[^c\{g\}]\uparrow \wedge \mathrm{S}(r,r') \wedge \mathrm{S}(x,x')$ $\wedge\, r'[f]x' \wedge x'[g]y \wedge x'[^c\{g\}]\uparrow \wedge r \neq_{\{f,h\}} r' \wedge x =_{\{g\}} x' \wedge x =_{^c\{g\}} x'$
(3)	R-Sims + G-SimFull	$\Rightarrow$	$r[f]x \wedge x[g]y \wedge x[^c\{g\}]\uparrow \wedge \mathrm{S}(r,r') \wedge \mathrm{S}(x,x)$ $\wedge\, r'[f]x \wedge r \neq_{\{f,h\}} r' \wedge x =_{\{\star\}} x'$
(4)	D-Feats-Sep	$\Rightarrow$	$r[f]x \wedge x[g]y \wedge x[^c\{g\}]\uparrow \wedge \mathrm{S}(r,r') \wedge \mathrm{S}(x,x)$ $\wedge\, r'[f]x \wedge r \neq_{\{f,h\}} r' \wedge r =_{\{f\}} r' \wedge x =_{\{\star\}} x'$
(5)	R-NSim-Sim	$\Rightarrow$	$r[f]x \wedge x[g]y \wedge x[^c\{g\}]\uparrow \wedge \mathrm{S}(r,r') \wedge \mathrm{S}(x,x)$ $\wedge\, r'[f]x \wedge r \neq_{\{h\}} r' \wedge r =_{\{f\}} r' \wedge x =_{\{\star\}} x'$

Figure 5.6: Transformation of Formula 5.1.

Definition 5.2 (Extended Literal and Constraint)**.** An *extended literal*[1] is either a literal or an information of separation of the form $\mathrm{S}(x,y)$.

An *extended constraint*[2] is either $\top$ or of the form $l_1^e \wedge \cdots \wedge l_n^e$ $(n \geq 1)$ where, for all i, l_i^e is an extended literal. Extended constraints can be seen as a possibly empty sets of extended literals, the empty set being $\top$. □

Moreover, we need rules to compute separated pairs of variable. This is the sense of the rules D-Feats-Sep and D-NSim.

Let us give an example of this mechanism applied to a constraint containing a negated similarity, where a positive similarity is deduced in the process. Consider Formula 5.1. A graphical representation is given in Figure 5.5.

$$r[f]x \wedge x[g]y \wedge x[^c\{g\}]\uparrow \wedge r'[f]x' \wedge x'[g]y \wedge x'[^c\{g\}]\uparrow \wedge r \neq_{\{f,h\}} r' \quad (5.1)$$

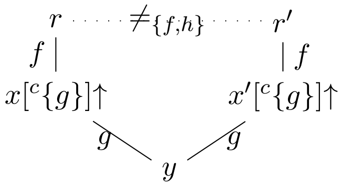


Figure 5.5: Formula 5.1

This constraint contains a negated similarity atom $r \neq_{\{f,h\}} r'$. If r and r' were forced to be equal in both f and h, then the whole formula would be unsatisfiable. In this example, there are no other atoms mentioning the feature h and we can thus expect the full constraint to be satisfiable – or at least the negated similarity atom to not pose problem. Let us now describe the steps of transformation from Formula 5.1 in $\mathcal{R}_2$. Intermediary steps can be found in Figure 5.6.

(0) Start from Formula 5.1. Notice the negated similarity atom $r \neq_{\{f,h\}} r'$.

(1) Rewrite the constraint using D-NSim. This introduces the separation literal $\mathrm{S}(r,r')$ and the pattern $r[f]x \wedge r'[f]x' \wedge \mathrm{S}(r,r')$. Rewrite the constraint again using D-Feats-Sep. This introduces the separation literal $\mathrm{S}(x,x')$. The system $\mathcal{R}_2$ has determined the set of separated pairs of variables. This introduces the patterns $x[g]y \wedge x'[g]y \wedge \mathrm{S}(x,x')$ and $x[^c\{g\}]\uparrow \wedge x'[^c\{g\}]\uparrow \wedge \mathrm{S}(x,x')$.

[1] Literals are defined in Definition 3.13.

[2] Constraints are defined in Definition 3.14.

(2) Rewrite the constraint using D-FeatsEq-Sep and D-Abs-Sep. This introduces the similarity atoms $x =_{\{g\}} x'$ and $x =_{^c\{g\}} x'$. These two similarity atoms together form a pattern.

(3) Rewrite the constraint using R-NSim-Sim. This merges the two similarity atoms into a full similarity atom $x =_\star x'$. The system $\mathcal{R}_2$ has deduced that the separated pair of variables x and x' were similar in every feature. This can then trigger the rule G-SimFull. which rewrites the whole formula, replacing x' by x. Some literals then disappear. More importantly, the pattern $r[f]x \wedge r'[f]x \wedge \mathsf{S}(r, r')$ appears

(4) Rewrite the constraint using D-FeatsEq-Sep. This deduces the similarity atom $r =_{\{f\}} r'$ and introduces the pattern $r \neq_{\{f,h\}} r' \wedge r =_{\{f\}} r'$.

(5) Rewrite the constraint using R-NSim-Sim. This transforms the negated similarity atom into $r \neq_{\{h\}} r'$. We reach an irreducible form and the whole formula was thus satisfiable. In particular, the negated similarity atom is satisfiable as one can realise a difference between r and r' in h. Had the initial negated similarity atom been $r \neq_{\{f\}} r'$, this step would have discovered $r \neq_\varnothing r'$ which would have led to a clash.

5.1.2 Properties of Irreducible Constraints of $\mathcal{R}_2$

Irreducible constraints of $\mathcal{R}_2$ do not have the property of garbage collection. This is directly related to the fact that negated atoms do not travel through similarity atoms in $\mathcal{R}_2$. There is thus information that is not duplicated among all variables, which means that some variables cannot be removed without losing such information. For instance, the constraint $c = x[f]\uparrow \wedge x \neq_{\{f\}} y$ is irreducible but it is not true that $\exists x \cdot c$ is equivalent to $\top$ as it implies that y has to have a feature f, that is $\exists x \cdot c$ implies $\neg y[f]\uparrow$.

The irreducible constraints of $\mathcal{R}_2$ are however still always satisfiable. This is stated in Theorem 5.1. In fact, $\mathcal{R}_2$ admits weaker versions of garbage collection. For instance, we can recover the property of garbage collection by forbidding the use of negated atoms. This is stated in Lemma 5.1. We can be more subtle by authorising negated atoms in the global part of a constraint. This is stated in Lemma 5.2. Finally, we conjecture that we can be even more subtle and authorise negated atoms also in the local part of a constraint as long as they are on variables that do not share a similarity atom with a global variable. This is stated in Conjecture 5.1.

Theorem 5.1 (Satisfiability of Irreducible Constraints). *A constraint that is irreducible with respect to $\mathcal{R}_2$ is satisfiable.*

Proof of Theorem 5.1 (idea). This proof is quite similar to that of Theorem 4.1, except that the system $\mathcal{R}_2$ does not give as strong guarantees on its irreducible constraints as $\mathcal{R}_1$. This is in particular because it does not have access to the splitting rules, as they introduce disjunctions. The absence of these rules is at the heart of the fact that $\mathcal{R}_2$ does not enjoy the property of garbage collection. Consider for instance Formula 5.2.

$$\exists x \cdot (\neg x[\star]\uparrow \wedge x =_G y \wedge x =_{^cG} z) \tag{5.2}$$

It expresses the fact that the local variable x must have a feature and must be similar to y in G and to z in ^cG. The information that this carries on the global variables y and z is that “either y has a feature in G or z has a feature in ^cG”. This fact is easily expressible but requires disjunction – $\neg y[G]\uparrow \vee \neg z[^cG]\uparrow$. The system $\mathcal{R}_1$, since it is allowed to introduce disjunctions, will be able to deduce that fact and to eliminate the variable x. The system $\mathcal{R}_2$, since it cannot introduce disjunctions, will never be able to remove x.

Let us dig a bit this example and show how both $\mathcal{R}_1$ and $\mathcal{R}_2$ handle such a constraint. Name the constraint in Formula 5.2. We will put it in irreducible form with respect to both $\mathcal{R}_1$ and $\mathcal{R}_2$ and assume we have a valuation μ satisfying the global part of the resulting constraint/s. We will then imagine that we are trying to define ρ satisfying the whole constraint.

- The constraint of Formula 5.2 is not irreducible with respect to $\mathcal{R}_1$. This is because of the splitting rule S-NAbs-Sim which requires us to split $\neg x[\star]\uparrow$ into $\neg x[G]\uparrow \vee \neg x[^cG]\uparrow$. In each of the obtained constraint, P-NAbs-Sim requires us to propagate the negated absence atoms to y and z respectively. We obtain two constraints that are irreducible with respect to $\mathcal{R}_1$:

  $$\exists x \cdot (\neg x[G]\uparrow \wedge x =_G y \wedge x =_{^cG} z) \wedge \neg y[G]\uparrow$$

  and

  $$\exists x \cdot (\neg x[^cG]\uparrow \wedge x =_G y \wedge x =_{^cG} z) \wedge \neg z[^cG]\uparrow$$

  If a valuation μ satisfies the global parts of the former (resp. the latter), then it satisfies $\neg y[G]\uparrow$ (resp. $\neg z[^cG]\uparrow$), which we can leverage to show that if ρ satisfies $x =_G y$ (resp. $x =_{^cG} z$), then it satisfies $\neg x[\star]\uparrow$ automatically. This reasoning works on *any* valuation μ and therefore we can conclude that x has become irrelevant and that Formula 5.2 is equivalent to $\neg y[G]\uparrow \vee \neg z[^cG]\uparrow$.

- The constraint of Formula 5.2 is irreducible with respect to $\mathcal{R}_2$, however, because $\mathcal{R}_2$ does not contain such splitting rules. It is not true that any valuation μ that satisfies the global part of this constraint can be extended to ρ that satisfies the whole constraint. Indeed, if we take t the empty feature tree and $\mu = [y \mapsto t, z \mapsto t]$, the two similarity atoms impose that $\rho(x) =_G t$ and $\rho(x) =_{^cG} t$ and therefore that $\rho(x) = t$, which contradicts $\rho \models_{\mathcal{FT}} \neg x[\star]\uparrow$.

  If we have control on the valuation μ that we take for the global part, then we can ensure that there is $g \in G$ (or resp. $g \in {}^cG$) such that $g \in \texttt{dom}(\mu(y))$ (resp. $g \in \texttt{dom}(\mu(z))$). Any valuation μ that satisfies this will then be extendable to x. Since this reasoning works on *some* valuations μ, we cannot conclude that x has become irrelevant, but we can conclude that the formula is satisfiable.

In the general case, this means that the proof of Theorem 5.1 cannot be as simple as that of Theorem 4.1. In particular, we will not be able to build an induction around a simple lemma as Lemma 4.7. We will still prove Theorem 5.1 by induction on the variables of the constraint, but the induction hypothesis will have to be much stronger to compensate for the information that the splitting rules would otherwise provide.

For the definition of the induction hypothesis and for the full proof, see Section 5.A. □

Lemma 5.1 (Garbage Collection of Irreducible Positive Constraints). *Let c be a constraint that only contains atoms and that is irreducible with respect to $\mathcal{R}_2$. Let X be a set of variables of c that is ancestor-closed. Then $\exists X \cdot c$ is equivalent to the global part of c with respect to X. In other words:*

$$\models_{\mathcal{FT}} \tilde{\forall} \cdot ((\exists X \cdot c) \leftrightarrow \mathcal{G}_X(c))$$

Proof. A constraint that is irreducible with respect to $\mathcal{R}_2$ and contains only atoms is in fact also irreducible with respect to $\mathcal{R}_1$. Such constraints thus indeed have the property of garbage collection, as stated in Theorem 4.1. □

Lemma 5.2 (Garbage Collection of Irreducible Constraints With Positive Local Part). *Let c be a constraint that is irreducible with respect to $\mathcal{R}_2$. Let X be a set of variables of c that is ancestor-closed and such that $\mathcal{L}_X(c)$ only contains atoms. Then $\exists X \cdot c$ is equivalent to the global part of c with respect to X.*

Proof. If $\mathcal{L}_X(c)$ only contains atoms and is irreducible with respect to $\mathcal{R}_2$, then it is also irreducible with respect to $\mathcal{R}_1$. By Theorem 4.1, $\exists X \cdot \mathcal{L}_X(c)$ is then equivalent to $\top$. Therefore, $\mathcal{G}_X(c) \wedge \exists X \cdot \mathcal{L}_X(c)$ – which is exactly $\exists X \cdot c$ – is equivalent to $\mathcal{G}_X(c)$. □

```
function apply-rule-2 (rule, constraint) : constraint or ⊥

function choose-rule-2 ($c$ : constraint) : rule
  return any rule of $\mathcal{R}_2$ applicable to $c$

function transform-2 ($c$ : constraint) : constraint or ⊥
  if $c$ is reducible in $\mathcal{R}_2$
    let $r$ = choose-rule-2($c$)
    match apply-rule-2($r$, $c$)
    | ⊥ -> return ⊥
    | $c'$ -> return transform-2($c'$)
  else
    return $c$
```

Figure 5.7: Function transform-2

Conjecture 5.1 (Garbage Collection of Irreducible Constraints With Positive Border)**.** *Let c be a constraint that is irreducible with respect to $\mathcal{R}_2 \cup \{$P-NFeat-Sim, P-NAbs-Sim, P-NSim-Sim$\}$. Let X be a set of variables of c that is ancestor-closed and such that there are no negated atoms in $\mathcal{L}_X(c)$ that shares a variable with a similarity atom in $\mathcal{L}_X(c) \cap \mathcal{L}_{{}^cX}(c)$. Then $\exists X \cdot c$ is equivalent to the global part of c with respect to X.*

In fact, the proof of Theorem 4.1 does not inspect the global part of the constraint, except the variables that share a similarity with the outside. For the rest, we only assume the existence of a valuation that satisfies them. On the contrary, the proof of Theorem 5.1 does inspect the whole constraint because it needs to ensure that its invariants are satisfied from the beginning. We believe however that it is possible to start not from the beginning – similarly as for Theorem 4.1 – as long as the separation between local and global parts is “clean enough” to ensure the validity of the proof invariants.

5.1.3 Deciding the Satisfiability of Constraints

Let us now define in Figure 5.7 a function transform-2 which decides the satisfiability of constraints. This simply goes by applying rules of $\mathcal{R}_2$ in any order until we get either $\bot$ or an irreducible constraint. Since all the rules of $\mathcal{R}_2$ are equivalences, the input constraint is equivalent to either $\bot$ of a satisfiable constraint.

In fact, transform-2 is the same as transform-1 – if it was using choose-rule-2 – except that the former can be defined in a simpler way because it does not involve rules that might introduce a new variable or a disjunction. Formally, if one takes c a constraint and feeds it to both transform-1 and transform-2[3] that, in both cases, use the same function choose-rule-2, we get the same output constraint c'.

Of course, transform-2 terminates on all its inputs, as stated in Lemma 5.3.

Lemma 5.3 (Termination of transform-2)**.** *The function transform-2 terminates on all inputs.*

Proof. This proof is similar to that of Lemma 4.8 except much simpler. We will define a measure on clash-free constraints that decreases at each step of transformation. This measure associates a tuple of 9 integer measures to a constraint c. The measures are defined as follows:

1. $\#\{x \mid x \in \mathcal{V}(c), x \notin \mathcal{V}_s(c)\}$, the unsolved variables.

[3] A constraint can be seen as a DXC with only one x-constraint with an empty quantifier block. Therefore, we can feed a constraint to the function transform-1.

Table 5.1: Decreasing lexicographic measure over constraints in transformation rules

	1	2	3	4	5	6	7	8	9
G-SimFull	$\downarrow$								
P-Feat-Sim	$=$	$\downarrow$							
P-Abs-Sim	$=$	·	$\downarrow$						
P-Sims	$=$	·	·	$\downarrow$					
D-Feats	$=$	$=$	·	$\downarrow$					
D-FeatsEq-Sep	$=$	$=$	·	$\downarrow$					
D-Abs-Sep	$=$	$=$	$=$	$\downarrow$					
D-Feats-Sep	$=$	$=$	·	·	$\downarrow$				
D-NSim	$=$	·	·	·	$\downarrow$				
D-NFeat-Feat	$=$	$=$	·	·		$\downarrow$			
R-NAbs-Abs	$=$	·	$=$	·		·	$\downarrow$		
R-NSim-Sim	$=$	·	·	$=$		·	·	$\downarrow$	
R-Sims	$=$	·	·	$=$		·	·	·	$\downarrow$

2. $\#\{x[f]y \mid x, y \in \mathcal{V}(c), f \in \mathcal{F}(c), x[f]y \notin c\}$, the number of spare feature atoms.
3. $\#\{x[F]\uparrow \mid x \in \mathcal{V}(c), F \in \mathcal{FS}^\star(c), x[F]\uparrow \not\models c\}$, the number of spare and non-subsumed absence atoms.
4. $\#\{x =_F y \mid x, y \in \mathcal{V}(c), F \in \mathcal{FS}^\star(c), x \neq y, x =_F y \not\models c\}$, the number of spare, non-reflexive and non-subsumed similarity atoms.
5. $\#\{\mathsf{S}(x,y) \mid x, y \in \mathcal{V}(c), \mathsf{S}(x,y) \notin c\}$, the number of spare separation literals.
6. $\#\{\neg x[f]y \in c\}$, the number of negated feature atoms in c.
7. $\sum_{\neg x[F]\uparrow \in c} q_c(F)$, the total quality of feature sets in negated absence atoms.
8. $\sum_{x \neq_F y \in c} q_c(F)$, the total quality of feature sets in negated similarity atoms.
9. $\#c$, the number of literals in c.

All measures are positive integers. This comes from the finiteness of the constraint, which implies that there is only a finite number of literals in it and that all the sets $\mathcal{V}(c)$, $\mathcal{F}(c)$ and $\mathcal{FS}^\star(c)$ are all finite.

Table 5.1 contains a summary of how this measure behaves with each step of transformation.

- G-SimFull solves a variable and thus decreases the number of unsolved variables (Measure 1).
- P-Feat-Sim, P-Abs-Sim and P-Sims introduce a new feature, absence or similarity atom respectively if it is not subsumed by the rest of the constraint. This decreases the number of spare feature, absence or similarity atoms (Measures 2, 3 and 4). This cannot increase the number of unsolved variables (Measure 1).
- D-Feats, D-FeatsEq-Sep and D-Abs-Sep introduce a new similarity atom if it is not subsumed by the rest of the constraint. Similarly to P-Sims, this decreases the number of spare similarity atoms (Measure 4). This cannot increase the number of unsolved variables (Measure 1) or the number of spare feature or absence atoms (Measures 2 and 3).
- D-Feats-Sep and D-NSimintroduce a new separation literal if it is not present in the rest of the constraint.

This decreases the number of spare separated literals (Measure 5). This cannot increase the number of unsolved variables (Measure 1) or the number of spare feature, absence or similarity atoms (Measures 2, 3 and 4).

- D-NFeat-Feat removes a negated feature atom from the constraint and thus decreases their number (Measure 6). This does not impact the number of unsolved variables (Measure 1) or the number of spare feature, absence or similarity atoms or separation literals (Measures 2, 3, 4 and 5).
- R-NAbs-Abs and R-NSim-Sim replace a negated absence or similarity atom by another one on the same variable/s. This improves the total quality of feature sets in negated absence or similarity atoms (Measures 7 and 8). This does not impact the number of unsolved variables (Measure 1), the number of spare feature, absence or similarity atoms or separation literals (Measures 2, 3, 4 and 5), or the number of negated feature atoms (Measure 6).
- Finally, R-Sims replaces two similarity atoms by one, thus reducing the total number of literals of the constraint (Measure 9). This does not impact the number of unsolved variables (Measure 1), the number of spare feature, absence or similarity atoms or separation literals (Measures 2, 3, 4 and 5), the number of negated feature atoms (Measure 6), or the quality of feature sets in negated absence or similarity atoms (Measures 7 and 8). In particular, this does not decrease the number of spare similarity atoms because this leaves the union of all feature sets in similarity atoms unchanged. □

5.1.4 Discussions

The function `transform-2` can be used as an unsatisfiability check. Similarly to `transform-1`, it is in fact *complete* in the sense that it detects unsatisfiability if and only if the constraint is indeed unsatisfiable. Moreover, it is incremental by nature: since the output constraint is equivalent to the input one, we can throw away the input constraint and keep only the output one. If one later adds other literals to the constraint, all the computation that has been done previously is still valid and only the computation that has to do with the new literals will take place.

The main difference in `transform-2` with `transform-1` is that the new version takes constraints as inputs and returns constraints. This is much more efficient as it avoids the explosion of disjunctions that can happen with `transform-1`, as discussed in the complexity considerations in Section 4.3.2.

As a counterpart, `transform-2` does not have the property of garbage collection that `transform-1` enjoys. `transform-2` does enjoy weaker forms of garbage collection, as discussed in Section 5.1.2. Depending of the context in which `transform-2` is meant to be used, the lack of garbage collection can be an issue. Note however that the situations that defeat even the results partial garbage collection of Lemma 5.2 and Conjecture 5.1 are situations that involve negated literals. These are precisely the situations leading to an explosion of cases in `transform-1`. The bottom line is that these situations are problematic and show limitations of FTS itself.

In the context of the CoLiS project, the occurrences of negated literals are rare as they do not occur naturally in specifications of utilities. An exception would be the utility `rmdir` which succeeds when a directory is empty (expressible with an absence atom) and fails when a directory is not empty (expressible with a negated absence atom). It is not such a problem in that its use is marginal in the corpus of Debian packages. Moreover, if negated literals are problematic, the negated absence atom remains the simplest one to handle as it is a unary predicate. It is in particular likely to behave well with respect to partial garbage collections like that of Conjecture 5.1.

Let us now discuss the notion of separated variables. The limitation of the rules D-FeatsEq-Sep and D-Abs-Sep to only separated variables does not bring any improvement in term of theoretical complexity. In practice,

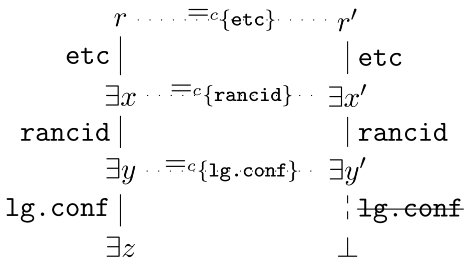


Figure 3.10: Specification of success case for `rm -R /etc/rancid/lg.conf`

Tidying Up Rules

T-NFeat-Abs	$\neg x[f]y \wedge x[F]\uparrow \wedge c$	$\Rightarrow$	$x[F]\uparrow \wedge c$	$(f \in F)$
T-AbsEmpty	$x[\varnothing]\uparrow \wedge c$	$\Rightarrow$	c	
T-NAbs-Feat	$\neg x[F]\uparrow \wedge x[f]y \wedge c$	$\Rightarrow$	$x[f]y \wedge c$	$(f \in F)$
T-NAbs-NAbs	$\neg x[F]\uparrow \wedge \neg x[G]\uparrow \wedge c$	$\Rightarrow$	$\neg x[F]\uparrow \wedge c$	$(F \subseteq G)$
T-SimRefl	$x =_F x \wedge c$	$\Rightarrow$	c	
T-SimEmpty	$x =_\varnothing y \wedge c$	$\Rightarrow$	c	
T-NSim-NSim	$x \neq_F y \wedge x \neq_G y \wedge c$	$\Rightarrow$	$x \neq_F y \wedge c$	$(F \subseteq G)$
T-SepRefl	$\mathsf{S}(x, x) \wedge c$	$\Rightarrow$	c	

Refinement Rules

R-Abs	$x[F]\uparrow \wedge x[G]\uparrow \wedge c$	$\Rightarrow$	$x[F \cup G]\uparrow \wedge c$

Figure 5.8: Extra Transformation Rules for System $\mathcal{R}_2$

however, and in the CoLiS project in particular, it does bring a huge difference.

Let us observe, as an example, the specification case for the success of `rm -R /etc/rancid/lg.conf`. It is shown in Figure 3.10, restated here for convenience. Such a case is typically built from a path coming from a Shell script. If the path is of size n, the specification cases will contain two strings of feature atoms of size n and about as many variables. These $2n$ variables will appear in about n similarity atoms. At this point, there are two remarks that can be done.

- Firstly, in formulas of such shapes, there is usually a clear "depth" of variables corresponding to the distance from the root in each filesystem. If separation literals appear, that will then be on variables of same depth. The separation literals will then propagate at every level, *linearly* in n. This makes a big difference when compared to the total number of pairs of variables, *quadratic* in n. In our example, there are three pairs of separated variables – (r, r'), (x, x') and (y, y') – while there are 21 pairs of variables in total.

- Secondly, in such a specification, all the similarity atoms are already maximal in the sense that they could not be made bigger by D-FeatsEq-Sep or D-Abs-Sep. This comes from the fact that we try to write our specifications as functions. This property of maximality of similarity atoms in specifications could be added to the list of properties that we aim at having on specifications, as described in Section 3.4.

Finally, note that there are many natural transformation rules that one might want to add to $\mathcal{R}_2$ in an actual implementation, especially rules that would allow tidying up the formula by removing useless atoms. Figure 5.8 present a few of such rules. We believe they make the presentation and the proofs heavier. They are therefore not included in this document.

5.2 A System With Efficient Pattern Recognition

The system $\mathcal{R}_2$ is better than $\mathcal{R}_1$ in that it does not share its Limitation 1 and introduce no disjunctions at all, especially when it handles negated literals. The two systems, however, have Limitation 2 in common: they consider constraints as sets of literals in which they find patterns. This process is fairly inefficient as patterns could occur anywhere in the formula.

Moreover, it makes the use of our solvers not so incremental. Indeed, assume you have a constraint c that is irreducible with respect to $\mathcal{R}_2$ and you want to consider $c \wedge c'$ where c' is any other constraint, not necessarily irreducible with respect to $\mathcal{R}_2$. This operation of adding literals to an irreducible constraint happens all the time in a symbolic engine as the one we use in the CoLiS project. The functions `transform-1` and `transform-2` presented up to here would only allow us to consider the whole constraint $c \wedge c'$, which completely looses the information that c is already irreducible with respect to $\mathcal{R}_1$ or $\mathcal{R}_2$.

These two problems have the same solution. The idea is to only introduce literals of given constraints one by one. Every time we introduce a new literal we apply $\mathcal{R}_2$ until an irreducible form is obtained. Only then can we consider introducing the next literal. This way, the place where the interactions take place is clear: they lie between the newly introduced literal and the rest of the constraint.

This section formalises this. In Section 5.2.1, we introduce *pointed constraints* and a system of transformations $\mathcal{R}_2^\bullet$ on such constraints. These definitions formalise respectively these constraints with one freshly introduced literal, and the rules that can apply in such situations. In Section 5.2.2, we study the relationship between $\mathcal{R}_2^\bullet$ and $\mathcal{R}_2$ and we show that $\mathcal{R}_2^\bullet$ is a sound strategy of application of rules of $\mathcal{R}_2$. This will in particular allow us to lift the results of $\mathcal{R}_2$ to $\mathcal{R}_2^\bullet$, most notably the proof of satisfiability of irreducible constraints. Finally, in Section 5.2.3, we introduce a function `transform-2-pointed` that decides satisfiability of constraints. We then discuss the approach of this section in Section 5.2.4.

5.2.1 Pointed Constraints and Transformation Rules – The System $\mathcal{R}_2^\bullet$

The system $\mathcal{R}_2$, presented in Section 5.1, works by matching patterns in a set of literals. Even by sorting the literals in a clever way, we would still need to explore most of the set to find matching patterns, as we have no way to know where the next pattern is going to occur.

To circumvent this problem, we introduce *pointed constraints*. A pointed constraint $\langle \pi \mid c \rangle$ is the pair of a *todo-stack* π and a *store* c. These pointed constraints make the pattern matching more efficient by forcing the patterns to only occur between the literal at the peek of the stack – called the *pointed literal* – and the store. We can then adapt $\mathcal{R}_2$ to deal with such pointed constraints. Let us define pointed constraint in Definition 5.3.

Definition 5.3 (Pointed Constraint)**.** A *pointed constraint* is a pair $\langle \pi \mid c \rangle$ of a *todo-stack* π of extended literals and an extended constraint c[4], called *store*. □

The forthcoming rule G-SimFull$^\bullet$ will enjoy a specific treatment. In order to make it efficient and sound, we need to split a constraint into a part only made of equalities (that is full similarity atoms) and the

[4] Extended literals and constraints are defined in Definition 5.2.

Table 5.2: Number of rules in $\mathcal{R}_2$ and $\mathcal{R}_2^\bullet$ and corresponding figures

	Clash		Positive		Negated		Total	
$\mathcal{R}_2$	5	Fig. 5.1	5	Fig. 5.2	8	Fig. 5.3	18	Fig. 5.4
$\mathcal{R}_2^\bullet$	6	Fig. 5.9	7	Fig. 5.10	14	Fig. 5.11	27	n/a

Pointed Clash Rules

$$\begin{array}{lrcll}
\text{C-CYCLE}^\bullet & \pi \langle x[f]y \mid \bigwedge_{i=0}^{n-1} z_i[f_i]z_{i+1} \wedge c & \Rightarrow & \bot & (n \geq 0, y = z_0, x = z_n) \\
\text{C-FEAT}^\bullet\text{-ABS} & \pi \langle x[f]y \mid x[F]\uparrow \wedge c & \Rightarrow & \bot & (f \in F) \\
\text{C-FEAT-ABS}^\bullet & \pi \langle x[F]\uparrow \mid x[f]y \wedge c & \Rightarrow & \bot & (f \in F) \\
\text{C-NABSEMPTY}^\bullet & \pi \langle \neg x[\varnothing]\uparrow \mid c & \Rightarrow & \bot & \\
\text{C-NSIMREFL}^\bullet & \pi \langle x \neq_F x \mid c & \Rightarrow & \bot & \\
\text{C-NSIMEMPTY}^\bullet & \pi \langle x \neq_\varnothing y \mid c & \Rightarrow & \bot &
\end{array}$$

Figure 5.9: System $\mathcal{R}_2^\bullet$ of Transformation Rules – Clash Rules

remaining part. Moreover, we need to consider the local and global parts of the latter. All these are defined in Definition 5.4.

Definition 5.4 (Equalities of a Constraint). The *equalities of a constraint* c, noted $\mathcal{E}(c)$, are all the full similarity atoms in the constraint c. We introduce notations for the remaining part of c and for the local and global parts of the latter:

$$\begin{aligned}
\mathcal{E}(c) &= \{x =_\star y \in c\} \\
\bar{\mathcal{E}}(c) &= c \backslash \mathcal{E}(c) \\
\mathcal{L}_X^{\bar{\mathcal{E}}}(c) &= \mathcal{L}_X(\bar{\mathcal{E}}(c)) \\
\mathcal{G}_X^{\bar{\mathcal{E}}}(c) &= \mathcal{G}_X(\bar{\mathcal{E}}(c))
\end{aligned}$$

□

We can now derive transformation rules for pointed constraints from the rules of $\mathcal{R}_2$. Since there is now a pointed literal, which stands out, we break the symmetry of the rules and thus, some rules need to be duplicated. For instance, the clash rule C-FEAT-ABS:

$$\begin{array}{lrcll}
\text{C-FEAT-ABS} & x[f]y \wedge x[F]\uparrow \wedge c & \Rightarrow & \bot & (f \in F)
\end{array}$$

gives birth to the two rules C-FEAT$^\bullet$-ABS and C-FEAT-ABS$^\bullet$:

$$\begin{array}{lrcll}
\text{C-FEAT}^\bullet\text{-ABS} & \pi \langle x[f]y \mid x[F]\uparrow \wedge c & \Rightarrow & \bot & (f \in F) \\
\text{C-FEAT-ABS}^\bullet & \pi \langle x[F]\uparrow \mid x[f]y \wedge c & \Rightarrow & \bot & (f \in F)
\end{array}$$

depending on whether the feature or the absence atom is pointed. By applying that process on all the 18 rules of $\mathcal{R}_2$, we obtain 27 rules in $\mathcal{R}_2^\bullet$. The clash rules, rules for positive literals and rules for negated literals are presented in Figures 5.9, 5.10 and 5.11 respectively. By lack of space, we cannot provide a single figure containing all the rules. Table 5.2 presents the increase in number of rules between $\mathcal{R}_2$ and $\mathcal{R}_2^\bullet$.

There are already a few things to note at this point. A natural question, for instance, is whether the todo-stack is actually a stack, or if it is a queue, and if there is a logic to the order in which the literals appear in the todo-stack. In fact, because of the fact that rules apply if there is no subsumption of the

Pointed Deduction Rules

D-FEATS•	$\pi \lessdot x[f]y \mid x[f]z \wedge c$	$\Rightarrow$	$\pi \lessdot x[f]y \lessdot y =_\star z \mid x[f]z \wedge c$	$(y \neq z, y =_\star z \not\preccurlyeq \langle \pi \mid c \rangle)$

Pointed Propagation Rules

P-FEAT•-SIM	$\pi \lessdot x[f]y \mid x =_G z \wedge c$	$\Rightarrow$	$\pi \lessdot z[f]y \lessdot x[f]y \mid x =_G z \wedge c$	$(f \in G, z[f]y \not\preccurlyeq \langle \pi \mid c \rangle)$
P-FEAT-SIM•	$\pi \lessdot x =_G z \mid x[f]y \wedge c$	$\Rightarrow$	$\pi \lessdot z[f]y \lessdot x =_G z \mid x[f]y \wedge c$	$(f \in G, z[f]y \not\preccurlyeq \langle \pi \mid c \rangle)$
P-ABS•-SIM	$\pi \lessdot x[F]\uparrow \mid x =_G z \wedge c$	$\Rightarrow$	$\pi \lessdot z[F \cap G]\uparrow \lessdot x[F]\uparrow \mid x =_G z \wedge c$	$(z[F \cap G]\uparrow \not\preccurlyeq \langle \pi \mid c \rangle)$
P-ABS-SIM•	$\pi \lessdot x =_G z \mid x[F]\uparrow \wedge c$	$\Rightarrow$	$\pi \lessdot z[F \cap G]\uparrow \lessdot x =_G z \mid x[F]\uparrow \wedge c$	$(z[F \cap G]\uparrow \not\preccurlyeq \langle \pi \mid c \rangle)$
P-SIMS•	$\pi \lessdot x =_F y \mid x =_G z \wedge c$	$\Rightarrow$	$\pi \lessdot y =_{F \cap G} z \lessdot x =_F y \mid x =_G z \wedge c$	$(y =_{F \cap G} z \not\preccurlyeq \langle \pi \mid c \rangle)$

Pointed Global Rules

G-SIMFULL•	$\pi \lessdot x =_\star y \mid c$	$\Rightarrow$	$(\pi \ll \mathcal{L}_x^{\bar{\mathcal{E}}}(c))\{x \mapsto y\} \lessdot x =_\star y \mid \mathcal{G}_x^{\bar{\mathcal{E}}}(c) \wedge \mathcal{E}(c)\{x \mapsto y\}$	$(x, y \in \mathcal{V}(c))$

Figure 5.10: System $\mathcal{R}_2^\bullet$ of Transformation Rules – Rules for positive literals

introduced literal, the order does not matter for correctness. It might matter for efficiency of the algorithm or cleanliness of the presentation.

Note also that the patterns of pointed rules always consider the todo-stack to be non-empty, that is they require the presence of a pointed literal. This means that a pointed constraint with an empty todo-stack is trivially irreducible with respect to $\mathcal{R}_2^\bullet$. Finally, some patterns of $\mathcal{R}_2$ are symmetric. This is for instance the case of D-FEATS whose pattern has two feature atoms $x[f]y$ and $x[f]z$ where the variables y and z have a symmetrical role. Such rules require only one pointed version – in this example, D-FEATS•.

The rule G-SIMFULL• deserves a word as it can be hard to read. This rule handles the case where the pointed literal is a full similarity atom. In this case, one of the two variables must disappear from all the literals – except the full similarity atom – by being rewritten into the other variable. Such a rewriting can however create new patterns in the constraint. This is not a problem for literals that are already in the todo-stack. Rewritten literals from the store, however, have to be extracted and put back into the todo-stack. It is not necessary to do this for solved similarity atoms of the store[5]. As an example, consider the pointed constraint of Formula 5.3.

$$\langle \varepsilon \lessdot x[g]\uparrow \lessdot x =_\star y \mid x[f]z \wedge y[f]z' \wedge w =_\star x \rangle \tag{5.3}$$

Its pointed literal is $x =_\star y$. Say we decide to rewrite x into y. There are two occurrences of x in the store, namely in the literals $x[f]z$ and $w =_\star x$. The latter is a solved similarity which we simply want to update to note now that w is a copy of y and not x. The former can simply not remain in the store. Indeed, it will be rewritten into $y[f]z$ which forms a pattern with $y[f]z'$. On such a constraint, G-SIMFULL• has to extract $x[f]z$ from the store, rewrite it and place it in the todo-stack. The todo-stack has to be rewritten as well.

[5] Actually, it is important not to do it if we hope to have our system to terminate.

Pointed Deduction Rules

D-NFeat$^\bullet$-Feat	$\pi \mathbin{\scriptstyle\langle} \neg x[f]y \mid x[f]z \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} y \neq_\star z \mid x[f]z \wedge c$	
D-NFeat-Feat$^\bullet$	$\pi \mathbin{\scriptstyle\langle} x[f]z \mid \neg x[f]y \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} x[f]z \mathbin{\scriptstyle\langle} y \neq_\star z \mid c$	
D-Feats$^\bullet$-Sep	$\pi \mathbin{\scriptstyle\langle} x[f]z \mid y[f]z' \wedge \mathsf{S}(x,y) \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} x[f]z \mathbin{\scriptstyle\langle} \mathsf{S}(z,z') \mid y[f]z' \wedge \mathsf{S}(x,y) \wedge c$	$(z \neq z', \mathsf{S}(z,z') \not\preceq \langle \pi \mid c \rangle)$
D-Feats-Sep$^\bullet$	$\pi \mathbin{\scriptstyle\langle} \mathsf{S}(x,y) \mid x[f]z \wedge y[f]z' \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} \mathsf{S}(x,y) \mathbin{\scriptstyle\langle} \mathsf{S}(z,z') \mid x[f]z \wedge y[f]z' \wedge c$	$(z \neq z', \mathsf{S}(z,z') \not\preceq \langle \pi \mid c \rangle)$
D-FeatsEq$^\bullet$-Sep	$\pi \mathbin{\scriptstyle\langle} x[f]z \mid y[f]z \wedge \mathsf{S}(x,y) \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} x[f]z \mathbin{\scriptstyle\langle} x =_{\{f\}} y \mid y[f]z \wedge \mathsf{S}(x,y) \wedge c$	$(x =_{\{f\}} y \not\preceq \langle \pi \mid c \rangle)$
D-FeatsEq-Sep$^\bullet$	$\pi \mathbin{\scriptstyle\langle} \mathsf{S}(x,y) \mid x[f]z \wedge y[f]z \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} \mathsf{S}(x,y) \mathbin{\scriptstyle\langle} x =_{\{f\}} y \mid x[f]z \wedge y[f]z \wedge c$	$(x =_{\{f\}} y \not\preceq \langle \pi \mid c \rangle)$
D-Abs$^\bullet$-Sep	$\pi \mathbin{\scriptstyle\langle} x[F]{\uparrow} \mid y[G]{\uparrow} \wedge \mathsf{S}(x,y) \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} x[F]{\uparrow} \mathbin{\scriptstyle\langle} x =_{F \cap G} y \mid y[G]{\uparrow} \wedge \mathsf{S}(x,y) \wedge c$	$(x =_{\{F \cap G\}} y \not\preceq \langle \pi \mid c \rangle)$
D-Abs-Sep$^\bullet$	$\pi \mathbin{\scriptstyle\langle} \mathsf{S}(x,y) \mid x[F]{\uparrow} \wedge y[G]{\uparrow} \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} \mathsf{S}(x,y) \mathbin{\scriptstyle\langle} x =_{F \cap G} y \mid x[F]{\uparrow} \wedge y[G]{\uparrow} \wedge c$	$(x =_{\{F \cap G\}} y \not\preceq \langle \pi \mid c \rangle)$
D-NSim$^\bullet$	$\pi \mathbin{\scriptstyle\langle} x \neq_F y \mid c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} x \neq_F y \mathbin{\scriptstyle\langle} \mathsf{S}(x,y) \mid c$	$(\mathsf{S}(x,y) \not\preceq \langle \pi \mid c \rangle)$

Pointed Refinement Rules

R-NAbs$^\bullet$-Abs	$\pi \mathbin{\scriptstyle\langle} \neg x[F]{\uparrow} \mid x[G]{\uparrow} \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} \neg x[F \backslash G]{\uparrow} \mid x[G]{\uparrow} \wedge c$
R-NAbs-Abs$^\bullet$	$\pi \mathbin{\scriptstyle\langle} x[G]{\uparrow} \mid \neg x[F]{\uparrow} \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} x[G]{\uparrow} \mid \neg x[F \backslash G]{\uparrow} \wedge c$
R-Sims$^\bullet$	$\pi \mathbin{\scriptstyle\langle} x =_F y \mid x =_G y \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} x =_{F \cup G} y \mid c$
R-NSim$^\bullet$-Sim	$\pi \mathbin{\scriptstyle\langle} x \neq_F y \mid x =_G y \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} x \neq_{F \backslash G} y \mid x =_G y \wedge c$
R-NSim-Sim$^\bullet$	$\pi \mathbin{\scriptstyle\langle} x =_G y \mid x \neq_F y \wedge c$	$\Rightarrow$	$\pi \mathbin{\scriptstyle\langle} x =_G y \mid x \neq_{F \backslash G} y \wedge c$

Figure 5.11: System $\mathcal{R}_2^\bullet$ of Transformation Rules – Rules for negative literals

(0)			$x =_{\{f,g\}} y \mid x[f]z \wedge \neg y[f]z' \wedge z[\star]\uparrow \wedge z'[\star]\uparrow$
(1)	P-FEAT-SIM$^\bullet$	$\Rightarrow$	$x =_{\{f,g\}} y \,{\scriptstyle\langle}\, y[f]z \mid x[f]z \wedge \neg y[f]z' \wedge z[\star]\uparrow \wedge z'[\star]\uparrow$
(2)	D-NFEAT-FEAT$^\bullet$	$\Rightarrow$	$x =_{\{f,g\}} y \,{\scriptstyle\langle}\, y[f]z \,{\scriptstyle\langle}\, z \neq_\star z' \mid x[f]z \wedge z[\star]\uparrow \wedge z'[\star]\uparrow$
(3)	D-NSIM$^\bullet$	$\Rightarrow$	$x =_{\{f,g\}} y \,{\scriptstyle\langle}\, y[f]z \,{\scriptstyle\langle}\, z \neq_\star z' \,{\scriptstyle\langle}\, \mathrm{S}(z,z') \mid x[f]z \wedge z[\star]\uparrow \wedge z'[\star]\uparrow$
(4)	D-ABS-SEP$^\bullet$	$\Rightarrow$	$x =_{\{f,g\}} y \,{\scriptstyle\langle}\, y[f]z \,{\scriptstyle\langle}\, z \neq_\star z' \,{\scriptstyle\langle}\, \mathrm{S}(z,z') \,{\scriptstyle\langle}\, z =_\star z' \mid x[f]z \wedge z[\star]\uparrow \wedge z'[\star]\uparrow$
(5)	G-SIMFULL$^\bullet$	$\Rightarrow$	$x =_{\{f,g\}} y \,{\scriptstyle\langle}\, y[f]z \,{\scriptstyle\langle}\, z \neq_\star z \,{\scriptstyle\langle}\, \mathrm{S}(z,z) \,{\scriptstyle\langle}\, z[\star]\uparrow \,{\scriptstyle\langle}\, z =_\star z' \mid x[f]z \wedge z[\star]\uparrow$
(6)	MERGE$^{\bullet 3}$	$\Rightarrow$	$x =_{\{f,g\}} y \,{\scriptstyle\langle}\, y[f]z \,{\scriptstyle\langle}\, z \neq_\star z \mid \mathrm{S}(z,z) \wedge z =_\star z' \wedge x[f]z \wedge z[\star]\uparrow$
(7)	C-NSIMREFL$^\bullet$	$\Rightarrow$	$\bot$

Figure 5.13: Transformation of Formula 5.5 by $\mathcal{R}_2^\bullet$

The result of this application of G-SIMFULL$^\bullet$ is shown in Formula 5.4.

$$\left\langle \varepsilon \,{\scriptstyle\langle}\, y[g]\uparrow \,{\scriptstyle\langle}\, y[f]z \,{\scriptstyle\langle}\, x =_\star y \mid y[f]z' \wedge w =_\star y \right\rangle \tag{5.4}$$

After this, no other rule can apply as the full similarity atom is solved. This literal should then be moved to the store. In fact, the rules of $\mathcal{R}_2^\bullet$ do not show how this process of integrating a literal from the todo-stack into the store. In that regard, there lacks one important rule which applies only when no other rule can apply. This rule, MERGE$^\bullet$, is the following:

MERGE$^\bullet$ $\quad \pi \,{\scriptstyle\langle}\, l \mid c \quad \Rightarrow \quad \pi \mid l \wedge c$

The fact that MERGE$^\bullet$ only applies when no other rule can will be enforced in the implementation described in Section 5.2.3.

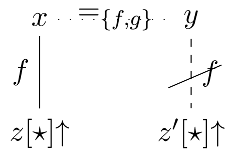


Figure 5.12: Formula 5.5

Let us now give an example of the transformation of a pointed constraint by $\mathcal{R}_2^\bullet$. Consider Formula 5.5. A graphical representation is given in Figure 5.12. Graphical representations of pointed constraints are the same as if they were simply constraints: we represent all the included literals.

$$x =_{\{f,g\}} y \mid x[f]z \wedge \neg y[f]z' \wedge z[\star]\uparrow \wedge z'[\star]\uparrow \tag{5.5}$$

Let us now describe the steps of transformation from Formula 5.5 in $\mathcal{R}_2^\bullet$. Intermediary steps can be found in Figure 5.13. Some steps represent the action of several rules when we believe it is not important for the reader to have them made fully explicit.

(0) Start from Formula 5.5. Notice the pattern $x =_{\{f,g\}} y \mid x[f]z$.

(1) Rewrite the pointed constraint using P-FEAT$^\bullet$-SIM. This introduces the feature atom $y[f]z$. Since this atom might trigger changes in the store, we do not introduce it in there just yet. Notice now the pattern $y[f]z \mid \neg y[f]z'$.

(2) Rewrite the pointed constraint using D-NFEAT-FEAT$^\bullet$. This removes the literal $\neg y[f]z'$ from the store and introduces the literal $z \neq_\star z'$. Since this literal might trigger changes in the store, we do not introduce it in there just yet. Indeed, Definition 5.1 requires that a pair of variables between which there is a negated similarity atom be considered separated. In $\mathcal{R}_2^\bullet$, this translates to the applicability of the rule D-NSIM$^\bullet$.

(3) Rewrite the pointed constraint using D-NSim$^\bullet$. This introduces the extended literal $\mathrm{S}(z, z')$ which creates the pattern $\mathrm{S}(z, z') \mid z[\star]\!\uparrow \wedge z'[\star]\!\uparrow$.

(4) Rewrite the pointed constraint using D-Abs-Sep$^\bullet$. This introduces the atom $z =_\star z'$ and creates the pattern $\pi \triangleleft z =_\star z' \mid c$.

(5) Rewrite the pointed constraint using G-SimFull$^\bullet$. Both z and z' could be rewritten into the other and, for simplicity, we will choose to rewrite z' into z. The literals of c rewritten by application of G-SimFull$^\bullet$ (here, there is only $z'[\star]\!\uparrow$) might trigger other patterns. They can therefore not be left in the store and are scheduled to be added again later. The solved similarity atom $z =_\star z'$ cannot trigger any subsequent rule and can therefore be merged. Similarly, the two literals $z[\star]\!\uparrow$ and $\mathrm{S}(z, z')$ cannot trigger any rule and can be merged.

(6) Rewrite the pointed constraint twice using Merge$^\bullet$. Since there is already $z[\star]\!\uparrow$ in the store, this simply removes the duplicate one. The next literal is the negated similarity atom $z \neq_\star z$.

(7) Rewrite the pointed constraint using C-NSimRefl$^\bullet$. This detects an inconsistency – the reflexive negated similarity atom – and replaces the whole pointed constraint by $\bot$.

Notice that, in this example, one can see $\pi \mid c$ as "$\pi \wedge c$" and apply rules of $\mathcal{R}_2$ to it. This shows a strong relationship between $\mathcal{R}_2^\bullet$ and $\mathcal{R}_2$. This is exactly the topic of the next subsection which will formalise this remark.

5.2.2 Links Between $\mathcal{R}_2^\bullet$ and $\mathcal{R}_2$

The rules of $\mathcal{R}_2^\bullet$ simulate rules of $\mathcal{R}_2$. In order to formalise that, we introduce the *semantics of a pointed constraint* in Definition 5.5. We then show that if a rule of $\mathcal{R}_2^\bullet$ applies to a pointed constraint p, then a rule of $\mathcal{R}_2$ applies to the semantics of p. Moreover, we show that Merge$^\bullet$, applied to a pointed constraint, leaves its semantics unchanged. These two properties will be described later in Lemmas 5.6 and 5.7.

Definition 5.5 (Semantics of Pointed Constraints)**.** The *semantics of a pointed constraint* p, written $[\![p]\!]$, is the constraints made of all the literals that occur in p. Formally:

$$\begin{array}{rcll} [\![\varepsilon]\!] & = & \top & \text{— Empty stack} \\ [\![\pi \triangleleft l]\!] & = & [\![\pi]\!] \wedge l & \text{— Non-empty stack} \\ [\![\langle \pi \mid c \rangle]\!] & = & [\![\pi]\!] \wedge c & \text{— Pointed constraint} \end{array}$$

□

In order to link the applicability of a regular transformation rule on the semantics and the applicability of a pointed transformation rule on the pointed constraint, we need to define what it means for a rule to be *applicable in a context*. This is the topic of Definition 5.6.

Definition 5.6 (Applicability of a Rule in a Context)**.** Given two constraints c and c', a transformation rule is said to be *applicable to* c *in the context* c' if it is applicable[6] to $c \wedge c'$ with its pattern fully contained in c. □

Informally, a rule is applicable to c on the context c' if it is applicable to c in the usual sense, but the side-conditions – and the conditions of subsumption in particular – also consider the literals of c'. This is stronger than the usual applicability. In particular, if a rule is applicable to a constraint c in any context, then it is applicable to c. The contrary is not always true. For instance, the rule P-Abs-Sim is applicable to $x[F]\!\uparrow \wedge x =_G y$ but it is not applicable to $x[F]\!\uparrow \wedge x =_G y$ *in the context of* $y[F]\!\uparrow$.

[6] The applicability of a rule is defined in Section 4.1.1.

The notion of *irreducibility of a constraint with respect to rules in a context* is fairly natural. It can be found in Definition 5.7.

Definition 5.7 (Irreducibility of a Constraint With Respect to a Rule in a Context)**.** A constraint c is *irreducible with respect to a set of rules* R *in a context* c' if non of the rules of R applies to c in the context c'. □

With this notion of irreducibility in a context, we can formalise what it means for a pointed constraint to be *well-formed* in Definition 5.8.

Definition 5.8 (Well-Formedness)**.** A pointed constraint $p = \langle \pi \mid c \rangle$ is *well-formed* if the following hold:

1. All full similarity atoms of the store c are solved inside p, that is for all $x =_\star y \in c$:

$$\mathcal{V}_s(\llbracket p \rrbracket) \cap \{x, y\} \neq \varnothing$$

2. The store c is irreducible with respect to $\mathcal{R}_2$ in the context π. □

The rest of this section will go as follows. We will first state and prove that all the rules of $\mathcal{R}_2^\bullet$ as well as Merge$^\bullet$ are written so as to respect well-formedness in Lemmas 5.4 and 5.5. We will then proceed to show in Lemma 5.6 that rules of $\mathcal{R}_2^\bullet$ simulate rules of $\mathcal{R}_2$, that is if a rule of $\mathcal{R}_2^\bullet$ applies to a pointed constraint p, then a rule of $\mathcal{R}_2$ applies to $\llbracket p \rrbracket$. We will then show in Lemma 5.7 that Merge$^\bullet$ leaves the semantics of pointed constraints unchanged. This will allow us to conclude that $\mathcal{R}_2^\bullet$ is a sound and complete strategy for $\mathcal{R}_2$ and proceed to give a new implementation of `transform-2` in Section 5.2.3.

Lemma 5.4 ($\mathcal{R}_2^\bullet$ conserves well-formedness)**.** *If* p *is well-formed and* $p \Rightarrow p'$ *via* $\mathcal{R}_2^\bullet$*, then* p' *is well-formed.*

Proof. Consider $p \Rightarrow p'$ via $\mathcal{R}_2^\bullet$. Let us discuss the rules of $\mathcal{R}_2^\bullet$ and discuss why they all respect well-formedness.

- Clash rules[7] respect well-formedness trivially because $\bot$ does not contain any full similarity atom (Point 1) and no rule of $\mathcal{R}_2$ can apply to $\bot$ (Point 2).
- Most other rules[8] (eg. P-Sims$^\bullet$) do not modify the story. They only introduce a literal into the todo-stack. Moreover, this literal only contains variables that were present in other unsolved literals. This means that Point 1 holds because the set of full similarity atoms of the store is unchanged and because it is not possible that the rule introduced a literal about a previously-solved variable. This also means that Point 2 holds because the store did not change while the todo-stack grew, making the subsumption only stronger (Lemma 4.4).
- Some remaining rules[9] do remove or modify a literal from the story or the todo-stack. However:
  - D-NFeat$^\bullet$-Feat and D-NFeat-Feat$^\bullet$ remove a negated feature atom. These do not have any impact on the subsumption.
  - R-NAbs$^\bullet$-Abs and R-NSim$^\bullet$-Sim refine a negated absence or similarity atom in the todo-stack. This only makes subsumption stronger but does not change anything about solved variables or patterns present in the store.

[7] C-Cycle$^\bullet$, C-Feat$^\bullet$-Abs, C-Feat-Abs$^\bullet$, C-NAbsEmpty$^\bullet$, C-NSimRefl$^\bullet$ and C-NSimEmpty$^\bullet$.

[8] D-Feats$^\bullet$, P-Feat$^\bullet$-Sim, P-Feat-Sim$^\bullet$, P-Abs$^\bullet$-Sim, P-Abs-Sim$^\bullet$, P-Sims$^\bullet$, D-Feats$^\bullet$-Sep, D-Feats-Sep$^\bullet$, D-FeatsEq$^\bullet$-Sep, D-FeatsEq-Sep$^\bullet$, D-Abs$^\bullet$-Sep, D-Abs-Sep$^\bullet$ and D-NSim$^\bullet$.

[9] D-NFeat$^\bullet$-Feat, D-NFeat-Feat$^\bullet$, R-NAbs$^\bullet$-Abs, R-NAbs-Abs$^\bullet$, R-Sims$^\bullet$, R-NSim$^\bullet$-Sim and R-NSim-Sim$^\bullet$.

- R-NAbs-Abs$^\bullet$ and R-NSim-Sim$^\bullet$ refine a negated absence or similarity atom in the store. This cannot create a new pattern for any rule of $\mathcal{R}_2$. Moreover, as for R-NAbs$^\bullet$-Abs and R-NSim$^\bullet$-Sim, this only makes subsumption stronger.

- R-Sims$^\bullet$ removes similarity atoms both from the todo-stack and the store and introduce a stronger similarity atom in the todo-stack. This only removes patterns from the store while making the subsumption stronger.

- Finally, G-SimFull$^\bullet$ does not make the full similarity atoms of the store unsolved. Moreover, if c is irreducible with respect to $\mathcal{R}_2$ in the context π, then $\mathcal{G}_x(c)$ is too, and $\mathcal{G}_x^{\bar{\varepsilon}}(c)$ also (Lemma 4.6). □

Lemma 5.5 (Merge$^\bullet$ conserves well-formedness)**.** *If p is well-formed and irreducible with respect to $\mathcal{R}_2^\bullet$ and $p \Rightarrow p'$ via Merge$^\bullet$, then p' is well-formed.*

Proof. Consider two pointed constraints p and p' such that p is well-formed and irreducible with respect to $\mathcal{R}_2^\bullet$ and $p \Rightarrow p'$ via Merge$^\bullet$. There exists π, l and c such that:

$$p = \langle \pi \,\langle\, l \mid c \rangle \qquad\qquad p' = \langle \pi \mid l \wedge c \rangle$$

Let us show that p' is well-formed. We need to show that the points in Definition 5.8 hold.

- For Point 1, let us take any full similarity atom in $l \wedge c$. If it is l then, because p is irreducible with respect to G-SimFull$^\bullet$, it is solved in p. Otherwise, it is in c and was solved in p before, and therefore in π, l and c and therefore in p'.

- For Point 2, notice that the notion of subsumption has not changed between p and p' since their semantics are equal. In general, that means that the side-conditions that held in p still hold in p'. This means that any rule whose pattern is fully in c was not applicable in $[\![p]\!]$ and is still not applicable in $[\![p']\!]$. Moreover, any rule whose pattern is in $l \wedge c$ and contains l would have a pointed counterpart in $\mathcal{R}_2^\bullet$ applicable in p, which contradicts the fact that p is irreducible with respect to $\mathcal{R}_2^\bullet$. □

Let us now show that if a rule of $\mathcal{R}_2^\bullet$ applies to a pointed constraint p, then a rule of $\mathcal{R}_2$ applies to the semantics of p. This is stated in Lemma 5.6. An illustration is shown in Figure 5.14.

$$\begin{tikzcd} p \arrow[r, Rightarrow, "\mathcal{R}_2^\bullet"] \arrow[d, "{[\![\cdot]\!]}"'] & p' \arrow[d, "{[\![\cdot]\!]}"] \\ {[\![p]\!]} \arrow[r, Rightarrow, "\mathcal{R}_2"] & {[\![p']\!]} \end{tikzcd}$$

Figure 5.14: Commutative diagram to illustrate Lemma 5.6

Lemma 5.6 ($\mathcal{R}_2^\bullet$ simulates a strategy for $\mathcal{R}_2$)**.** *For any pointed constraints p and p' such that p is well-formed and $p \Rightarrow p'$ via $\mathcal{R}_2^\bullet$, then $[\![p]\!] \Rightarrow [\![p']\!]$ via $\mathcal{R}_2$.*

Proof. Let us take p and p' two pointed constraints such that $p \Rightarrow p'$ via $\mathcal{R}_2^\bullet$ and p is well-formed. Assume for instance that $p \Rightarrow p'$ via P-Abs$^\bullet$-Sim. In this situation, we have:

$$\begin{aligned} p &= \langle \pi \,\langle\, x[F]\!\uparrow \mid x =_G z \wedge c \rangle \\ p' &= \langle \pi \,\langle\, z[F \cap G]\!\uparrow \,\langle\, x[F]\!\uparrow \mid x =_G z \wedge c \rangle \\ [\![p]\!] &= [\![\pi]\!] \wedge x[F]\!\uparrow \wedge\, x =_G z \wedge c \\ [\![p']\!] &= [\![\pi]\!] \wedge z[F \cap G]\!\uparrow \wedge\, x[F]\!\uparrow \wedge\, x =_G z \wedge c \end{aligned}$$

with $z[F \cap G]\!\uparrow \not\sqsubseteq \langle \pi \mid c \rangle$.

Let us show that $[\![p]\!] \Rightarrow [\![p']\!]$ via P-Abs-Sim. Clearly, the pattern for this rule, $x[F]\!\uparrow \wedge\, x =_G z$ is present in $[\![p]\!]$. Moreover, the side-condition is respected because $z[F \cap G]\!\uparrow \not\sqsubseteq \langle \pi \mid c \rangle$ is exactly the same as $z[F \cap G]\!\uparrow \not\sqsubseteq [\![\pi]\!] \wedge c$. P-Abs-Sim can therefore indeed apply on $[\![p]\!]$ and yields $[\![p']\!]$.

$$\begin{tikzcd}
\langle \varepsilon \mathbin{«} c \mid \top \rangle = p_0 \arrow[rr, Rightarrow, "(\mathcal{R}_2^\bullet)^{n_0}"] \arrow[d, "\llbracket\cdot\rrbracket"'] & & p_1 \arrow[rr, Rightarrow, "\textsc{Merge}^\bullet"] \arrow[dr, "\llbracket\cdot\rrbracket"] & & p_1' \arrow[rr, Rightarrow, "(\mathcal{R}_2^\bullet)^{n_1}"] \arrow[dl, "\llbracket\cdot\rrbracket"'] & & \cdots \arrow[rr, Rightarrow, "\textsc{Merge}^\bullet"] \arrow[dr, "\llbracket\cdot\rrbracket"] & & p_k = \langle \varepsilon \mid c' \rangle \arrow[dl, "\llbracket\cdot\rrbracket"'] \\
c = \llbracket p_0 \rrbracket \arrow[rrr, Rightarrow, "(\mathcal{R}_2)^{n_0}"] & & & \llbracket p_1 \rrbracket \arrow[rrr, Rightarrow, "(\mathcal{R}_2)^{n_1}"] & & & \cdots \quad \cdots & \llbracket p_k \rrbracket = c' &
\end{tikzcd}$$

Figure 5.16: Illustration of $\mathcal{R}_2^\bullet \cup \{\textsc{Merge}^\bullet\}$ as a full strategy for $\mathcal{R}_2$

All the other cases are similar. □

Note that well-formedness explains that $\mathcal{R}_2^\bullet$ is a complete strategy for $\mathcal{R}_2$ as it means that if $\langle \pi \mathbin{‹} l \mid c \rangle$ is irreducible with respect to $\mathcal{R}_2^\bullet$, then $l \wedge c$ is irreducible with respect to $\mathcal{R}_2$ in the context π.

$$\begin{tikzcd}
p \arrow[rr, Rightarrow, "\textsc{Merge}^\bullet"] \arrow[dr, "\llbracket\cdot\rrbracket"'] & & p' \arrow[dl, "\llbracket\cdot\rrbracket"] \\
& \llbracket p \rrbracket &
\end{tikzcd}$$

Figure 5.15: Commutative diagram to illustrate Lemma 5.7

The additional rule, $\textsc{Merge}^\bullet$, has no impact on the semantics. In that sense, it is only a logistical rule of $\mathcal{R}_2^\bullet$. This is stated in Lemma 5.7. An illustration is shown in Figure 5.15.

Lemma 5.7 ($\textsc{Merge}^\bullet$ **leaves the semantics unchanged).** *If $p \Rightarrow p'$ via $\textsc{Merge}^\bullet$, then* $\llbracket p' \rrbracket = \llbracket p \rrbracket$.

Lemmas 5.6 and 5.7 allow us to see $\mathcal{R}_2^\bullet$ as a strategy for $\mathcal{R}_2$. The well-formedness allows to conclude that pointed constraints that are irreducible with respect to both $\mathcal{R}_2^\bullet$ and $\textsc{Merge}^\bullet$ are of the form $\langle \varepsilon \mid c \rangle$ with c irreducible with respect to $\mathcal{R}_2$. Lemmas 5.4 and 5.5 explain that well-formedness is respected by $\mathcal{R}_2^\bullet$ and $\textsc{Merge}^\bullet$. All these results are the bricks to build a new decision procedure for the satisfiability of constraints. This is the topic of the next subsection.

5.2.3 Deciding the Satisfiability of Constraints

The previous subsection built the tools to justify that we can use $\mathcal{R}_2^\bullet \cup \{\textsc{Merge}^\bullet\}$ as a full strategy of evaluation for $\mathcal{R}_2$. Indeed, say one wants to apply $\mathcal{R}_2$ fully on a constraint c.

1. We start by defining a pointed constraint $p_0 = \langle \varepsilon \mathbin{«} c \mid \top \rangle$. p_0 is trivially well-formed. Moreover, $\llbracket p_0 \rrbracket = c$.
2. One can then apply $\mathcal{R}_2^\bullet$ on p_0 for as many times as it is possible – call that number n_0. This yields a pointed constraint p_1. By Lemma 5.6, there are n_0 steps of $\mathcal{R}_2$ that allow to transform $\llbracket p_0 \rrbracket$ into $\llbracket p_1 \rrbracket$. By Lemma 5.4, p_1 is well-formed.
3. One can then apply $\textsc{Merge}^\bullet$ on p_1. This yields a pointed constraint p_1'. By Lemma 5.7, $\llbracket p_1' \rrbracket = \llbracket p_1 \rrbracket$. By Lemma 5.5, and since p_1 is irreducible with respect to $\mathcal{R}_2^\bullet$, p_1' is well-formed.
4. We can repeat the steps 2 and 3 as many times as necessary until we reach a pointed constraint p_k to which neither $\mathcal{R}_2^\bullet$ nor $\textsc{Merge}^\bullet$ can apply. p_k is well-formed and there is a chain a transformations of $\mathcal{R}_2$ from $\llbracket p_0 \rrbracket$ to $\llbracket p_k \rrbracket$. Since $\textsc{Merge}^\bullet$ is not applicable to p_k, p_k must be of the form $\langle \varepsilon \mid c' \rangle$. Since p_k is well-formed, c' is irreducible with respect to $\mathcal{R}_2$. Moreover, $\llbracket p_k \rrbracket = c'$.

This whole process is illustrated in Figure 5.16.

Figure 5.17 introduces a function `transform-2-pointed` which does exactly what we have described in this example. It assumes given a well-formed pointed constraint. Instead of using `transform-2`(c) from Section 5.1.3, one can thus use `transform-2-pointed`$(\langle \varepsilon \mathbin{«} c \mid \top \rangle)$. This is basically doing the

```
function apply-rule-2-pointed (rule, pointed-constraint)
    : pointed-constraint or $\bot$

function choose-rule-2-pointed ($p$ : pointed-constraint) : rule
  return any rule of $\mathcal{R}_2^\bullet$ applicable to $p$

function transform-2-pointed ($p$ : pointed-constraint)
    : constraint or $\bot$
  if $p$ is reducible in $\mathcal{R}_2^\bullet$
    let $r$ = choose-rule-2-pointed($p$)
    match apply-rule-2-pointed($r$, $p$)
    | $\bot$ -> return $\bot$
    | $p'$ -> return transform-2-pointed($p'$)
  else
    match $p$
    | $\langle \varepsilon \mid c \rangle$ -> return $c$
    | $\langle \pi \langle l \mid c \rangle$ -> return transform-2-pointed($\langle \pi \mid l \land c \rangle$)
```

Figure 5.17: Function transform-2-pointed

same thing except that transform-2-pointed does not suffer the issue of transform-2 which requires matching patterns anywhere in a constraint.

This is particularly useful to decide the satisfiability of a constraint incrementally. Say one wants to decide whether $c \land c'$ is satisfiable, knowing that c' is irreducible with respect to $\mathcal{R}_2$. In $\mathcal{R}_2$, the only solution is to call the whole function transform-2 on $c \land c'$. Using $\mathcal{R}_2^\bullet$, we can leverage the fact that c' is already irreducible with respect to $\mathcal{R}_2$ to avoid trying to find patterns in it.

Consider Figure 5.18. In this figure, we redefine transform-2 as a simpler wrapper around transform-2-pointed. We also define a function add-transform-2 that applies $\mathcal{R}_2$ on $c \land c'$ when c' is irreducible with respect to $\mathcal{R}_2$. This basically consists in considering $\langle \varepsilon \ll c \mid c' \rangle$. However, the pointed constraint given to transform-2-pointed must be well-formed. Therefore, we need to make sure first that all the full similarity constraints of c' are solved. This goes by rewriting c using the rule G•-SimFull:

$$\textsc{G}^\bullet\textsc{-SimFull} \qquad \pi \mid x =_\star y \land c \quad \Rightarrow \quad \pi\{x \mapsto y\} \mid x =_\star y \land c \quad (x \in \mathcal{V}(\pi), x \notin \mathcal{V}(c))$$

5.2.4 Discussions

The technique developed in this section is not specific to $\mathcal{R}_2$ in any way. It can very easily be adapted to other systems such as $\mathcal{R}_1$ or $\mathcal{R}_2$ extended with all the extra rules mentioned in Section 5.1.4.

One natural consideration with these pointed rules is whether it makes a difference to insert the new literals in the todo-stack under the pointed literal or above it – at the top of the stack. Consider for instance the rule P-Feat-Sim•.

$$\textsc{P-Feat-Sim}^\bullet \qquad \pi \langle x =_G z \mid x[f]y \land c \quad \Rightarrow \quad \pi \langle z[f]y \langle x =_G z \mid x[f]y \land c \qquad (f \in G, z[f]y \not\preceq \langle \pi \mid c \rangle)$$

Is there a difference between this formulation or a formulation in which the added literal $z[f]y$ appeared on top of the stack, as in the example below?

$$\textsc{P-Feat-Sim}^\bullet\textsc{(alt)} \qquad \pi \langle x =_G z \mid x[f]y \land c \quad \Rightarrow \quad \pi \langle x =_G z \langle z[f]y \mid x[f]y \land c \qquad (f \in G, z[f]y \not\preceq \langle \pi \mid c \rangle)$$

```
function replace-in-todo-stack ($p$ : pointed-constraint)
    : pointed-constraint
  if $\text{G}^\bullet$-SimFull is applicable to $p$
    return replace-in-todo-stack(apply-rule-2($\text{G}^\bullet$-SimFull, $p$))
  else
    return $p$

function add-transform-2 ($c$ : constraint, $c'$ : constraint)
    : constraint or $\bot$
  let $p$ = $\langle \varepsilon \ll c \mid c' \rangle$
  let $p'$ = replace-in-todo-stack($p$)
  return transform-2-pointed($p'$)

function transform-2 ($c$ : constraint) : constraint or $\bot$
  return add-transform-2-pointed($c$, $\top$)
```

Figure 5.18: Functions transform-2 and add-transform-2

The short answer is that, because most of our rules have side-conditions about subsumption, there is no difference at all in term of correction of the algorithm. In fact, because of this very reason, the order of literals in the stack does not really matter for correction of the algorithm.

There remains considerations of complexity. In that regard, we believe the way to improve efficiency is to write rules in a way that increases the chances to find a clash. For instance, when introducing a similarity, it might be interesting to do so on the top of the stack. Indeed, these literals can bring huge changes to the store, including possibly by rewriting one variable into the other. Such a computation is more likely to detect a clash between several literals. We do not know of formal arguments supporting one order of the other. In an actual implementation, the right way to proceed is therefore to check such intuitions by testing and benchmarking different solutions.

It might seem unsatisfactory that the test of subsumption needs to consider the todo-stack as well as the store. Indeed, this means that a lot of rules need to crawl the whole todo-stack to decide whether they are applicable, which seem to defeat the very locality provided by pointed constraints. As mentioned in Section 4.1, the test of subsumption is local in the sense that we only need to check the literals that use exactly the same variables as the literal we are considering. In an actual implementation, that means that we want to organise both the store and the todo-stack so as to have an easy way to access the literals by the variables they contain.

5.3 Threaded Constraints

Section 5.1 has presented a system of rules $\mathcal{R}_2$ that do not introduce disjunctions in its handling of constraints and Section 5.2 has presented an efficient and incremental way to test satisfiability of these constraints. This system and its decision procedure are an important step towards computing faster on specifications. However, they do not solve the problem of disjunctions coming from the specifications themselves, and in particular the ones coming from the function noresolve, presented in Figure 3.20 and restated in this section for convenience.

Unfortunately, our constraints are not rich enough to express the non-resolution of a path. We envision two solutions to this problem. The first approach is to extend the logical language by introducing new predicates which express properties on paths. This has been done for feature tree logics in the past [Backofen & Smolka 1995; Backofen 1995] as a way to obtain the property of quantifier elimination in the strict

```
function noresolve-s($x$ : variable, $\pi$ : variable stack, $q$ : path)
    : $\Sigma_1$-formula
  match $q$
  |    /   -> return $\bot$
  |   $f/q'$ -> return $x[f]\uparrow \vee \exists y \cdot (x[f]y \wedge$ noresolve-s$(y, \pi ‹ x, q'))$
  |   ./$q'$ -> return noresolve-s$(x, \pi, q')$
  |  ../$q'$ ->
    match $\pi$
    | $\varepsilon$ -> return noresolve-s$(x, \varepsilon, q')$
    | $\pi' ‹ y$ -> return noresolve-s$(y, \pi', q')$

function noresolve($r$ : variable, cwd : path, $p$ : path)
    : $\Sigma_1$-formula
  match $p$
  | abs($q$) -> return noresolve-s$(r, \varepsilon,$     $q)$
  | rel($q$) -> return noresolve-s$(r, \varepsilon,$ cwd/$q)$
```

Figure 3.20: Function noresolve for any path

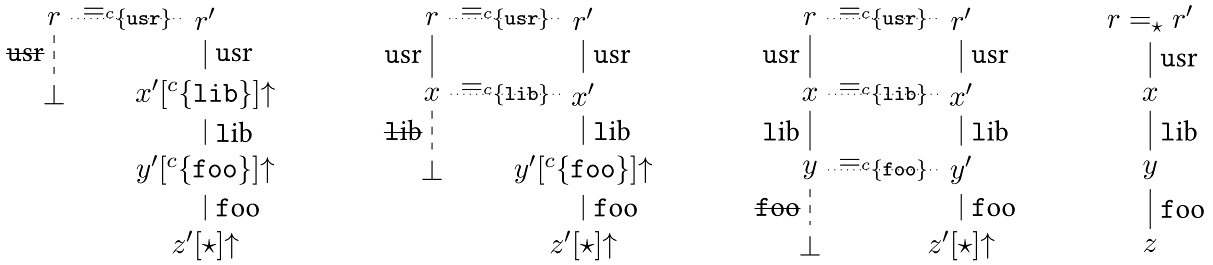


Figure 5.19: Specification cases of mkdir -p /usr/lib/foo

sense. We refer the reader to the introduction of Section 4.2 for this aspect. In the case of feature tree logics, the needed extension of the language is however substantial, and that is already the case without a similarity predicate.

For instance, if q is a list of features, what can be said of $\exists x \cdot (x[q]\uparrow \wedge x[f]y)$, where $x[q]\uparrow$ is expresses that the path q does not exist in x? What does it imply on y? In this case, it depends whether q starts with f or not. If it does, that is if $q = f › q'$, then this implies $y[q']\uparrow$. If it does not, then this does not imply anything on y. This example shows that the handling of path constraints requires new rules that will inspect paths and potentially propagate path information from variables to others.

This first approach is also specific to path constraints. It does allow us to reimplement resolve as one path atom and noresolve as one negated path atom. Although resolve is the main reason why we are considering this issue, we would also enjoy it if it could be more general. Consider the utility call mkdir -p /usr/lib/foo for example. mkdir -p creates the given directories recursively if they do not exist already: if /usr does not exist, it is created, then if /usr/lib does not exist, it is created, and finally, if /usr/lib/foo does not exist, it is created. The specification for this call comprises four cases depending on the status of /usr/lib/foo. They are represented in Figure 5.19. This specification requires disjunctions as it is out of reach of constraints, even extended with path atoms.

A second approach, presented in this section, is to extend the expressivity of constraints so that they can hide certain forms of disjunctions. We use our test of satisfiability as a test of entailment which we can then use to build a decision procedure for such richer constraints called *threaded*. These threaded constraints are the pair of a main constraint and a list of threads. Threads are basically of the form $l \rightarrow t$ where l is a literal and t is another threaded constraint. They are used to postpone computation until we reach a situation where we know for sure that l holds, which is when l is entailed by the main constraint.

This approach is quite similar to the notion of *residuation* used in constraint logic programming as a strategy to mitigate combinatorial explosions by delaying computation [Smolka 1993]. This is used in the Oz programming model to implement *task synchronisation* [Smolka 1995]. The term of residuation itself comes from previous work on logics and functional programming also introducing mechanisms to delay computation [Aït-Kaci & Nasr 1989]. These notions of having a main constraint from which we can *ask* entailment questions and *tell* new facts also reminds of the work of Saraswat and Rinard on concurrent constraint programming [Saraswat & Rinard 1990].

We introduce such threaded constraints in Section 5.3.1 and show how they allow us to redefine `noresolve`. We discuss properties of such constraints in Section 5.3.2. Finally, we show how to build functions to reason about such constraints Section 5.3.3 and we discuss them in Section 5.3.4.

5.3.1 Entailment and Threaded Constraints

Let us first define the notion of *entailment* of a formula by another one in Definition 5.9 and state its relationship to satisfiability in Lemma 5.8.

Definition 5.9 (Entailment)**.** A formula ϕ *entails* a formula ψ, written $\phi \models \psi$ if ϕ implies ψ in any valuation, that is if $\models_{\mathcal{FT}} \tilde{\forall} \cdot (\phi \rightarrow \psi)$.

A formula ϕ *disentails* a formula ψ if ϕ entails $\neg\psi$.[10] □

Lemma 5.8 (Entailment and Satisfiability)**.** *ϕ entails ψ if and only if the conjunction of ϕ and the negation of ψ is unsatifiable. Formally, $\phi \models \psi$ if and only if $\models_{\mathcal{FT}} \neg\tilde{\exists} \cdot (\phi \wedge \neg\psi)$.*

Proof. For any formulas ϕ and ψ, $\phi \rightarrow \psi$ is defined as $\neg\phi \vee \psi$ and the following chain of equivalences holds:

$$\begin{aligned} & \tilde{\forall} \cdot (\neg\phi \vee \psi) \\ \leftrightarrow\ & \neg\neg\tilde{\forall} \cdot (\neg\phi \vee \psi) \\ \leftrightarrow\ & \neg\tilde{\exists} \cdot \neg(\neg\phi \vee \psi) \\ \leftrightarrow\ & \neg\tilde{\exists} \cdot (\phi \wedge \neg\psi) \end{aligned}$$

Therefore, $\phi \models \psi$ if and only if we do not have $\models_{\mathcal{FT}} \tilde{\exists} \cdot (\phi \wedge \neg\psi)$. □

Lemma 5.8 shows that the notion of entailment is close to that of satisfiability. We can therefore leverage our results on the satisfiability from previous sections to handle this new notion.

We already know how to handle constraints incrementally and decide their satisfiability. We are going to build a solver able to handle *threaded constraints*. They are composed of a *main constraint* to which we add *threads*. Each thread is an implication whose left-hand side is a literal called *guard* and whose right-hand side is another threaded constraint. The formal definition of threaded constraints can be found in Definition 5.10.

[10] Note that disentailing is different from not entailing.

Definition 5.10 (Threaded Constraints). A *threaded constraint* t is a formula of the form

$$t = c \wedge \bigwedge_{i=1}^{n} (l_i \rightarrow t_i)$$

where c is a constraint, $n \geq 0$ and for all i, l_i is a literal and t_i is a threaded constraint. This definition is taken to be inductive, that is we consider only finite formulas. The base case for the induction occurs when $n = 0$. c is the *main constraint* of t. Each $l_i \rightarrow t_i$ is a *thread* of t. Each l_i is the *guard* of the thread $l_i \rightarrow t_i$. We typically use τ to represent the part of the threaded constraint consisting of only threads. □

Basically, we are going to use the current solver as before on the main constraint. The threads will remain inactive until the guard is entailed by the main constraint, in which case the threads will be *activated* and their threaded constraints merged with the main constraint. This is a way to postpone some computation until the moment when we are sure that it does matter. In term of worst-case complexity, this strategy does not bring any improvement. We will see, however, that in a lot of real-world situations, postponing is useful and important.

Such threaded constraints can be used to express non-resolution of a path. As an example, recall Formula 3.8 that specifies the preconditions of the error cases of `rm /etc/rancid/lg.conf`:

$$\begin{array}{lcll} {\phi_2}^{(p)}(r) & = \exists x, y \cdot & r[\mathtt{etc}]\uparrow \\ & & \vee\ (r[\mathtt{etc}]x \wedge x[\mathtt{rancid}]\uparrow) \\ & & \vee\ (r[\mathtt{etc}]x \wedge x[\mathtt{rancid}]y \wedge y[\mathtt{lg.conf}]\uparrow) \end{array} \tag{3.8}$$

These are in fact the result of the unfolding of the `noresolve` macro, defined in Figure 3.20. We can however unify[11] these specification cases in one thread by keeping the disjunctions inside of the formula:

$$\phi_2^{(p)}(r) = \exists x, y \cdot (\neg r[\mathtt{etc}]\uparrow \rightarrow (r[\mathtt{etc}]x \wedge (\neg x[\mathtt{rancid}]\uparrow \rightarrow (x[\mathtt{rancid}]y \wedge y[\mathtt{lg.conf}]\uparrow)))) \tag{5.6}$$

This presentation can be seen as a lazy form of the non-resolution of the path `/etc/rancid/lg.conf`. The macro `noresolve` can thus be slightly rewritten to produce a similar thread instead of a disjunction. The new formulation can be found in Figure 5.20.

These threaded constraints can also be used to merge several specification cases into one. Let us consider the specification of `rmdir /usr/lib`. It has one success case (Formula 5.7) and two error cases (Formulas 5.8 and 5.9). Graphical representations can be found in Figures 5.21 and 5.22.

$$\begin{array}{lcl} {\phi_1}^{(p)}(r) & = & \exists x, y, z \cdot (r[\mathtt{usr}]x \wedge x[\mathtt{lib}]y \wedge y[\star]\uparrow) \\ {\phi_1}^{(t)}(r, r') & = & \exists x, x' \cdot (r =_{c\{\mathtt{usr}\}} r' \wedge x =_{c\{\mathtt{lib}\}} x' \wedge r'[\mathtt{usr}]x' \wedge x'[\mathtt{lib}]\uparrow) \\ \phi_1 \quad (r, r') & = & {\phi_1}^{(p)}(r) \wedge {\phi_1}^{(t)}(r, r') \end{array} \tag{5.7}$$

$$\begin{array}{lcl} {\phi_2}^{(p)}(r) & = & \exists x \cdot (\neg r[\mathtt{usr}]\uparrow \rightarrow (r[\mathtt{usr}]x \wedge x[\mathtt{lib}]\uparrow)) \\ {\phi_2}^{(t)}(r, r') & = & r =_\star r' \\ \phi_2 \quad (r, r') & = & {\phi_2}^{(p)}(r) \wedge {\phi_2}^{(t)}(r, r') \end{array} \tag{5.8}$$

$$\begin{array}{lcl} {\phi_3}^{(p)}(r) & = & \exists x, y \cdot (r[\mathtt{usr}]x \wedge x[\mathtt{lib}]y \wedge \neg y[\star]\uparrow) \\ {\phi_3}^{(t)}(r, r') & = & r =_\star r' \\ \phi_3 \quad (r, r') & = & {\phi_3}^{(p)}(r) \wedge {\phi_3}^{(t)}(r, r') \end{array} \tag{5.9}$$

[11] Using the fact that $A \vee B$ is equivalent to $\neg A \rightarrow B$ by definition.

```
function noresolve-s($x$ : variable, $\pi$ : variable stack, $q$ : path)
    : $\Sigma_1$-formula
  match $q$
  |   /   -> return $\bot$
  |    $f$ -> return $x[f]\uparrow$
  |  $f/q'$ -> return $\neg x[f]\uparrow \rightarrow \exists y \cdot (x[f]y \wedge$ noresolve-s$(y, \pi \lessdot x, q'))$
  |  ./$q'$ -> return noresolve-s$(x, \pi, q')$
  | ../$q'$ ->
    match $\pi$
    | $\varepsilon$ -> return noresolve-s$(x, \varepsilon, q')$
    | $\pi' \lessdot y$ -> return noresolve-s$(y, \pi', q')$

function noresolve($r$ : variable, cwd : path, $p$ : path)
    : $\Sigma_1$-formula
  match $p$
  | abs($q$) -> return noresolve-s($r, \varepsilon,$     $q$)
  | rel($q$) -> return noresolve-s($r, \varepsilon,$ cwd/$q$)
```

Figure 5.20: Function noresolve, threaded

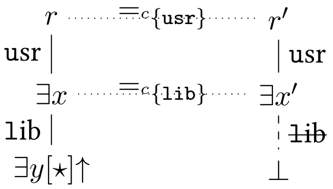


Figure 5.21: Specification of success cases for `rmdir /usr/lib`

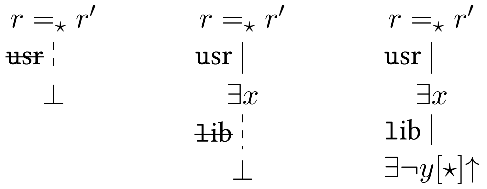


Figure 5.22: Specification of error cases for `rmdir /usr/lib`

```
function ifresolve-s(x : variable, π : variable stack, q : path,
      F : variable -> Σ1-formula) : Σ1-formula
  match q
  |    /   -> return F(x)
  |   f/q' -> return ¬x[f]↑ → ∃y · (x[f]y ∧ ifresolve-s(y, π ‹ x, q', F))
  |   ./q' -> return ifresolve-s(x, π, q', F)
  |  ../q' ->
    match π
    | ε -> return ifresolve-s(x, ε, q', F)
    | π' ‹ y -> return ifresolve-s(y, π', q', F)

function ifresolve(r : variable, cwd : path, p : path,
      F : variable -> Σ1-formula) : Σ1-formula
  match p
  | abs(q) -> return ifresolve-s(r, ε,     q, F)
  | rel(q) -> return ifresolve-s(r, ε, cwd/q, F)
```

Figure 5.23: Function ifresolve

We can remove disjunctions from the error cases, not only by using the new form of noresolve, but also by merging the preconditions of the error cases together, as shown in Formula 5.10.

$$\phi^{(p)}_{2-3}(r) \;=\; \exists x, y, z \cdot (\neg r[\texttt{usr}]\uparrow \rightarrow (r[\texttt{usr}]x \wedge (\neg x[\texttt{lib}]\uparrow \rightarrow (x[\texttt{lib}]y \wedge \neg y[\star]\uparrow)))) \tag{5.10}$$

Which can be read as: if the path /usr/lib resolves, then it points to something that is not empty.[12] This suggests the introduction of a macro ifresolve that would take as argument the input root, the current working directory, the path to resolve, and a function taking a variable as input and returning a threaded constraint on this variable. The definition of such a macro can be found in Figure 5.23. We can then give the precondition of the error cases of a generic rmdir q/f in Formula 5.11.

$$\phi^{(p)}_{2-3}(r) \;=\; \texttt{ifresolve}(r, \texttt{cwd}, \texttt{p}, (\texttt{fun}\ z\ \texttt{->}\ \neg z[\star]\uparrow)) \tag{5.11}$$

Moreover, we can simply rewrite $\texttt{noresolve}(r, \texttt{cwd}, \texttt{p})$ as $\texttt{ifresolve}(r, \texttt{cwd}, \texttt{p}, \bot)$.

Let us now show how to reason with such threaded constraints. We add the rule T-GuardEntailed that describes how to *activate* a thread:

$$\textsc{T-GuardEntailed} \qquad c \wedge (l \rightarrow t) \wedge \tau \quad \Rightarrow \quad c \wedge t \wedge \tau \qquad (c \models l)$$

This rule takes a constraint c and a thread $l \rightarrow t$ and *activates* the thread – that is, adds t to c – when c entails its guard l. If c does not entail l, the thread is said to be *inactive*. These two notions are defined in Definitions 5.11 and 5.12.

Definition 5.11 (Activation of a Thread)**.** In a threaded constraint $c \wedge (l \rightarrow t) \wedge \tau$ where c is a constraint, τ is a list of threads and $c \models l$, *activating the thread* $l \rightarrow t$ means returning $c \wedge t \wedge \tau$. □

Definition 5.12 (Inactive Thread)**.** In a threaded constraint $c \wedge (l \rightarrow t) \wedge \tau$ where c is a constraint and τ is a list of threads, the thread $l \rightarrow t$ is *inactive* if c does not entail l. □

Testing whether $c \models l$ is easy because, by Lemma 5.8, it is equivalent to test that $c \wedge \neg l$ is unsatifiable. Here, $c \wedge \neg l$ is a constraint and we can test their satisfiability using transform-2. In the case where c is

[12] If we handle file types, this would be: if the path /usr/lib resolves, then if it is a directory, then it is empty.

$$
\begin{array}{llcl}
(0) & & & r[f]z \land z[\star]\uparrow \\
 & & & \land (\neg r[f]\uparrow \rightarrow (r[f]x \land (\neg x[g]\uparrow \rightarrow (x[g]y \land y[h]\uparrow)))) \\
(1) & \text{T-GuardEntailed} & \Rightarrow & r[f]z \land z[\star]\uparrow \land r[f]x \\
 & & & \land (\neg x[g]\uparrow \rightarrow (x[g]y \land y[h]\uparrow)) \\
(2) & (\mathcal{R}_2)^\star & \Rightarrow & r[f]x \land x[\star]\uparrow \land z =_\star x \\
 & & & \land (\neg x[g]\uparrow \rightarrow (x[g]y \land y[h]\uparrow))
\end{array}
$$

Figure 5.24: Transformation of Formula 5.12

already known to be irreducible with respect to $\mathcal{R}_2$, we can use the incrementality of `add-transform-2` to make this test of entailment almost trivial.

Let us consider Formula 5.12 as an example. It combines a thread similar to that of Formula 5.6 in conjunction to the main constraint $r[f]z \land z[\star]\uparrow$.

$$
\begin{array}{l}
r[f]z \land z[\star]\uparrow \\
\quad \land (\neg r[f]\uparrow \rightarrow (r[f]x \\
\qquad\qquad\qquad\quad \land (\neg x[g]\uparrow \rightarrow (x[g]y \land y[h]\uparrow))))
\end{array}
\tag{5.12}
$$

Let us now describe the steps of transformation from Formula 5.12 using $\mathcal{R}_2$ and our new rule T-GuardEntailed. Intermediary steps can be found in Figure 5.24.

(0) Start from Formula 5.12. Notice that the main constraint is irreducible with respect to $\mathcal{R}_2$ and that it entails the guard $\neg r[f]\uparrow$. Indeed, $r[f]z \land z[\star]\uparrow \land \neg\neg r[f]\uparrow$ leads to a clash between $r[f]z$ and $r[f]\uparrow$.

(1) Rewrite the threaded constraint using T-GuardEntailed. This activates the thread and adds $r[f]x \land (\neg x[g]\uparrow \rightarrow (x[f]y \land y[h]\uparrow))$ to the main constraint. The left-hand side of this conjunction joins the main constraint while the right-hand side joins the list of threads.

(2) Rewrite the main constraint using $\mathcal{R}_2$. The steps are not detailed. Notice now that the main constraint is irreducible with respect to $\mathcal{R}_2$ and that it does not entail the guard $\neg x[g]\uparrow$: the thread is inactive.[13]

In this example, the obtained formula is satisfiable. Unfortunately, we cannot always use $\mathcal{R}_2$ and T-GuardEntailed to decide satisfiability easily. This is the topic of the next subsection.

5.3.2 Properties of Threaded Constraints

Let us first state in Lemma 5.9 that these two newly added rules behave nicely.

Lemma 5.9 (T-GuardEntailed is an Equivalence)**.** *T-GuardEntailed performs an equivalence, that is if $t \Rightarrow t'$ via T-GuardEntailed, then $\models_{\mathcal{FT}} \tilde{\forall} \cdot (t \leftrightarrow t')$.*

Now comes the question of deciding the satisfiability of threaded constraints. The satisfiability of a threaded constraint with just one thread comes easily from the side-condition of the rule T-GuardEntailed.

Lemma 5.10 (Satisfiability of Threaded Constraints With One Inactive Thread)**.** *A threaded constraint with only one inactive thread is satisfiable. That is, for all threaded constraint $t = c \land (l \rightarrow t')$ where c is a constraint, if $l \rightarrow t'$ is inactive in t, then $\models_{\mathcal{FT}} \tilde{\exists} \cdot t$.*

[13] We can in fact see here that the main constraint *disentails* the guard, that is entails its negation $x[g]\uparrow$. This means that the thread can never be activated and could be discarded from the threaded constraint.

Proof. If $c \wedge (l \rightarrow t')$ is irreducible with respect to T-GuardEntailed, then it has to be because the side-condition is not respect and, therefore, c does not entail l. By Lemma 5.8, $c \wedge \neg l$ is satisfiable. And $c \wedge \neg l$ implies $c \wedge (l \rightarrow t')$. □

As soon as one considers a constraint with several inactive threads, the satisfiability is not guaranteed. Consider Formula 5.13 as an example of this. [14]

$$\top \wedge (x[f]\uparrow \rightarrow \bot) \wedge (\neg x[f]\uparrow \rightarrow \bot) \tag{5.13}$$

Formula 5.13 is clearly unsatifiable although the two threads are inactive (because $\top$ does not entail $x[f]\uparrow$ nor $\neg x[f]\uparrow$). In the context of the CoLiS project, the huge majority of our threads have a guard that is negated absence atom on a singleton. Sadly, even for such limited guards, the inactivity of threads does not imply that the whole threaded constraint is satisfiable. Consider Formula 5.14 as an example of this.

$$\neg x[f,g]\uparrow \wedge (\neg x[f]\uparrow \rightarrow \bot) \wedge (\neg x[g]\uparrow \rightarrow \bot) \tag{5.14}$$

Formula 5.14 is also unsatifiable although its two threads are inactive (because $\neg x[f,g]\uparrow$ does not entail $\neg x[f]\uparrow$ nor $\neg x[g]\uparrow$). If we restrict threaded constraints to only use negated absence atoms in guards and to not use any negated atom in the main constraint, then we finally have the result that we want. This is stated in Lemma 5.11.

Lemma 5.11 (Satisfiability of Threaded Constraints With Only Positive Literals in Main Constraint and Only Negated Absence Atoms in Guards)**.** *For all threaded constraint $t = c \wedge \bigwedge_i (\neg x_i[F_i]\uparrow \rightarrow t_i)$, where c is a constraint with only positive literals and for all i, x_i is a variable, F_i is a set of features and t_i is a threaded constraint, if for all i, the thread $\neg x_i[F_i]\uparrow \rightarrow t_i$ is inactive, then t is satisfiable.*

Proof. In fact, when there are no negated atoms in a satisfiable constraint, then it enjoys a *minimal* valuation, that is a valuation that is constructed from the feature atoms only, not adding any other feature. For each variable x in such a constraint c enjoying such a minimal valuation ρ, $\texttt{dom}(\rho(x)) = \{f \mid \exists y \cdot x[f]y \in c\}$.[15]

If such a constraint c happens to not entail several negated absence atoms of the form $\neg x_i[F_i]\uparrow$, it means that $c \wedge x_i[F_i]\uparrow$ is satisfiable. One can in fact notice that it is satisfiable by the minimal valuation that satisfies c. The same valuation therefore satisfies all the $c \wedge x_i[F_i]\uparrow$, which means that it also satisfies $c \wedge \bigwedge_i x_i[F_i]\uparrow$ and therefore $c \wedge \bigwedge_i (\neg x_i[F_i]\uparrow \rightarrow t_i)$. □

Within the CoLiS project, we find ourselves in this situation most of the time, although not always. Of course, there is always the possibility to unfold the threads by replacing $l \rightarrow t$ by the disjunction of $\neg l$ and the unfolding of t. However, for n threads, this creates a DNF of at least 2^n constraints[16]. We will show in Section 5.3.3 how this unfolding is done. We will see that, in practice, this is an efficient way of doing things.

In fact, the problem of deciding the satisfiability of threaded constraints is actually quite easily shown to be NP-hard, even if one allows only negated absence atoms in the guards, as it is the case in the CoLiS project. In fact, it is also NP-hard if we only allow threads of the restricted form $\neg x[f]\uparrow \rightarrow x[f]y$ using which we can still write `noresolve`. These three facts are stated in Lemma 5.12.

[14] Of course, such a formula might seem silly. It is susceptible to appear in real-world situation, although not with $\bot$ directly but with, in each thread, a constraint that is inconsistent with the main constraint.

[15] This shows in the proof of Theorem 5.1 by the fact that, where there are no negated atoms, then $\mathcal{D}$ is empty.

[16] And potentially much more depending on the complexity of the threads in this threaded constraint.

Lemma 5.12 (NP-Hardness of Satisfiability of Threaded Constraints)**.** *The problem of deciding the satisfiability of threaded constraints is NP-hard*

1. *in general,*
2. *on threaded constraints where guards only contain negated absence atoms,*
3. *on threaded constraints where threads are always of the form $\neg x[f]\uparrow \rightarrow x[f]y$ for some x, y and f.*

Proof. Point 1 is of course a consequence of Point 2 or Point 3. For the proof of the last two points, we first introduce an intermediary satisfiability problem LIMSAT in Definition 5.13 and show that it is NP-complete in Lemma 5.13. We then come back to threaded constraints and show that we can encode any LIMSAT instance in a threaded constraint. □

Definition 5.13 (LIMSAT)**.** LIMSAT[17] is a Boolean satisfiability problem that focuses on Boolean formulas under the form of a conjunction of disjunctions. Each disjunction can be of the form:

$$\begin{array}{rcl} \bar{y_1} \vee \cdots \vee \bar{y_n} & — & \text{only negative variables} \\ x \ \vee \bar{y_1} \vee \cdots \vee \bar{y_n} & — & \text{one positive variable and then only negative variables} \\ x_1 \vee x_2 \qquad\qquad & — & \text{two positive variables} \end{array}$$

□

Lemma 5.13 (NP-Completeness of LIMSAT)**.** *LIMSAT is NP-complete.*

Proof. Any instance of LIMSAT is obviously an instance of SAT [Cook 1971; Garey & Johnson 1979]. Take now an instance of 3-SAT [Cook 1971; Garey & Johnson 1979] and consider the disjunctions that do not belong to LIMSAT directly. They are the 3-disjunctions containing at least two positive variables. Name those x_1 and x_2, name the third literal l, take two fresh variables a and b and apply then the following transformation everywhere in the formula:

$$x_1 \vee x_2 \vee l \quad \rightsquigarrow \quad (x_1 \vee a) \wedge (x_2 \vee b) \wedge (l \vee \bar{a} \vee \bar{b})$$

The transformation is polynomial (it multiplies the size of the formula by at most 3) and the initial and resulting formulas are equisatisfiable. Moreover, it is a LIMSAT formula.[18]Since 3-SAT is NP-complete, then so is LIMSAT. The technique in this proof comes directly from the technique used to prove NP-completeness of 3-SAT. □

Proof of Lemma 5.12, Point 2. Take any LIMSAT formula. Using its boolean variables as features, and only one tree variable r, transform all disjunctions from the LIMSAT problem using the following:

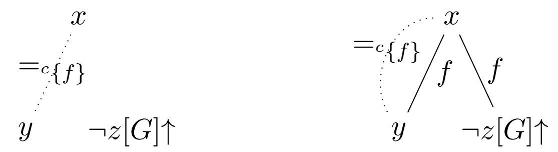
$$\begin{array}{lcr} \bar{y_1} \vee \cdots \vee \bar{y_n} & \rightsquigarrow & \neg r[y_1, \dots, y_n]\uparrow \\ x \ \vee \bar{y_1} \vee \cdots \vee \bar{y_n} & \rightsquigarrow & \neg r[x\ ]\uparrow \rightarrow \neg r[y_1, \dots, y_n]\uparrow \\ x_1 \vee x_2 & \rightsquigarrow & \neg r[x_1]\uparrow \rightarrow \ \ r[x_2]\uparrow \qquad\quad \end{array}$$

The transformation is polynomial and yields indeed a threaded constraint where threads use only absence atoms. The threaded constraint is satisfiable in $\mathcal{FT}$ if and only if the LIMSAT instance is satisfiable. □

Proof of Lemma 5.12, Point 3. It is to be noted that the same result can be obtained even when restraining the form of threaded constraints to $\neg r[x]\uparrow \rightarrow r[x]v$, form which is absolutely required in CoLiS.

[17] We could not find any reference to the LIMSAT problem in literature. This is however just “yet another boolean satisfiability problem”.

[18] It is also a formula of 3-LIMSAT, where 3-LIMSAT is LIMSAT except all constraints have at most three literals, thus proving that 3-LIMSAT is also NP-complete. Not that this is a useful comment though.



(a) Left-hand side (b) Right-hand side

Figure 5.25: Formula 5.17

Consider Formula 5.15 and assume $f \notin G$[19].

$$\neg x[f]\uparrow \rightarrow \neg x[G]\uparrow \tag{5.15}$$

It is equivalent to Formula 5.16 and its unfolded version Formula 5.17. A graphical representation of the latter can be found in Figure 5.25.

$$\exists y, z \cdot (\neg x[f]\uparrow \rightarrow x[f]y) \wedge (\neg x[f]\uparrow \rightarrow x[f]z) \wedge x =_{c\{f\}} y \wedge \neg z[G]\uparrow \tag{5.16}$$

$$\begin{aligned} &\exists y, z \cdot (x[f]\uparrow \wedge x =_{c\{f\}} y \wedge \neg z[G]\uparrow) \\ \vee\ &\exists y, z \cdot (x[f]y \wedge x[f]z \wedge x =_{c\{f\}} y \wedge \neg z[G]\uparrow) \end{aligned} \tag{5.17}$$

The idea behind such a formula is the following: we want to only allow feature atoms in the right-hand-side of a thread. The two feature atoms on the same variable and feature cover hide then an equality which can later be discovered through D-FEATS. We can thus use that to hide the literal we need on the variable z and, if the thread activates, then the literal will apply to the other variable too. However, we cannot directly link z and x that way because the feature atoms must come from x, and cycles are forbidden. We can use a similarity atom to link x and y on anything but f (otherwise, there would be a cycle). Since we assumed that $f \notin G$, this last bit is not a limitation. □

This result does not make threaded constraints useless. It simply means that, in the general case, we will have no other choice but to unfold the threaded constraint into an exponentially bigger DNF. This means that these threaded constraints bring no improvement to the worst-case situation. There are two important points to be noted however. Firstly, as mentioned previously, in the CoLiS project, we often find ourselves in the situation where all guards are negated absence atoms and there are no negated atoms in the main constraint. In this case, by Lemma 5.11, the satisfiability is guaranteed. Secondly, and more importantly, the unfolding does not have to be done in one go. It is possible to test all the possible unfoldings in a lazy manner also. It is then sufficient to stop as soon as we have found a satisfiable unfolding. In practice, and in the CoLiS project in particular, the first unfolding (consisting of the negation of all the guards) is satisfiable. This will be described in Section 5.3.3.

5.3.3 Implementation of Threaded Constraints

In this subsection, let us implement a solver for threaded constraints. It is a fairly simple task: we use the solvers for constraints on the main constraint and once we reach an irreducible form, we check whether it entails the guards. The threads whose guard is entailed are activated and their threaded constraint merged with the main constraint. We keep doing that until all the remaining threads are inactive.

[19] This is not a limitation as we can preprocess the LIMSAT instance to remove any disjunctive clause that contains both a variable and its negation.

```
1  function add-transform-2-main-constraint
2      ($c \land \tau$  : threaded-constraint,
3       $c' \land \tau'$ : threaded-constraint) : threaded-constraint or $\bot$
4    match add-transform-2($c$, $c'$)
5    | $\bot$ -> return $\bot$
6    | $c''$ -> return $c'' \land \tau \land \tau'$
7
8  function entails($c \land \tau$ : threaded-constraint, $l$ : literal) : boolean
9    match add-transform-2($\neg l$, $c$)
10   | $\bot$ -> return true
11   | _ -> return false
12
```

Figure 5.26: Helper functions for activate-threads

Let us start in Figure 5.26 by defining two helpers: add-transform-2-main-constraint (Line 1) and entails (Line 8). The former is a simple wrapper around add-transform-2: it takes two threaded constraints, applies add-transform-2 on their main constraints and merges their threads.

The latter takes a threaded constraint $c \land \tau$ and a literal l and tests whether c entails l. This is done by checking whether $c \land \neg l$ is unsatisfiable. Note that this test of entailment is correct because if $c \models l$ then $c \land \tau \models l$. It is not complete in the sense that it is possible to have a threaded constraint t and a literal l such that $t \models l$ but entails(t, l) = false. This comes from the fact that we only consider threads once they have been activated. This is not a problem as we are already incomplete as discussed in the previous subsection.

Consider now Figure 5.27. It defines the main function of this solver: activate-threads. This function takes a constraint and a list of threads and tries to activate all the threads. It returns a threaded constraint in which all threads are inactive. It works as follows:

- activate-threads (Line 21) is in fact a thin wrapper around activate-threads-loop (Line 1). The loop is started with a constraint, a list of threads and false. The constraint is generalised to a threaded constraint t so that it can contain threads that have already been processed[20]. The list of threads τ is the same. The boolean activated indicates whether a thread has been activated on this round of the loop. If so, it is necessary to do one more round because the activation can change the main constraint and make some previously inactive threads active.

- activate-threads-loop starts by pattern matching on the threads τ (Line 5). If there are no threads (Line 6), we are at the end of a round of the loop. If one thread has been activated, we start again recursively (Line 9). Otherwise, we are done (Line 11): the main constraint of t is irreducible and all its threads are inactive.

- If there are threads (Line 13), we consider one of them, $l \rightarrow t'$, and name the others τ'. We check whether t entails the guard l (Line 14).

- If it does, we activate the thread (Line 15) and add t' to t. The result can be $\bot$ if we detect an unsatisfiability. In that case, we return $\bot$ immediately (Line 16). Otherwise, we continue looping through the threads after having marked the activated boolean as true (Line 17).

- If it does not (Line 18), the thread is inactive. We add it to t and loop on the other threads (Line 19).

[20] And detected as inactive, for instance.

```
function activate-threads-loop
    ($t$ : threaded-constraint, $\tau$ : list of threads
     activated : boolean) : threaded-constraint or $\bot$

  match $\tau$
  | $\top$ ->
    if activated
      let $c \wedge \tau'$ = $t$
      return activate-threads-loop($c$, $\tau'$, false)
    else
      return $t$

  | $(l \rightarrow t') \wedge \tau'$ ->
    if entails($t$, $l$)
      match add-transform-2-main-constraint($t'$, $t$)
      | $\bot$ -> return $\bot$
      | $t''$ -> return activate-threads-loop($t''$, $\tau'$, true)
    else
      return activate-threads-loop($t \wedge (l \rightarrow t')$, $\tau'$, activated)

function activate-threads
    ($c$ : constraint, $\tau$ : list of threads)
     : threaded-constraint or $\bot$
  return activate-threads-loop($c$, $\tau$, false)
```

Figure 5.27: Function activate-threads, key element to transform-2-threaded and add-transform-2-threaded

```
function add-transform-2-threaded
    ($c \wedge \tau$  : threaded-constraint,
     $c' \wedge \tau'$ : threaded-constraint) : threaded-constraint or $\bot$
  match add-transform-2($c$, $c'$)
  | $\bot$ -> return $\bot$
  | $c''$ -> return activate-threads($c''$, $\tau \wedge \tau'$)

function transform-2-threaded ($t$ : threaded-constraint)
    : threaded-constraint or $\bot$
  return add-transform-2-threaded($t$, $\top$)
```

Figure 5.28: Functions transform-2-threaded and add-transform-2-threaded

```
 1 function add-check-sat-threaded
 2     ($c \wedge \tau$  : threaded constraint,
 3      $c' \wedge \tau'$ : threaded constraint) : boolean
 4   match add-transform-2($c$, $c'$)
 5   | $\bot$ -> return false
 6   | $c''$ ->
 7     match $\tau \wedge \tau'$
 8     | $\top$ -> return true
 9     | $(l \rightarrow t) \wedge \tau''$ ->
10       match add-transform-2($\neg l$, $c''$)
11       | $c'''$ ->
12         if check-sat-threaded($c''' \wedge \tau''$)
13           return true
14         else
15           return add-check-sat-threaded($t$, $c'' \wedge \tau''$)
16       | $\bot$ -> return add-check-sat-threaded($t$, $c'' \wedge \tau''$)
17
18 function check-sat-threaded($t$ : threaded-constraint) : boolean
19   return add-check-sat-threaded($t$, $\top$)
```

Figure 5.29: Function check-sat-threaded

Consider finally Figure 5.28. It gives the implementation for the two functions transform-2-threaded and add-transform-2-threaded. These are simple wrappers around activate-threads. add-transform-2-threaded takes two threaded constraint t and t', where t' has its main constraint irreducible, and computes $t \wedge t'$. This is simply done by calling add-transform-2 on the main constraints of t and t' and by then trying to activate all the threads of both t and t'. transform-2-threaded is a simple wrapper around add-transform-2-threaded where the second constraint is $\top$.

These functions make a heavy use of the incrementality of $\mathcal{R}_2$ and $\mathcal{R}_2^\bullet$, provided by add-transform-2: In activate-threads-loop, we keep the main constraint of the threaded constraint irreducible which allows to use add-transform-2 for the activation of threads and to therefore only compute on the patterns that are added by the activated thread.

The functions transform-2-threaded and add-transform-2-threaded show how easy it is to compute on threaded constraint once we have an incremental solver for constraints. These two functions return either $\bot$ or a threaded constraint whose main constraint is irreducible and whose threads are inactive. As discussed in the previous subsection, this is not sufficient to guarantee that the whole threaded constraint is satisfiable. The only way to know for sure that is to unfold all the implications as disjunctions and to check that the resulting DNF is indeed satisfiable. We are however going to show that this can be done in a lazy manner.

Consider Figure 5.29. It defines a function check-sat-threaded which takes a threaded constraint whose main constraint is irreducible and checks whether it is satisfiable. This is done lazily by enumerating all possible unfolding but stopping as soon as a satisfiable one has been found. Of course, the complexity is still exponential in the number of threads of the constraint in the worst case. check-sat-threaded works as follows:

- check-sat-threaded (Line 18) is in fact a wrapper around add-check-sat-threaded (Line 1). This function takes two threaded constraints $c \wedge \tau$ and $c' \wedge \tau'$, where c' is an irreducible constraint, and checks whether $c \wedge c' \wedge \tau \wedge \tau'$ is satisfiable. This function is necessary for technical reasons, in order to ensure the invariant that c' is always irreducible and that, therefore, we can use the efficient and incremental solver add-transform-2.

- add-check-sat-threaded starts by computing the irreducible form of the conjunction of two main constraints $c \wedge c'$ (Line 4). If the result is $\bot$, then we can return false right away (Line 5).
- If the result is not false, it is a satisfiable constraint c''. We then consider the given threads, $\tau \wedge \tau'$ (Line 7). If there are none, then $c \wedge c' \wedge \tau \wedge \tau'$ is equivalent to c'' which is satisfiable: we can return true (Line 8).
- If there are threads, we pick one, $l \rightarrow t$ and we name the others τ'' (Line 9). We need to check whether at least one of the unfoldings of $l \rightarrow t$ is satisfiable, that is if one of $c'' \wedge \neg l \wedge \tau''$ or $c'' \wedge t \wedge \tau''$ is satisfiable. We start by computing the irreducible form of $c'' \wedge \neg l$ (Line 10).
- If the computation returns a constraint, we name it c'''. c''' is irreducible and therefore satisfiable. We proceed by checking whether $c''' \wedge \tau''$ is satisfiable (Line 12). If the result is true, we return right away (Line 13).
- If this check fails or if the irreducible form of $c'' \wedge \neg l$ is $\bot$, then we can give up on this unfolding. We thus proceed to check whether $c'' \wedge t \wedge \tau''$ is satisfiable and this is done with a simple recursive call (Lines 15 and 16).

5.3.4 Discussions

The three functions transform-2-threaded, add-transform-2-threaded and check-sat-threaded give us a fine-grained control over computation on threaded constraints. The two former can be seen as fast but imprecise while the latter is slow and precise. In CoLiS, that means we use the fast ones as a backend for symbolic execution engine (see Section 7.2) and the precise one only at the very end to remove unsatisfiable traces whose unsatisfiability was not detected on the fly. This brings a huge improvement. Moreover, since we only use very specific threads, we will only seldom keep an unsatisfiable threaded constraint in the engine. Moreover, in the huge majority of cases, the very first unfolding will be satisfiable, making the final check very fast.

In practice, the symbolic engine from CoLiS includes a very ad hoc implementation of these threaded constraints as it only supports threads of the form $\neg x[f]\uparrow \rightarrow x[f]y$, which we call *maybe atoms*. Since they are so specific, we can include the handling of maybe atoms directly in the solver and not externally as done in this section for the general case. These maybe atoms are sufficient to remove some disjunctions of noresolve and ifresolve – that is the same disjunctions as we have removed in Section 5.3.1. They are however not general enough to remove disjunctions coming from file kinds. Indeed, additional cases of non-resolution come from the fact that paths may exist but not be directories, which we cannot express only with our maybe atoms and without disjunctions.

We ran our analysis on the 12,592 packages of Debian that contain maintainer scripts for a total of 28,814. We ran it three times with the same version of colis-batch [23, Commit 66704e] and only slightly different versions of colis-language.

1. The first run includes colis-language [11, Commit 74e73d5] in the version used in all the other presented material of this thesis. This includes the handling of maybe atoms in the solver and their use in noresolve and ifresolve.
2. The second run includes colis-language [11, Commit 74e73d5] from which we have rewritten noresolve and ifresolve to not use maybe atoms. They are still present in the solver but no use is made of them.

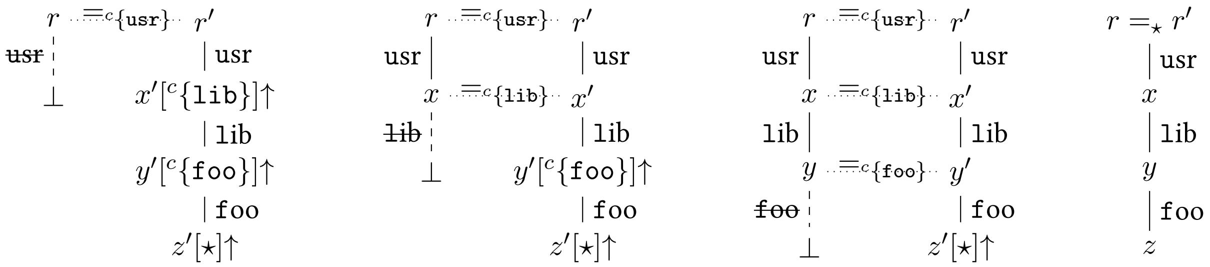


Figure 5.19: Specification cases of mkdir -p /usr/lib/foo

3. The third run includes colis-language [11, Commit 74e73d5] from which we have rewritten noresolve and ifresolve and eliminated the handling of maybe atoms in the solver.

These three runs share the same configuration of colis-batch. They are ran on a machine equipped with 40 hyperthreaded Intel Xeon CPU @ 2.20GHz, and 750GB of RAM. To obtain a reasonable execution time, we limit the processing of one script to 60 seconds and 8GB of RAM. This means that, in order to compare the performances of the three runs, we do not only need to consider the time spent on the analysis but also the number of scripts that reached a timeout.

The two last runs present no significant difference, which means that adding support for threaded constraint in the solver does not include a slowdown on the regular constraints. The first run terminates in less than half an hour when the two other take more than five hours. The implementation of threaded constraints therefore brings us a speedup of at least an order of magnitude.

Let us go back to threaded constraints as described in this section. One could imagine some extra rules about threads. The first one is a rule of cleanup, T-GuardDisentailed, which discards a threads when we know that the main constraint disentails its guard.

$$\text{T-GuardDisentailed} \qquad c \wedge (l \to t) \wedge \tau \quad \Rightarrow \quad c \wedge \tau \qquad (c \models \neg l)$$

This rule has been omitted from the presentation for consistency – as we have never included optional rules that would only cleanup so far – but it really easy to add to the implementation of transform-2-threaded and add-transform-2-threaded.

One can then imagine two rules about the entailment or disentailment of the threaded constraint in a thread – that is the t in $l \to t$. These are the rules T-ThreadedEntailed and T-ThreadedDisentailed.

$$\begin{array}{llcll}
\text{T-ThreadedEntailed} & c \wedge (l \to t) \wedge \tau & \Rightarrow & c \wedge \tau & (c \models t) \\
\text{T-ThreadedDisentailed} & c \wedge (l \to t) \wedge \tau & \Rightarrow & c \wedge \neg l \wedge \tau & (c \models \neg t)
\end{array}$$

These two rules are very similar to T-GuardEntailed and T-GuardDisentailed. They introduce however a lot of difficulties as threaded constraints are rich objects – contrary to the guards that are limited to literals – on which it is complicated to test entailment.

Finally, let us come back on the example of mkdir -p /usr/lib/foo mentioned in the introduction. The specification of mkdir -p /usr/lib/foo is presented in Figure 5.19, restated here for convenience. Such a specification can in fact be written as one single specification case containing only one threaded constraint. Consider Formula 5.18

$$\begin{aligned}
\exists x, y, x', y', z' \cdot (&r'[\texttt{usr}]x' \wedge x'[\texttt{lib}]y' \wedge y'[\texttt{foo}]z' \wedge r =_{{}^c\{\texttt{usr}\}} r' \\
&\wedge (\ \ r[\texttt{usr}]\uparrow \rightarrow (x'[{}^c\{\texttt{lib}\}]\uparrow \wedge y'[{}^c\{\texttt{foo}\}]\uparrow \wedge z'[\star]\uparrow)) \\
&\wedge (\neg r[\texttt{usr}]\uparrow \rightarrow (r[\texttt{usr}]x \wedge x =_{{}^c\{\texttt{lib}\}} x' \\
&\qquad\qquad \wedge (\ \ x[\texttt{lib}]\uparrow \rightarrow (y'[{}^c\{\texttt{foo}\}]\uparrow \wedge z'[\star]\uparrow)) \\
&\qquad\qquad \wedge (\neg x[\texttt{lib}]\uparrow \rightarrow (x[\texttt{lib}]y \wedge y =_{{}^c\{\texttt{foo}\}} y' \\
&\qquad\qquad\qquad\qquad \wedge (\ \ y[\texttt{foo}]\uparrow \rightarrow z'[\star]\uparrow) \\
&\qquad\qquad\qquad\qquad \wedge (\neg y[\texttt{foo}]\uparrow \rightarrow y[\texttt{foo}]z'))))))
\end{aligned} \tag{5.18}$$

It works as follows.

- The main constraint, $r'[\texttt{usr}]x' \wedge x'[\texttt{lib}]y' \wedge y'[\texttt{foo}]z' \wedge r =_{{}^c\{\texttt{usr}\}} r'$ contains the part of the formula that is common to all the specification cases. There are then two threads, depending on whether `usr` exists in r.

- If `usr` does not exist in r, that is if the main constraint entails $r[\texttt{usr}]\uparrow$, then we are in the first case of Figure 5.19 and we need to add the three absence atoms on x', y' and z'.

- If `usr` does exist in r, that is if the main constraint entails $\neg r[\texttt{usr}]\uparrow$, then we are in the three other cases of Figure 5.19. We can then add $r[\texttt{usr}]x \wedge x =_{{}^c\{\texttt{lib}\}} x'$ to the main constraint and ready two new guards depending on whether `lib` exists in x.

- If `lib` does not exist in x, that is if the main constraint entails $x[\texttt{lib}]\uparrow$, then we are in the second case of Figure 5.19 and we need to add the two absence atoms on y' and z'.

- If `lib` does exist in x, that is if the main constraint entails $\neg x[\texttt{lib}]\uparrow$, then we are in two last cases of Figure 5.19. We can then add $x[\texttt{lib}]y \wedge y =_{{}^c\{\texttt{foo}\}} y'$ to the main constraint and ready two new guards depending on whether `foo` exists in y.

- If `foo` does not exist in y, that is if the main constraint entails $y[\texttt{foo}]\uparrow$, then we are in the third case of Figure 5.19 and we need to add the absence atom on z'.

- Finally, if `foo` does exist in y, that is if the main constraint entails $\neg y[\texttt{foo}]\uparrow$, then we are in the last cases of Figure 5.19. Of course, we have built two chains of feature and similarity atoms starting from r and r' and following `/usr/lib/foo`. We however need to ensure that `/usr/lib/foo` exists in r and r' and that r and r' are equal. We can do this by adding either $y[\texttt{foo}]z \wedge r =_\star r'$ or simply $y[\texttt{foo}]z'$ to the main constraint.

The same process of course works for paths of any length. In any case, it encodes the specification of `mkdir -p` p as one specification case. This is an example of the expressivity provided by threaded constraints.

Appendix 5.A Proof of Theorem 5.1

Introduction. This proof is quite similar to that of Theorem 4.1, except that the system $\mathcal{R}_2$ does not give as strong guarantees on its irreducible constraints as $\mathcal{R}_1$. This is in particular because it does not have access to the splitting rules, as they introduce disjunctions. The absence of these rules is at the heart of the fact that $\mathcal{R}_2$ does not enjoy the property of garbage collection. Consider for instance Formula 5.2, restated here.

$$\exists x \cdot (\neg x[\star]\uparrow \wedge x =_G y \wedge x =_{{}^cG} z) \tag{5.2}$$

It expresses the fact that the local variable x must have a feature and must be similar to y in G and to z in ${}^cG$. The information that this carries on the global variables y and z is that "either y has a feature in G or z has a feature in ${}^cG$". This fact is easily expressible but requires disjunction – $\neg y[G]\uparrow \vee \neg z[{}^cG]\uparrow$. The system $\mathcal{R}_1$, since it is allowed to introduce disjunctions, will be able to deduce that fact and to eliminate the variable x. The system $\mathcal{R}_2$, since it cannot introduce disjunctions, will never be able to remove x.

Let us dig a bit this example and show how both $\mathcal{R}_1$ and $\mathcal{R}_2$ handle such a constraint. Consider the constraint in Formula 5.2. We will put it in irreducible form with respect to both $\mathcal{R}_1$ and $\mathcal{R}_2$ and assume we have a valuation μ satisfying the global part of the resulting constraint/s. We will then imagine that we are trying to define ρ satisfying the whole constraint.

- The constraint of Formula 5.2 is not irreducible with respect to $\mathcal{R}_1$. This is because of the splitting rule S-NAbs-Sim which requires us to split $\neg x[\star]\uparrow$ into $\neg x[G]\uparrow \vee \neg x[{}^cG]\uparrow$. In each of the obtained constraint, P-NAbs-Sim requires us to propagate the negated absence atoms to y and z respectively. We obtain two constraints that are irreducible with respect to $\mathcal{R}_1$:
$$\exists x \cdot (\neg x[G]\uparrow \wedge x =_G y \wedge x =_{{}^cG} z) \wedge \neg y[G]\uparrow$$
and
$$\exists x \cdot (\neg x[{}^cG]\uparrow \wedge x =_G y \wedge x =_{{}^cG} z) \wedge \neg z[{}^cG]\uparrow$$
If a valuation μ satisfies the global parts of the former (resp. the latter), then it satisfies $\neg y[G]\uparrow$ (resp. $\neg z[{}^cG]\uparrow$), which we can leverage to show that if ρ satisfies $x =_G y$ (resp. $x =_{{}^cG} z$), then it satisfies $\neg x[\star]\uparrow$ automatically. This reasoning works on *any* valuation μ and therefore we can conclude that x has become irrelevant and that Formula 5.2 is equivalent to $\neg y[G]\uparrow \vee \neg z[{}^cG]\uparrow$.

- The constraint of Formula 5.2 is irreducible with respect to $\mathcal{R}_2$, however, because $\mathcal{R}_2$ does not contain such splitting rules. It is not true that any valuation μ that satisfies the global part of this constraint can be extended to ρ that satisfies the whole constraint. Indeed, if we take t the empty feature tree and $\mu = [y \mapsto t, z \mapsto t]$, the two similarity atoms impose that $\rho(x) =_G t$ and $\rho(x) =_{{}^cG} t$ and therefore that $\rho(x) = t$, which contradicts $\rho \models_{\mathcal{FT}} \neg x[\star]\uparrow$.

  If we have control on the valuation μ that we take for the global part, then we can ensure that there is $g \in G$ (or resp. $g \in {}^cG$) such that $g \in \mathtt{dom}(\mu(y))$ (resp. $g \in \mathtt{dom}(\mu(z))$). Any valuation μ that satisfies this will then be extendable to x. Since this reasoning works on *some* valuations μ, we cannot conclude that x has become irrelevant, but we can conclude that the formula is satisfiable.

In the general case, this means that the proof of Theorem 5.1 cannot be as simple as that of Theorem 4.1. In particular, we will not be able to build an induction around a simple lemma as Lemma 4.7. We will still prove Theorem 5.1 by induction on the variables of the constraint, but the induction hypothesis will have to be much stronger to compensate for the information that the splitting rules would otherwise provide.

Let us take now take c, a constraint that is irreducible with respect to $\mathcal{R}_2$. Note that, as any constraint, it is different from $\bot$ by definition. Let us show that c is satisfiable. In order to do that, we are going to build a valuation ρ that satisfies c, by induction on the variables of c. We will make sure to define "lower" variables before "higher" ones, that is we will make sure that, when there is $x[f]y \in c$, $\rho(y)$ is defined before $\rho(x)$.

Let us take $<$ a total order on the variables of c that respects $y < x$ if $x[f]y \in c$. Such an order exists, by non-applicability of C-Cycle. We are going to define ρ by induction on the variables in increasing order (for $<$). For any variable x, we use the notations:
$$c_{<x} = \{l \mid l \in c, \forall y \in \mathcal{V}(l), y < x\}$$
$$c_{\leq x} = \{l \mid l \in c, \forall y \in \mathcal{V}(l), y \leq x\}$$

to talk about the parts of the constraints containing only variables smaller than x.

Definition of $\mathcal{D}$. How can we ensure that we will never find ourselves in the situation of the example, where we consider the constraint of Formula 5.2 and have already defined $\mu = [y \mapsto t, z \mapsto t]$ which makes it impossible to define ρ, an extension of μ to x that satisfies the whole constraint?

Our solution is to pick a set of features that covers all the negated absence atoms and to ensure (in the induction hypothesis) that, for each variable, if possible, the valuation is defined on these features. In the aforementioned example, that means we would pick $f \in \star$ – that is any f – and require in the induction hypothesis that, if possible, $f \in \texttt{dom}(\mu(y))$ and $f \in \texttt{dom}(\mu(z))$. In this example, this is possible because there are no other constraints on either y and z. It will therefore be possible to choose $\rho(x)(f) = \rho(y)(f)$ or $\rho(x)(f) = \rho(z)(f)$ depending on whether $f \in G$ which will satisfy both $\neg x[\star]\uparrow$, $x =_G y$ and $x =_{^cG} z$.

In fact, we will use the same set of features to cover all the negated similarity atoms for similar reasons. Additionally, we will add a feature for each separated pair of variables in order to ensure that the values given to them are indeed different.

The set containing all these chosen features is named $\mathcal{D}$. It is defined as follows.

1. For each $\neg x[F]\uparrow \in c$, take $f \in F$ and add it to $\mathcal{D}$. This is possible because F is never empty, thanks to C-NAbsEmpty.
2. For each $x \neq_F y \in c$, take $f \in F$ and add it to $\mathcal{D}$. This is possible because F is never empty, thanks to C-NSimEmpty.
3. For each $\mathrm{S}(x, y) \in c$ with $x \neq y$, take f such that there is no $x =_F y \in c$ with $f \in F$ and add it to $\mathcal{D}$. This is always possible, because there cannot be more that one similarity for x and y (by non-applicability of R-Sims) and that one similarity cannot be full (by $x \neq y$ and non-applicability of G-SimFull).

Of course, all the variables cannot necessarily be defined on all these features. For each variable x, we thus define $\mathcal{D}_x$ by removing all the features of $\mathcal{D}$ that appear in a feature atom from x, in an absence atom for x or in a similarity atom linking x to a lower variable. The formal definition is:

$$\begin{aligned}\mathcal{D}_x = \mathcal{D} & \backslash \{f \mid \exists y \cdot x[f]y \in c\} \\ & \backslash \{f \mid \exists F \cdot x[F]\uparrow \in c, f \in F\} \\ & \backslash \{f \mid \exists y, F \cdot y < x, x =_F y \in c, f \in F\}\end{aligned}$$

Note that $\mathcal{D}$ and, for all x, $\mathcal{D}_x$ are finite as there is a finite number of literals in the constraint. Note also that, for all x, $\mathcal{D}_x \subseteq \mathcal{D}$.

Fresh trees. These sets $\mathcal{D}_x$ will be useful to satisfy negated absence atoms but also negated similarity atoms and to ensure separation between variables. For the former, it is not important which tree will be chosen to be put under the feature of $\mathcal{D}_x$. For the two latter, however, it is. Indeed, how does one make sure that the tree chosen in the definition of a variable will not happen to be exactly the wrong tree? Consider Formula 5.19 as an example.

$$x[f]y \wedge x \neq_{\{f\}} z \tag{5.19}$$

Assume we have $z < y < x$. Assume we have μ defined on z and y. There is $f \in \mathcal{D}$ and $f \in \mathcal{D}_z$ and therefore $f \in \texttt{dom}(\mu(z))$. If we have no control over it, what prevents $\mu(z)(f)$ from being equal to $\mu(y)$?

Our solution to this problem will be to define *fresh trees*. We will ensure during all the induction that the fresh trees are different from one another and different from any value given to a variable. The former is easy: we will have an infinite supply of fresh trees from which we will only use a finite number. We will therefore have all the freedom to choose different fresh trees every time. For the latter, we are going to take a special feature f_0 that does not appear in any feature atom or negated feature atom of c and that does not appear in $\mathcal{D}$.

Formally, take:

$$f_0 \notin \{f \mid \exists x, y \cdot x[f]y \in c\} \cup \{f \mid \exists x, y \cdot \neg x[f]y \in c\} \cup \mathcal{D}$$

This is always possible if the set of features $\mathcal{F}$ is infinite.[21] A *fresh tree* is any tree t such that $f_0 \in \mathtt{dom}(t)$. There is an infinity of such trees. In this proof, we are going to ensure that no value given to a variable is ever fresh.

Induction hypothesis. For all $x \in \mathcal{V}(c)$, there exists $\rho_{\leq x}$ a valuation over the variables smaller or equal to x such that:

1. $\rho_{\leq x}$ satisfies the part of the formula containing only variables smaller or equal to x. In other words, $\rho_{\leq x} \models_{\mathcal{FT}} c_{\leq x}$.
2. For all $y \leq x$, $\rho_{\leq x}(y)$ is defined on all the features of $\mathcal{D}$ unless there is an absence atom covering them. In other words, for all $f \in \mathcal{D}$, either there exists $y[F]\uparrow \in c_{\leq x}$ with $f \in F$ or $f \in \mathtt{dom}(\rho_{\leq x}(y))$. Note that we are talking about $\mathcal{D}$ here, not $\mathcal{D}_y$.
3. For all $\mathrm{S}(y, z) \in c_{\leq x}$ with $y \neq z$, $\rho_{\leq x}(y)$ and $\rho_{\leq x}(z)$ are different on all the features of $\mathcal{D}$ unless there is a similarity atom covering them. In other words, for all $f \in \mathcal{D}$, either there exists $y =_F z \in c_{\leq x}$ with $f \in F$ or $\rho_{\leq x}(y)$ and $\rho_{\leq x}(z)$ are different in f: either $f \in \mathtt{dom}(\rho_{\leq x}(y))$ but $f \notin \mathtt{dom}(\rho_{\leq x}(z))$, or the contrary, or $f \in \mathtt{dom}(\rho_{\leq x}(y))$ and $f \in \mathtt{dom}(\rho_{\leq x}(z))$ but $\rho_{\leq x}(y)(f) \neq \rho_{\leq x}(z)(f)$.
4. For all $y \leq x$, $\rho_{\leq x}(y)$ is not a fresh tree, that is $f_0 \notin \mathtt{dom}(\rho_{\leq x}(y))$.
5. For all $y \leq x$, for all $f \in \mathtt{dom}(\rho_{\leq x}(y))$, either there is a feature atom $y[f]z \in c_{\leq x}$ or $\rho_{\leq x}(y)(f)$ is a fresh tree.
6. For all $y, z \leq x$, $y \neq z$for all $f \in \mathtt{dom}(\rho_{\leq x}(y)) \cap \mathtt{dom}(\rho_{\leq x}(z))$, if $\rho_{\leq x}(y)(f)$ and $\rho_{\leq x}(z)(f)$ are both fresh trees, then either there is a similarity atom $y =_F z \in c_{\leq x}$ such that $f \in F$ or $\rho_{\leq x}(y)(f) \neq \rho_{\leq x}(z)(f)$.

Point 1 is the target of the proof. When given the biggest variable x, it simply states that $\rho_{\leq x} \models_{\mathcal{FT}} c$. Points 2 and 3 state exactly what we have been building $\mathcal{D}$ for. The former states that we try our best to have all the features of $\mathcal{D}$ present in the tree associated with each variables. The only situation in which we cannot is when there exists an absence atom. This point will be used to handle negated absence atoms. The latter states that we try out best to realise a difference in all the features of $\mathcal{D}$ between each pair of separated variables. The only situation in which we cannot is when there exists a similarity atom. This point will be used to handle negated similarity atoms. The three remaining points, points 4, 5 and 6, are technical points necessary to ensure that the so-called fresh trees respect the properties that we expect from them, that is that they are different from any value that is defined in other places.

Beginning of the induction. Let us show induction hypothesis by induction on the variables of c, following $<$. For any variable x, we will assume that the property holds for $\rho_{<x}$ and $c_{<x}$ and we will show that it also holds for $\rho_{\leq x}$ and $c_{\leq x}$.

[21] This hypothesis of infinity of $\mathcal{F}$, only used here, can be avoided by defining fresh trees as trees containing a specific pattern that we then carefully avoid reproducing in the other trees. This adds technicalities to the proof and we do not think that the result is worth the extra technical points in this presentation.

Note that, for all variable x, since c is irreducible with respect to $\mathcal{R}_2$, both $c_{<x}$ and $c_{\leq x}$ are also irreducible with respect to $\mathcal{R}_2$. This is because Lemma 4.6 holds here, as $c_{<x}$ and $c_{\leq x}$ can be seen as global parts of c for some sets:

$$c_{<x} = \mathcal{G}_{\{y | y \geq x\}}(c) \qquad\qquad c_{\leq x} = \mathcal{G}_{\{y | y > x\}}(c)$$

If the proof by induction holds, then the property holds for all $x \in \mathcal{V}(c)$. The smallest and the biggest variables are particularly interesting for us.

- If x is a minimal element, then the property trivially holds as there are no variable smaller than x, $\rho_{<x}$ is the valuation of empty domain and $c_{<x}$ is empty. This will serve as base case for the induction.
- If x is a maximal element, then the induction hypothesis implies that there exists $\rho_{\leq x}$ such that $\rho_{\leq x} \models_{\mathcal{FT}} c_{\leq x}$. Since $c_{\leq x} = c$ in this situation, then c is satisfiable.

Extension of $\rho_{<x}$ to $\rho_{\leq x}$. We define $\rho_{\leq x}$ by extending $\rho_{<x}$ to x. That is, we take $\rho_{\leq x}(y) = \rho_{<x}(y)$ for all $y < x$. For x, we define $\rho_{\leq x}(x)$ on the domain:

$$\mathtt{dom}(\rho_{\leq x}(x)) = \mathcal{D}_x \cup \{f \mid \exists y \cdot x[f]y \in c\} \cup \bigcup_{\substack{x =_F y \in c \\ y < x}} (\mathtt{dom}(\rho_{<x}(y)) \cap F)$$

For each $f \in \mathcal{D}_x$, take fresh_f a fresh tree not included in $\{\rho(y)(f) \mid y < x, f \in \mathtt{dom}(\rho(y))\}$. This is possible as there are only a finite number of $y < x$ but an infinite number of fresh trees.

$\rho_{\leq x}(x)$ is defined on all f in the aforementioned domain as follows:

$$\rho_{\leq x}(x)(f) = \begin{cases} \rho_{<x}(y) & \text{if } x[f]y \in c_{\leq x} & (5.20) \\ \rho_{<x}(y)(f) & \text{if } x =_F y \in c_{\leq x} \text{ with } y < x,\ f \in F \text{ and } f \in \mathtt{dom}(\rho_{<x}(y)) & (5.21) \\ \mathit{fresh}_f & \text{if } f \in \mathcal{D}_x & (5.22) \end{cases}$$

This definition indeed defines a correct valuation. The proof is exactly the same as in Section 4.A. By definition of $\mathcal{D}_x$, case 5.22 is disjoint from the cases 5.20 and 5.21. These two cases are not disjoint but, by non-applicability of C-Cycle, D-Feats, P-Feat-Sim and P-Sims, they are consistent nonetheless.

Order of the proof. The proofs of the various points of the induction hypothesis on $\rho_{\leq x}$ and $c_{\leq x}$ use of course the induction hypothesis that give them results on $\rho_{<x}$ and $c_{<x}$. However, some points also use the fact that other points have already been shown to be true on $\rho_{\leq x}$ and $c_{\leq x}$. This is not a problem as long as there is no circular dependency between the points.

Figure 5.30 shows a graph of dependencies of the points between themselves. An arrow from a point a to a point b means that the proof of point a uses the result of the proof of point b.

This poses a problem for the presentation of this proof because it means that the three technical points, Points 4, 5 and 6, have to be proven before Points 2 and 3 which, themselves, have to be proven before Point 1. We believe this makes the proof complicated to read as one does not understand the necessity of these technicalities.

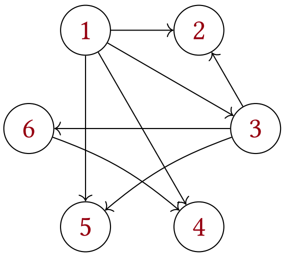


Figure 5.30: Dependencies of the proofs of the six points of the induction hypothesis

We will therefore present the proofs of all six points in order in which they appear in the induction hypothesis. We will clearly make the different in the text between the use of "point 4" and "induction hypothesis, point 4". The former is a use of the result of point 4 on $\rho_{\leq x}$. The latter is a use of the induction hypothesis on $\rho_{<x}$.

Proof of point 1. Let us proceed to the heart of the proof and show that $\rho_{\leq x}$ satisfies $c_{\leq x}$. By induction hypothesis, point 1, $\rho_{<x} \models_{\mathcal{FT}} c_{<x}$. Since $\rho_{\leq x}(y) = \rho_{<x}(y)$ for all $y < x$, there only remains to show that $\rho_{\leq x}$ satisfies the literals of $c_{\leq x}$ that actually mention x. We reason by exhaustive analysis over all literal forms. This gives us six cases: feature atom (1), negated feature atom (2), absence atom (3), negated absence atom (4), similarity atom (5) and negated similarity atom (6). Since feature and similarity atoms are binary predicates, there will be sub-cases depending on whether both their variables are x or only one.

1. Firstly, let us consider feature atoms. We have three sub-cases, as such atoms can be $x[f]x$ (1a), $x[f]y$ with $y < x$ (1b) or $y[f]x$ with $y < x$ (1c).
   (a) It is impossible to have a feature atom $x[f]x$ in $c_{\leq x}$ by non-applicability of C-Cycle.
   (b) The feature atoms $x[f]y$ in $c_{\leq x}$ with $y < x$ are satisfied by definition of ρ, case 5.20.
   (c) It is impossible to have a feature atom $y[f]x$ in $c_{\leq x}$ with $y < x$ by definition of the order $<$.
2. Secondly, let us consider negated feature atoms. We have three sub-cases, as such atoms can be $\neg x[f]x$ (2a), $\neg x[f]y$ with $y < x$ (2b) or $\neg y[f]x$ with $y < x$ (2c).
   (a) The negated feature atoms $\neg x[f]x$ in $c_{\leq x}$ are trivially satisfied in our model of finite feature trees.
   (b) Consider a negated feature atom $\neg x[f]y$ in $c_{\leq x}$ with $y < x$. If $f \notin \mathtt{dom}(\rho_{\leq x}(x))$, then this literal is satisfied. Assume now that $f \in \mathtt{dom}(\rho_{\leq x}(x))$. By point 5, either there exists a feature atom $x[f]z \in c_{\leq x}$ or $\rho_{\leq x}(x)(f)$ is a fresh tree. The former is impossible, by non-applicability of D-NFeat-Feat. By point 4, $\rho_{\leq x}(y)$ is not a fresh tree. Therefore $\rho_{\leq x}(x)(f) \neq \rho_{\leq x}(y)$ and our literal is satisfied.
   (c) See 2b.
3. Thirdly, let us consider absence atoms $x[F]\uparrow$ in $c_{\leq x}$. Let us take any $f \in F$ and show that $f \notin \mathtt{dom}(\rho_{\leq x}(x))$. By definition of ρ, this amounts to show that f is not in $\mathcal{D}_x$ (3a) nor in $\{f \mid \exists y \cdot x[f]y \in c\}$ (3b) nor in any $\mathtt{dom}(\rho_{<x}(y)) \cap G$ for any $x =_G y$ in $c_{\leq x}$ with $y < x$ (3c).
   (a) f is not in $\mathcal{D}_x$ because, by definition of $\mathcal{D}$, $\mathcal{D}_x$ does not contain any feature coming from an absence atom on x in c, which is precisely what f is.
   (b) f is not in $\{f \mid \exists y \cdot x[f]y \in c\}$ as that would imply the existence of a y such that $x[f]y \in c$. By definition of the order $<$, $y < x$ so $x[f]y$ is would also be in $c_{\leq x}$. Finally, by non-applicability of C-Feat-Abs, and because we have an absence atom $x[F]\uparrow \in c_{\leq x}$, this is not possible.
   (c) f is not in any $\mathtt{dom}(\rho_{<x}(y)) \cap G$ for any $x =_G y$ in $c_{\leq x}$ with $y < x$. Indeed, assume the existence of such a similarity atom $x =_G y$ with $f \in G$. By non-applicability of P-Abs-Sim, there is be $\bigwedge_i y[H_i]\uparrow \in c_{\leq x}$ such that $F \cap G \subseteq \bigcup_i H_i$. Since $f \in F \cap G$, that implies the existence of an i_0 such that $f \in H_{i_0}$. By induction hypothesis, point 1, $\rho_{<x}$ satisfies $y[H_{i_0}]\uparrow$ and therefore $f \notin \mathtt{dom}(\rho_{<x}(y))$.

4. Fourthly, let us consider negated absence atoms $\neg x[F]\uparrow$ in $c_{\leq x}$. By definition of $\mathcal{D}$, there is a feature $f \in F \cap \mathcal{D}$. By point 2, either there exists $x[G]\uparrow \in c_{\leq x}$ with $f \in G$ or $f \in \texttt{dom}(\rho_{\leq x}(x))$. The former is not possible because of non-applicability of R-NAbs-Abs. The latter implies that our literal is satisfied.

5. Fifthly, let us consider similarity atoms. We have two sub-cases, as such atoms can be $x =_F x$ (5a) or $x =_F y$ with $y < x$ (5b).[22]

   (a) The reflexive similarity atoms are trivially satisfied.

   (b) The similarity atoms $x =_F y$ in $c_{\leq x}$ with $y < x$ are satisfied by definition of ρ, case 5.21.

6. Sixthly and lastly, let us consider negated similarity atoms. We have two sub-cases, as such atoms can be $x \neq_F x$ (6a) or $x \neq_F y$ with $y < x$ (6b).[23]

   (a) It is impossible to have a negated similarity atom $x \neq_F x$ in $c_{\leq x}$ by non-applicability of C-NSimRefl.

   (b) Consider a negated similarity atom $x \neq_F y$ in $c_{\leq x}$ with $y < x$. By definition of $\mathcal{D}$, there is a feature $f \in F \cap \mathcal{D}$. By non-applicability of D-NSim, there is $\mathsf{S}(x, y) \in c_{\leq x}$. By point 3, either there exists $x =_F y \in c_{\leq x}$ with $f \in F$ or $\rho_{\leq x}(x)$ and $\rho_{\leq x}(y)$ differ in f. The former is not possible because of non-applicability of R-NSim-Sim. The latter implies that our literal is satisfied.

Proof of point 2. By induction hypothesis, point 2, for all $y < x$ and for all $f \in \mathcal{D}$, either there is an absence atom $y[F]\uparrow \in c_{<x}$ with $f \in F$ or $f \in \texttt{dom}(\rho_{<x}(y))$. Since $\rho_{\leq x}$ is an extension of $\rho_{<x}$, this is also true for $\rho_{\leq x}$. There remains to show that for all $f \in \mathcal{D}$, either there is an absence atom $x[F]\uparrow \in c_{\leq x}$ with $f \in F$ or $f \in \texttt{dom}(\rho_{\leq x}(x))$.

By definition of ρ, case 5.22, if f is not only in $\mathcal{D}$ but also in $\mathcal{D}_x$, then $f \in \texttt{dom}(\rho_{\leq x}(x))$. If f is not in $\mathcal{D}_x$ then, by definition of $\mathcal{D}$, it must be in $\{f \mid \exists y \cdot x[f]y \in c\}$ (1), $\{f \mid \exists F \cdot x[F]\uparrow \in c, f \in F\}$ (2) or $\{f \mid \exists y, F \cdot y < x, x =_F y \in c, f \in F\}$ (3). Let us analyse these three cases separately.

1. If f is in $\{f \mid \exists y \cdot x[f]y \in c\}$, then there exists a feature atom $x[f]y \in c$. By definition of the order $<$, $y < x$ and this feature atom is therefore also in $c_{\leq x}$. By definition of ρ, case 5.20, $f \in \texttt{dom}(\rho_{\leq x}(x))$.

2. If f is in $\{f \mid \exists F \cdot x[F]\uparrow \in c, f \in F\}$, then there exists an absence atom $x[F]\uparrow$ in c – and thus in $c_{\leq x}$ – with $f \in F$.

3. If f is in $\{f \mid \exists y, F \cdot y < x, x =_F y \in c, f \in F\}$, then there exists a similarity atom $x =_F y \in c$ with $y < x$ with $f \in F$. By induction hypothesis, point 2, applied on y, either there exists $y[G]\uparrow \in c$ with $f \in G$ (3a) or $f \in \texttt{dom}(\rho_{<x}(y))$ (3b).

   (a) If there exists $y[G]\uparrow \in c$ with $f \in G$ then, by non-applicability of P-Abs-Sim, there is $\bigwedge_i x[H_i]\uparrow \in c$ with $F \cap G \subseteq \bigcup_i H_i$. Since $f \in F \cap G$, there exists an i_0 such that $f \in H_{i_0}$.

   (b) If $f \in \texttt{dom}(\rho_{<x}(y))$ then, by definition of ρ, case 5.21, $f \in \texttt{dom}(\rho_{\leq x}(x))$.

Proof of point 3. By induction hypothesis, point 3, for all $\mathsf{S}(y, z)$ in $c_{<x}$ with $y \neq z$, and for all $f \in \mathcal{D}$, either there exists $y =_F z \in c_{<x}$ with $f \in F$ or $\rho_{<x}(y)$ and $\rho_{<x}(z)$ are different in f. Since $\rho_{\leq x}$ is an extension of $\rho_{<x}$, then the point holds for $\rho_{\leq x}$ too. There remains to prove that, for all $\mathsf{S}(x, y)$ with $y < x$, and for all $f \in \mathcal{D}$, either there exists $x =_F y \in c_{\leq x}$ with $f \in F$ of $\rho_{\leq x}(x)$ and $\rho_{\leq x}(y)$ are different in f.

[22] There is no third sub-case for $y =_F x$ as similarity atoms are seen as symmetric.

[23] There is no third sub-case for $y \neq_F x$ as negated similarity atoms are seen as symmetric.

Assume thus the existence of $\mathrm{S}(x, y) \in c_{\leq x}$. Take any $f \in \mathcal{D}$. We are going to show that either there exists $x =_F y$ with $f \in F$ or $\rho_{\leq x}(x)$ and $\rho_{\leq x}(y)$ are different in f. Let us distinguish immediately 9 cases depending on how $\rho_{\leq x}(x)$ and $\rho_{\leq x}(y)$ are in f. These 9 cases come from 3 cases for x and 3 cases for y. The 3 cases for x are the following: either $f \notin \mathtt{dom}(\rho_{\leq x}(x))$ or $\rho_{\leq x}(x)(f)$ is a fresh tree or $\rho_{\leq x}(x)(f)$ is not a fresh tree. The 3 cases for y are similar. In fact, out of these 9 cases, 6 can trivially be removed. Indeed, if one of $\rho_{\leq x}(x)$ and $\rho_{\leq x}(y)$ is undefined in f but not the other, then they differ trivially. This removes 4 cases. If they are both defined but one is fresh and not the other, then they differ trivially again. This removes 2 more cases. Basically, the only 3 interesting cases are when $\rho_{\leq x}(x)$ and $\rho_{\leq x}(x)$ have the same status in f. We will thus consider these 3 cases: either $f \notin \mathtt{dom}(\rho_{\leq x}(x))$ and $f \notin \mathtt{dom}(\rho_{\leq x}(x))$ (1) or $\rho_{\leq x}(x)(f)$ and $\rho_{\leq x}(y)(f)$ are both fresh trees (2) or they are both not fresh trees (3).

1. If $f \notin \mathtt{dom}(\rho_{\leq x}(x))$ and $f \notin \mathtt{dom}(\rho_{\leq x}(y))$ then, by point 2, there are two absence atoms $x[F]\uparrow$ and $y[G]\uparrow$ in c with $f \in F$ and $f \in G$. By non-applicability of D-Abs-Sep, there is then $\bigwedge_i x =_{H_i} y$ in c such that $F \cap G \subseteq \bigcup_i H_i$.[24] Since $f \in F \cap G$, there is an i_0 such that $f \in H_{i_0}$.

2. If $\rho_{\leq x}(x)(f)$ and $\rho_{\leq x}(y)(f)$ are both fresh trees then, by point 6, either there is $x =_F y$ in c with $f \in F$ or $\rho_{\leq x}(x)(f) \neq \rho_{\leq x}(y)(f)$.

3. If $\rho_{\leq x}(x)(f)$ and $\rho_{\leq x}(y)(f)$ are both not fresh trees then, by point 5, there are feature atoms $x[f]z$ and $y[f]z'$ in $c_{\leq x}$. We distinguish two sub-cases depending on whether $z = z'$ (3a) or not (3b).

   (a) If $z = z'$ then, by non-applicability of D-FeatsEq-Sep, there is $\bigwedge_i x =_{H_i} y$ in $c_{\leq x}$ such that $f \in \bigcup_i H_i$.[25] There is therefore an i_0 such that $f \in H_{i_0}$.

   (b) If $z \neq z'$ then, by non-applicability of D-Feats-Sep, there is $\mathrm{S}(z, z')$ in $c_{\leq x}$. By definition of the order $<$, $z < x$ and $z' < x$ so $\mathrm{S}(z, z')$ is also in $c_{<x}$. We can thus use induction hypothesis, point 3, which tells us that for all $g \in \mathcal{D}$, either there is a similarity $z =_G z' \in c_{<x}$ with $g \in G$ or there is a difference between $\rho_{<x}(z)$ and $\rho_{<x}(z')$ in g. By non-applicability of R-Sims, there exists at most one similarity atom between z and z'. By non-applicability of G-SimFull, if there is one similarity atom, then it is not full. by definition of $\mathcal{D}$, there is therefore $g \in \mathcal{D}$ that is not covered by this similarity. There is therefore a difference between $\rho_{<x}(z)$ and $\rho_{<x}(z')$ in g and: $\rho_{\leq x}(x)(f) = \rho_{<x}(z) \neq \rho_{<x}(z') = \rho_{\leq x}(y)(f)$.

Proof of point 4. By induction hypothesis, point 4, for all $y < x$, $\rho_{<x}(y)$ is not a fresh tree. Since $\rho_{\leq x}$ is an extension of $\rho_{<x}$, the for all $y < x$, $\rho_{\leq x}(y)$ is not a fresh tree. There remains to show that $\rho_{\leq x}(x)$ is not a fresh tree either.

The fact that $\rho_{\leq x}(x)$ is not a fresh tree comes directly from its definition. Indeed:

$$\mathtt{dom}(\rho_{\leq x}(x)) = \mathcal{D}_x \cup \{f \mid \exists y \cdot x[f]y \in c\} \cup \bigcup_{\substack{x =_F y \in c \\ y < x}} (\mathtt{dom}(\rho_{<x}(y)) \cap F)$$

By definition of fresh trees, $f_0 \notin \mathcal{D}_x \cup \{f \mid \exists y \cdot x[f]y \in c\}$. By induction hypothesis, point 4, for all $y < x$, $f_0 \notin \mathtt{dom}(\rho_{<x}(y))$. Therefore, $f_0 \notin \mathtt{dom}(\rho_{\leq x}(x))$ and $\rho_{\leq x}(x)$ is not a fresh tree.

Proof of point 5. By induction hypothesis, point 5, for all $y < x$, if $f \in \mathtt{dom}(\rho_{<x}(y))$, either there is a feature atom $y[f]z \in c_{<x}$ or $\rho_{<x}(y)(f)$ is a fresh tree. Since $\rho_{\leq x}$ is an extension of $\rho_{<x}$, then this is also true of $\rho_{\leq x}(y)$ for all $y < x$. There remains to show that it is also true for $\rho_{\leq x}(x)$.

[24] In fact, by non-applicability of R-Sims, there is only one such similarity.

[25] In fact, by non-applicability of R-Sims, there is only one such similarity.

Let us thus take $f \in \mathtt{dom}(\rho_{\leq x}(x))$ and show that either there is a feature atom $x[f]y \in c_{\leq x}$ or $\rho_{\leq x}(x)(f)$ is a fresh tree. Let us consider the three cases of the definition of ρ.

- case 5.20 implies precisely that there exists $x[f]y \in c_{\leq x}$.
- case 5.22 defines precisely $\rho_{\leq x}(x)(f)$ as a fresh tree.
- case 5.21 implies the existence of a similarity atom $x =_F z$ in $c_{\leq x}$ with $z < x$, $f \in F$ and $f \in \mathtt{dom}(\rho_{<x}(z))$. By induction hypothesis, point 5, either there is a feature atom $z[f]y$ in $c_{<x}$ or $\rho_{<x}(z)(f)$ is a fresh tree. In the first case, by non-applicability of P-Feat-Sim, there is a feature atom $x[f]y$ in $c_{\leq x}$. In the second case, since $\rho_{\leq x}(x)(f) = \rho_{<x}(z)(f)$, then $\rho_{\leq x}(x)(f)$ is fresh.

Proof of point 6. By induction hypothesis, point 6, for all $y, z < x, y \neq z$, and for all $f \in \mathtt{dom}(\rho_{<x}(y)) \cap \mathtt{dom}(\rho_{<x}(z))$, if $\rho_{<x}(y)(f)$ and $\rho_{<x}(z)(f)$ are both fresh trees, then either there is a similarity atom $y =_F z$ in $c_{<x}$ such that $f \in F$ or $\rho_{<x}(y)(f) \neq \rho_{<x}(z)(f)$. Since $\rho_{\leq x}$ is an extension of $\rho_{<x}$, this is also true for $\rho_{\leq x}$. There remains to show that for all $y < x$, and for all $f \in \mathtt{dom}(\rho_{\leq x}(x)) \cap \mathtt{dom}(\rho_{\leq x}(y))$, if $\rho_{\leq x}(x)(f)$ and $\rho_{\leq x}(y)(f)$ are both fresh trees, then either there is a similarity atom $x =_F y$ in $c_{\leq x}$ such that $f \in F$ or $\rho_{\leq x}(x)(f) \neq \rho_{\leq x}(y)(f)$.

Let us thus take $y < x$ and $f \in \mathtt{dom}(\rho_{\leq x}(x)) \cap \mathtt{dom}(\rho_{\leq x}(y))$. Let us consider the three cases of the definition of ρ.

- case 5.20 implies that there exists $x[f]z \in c_{\leq x}$, with $\rho_{\leq x}(x)(f) = \rho_{\leq x}(z)$. By point 4, however, $\rho_{\leq x}(z)$ cannot be fresh, making this case impossible.
- case 5.22 defines $\rho_{\leq x}(x)(f)$ as a fresh tree that is different from all the $\rho_{\leq x}(z)(f)$ with $z < x$. In particular, we have $\rho_{\leq x}(x)(f) \neq \rho_{\leq x}(y)(f)$.
- case 5.21 implies the existence of a similarity atom $x =_F z$ in $c_{\leq x}$ with $z < x$, $f \in F$ and $f \in \mathtt{dom}(\rho_{<x}(z))$. Since $\rho_{\leq x}(x)(f) = \rho_{\leq x}(z)(f)$, the latter is also fresh. Induction hypothesis, point 6, applied to y and z, tells us that either there is a similarity $y =_G z$ in $c_{<x}$ with $f \in G$ or $\rho_{<x}(y)(f) \neq \rho_{<x}(z)(f)$. In the first case, by non-applicability of P-Sims, there is $\bigwedge_i x =_{H_i} y$ in $c_{\leq x}$ such that $F \cap G \subseteq \bigcup_i H_i$. Since $f \in F \cap G$, there is i_0 such that $f \in H_{i_0}$. In the second case, $\rho_{\leq x}(x)(f) = \rho_{\leq x}(z)(f) \neq \rho_{\leq x}(y)(f)$.

Chapter 6

Modelisation of POSIX Shell

"Using a console without a proper shell; it is not going to be safe!"[1]
— The Doctor, *Doctor Who (2005), season 6 episode 4*

Disclaimer: The work presented in this chapter is joint with Benedikt Becker, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu and Ralf Treinen. Since we have participated in each aspect of this work, we believe its presentation does belong here.

In our work, we need to model the maintainer scripts of Debian. Three things guide us in this modelisation. Firstly, the Debian Policy [22] describes what can be expected of maintainer scripts. In particular, it states that one can always assume the Shell of a Debian system to be that of POSIX [20]. Incidentally, the huge majority – 99% – of maintainer scripts are written in this language. Secondly, and following this remark, the POSIX standard [20] describes extensively the POSIX Shell language. We will see in this chapter, however, that this standard is hard to understand and implement correctly. Thirdly, and most importantly, we are guided in our modelisation by the actual use that is made of these scripts. Indeed, we only need to model the subset of Shell that is used in these maintainer scripts.

Following these remarks, we get to choose between a complete generic modelisation of Shell or a specific modelisation of a subset of Shell. The former brings genericity and ensures that we can treat all Shell scripts. The latter, although ad hoc, has the advantage that we only need to handle a small part of what can be found in Shell scripts. It however requires us to be able to define what subset of Shell is necessary to model and what can be left out. Whatever decision we take, there will remain a crucial question: how can we be sure that this modelisation is correct with respect to Shell? That is, how can we ensure that we handle correctly the syntax and the semantics of Shell in our tool?

There have been few attempts to formalize the Shell. The work behind Abash [Mazurak & Zdancewic 2007] contains a formalisation of the part of the semantics concerned with variable expansion and word splitting. The Abash tool itself performs abstract interpretation to analyse possible arguments passed by

[1] I wanted to write a quote on Shell, but I did not know whether I should use `'`, `"`, `` ` `` or `$(`.

bash scripts to Unix commands, and thus to identify security vulnerabilities in bash scripts. It is however limited to this particular point of bash scripts.

Several other tools can spot certain kinds of errors in Shell scripts. The checkbashisms [12] script detects usage of bash-specific syntax in Shell scripts, it is based on matching Perl regular expressions against a normalised Shell script text. It does not include a parser for any variant of Shell. This tool is currently used in Debian as part of the lintian package analysing suite.

The tool shellcheck [17] detects error-prone usage of the Shell language. This tool is written in Haskell with the parser combinator library Parsec. Therefore, there is no YACC grammar in the source code to help us determine how far from the POSIX standard the language recognised by shellcheck is. Besides, the tool does not produce intermediate concrete syntax trees which forces the analyses to be done on the fly during parsing itself. This approach lacks modularity since the integration of any new analysis requires the modification of the parser source code. Nevertheless, as it is hand-crafted, the parser of shellcheck can keep a fine control on the parsing context: this allows for the generation of very precise and helpful error messages.

More recently, Greenberg has started a line of work on the formalisation of the semantics of Shell [Greenberg 2017; Greenberg 2018a; Greenberg 2018b]. The main result of this line of work is Smoosh [16; Greenberg & Blatt 2019], an executable formal semantics for Shell. It is heavily tested to show that it conforms to the POSIX standard. This makes it a canonical implementation and a reference semantics for Shell. The development of Smoosh led to the discovery of numerous bugs in widely used implementations of Shell as well as in the POSIX standard and its test suite. For the syntactic analysis, Smoosh still relies on external parsers and mainly the one provided with dash through libdash, although there exists work to integrate it with our parser, Morbig, described in Section 6.1.2.

This chapter is organised in two sections. Syntactic aspects of the modelisation of Shell are discussed in Section 6.1 and semantic aspects in Section 6.2.

6.1 Syntactic Aspects

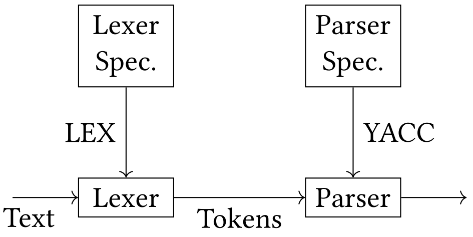


Figure 6.1: Standard pipeline of lexing and parsing commonly found in compilers and interpreters

Syntactic analysis is most often decomposed into two distinct phases [Aho et al. 1986; Levine et al. 1992]. The first phase is that of *lexical analysis* – aka. *lexing* – which synthesises a stream of tokens from a stream of input characters by recognising tokens as meaningful character sequences and by ignoring insignificant character sequences such as layout. The second phase is that of *parsing* which synthesises a *parse tree* from the stream of tokens according to some formal grammar.

For many languages, these two phases are described in a document that contains specifications for the lexer and the parser under the form of LEX- and YACC-like formats. It is then easy for anyone aiming at writing a new compiler or interpreter for this language to take the standard, feed these specifications to tools similar to LEX and YACC. The result is a program dealing with the syntactic analysis of the language in question. A representation of this process can be found in Figure 6.1.

This approach has several advantages. The LEX/YACC input is a high-level formal description of the target language. It is then easy to compare the one given in the standard to the one given in the implementation and check that they indeed match. Moreover, these two formats being fairly universal, it is easy for any programmer that has used them to apprehend the new language. The LEX/YACC code generators produce low-level efficient code using well-understood computational devices, such as finite-state transducers for lexical analysis and pushdown automata for parsing.

Unfortunately, nothing of this usual way of doing things is applicable for Shell. The POSIX standard does not provide but a low-level description of the lexical analysis. It does provide a high-level YACC-style grammar but this grammar is annotated by several rules that change the behaviour of the parser – and even of the lexer. POSIX standard aside, the Shell language itself is hard (and actually impossible in general) to parse using the standard decomposition described above, and more generally using the standard parsing tools and techniques.

These difficulties not only raise a challenge in terms of programming but also in terms of trustworthiness. In Section 6.1.1, we describe why the usual approach cannot work for Shell. In Section 6.1.2, we describe our implementation and how we nonetheless managed to maintain an important part of generated code in our implementation. We then discuss our attempts to guarantee the quality of our implementation in Section 6.1.3.

6.1.1 Horrors in the Syntax of Shell

This section does not aim at being a comprehensive list of pitfalls that one can encounter while considering the problem of syntactic analysis of Shell. For more information on this topic, we invite the reader to refer to our previous work [Jeannerod et al. 2017b; Régis-Gianas et al. 2020] or to other documents detailing such pitfalls [20; Garfinkel et al. 1994; Greenberg & Blatt 2019].

As explained before, in usual programming languages, most of the categories of tokens are specified by means of regular expressions. Lexer generators (eg. LEX) conveniently turn such high-level specifications into efficient finite state transducers, which makes the resulting implementation both reliable and efficient. The token recognition process for the Shell language is described in the POSIX standard [20, Section 2.3], unfortunately without using any regular expressions. While other languages use regular expressions with a longest-match strategy to delimit the next lexeme in the input, the specification of the Shell language uses a low-level state machine which explains instead how tokens must be delimited in the input and how the delimited chunks of input must be classified into two categories: words and operators.

Consider Figure 6.2 as an example. By the lexical conventions of most programming languages, the first line would be decomposed as five distinct tokens (namely `BAR`, `=`, `'foo'`, `"ba"` and `r`) while the lexical conventions of the Shell language considers the entire line `BAR='foo'"ba"r` as a single token, classified into the category of words. On the second line, the input is split into the tokens `X=0`, `echo` and `x$BAR" "$(echo $(date))`. Notice that the third token contains *subshells*, that is nested quotations of the form `$(···$(···))` which themselves can contain any piece of Shell code, including complex control structures.

```
BAR='foo'"ba"r
X=0 echo x$BAR" "$(echo $(date))
```

Figure 6.2: Example words

From the lexical point of view, a subshell invocation is simply a word. Delimiting these subshell invocations is hardly reducible to regular expression matching. Indeed, to determine the end of a subshell invocation, it is necessary to recursively call the Shell command parser so that it consumes the rest of the input until a complete command is parsed. Consider Figure 6.3 as an example. On this command, determining if the right parenthesis is ending the subshell requires deciding if the parenthesis is escaped or not. However, as explained in the

```
echo `echo \`echo $(echo foo\\\\\\)\``
```

Figure 6.3: Nested subshells

```
for do in for do in echo done; do echo $do; done
```

Figure 6.5: Promotion of a word to a reserved word

```
f=~niols/"$(echo foo)${x:=bar}"'$baz'[a-b]*
```

Figure 6.6: A word can have many components

previous section, this analysis is non trivial and more or less requires to perform a full syntactical analysis of the input.

While the recognition of tokens is independent from the parsing context, their classification into words, operators, newlines and end-of-file markers must be refined further to obtain the tokens actually used in the formal grammar specified by the standard. While chunks categorised as operators are easily transformed into a more specific token, inputs chunk categorised as words can be promoted to reserved words or to assignment words only under specific, *ad hoc* conditions; otherwise the word is not promoted and stays a regular word. This means that the lexical analysis has to depend on the state of the parser.

```
CC=gcc make
make CC=cc
"./X"=1 echo
```

Figure 6.4: Promotion of a word to an assignment

The promotion of a word to an assignment depends both on the position of this word in the input and on the string representing that word. The string must be of the form `w=u` where the substring `w` must be a valid name. Consider Figure 6.4. On the first line, the word `CC=gcc` is recognised as a word assignment of `gcc` to `CC` because `CC` is a valid name for a variable, and because `CC=gcc` is written just before the command name of the simple command `make`. On the second line, the word `CC=cc` is not promoted to a word assignment because it appears after the command name of a simple command. On the last line, since `"./X"` is not a valid name for a Shell variable, the word `"./X=1"` is not promoted to a word assignment and is interpreted as the command name of a simple command.

The first side-rule of the Shell grammar given in the POSIX standard [20, Section 2.10.2] requires that a word is promoted to a reserved word if the parser state is expecting this reserved word at the current point of the input. If a word that is a potential reserved word is located where a reserved word is not expected, it is not promoted and interpreted as any other word. Consider Figure 6.5 as an example. In that example, the first occurrence of `do` as well as the words between the first occurrence of `in` and the first semicolon are not promoted to reserved words while the other occurrences of `for`, `do`, `in` and `done` are. There are exceptions to this rule as some reserved words can never appear in the position of a command. This is for instance the case of `else`. If the word `else` occurs in the position of a command, it will be promoted to reserved word but the parser will later reject such an input if there is no matching `if`.

The semantic value of a word can be complex since it can be made of subshell invocations, variables and literals. The script in Figure 6.6 is a single word read as an assignment word by the grammar. The right-hand-side of this assignment is a sequence starting with a so-called “tilde-prefix”, followed by a double-quoted sequence followed by a literal. The double-quoted sequence is itself composed of a subshell invocation represented by the concrete syntax tree of its command, followed by a variable that uses the default value `bar` when expanded. The double-quoted word is completed with a literal `baz`, a bracket range expression and pattern-matching operator matching all words.

In fact, the lexical analysis also depends on the evaluation of the Shell script. Indeed, the `alias` builtin command of the Shell amounts to the dynamic definition of macros that are expanded just before lexical analysis. Therefore, even the lexical analysis of a Shell script cannot be done without executing it, that is, lexical analysis of unrestricted Shell scripts is undecidable.

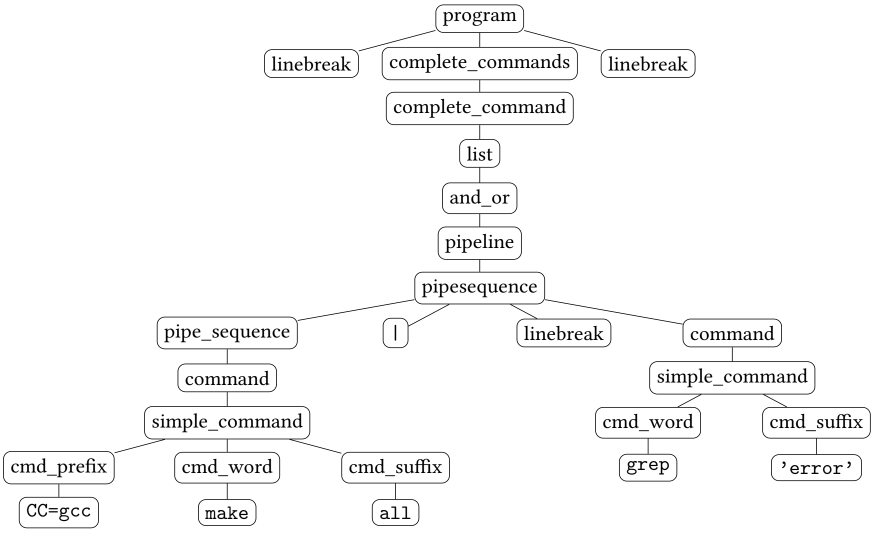


Figure 6.8: Parse tree for `CC=gcc make all | grep 'error'`

Consider Figure 6.7 as an example. To decide if `for` in the last line is a reserved word, a lexer must be able to know the success of an arbitrary program `./foo`, which is impossible to do statically. Hence, the lexer must wait for the evaluation of the first command before parsing the second one. Moreover, Shell comprises the builtin `eval` which allows for execution of arbitrary code built from a string. The use of such a builtin therefore makes the lexical analysis undecidable without executing code on-the-fly.

```
if ./foo; then
  alias x="ls"
else
  alias x=""
fi
x for i in a b; do
  echo $i
done
```

Figure 6.7: Lexical analysis is undecidable

6.1.2 Morbig, A Static Parser for Shell

We introduce Morbig [29], a static parser for a subset of the Shell language. It constructs a *concrete syntax tree* of a complete script without evaluating constructs of the language. The only limitations of Morbig are that it cannot handle Shell constructs that are inherently dynamic in nature: the `eval` builtin, unrestricted use of the `alias` builtin and premature termination of a script by an `exit` with trailing garbage in the file. These restrictions are justified by the static nature of our parser.

Morbig is designed for a variety of applications, including statistical analysis of the concrete syntax of scripts (see Section 7.1). Therefore, contrary to parsers typically found in compilers or interpreters, Morbig does not produce an abstract syntax tree from a syntactically correct source but a *parse tree* instead. A parse tree – or *concrete syntax tree* – is a tree whose nodes are grammar rule applications. See Figure 6.8 for an example parse tree for `CC=gcc make all | grep ’error’`. Because we need concrete syntax trees and because we want high assurance about the compliance of the parser with respect to the POSIX standard, reusing an existing parser implementation such as that of `libdash` was not an option. Our research project required the reimplementation of a static parser from scratch.

Before giving more details about the implementation choices, let us sum up a list of the main requirements that are implied by the technical difficulties explained in Section 6.1.1.[2]

1. The lexical analysis must be defined in terms of token delimitations, not in terms of token (regular)

[2] There are in fact other requirements related to technical difficulties that we have chosen not to describe in Section 6.1.1. We refer an interested reader to more complete articles on the topic [Régis-Gianas et al. 2020].

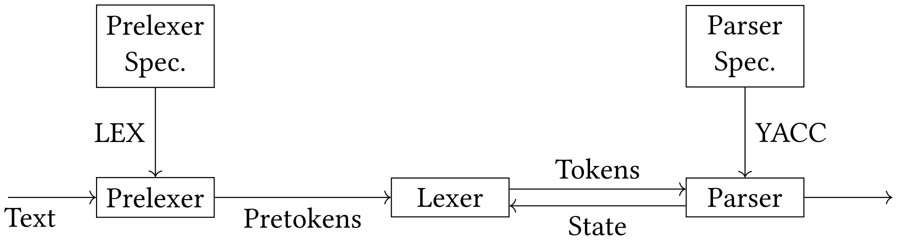


Figure 6.9: Architecture of Morbig

languages recognition.

2. The lexical analysis must be aware of the parsing context and of some contextual information like the nesting of double quotes and subshell invocations.

3. The parser must be reentrant, that is we must be able to intertwine runs of the main parser for the whole script and of sub-parsers for portions of the script. These sub-parsers are used by the lexical analysis to determine the end of words that involve subshells.

4. For the same reason, the syntactic analysis must be able to return the longest syntactically valid prefix of the input.

5. The parser must forbid certain specific applications of the grammar production rules. This point comes from the annotations of the grammar one can find in the POSIX standard which, in certain situations, prevent a specific grammar rule from applying.

In addition to these technical requirements, there is an extra methodological one: the mapping between the POSIX specification and the source code must be as direct as possible.

The tight interaction between the lexer and the parser prevents us from writing our syntactic analyser following the traditional design found in most textbooks [Aho et al. 2006; Levine et al. 1992], that is a pipeline of a lexer followed by a parser. Hence, we cannot use either the standard interfaces of code generated by LEX and YACC, because these interfaces have been designed to fit this traditional design. There exist alternative parsing technologies[3], that could have offered elegant answers to many of the requirements enumerated previously, but we believe that none of them fulfils our requirements [Régis-Gianas et al. 2020].

In this situation, one could give up using code generators and fall back to the implementation of a hand-written character-level parser. This is done in `dash` for instance: the parser of `dash` 0.5.7 is made of 1569 hand-crafted lines of C code. This parser is hard to understand because it is implemented by low-level mechanisms that are difficult to relate to the high-level specification of the POSIX standard: for example, lexing functions are implemented by means of gotos and complex character-level manipulations; the parsing state is encoded using activation and deactivation of bit fields in one global variable; some speculative parsing is done by allowing the parser to read the input tokens several times, etc.

Our main design choice is not to give up on modularity. As shown in Figure 6.9, the architecture of our syntactic analyser is similar to the common architecture found in textbooks as we clearly separate the lexing phase and the parsing phase in two distinct modules with clear interfaces. Let us now describe the original aspects of this architecture.

As suggested by the illustration, we decompose lexing into two distinct sub-phases. The first phase, called

[3] For instance scannerless generalised LR parsers or topdown general parsing combinators

prelexing is implementing the *token recognition* process of the POSIX standard. This parsing-independent step classifies the input characters into three categories of *pretokens*: operators, words and potentially significant layout characters (newline characters and end-of-input markers). This module is implemented using OCamlLex [37, Chapter 13], a lexer generator distributed with the OCaml language.

The second phase of lexing is parsing-dependent. As a consequence, a bidirectional communication between the lexer and the parser is needed. On one side, the parser is waiting for a stream of tokens to reconstruct a parse tree. On the other side, the lexer needs some parsing context to promote words to keywords or to assignment words, etc. We manage to implement all these *ad hoc* behaviours using speculative parsing, which is easily implemented thanks to the incremental and purely functional interface produced by the parser generator Menhir [27].

OCamlLex is the lexer generator of the OCaml programming language. It extends the specification language of LEX with many features, two of which are exploited in our implementation. Firstly, a lexer of OCamlLex can be defined by a set of mutually recursive entry points, which allows it to define concatenation of distinct sub-languages in a modular and readable way. Thanks to this organisation of the lexical rules, we were able to separate the lexer into a set of entry points where each entry point refers to a specific part of the POSIX standard. This structure of the source code eases documentation and code reviewing and hence increases its reliability. Secondly, the entry points of an OCamlLex lexer can be parameterised by arguments. These arguments are typically used to have the lexer track contextual information along the recognition process. Combined with recursion, these arguments provide extra expressive power, which allows our lexer to parse nested structures (eg. parenthesised quotations) even if they are not regular languages. In addition, the parameters of the lexer entry points make it possible for several lexical rules to be factorised out in a single entry point.

Menhir [27] is parser generator for the OCaml programming language. While usual YACC-generated parsers either produce a semantic value or fail if a syntax error is detected, Menhir provides an incremental interface which allows the lexer to read the state of the parser between execution steps and, thanks to functionality of the interface, to backtrack to a previous state if necessary. The lexer can then simply perform some speculative parsing to determine whether a token is compatible with the current parsing state. This is particularly useful to deal with the promotion of words to reserved words. The implementation of this feature basically consists in running the parser a first time with the word promoted to reserved word. If this results in a syntax error, we run the parser again without promoting the word.

From the programming point of view, backtracking is as cheap as declaring a variable to hold the state to recover it if a speculative parsing goes wrong. From the computational point of view, thanks to sharing, the overhead in terms of space is negligible and the overhead in terms of time is reasonable since we never transmit more than one input token to the parser when we perform such speculative parsing.

Another essential advantage of the functionality of the interface of Morbig is the fact that the parsers generated by Menhir are then *reentrant* by construction, which means the multiple instances of our parser can be running simultaneously. This property is needed in our case because the prelexer can trigger new instances of the parser to deal with subshell invocations. As it is very hard to delimit correctly subshell invocation without parsing their content, these sub-parser are given the entire input suffix and are responsible for finding the end of this subshell invocation by themselves.

Morbig [29] is Free Software, published under the GPL3 license. On a i7-4600U CPU @ 2.10GHz with 4 cores, an SSD hard drive and 8GB of RAM, it takes 7.38s to parse the 31,330 POSIX Shell scripts among the 31,582 maintainer scripts in the Debian GNU/Linux distribution and to serialise the corresponding concrete syntax trees on the disk. Our parser fails on only one script which uses indeed a `bash`-specific extension of the syntax. The average time to parse a script from the corpus of Debian maintainer scripts is

Table 6.1: Comparison of Morbig and `dash` on the whole corpus from Software Heritage. The percentages are in function of the total number of scripts.

`dash` Morbig	All	Accepted	Rejected
All	7,436,215 (100%)	5,981,054 (80%)	1,455,161 (20%)
Accepted	5,609,366 (75%)	5,607,331 (75%)	2,035 (<1%)
Rejected	1,826,849 (25%)	373,723 (5%)	1,453,126 (20%)

therefore 0.2ms (with a standard deviation which is less than 1% of this duration). The maximum parsing time is 70ms, reached for the `prerm` script of package `w3c-sgml-lib_1.3-1_all` which is 1121 lines long. We compared Morbig to `dash` on the whole archive of Shell scripts from Software Heritage, containing 7,436,215 scripts in total (see Section 6.1.3 for details). We used a machine with an Intel Xeon Processor E5-4640 v2 @ 2.20GHz with 40 cores and 756GB of RAM, where all the scripts were loaded in a tmpfs in RAM. It takes 400s to `dash` and 3400s to Morbig to parse all these scripts. This means respectively 19,000 and 2200 scripts per second. Although `dash` is faster, the difference is less than an order of magnitude.

6.1.3 Validation

What makes us believe that our approach to implement the POSIX standard will lead to a parser that can be trusted? Actually, as the specification is informal, it is impossible to prove our code formally correct. We actually do not even claim the absence of bugs in our implementation.

To improve our chance to converge to a trustworthy implementation, the development of Morbig follows four guidelines. Firstly, the source code of Morbig contains almost 20% of comments. We tried to quote the POSIX specification related to each code fragment so that a code reviewer can evaluate the adequacy between the implementation and its interpretation of the specification. We also document every implementation choice we make and we explain the programming technique used to ease the understanding of the unorthodox parts of the program, typically the speculative parsing.

Secondly, we commit ourselves to not modifying the official BNF of the grammar despite its incompleteness or the exotic nine side rules described earlier. BNF is the most declarative and formal part of the specification, knowing that our generated parser recognises the same language as this BNF argues in favour of trusting our implementation.

Thirdly, Morbig comes with a test suite which follows the same structure as the specification: for every section of the POSIX standard, we have a directory containing the tests related to that section. At this time, the test suite is relatively small since it contains just 185 tests. A code reviewer may still be interested by this test suite to quickly know if some corner case of the specification has been tested and, if not, to contribute to the test suite by the addition of a test for this corner case.

Fourthly, and in order to disambiguate several paragraphs of the standard, we have checked that the behaviour of Morbig coincides with the behaviour of Shell implementation which are believed to be POSIX-compliant, typically `dash` and `bash` (in POSIX mode).

As an additional guarantee, we ran both Morbig and `dash` on all the files detected as Shell scripts in the Software Heritage archive [33; 34; Abramatic et al. 2018]. This archive contains all the Shell scripts in GitHub, and more, for a total of 7,436,215 files. Table 6.1 shows general numbers about what both parsers accept or reject in this archive. On most scripts (95%), Morbig and `dash` do agree. It is interesting to consider the cases where they disagree, because this is where one can find bugs in one parser or the other.

Out of the scripts accepted by dash and rejected by Morbig the majority (350,259, ie. 94% and 4.7% of the total) contains bash-specific constructs in words. dash, in parse-only mode, separates words but does not look into them, hence it will only refuse them when executing the script. Morbig, on the other hand, does parse words and rejects such scripts. This is neither a bug in dash nor in Morbig as the POSIX standard does not specify whether such invalid words must be rejected during parsing or during execution. The remaining 23,464 (0.3% of the corpus) that are accepted by dash and rejected by Morbig are due to remaining bugs in Morbig or in dash.

There are only 0.03% of scripts which are accepted by Morbig and refused by dash. These are either due to bugs in Morbig, or in dash, or to the fact that the standard is ambiguous.

6.2 Semantic Aspects

The syntax of Shell is convoluted, and semantics is not any better. It can be treacherous for both the developers and the analysis tools. Based on Morbig, the parser for POSIX Shell described in Section 6.1, we have designed a statistical[4] analyser for the corpus of Shell scripts we are interested in. This statistical analyser is described in Section 7.1. We used this statistical analyser in order to know which features of Shell are mostly used in our corpus, and which features we may safely ignore. Based on this, we developed an intermediate language for Shell scripts, called CoLiS, which we will briefly describe in this section.

Since the CoLiS language is meant to be at the base of analysis and verification tools, its design has been guided by the following principles:

- CoLiS must be *cleaner* than Shell: we ignore the dangerous structures (like eval allowing to execute arbitrary code given as a string) and we make more explicit the dangerous constructions that we cannot eliminate.
- CoLiS must have clear syntax and semantics. The goal is to help in the writing of analysis tool so that one can easily be convinced of the soundness of these tools without having to care about the pitfalls of the syntax or the semantics of the underlying language.
- An automated conversion from Shell to CoLiS must be possible. Moreover, this conversion must not be "too clever" because it has to be trusted that it is correct with respect to the semantics of Shell and CoLiS. For this reason, the CoLiS language cannot be fundamentally different from Shell.

CoLiS is not conceived as a replacement of Shell in the software packages. If that was our goal, we would have designed a declarative language as a replacement, similar to how systemd has nowadays mostly replaced System-V init scripts.

In Section 6.2.1, we describe semantic features of Shell that make it hard to deal with for our analysis tool. In Section 6.2.2, we quickly present our intermediary language, CoLiS, the improvements it brings compared to Shell and the automated conversion from Shell. In Section 6.2.3, we describe the implementation of an interpreter for CoLiS and discuss the validation of CoLiS, its semantics and its conversion.

6.2.1 Horrors in the Semantics of Shell

The Shell language includes features that are well-known from other imperative programming languages, like variable assignments, conditional branching, loops – both for and while. Shell scripts may call Unix utilities which in particular may operate on the filesystem, but these utilities are not part of the Shell

[4] *Statistical*, not static! Well also static, but not only static.

```
f=~niols/"$(echo foo)${x:=bar}"'$baz'[a-b]*
```

Figure 6.6: A word can have many components

language itself, and not in the scope of the present chapter – they have been handled in Chapter 3. Without going into the details of the Shell language, there are some peculiarities which are of importance for the design of the CoLiS language.

The evaluation of expressions in Shell is done using a really expressive *expansion* mechanism. Consider Figure 6.6 for instance, restated here for convenience. Figure 6.6 is only one word containing:

- unquoted literals `f=`,
- tildes `~niols/` expansing to the home of the user `niols` if they exist,
- quoted parts `"$(echo foo)${x:-bar}"` in which expansion still happens,
- subshell invocations `$(echo foo)` which can evaluate arbitrary commands and may even contain control structures (eg. `for` loops),
- complex parameters `${x:=bar}` which expand to various things depending on the state of the variable it concerns (in this example, if `$x` is unset or set to the empty string, then it is assigned the value `bar`),
- quoted literals `'$baz'` that are not expansed even if they contain any of the above,
- and globs `[a-b]` and `*` which are basically regular expressions that change the expansion of the whole word depending on the contents of the filesystem that they match.

The expansion might fail (eg. if a subshell invocation fails) which may affect the behaviour of the whole script around the word in question.

```
path='/home'
path="$path/niols"
args='-l -a'
args="$args -h"
ls $args $path
```

Figure 6.10: Strings and lists of strings

Variables are not declared, and there is no static type discipline. In principle, values are just strings, but it is common practice in Shell scripts to abuse these strings in order to represent other kind of data structures. The most common example would be to abuse strings in order to represent lists of strings, by assuming that the elements of a list are separated by the so-called *internal field separator* (usually the blank symbol). Consider Figure 6.10 as an example. In this example, the variable `$path` is thought of as a string – and even a path – which is first set to contain `/home` and then extended to contain `/home/niols`. The variable `$args`, on the other hand, is thought of as a list – of command-line arguments – which is first set to contain the two elements `-l` and `-a` and then extended to contain `-l`, `-a` and `-h`. The last command is then a call to the utility `ls` with four arguments: `-l`, `-a`, `-h` and `/home/niols`. This pattern is quite common and quite resistant to static analysis. In particular, how can one make the difference between a misuse of a string[5] or a correct use of a list?

```
f () { g; }
g () { a=bar; }
a=foo
f
echo $a
```

Figure 6.11: Fully-dynamic scoping

Functions may access non-local variables. However, this is done according to the chronological order of the variables on the execution stack (dynamic scoping), not according to the syntactic order in the script (lexical scoping). Consider Figure 6.11 as an example. We first define two functions. `f` only calls another function `g` and `g` only updates a variable `$a` so that it contains `bar`. We then define the variable `$a` to contain `foo` and call `f`. This whole script is perfectly valid, and the result is a variable `$a` that contains `bar`. Note that, although the function `g` is defined after `f`, it is not a problem for `f` to be calling `g`. Note

[5] A tiny space in a string in a maintainer script can cause really impressive damage [15].

also that at the moment when f is defined, there is no way telling whether the call to g will fail, or be a call to an external utility or be a call to a function.

The semantics of Shell can be modified during the execution. This can be done by modifying the *internal field separator* or by calling the set builtin utility. The internal field separator is a variable, $IFS, which specifies which characters count as separator within values, and therefore where the Shell should cut values in the execution. Consider Figure 6.12 as an example. In this example, we first set a variable $file to /sys-custom which we plan to remove. However, if a modification of the IFS occurs in the meantime and changes the IFS to -, we will end up removing /sys and custom which is most likely not what we intended.

```
file=/sys-custom
IFS=-
rm -r $file
```

Figure 6.12: Example script modifying $IFS

The set utility allows a script to set or unset flags that change the behaviour of Shell. These flags can be:[6]

- -a, which make every assignment become an export,
- -C, which prevents the Shell from overwriting existing files by default,
- -e, which makes Shell exit immediately when a command fails, except when this failure is caught,
- -f, which disables pathnames, that is interpretation of globs and tilde prefixes,
- -u, which the Shell fail when expanding parameters that are unset.

```
set -e
! true
echo foo
false && true
echo bar
false
echo baz
```

Figure 6.13: Example of surprising semantics with set -e

The -e flag is particularly interesting for us as it is made mandatory in maintainer scripts by the Debian Policy.[7] It brings a form of mechanism of exceptions to the Shell. This mechanism can however be pretty surprising. Consider Figure 6.13 as an example. The first line activates the -e flag of set. We then have three instances of echo intertwined with various commands that may fail. ! true fails because true always succeeds and ! inverts return code of its command. The whole command however does not kill the Shell and the script continues, printing foo. false && true always fails because false and && fails if its first command fails. The whole command however does not kill the Shell and the script continues, printing bar. false always fails and, this time, the command does kill the Shell, stopping the script and therefore never printing baz. This example shows that, in this mode of Shell, ! true, false && true and false do not have the same semantics.

6.2.2 The CoLiS Language

The CoLiS language was first presented in 2017 [Jeannerod et al. 2017a]. Its design aimed at avoiding some pitfalls of the Shell, and at making explicit the dangerous constructions which we cannot eliminate. It was later improved upon [Becker et al. 2019; Becker et al. 2020] to increase the number of Debian maintainer scripts that could be analysed by adding more constructs and to align the previous semantics to the one of the Shell.[8]

CoLiS has a clear syntax and a *formally* defined semantics. We provide an automated and direct conversion from Shell. The correctness of the conversion from Shell to CoLiS cannot be proven formally but must be trusted based on testing and manual review of its code.

[6] Reading this list actually makes you wonder why all these options are disabled by default. We personally systematically start our Shell scripts with set -euC.

[7] Not exactly, but close: “every script should use set -e or check the exit status of every command” [22, Section 10.4].

[8] In other words, to fix bugs in our semantics.

```
if [ -h /etc/rancid/lg.conf ]; then
  rm /etc/rancid/lg.conf
fi
if [ -e /etc/rancid/apache.conf ]; then
  rm /etc/rancid/apache.conf
fi
```

Figure 1.5: `preinst` script of the `rancid-cgi` package

```
if test [ '-h'; '/etc/rancid/lg.conf' ] then
  rm [ '/etc/rancid/lg.conf' ]
fi
if test [ '-e'; '/etc/rancid/apache.conf' ] then
  rm [ '/etc/rancid/apache.conf' ]
fi
```

Figure 1.6: `preinst` script of the `rancid-cgi` package in CoLiS

```
#!/bin/sh
set -e

if [ ! -e /usr/local/lib/ocaml ]; then
  if mkdir /usr/local/lib/ocaml 2>/dev/null; then
    chown root:staff /usr/local/lib/ocaml
    chmod 2775 /usr/local/lib/ocaml
  fi
fi

[...]

for i in /usr/lib/ocaml/3.06 /etc/ocaml /var/lib/ocaml
do
  if [ -e $i/ld.conf ]; then
    echo "Removing leftover $i/ld.conf"
    rm -f $i/ld.conf
    rmdir --ignore-fail-on-non-empty $i
  fi
done
```

Figure 6.14: `postinst` script of the `ocaml-base-nox` package (excerpt; cleaned up)

```
begin
  true;

  if test [ '!'; '-e'; '/usr/local/lib/ocaml' ] then
    if mkdir [ '/usr/local/lib/ocaml' ] then
      begin
        chown [ 'root:staff'; '/usr/local/lib/ocaml' ];
        chmod [ '2775'; '/usr/local/lib/ocaml' ]
      end
    fi
  fi;

  [...]

  for i in [ '/usr/lib/ocaml/3.06'; '/etc/ocaml'; '/var/lib/ocaml' ]
  do
    if test [ '-e'; split i '/ld.conf' ] then
      begin
        echo [ 'Removing leftover ' i '/ld.conf' ] ;
        rm [ '-f'; split i '/ld.conf' ] ;
        rmdir [ '--ignore-fail-on-non-empty'; split i ]
      end
    fi
  done
end
```

Figure 6.15: `postinst` script of the `ocaml-base-nox` package in CoLiS (excerpt; cleaned up)

Figures 1.5 and 1.6 show the Shell and CoLiS versions of the `preinst` script of the `rancid-cgi` package. They are restated here for convenience. For longer scripts that include other control structures, variables and richer words, we refer the reader to Figures 6.14 and 6.15 which show the Shell and CoLiS versions of the `postinst` script of the `ocaml-base-nox` package. From these scripts, we can already note quite a few differences between Shell and CoLiS.

- The syntax of CoLiS requires mandatory usage of delimiters for string arguments and for lists of arguments. Generally speaking, the syntax of CoLiS is designed so as to remove potential ambiguities.
- The Shell utility call `set -e` is translated to `true` in CoLiS. This is because we avoid modelling `set` in CoLiS and we enforce at conversion time that scripts start by a call to `set -e`, in line with what the Debian Policy requires.
- The redirection `2> /dev/null` disappears completely in CoLiS. This is because it affects only the error output of the Shell, which usually has no impact on the semantics of a script. The conversion ensures that this hypothesis is respected throughout the whole script by keeping track of redirections and rejects scripts where the error output is redirected to standard output or to files.
- The Shell word `$i/ld.conf`, containing a variable, is converted to the CoLiS expression `split i '/ld.conf'`. We can note the presence of the keyword `split` that makes explicit the fact that the value of the variable `$i` will go through a phase where it will be split at its blank characters. As a comparison, the quoted Shell word `"$i/ld.conf"` is converted to the CoLiS expression `i '/ld.conf'` where the splitting will not take place. This is an example of our policy of making peculiarities of Shell more explicit in CoLiS.

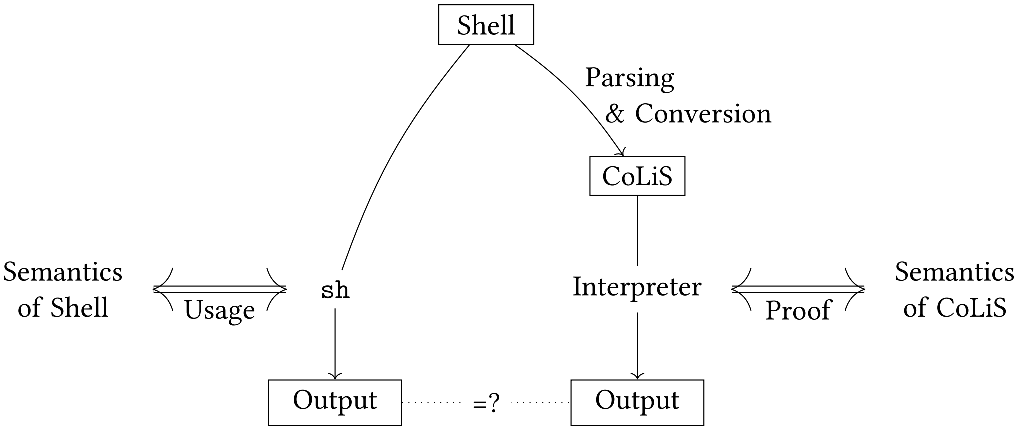


Figure 6.16: Validating the semantics of CoLiS with respect to that of Shell

The toolchain for analysing CoLiS scripts is designed with formal verification in mind: the syntax, semantics, and interpreters of CoLiS are implemented using the Why3 environment [Bobot et al. 2015] for formal verification. More precisely, the syntax of CoLiS is defined abstractly (as abstract syntax trees, AST for short) by an algebraic datatype in Why3. Then the semantics is defined by a set of inductive predicates [Becker et al. 2019] that encodes a standard, big-step operational semantics. The semantic rules cover the contents of variables and input/output buffers used during the evaluation of a CoLiS script, but they do not specify the contents of the filesystem and the behaviour of Unix utilities. The judgements and rules are parameterised by bounds on the number of loop iterations and the number of (recursively) nested function calls to allow for formalising the correctness of the symbolic interpreter. The symbolic interpreter will be described in Section 7.2.

The conversion from Shell to CoLiS is done automatically, but is not formally proven. Indeed, a formal semantics of Shell was missing until very recently [Greenberg & Blatt 2019]. For the control flow constructs, the AST of the Shell script is converted into the AST of CoLiS. For the strings (words in Shell), the translation generates either a string CoLiS expression or a list of CoLiS expressions depending on the content of the Shell string. This conversion makes explicit the string evaluation in Shell, in particular the implicit string splitting. At the present time, the converter rejects 23% of Shell scripts, either because it does not cover the full constructs of the Shell or because the CoLiS language is not rich enough to encode them (eg. usage of globs, variables with parameters, and advanced uses of redirections).

6.2.3 A Concrete Interpreter for the CoLiS Language

A concrete interpreter for the CoLiS language is implemented in Why3 [11]. Its formal specifications (preconditions and post-conditions) state the soundness of the interpreter, ie. that any result corresponds to the formal semantics with unbounded number of loop iterations and unbounded nested function calls. The specifications are checked using automated theorem provers [Jeannerod et al. 2017a].

The conformance of the semantics of CoLiS with that of Shell is not proven formally but tested by manual review and some automatic testing. For the latter, we developed a tool that automatically compares the results of the CoLiS interpreter on the CoLiS script with the results of the Debian default Shell (`dash`) on the original Shell script. This tool uses a test suite of Shell scripts built to cover the whole constructs of the CoLiS language. Its functioning is fairly simple. It is illustrated in Figure 6.16. The tool takes as input hand written Shell scripts that cover various features of the CoLiS language. These scripts do not use complex utilities but just enough to show corner cases of the semantics of constructs of Shell. They are evaluated with a Shell interpreter – usually `dash` – on one hand and converted to CoLiS and evaluated with the interpreter for CoLiS on the other hand. The output is then compared. When the outputs differ, this indicates a bug:

- either in the parsing and conversion,
- or in the semantics of CoLiS,
- or in the implementation of the Shell interpreter.

The bug cannot be in the semantics of Shell as it is our reference. It can also not be in the interpreter of CoLiS because it is proven to be sound with respect to the semantics. This test suite allowed us to fix the conversion and the formal semantics of CoLiS. As an additional outcome, it revealed a lack of conformance between dash and the POSIX standard.[9]

Since our approach in the CoLiS project is bug-oriented, this also means that it is not crucial for the semantics of CoLiS to be corresponding to that of Shell. A bug in the semantics of CoLiS would simply lead to a report containing an unreproducible bug, which we would then track down back to the semantics of CoLiS.

[9] See https://www.mail-archive.com/dash@vger.kernel.org/msg01683.html.

Chapter 7

Applications & Results

Disclaimer: The work presented in this chapter is joint with Benedikt Becker, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu and Ralf Treinen. Since we have participated in each aspect of this work, we believe its presentation does belong here.

This chapter describes applications of all the material presented in this thesis. In Section 7.1, we present the statistic analysis we ran on the corpus of maintainer scripts from Debian packages as well as some results. In Section 7.2, we present the symbolic execution of Shell scripts. Finally, in Section 7.3, we present the results that we obtained by running our toolchain on all Debian packages.

7.1 Statistic Analysis of Corpuses of Maintainer Scripts

7.1.1 Writing Analysers for Corpuses of Shell Scripts

One of the first questions which we have investigated in the CoLiS project is which features of the Shell, which Unix utilities and which options to these utilities are used and at which frequency. Such knowledge was meant to guide us in the design of our intermediary language CoLiS, described in Section 6.2. First analyses were done with simple `grep` scripts. These allowed rough estimates of the number of constructs (eg. `if`, `while`, `case`) used in the corpus, for instance. As clever as these regular expressions might have been[1], they quickly showed severe limitations, due to the difficulties of lexical analysis.

The development of the Morbig parser, described in Section 6.1, allowed us to move these analyses to another level. Indeed, since Morbig provides concrete syntax trees for Shell scripts, this allows us to define rich analysers of such syntax trees. These can easily replace complex regular expressions. Since they can be written in general-purpose programming languages, they also can provide much richer analyses.

The difficulty of writing such analysers lies in the number of different syntactic constructions of a realistic language like shell: the concrete syntax trees have 108 distinct kinds of node. Even if most of the time a single analysis focuses on a limited number of kinds of nodes, the analyser must at least traverse the other kinds of node to reach the interesting ones. Even once on analyser is written, it is not easy to write a variation of it quickly. Indeed, we use OCaml which has a lexical scoping. It is therefore not possible to redefine a small portion of the traversal functions without have to redefine them all.

[1] And believe us when we say we spent some time tuning them and making them more and more clever.

```
 1  let number_of_for program =
 2    let count = ref 0 in
 3    let visitor = object (self)
 4        inherit [_] iter as super
 5
 6        method! visit_for_clause () for_clause =
 7          incr count;
 8          super#visit_for_clause () for_clause
 9      end
10    in
11    visitor#visit_program () program;
12    !count
```

Figure 7.1: Function counting the number of for loops in a Shell script using a visitor

This problem is well-known in software engineering and it enjoys a well-known solution as well: the visitor design pattern [Gamma 1995]. We follow a slightly modified version of this design pattern [Jeannerod et al. 2017b]. We use the OCaml extension *Visitors* [26; Pottier 2017] to generate basic visitors that traverse concrete syntax trees without doing anything. These visitors are objects with a method for each kind of nodes of the CST. These methods expect as arguments the children of the nodes they handle. Writing a custom visitor then simply consists in inheriting from the basic visitors and overriding the methods corresponding to the few syntactic constructs that we are interested in. Writing an analyser consists in writing a visitor that gathers data by traversing the CST of all the scripts and writing a report function that uses these data to highlight the information we are looking for.

Consider Figure 7.1 as an example use for a visitor. It defines an OCaml function `number_of_for`, Line 1. It works as follows.

- `number_of_for` takes the concrete syntax tree of a Shell script as argument, named `program`.
- Line 2, we initialise a reference `count` which will contain the number of `for`-loop in the script.
- Line 3, we define an object `visitor` which inherits from the `iter` class (Line 4). `iter` is a class of objet that traverse the whole concrete syntax tree of a Shell script while doing nothing. It is generated from the type of the concrete syntax tree using *Visitors* [26].
- Line 6, we override the only method of interest to us, `visit_for_clause`. This method is called whenever there is a node of type `for_clause` in a concrete syntax tree. It receives as argument a value `for_clause` which describes the contents of the for clause in question.
- Line 7, our implementation of `visit_for_clause` simply increments the counter and then falls back on the behaviour of `visit_for_clause` in the `iter`[2] class (Line 8), which will keep exploring the concrete syntax tree.
- Line 11, once this visitor is defined, we only need to pass it the whole concrete syntax tree. Once the iteration is done, `count` (Line 12) contains the number of nodes of type `for_clause` in the concrete syntax tree, and thus the number of `for`-loops in the Shell script.

This design patterns allows us to easily write functions that traverse concrete syntax tree to compute various analyses on the shell scripts to which they correspond. We can then easily write functions that consider not one script but a corpus of scripts, iterate over them all and computes statistics over the use

[2] In fact, for such a simple thing as counting `for`-loop, a `reduce` visitor would be more appropriate.

Table 7.1: Builtins which may render analysis impossible

Builtin	Occ.	Files
alias	2	2
eval	42	30

Table 7.2: Sequential control structures

Structure	Occ.	Files
if	56,643	27,122
while	4,045	3,929
until	1	1
for	3,564	2,400
case	6,227	5,296

Table 7.3: Process creation and communication.

Construct	Occ.	Files
subshell	431	356
|	12,225	6,154
trap	32	28
kill	39	35
&	8	7

of various features, utilities, etc.[3]

7.1.2 Gallery of Analyses

Let us quickly list some results of various analyses that we ran on the corpus of all Shell scripts in Debian packages.

Table 7.1 summarises the occurrences of Shell builtins which may render syntactic analysis impossible, as explained in Section 6.1. The alias builtin appears only twice in our corpus, and both occurrences are at the top level. There are 42 occurrences of eval in 30 scripts.

The occurrences of the different sequential utility structures of the Shell are given in Table 7.2. Constructs related to process creation and communication are given in Table 7.3. This table shows that the use of &, which creates an asynchronous execution, is very rare in maintainer scripts. This observation, together with the fact that dpkg does not allow for concurrent execution of maintainer scripts, justifies our decision to ignore concurrency in the modelisation of Shell scripts. The five most frequent simple Shell builtins are listed in Table 7.4. The dot symbol, which is used to include another file in the Shell script, has almost 5,000 occurrences and hence must be handled in our treatment of Shell scripts. It is handled by the conversion mechanism which inlines the included Shell script into the other one.

The construction of the concrete syntax trees allows us to go further than just simple counting of occurrences of reserved words, and do a more *structural* analysis of Shell scripts. For instance, a significant portion of the variables defined in maintainer scripts are in fact constants: We found that in 1,295 out of

[3] An other use case for these visitors can be found in our tool lintshell [28]. Lintshell is a linter which uses this design pattern to define syntactic checks on Shell scripts.

Table 7.4: Simple Shell builtins

Builtin	Occ.	Files
set	30,817	30,579
exit	13,915	8,685
echo	10,770	5,010
true	10,740	3,966
.	4,922	2,900

Table 7.5: The ten most used Unix utilities acting on the file system

Utility	Occ.	Utility	Occ.
[	47,633	find	2,144
which	12,669	xargs	1,907
rm	10,383	test	1,594
grep	5,138	chmod	1,562
read	3,896	chown	1,504

Table 7.6: Options of ln

Options	Occ.	Options	Occ.
-s	333	*(none)*	5
-f -s	210	-f	4
-r -s	31	-S -b -s	4
-f -n -s	10	-b -f -s	3
-s -v	5	*total*	605

Table 7.7: Top 5 Debian-specific utilities

Utility	Occ.	Files
dpkg-maintscript-helper	9,992	3,889
dpkg	6,862	6,518
deb-systemd-helper	4,530	1,029
update-alternatives	3,616	2,350
update-menus	3,363	3,336

Table 7.8: Number of scripts using exotic utilities

Level	Number	%	Level	Number	%
1	693	2.20%	50	3,286	10.44%
2	1,032	3.28%	100	4,058	12.89%
5	1,459	4.63%	200	5,232	16.62%
10	1,794	5.70%	500	8,095	25.71%
25	2,364	7.51%			

2,841 cases (33%), a Shell variable is assigned to only once in a script, and this assignment occurs at top level.

Function definitions are quite frequently used in maintainer scripts: we found 3,455 function definitions in 1,500 files. Only one single function definition is recursive[4]. We also found nine maintainer scripts which contain multiple definitions of the same function, four scripts which define the same function differently in the two branches of an if-then-else and one script containing two *slightly* different definitions for the same function, which could be improved by factorising the large common part of the two definitions.

Our tool provides statistics on the number of occurrences of each possible combination of options. Table 7.6, for example, yields the combination of options observed for the ln utility, together with their number of occurrences. One important conclusion for us is that 596 out of the 605 invocations of ln create symbolic links instead of hard links. The possibility of multiple hard links in a file system are a problem for any formal model of file systems since it means that one has to use acyclic directed graphs as a model, instead of the much simpler trees. The fact that the creation of multiple hard links (ln without the -s option) is rather rare justifies our decision to consider file systems as trees, at least in a first approach.

Table 7.7 yields the 5 most frequently used Debian-specific utilities in our corpus. These utilities are much harder to model than the standard Unix utilities since they typically manipulate the contents of files. The statistics on utility usage help us to focus on the most important ones among these complex utilities.

Finally, the frequency of the different Unix utilities in the corpus also allows us to estimate how many scripts we would have to discard from our analysis if we restricted ourselves to scripts using only frequently used utilities or, conversely, how many more scripts we would support by supporting more utilities. We define, for any natural number i, an *exotic utility of level* i to be a utility that is not found in more than i scripts. For given levels of exotism, we count the number of utilities that are exotic of this level and, more importantly, the number of scripts that do not use any of these utilities. Table 7.8 tells how many scripts use exotic utilities. For instance, 1,794 scripts use at least one utility that occurs in at most 10 scripts.

[4] The function run_utility in the postinst script of the package rt4-extension-assettracker, version 3.0.0-1

7.2 Symbolic Interpretation of Shell Scripts

7.2.1 Symbolic Interpretation of Shell Scripts

In addition to the concrete interpreter, described in Section 6.2.3, we designed and implemented a symbolic interpreter for the CoLiS language, also in Why3 [Becker et al. 2020]. Guided by a proof-of-concept symbolic interpreter for a simple IMP language [Becker & Marché 2020], the main design choices for the symbolic interpreter of CoLiS are:

- Variables are *not* interpreted abstractly: when executing an installation script, the concrete values of the variables are known. On the other hand, the state of the filesystem is not known precisely, and it is represented symbolically using constraints.[5]
- The symbolic engine is generic with respect to the utilities: their specifications in terms of symbolic input/output relations are taken as parameters.
- The number of loop iterations and the number of (recursively) nested function calls is bounded *a priori*, the bound is given by a global parameter set at the interpreter call.

The Why3 code for the symbolic interpreter is annotated with post-conditions to express that it computes an *over-approximation* [Becker & Marché 2020] of the concrete states that are reachable without exceeding the given bound on loop iterations. This means that any concrete trace of execution that can be described by the semantics – and ran by the concrete interpreter[6] – will be present in the symbolic traces. More formally, let us take any feature tree t. The concrete execution of a script on t returns an output tree t'. Now let us take S a set of symbolic states, that is basically a set of threaded constraints on one variable r. Assume one of these constraints is c such that $[r \mapsto t] \models_{\mathcal{FT}} c$. The symbolic execution of the same script on S returns an output set of symbolic states S'. The property of over-approximation guarantees that there is a state $c' \in S'$ such that $[r \mapsto t'] \models_{\mathcal{FT}} c'$. This property is formally proven using automated provers. The OCaml code is automatically extracted from Why3, and provides an executable symbolic interpreter with strong guarantees of soundness with respect to the concrete formal semantics.

Notice that our symbolic engine neither supports parallel executions, nor file permissions or file timestamps. This is another source of over-approximation, but also under-approximation, meaning that our approach can miss bugs whose triggering relies on the former features.

The symbolic interpreter provides a symbolic semantics for the given script: given an initial symbolic state that represents the possible initial shape of the filesystem, it returns a triple of sets of symbolic input/output relations, respectively for normal result, error result (corresponding to non-zero exit code) and result when a loop limit is reached.

Let us consider a toy implementation of this symbolic engine, given in Figure 7.2.

- The main function of the interpreter, `interp`, is given Line 35. takes a command `cmd` in CoLiS and a list of states `states` and returns a pair of list of states, representing the success and the errors of the execution. The type for states is defined Line 6. It is simply a threaded constraint with two variables representing the input and the output roots.
- The interpretation of control structures is not very surprising. As an example, the interpretation of `if cmd1 then cmd2 else cmd3` is given Line 43. It simply consists in interpreting `cmd1` in `states`. This returns two lists of states, `success1` and `errors1` in which one can then interpret

[5] The implementation of the symbolic interpreter is modular and accepts any backend – not only FTS – that has the right properties. A sub-group of the CoLiS project is working on providing a backend for symbolic execution based on tree transducers.

[6] As long as this is feasible while respecting the bound on the loop iterations.

```
type case = (variable × variable) -> threaded-constraint

function spec-utility-call(name : string, args : list of strings)
    : list of cases × list of cases

type state = threaded-constraint × variable × variable

function fresh-variable() : variable

function apply-case-to-state(case : case, state : state) : state or ⊥
  let (t, r, r′) = state
  let r″ = fresh-variable()
  match add-transform-2-threaded(case(r′, r″), t)
  | ⊥ -> return ⊥
  | t′ -> return (t, r, r″)

function apply-cases-to-state(cases : list of cases, state : state)
    : list of states
  match cases
  | ε -> return ε
  | case›cases’ ->
    let states' = apply-cases-to-state(cases', state)
    match apply-case-to-state(case, state)
    | ⊥ -> return states
    | state' -> return state’›states’

function apply-cases-to-states(cases : list of cases,
     states : list of states) : list of states
  match states
  | ε -> return ε
  | state›states’ ->
    return apply-cases-to-state(cases, state)
          » apply-cases-to-states(cases, states')

function interp(cmd : colis, states : list of states)
    : list of states × list of states
  match cmd
  | UtilityCall(name, args) ->
    let (success, errors) = spec-utility-call(name, args)
    return (apply-cases-to-states(success, states),
            apply-cases-to-states(errors, states))

  | IfThenElse(cmd1, cmd2, cmd3) ->
    let (success1, errors1) = interp(cmd1, states)
    let (success2, errors2) = interp(cmd2, success1)
    let (success3, errors3) = interp(cmd3, errors1)
    return (success2»success3, errors2»errors3)

  [...]
```

Figure 7.2: Example code for the symbolic interpreter

cmd2 and cmd3, obtaining four lists of states, success2 and errors2 on one hand and success3 and errors3 on the other hand. The interpretation of the whole structure is then the concatenation of success2 and success3 for success cases and the concatenation of errors2 and errors3 for error cases.

- The interpretation of utility calls is where we get to see the interaction between the symbolic engine and the solvers for FTS. From the point of view of interp, the interpretation of utility calls is fairly simple. It consists in calling a function spec-utility-call generating the specification of the particular utility call. It is then only a matter of applying the success and error specification cases to states, which generates two lists of states, success and error respectively.
- Each specification case returned by spec-utility-call is in fact a function from a pair of variables to a threaded constraint. The type of a case is defined Line 1. This is a way to allow the client code to decide on which root variables to generate the specification cases. The generation of specification cases is described in Section 3.3.
- There are several things to note on the application of specification cases to states. First of all, one applies a list of specification cases to a list of states. This is done by applying each specification case to each state, producing a quadratic explosion. This explosion was mentioned in Section 3.4 and is particularly visible in the example code of the two iteration functions apply-cases-to-states and apply-cases-to-state, defined Lines 27 and 17 respectively. The former takes a list of cases and a list of states and iterates apply-cases-to-state on each given state. The latter takes a list of cases and one state and iterates apply-case-to-state on each given case. We can already note at this point that apply-case-to-state can fail and return $\bot$, in which case the result is simply dropped. This mechanism is in fact very important as this is the place where unsatisfiable states are eliminated.
- Finally, apply-case-to-state, defined Line 10, takes a specification case and a state and tries applying the former to the latter. Let us consider that the state is composed of a threaded constraint t and two root variables r and r' and that the specification case is named case. r represents the root at the beginning of the execution of the script. r' represents the root before the execution of the utility call. Applying case to (t, r, r') consists first in generating a new root variable, Line 12, which will represent the root after the execution of the utility call. This is done by calling a function fresh-variable that is part of the interface of the FTS solver. It generates a fresh variable, that is a variable that has never been previously encountered in the whole execution. In this case, the utility call "happens" between r' and r''. We can therefore instantiate the specification case on these variables by calling case(r', r''). This gives us a threaded constraint which can be added to the current one, t by calling add-transform-2-threaded. The underlying work of add-transform-2-threaded is discussed in Section 5.3. The result of this call can detect an unsatisfiability by returning $\bot$. Such a result means that the given specification case and state are not consistent and that their application should be discarded. If the call does not fail, it returns a new threaded constraint t' equivalent to case(r', r'') $\wedge t$ – but more efficient as it is already the result of some computation by the solver – which can be returned.

7.2.2 An Example

Let us develop this execution by hand on the beginning of the postinst script of the ocaml-base-nox package shown in Figure 6.14. Of course, the symbolic interpreter does not run on the Shell script itself but on the CoLiS script obtained by parsing and conversion, as described in Chapter 6. The CoLiS corresponding to the postinst Shell script of the ocaml-base-nox package is shown in Figure 6.14. This figure is restated here for convenience.

```
begin
  true;

  if test [ '!'; '-e'; '/usr/local/lib/ocaml' ] then
    if mkdir [ '/usr/local/lib/ocaml' ] then
      begin
        chown [ 'root:staff'; '/usr/local/lib/ocaml' ];
        chmod [ '2775'; '/usr/local/lib/ocaml' ]
      end
    fi
  fi;

  [...]

  for i in [ '/usr/lib/ocaml/3.06'; '/etc/ocaml'; '/var/lib/ocaml' ]
  do
    if test [ '-e'; split i '/ld.conf' ] then
      begin
        echo [ 'Removing leftover ' i '/ld.conf' ] ;
        rm [ '-f'; split i '/ld.conf' ] ;
        rmdir [ '--ignore-fail-on-non-empty'; split i ]
      end
    fi
  done
end
```

Figure 6.15: postinst script of the ocaml-base-nox package in CoLiS (excerpt; cleaned up)

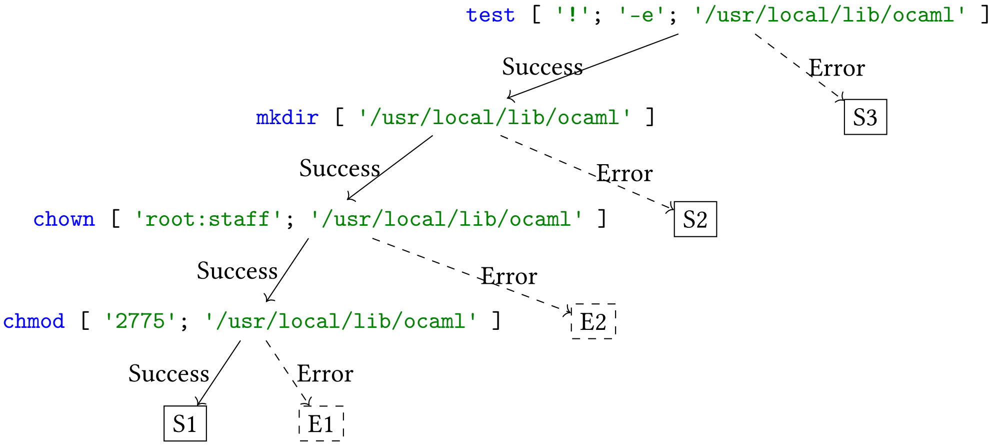
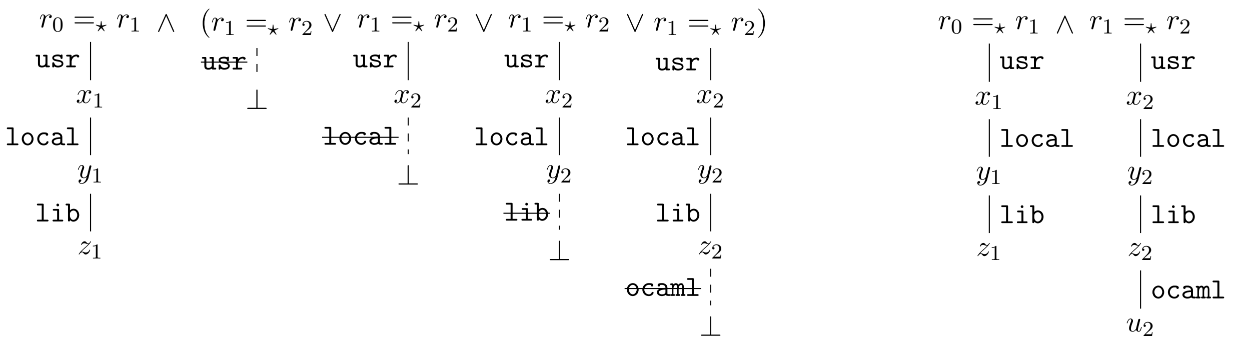

test ['!'; '-e'; '/usr/local/lib/ocaml']
Success
Error
S3
mkdir ['/usr/local/lib/ocaml']
Success
Error
S2
chown ['root:staff'; '/usr/local/lib/ocaml']
Success
Error
E2
chmod ['2775'; '/usr/local/lib/ocaml']
Success
Error
S1
E1

Figure 7.3: Traces of execution of the first command in the postinst script of the ocaml-base-nox package



(a) Success state/s (the threaded constraint is unfolded)

(b) Error state

Figure 7.4: Formulas 7.2 and 7.3

Let us thus develop, by hand, the symbolic execution on this script. The first complex command (Lines 4 to 11) comprises the traces of execution given in Figure 7.3. We simply need to call `interp` on this script with a list of states as argument. This list of states must represent the transformation that was performed before the execution of the script start. Of course, no transformation was actually performed, and we could then provide one single state containing $(r_0 =_\star r_1\texttt{,}\ r_0\texttt{,}\ r_1)$. This however leads to the detection of a lot of potential bugs that have to do with the non-existence of usual directories, like `/usr`. Luckily, the Debian Policy specifies that one can always expect a Debian system to respect the *filesystem hierarchy standard* – or *FHS* for short [36]. This standard requires the existence of numerous directories. The only one of interest for our example is `/usr/local/lib`. Let us therefore call `interp` on the `postinst` script of the `ocaml-base-nox` package with initial state Formula 7.1;

$$((r_0 =_\star r_1 \wedge r_1[\texttt{usr}]x_1 \wedge x_1[\texttt{local}]y_1 \wedge y_1[\texttt{lib}]z_1)\texttt{,}\ r_0\texttt{,}\ r_1) \tag{7.1}$$

The first utility call that gets interpreted is that of `test [ '!'; '-e'; '/usr/local/lib/ocaml' ]`. This call succeeds when `/usr/local/lib/ocaml` does not exist and fails otherwise. The interpreter then gets the semantics of this call from the specifications database and applies it to the one state of Formula 7.1. This gives the two states of Formulas 7.2 and 7.3. Graphical representations can be found in Figure 7.4. The existential quantifiers are left out for readability; every variable except r_0 and r_2 is existentially quantified.

$$\begin{aligned}&(r_0 =_\star r_1 \wedge r_1[\texttt{usr}]x_1 \wedge x_1[\texttt{local}]y_1 \wedge y_1[\texttt{lib}]z_1)\\ &\wedge (r_1 =_\star r_2 \wedge (\neg r_2[\texttt{usr}]\uparrow \to (r_2[\texttt{usr}]x_2 \wedge (\neg x_2[\texttt{local}]\uparrow \to (x_2[\texttt{local}]y_2\\ &\qquad\qquad\qquad\qquad \wedge (\neg y_2[\texttt{lib}]\uparrow \to (y_2[\texttt{lib}]z_2 \wedge z_2[\texttt{ocaml}]\uparrow)))))))\end{aligned} \tag{7.2}$$

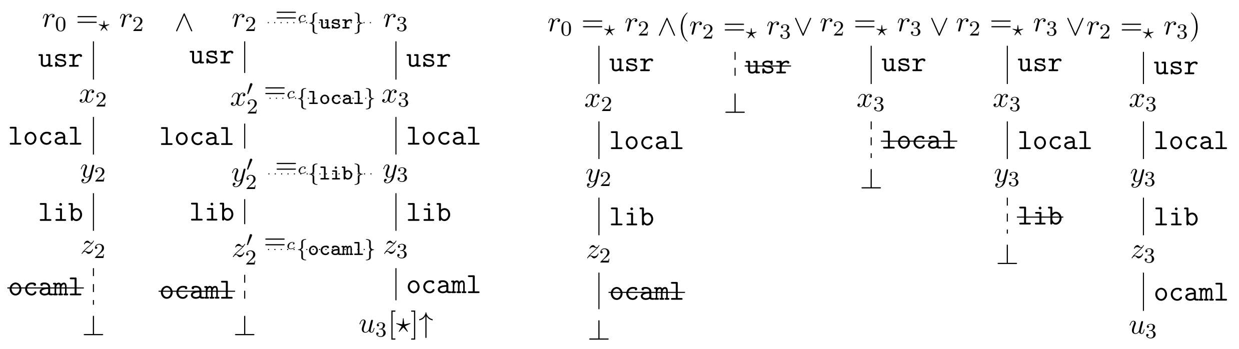
$$\begin{aligned}&(r_0 =_\star r_1 \wedge r_1[\texttt{usr}]x_1 \wedge x_1[\texttt{local}]y_1 \wedge y_1[\texttt{lib}]z_1)\\ &\wedge (r_1 =_\star r_2 \wedge r_2[\texttt{usr}]x_2 \wedge x_2[\texttt{local}]y_2 \wedge y_2[\texttt{lib}]z_2 \wedge z_2[\texttt{ocaml}]u_2)\end{aligned} \tag{7.3}$$

It is already possible for the solver to make these formulas more compact, which gives in the end the two states of Formulas 7.4 and 7.5.

$$r_0 =_\star r_2 \wedge r_2[\texttt{usr}]x_2 \wedge x_2[\texttt{local}]y_2 \wedge y_2[\texttt{lib}]z_2 \wedge z_2[\texttt{ocaml}]\uparrow \tag{7.4}$$

$$r_0 =_\star r_2 \wedge r_2[\texttt{usr}]x_2 \wedge x_2[\texttt{local}]y_2 \wedge y_2[\texttt{lib}]z_2 \wedge z_2[\texttt{ocaml}]u_2 \tag{7.5}$$

The error of `test` leads directly to the successful end of the `if` construct (S3 in Figure 7.3). The success, however, enters the `then` part and we proceed with the interpretation of the utility call `mkdir [ '/usr/local/lib/ocaml' ]`. The interpreter gets its semantics from the specifications database and



(a) Success state

(b) Error state/s (the threaded constraint is unfolded)

Figure 7.5: Formulas 7.6 and 7.7

applies it to the one state of Formula 7.4. This gives the two states of Formulas 7.6 and 7.7. Graphical representations can be found in Figure 7.5. Again, the existential quantifiers are left out for readability; every variable except r_0 and r_3 is existentially quantified.

$$\begin{aligned} & r_0 =_\star r_2 \wedge r_2[\texttt{usr}]x_2 \wedge x_2[\texttt{local}]y_2 \wedge y_2[\texttt{lib}]z_2 \wedge z_2[\texttt{ocaml}]\uparrow \\ & \wedge r_2[\texttt{usr}]x'_2 \wedge x'_2[\texttt{local}]y'_2 \wedge y'_2[\texttt{lib}]z'_2 \wedge z'_2[\texttt{ocaml}]\uparrow \\ & \wedge r_2 =_{c\{\texttt{usr}\}} r_3 \wedge x'_2 =_{c\{\texttt{local}\}} x_3 \wedge y'_2 =_{c\{\texttt{lib}\}} y_3 \wedge z'_2 =_{c\{\texttt{ocaml}\}} z_3 \\ & \wedge r_3[\texttt{usr}]x_3 \wedge x_3[\texttt{local}]y_3 \wedge y_3[\texttt{lib}]z_3 \wedge z_3[\texttt{ocaml}]u_3 \wedge u_3[\star]\uparrow \end{aligned} \tag{7.6}$$

$$\begin{aligned} & (r_0 =_\star r_2 \wedge r_2[\texttt{usr}]x_2 \wedge x_2[\texttt{local}]y_2 \wedge y_2[\texttt{lib}]z_2) \\ & \wedge (r_2 =_\star r_3 \wedge (\neg r_3[\texttt{usr}]\uparrow \to (r_3[\texttt{usr}]x_3 \wedge (\neg x_3[\texttt{local}]\uparrow \to (x_3[\texttt{local}]y_3 \\ & \qquad \wedge (\neg y_3[\texttt{lib}]\uparrow \to (y_3[\texttt{lib}]z_3 \wedge z_3[\texttt{ocaml}]u_3))))))) \end{aligned} \tag{7.7}$$

The solver can make these formulas more compact. In fact, while doing so, it will detect that the error case, leading to a successful end of the inner `if` construct (S2 in Figure 7.3), is simply unsatisfiable. In the end, only the success state of Formula 7.8 remains

$$\begin{aligned} & r_0[\texttt{usr}]x_0 \wedge x_0[\texttt{local}]y_0 \wedge y_0[\texttt{lib}]z_0 \wedge z_0[\texttt{ocaml}]\uparrow \\ & \wedge r_0 =_{c\{\texttt{usr}\}} r_3 \wedge x_0 =_{c\{\texttt{local}\}} x_3 \wedge y_0 =_{c\{\texttt{lib}\}} y_3 \wedge z_0 =_{c\{\texttt{ocaml}\}} z_3 \\ & \wedge r_3[\texttt{usr}]x_3 \wedge x_3[\texttt{local}]y_3 \wedge y_3[\texttt{lib}]z_3 \wedge z_3[\texttt{ocaml}]u_3 \wedge u_3[\star]\uparrow \end{aligned} \tag{7.8}$$

The interpretation will then proceed with `chown [ 'root:staff'; '/usr/local/lib/ocaml' ]` and `chmod [ '2775'; '/usr/local/lib/ocaml' ]`. Since users, groups and permissions are abstracted away in our model, these two utility calls are equivalent to a test of existence of `/usr/local/lib/ocaml`. Of course, since we come from the success case of `mkdir [ '/usr/local/lib/ocaml' ]`, both these calls will succeed. Their success case will not change anything to Formula 7.8 and their error case/s, leading to errored end of the `if` constructs (E1 and E2 in Figure 7.3), will turn out to be unsatisfiable.

The result of the interpretation of this first part of the script is that only S1 and S3 are reachable – all the other traces are not. Moreover, the interpreter only carries one state for each of these traces, far from the worst-case exponential explosion. The symbolic execution benefits from the intertwining of steps of execution of the solver. Indeed, without that, unreachable traces would still be explored, leading to a loss of time in the execution. In this script, for instance, the trace ending in S2 would otherwise carry its execution in the rest of the script, leading to the exploration of dozen useless traces. Of course, this is only a gain of speed if running the solver again after every step only takes a short amount of time. This is where the incrementallity of our solver plays a crucial role.

Table 7.9: Bugs found between 2016 and 2019 in Debian *sid* distributions

Bugs	Closed	Detected by	Reports	Examples
95	56	parser	[3]	not using `-e` mode
6	4	parser & manual	[7]	unsafe or non-POSIX constructs
34	24	corpus mining	[2; 5]	wrong options, mixed redirections
9	7	conversion	[4]	wrong test expressions
5	2	symbolic execution	[7; 8; 10]	try to remove a directory with `rm`
3	3	formalisation	[6]	bug in `dpkg-maintscript-helper`
151	92			

7.3 Analysing Installation Scenario of Corpuses of Debian Packages

7.3.1 Coverage of the case study

The main experimental result of our thesis and of the CoLiS project is the analysis of full corpuses of Debian packages. We execute the analysis on a machine equipped with 40 hyperthreaded Intel Xeon CPU @ 2.20GHz, and 750GB of RAM. To obtain a reasonable execution time, we limit the processing of one script to 60 seconds and 8GB of RAM. The time limit might seem low, but the experience shows that the few scripts (in 30 packages) that exceed this limit actually require hours of processing because they make a heavy use of `dpkg-maintscript-helper`. On our corpus of 12,592 packages with 28,814 scripts, the analysis runs in about half an hour.

All of those scripts that are syntactically correct with respect to the POSIX standard (99.9%) are parsed successfully by our parser. The conversion of the parsed scripts into our intermediary language CoLiS succeeds for 77% of them; the translation fails mainly because of the use of globs, variables with parameters and advanced uses of redirections.

Our toolchain then attempts to run 113,328 scenarios (12,592 packages with scripts, 9 scenarios per package). Out of those, 45,456 scenarios (40%) are run completely and 13,149 (12%) partially. This is because scenarios have several branches and although a branch might encounter failure, we try to get some information on execution of other branches. For the same reason, one scenario might encounter several failures. In total, we encounter 67,873 failures. The origins of failures are multiple, but the two main ones are:

- trying to execute a scenario that includes a script that we cannot convert (28% of failures),
- or the scripts might use utilities unsupported by our tools, or unsupported features of supported utilities (71% of failures).

Among the scenarios that we manage to execute at least partially, 19 reach an unexpected end state. These are potential bugs. We have examined them manually to remove false positives due to approximations done by our methodology or the toolchain. We discuss in Section 7.3.2 the main classes of true bugs revealed by this process.

7.3.2 Bugs found

We ran our toolchain [29; 24; 11; 23] on several snapshots of the Debian *sid* distribution taken between 2016 and 2019, the latest one being October 6, 2019. We reported over this period a total of 151 bugs to the Debian Bug Tracking System [35]. Some of them have immediately been confirmed by the package maintainer (for instance, [9]), and 92 of them have already been resolved.

Table 7.9 summarises the main categories of bugs we reported. Simple lexical analysis already detects 95 violations of the Debian Policy, for instance scripts that do not specify the interpreter to be used, or that do not use the `-e` mode [3]. The Shell parser (Section 6.1) detects 3 scripts that use Shell constructs not allowed by the POSIX standard, or in a context where the POSIX standard states that the behaviour is undefined [7]. There are also 3 miscellaneous bugs, like using unsafe Shell constructs. The mining tool (Section 7.1) detects 5 scripts that invoke Unix utilities with wrong options and 29 scripts that mix up redirection of standard-output and standard-error. The conversion from the Shell to the CoLiS language (Section 6.2) detects 9 scripts with wrong test expressions [4]. These may stay unnoticed during superficial testing since the Shell confuses, when evaluating the condition of an if-then-else, an error exception with the Boolean value *False*.

Inspection of the symbolic semantics extracted by the symbolic execution finds 5 scripts with semantic errors. Among these is the bug [9] of the package `rancid-cgi` already explained in Section 1.3.

We found 3 bugs during the formalisation of Debian tools. These include in particular a bug [6] in the `dpkg-maintscript-helper` function which is used 10,306 times in our corpus of maintainer scripts, and was fixed in the meantime.

We found that identifying bugs in maintainer scripts always requires human examination. Automated tools allow to point out potential problems in a large corpus, but deciding whether such a problem actually deserves a bug report, and of what severity level, requires some experience with the Debian processes. This is most visible with semantic bugs in scripts, since an error exit code does not imply that there is a bug. Indeed, if a script detects a situation it cannot handle then it *must* signal an error and produce a useful error message. Deciding whether a detected error case is justified or accidental requires human judgement.

Filling bug reports demands some caution, and observance of rules and common practices in the community. For instance, the Debian Developers Reference [14] requires approval by the community before so-called *mass bug filling*. Consequently, we always sought for advice before sending batches of bugs, either on the Debian developers mailing list, or during Debian conferences.

Chapter 8

Conclusion

8.1 Contributions

Throughout this thesis, we presented work revolving around the case study of bringing formal program analysis techniques to the quality assurance of Debian packages, and their maintainer scripts in particular. This includes various aspects, ranging from theoretical research to more concrete implementation and testing considerations, not forgetting interpretation and modelisation of standards written in natural language.

Our main contribution lies in our work on decision procedures for FTS. This comprises the design of a decision procedure for first-order formula and a result of decidability of the first-order theory of FTS which was an open problem before this work took place. This also covers more practical considerations as the goal is to use FTS – or a chosen subset – in our tool. These considerations include work on an efficient way to handle the negated similarity predicates, a way to formalise an intermediary step between systems of transformation rules and an implementation and support for threaded constraints.

Another important part of our work lies in the modelisation of Unix filesystems and utilities and of POSIX Shell. This comprises the interpretation of informal standards in natural language – POSIX and the Debian Policy – and reflection on the abstractions that make sense. The latter includes a statistic analysis of packages to discover common uses within the maintainer scripts of Debian.

Finally, our work comprises concrete contributions to the implementation of a toolchain able to analyse Debian packages. This encompasses the implementation of a parser and conversion chain from Shell to our intermediary language as well as the implementation of the aforementioned solver for a subset of FTS. This comprises questions of validation of these tools and the establishment of testing procedures for the various unformalised components of the toolchain (eg. by comparing Morbig to `dash` and CoLiS to Shell). Finally, this includes work to scale our analysis to thousands of packages in a reasonable time, and work to return human readable reports. The result is the discovering and report of a total of 151 bugs to the Debian Bug Tracking System

8.2 Limitations & Perspectives

8.2.1 About a Solver for (Threaded) Constraints of FTS

A first and most obvious limitation lies in the actual implementation of a solver for FTS. The toolchain of our project includes an implementation close to that described in Chapter 5. Its support of FTS is however partial, in particular in the handling of negative literals. The implementation of threaded constraints is also partial and only the specific threads required by CoLiS are supported (see Section 5.3.4).

One of our short term goals would be to write an implementation for the solver that follows exactly what is described Chapter 5, with full support for FTS, pointed constraints and threaded constraints. This implementation would allow us to measure the efficiency of the various optimisations presented in this thesis. We could for instance try to assess the improvement brought by:

- the use of $\mathcal{R}_2$ instead of $\mathcal{R}_1$,
- the extension of $\mathcal{R}_2$ with extra cleanup rules (see Section 5.1.4),
- different orders in which pointed constraints handle their literals (see Section 5.2.4),
- the use of threaded constraints in the solver (see Section 5.3),
- different formulations of the same formulas using different threads – eg. `noresolve` as described in Section 5.3.1 or as currently implemented in `colis-language` (see Section 5.3.4).

This requires the definition of a corpus of test formulas, covering all the aspects of FTS, on which to compare efficiency and run benchmarks. Of course, a particularly interesting (for us) subset of formulas will be the one generated by symbolic execution of Shell scripts in our toolchain.

8.2.2 About a Solver for First-Order Formulas of FTS

There is currently no implementation of a solver for formulas of FTS that are not Σ_1. It would be easy to add support – although a very inefficient one – for any first-order formulas by simply following the implementation described in Chapter 4. This would already allow us to check automatically a lot of properties as long as they can be expressed as formulas with only few quantifier alternations (see for instance Section 8.2.4).

Longer term research could involve the development of more efficient algorithm for deciding first-order formulas, or at least interesting subsets of such formulas. It could be interesting to investigate algorithms for deciding entailment of any Σ_1-formulas, for instance.[1]

8.2.3 About the Expressivity of FTS

Some future work could involve increasing the expressivity of FTS to support more features of Unix utilities. This includes the handling of utility arguments, standard inputs and outputs, return codes, etc. directly in the logic instead of concretely in the symbolic engine.

This also includes the handling of more complex transformations like that of `find` or like the interleavings of trees as created by `cp -R`. Expressing properties about this interleaving requires to be able to handle predicates that mention the union of two feature trees and the inclusion of a feature tree in and other one (see Section 3.4.1).

FTS could also be extended by supporting features as first class objects. Although the first-order theory of such logics is not decidable, it can be possible to recover decidability by limiting the quantification over features. One can thus hope that it is possible to decide the satisfiability of Σ_1-formulas at least. In turn, and by adding limited constraints over strings, this could for instance allow to model the use of *globs* – as in the command `rm *.tex` which removes all the files whose name ends in `.tex` from the current working.

[1] Section 5.3 uses the satisfiability of Σ_1-formulas to obtain a test of entailment $\phi \models \psi$ by checking for unsatisfiability of $\phi \wedge \neg\psi$. This means it can currently only check the entailment of a Π_1-formula ψ by a Σ_1-formula ϕ, and probably not in such an efficient way.

FTS could also be extended to support paths. This change would in particular allow us to compare the performances brought by such an extension with the performances of threaded constraints to express the same formulas (see Section 5.3).

Finally, FTS could be extended to support paths as first class objects. This could be an important step towards the modelisation of symbolic links in the logic. Indeed, the resolution of a path `/usr` from a variable r can then succeed either if there exists a variable x such that $r[\texttt{usr}]x$, or if there exists a path p such that `usr` is a symbolic link in r pointing to p.

8.2.4 About Specifications

Although we have been careful while writing them, there is no guarantee that the specifications we gave to Unix utilities are correct models of what they are actually doing. We see two directions which we could follow in order to improve the quality of these specifications.

- Firstly, we documented some properties – under the form of formulas of FTS – which we expect from specifications in Section 3.4.1. Some of them can be checked by a solver for any Σ_1-formulas (completeness, determinism, functionality) and some require at least a solver for Π_2-formulas (coherence, totality). With such solvers (which we do not currently have; see Sections 8.2.1 and 8.2.2), we could automatically check and report on utilities whose specification does not respect these properties. This would increase our confidence in the fact that our specifications are correct.

- Secondly, we could also compare our modelled utilities with the actual ones from GNU by generating tests based on the specifications. This would allow us to check whether our specifications matches with actual used implementations. In a second step, that would also allow us to generate test batteries to check whether a given implementation is POSIX-compliant.

8.2.5 About the Coverage of our Toolchain

The current state of our toolchain has a number of limitations. The most visible limitation lies in the number of maintainer scripts that are accepted by our toolchain and the number of scenarios that our tool manages to run. As of October 6, 2019, 77% of maintainer scripts get successfully converted to the intermediary language CoLiS and 40% of scenarios are ran completely – that is without problems – and 12% partially. Most of the problems met during the execution of scenarios come from utilities that are unsupported by our tool (70% of problems) and scripts that have not been converted (28% of problems). In order to increase the coverage of our analysis, there are several points which we can improve.

- Firstly, our tool can support more Shell scripts. This can be achieved by improving the conversion to handle specific expressions in an ad hoc and subtle way. This can also be done by extending the CoLiS language so that it supports other aspects of the Shell that are used in maintainer scripts.

- Secondly, our tool should support more Unix utilities or more aspects of the currently supported utilities. For some of them, this is only a matter of time spent reading their description and writing their specification. For some others, the current expressivity of FTS is not enough (see Section 8.2.3).

- Thirdly and finally, our model could be extended to cover more features of Unix filesystems, such as permissions, file contents, hard and symbolic links, etc.

8.2.6 About Finding More Bugs with our Toolchain

Currently, the semantic bugs that we find only have to do with exit statuses in scenarios. For instance, in the case of our running example, `rancid-cgi`, we found a bug because the installation of this package could reach the “Not-Installed” state of `dpkg`.

We however aim at finding bugs that occur even when the exit status of `dpkg` is legit. For instance:

- the installation of a package should never modify the home of the users,
- maintainer script should be idempotent – that is running it twice should give the same result as running it once[2] –,
- installing and removing a package should leave the filesystem unchanged[3],
- updating a package should leave the filesystem in the same state as if the new version was installed from scratch,
- etc.

For instance, the fact that the semantics for the installation of a package does not modify the home of the users can be expressed as a simple entailment. The semantics of an installation being a Σ_1-formula $\phi(r, r')$, the entailment $\phi(r, r') \models r =_{\{\texttt{home}\}} r'$ indeed expresses that `/home` is untouched.

As another example, the idempotency of the semantics of a maintainer script – which is also a Σ_1-formula $\phi(r, r')$ – can be expressed as the Π_2-formula $\forall r, r', r'' \cdot ((\phi(r, r') \wedge \phi(r', r'')) \leftrightarrow \phi(r, r''))$[4]. A weaker form can be expressed as the entailment $(\phi(r, r') \wedge \phi(r', r'')) \models r' =_\star r''$.

As in Section 8.2.4, checking these properties therefore requires the existence of solvers for the entailment of Σ_1-formulas or even for the validity of Π_2, which we do not currently have (see Sections 8.2.1 and 8.2.2).

8.2.7 About Finding Less Bugs with our Toolchain

If our tool allows to pinpoint problems, it still requires an important human intervention to decide whether they are reasonable or whether they should be considered to be bugs. These limitations in the automation can be mitigated by letting the tool automatically run some checks on the potential bugs.

- Firstly, we can hope that checking the properties on the semantics of maintainer scripts described in Section 8.2.6 would make our summary report better at showing likely bugs. In our running example `rancid-cgi`, for instance, we found a bug where the installation of this package could reach the “Not-Installed” state in `dpkg` while having modified the filesystem. This bug was however buried under other packages that would reach the “Not-Installed” state without performing any transformation. Such false positives could easily be automatically detected by asking to a solver whether their semantics implies that the input and the output are equal or not. Such a check would then increase the confidence in the fact that reported problems are bugs by filtering out false positives.

- Secondly, we could automatically extract from the semantics of maintainer scripts steps to reproduce the potential problems that are reached. This would allow for better and more understandable bug reporting. This would also allow for an automatic reproduction of bugs in real condition in order to remove false positives.

8.2.8 About the Accessibility of our Toolchain

We strongly believe in making tools like our toolchain available to anyone in order to actually make them useful. All our tools are open source and can be easily found online [13]. We do not consider that enough

[2] In fact, the idempotency in maintainer scripts means that if the first run is successful, then the second run should just ensure that everything is the way it ought to be, and if the first run failed, then the second call should merely do the things that were left undone the first time [22, Section 6.2].

[3] Actually, this would be installing and purging a package. The removal of a package may leave configuration files in `/etc`.

[4] Actually, as said in footnote 2, the problem of idempotency is more complex than that. This would however be a nice first approximation.

and we would therefore want to let our toolchain run regularly on all the Debian packages, to make the reports available easily, or even to fill in bug reports automatically.

We would also want to make this tool easy to use, so that anyone can check their own packages before upload, either on their own machine or sending their package to website and getting a report[5].

8.2.9 About the Generalisation of our Toolchain

Finally, it is tempting to generalise the work of the CoLiS project. We see three ways to generalise, of increasing complexity.

- Firstly, we could generalise our analysis to other packages than that of Debian. Debian being the base of a lot of derived distribution (eg. the Ubuntu family), there is a huge amount of packages that are not supported by Debian but that are installable with `dpkg`. It would be interesting to run our tool on such corpuses of packages.
- Secondly, we could generalise our analysis to other package managers than `dpkg`. A lot of package managers indeed rely on mechanisms that are similar. Shell scripts often have a preponderant position in such tools, in similar environment and similar bugs are therefore to be expected.
- Thirdly, we could generalise our analysis to any Shell script. This seems like a much more complex work as we would loose the whole (convenient) context of Debian. This means that we should expect a lot more linguistic features used in Shell scripts than what we currently have to deal with.

[5] One of our dreams would be for the tool to run easily in the browser, which is not so complicated with `js_of_ocaml` for instance.

Appendices

References

This bibliography does not contain the miscellaneous elements. See later for these elements.

[Abate et al. 2012] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. “Dependency solving: A separate concern in component evolution management”. In: *Journal of Systems and Software*, volume 85, issue 10, October 2012, pages 2228–2240.
URL: https://doi.org/10.1016/j.jss.2012.02.018.

[Abramatic et al. 2018] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. “Building the universal archive of source code”. In: *Communications of the ACM*, volume 61, issue 10, September 2018, pages 29–31.
URL: https://doi.org/10.1145/3183558.

[Aho et al. 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. ISBN: 0-201-10088-6.
URL: https://www.worldcat.org/oclc/12285707.

[Aho et al. 2006] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[Aït-Kaci 1986] Hassan Aït-Kaci. “An algebraic semantics approach to the effective resolution of type equations”. In: *Theoretical Computer Science*, volume 45, issue, 1986, pages 293–351. ISSN: 0304-3975.
URL: https://doi.org/10.1016/0304-3975(86)90047-2.

[Aït-Kaci et al. 1994] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. “A Feature-Based Constraint System for Logic Programming with Entailment”. In: *Theoretical Computer Science*, volume 122, issue 1-2, January 3, 1994, pages 263–283.

[Aït-Kaci & Nasr 1986] Hassan Aït-Kaci and Roger Nasr. “LOGIN: A Logic Programming Language with Built-In Inheritance”. In: *Journal of Logic Programming*, volume 3, issue 3, October 1986, pages 185–215.
URL: https://doi.org/10.1016/0743-1066(86)90013-0.

[Aït-Kaci & Nasr 1989] Hassan Aït-Kaci and Roger Nasr. “Integrating Logic and Functional Programming”. In: *Lisp and Symbolic Computation*, volume 2, issue 1, February 1989, pages 51–89.
URL: https://doi.org/10.1007/BF01806313.

REFERENCES

[Aït-Kaci & Podelski 1993] Hassan Aït-Kaci and Andreas Podelski. "Towards a Meaning of LIFE". In: *Journal of Logic Programming*, volume 16, issue 3, July–August 1993, pages 195–234.
URL: https://doi.org/10.1016/0743-1066(93)90043-G.

[Backofen 1995] Rolf Backofen. "A Complete Axiomatization of a Theory with Feature and Arity Constraints". In: *Journal of Logic Programming*, volume 24, issue 1-2, July–August 1995, pages 37–71.
URL: https://doi.org/10.1016/0743-1066(95)00033-G.

[Backofen & Smolka 1995] Rolf Backofen and Gert Smolka. "A Complete and Recursive Feature Theory". In: *Theoretical Computer Science*, volume 146, issue 1-2, July 24, 1995, pages 243–268.
URL: https://doi.org/10.1016/0304-3975(94)00188-O.

[Backofen & Treinen 1998] Rolf Backofen and Ralf Treinen. "How to Win a Game with Features". In: *Information and Computation*, volume 142, issue 1, April 10, 1998, pages 76–101.
URL: https://doi.org/10.1006/inco.1997.2691.

[Becker et al. 2019] Benedikt Becker, Claude Marché, Nicolas Jeannerod, and Ralf Treinen. *Revision 2 of CoLiS language: formal syntax, semantics, concrete and symbolic interpreters.* Technical Report. HAL Archives Ouvertes, October 2019.
URL: https://hal.inria.fr/hal-02321743.

[Becker et al. 2020] Benedikt Becker, Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, and Ralf Treinen. "Analysing installation scenarios of Debian packages". In: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II.* Edited by Armin Biere and David Parker. Volume 12079. Lecture Notes in Computer Science. Springer, April 17, 2020, pages 235–253.
URL: https://doi.org/10.1007/978-3-030-45237-7_14.

[Becker & Marché 2020] Benedikt Becker and Claude Marché. "Ghost Code in Action: Automated Verification of a Symbolic Interpreter". In: *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers.* Edited by Supratik Chakraborty and Jorge A. Navas. Volume 12031. Lecture Notes in Computer Science. Springer, March 14, 2020, pages 107–123.
URL: https://doi.org/10.1007/978-3-030-41600-3_8.

[Bobot et al. 2015] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. "Let's verify this with Why3". In: *International Journal on Software Tools for Technology Transfer*, volume 17, issue 6, November 2015, pages 709–727.
URL: https://doi.org/10.1007/s10009-014-0314-5.

[Comon & Lescanne 1989] Hubert Comon and Pierre Lescanne. "Equational Problems and Disunification". In: *Journal of Symbolic Computation*, volume 7, issue 3-4, March–April 1989, pages 371–425.
URL: https://doi.org/10.1016/S0747-7171(89)80017-3.

[Cook 1971] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA.* Edited by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. ACM, May 1971, pages 151–158.
URL: https://doi.org/10.1145/800157.805047.

[Davey & Priestley 2002] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order.* 2nd edition. Cambridge University Press, April 18, 2002. ISBN: 9780511809088.
URL: http://doi.org/10.1017/CBO9780511809088.

[Gamma 1995] Erich Gamma. *Design patterns: elements of reusable object-oriented software.* Pearson Education India, 1995.

[Garey & Johnson 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979. ISBN: 0-7167-1044-7.

[Garfinkel et al. 1994] Simson Garfinkel, Daniel Weise, and Steven Strassmann. *The UNIX-HATERS Handbook.* 1994. ISBN: 1-56884-203-1.

[Greenberg 2017] Michael Greenberg. “Understanding the POSIX Shell as a Programming Language”. In: *Off the Beaten Track 2017.* Paris, France, January 2017.

[Greenberg 2018a] Michael Greenberg. “The POSIX shell is an interactive DSL for concurrency”. In: *DSLDI 2018.* 2018.

[Greenberg 2018b] Michael Greenberg. “Word Expansion Supports POSIX Shell Interactivity”. In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming.* Programming’18 Companion. Nice, France: Association for Computing Machinery, April 2018, pages 153–160. ISBN: 9781450355131.
URL: https://doi.org/10.1145/3191697.3214336.

[Greenberg & Blatt 2019] Michael Greenberg and Austin J. Blatt. “Executable Formal Semantics for the POSIX Shell”. In: *Proceedings of the ACM on Programming Languages*, volume 4, issue POPL, December 2019.
URL: https://doi.org/10.1145/3371111.

[Hodges 1993] Wilfrid Hodges. *Model theory.* Volume 42. Encyclopedia of mathematics and its applications. Cambridge University Press, 1993. ISBN: 978-0-521-30442-9.

[Jeannerod et al. 2017a] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. “A Formally Verified Interpreter for a Shell-Like Programming Language”. In: *Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers.* Edited by Andrei Paskevich and Thomas Wies. Volume 10712. Lecture Notes in Computer Science. Springer, December 15, 2017, pages 1–18.
URL: https://doi.org/10.1007/978-3-319-72308-2_1.

[Jeannerod et al. 2017b] Nicolas Jeannerod, Yann Régis-Gianas, and Ralf Treinen. *Having Fun With 31.521 Shell Scripts.* Technical Report. HAL Archives Ouvertes, 2017.
URL: https://hal.archives-ouvertes.fr/hal-01513750.

[Jeannerod et al. 2019] Nicolas Jeannerod, Yann Régis-Gianas, Claude Marché, Mihaela Sighireanu, and Ralf Treinen. *Specification of UNIX Utilities.* Technical Report. HAL Archives Ouvertes, October 2019.
URL: https://hal.inria.fr/hal-02321691.

[Jeannerod & Treinen 2018] Nicolas Jeannerod and Ralf Treinen. “Deciding the First-Order Theory of an Algebra of Feature Trees with Updates”. In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings.* Edited by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Volume 10900. Lecture Notes in Computer Science. Springer, June 30, 2018, pages 439–454.
URL: https://doi.org/10.1007/978-3-319-94205-6_29.

[Levine et al. 1992] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. 1992.

[Maher 1988] Michael J. Maher. “Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees”. In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS ’88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 1988, pages 348–357.
URL: https://doi.org/10.1109/LICS.1988.5132.

[Malcev 1971] Anatolií Ivanovič Mal’cev. “Axiomatizable Classes of Locally Free Algebras of Various Types”. In: *The Metamathematics of Algebraic Systems*. Edited by Anatolií Ivanovič Mal’cev. Volume 66. Studies in Logic and the Foundations of Mathematics. Elsevier, 1971. Chapter 23, pages 262–281.
URL: https://doi.org/10.1016/S0049-237X(08)70560-3.

[Mancinelli et al. 2006] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. “Managing the Complexity of Large Free and Open Source Package-Based Software Distributions”. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*. IEEE Computer Society, 2006, pages 199–208.
URL: https://doi.org/10.1109/ASE.2006.49.

[Mazurak & Zdancewic 2007] Karl Mazurak and Steve Zdancewic. “Abash: Finding Bugs in Bash Scripts”. In: *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*. PLAS ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pages 105–114. ISBN: 9781595937117.
URL: https://doi.org/10.1145/1255329.1255347.

[Ntzik & Gardner 2015] Gian Ntzik and Philippa Gardner. “Reasoning about the POSIX file system: local update and global pathnames”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Edited by Jonathan Aldrich and Patrick Eugster. ACM, October 2015, pages 201–220.
URL: https://doi.org/10.1145/2814270.2814306.

[Pottier 2017] François Pottier. “Visitors Unchained”. In: *Proceedings of the ACM on Programming Languages*, volume 1, issue ICFP, August 2017.
URL: https://doi.org/10.1145/3110272.

[Régis-Gianas et al. 2020] Yann Régis-Gianas, Nicolas Jeannerod, and Ralf Treinen. “Morbig: A Static parser for POSIX shell”. In: *Journal of Computer Languages*, volume 57, issue, April 2020, page 100944.
URL: https://doi.org/10.1016/j.cola.2020.100944.

[Saraswat & Rinard 1990] Vijay A. Saraswat and Martin C. Rinard. “Concurrent Constraint Programming”. In: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*. Edited by Frances E. Allen. ACM Press, 1990, pages 232–245.
URL: https://doi.org/10.1145/96709.96733.

[Smolka 1992] Gert Smolka. “Feature-Constraint Logics for Unification Grammars”. In: *Journal of Logic Programming*, volume 12, issue 1-2, January 1992, pages 51–87.
URL: https://doi.org/10.1016/0743-1066(92)90039-6.

[Smolka 1993] Gert Smolka. "Residuation and Guarded Rules for Constraint Logic Programming". In: *Constraint Logic Programming: Selected Research*. Edited by Frédéric Benhamou and Alain Colmerauer. The MIT Press, 1993, pages 405–419.

[Smolka et al. 1993] Gert Smolka, Martin Henz, and Jörg Würtz. "Object-Oriented Concurrent Constraint Programming in Oz". In: *Grundlagen und Anwendungen der Künstlichen Intelligenz, 17. Fachtagung für Künstliche Intelligenz, Humboldt-Universität zu Berlin, 13.-16. September 1993, Proceedings*. Edited by Otthein Herzog, Thomas Christaller, and Dieter Schütt. Informatik Aktuell. Springer, 1993, pages 44–59.

[Smolka 1995] Gert Smolka. "The Oz Programming Model". In: *Computer Science Today: Recent Trends and Developments*. Edited by Jan van Leeuwen. Volume 1000. Lecture Notes in Computer Science. Springer, 1995, pages 324–343.
URL: https://doi.org/10.1007/BFb0015252.

[Smolka & Treinen 1994] Gert Smolka and Ralf Treinen. "Records for Logic Programming". In: *Journal of Logic Programming*, volume 18, issue 3, April 1994, pages 229–258.
URL: https://doi.org/10.1016/0743-1066(94)90044-2.

[Stump et al. 2001] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. "A Decision Procedure for an Extensional Theory of Arrays". In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 2001, pages 29–37.
URL: https://doi.org/10.1109/LICS.2001.932480.

[Treinen 1993] Ralf Treinen. "Feature Constraints with First-Class Features". In: *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*. Edited by Andrzej M. Borzyszkowski and Stefan Sokolowski. Volume 711. Lecture Notes in Computer Science. Springer, 1993, pages 734–743.
URL: https://doi.org/10.1007/3-540-57182-5_64.

[Treinen 1997] Ralf Treinen. "Feature Trees over Arbitrary Structures". In: *Specifying Syntactic Structures*. Edited by Patrick Blackburn and Maarten de Rijke. CSLI Publications and FoLLI, 1997. Chapter 7, pages 185–211.

[Turing 1937] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society*, volume s2-42, issue 1, January 1937, pages 230–265. ISSN: 0024-6115.
URL: https://doi.org/10.1112/plms/s2-42.1.230.

[Vorobyov 1996] Sergei G. Vorobyov. "An Improved Lower Bound for the Elementary Theories of Trees". In: *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*. Edited by Michael A. McRobbie and John K. Slaney. Volume 1104. Lecture Notes in Computer Science. Springer, 1996, pages 275–287.
URL: https://doi.org/10.1007/3-540-61511-3_91.

References – Miscellaneous

[1] Debian Bug #431131. *cmigrep: broken emacsen-install script*. June 2007.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=431131.

[2] Debian Bug #841934. *dibbler-server: postinst contains invalid command.* October 2016.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=841934.

[3] Debian Bug #866249. *authbind: maintainer script(s) not using strict mode.* June 2017.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=866249.

[4] Debian Bug #900493. *python3-neutron-fwaas-dashboard: incorrect test in postrm.* May 2018.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=900493.

[5] Debian Bug #908189. *dict-freedict-all: postinst script has a wrong redirection.* September 2018.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=908189.

[6] Debian Bug #922799. *[dpkg-maintscript-helper] bug in finish_dir_to_symlink.* February 2019.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=922799.

[7] Debian Bug #925006. *preinst script not POSIX compliant.* March 2019.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=925006.

[8] Debian Bug #929706. *sgml-base: preinst may fail *silently*.* May 2019.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=929706.

[9] Debian Bug #942388. *rancid-cgi: preinst may fail and not rollback a change.* October 2019.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=942388.

[10] Debian Bug #942392. *ndiswrapper: when "postrm purge" fails it may have deleted some config files.* October 2019.
URL: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=942392.

[11] Benedikt Becker, Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, and Ralf Treinen. *`colis-language`: a symbolic analyser for shell scripts.* GitHub Repository.
URL: https://archive.softwareheritage.org/swh:1:dir:5af8ddcd130f0ec7b63cee9686b2e7a394805a8

[12] Richard Braakman, Josip Rodin, Julian Gilbey, and Mark Hobley. *`checkbashisms`.*
URL: https://sourceforge.net/projects/checkbaskisms/.

[13] *CoLiS – ANR project ANR-15-CE25-0001.* GitHub Organisation.
URL: https://github.com/colis-anr.

[14] Developer's Reference Team. *Debian Developers Reference.* October 2019.
URL: https://www.debian.org/doc/manuals/developers-reference/.

[15] GitHub User `ginoputrino`. *Install script does `rm -rf /usr` for ubuntu.* GitHub Issue `MrMEEE/bumblebee` #123. May 2011.
URL: https://github.com/MrMEEE/bumblebee-Old-and-abbandoned/issues/123.

[16] Michael Greenberg and Austin J. Blatt. *Smoosh, the Symbolic, Mechanized, Observable, Operational SHell: an executable formalization of the POSIX shell standard.* GitHub Repository.
URL: https://archive.softwareheritage.org/swh:1:dir:4de38fb893247f0620222d78e80eabf1c830b25

[17] Vidar Holen. *shellcheck: A shell script static analysis tool.*
URL: https://github.com/koalaman/shellcheck.

[18] IEEE and The Open Group. *The Open Group Base Specifications Issue 7, 2018 edition.* 2018.
URL: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.

[19] IEEE and The Open Group. *The Open Group base Specifications Issue 7, 2018 edition. Volume XBD: Base Definitions.* 2018.
URL: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.

[20] IEEE and The Open Group. *The Open Group base Specifications Issue 7, 2018 edition. Volume XCU: Shell & Utilities.* 2018.
URL: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.

[21] IEEE and The Open Group. *The Open Group base Specifications Issue 7, 2018 edition. Volume XSH: System Interfaces.* 2018.
URL: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.

[22] Ian Jackson, Christian Schwarz, Russ Allbery, and Sean Whitton. *Debian Policy Manual, version 4.4.1.1.* September 2019.
URL: https://www.debian.org/doc/debian-policy/.

[23] Nicolas Jeannerod. *colis-batch, a tool to run colis-language on packages and corpuses of packages.* GitHub Repository.
URL: https://archive.softwareheritage.org/swh:1:dir:2ec3501225000c1df6b12318e421cad59de957

[24] Nicolas Jeannerod. *Morsmall, a concise AST for POSIX shell.* GitHub Repository.
URL: https://archive.softwareheritage.org/swh:1:dir:29da9e4a2314493aa4eadf86c81c8645210057

[25] Nicolas Jeannerod and Ralf Treinen. *shstats, a statistical analyzer for corpora of shell scripts.* GitHub Repository.
URL: https://archive.softwareheritage.org/swh:1:dir:c096dc18c53b88342837b28d8386744023c859

[26] François Pottier. *Visitors: an OCaml syntax extension which generates object-oriented visitors for traversing and transforming data structures.*
URL: https://archive.softwareheritage.org/swh:1:dir:e16d1eeb404b57f6bb6ed284ae8ea0173b8299

[27] François Pottier and Yann Régis-Gianas. *Menhir: An LR(1) parser generator for OCaml.*
URL: https://archive.softwareheritage.org/swh:1:dir:48045c3ab0b2be2bbb89079c1e9e49bcf49e3f

[28] Yann Régis-Gianas and Nicolas Jeannerod. *lintshell, a user-extensible lint for POSIX shell.* GitHub Repository.
URL: https://archive.softwareheritage.org/swh:1:dir:d45c920801b1ed2f6ac1267b26285e666106c7

[29] Yann Régis-Gianas, Nicolas Jeannerod, and Ralf Treinen. *Morbig, A Static Parser for POSIX Shell.* GitHub Repository.
URL: https://archive.softwareheritage.org/swh:1:dir:9eae6fd10c5a3eedf7c7ddfa3a3546254e52dd

[30] *Report of colis-batch 6a1657c running colis-language 74e73d5 on Debian sid for amd64, including contrib and non-free, as of October 6, 2019.* Zenodo Archive.
URL: https://doi.org/10.5281/zenodo.4471388.

[31] Joseph Ritson. "The Valentine". In: *Gammer Gurton's Garland or The Nursery Parnassus: A Choice Collection of Pretty Songs and Verses for the Amusement of all Little Good Children who can neither read nor run.* Edited by R. Christopher. 1784, pages 39–40.

[32] Roland Rosenfeld. *rancid-cgi: looking glass CGI based on rancid tools.* Debian Package.
URL: https://packages.debian.org/en/sid/rancid-cgi.

[33] *Software Heritage.*
URL: https://www.softwareheritage.org/.

[34] *Software Heritage archive.*
URL: https://archive.softwareheritage.org/.

[35] The Debian Project. *Bugs tagged colis.*
URL: https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org.

[36] The Linux Foundation. *Filesystem Hierarchy Standard, version 3.0.* June 3, 2015.
URL: https://refspecs.linuxfoundation.org/fhs.shtml.

[37] *The OCaml system release 4.11 – Documentation and user's manual.* August 19, 2020.
URL: http://caml.inria.fr/pub/docs/manual-ocaml/.

List of Figures

List of Tables

List of Definitions

List of Lemmas

List of Theorems

Index of Concepts

Index of Notations