Université de Lille,
École doctorale Mathematics and Digital Sciences

Vérification de transformations de données arborescentes

Les transducteurs d'arbres appliqués à un problème de vérification sur des scripts shell

Thèse de doctorat en informatique

présentée et soutenue par

Paul D. Gallot

le 16 décembre 2021 *devant le jury composé de:*

Présidente du jury	Sophie TISON	Professeure	Unversité de Lille
Directeur de thèse	Sylvain SALVATI	Professeur	Université de Lille
Directeur de thèse	Aurélien LEMAY	Maître de conférences	Université de Lille
Rapporteur	Pierre-Alain REYNIER	Professeur	Université Aix-Marseille
Rapporteur	Sylvain SCHMITZ	Professeur	Université Paris Diderot
Examinateur	Patrick BAILLOT	Directeur de recherche	Université de Lille
Examinateur	Sebastian MANETH	Professor	University of Bremen

Région Hauts-de-France

Cette thèse est financée par la région Hauts-de-France
et l'Agence Nationale de la Recherche.

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

Université de Lille,
École doctorale Mathematics and Digital Sciences

Safety of transformations of data trees

Tree transducer theory applied to a verification problem on shell scripts

PhD thesis of computer science

by

Paul D. Gallot

Defended on December 16^{th} 2021 *before a jury composed of:*

Sophie TISON	Professor	Unversité de Lille	Jury President
Patrick BAILLOT	Directeur de recherche	Université de Lille	Examiner
Aurélien LEMAY	Maître de conférences	Université de Lille	Supervisor
Sebastian MANETH	Professor	Universität Bremen	Examiner
Pierre-Alain REYNIER	Professor	Univesité Aix-Marseille	Reviewer
Sylvain SALVATI	Professor	Université de Lille	Supervisor
Sylvain SCHMITZ	Professor	Université Paris Diderot	Reviewer

This thesis is funded by the Hauts-de-France Region
and the Agence Nationale de la Recherche

Abstract

This thesis aims at studying formal modelisations of tree transformations, with a focus on tree transducers.

In particular, we want to use tree transformations to represent operations on file systems, which are represented as tree structures. More precisely, we modelise operations performed by Shell scripts used to install and remove packages on Debian GNU/Linux distributions. The Shell scripting language provides access to Unix commands performing changes on file systems in addition to other tools. We model Unix file systems as feature trees and we represent the actions of Unix commands on a file system using a model we call tree pattern transducers. This model uses tree patterns to represent modifications on feature trees, and a system of constraints to represent the domains of tree transformations. We translate Unix commands into this model. We then provide an algorithm for computing the composition of tree pattern transducers.

We implement a tool for finding the configurations of the file system which can make a given Shell script fail. Instead of computing the transducers representing scripts recursively on the structure of scripts, we only compute their domains. This allows us to reduce the complexity of our algorithm. The domains of transformations performed by scripts are computed recursively, starting with the last command in the script. This amounts to computing successive inverse images of the commands in a script. We examine the pros and cons of this algorithm and we implement it. The implementation is then tested on a corpus of Debian package scripts. To better inform the discussion around this algorithm's complexity, we give proof that the problem we are solving is NP-hard, even on very restricted sets of scripts.

In a more theoretical direction, we use techniques from the field of functional programming to shed new light on known models of transducers. We contribute a new class of transducers named High-Order Deterministic tree Transducers (HODT) which generalizes some known models of tree transducers. HODT are defined similarly to deterministic Top-down tree transducers (DTOP), but the output of rules are simply-typed λ-terms. We show how putting constraints on these terms yields different known classes of transducers: restriction to terms of order 0 yields the class of DTOP, while restriction to terms of order ≤ 1 yields the class of Macro Tree Transducers (MTT). We give a procedure for computing the composition of two HODT based on models of the λ-caclculus. We show that the order of the composition is the sum of the orders of the composed transducers, which gives an interesting explanation as to why DTOP are closed under composition (as HODT of order 0) but not MTT (HODT of order ≤ 1).

In particular we study the restriction of HODT to linear terms, to which we add an inspection by a bottom-up tree automaton. We show that this model represents the same tree-to-tree functions as other known classes of transducers, most notably finite-copying MTT and Transductions defined by Monadic Second-Order logic (MSOT). We then prove a similar result for the restriction to almost linear terms and both attribute tree transducers (ATT) and an extension of MSOT called Monadic Second-Order logic tree

Transductions with Sharing of subtrees (MSOTS). We then give a specialized procedure for the composition of HODT which preserves linearity. This procedure relies on linear logic and coherence spaces. Because the time complexity of this procedure largely depends on the order of transducers, we give a procedure to reduce the order of linear and almost linear transducers. We argue that composition algorithms for equivalent classes of transducers can also be decomposed into two steps: firstly, computing composition in a meta-class of transducers ressembling HODT and secondly, reducing the order to fall back to the initial model of transducers.

In the last part, we prove that the word language $MIX_2 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$, also known as the double-sided Dyck language D_2^*, is not an EDT0L. This implies that it cannot be the output language of an MSOT from a regular language.

Résumé

Cette thèse présente une étude des modélisations formelles de transformations de structures arborescentes centrée sur les transducteurs d'arbres.

En particulier nous voulons utiliser les transformations d'arbres pour représenter des opérations sur des systèmes de fichiers vus comme des structures arborescentes. Plus précisément nous modélisons les modifications opérées par les scripts Shell d'installation et de désinstallation de paquets sur les distributions GNU/Linux Debian.

Le langage de script Shell permet entre autres d'utiliser des commandes Unix pour modifier le système de fichiers. Nous représentons les arborescences de fichiers par des *feature trees*, et l'action des commandes Unix par un modèle que l'on appelle *tree pattern transducers*. Ce modèle utilise des templates d'arbre ou *tree patterns* pour représenter les modifications de *feature trees*, et un système de contraintes pour représenter les domaines de ces transformations. Nous traduisons les commandes Unix dans ce modèle, puis nous donnons un algorithme pour calculer les compositions de *tree pattern transducers*.

Nous présentons l'implémentation d'un outil qui trouve les configurations du système de fichiers qui ferais échouer un script Shell donné. Plutôt que de calculer récursivement les transducteurs représentants l'action des scripts, nous calculons seulement leurs domaines. Ceci réduit la complexité de l'algorithme. Les domaines des transformations sont calculés récursivement en commençant par la dernière commande du script. Cela revien à calculer successivement les images inverses des commandes d'un script.

Nous analysons les avantages et les inconvénients de cette approche. Nous en testons l'implémentation sur un corpus de scripts Shell de maintenance de paquets Debian. Pour éclairer l'analyse de la complexité de l'algorithme, nous présentons une preuve que le problème est NP-dur, même sur des ensembles très restreints de scripts.

Une seconde partie plus théorique de cette thèse utilise une approche programmation fonctionnelle des transformations d'arbres pour mieux comprendre les modèles de transducterus d'arbres existants. Nous présentons une nouvelle classe de transducteurs baptisée *High-Order Deterministic tree Transducers* (HODT) qui généralise plusieurs modèles connus de transducteurs d'arbres. Les HODT sont définis comme des transducteurs d'arbres déterministes top-down (DTOP), sauf que les règles peuvent produire des λ-termes simplement typés. Mettre différentes contraintes sur ces λ-termes permet d'obtenir différentes classes connues de transducteurs : la restriction au termes d'ordre 0 (arbres) donne la classe DTOP, et la restriction aux termes d'ordre 1 donne la classe des transducteurs d'arbres à macros (MTT). Nous présentons une procédure pour calculer la composition de deux HODT basée sur les modèles du λ-calcul. Nous montrons que l'ordre de la composition de deux HODT est la somme des ordres des transducteurs composés, ce qui donne nouvelle explication à ce que la classe DTOP est fermée par composition (HODT d'ordre 0), mais pas la classe MTT (HODT d'ordre ≤ 1).

Nous étudions plus en détail la restriction de HODT aux termes linéaires, à laquelle on ajoute une inspection par un automate d'abre bottom-up. Ce modèle est équivalent à plusieurs modèles de transducteurs connus, notamment *finite-copying* MTT et les transductions définies par les formules de la logique monadique du second ordre (MSOT). Nous prouvons aussi que la restriction aux λ-termes quasi-linéaires est équivalente aux modèle des transducteurs d'arbres à attributs (ATT), et aux transductions définies par les formules de la logique monadique du second ordre avec partage de sous-arbres (MSOTS). Nous donnons une nouvelle procédure de composition qui conserve la linéarité. Cette procédure repose sur la logique linéaire et les espaces cohérents. Comme la complexité de cette procédure dépend fortement de l'ordre des transducteurs, nous donnons également une procédure pour réduire l'ordre des HODT linéaires et quasi-linéaires. Nous soutenons que les algorithmes de composition de modèles de transducteurs équivalents peuvent aussi être décomposés en deux étapes : une première étape de composition dans une meta-classe de transducteurs similaire à HODT, puis une deuxième étape de réduction d'ordre qui permet de retomber sur le modèle de transducteurs initial.

Dans une dernière partie nous démontrons que le langage $MIX_2 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$, ou *double-sided Dyck language* D_2^*, n'est pas un EDT0L. Par conséquent il ne peut pas être le langage de sortie d'une transduction MSO depuis un langage régulier.

Acknowledgements & Remerciements

I would like to start these acknowledgements by warmly thanking my reviewers, Pierre-Alain Reynier and Sylvain Schmitz, for accepting to read this manuscript in such a short delay, and my examiners, Patrick Baillot, Sebastian Maneth and Sophie Tison, for accepting to be part of my jury. I am honoured by the attention you have given to my work.

Thank you Sylvain and Aurélien, my supervisors, thank you for helping me when I needed it and trusting me when I needed it. Thank you for all the help you provided. Thank you for all our talks, academic and otherwise, which fostered both my scientific and personal development.

Thank you to all members of the Links research team, you have all shown me goodwill and understanding, and made me feel welcome in the team. I would not have made it through without all of you. I especially want to thank fellow PhD students Momar, Nicolas, Lily and Corentin, and engineers Nicolas and Jeremy for all the quirky and interesting conversations which made life in the team so wonderfull.

Thank you Magnet team members, with whom we shared laughs and thursday cakes, thank you Onkar, Cesar, Carlos, Nathalie, Arijus, Brij, Mariana and Marc. Thank you Michal. Thank you Yiding. Merci à Julia, Émilie et tous mes amis du M3, ce fut toujours un plaisir de venir vous voir pour échanger des anecdotes.

Merci à tous les non-chercheurs d'inria, merci à Nathalie sans qui je me serais perdu cent fois dans les méandres administratifs de l'administration française, merci à Ingrid et Marie pour vos conseils toujours avisés, merci à Charlotte pour les bons moments passés à l'AGOS. Merci à Aurélien et Florent pour les conseils en sport et en informatique. Merci à Carine, Julien, Géraldine, Manon, Marie-B, Cyril, Anne, Adrien.

Merci à tous mes amis Lillois, merci à vous Athénaïs, Adrien et Lucie pour les meilleurs moments de ma vie d'étudiant, pour nos discussions, les sérieuses et les autres, merci pour votre soutien irremplacable, je n'y serais jamais arrivé sans vous. Merci à Laurie pour les Tuche party, merci à Julie pour les fou-rires, merci à Alice et Hooper pour les super balades, merci à Cédric, Wilhelm et Lucas pour les apéros qui n'en finissent pas, merci à Dimitri et Alyx pour les soirées et dîner-spectacle, merci à Quentin et Omar pour les après-midi jeux.

Merci à tous ceux qui, malgré la distance, sont resté des amis précieux, merci à François, merci à Léo, Damien, Antoine pour les soirées JDR, merci à Alexandre, Théotime, Pierre et Gabriel pour nos soirées d'entrainement, merci à Élodie, Rémi, Thomas, Chloé, Kevin et Cindy pour tous les bons moments passés ensemble. Merci à Servane, Ninon, Jim et

Léo.

Merci à ma famille, merci à vous maman et papa pour votre soutien indéfectible. Merci à Marceau, Chloé, Gaspar, Robin, Basile, Elisa, Adèle, Lucas, Solène, Louis, Léonard, Louise, Lazare, Bastien, Adrien, Anna, Mamie Renée, Papi Frantz, Papi Daniel, Marie-Claire et Jean-Marc, Jean-Jacques et Sabine, Agnès et Julien, Annick et Vincent, Nicole, Chantal et Max, je vous aime tous sans exception.

Merci à mes colocataires qui m'ont supportés malgré les aléas de la thèse. Merci en particulier à Gilles, Céline, Julien et Olivier avec qui j'ai été confiné, personne d'autre n'aurais pu faire du confinement une expérience aussi fun et agréable. Merci à Alexia pour toute ta bienveillance. Merci à Nolwenn, Pierre, Florine, Louanne, Mélanie, Sohane, Antoine, Maurine, Mathilde, Amar, Corentin, Valentin. Merci à Damien sans qui j'aurais oublié d'écrire ces pages de remerciements.

Contents

Part I

Part 1 : Introduction and state of the art

Chapter 1

Introduction

1.1 Motivations

There is a long history of devices used to aid computation, but none had more profound consequences on society than modern computers. Before computers, such devices were designed to perform only one task. Computers were different because they could be used for a variety of different tasks, provided you gave them the appropriate programs.

The design of such programs was simple in the early years of programming, but as computers' speed and memory improved, the sizes and complexities of programs increased. Naturally, with larger and more complex programs came more problems and more bugs. Testing programs has become harder because multiple programs are stored and run at the same time. Each combination potentially creates unforeseen interactions. The internet has fostered sharing of programs up to the point that no two systems are likely to use the same combinations of software.

Operating systems have been developed in order to manage ressources between programs and improve processing speed. A particular distribution of interest to us is the *Debian GNU/Linux* operating system. It is free and open-source, meaning it allows users to freely study, copy and modify the code, in an attempt to encourage people to improve it and build upon it. This system grants access through the internet to a large compendium of programs written by lots of different people. Such programs are organized into packages; their installation, update and removal are handled by each package's *maintainer scripts*, which perform the necessary operations on the file system. Such scripts are a critical part in the maintenance of Debian systems.

However, because maintainer scripts are used on a great variety of system configurations, they can produce bugs that remain undetected by standard testing procedures. This problem is especially hard on Unix systems because packages have many interdependencies and can be written by a wide range of different people. As a consequence, there is a need for methods to help to reduce the number of bugs in maintainer scripts.

The Debian community has been working on such methods to avoid bugs in maintainer scripts. These methods are mainly based on the enforcement of practices in writting code for maintainer scripts. A Debian team is dedicated to the quality assurance of packages[1].

The Debian Policy[2] is a document aiming to normalise the writing of packages. It preconizes a flow structure for the installation of packages which organizes the calls to maintainer scripts. This document also provides guidelines on the behaviour and syntax of those scripts.

The Debian community has also implemented tools, like Lintian[3] and piuparts[4], that automatically check the adherence of maintainer scripts to the Debian Policy guidelines and that test the installation of packages. Some initiatives also use formal methods, for example the EDOS and Mancoosi projects [25, 1], which check the dependencies of packages, ensure the consistency of installed software and aim to improve the overall quality of Debian software installation.

1.2 Our approach

As part of the CoLiS project, our approach to this problem is to apply formal verification methods to maintainer scripts. In particular, we aim at representing the action of scripts on file systems as operations on formal tree structures. The final goal being the implementation of a tool looking for bugs in maintainer scripts, and especially those bugs happening on specific file systems.

The CoLiS approach differs from previous attempts at improving the quality of scripts in its use of formal verification techniques, and its objective to contend with the diversity of possible configurations of file systems. This could allow us, for example, to uncover bugs which depend on a specific set of packages being already installed on your computer.

To formally represent tree transformations we use tree transducers. There exists many models of tree transducers. Their defining characteristic is that they compute output trees recursively on the structure of their input. Other models of tree transformations exist. For example logical models like Monadic Second-Order logic (MSO) only describe transformations without showing how to effectively compute them.

To better understand the subtleties of tree transformations we will explore the theoretical foundations of tree transducers. In particular we study theoretical models of tree transducer under the lens of functional programming. In particular we design a model of transducers named High-Order Deterministic tree Transducers (HODT) based on simply-typed λ-calculus. This model provides a generalization of known models of transducers. In addition, we use methods from functional programming to design efficient algorithms for our model. We study restrictions of HODT representing classes of transformations pertinent for the CoLiS project. This grants us a deeper understanding of the problem of CoLiS, especially on notions of domains of transformations.

Using this theoretical understanding, we opt to build a tailor-made model for our implementation in CoLiS. The main reason why we do not use the HODT model for

[1] The Debian Quality Assurance Team, see `https://qa.debian.rog/`

[2] see `https://www.debian.org/doc/debian-policy/`

[3] See `https://lintian.debian.org/`

[4] See `https://piuparts.debian.org/`

CoLiS is the particular structure of file systems. The model we use is named tree pattern transducers. It relies on tree patterns to describe modifications of trees, and on a system of constraints on trees to represent the domains of transformations.

1.3 The CoLiS project

Part of this thesis presents work done in the scope of the CoLiS project. The project is financed by the french institution Association Nationale de la Recherche (ANR). This chapter presents the topic of CoLiS and the difficulties that come with it. It exposes the parts of the project which are pertinent to our work. It also briefly explains our approach for the project.

The CoLiS acronym stands for Correctness of Linux Scripts. The project's stated goal is to apply techniques from deductive program verification and analysis of tree transformations to the problem of analyzing Shell scripts used in software installation. We view installation scripts as tree transformations because they transform file systems, which can be seen as tree structures.

The main part of the CoLiS project is the implementation of a toolchain able to analyse software packages of the Debian distribution. The implementation of this toolchain can be divided into two parts: a front-end which parses and treats the parts of scripts that do not interact with file system, and a back-end which checks interactions with the file system. Two versions of the back-end were implemented. The first implementation uses a modelisation of tree transformations based on a logical paradigm named feature tree logic [19]. This first implementation was developed by our collaborators on the CoLiS project and is therefore not part of our contribution. The second implementation, developed by us, is the topic of part II of this thesis.

This section is divided into three subsections. In subsection 1.3.1 we present the first implementation of the CoLiS toolchain, which is not part of our contribution, and its results. In subsection 1.3.2 we dwell on the difficulties of representing file systems as trees. In subsection 1.3.3 we discuss the properties of transducers which would be most useful for representing tree transformations in the CoLiS project.

1.3.1 The CoLiS toolchain

We present here part of the work of Nicolas Jeannerod, Benedikt Becker, Yann Régis Gianas, Claude Marché, Mihaela Sighireanu, and Ralf Treinen in the scope of the CoLiS project. In particular we present a toolchain used to analyse individual Shell scripts used in the installation and removal of Debian packages. We present it as it was in late 2020 because this version was the basis for our work. It has been updated several times since and may be updated again in the future.

Another tool, which we do not present in this thesis, was developed by the same authors in order to recreate the different installation scenarios of Debian packages. This is because the installation of a Debian package usually relies on running several scripts in a specific order, as dictated by the Debian Policy Manual[5].

The toolchain is composed of several parts which are described in detail in [19]. In our case it can be broadly simplified into two:

- a front-end part (we will call it the *concrete interpreter*) which parses scripts and tests the parts which do not interact with the file system,
- a back-end part (we will call it the *symbolic interpreter*) which checks the interactions with the file system.

As running example in this section we use the maintainer script `preinst` from package `arpalert_2.0.12-3+b1` represented in Figure 1.1. This script is run before the installation of the package. This particular version comes from the Debian *sid* distribution released on 2019-10-06.

The tool distinguishes between calls to Unix commands which interact with the file system and those who do not. From now on we call *file system commands* those calls that interact with the file system. For example, the *file system commands* of the script in Figure 1.1 are `[ -d $DBDIR -a -d $BACKUPDIR ]`, `[ -d $BACKUPDIR/$NAME ]`, `rm -r $BACKUPDIR/$NAME` and `cp -rp $DBDIR $BACKUPDIR/$NAME`. Note that it includes calls which modify the file system, but also calls which only test the existence of files or directories. Also, whether a call is considered a *file system command* can depend on the command's arguments, for example the call `[ '0'='1' ]` is not a *file system command.*

The concrete interpreter

Simply put, the *concrete interpreter* takes as input a Shell script and starts to run the script normally, until it meets a call to a *file system command.* When it encounters such a call, it creates two branches for the execution of the script: one where the command succeeds and one where it fails. When such a command fails it may induce the script to fail, but if the command is the condition of an `if` statement then the execution can keep going. Then it resumes both executions, branching again at each *file system command.* In order to avoid infinite branching in loops, it assumes a bound on the number of iterations of loops.

The execution scenarios of the example script of Figure 1.1 are showed in Figure 1.2. For this script we can see that the concrete interpreter substitutes the variables and branches only on the *file system commands.* It produces 7 distinct executions: the script succeeds on executions `S1`,`S2`,`S3` and `S4`; it fails on executions `E1`,`E2`,`E3`. Note that the concrete interpreter sees the logical operator `-a` (logical *and*) in the call `[ -d /var/lib/arpalert -a -d /var/backups ]` and splits it into `[ -d /var/lib/arpalert ]` and `[ -d /var/backups ]`. The call `[ -d /var/lib/arpalert ]` goes first and if it fails the second call `[ -d /var/backups ]` is skipped, as dictated by the specification. Note also that the failure of calls in the condition of `if` statements do not induce the script to fail, it simply signals that the condition is not fulfilled.

[5] https://www.debian.org/doc/debian-policy/

```
#!/bin/sh
# based on arpwatch.preinst: v4 2004/08/14 KELEMEN Peter <fuji@debian.org>
# arpalert.preinst: v1 2006/10/12 Jan Wagner <waja@cyconet.org>

set -e

NAME=arpalert
DBDIR=/var/lib/$NAME
DBFILE=arp.dat
BACKUPDIR=/var/backups

# Back up collected ARP databases.
if [ -d $DBDIR -a -d $BACKUPDIR ]; then
    if [ -d $BACKUPDIR/$NAME ]; then
        rm -r $BACKUPDIR/$NAME
    fi
    cp -rp $DBDIR $BACKUPDIR/$NAME
fi
```

Figure 1.1 – script `preinst` of package `arpalert_2.0.12-3+b1`

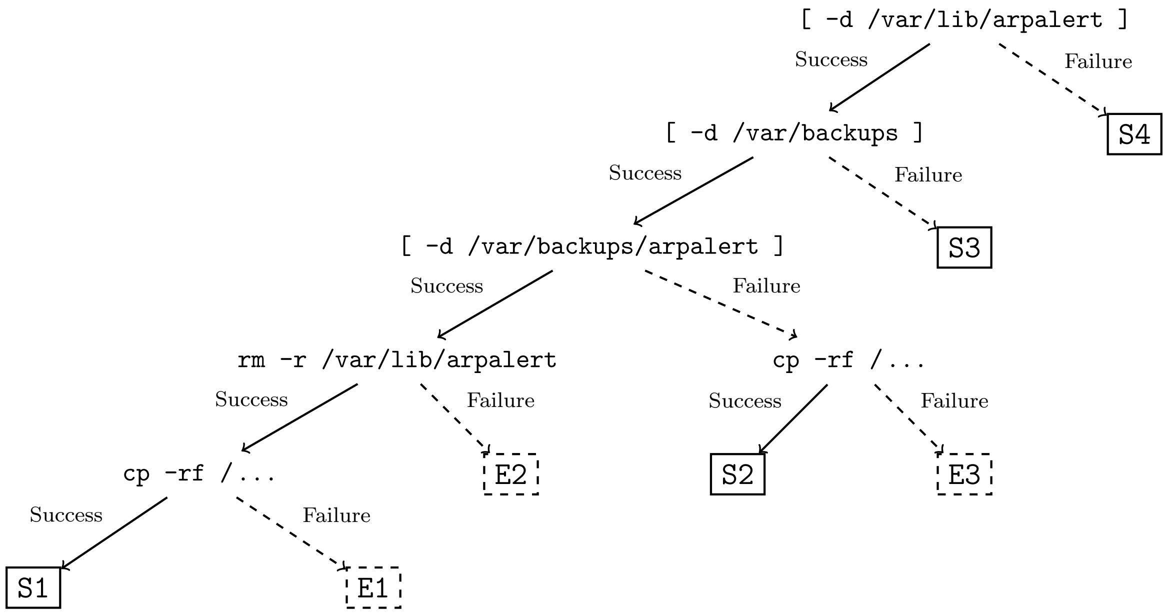


Figure 1.2 – execution scenarios for the script of Figure 1.1

When the branched out executions of the script end, with the script either failing or succeeding, the *concrete interpreter* gives a summary of each execution scenario. These summaries are called *script traces*, they only record a list of the *file system commands* and, for each, whether they failed or succeeded in the corresponding scenario. They are defined as follows:

Definition 1 A *trace atom* is either *success*(`command`) or *failure*(`command`) where `command` is a *file system command*, i.e. a call to a Unix command, including the command's arguments, which interacts with the file system. *success*(`command`) is called a *success atom* and *failure*(`command`) is called a *failure atom*.

A *script trace* is a finite ordered list of *trace atoms*.

For example, the script traces of the execution scenarios `S3` and `E2` shown in Figure 1.2 are:

`S3` → (*success*(`[ -d /var/lib/arpalert ]`), *failure*(`[ -d /var/backups ]`))
`E2` → (*success*(`[ -d /var/lib/arpalert ]`), *success*(`[ -d /var/backups ]`),
success(`[ -d /var/backups/arpalert ]`), *failure*(`rm -r /var/lib/arpalert`))

Then each execution scenario is associated with a *script trace* representing the list of *file system commands* used in this scenario, in the order in which they are run, and, for each *file system command*, whether the call succeeded or not.

In each execution scenario the script either succeeds of fails, so the *concrete interpreter* computes two sets of *script traces*: one set of execution scenarios where the script succeeds, and one set of execution scenarios where the script produces an error.

There may exist some scenarios where the tool is not sure how the script executes, either because the concrete interpreter uses approximations or because the POSIX specification is non-deterministic. This can produce a third set of *script traces*, we call it the set of *undefined script traces*.

The symbolic interpreter

The *symbolic interpreter* takes as input the three sets of success, failure and undefined *script traces* produced by the *concrete interpreter*. For each script trace, it computes the modifications performed by the script trace on the file system, and checks if the conditions on the success and failure of *file system commands* are met.

This way the symbolic interpreter computes the sets of initial configurations of the file system which are compatible with each script trace. If no initial configuration is compatible with a given script trace then the corresponding execution scenario is impossible. All the script traces which have at least one possible initial configuration represent possible execution scenarios of the studied script. The three sets of script traces are then updated by removing the impossible ones, what remains are all the possible execution cases of the script.

On the example script of Figure 1.1, the script trace of execution case `S3` is possible when the file system has a directory at path `/var/lib/arpalert` but not at path

`/var/backups`. However the script trace of execution case `E2` has no compatible configuration of the file system: the success of command `[ -d /var/lib/arpalert ]` implies that there is a directory at path `/var/lib/arpalert` which is incompatible with the failure of command `rm -r /var/lib/arpalert`. In fact all the executions which make the script fail (executions `E1`, `E2` and `E3`) have no compatible configurations of the file system.

The symbolic interpreter uses a representation of tree transformations based on a logic formalism named *feature tree logics* which we do not explain here. As part of our contribution we design another version of the symbolic interpreter, this second version is detailed in part II.

Interfacing concrete and symbolic interpreter

The two parts of the tool could be made to work sequentially, by first running the concrete interpreter and then giving the resulting *script traces* to the symbolic interpreter. But, in order to improve time complexity, they actually go back and forth.

We have seen that the concrete interpreter computes a tree structure of execution scenarios, branching on each *file system command*, as in Figure 1.2. In this tree structure, paths to leaves represent full executions of the script, and paths to inner nodes represent partial executions of the script. In actuality the concrete interpreter sends to the symbolic interpreter partial script traces. When the symbolic interpreter identifies a *script trace* without compatible file system configurations, it allows the concrete interpreter to cut a branch of executions of the script.

Using dynamic programming, the symbolic interpreter can use previous computations of partial *script traces* of length n to compute partial *script traces* and check if they have compatible file system configurations. So, even in the worst case where no *script trace* are removed, the back-and-forth has the same complexity as using the interpreters sequentially.

The output of the toolchain is a list of execution cases of the input script with, for each case, a report stating:

- whether the execution case is a success, failure or undefined case,
- a description of the execution provided by the *concrete interpreter* including the script's flow, the success and failures of command calls and the standard output,
- a description of the configurations of the file system which can lead to this execution, produced by the symbolic interpreter.

Practical results

Here we summarize the results obtained from the CoLiS toolchain which are not part of our contribution.

The toolchain has been used to detect errors and bugs on large corpuses of maintainer scripts of Debian packages. Specifically on snapshots of the Debian *sid* distribution ("unstable" distribution where packages are made public before they are considered safe enough for the stable version of Debian). Between 2016 and 2019 a total of 151 bugs were reported to the Debian Bug Tracking System, of which 92 had been resolved as of march 30^{th} of 2021 (date of publication of [19]).

1.3.2 Modelisation of filesystems

Because our goal is to check which file systems are compatible with which executions of scripts, our choice for the modelisation of file systems is a crucial decision point in designing our algorithm. We should strive for a compromise between the tractability of our algorithm and the accuracy of the model with respect to actual file systems.

In the version of the toolchain described in this chapter, file systems are modeled as rooted tree structures. Nodes are labeled with filetypes. There are 7 different filetypes (as mentioned in the POSIX standard[6]): directories, regular files, symbolic links, FIFO special files, block devices, character devices and sockets. Only directory nodes can have child nodes.

Filenames are modeled as labels of edges, a node's filename is the label of the edge connecting it to its parent node. Two edges leaving the same parent node have distinct labels. The set of filenames is infinite, although in actual file systems, filenames cannot be longer than 255 characters.

Symbolic links are only considered as any other type of non-directory file. Modeling symbolic links as they are used in file systems would require to model file systems as graph structures instead of tree structures. This would make the verification process much more complicated. Also, the symbolic interpreter is supposed to anticipate all possible configurations of the file system, and doing so with symbolic links would yield more intricate and unrealistic configurations. Those could make the output of the tool (the reports on possible execution cases) bigger and harder to understand. Furthermore it could prevent the concrete interpreter from cutting branches of executions which would poorly impact the complexity of the algorithm.

The contents of files are not modeled, it would require to understand all file formats which is unrealistic. Timestamps would offer little in terms of verification so they are not represented. Permissions are not represented either since the tool treats installation and uninstallation scripts, which should be run with superuser privileges. To be fair there are configurations where other scripts are called inside installation scripts, and in such cases it is pertinent to check their execution permissions, but in most of these cases the tool has no access to the scripts' code, so finding bugs in these calls usually requires human intervention.

1.3.3 Modeling tree transformations for CoLiS

To build our version of the symbolic interpreter, we want to represent the action of script commands on file systems using a model of tree transducers. The tree transformation

[6] provided by the IEEE and The Open Group. `http://pubs.opengroup.org/onlinepubs/9699919799/`

performed by a *script trace* is the composition of the transformations of the *trace atoms* in the trace. So our criteria for choosing a model of transducer for CoLiS are the following:

1. **Expressiveness:** the model needs to be able to represent the action of *trace atoms* (and therefore of Unix commands) on file systems,

2. **Composition:** we need to be able to compute efficiently the composition of two tree transformations in the model.

For the project we also need to be able to compute the domain of a tree transformation i.e. the set of trees on which it is defined, but this is often implied by the criterion of composition.

1.4 Theoretical models of tree transformations

In this section we present several known theoretical models representing transformations of tree structures, with the prior goal of finding a model suited to represent tree transformations performed by scripts on file systems. Most of these models come from the tree transducers literature, but we also present models based on logical formulas, notably tree transformations defined by Monadic Second-Order Logic formulas.

We studied some of these models as potential candidates for the representation of scripts in the CoLiS project. The Macro Tree Transducer (MTT) model was of particular interest to us, and especially its Single-Use Restricted variant (MTT^R_{sur}) [14]. Indeed the MTT^R_{sur} model is expressive enough to represent simple operations on trees like copying a subtree from one path to another, but its single-use restricted condition keeps it from creating an arbitrary number of copies. MTT^R_{sur} also has the benefit of defining a class of transformations closed under composition, as opposed to MTT.

The problem with the MTT^R_{sur} model is that the only available algorithms able to compute its composition consist in translating our MTT^R_{sur} in another model, Streaming Tree Transducers (STT) [2] or MSO definable Transductions (MSOT) [10, 11], and computing the composition in those models. We could have used one of those two models where composition algoritms exist, but the algorithm for STT has non-elementary time complexity, and we preferred to avoid logical formalisms like MSOT.

Instead we used tools from functional programming to better understand the mechanisms at play in the composition of transducers, as a result we defined a model of transducers based on functional programming which also generalizes MTT, MTT^R_{sur} and other models of transducers pertinent for CoLiS. We call this model High-Order Deterministic tree transducers and we explore its most interesting properties in part III.

1.4.1 Models of tree transformations

In this section we explore different models for representing classes of tree transformations, and especially models of tree transducers.

These models are all defined on ranked ordered trees. A tree can be inductively defined as being composed of a root node and a finite set of other trees called its child trees. Each node is labeled with a symbol from a finite *alphabet* of symbols. An alphabet is *ranked* if with each symbol a in it is associated with an integer n, called its arity, such that all node labeled a has exactly n child trees. Trees are *ordered* if sets of child trees are ordered, when they are ordered we usually note sets of child trees as a tuple. When we talk about trees in this section we mean ranked ordered trees.

A tree t composed of a root node labeled a and a tuple of child trees $(t_1, \ldots, t_n)$ is noted $t = a(t_1, \ldots, t_n)$. For each ranked alphabet Σ we note $\mathcal{T}(\Sigma)$ the set of ordered trees on alphabet Σ. Given two ranked alphabets Σ_ι and Σ_o, a *tree transformation* from Σ_ι to Σ_o is a total or partial function from $\mathcal{T}(\Sigma_\iota)$ to $\mathcal{T}(\Sigma_o)$.

Tree transducers are abstract models used to represent classes (sets) of tree transformations. Their main characteristic is that they describe how their output tree is computed, and this computation is done by traversing their input tree.

Deterministic TOP-down tree transducers (DTOP) [28] It is one of the simplest models of tree transducers. Most other models of transducers will be described as variations of this one, so we give a formal definition of it:

Definition 2 A Deterministic TOP-down tree transducer (DTOP) is a tuple $(Q, \Sigma_\iota, \Sigma_o, q_0, R)$ where :

- Q is the finite set of *states*
- Σ_ι is the ranked *input alphabet*
- Σ_o is the ranked *output alphabet*
- q_0 is the *initial state*
- R a set of *rules* of the form

  $$q(a(x_1, \ldots, x_n)) \rightarrow t$$

  where $q \in Q$, $a \in \Sigma_\iota$, $x_1, \ldots x_n$ are variables representing child trees of the node labeled a, and t is a tree on alphabet Σ_o except that some of its subtrees are be replaced with the application $q'(x_i)$ of a state $q' \in Q$ to a variable x_i with $i \leq n$.

It is deterministic when it has at most one rule for each possible left hand side of rule, i.e. for each state $q \in Q$ and for each symbol $a \in \Sigma_\iota$, there is at most one rule in R with q and a on its left side.

Applying a state q to a tree $a(t_1, \ldots, t_n)$ on Σ_ι consists in using applying a rule from R of the form $q(a(x_1, \ldots, x_n)) \rightarrow t$ (if it exists), the result is obtained from t by replacing in it all subtree of the form $q'(x_i)$ with the result of applying q' to the child tree t_i.

Such a transducer associates with each input tree t on alphabet Σ_ι the result of applying the initial state q_0 to t.

This transducer is called top-down because it starts by applying a rule to the top of the tree (the root node), and then recursively applies rules to its child trees.

Deterministic TOP-down tree transducers with Regular look-ahead (DTOP^R) Each such transducer is equipped with a *bottom-up tree automaton* called its *look-ahead automaton*, this automaton is used on the input of a transducer before applying rules of the transducer.

Definition 3 A Bottom-up Tree automaton (BOT) is a tuple (P, Σ_ι, R') where:

- P is the finite set of states
- Σ_ι is the ranked *input alphabet*
- R' is a finite set of rules of the form

  $$a(p_1, \ldots, p_n) \to p$$

  where $p, p_1, \ldots, p_n \in P$ and $a \in \Sigma_\iota$.

  It is *deterministic* when there is at most one rule in R' for each possible left hand side. It is *non-deterministic* otherwise.

This definition differs from the classical one in that there are no final states, this is because it is used as a *look-ahead automaton.*

This automaton associates one of its states with each node of the input tree, these states provide information about the bottom of part of the tree and help guide the rules of the transducer. The rules of this transducer are of the form:

$$q(a(x_1, \ldots, x_n)\langle p_1, \ldots, p_n\rangle) \to t$$

where $p_1, \ldots, p_n$ are the states associated, by the look-ahead automaton, with the trees represented by variables $x_1, \ldots, x_n$ respectively. Such a rule can only be applied to a tree $a(t_1, \ldots, t_n)$ if the look-ahead automaton associates with each tree t_i state p_i (for $i \leq n$).

Such a transducer is called *deterministic* when there is at most one rule for each possible left hand side.

The DTOP^R model is strictly more *expressive* than the DTOP model, meaning DTOP^R can describe strictly more tree transformations than DTOP.

We define DTOP and DTOP^R here because they are a good starting point into the literature of tree transducers. It is also useful for us to illustrate some intuitions about computations of tree transducers in general, for example to clarify why we used functional programming to represent transducers (more detail on this in subsection 1.4.4).

The DTOP and DTOP^R models were not considered for representing scripts in CoLiS because they are not expressive enough to represent some operations of scripts, for example the operation of copying a directory from one path to another in a tree.

Macro Tree Transducers (MTT) [14] This model is similar to DTOP, with the addition that states of the transducer can use variables representing trees on the output alphabet. Each state is associated with a fixed number m called its arity, which is the number of variables it uses. The rules of those transducers are of the form:

$$q(a(x_1, \ldots, x_n), y_1, \ldots, y_m) \to t$$

where $y_1, \ldots, y_m$ are variables representing trees on the output alphabet Σ_o and, in tree t, variables y_i for $i \leq m$ can be used as subtrees, and nodes labeled with the application $q'(x_i)$ are nodes with the arity of q'.

The result of applying a rule $q(a(x_1, \ldots, x_n), y_1, \ldots, y_m) \to t$ to a tree $a(t_1, \ldots, t_n)$ on Σ_ι and trees $t_{o,1}, \ldots, t_{o,m}$ on Σ_o is obtained from t by replacing each variable y_j with $t_{o,j}$ (for $j \leq m$), and by replacing each subtree $q'(x_i)(t'_1, \ldots t'_{m'})$ with the application of q' to tree t_i on Σ_ι and trees $t'_1, \ldots t'_{m'}$ on Σ_o.

In a sense, a DTOP can be seen as a MTT whose states all have arity 0.

We can also add a regular look-ahead to the MTT model, this model is noted MTT^R and is defined similarly to DTOP^R.

Monadic Second-Order logic defined Transductions (MSOT) [10, 11] In this model, a tree transformation is described by means of logical formulas. The logical language is that of Monadic Second Order logic on finite trees. A key feature of this logic is its capacity to quantify over sets of individuals. It has become a prominent object of study because of its tight connection with finite state machines. Representing transformations of structures such as graphs, trees or strings by means of this logic originates in the work of Courcelle [10, 11]. These transformations have been, since then, widely explored and characterized by other means. Of particular interest to us is the characterization of the tree transformations they define with MTT$^R_{sur}$ [14].

Monadic Second-Order logic defined Transductions with Sharing of subtrees (MSOTS) [10, 11] This model is similar to the previous MSOT model but differs in that, instead of outputting trees, it outputs trees with *sharing*, or said differently, *directed acyclic graphs* (DAGs). MSOT has a restricted capacity of copying as there is a uniform bound on the number of copies of a given subtree of the input. This restriction disappears for MSOTS as the mechanism of *sharing* allows for unbounded copying.

We also consider models of transducers which we do not define here. There are Attribute Tree Transducers (ATT) [7] and Streaming Tree Transducers (STT) [2]. We also consider the restrictions of ATT and MTT^R with the *Single-Use Restricted* property [12, 13], those are noted ATT$_{sur}$ and MTT$^R_{sur}$ respectively. The models ATT and ATT$_{sur}$ are defined in subsection 7.3.1.

1.4.2 Expressivity of tree transformation models

In this section we discuss the relative expressiveness of the presented models of tree transformations.

$$\mathrm{MTT}^R_{sur} \Leftrightarrow \mathrm{ATT}_{sur} \Leftrightarrow \mathrm{MSOT} \Leftrightarrow \mathrm{STT}$$

$$\mathrm{ATT} \Leftrightarrow \mathrm{MSOTS}$$

Figure 1.3 – Equivalences between classes of tree transformations

$$\begin{tikzcd}
& \mathrm{DTOP}^R \arrow[r, Rightarrow] & \begin{array}{c}\mathrm{MTT}^R_{sur}\\ \mathrm{ATT}_{sur}\\ \mathrm{MSOT}\\ \mathrm{STT}\end{array} \arrow[r, Rightarrow] \arrow[dd, Rightarrow] & \begin{array}{c}\mathrm{ATT}\\ \mathrm{MSOTS}\end{array} \\
\mathrm{DTOP} \arrow[ur, Rightarrow] \arrow[dr, Rightarrow] & & & \\
& \mathrm{MTT} \arrow[r, Rightarrow] & \mathrm{MTT}^R &
\end{tikzcd}$$

Figure 1.4 – Relations of inclusion between classes of tree transformations

Some models have been proven to be equivalent, meaning that they represent the same classes of tree transformations. It has been shown that MSOT is equivalent to ATT_{sur}[7], to MTT^R_{sur}[12, 13] and to STT[2]. It has also been shown in [7] that MSOTS is equivalent to ATT.

Equivalences between classes are shown in Figure 1.3, and relations of inclusion between classes are shown in Figure 1.4

1.4.3 Composition of tree transformations

Transducers describe functions on trees, the operation of composition on functions then induces an operation of composition on tree transducers. The composition of two functions described by transducers cannot always be represented as a transducer. For the CoLiS project we need a model in which we can always compute the composition of transducers. So as to select an adequate class of tree transformations the closure under composition of the class is key for us. A class is *closed under composition* when the composition of two of its transformations is one of its transformations.

It has been shown that the classes of DTOP, DTOP^R, STT and MSOT are closed under composition[2, 11]. Because MTT_{sur} and ATT_{sur} are equivalent to MSOT and STT, those are also classes closed under composition.

The classes of MTT, MTT^R, ATT and MSOTS are not closed under composition.

1.4.4 The functional programming approach

Tree transducers are distinguished from other types of representations of tree transformations mainly by their recursive way of computing on trees. Trees form a structure particularly suited for such computations because of their recursively defined structure. When computing the output of a DTOP for example, applying a state to a node launches recursive calls of other states on the subtrees of that node. Such recursive calls can be seen

$\mathcal{T} = (Q, \Sigma_\iota, \Sigma_o, q_0, R)$ with:

- $Q = \{q_0, q_1\}$ with q_0 of arity 0 and q_1 of arity 1,
- $\Sigma_\iota = \{f, a\}$ with f of arity 2 and a of arity 0,
- $\Sigma_o = \{g, b\}$ with g of arity 1 and b of arity 0,
- The rules of R are:

$$\begin{array}{ll} q_0(f(x_1, x_2)) & \to g(\,q_1(x_1)(\,q_1(x_2)(b)\,)\,) \\ q_0(a) & \to b \\ q_1(f(x_1, x_2), y_1) & \to g(\,q_1(x_1)(\,q_1(x_2)(y_1)\,)\,) \\ q_1(a, y_1) & \to y_1 \end{array}$$

Figure 1.5 – Definition of the MTT $\mathcal{T}$

```ocaml
type input = f of input * input | a
type output = g of output | b

let rec state_q0 t = match t with
    | f(x1,x2) -> g (state_q1 (x1) (state_q1 (x2) (b)))
    | a        -> b
and state_q1 t y1 = match t with
    | f(x1,x2) -> g (state_q1 (x1) (state_q1 (x2) (y1)))
    | a        -> y1

val state_q0 : input -> output = <fun>
val state_q1 : input -> output -> output = <fun>
```

Figure 1.6 – Functional program in OCaml representing MTT $\mathcal{T}$

as *pure functions*, meaning the computation of one such call is determined only by the subtree given as argument, and can be seen as isolated from the rest of the transducer's computation. This is also true for computations of other models of transducers like MTT or STT.

For example the MTT T shown in Figure 1.5 can be represented by the program shown in Figure 1.6 in the functional programming language OCaml.

This means that we can see tree transducers as a particular type of functional programs. This is why we choose to apply functional programming techniques to tree transformations and especially tree transducers.

This leads us to design a new model of tree transducers called High-Order Deterministic tree Transducers (HODT) based on simply-typed λ-calculus. This model generalizes several known models of tree transformations and, by using techniques from functional programming, we are able to design two algorithms for computing the composition of HODT.

1.5 Contribution and Plan of the thesis

1.5.1 High-Order Deterministic tree Transducers

One of the main contributions of this thesis is the model of High-Order Deterministic tree Transducers (HODT). They are defined similarly to DTOP, except that rules produce simply-typed λ-terms instead of trees. They are similar to the model of high-level tree transducers defined in [16], except that they only consider *safe* λ-calculus.

In general the HODT model is more expressive than the other models described in this chapter, but by putting simple restrictions on the λ-terms produced by rules, we reach the same classes of tree transformations as known models of transducers. For example the restriction to terms of order 0, noted $\mathrm{HODT}_{\leq 0}$, is equivalent to DTOP, and the restriction to terms of order 1, noted $\mathrm{HODT}_{\leq 1}$, is equivalent to MTT. More interestingly, when we add a *regular look-ahead* to our HODT, we find that the restriction to linear terms, noted $\mathrm{HODTR}_{\mathrm{lin}}$, is equivalent to MSOT and MTT^{R}_{sur}, and the restriction to almost-linear terms, noted $\mathrm{HODTR}_{\mathrm{al}}$, is equivalent to MSOTS and ATT. So HODT gives a generalisation of all these different models.

The use of λ-calculus allows us to use powerful tools from the field of functional programming, which grants better insight into the deeper mechanisms of the tree transducers generalized by our model. Most notably we give two algorithms, based on models of the λ-calculus, for computing the composition of two transducers. We observe that when we compose two HODT, the order of the resulting transducer is the sum of the transducers of which it performs the composition. This is coherent with the fact that DTOP ($\mathrm{HODT}_{\leq 0}$) are closed under composition but not MTT ($\mathrm{HODT}_{\leq 1}$).

In the special cases of $\mathrm{HODTR}_{\mathrm{lin}}$ and $\mathrm{HODTR}_{\mathrm{al}}$, we give a procedure for reducing the order of transducers. This allows to represent $\mathrm{HODTR}_{\mathrm{lin}}$ and $\mathrm{HODTR}_{\mathrm{al}}$ as transducers whose rules produce tuples of tree contexts.

1.5.2 Tree transducers in the CoLiS project

Our goal in the CoLiS project is to design and implement an algorithm for the task of the symbolic interpreter: computing the set of configurations of the file system which are compatible with a given *script trace* (definition 1), and checking if this set is empty. As opposed to the symbolic interpreter based on a logical framework, we want to make use of the scientific literature on tree transducers for our algorithm.

The model of $\mathrm{HODTR}_{\mathrm{lin}}$ could have been a good fit for the CoLiS project. It is expressive enough to represent Unix commands like `mv` and `cp`, and we have an algorithm for composition with reasonable complexity. The problem is that they only work on ranked ordered trees. We can use an encoding of unranked trees as binary trees to overcome the ranked limitation. We have looked for ways to go from ordered to unordered trees, for example in [8], operations on ordered trees are considered as operations on unordered trees when they obey a condition of commutativity on child trees. But Unix commands do not fulfill this commutativity condition because they use filenames to navigate file systems, so this particular approach does not work. For these reasons we decided not to use $\mathrm{HODTR}_{\mathrm{lin}}$, or any restriction of HODT, for our implementation in CoLiS.

We were not able to use $\mathrm{HODTR}_{\mathrm{lin}}$ directly, but our work on $\mathrm{HODTR}_{\mathrm{lin}}$, especially the sharp management of the look-ahead to describe domains of composition of transducers, instructed our approach to the modelisation of Unix commands. We designed a model tailor-made for the representation of Unix commands in CoLiS. This model is named tree pattern transducers. It uses tree patterns to represent modifications on trees, and a system of constraints to represent domains of transformations.

We then designed an algorithm for computing the composition of such transducers. We especially tried to optimize the complexity of computing the domain of the composed transducer. Our work on domains of compositions of tree transformations with $\mathrm{HODTR}_{\mathrm{lin}}$ was useful to us in that regard. We focused on the computation of domains because of the particular approach we took for the implementation in the CoLiS project.

The backward approach Our initial ambition for CoLiS was to use tree transducers to represent the action of Unix commands on file systems. But an alternative to this first approach is to only compute sets of trees representing configurations of the file system compatible with executions of scripts. Given a *script trace* there are two ways of doing this: one is to compute the images of the set of all trees, through each successive command in the trace (seeing commands as tree-to-tree functions), starting with the first command. The other is to compute the successive *inverse images* through the commands, starting with the last command. These correspond to two ways of checking the compatibility of commands with the file system: either going *forward* in the execution of the script, starting with the first command, or going *backward* in the execution, starting with the last command.

Our chosen approach is to go *backward*, it mainly has two benefits. First it allows to directly compute the set of initial configurations of the file system that are compatible with the input script trace. The *forward* approach would yield the set of *final* configurations instead, which is less helpful when our goal is to detect bugs and explain to a human how it arises.

The second benefit of the *backward* approach is its algorihmic complexity. Simply put, the sets of trees which are direct images of commands are harder to represent than inverse images. This is because of the operation of copying subtrees. When a directory is copied, the direct image of the set of all trees has two directories which must be identical, whereas the inverse image has no such type of constraint. Through successive direct images, constraints of equality between directories can compound, meaning that a new constraint on one directory can propagate to other directories which makes verification more complex. On the other hand computing successive inverse images through copying transformations only requires the unification of two directories, without chains of implications. It is for the same reason that DTOP do not preserve the regularity of tree languages, but their inverse images do. In fact, the very idea of basing our verification algorithm on inverse images takes its root in this very general property of transducers: *regularity of languages is preserved by inverse images of transducers.*

Our contribution in the CoLiS toolchain We used the *backward* approach and the model of *tree pattern transducers* to implement our version of the symbolic interpreter. This part of our contribution includes the design of tree pattern transducers, the modelisation of Unix commands in this model, an algorithm for computing the composition of these transducers and another for computing inverse images, and finally the implementation of the symbolic interpreter and its integration with the concrete interpreter.

1.5.3 The MIX language

We have included a result we obtained in the beginning of the thesis. At the time we were studying the expressivity of MSOT on strings. In particular, we have obtained an impossibility result about a particular language called MIX_2. This language is defined as $MIX_2 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$, also known as the double-sided Dyck language D_2^*. We show that this language cannot be the output language of an MSOT from a regular language. Actually our result is slightly stronger and shows that this language is not an EDT0L.

By the end of the 70s and in the early 80s, the difference between EDT0L and context-free languages were widely studied and considered difficult. This result is part of this tradition and shows that one of the simplest context-free language is not an EDT0L.

1.5.4 Plan of the thesis

The contributions of this thesis are broadly divided into three parts.

1. Our first contributions, presented in part II, are the implementation of a tool checking Debian installation scripts for bugs in their interaction with the file system, and the design of its algorithm. The modelisation of Unix file systems and of the flow structure of scripts is described in chapter 2. Our design of a model of tree transformations, of formal algorithms on this model, and the modelisation of Unix commands are presented in chapter 3. Our implementation of the tool checking scripts is presented in chapter 4.

2. Our main theoretical contribution, presented in part III, is the design of the model of High-Order Deterministic tree transducers (HODT) which generalizes several known classes of transducers. We define the model and its variants, and we compare their domains and expressiveness in chapter 5. Two different algorithms computing the composition of our transducers based on semantics of the λ-calculus are described and their relative complexities are discussed in chapter 6. Finally we prove that the linear and almost-linear variants of our model are respectively equivalent to MSOT and MSOTS in chapter 7.

3. Our last contribution, presented in chapter 8 in part IV, is a proof that the language MIX, which is the commutative closure of the Dyck language, is not in the class of languages EDT0L, which implies that it is not an output language of a word-to-word transduction defined by Monadic Second-Order logic.

Finally we conclude and examine research paths left open after this work.

Part II

Part 2 : Contribution to the CoLiS project

Chapter 2

Abstraction of filesystems and scripts in the CoLiS project

In this part of the thesis we present our practical contribution to the CoLiS project culminating with the implementation of a tool able to check for errors in Shell scripts, in particular those arising from the scripts' interactions with the file system. Chapter 3 presents the model of *tree pattern transducers* used to represent the action of scripts on the file system. Chapter 4 presents the implementation based on *tree pattern transducers*.

Here, in chapter 2, we expose the problem we are trying to solve as part of the CoLiS project, and we discuss the abstractions used to formalise it. This discussion reiterates and builds on arguments given in subsection 1.5.2 of the introduction.

This chapter is organized as follows. In section 2.1 we formalize the problem. In section 2.2 we detail the control flow tools used in scripts and we deal with them. In section 2.3 we give our modelisation of file systems. Finally in section 2.4 we use a reduction from the SAT problem to prove that the problem we are trying to solve is NP-hard.

2.1 Abstraction of the problem

The goal of this part of the thesis is to check whether scripts can execute on file systems without errors. A simple way to check if a script can run without errors is to run it, but a caveat of this approach is that the execution of a script depends on the file system. What we want to do is account for these interactions between scripts and file systems.

To that aim we reuse code from a tool developed by Nicolas Jeannerod, Benedikt Becker, Ralf Treinen, Mihaela Sighireanu, Claude Marché and Yann Régis Gianas as part of the CoLiS project. The full toolchain is described in depth in [19], in this work we only reuse the part of the tool which we described as the *concrete interpreter* in subsection 1.3.1. We use this *concrete interpreter* as a front-end for our tool, part II of this thesis is dedicated to building the corresponding back-end part of our tool and describing its underlying properties. We start with a precise recall of what the *concrete interpreter* does.

```
    if test -f /usr/share/bash; then
        touch /usr/share/bash
    else
        cp /bin/bash /usr/share/bash
    fi

```

Figure 2.1 – Example of Shell script

Concrete interpreter

The *concrete interpreter*'s function is to handle the parts of scripts which do not interact with the file system, in practice it takes a script as input and computes a list of execution scenarios of this script, which are summarized as *script traces*. We first recall the definition of script traces.

Definition 1 A *trace atom* is either *success*($\texttt{command}$) or $failure(\texttt{command})$ where `command` is a call to a Unix command, including the command's arguments, which interacts with the file system. *success*($\texttt{command}$) is called a *success atom* and $failure(\texttt{command})$ is called a *failure atom*.

A *script trace* is a finite ordered list of *trace atoms*.

We show what the *concrete interpreter* computes on an example script (different from the example of subsection 1.3.1). The script in Figure 2.1 has 2 success script traces:

$$\texttt{S1} = (\mathit{success}(\texttt{test -f /usr/share/bash}), \mathit{success}(\texttt{touch /usr/share/bash}))$$
$$\texttt{S2} = (failure(\texttt{test -f /usr/share/bash}), \mathit{success}(\texttt{cp /bin/bash /usr/share/bash})$$

and 2 error script traces:

$$\texttt{E1} = (\mathit{success}(\texttt{test -f /usr/share/bash}), failure(\texttt{touch /usr/share/bash}))$$
$$\texttt{E2} = (failure(\texttt{test -f /usr/share/bash}), failure(\texttt{cp /bin/bash /usr/share/bash})$$

For handling loops, in particular potentially infinite loops like `while`, the concrete interpreter assumes there is a bound on their number of iterations. The alternative, computing possibly infinite iterations of functions on tree structures, would make the problem undecidable. In fact the halting problem on Turing machine can be reduced to this problem by representing the tape as in tree structure.

This approximation of the problem may seem substantial but, in practice, maintainer scripts rarely use loops with an arbitrary number of iterations.

The concrete interpreter produces these *script traces* which are the input of the symbolic interpreter.

Symbolic interpreter

One of our main contributions to the CoLiS project is the design of an algorithm for the symbolic interpreter, its implementation and underlying formal properties. The symbolic interpreter takes as input a *script trace* and outputs the set of file systems which are compatible with it.

As stated in subsection 1.5.2, our approach consists in computing successive inverse images throught each command in a *script trace*. But we still need to formalize the action of Unix commands on file systems as tree-to-tree functions. For this we want a model of transducers with which we are able to:

- represent the action on file systems of Shell commands like `mkdir` and `cp`, both when they succeed and when they fail,
- represent the composition of any two transducers as one transducer,
- compute on which input trees a transducer is defined (i.e. computing its domain).

It would be possible to represent arguments of script commands as input of our transducers, but we choose not to because it would greatly increase the complexity of the model. Instead we choose to use transducers parametrized by arguments of commands. In others words the transducer associated with a script command will also depend on the command's context (including option, arguments and current working directory).

Our chosen model of tree transducers, tree pattern transducers, is exposed in chapter 3.

2.2 Control flow structure of Shell scripts

This section is not part of our contribution, in it we briefly present how the control flow structure of scripts impacts their executions and their corresponding *script traces*. In particular we see how script traces of `if` statements and loops are computed. We also introduce the notion of *uninterrupted composition* of commands which is useful to model Unix commands used on lists of arguments.

We recall the definition of *script traces*:

Definition 1 A *trace atom* is either $success(\texttt{command})$ or $failure(\texttt{command})$ where `command` is a *file system command*, i.e. a call to a Unix command, including the command's arguments, which interacts with the file system. $success(\texttt{command})$ is called a *success atom* and $failure(\texttt{command})$ is called a *failure atom.*

A *script trace* is a finite ordered list of *trace atoms.*

The definition of script traces can be given by induction on the structure of scripts. Scripts can contain a variety of different constructs, but we only show how script traces of sequential composition, `if` statements and `while` loops are computed. The behaviour of other structures of scripts like `case` and `for` can be deduced from the behaviours of sequential composition, `if` and `while`.

For all script s, we note $[\![s]\!]^{\uparrow}$ the set of script traces representing succeeding executions of s (executions with exit code 0), and $[\![s]\!]_{\downarrow}$ the sets of script traces representing its executions with errors (i.e. executions where the exit code is not 0). For all two sets S_1 and S_2 of script traces, we note $S_1 \cup S_2$ their union, and we note $S_1 \circ S_2$ the set of script traces obtained by concatenating a script trace in S_1 with a script trace from S_2. The operation $\circ$ is called composition, it is associative and we write S_1^n the composition defined by $S_1^1 = S_1$ and $S_1^{n+1} = S_1 \circ S_1^n$ for all $n \geq 1$.

Then we have the following equations:

- For all scripts s_1, s_2, the sequential composition of s_1 and s_2 is noted $s_1\,;s_2$, and its associated sets of script traces are:
  $\llbracket s_1\,;s_2 \rrbracket^{\uparrow} = \llbracket s_2 \rrbracket^{\uparrow} \circ \llbracket s_1 \rrbracket^{\uparrow}$
  $\llbracket s_1\,;s_2 \rrbracket_{\downarrow} = \llbracket s_1 \rrbracket_{\downarrow} \cup (\llbracket s_2 \rrbracket_{\downarrow} \circ \llbracket s_1 \rrbracket^{\uparrow})$
  This amounts at saying that $s_1\,;s_2$ fails either because s_1 fails (and s_2 is not run) or because s_2 fails after s_1 is run successfully.

- For all script s_1, s_2, s_3, noting s the following conditionnal statement:

  ```
  if $s_1$; then
     $s_2$
  else
     $s_3$
  fi
  ```

  The success and failure script traces of s are:
  $\llbracket s \rrbracket^{\uparrow} = (\llbracket s_2 \rrbracket^{\uparrow} \circ \llbracket s_1 \rrbracket^{\uparrow}) \cup (\llbracket s_3 \rrbracket^{\uparrow} \circ \llbracket s_1 \rrbracket_{\downarrow})$
  $\llbracket s \rrbracket_{\downarrow} = (\llbracket s_2 \rrbracket_{\downarrow} \circ \llbracket s_1 \rrbracket^{\uparrow}) \cup (\llbracket s_3 \rrbracket_{\downarrow} \circ \llbracket s_1 \rrbracket_{\downarrow})$

- For all scripts s_1, s_2, noting s the following `while` loop:

  ```
  while $s_1$
  do
     $s_2$
  done
  ```

  we associate with s the following transducers:

  $\llbracket s \rrbracket^{\uparrow} = \llbracket s_1 \rrbracket_{\downarrow} \cup (\llbracket s_1 \rrbracket_{\downarrow} \circ \llbracket s_2 \rrbracket^{\uparrow} \circ \llbracket s_1 \rrbracket^{\uparrow}) \ldots (\llbracket s_1 \rrbracket_{\downarrow} \circ (\llbracket s_2 \rrbracket^{\uparrow} \circ \llbracket s_1 \rrbracket^{\uparrow})^n)$
  $\llbracket s \rrbracket_{\downarrow} = (\llbracket s_2 \rrbracket_{\downarrow} \circ \llbracket s_1 \rrbracket_{\downarrow}) \cup (\llbracket s_2 \rrbracket_{\downarrow} \circ \llbracket s_1 \rrbracket_{\downarrow} \circ \llbracket s_2 \rrbracket^{\uparrow} \circ \llbracket s_1 \rrbracket^{\uparrow}) \ldots (\llbracket s_2 \rrbracket_{\downarrow} \circ \llbracket s_1 \rrbracket_{\downarrow} \circ (\llbracket s_2 \rrbracket^{\uparrow} \circ \llbracket s_1 \rrbracket^{\uparrow})^n)$

  The number n here is the bound on the number of iterations of the loop, it can be set when launching the concrete interpreter.

Uninterrupted composition

Some Unix commands like `mkdir` or `rm` are usually run on one argument, but they can be used on a list of arguments in order to repeat the command on each argument, but this repetition is not exactly equivalent to the sequential composition of commands. For example the command `mkdir /bin /boot` is not equivalent to the sequential composition `mkdir /bin; mkdir /boot`, rather it is equivalent to `mkdir /bin` $\divideontimes$ `mkdir /boot` where $\divideontimes$ represents the operation of *uninterrupted composition* on scripts.

The difference with sequential composition is that an error in the first component of the composition does not keep the second component from being run. In terms of script traces this is defined by;

$$\llbracket s_1 \divideontimes s_2 \rrbracket^{\uparrow} = \llbracket s_2 \rrbracket^{\uparrow} \circ \llbracket s_1 \rrbracket^{\uparrow}$$
$$\llbracket s_1 \divideontimes s_2 \rrbracket_{\downarrow} = (\llbracket s_2 \rrbracket_{\downarrow} \circ \llbracket s_1 \rrbracket^{\uparrow}) \cup ((\llbracket s_2 \rrbracket_{\downarrow} \cup \llbracket s_2 \rrbracket^{\uparrow}) \circ \llbracket s_1 \rrbracket_{\downarrow})$$

Notice that uninterrupted composition is associative (like sequential composition), so we can talk about the uninterrupted composition of a sequence of more than two commands or scripts without any ambiguity.

Uninterrupted composition will be useful when defining the behaviour of Unix commands in chapter 3.

2.3 Abstraction of file systems

File systems under Linux operating systems are very elaborate. In this section we present a model of file systems that focuses on their hierarchical properties. This model is the same as the one described in subsection 1.3.2. Our simplifications make a compromise between tractability and accuracy with respect to actual file systems.

We model file systems as rooted tree structures. Edges and nodes are labeled. Labels of edges correspond to filenames. Labels of nodes correspond to types of files. Edges from mother to daughter are labeled by the daughter's name. Two distinct edges leaving a same node have distinct labels. In actual file systems, filenames cannot be longer than 255 characters, but we will work under the approximation that the set of filenames is infinite. Each node has a type amongst the 7 different types of files mentioned in the POSIX standard: directories, regular files, symbolic links, FIFO special files, block devices, character devices and sockets. Only directory nodes can have child nodes.

Although we differenciate between symbolic links and other types of files, we only consider them as leaf nodes. The reason why we do not want to model symbolic links is because it would require us to model file systems as graph structures instead of tree structures. This would make the verification process much more complicated. Also, the goal of our tool is to detect bugs which happen in specific configurations of file systems, and handling symbolic links could yield overly intricate and unrealistic configurations.

We do not model file contents. It would require to understand all file formats which is unrealistic. Timestamps would offer us little in terms of verification. Since we focus on installation and uninstallation scripts, which should be run with superuser privileges, we choose to forget about permissions.

Formal representation: we use the same formal representation of file systems in the whole CoLiS project. It is the same as the one used by other members of the project, including the implementation discussed in subsection 1.3.1. This representation is also described in [20]. Mentions of trees, file hierarchies and file systems in this part of the manuscript (part II) refer to this definition.

Definition 4 The set of UNIX features is the set of words over the alphabet of unicode characters which are neither `.`, `..`, nor contain any occurrence of the character `/`. We note it $\mathcal{F}$.

As stated in the UNIX standard[1], there exists seven file types :

- regular files, which we'll note here reg
- directories, noted here dir
- symbolic links, noted here $\mathsf{symlink}$
- FIFO special files, noted here fifo
- block devices, noted here block
- character devices, noted here char
- sockets, noted here sock

We note $\texttt{types} = \{\mathsf{dir}, \mathsf{reg}, \mathsf{symlink}, \mathsf{fifo}, \mathsf{block}, \mathsf{char}, \mathsf{sock}\}$ the set of UNIX file types. In our modelisation of file systems as trees, nodes are labeled with their file type and edges are labeled with filenames. So labels of nodes are in `types` and labels of edges are in $\mathcal{F}$.

In OCaml, feature trees could be implemented as in Figure 2.2.

```
module SMap = Map.Make(String)

type featureTree = node SMap.t
and node = Dir of featureTree | Reg | SymLink | Fifo | BlockDev |
    CharDev | Socket
(* here "node SMap.t" denotes the type of mappings from strings to nodes
    , it is similar to type "String -> node" for example *)
```

Figure 2.2 – Representation of feature trees as an OCaml type

Definition 5 *Feature trees* are inductively defined as :

$$\mathcal{T} = \mathcal{F} \rightsquigarrow ((\texttt{types} \setminus \{\mathsf{dir}\}) \cup (\{\mathsf{dir}\} \times \mathcal{T}))$$

where $\mathcal{F} \rightsquigarrow X$ denotes the set of mappings (partial functions with finite domain) from $\mathcal{F}$ to X.

The mapping map with domain $\{x_1, \ldots, x_n\}$ is noted $[x_1/\mathsf{map}(x_1), \ldots, x_n/\mathsf{map}(x_n)]$.

For example, the feature tree containing a regular file named `cat` inside a directory `/bin` is: $[\texttt{bin}/(\mathsf{dir}, [\texttt{cat}/\mathsf{reg}])]$. In general we use a simplified inline notation where slashes and dir are ommited and other file types are noted as exponents of their file names. So $[\texttt{bin}/(\mathsf{dir}, [\texttt{cat}/\mathsf{reg}])]$ is simplified to:

$$[\texttt{bin}[\texttt{cat}^{\mathsf{reg}}]]$$

[1] http://pubs.opengroup.org/onlinepubs/9699919799/

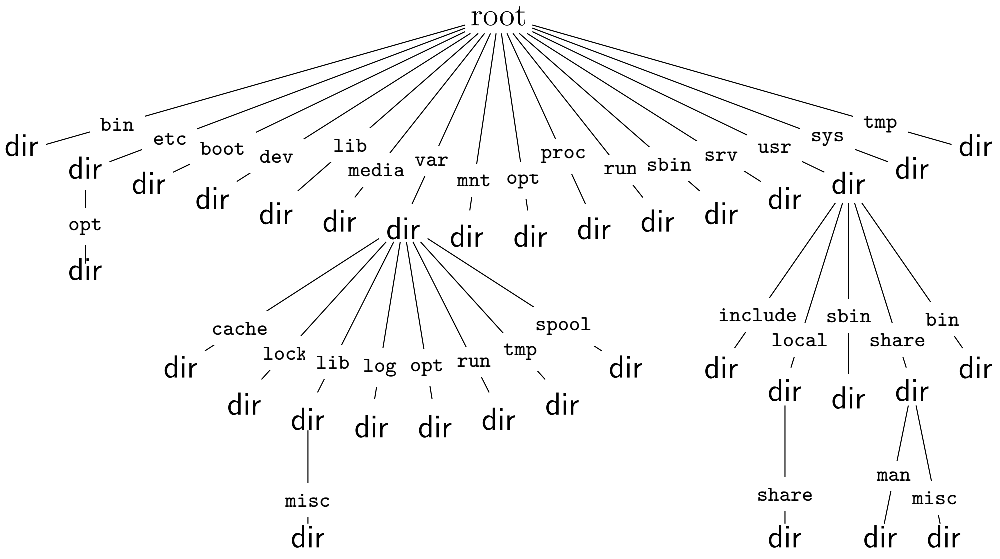


Figure 2.3 – Feature tree representing the directories of the Filesystem Hierarchy Standard

We will often write paths in such trees using the traditional UNIX notation: features in a path are listed in order and separated by the symbol `/`.

We can also represent feature trees graphically: in Figure 2.3 we represent the set of directories prescribed by the Filesystem Hierarchy Standard[2] (or FHS) for linux systems.

Some directories present in the FHS are not represented because they are optional or dependent on the distribution or hardware, or because they have been recently introduced or deprecated in the standard. We also do not represent standard files like `/bin/cat`.

This amounts to saying that a feature tree is a finite unordered tree where nodes are labeled with file types in `types` and edges are labeled with features in $\mathcal{F}$. Each node in a feature tree has a finite number of outgoing edges, and all outgoing edges of a node carry different names. Leaves may be either one of the non-directory file types or an empty directory. Inner nodes must be directories.

With this abstraction of file systems, we want to use formal tools on scripts in order to check when specific file systems can produce errors in scripts. For this we turn to tree transducers as a way to model scripts and their action on file systems.

2.4 NP-hardness of verification on scripts

In this section we prove that the problem of checking if a script can succeed is NP-hard, even with constraints on the set of commands allowed in the scripts. Most notably we show this is true for scripts containing only `cp -r`, `rm -r`, `rmdir` and `mkdir` type of commands, without any control flow constructs like `if` or `while`. We note this problem

[2] `https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard`

$SCT_{\{\texttt{cp -r, rm -r, rmdir, mkdir}\}}$.

This implies that the problem solved by the symbolic interpreter, deciding if a script trace has a compatible configuration, is also NP-hard. This is because scripts without control flow tools have only one script trace on which they succeed.

We do this through reduction from the SAT problem. Let ϕ be a SAT formula in conjunctive form:

$$\phi = C_1 \wedge \cdots \wedge C_m$$

We note $n = |\phi|$ the size of ϕ. We note V the set of boolean variables in ϕ and $h = |V|$ its cardinal. We note $y_1, \dots, y_h$ the variables in V. For all $y_i \in V$ we note $\overline{y_i}$ the corresponding negative literal. We note $W = V \cup \overline{V}$ the set of literals on V. We consider the function $x \mapsto \overline{x}$ to be an involution on W, i.e. we have $\overline{\overline{x}} = x$ for all $x \in W$.

In the reduction, we use the file system to represent valuations of the boolean variables in V. Then we will build a script which succeeds on a file system only if it represents a valuation of variables that satisfies formula ϕ.

To this end we injectively associate a feature with each literal $x \in W$, we will simply note x the feature associated with the literal x and assume context can clear any misunderstanding.

To represent trees we use the simplified inline notation. For example the expression $[\texttt{bin}[\texttt{cat}^{\textsf{reg}}, \texttt{cp}^{\textsf{reg}}], \texttt{usr}[\texttt{local}[], \texttt{share}[]]]$ denotes the tree shown in Figure 2.4.

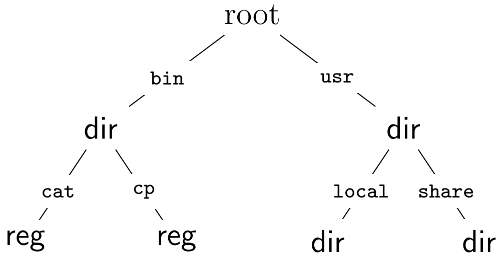


Figure 2.4 – Representation of feature tree $[\texttt{bin}[\texttt{cat}^{\textsf{reg}}, \texttt{cp}^{\textsf{reg}}], \texttt{usr}[\texttt{local}[], \texttt{share}[]]]$

To each valuation θ of the variables in V we associate a directory named $\texttt{val}$ containing, for each variable $a \in V$:

- if $\theta(a) = \mathit{true}$, a directory $a[a[]]$ and a directory $\overline{a}[]$,
- if $\theta(a) = \mathit{false}$, a directory $a[]$ and a directory $\overline{a}[\overline{a}[]]$.

We note it $\texttt{val}(\theta)$. For example if $V = \{a, b\}$ and θ is such that $\theta(a) = \mathit{true}$ and $\theta(b) = \mathit{false}$, then $\texttt{val}(\theta)$ denotes the directory $\texttt{val}[\,a[a[]], \overline{a}[], \overline{b}[\overline{b}[]], b[]\,]$ represented in Figure 2.5.

To each clause C_i we associate a feature noted $\tilde{C_i}$. To each valuation θ of the variables in V and each clause C_i of ϕ we associate the directory $\mathsf{eval}(\theta, C_i) = \tilde{C_i}[\top[\top[\dots\top[]\dots]]]$ where the number of $\top$ is $n + 1$ and n is the number of literals in C_i which are true in valuation θ.

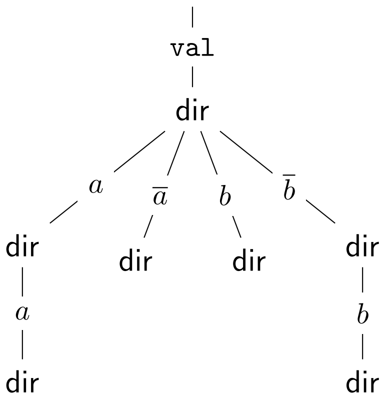


Figure 2.5 – Feature tree `val`(θ) when $\theta(a) = true$ and $\theta(b) = false$

We are going to build a script `s`(ϕ) which succeeds on a file system fs if and only if fs is of the form : $[\mathsf{val}(\theta), \mathsf{eval}(\theta, C_1), \ldots, \mathsf{eval}(\theta, C_m)]$ where θ is a valuation of variables which satisfies ϕ.

This script is composed of 3 main "subscripts", each making sure one of the following constraints are verified:

1. the script `checkVal`(ϕ) ensures that the directory at path `/val` is indeed of the form val(θ) where θ is a valuation of the variables in V,

2. the script `evalClauses`(ϕ) checks that, for each clause C_i, the directory at path `/`$\tilde{C_i}$ is indeed of the form eval(θ, C_i) where θ is the same valuation as the one described by val(θ),

3. the script `checkClauses`(ϕ) verifies that, for each clause C_i, at least one literal in the clause is true (assuming that the directory at path `/`$\tilde{C_i}$ is of the form eval(θ, C_i)).

Precisely the script `s`(ϕ) is of the form:

`init`(ϕ)`;` `checkVal`(ϕ)`;` `evalClauses`(ϕ)`;` `checkClauses`(ϕ)

where the script `init`(ϕ) initiates the computation by creating subdirectories so that following subscripts can work independently.

The script `init`(ϕ)

Each of the three main subscripts alters parts of the file system which they are checking (val(θ) or eval(θ, C_i) for $i \leq m$), we copy these parts first so that each subscript can work on a copy of what it needs to check without interacting with other subscripts.

The job of `init`(ϕ) is to create a subdirectory `eval`, in which the script `evalClauses`(ϕ) will work, and to copy in this directory the content of val(θ) and eval(θ, C_i) for all $i \leq m$:

`init`(ϕ) $=$ `mkdir /eval; setup`(C_1)`;` $\ldots$ `setup`(C_{m-1})`; setup`(C_m)

with, for all $i \leq m$:

$$\begin{array}{ll}\texttt{setup}(C_i) = & \texttt{mkdir /eval/}\tilde{C_i}\texttt{;} \\ & \texttt{cp -r /}\tilde{C_i}\texttt{/ /eval/}\tilde{C_i}\texttt{/} \\ & \texttt{cp -r /val/ /eval/}\tilde{C_i}\texttt{/;}\end{array}$$

The script `checkVal`(ϕ) will work exclusively in directory `/val`, while `evalClauses`(ϕ) will work in directory `/eval` and `checkClauses`(ϕ) will work in directories `/`$\tilde{C_i}$`/` for $i \leq m$.

The script `checkVal`(ϕ)

The script `checkVal`(ϕ) needs to check that the directory at path `/val` is of the form val(θ) for some valuation θ. This means that, for each variable $a \in V$, it contains either:

- a directory $a[a[]]$ and a directory $\overline{a}[]$ (if $\theta(a) = true$),
- a directory $a[]$ and a directory $\overline{a}[\overline{a}[]]$ (if $\theta(a) = false$).

In order to check this we use a special property of the command `cp -r`: if the destination path of the `cp -r` exists, then the command copies the directory from the source path *inside the directory at the destination path*; however if the destination path of the `cp -r` does not exist, then the command creates the directory at the destination path and copies *the content of* the directory from the source path *directly at the destination path.*

For example, it means that if you run the command `cp -r` $\overline{a}$`/` a`/`a`/` in a directory where you have a subdirectory $\overline{a}[]$ and a subdirectory $a[a[]]$, you end up with a subdirectory $a[a[\overline{a}[]]]$.

But if you run that same command `cp -r` $\overline{a}$`/` a`/`a`/` in a directory where you have a subdirectory $\overline{a}[\overline{a}[]]$ and a subdirectory $a[]$, you end up with the same result: a subdirectory $a[a[\overline{a}[]]]$.

After the command `cp -r` $\overline{a}$`/` a`/`a`/`, we can check that we have a directory $a[a[\overline{a}[]]]$ with a succession of `rmdir` commands (since `rmdir` only succeeds if its target directory is empty):

$$\texttt{rmdir } a/a/\overline{a}\texttt{; rmdir } a/a\texttt{; rmdir } a$$

So the following script succeeds if and only if it is run in a directory which contains either two directories $a[a[]]$ and $\overline{a}[]$ or two directories $a[]$ and $\overline{a}[\overline{a}[]]$:

$$\texttt{cp -r } \overline{a}/\ a/a/\texttt{; rmdir } a/a/\overline{a}\texttt{; rmdir } a/a\texttt{; rmdir } a$$

This property of the command `cp -r` is important because it allows to check a *disjunction* of two configurations of the input tree which are *not comparable through the relation of prefix* on trees. The command `mv` also has this property. We will see later in chapter 3 how the representation of sets of trees using tree prefixes leads to an algorithm for our verification problem, and how this specific property of `cp` and `mv` increases the complexity of this algorithm from linear time to NP time. We will also use this property in the script `evalClauses`(ϕ).

So, for each variable $a \in V$ we define the script `checkVar`(a) by:

$$\begin{aligned}\texttt{checkVar}(a) = {} & \texttt{cp -r /val/}\overline{a}\texttt{/ /val/}a\texttt{/}a\texttt{/}\\ & \texttt{rmdir /val/}a\texttt{/}a\texttt{/}\overline{a}\texttt{;}\\ & \texttt{rmdir /val/}a\texttt{/}a\texttt{;}\\ & \texttt{rmdir /val/}a\texttt{;}\\ & \texttt{rm -r /val/}\overline{a}\end{aligned}$$

Now we can define checkVal(ϕ) as:

$$\texttt{checkVal}(\phi) = \ \texttt{checkVar}(y_1)\texttt{;} \ \ \ldots \ \ \texttt{checkVar}(y_h)\texttt{; rmdir /val/}$$

where $y_1, \ldots, y_h$ are the boolean variables in V. We purge the directory */val/* with `rmdir /val/` in order to make sure that there were no other files in it. This script succeeds if and only if the directory at path `/val` of the form val(θ) for some valuation θ of the variables in V. We can also say that if the script succeeds then the valuation θ is unique (since it is determined by the file system).

The script evalClauses(ϕ)

The script evalClauses(ϕ) checks each clause one at a time:

$$\texttt{evalClauses}(\phi) = \texttt{evalClause}(C_1)\texttt{;} \ \ \ldots \ \ \texttt{evalClause}(C_m)$$

For each clause C_i, the script evalClause(C_i) checks that the directory at path /eval/$\tilde{C}_i$/$\tilde{C}_i$/ is of the form eval(θ, C_i) where θ is the valuation described by the directory val(θ) at path /eval/$\tilde{C}_i$/val/. Note that evalClause(C_i) will only work inside the directory at path /eval/$\tilde{C}_i$/ created by the script init(ϕ) (described above), with the directory at path /eval/$\tilde{C}_i$/$\tilde{C}_i$/ a copy of the directory at path /$\tilde{C}_i$/, and the one at path /eval/$\tilde{C}_i$/val/ a copy of the one at path `/val/`.

Noting $x_1, \ldots, x_k$ the literals in the clause C_i and $\overline{x_1}, \ldots, \overline{x_k}$ their negation, we define the script evalClause(C_i) as:

$$\begin{aligned}\texttt{evalClause}(C_i) = {} & \texttt{cp -r /eval/}\tilde{C}_i\texttt{/}\tilde{C}_i\texttt{/}\top\texttt{/ /eval/}\tilde{C}_i\texttt{/}\overline{x_1}\texttt{/}\overline{x_1}\texttt{/;}\\ & \texttt{cp -r /eval/}\tilde{C}_i\texttt{/}\overline{x_1}\texttt{/}\overline{x_1}\texttt{/}\top\texttt{/ /eval/}\tilde{C}_i\texttt{/}\overline{x_2}\texttt{/}\overline{x_2}\texttt{/}\\ & \vdots\\ & \texttt{cp -r /eval/}\tilde{C}_i\texttt{/}\overline{x_{k-1}}\texttt{/}\overline{x_{k-1}}\texttt{/}\top\texttt{/ /eval/}\tilde{C}_i\texttt{/}\overline{x_k}\texttt{/}\overline{x_k}\texttt{/}\\ & \texttt{rmdir /eval/}\tilde{C}_i\texttt{/}\overline{x_k}\texttt{/}\overline{x_k}\texttt{/}\top\texttt{/}\end{aligned}$$

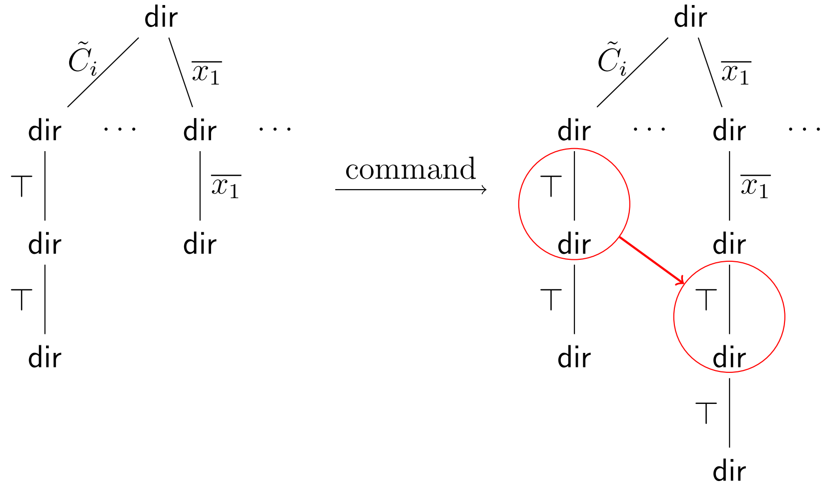
Here we use the same property of `cp -r` as for checkVal(ϕ): if the destination path exists then the command copies the directory from the source path *with its feature* $\top$, but if the destination path does not exists then the command copies only the content of the directory at the source and puts it in the directory at the destination path.

We see how it works on the following example. Since we only work inside the directory at path /eval/$\tilde{C}_i$/, we represent only the content of this directory. We start with the first command:

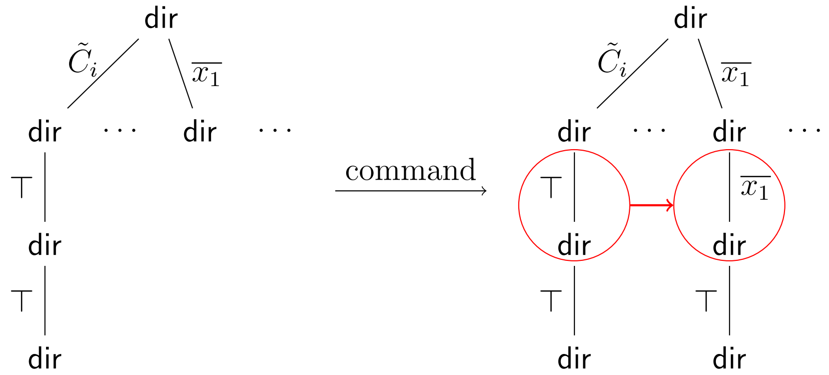
$$\texttt{cp -r /eval/}\tilde{C}_i\texttt{/}\tilde{C}_i\texttt{/}\top\texttt{/ /eval/}\tilde{C}_i\texttt{/}\overline{x_1}\texttt{/}\overline{x_1}\texttt{/}$$

It can behave in two ways depending on whether the directory at path `/eval/`$\tilde{C}_i$`/`$\overline{x_1}$`/`$\overline{x_1}$`/` exists or not. In order to better understand the difference between these two cases we add red arrows linking directories to their copy:

case 1 :



case 2 :



Here we see more clearly the differences between the behaviours of `cp -r`. In the first case the source directory is copied inside the destination directory, while in the second case the source directory is copied *in place of* the destination directory, changing the directory's name ⊤ to the name $\overline{x_1}$ specified in the destination path.

On this file system we can see that the number of occurrences of ⊤ transmitted to the target directory (path `/eval/`$\tilde{C}_i$`/`$\overline{x_1}$`/`$\overline{x_1}$`/`) depends on whether or not the path $\overline{x_1}$`/`$\overline{x_1}$`/` already existed before the command.

We know that, by definition of $\mathsf{val}(\theta)$, case 1 corresponds to $\theta(\overline{x_1}) = \mathit{true}$ and $\theta(x_1) = \mathit{false}$. In this case the length of the chain of ⊤ directories at path `/eval/`$\tilde{C}_i$`/`$\overline{x_1}$`/`$\overline{x_1}$`/` is *equal* to the length of the chain of ⊤ directories at path `/eval/`$\tilde{C}_i$`/`$\tilde{C}_i$`/`.

On the other hand, if $\theta(x_1) = \mathit{true}$ then $\theta(\overline{x_1}) = \mathit{false}$ and the length of the chain of ⊤ directories is *decremented*.

In summary, the length of the chain of ⊤ directories copied by each `cp -r` command is decremented for each literal x_j in the clause C_i such that $\theta(x_j) = \mathit{true}$, for $j \leq k$.

At the end of `evalClause`(C_i) we use `rmdir /eval/`$\tilde{C}_i$`/`$\overline{x_k}$`/`$\overline{x_k}$`/⊤/` to make sure that the directory at path `/eval/`$\tilde{C}_i$`/⊤` is empty. So the length of the chain of ⊤ directories is 1 at the end of `evalClause`(C_i). Since the length of the chain of ⊤ directories has been decremented for each literal in C_i that is true according to valuation θ, the length of the chain of $\top$ directories was $n+1$ at the start of $\texttt{evalClause}(C_i)$ where n is the number of literals of C_i which are true according to θ.

Note that, with this definition of $\texttt{evalClause}(C_i)$, the directory at path $\texttt{/eval/}\tilde{C_i}\texttt{/}\tilde{C_i}\texttt{/}$ could contain other files or directories than $\top$, so we have not strictly proven that this directory is of the form $\mathsf{eval}(\theta, C_i)$. It is possible to purge any unwanted files or directories by adding $\texttt{rm -r /eval/}\tilde{C_i}\texttt{/}\overline{x_j}\texttt{/}\overline{x_j}\texttt{/}\top\texttt{/; rmdir /eval/}\tilde{C_i}\texttt{/}\overline{x_j}\texttt{/}\overline{x_j}\texttt{/}$ at the end of $\texttt{evalClause}(C_i)$ for each x_j in clause C_i. This purging makes the proof clearer, but is not necessary for the reduction to work because the number of piled up $\top$ directories is still $n+1$ where n is the number of true literals in the clause.

We have proven that, by applying $\texttt{evalClause}(C_i)$ for each clause, we check that the directories at paths $\texttt{/eval/}\tilde{C_i}\texttt{/}\tilde{C_i}\texttt{/}$ initially were of the form $\mathsf{eval}(\theta, C_i)$, and therefore its copy (directory at path $\texttt{/}\tilde{C_i}\texttt{/}$) is also of the form $\mathsf{eval}(\theta, C_i)$.

$\texttt{checkClauses}(\phi)$

The last thing to check is that each clause C_i has at least one literal which is true according to the valuation θ. Since the length of the chain of $\top$ directories is $n+1$ with n the number of true literals in clause C_i, we need to check that $2 \leq n+1$. This is done by checking that there are at least two piled up $\top$ directories in the directory at path $\texttt{/}\tilde{C_i}\texttt{/}$, we use the script: $\texttt{rm -r /}\tilde{C_i}\texttt{/}\top\texttt{/}\top\texttt{/}$ which fails only if there is nothing at path $\texttt{/}\tilde{C_i}\texttt{/}\top\texttt{/}\top\texttt{/}$. So:

$$\begin{aligned} \texttt{checkClauses}(\phi) = \ & \texttt{rm -r /}\tilde{C_1}\texttt{/}\top\texttt{/}\top\texttt{/;} \\ & \vdots \\ & \texttt{rm -r /}\tilde{C_m}\texttt{/}\top\texttt{/}\top\texttt{/} \end{aligned}$$

With this we have checked that each clause contains at least one literal which is true according to a valuation θ described by the file system.

This proves that there is a file system on which the script $\texttt{s}(\phi)$ succeeds *if and only if* there exists a valuation θ which satisfies formula ϕ. Hence we have a linear reduction from the SAT problem to $SCT_{\{\texttt{cp -r, rm -r, rmdir, mkdir}\}}$ and, since the SAT problem is NP-hard, the $SCT_{\{\texttt{cp -r, rm -r, rmdir, mkdir}\}}$ problem is NP-hard.

In the reduction we use `mkdir` only in the $\texttt{init}(\phi)$ part of the script. We use it only to make sure that directories $\texttt{/eval/}\tilde{C_i}$ for $i \leq m$, are empty before the rest of the computation of the script. For each such directory we could, instead of using `mkdir`, assume that the directory already exists and use `rm -r` on each of the filenames which could interfere with the proper computation of the script. Those are, for each directory $\texttt{/eval/}\tilde{C_i}$, the filenames: $\tilde{C_i}, x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}$. The `mkdir /eval` can also be removed since we only work in its subdirectories $\texttt{/eval/}\tilde{C_i}$ for $i \leq m$. Which means that the reduction works even without the `mkdir` command. We chose to present the reduction with `mkdir` first because it makes the proof simpler. So the problem $SCT_{\{\texttt{cp -r, rm -r, rmdir}\}}$ is also NP-hard.

Part of the problem we are trying to solve in the CoLiS project is, given a script trace, decide whether it has at least one compatible configuration. The reduction from SAT also works for this problem, so it is also NP-hard.

This complexity result is important to have in mind when we later present our model for representing the action of scripts on file systems as tree transducers. Indeed we will see that our algorithm for checking script traces in general runs in exponential time (exponential in the length of the trace), but that exponential blow-ups only appear at places where `cp` and `mv` are used. In particular we will see that when commands `cp` and `mv`, and options `-a` and `-o` of commands `test` and `[ . ]`, are forbidden, we can check script traces in polynomial time. We define and explain properly the property which makes `mv` and `cp` special at the end of section 4.1.2, with definition 19.

Note that options `-a` and `-o` of commands `test` and `[ . ]` are forbidden because they perform the logical operations *and* and `or` respectively. In the implementation, the concrete interpreter splits each such call into several *trace atoms*, creating more execution cases and therefore more script traces. So the NP-hardness from logical operations is born by the concrete interpreter. This is why we can still afford to be polynomial on the problem of checking one script trace.

Chapter 3

A model of transducers for the CoLiS project

In this chapter, we propose a very simple notion of transducer for file systems that captures the actions of scripts. The closest notion of tree transformation in the literature is that of tree patterns or guarded rewriting rules. It is however much simpler than term rewritting systems in that the action of a transducer is limited to the application of at most one rule.

This chapter is organized as follows. Section 3.1 defines the model of tree pattern transducers. Section 3.2 presents the tree pattern transducers representing the behaviour of the usual script commands which interact with the file system. Section 3.3 presents an algorithm for computing the composition of tree pattern transducers.

3.1 Tree pattern transducers

Before formally defining tree pattern transducers, we show how it works on a few examples. Since Unix commands can apply different transformations on the file system depending on whether they succeed or fail, we use the trace atom[1] $success$(`comm`) to talk about the command `comm` when it succeed, and $failure$(`comm`) for when it fails.

For example the tree transformation associated with $success$(`mkdir /bin/foo`) adds a `dir` node at path `/bin/foo`, and its domain is the set of trees where there is a `dir` node at path `/bin` but there is no node at path `/bin/foo`. This tree transformation can be represented with the rewriting rule in Figure 3.1 and with the constraint $\mathsf{dir}(\texttt{/bin}) \wedge \neg\mathsf{node}(\texttt{/bin/foo})$.

Rewriting rules allow us to move or copy directories by using variables. A command may also apply different rules depending on the state of the file system. We can see this on the rewriting rules R_1 and R_2 of the command `mv /tmp /bin` in Figure 3.2. There is a different constraint for each rule. The constraint for rule R_1 is $C_1 = \mathsf{dir}(\texttt{/tmp}) \wedge \mathsf{dir}(\texttt{/bin}) \wedge \neg\mathsf{node}(\texttt{/bin/tmp})$, the constraint for rule R_2 is $C_2 = \mathsf{dir}(\texttt{/tmp}) \wedge \neg\mathsf{node}(\texttt{/bin})$. Note that those are not the only rules of `mv /tmp /bin` since it can also move other types of files than just directories.

[1] For trace atoms see definition 1

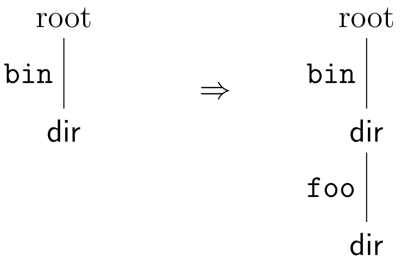


Figure 3.1 – Rewriting rule for *success*(`mkdir /bin/foo`)

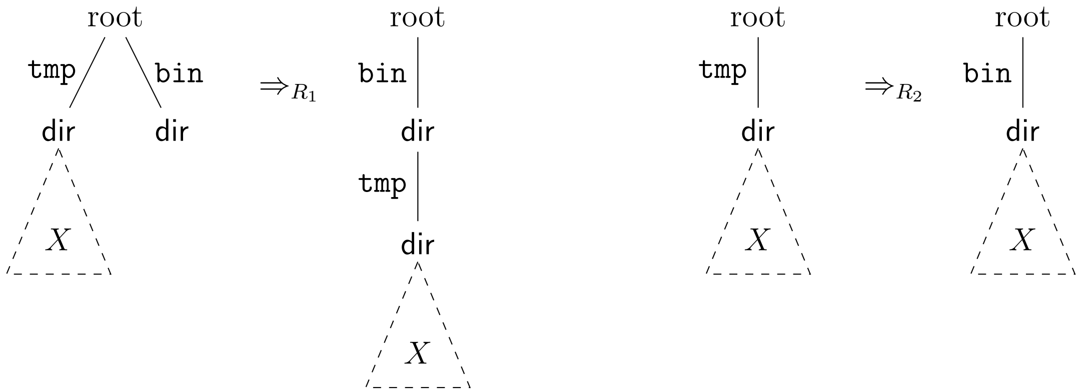


Figure 3.2 – Rules R_1 and R_2 for *success*(`mv /tmp /bin`)

Next we formally define *tree constraints* and *tree pattern rewriting rules.*

Tree constraints

Recall that `types` is the finite set of file types, and $\mathcal{F}$ is the infinite set of features (which represent filenames).

Definition 6 Let $p \in \mathcal{F}^*$ be a path and $E \subseteq \mathcal{F}$ be a finite or cofinite subset of $\mathcal{F}$, we define several types of *atomic tree constraints*:

- $\mathsf{filetype}(p)$, where $\mathsf{filetype} \in$ `types` is a file type and $p \in \mathcal{F}^*$ is a path. The semantics of this constraint is the set $\llbracket \mathsf{filetype}(p) \rrbracket$ containing the trees which have a node at path p of file type $\mathsf{filetype}$.
- $\exists(p, E)$, the *existential feature constraint* at path p on set the $E \subseteq \mathcal{F}$ of features. Its semantics is the set of trees in which p is a directory and there exists f in E such that *there is a node* at path p/f.

A *tree constraint* is a boolean combination of *atomic tree constraints.* Semantics of tree constraints are given by, for all atomic constraints C_1, C_2:

- $\llbracket C_1 \wedge C_2 \rrbracket = \llbracket C_1 \rrbracket \cup \llbracket C_2 \rrbracket$

- $[\![C_1 \vee C_2]\!] = [\![C_1]\!] \cap [\![C_2]\!]$
- $[\![\bot]\!] = \emptyset$
- $[\![\top]\!] = \mathcal{T}$
- $[\![\neg C_1]\!] = \mathcal{T} \setminus [\![C_1]\!]$

A *tree constraint literal* is either an *atomic tree constraint* or the negation of an *atomic tree constraint*.

For example the command `[ -f /bin/foo ]` only succeeds when there is a regular file at path `/bin/foo`. So we associate with the trace atom *success*(`[ -f /bin/foo ]`) the *tree constraint* $C = \mathsf{reg}(\texttt{/bin/foo})$.

The command `rmdir /boot` fails either when the directory `/boot` exists and is not empty, or when there is no directory `/boot`. So we associate with the trace atom $failure$(`rmdir /boot`) the *tree constraint* $C = \exists(\texttt{/boot}, \mathcal{F}) \vee \neg\mathsf{dir}(\texttt{/boot})$.

In order to simplify notations, we will use the following notations for all path p:

- $\mathsf{NonEmptyDir}(p) = \exists(p, \mathcal{F})$
- $\mathsf{EmptyDir}(p) = \mathsf{dir}(p) \setminus \exists(p, \mathcal{F})$
- $\mathsf{node}(p) = \bigcup_{t \in \texttt{types}} t(p)$
- $\mathsf{file}(p) = \mathsf{node}(p) \setminus \mathsf{dir}(p)$

Tree patterns

Definition 7 Given a set V of variables, *tree patterns* with variables in V are inductively defined (similarly to feature trees) as:

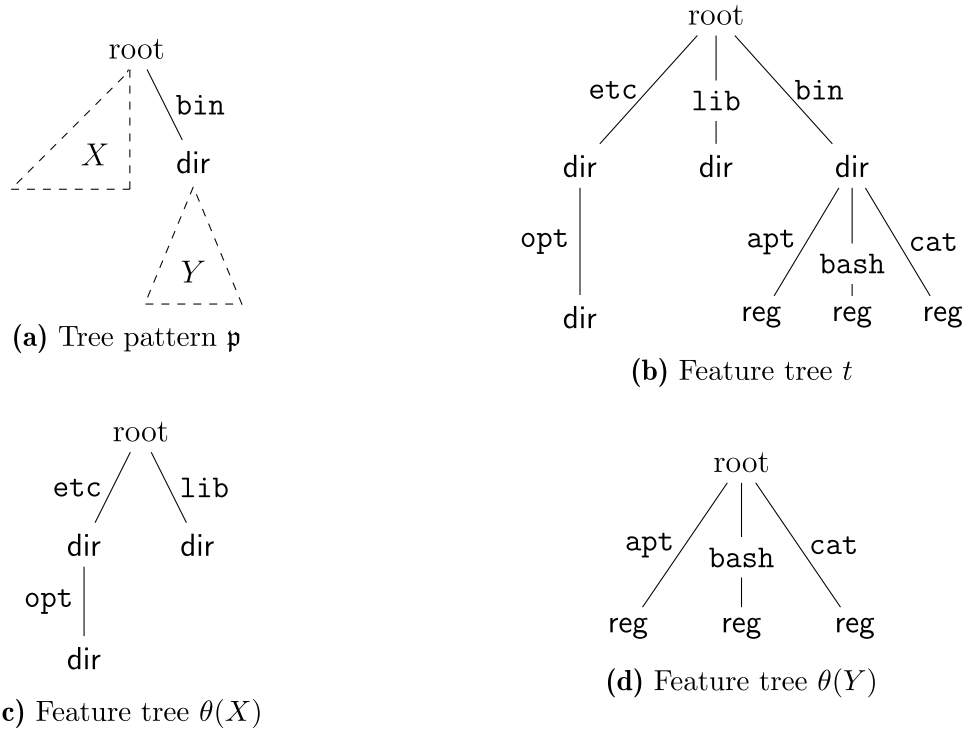
$$\mathcal{TP} = (V \cup \{\bot\}) \times (\mathcal{F} \rightsquigarrow ((\texttt{types} \setminus \{\mathsf{dir}\}) \cup (\{\mathsf{dir}\} \times \mathcal{TP})))$$

where $\mathcal{F} \rightsquigarrow X$ denotes the set of finite mappings (i.e. partial functions with finite domain) from $\mathcal{F}$ to X, and $\bot \notin V$ represents a default variable with which we associate the empty mapping.

We say that a *valuation* $\theta : V \to \mathcal{T}$ of the variables in V is *compatible* with a tree pattern $\mathfrak{p}$ if and only if:

- the variables in $\mathfrak{p}$ are in V,
- for each directory in $\mathfrak{p}$ of the form $(\mathsf{dir}, (x, \mathsf{map}))$, with $\mathsf{map} \in \mathcal{F} \rightsquigarrow ((\texttt{types} \setminus \{\mathsf{dir}\}) \cup (\{\mathsf{dir}\} \times \mathcal{TP}))$ and $x \in V$, the domains of the finite mappings map and $\theta(x)$ (i.e. the filenames of child nodes) are disjoint subsets of $\mathcal{F}$. This condition prevents the valuation from adding a file when there is already a file with the same name in the pattern.

If they are *compatible* then θ associates with $\mathfrak{p}$ the tree obtained from $\mathfrak{p}$ by substituting variables using θ, and substituting $\perp$ with the empty mapping, the resulting tree is noted $\theta(\mathfrak{p})$.



(a) Tree pattern $\mathfrak{p}$

(b) Feature tree t

(c) Feature tree $\theta(X)$

(d) Feature tree $\theta(Y)$

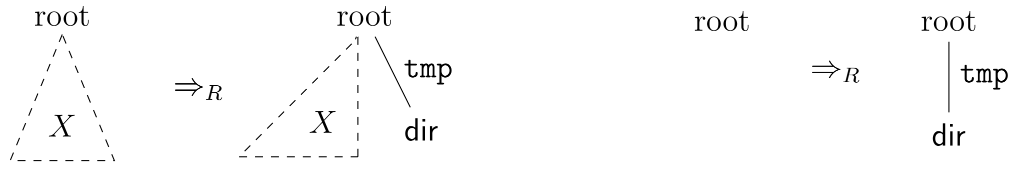
Figure 3.3 – Example of tree pattern $\mathfrak{p}$, feature tree t and valuation θ such that $t = \theta(\mathfrak{p})$

We can see how a valuation θ associates a pattern $\mathfrak{p}$ with a tree t in Figure 3.3. In this example θ is compatible with $\mathfrak{p}$ because $\theta(X)$ has no node at path `bin`.

The specificity of this model is that each variable represents a mapping associating features (i.e. filenames) with trees, but a node with a variable can still have child nodes given by the pattern. After a substitution is applied to a pattern, the set of child nodes of any given node becomes the union of the children given by the substitution and those given by the pattern. Because two child nodes must have distinct filenames, the child nodes given by the substitution and by the pattern must have distinct filenames. To forbid such *collisions* of filenames, a notion of *compatibility* between substitutions and patterns is introduced.

A benefit of this definition of patterns is that the matching between a tree and a pattern is *deterministic*, meaning if a tree t matches a pattern $\mathfrak{p}$ then there is a unique substitution θ such that $t = \theta(\mathfrak{p})$.

Definition 8 A *tree pattern rewriting rule* is given by a pair of tree patterns $(\mathfrak{p}_1, \mathfrak{p}_2)$ on a set V of variables such that each variable $X \in V$ occurs at most once in $\mathfrak{p}_1$. The pattern $\mathfrak{p}_1$ is called the input tree pattern and $\mathfrak{p}_2$ is the output tree pattern.



(a) Full version

(b) Simplified version

Figure 3.4 – Rewriting rule for trace atom *success*(`mkdir /tmp`)

Each *tree pattern rewriting rule* $R = (\mathfrak{p}_1, \mathfrak{p}_2)$ induces a relation g_R from trees to trees we call a *tree pattern relation* defined by:

$$g_R = \{(\theta(\mathfrak{p}_1), \theta(\mathfrak{p}_2)) \mid \theta \in V \to \mathcal{T}\}$$

For all pair of trees $(t_1, t_2) \in g_R$ we can also write: $t_1 \to_R t_2$.

The *domain* of the *tree pattern relation* g_R is

$$\mathrm{dom}(g_R) = \{\theta(\mathfrak{p}_1) \mid \theta \text{ is a valuation compatible with } \mathfrak{p}_1 \text{ and } \mathfrak{p}_2\}$$

For example the trace atom *success*(`mkdir /tmp`) is associated with the tree constraint ¬node(`/tmp`) and the rewriting rule shown in Figure 3.4. In order to simplify graphical representation of rules, we will omit variables when they appear exactly once on either side of a rule and at the same path. We show the full and simplified version of the rule of *success*(`mkdir /tmp`) respectively in Figure 3.4a and Figure 3.4b.

We use *input tree constraints* to put restrictions on the domain of *tree pattern rewriting rules*. *Tree constraints* allow more control over the domains of transducers than simple *tree pattern rewriting rule*. For example, the command `rmdir /bin` fails if and only if the directory at path `/bin` is not empty. We can express this domain with the tree constraint NonEmptyDir(`/bin`), but not with just a rewriting rule.

Tree constraints also allow simpler representation of error cases of commands, indeed these often consist in checking the existence or absence of a file or directory without changing the file system. In these cases the rewriting rule $R = ((X, []), (X, []))$, where $[]$ is the empty mapping and X is a variable, can perform the identity function, and the condition is expressed with a tree constraint.

Most importantly, input tree constraints characterize (by definition) the domains of tree pattern transducers. So the input tree constraints will be important in the implementation where we only compute the domains of compositions of transducers.

The action of most commands on the file system is deterministic, this means that any constraint expressed on the output of such a command can equivalently be expressed as a constraint on its input. This is useful when trying to compute the rewriting rule and input constraint of the composition of two commands. Indeed in a composition, the input constraint of the second command, which is a constraint on the output of the first command, has to be expressed as a constraint on the input of the first command.

However the command `cp` with option `-r` does not have a deterministic specification, or rather the specification leaves some choices to the implementer, so different implementations of the command may behave differently. When a directory is copied using `cp -r`, the files it contains are copied one by one in the destination directory either until all files are copied or until it tries to copy a file where there is already a file of the same name. In the second case the command stops and sends an error without reverting the files it has copied thus far. The specification does not say in which order the files should be copied, this means that, depending on the implementation of `cp` on your computer, when a `cp -r` fails in this way, some files may have been copied to the destination directory, but we cannot say which. This is one of the reasons why `cp -r` requires our transducer model to have tree constraints for both the input and output of rewriting rules (we will see this in detail when we present the exhaustive list of rules for `cp -r` in section 3.2).

Tree pattern Transducers

Definition 9 A *tree pattern transformation* is a tuple (C_i, R, C_o) composed of an input *tree constraint* C_i, a *tree pattern rewriting rule* R and an output *tree constraint* C_o such that $[\![C_i]\!] \subseteq g_R^{-1}(C_o)$.

Each *tree pattern transformation* $T = (C_i, R, C_o)$ induces a relation $g_T : \mathcal{T} \times \mathcal{T}$ from feature trees to feature trees defined by:

$$g_T = g_R \cap ([\![C_i]\!] \times [\![C_o]\!])$$

The domain $\mathrm{dom}(T)$ of T is the domain of g_T, and, since $[\![C_i]\!] \subseteq g_R^{-1}(C_o)$:

$$\mathrm{dom}(T) = \mathrm{dom}(g_T) = [\![C_i]\!]$$

A *tree pattern transducer* is a list $(T_1, \ldots, T_n)$ of *tree pattern transformations* whose domains $\mathrm{dom}(T_1), \ldots, \mathrm{dom}(T_n)$ are disjoint. Each *tree pattern transducer* τ induces the relation $g_\tau : \mathcal{T} \times \mathcal{T}$ defined by:

$$g_\tau = \bigcup_{1 \leq i \leq n} g_{R_i}$$

If the domains of two *tree pattern transducers* $\tau_1 = (T_1, \ldots, T_n)$ and $\tau_2 = (T_{n+1}, \ldots, T_m)$ are disjoint, then the *disjunction of* τ_1 *and* τ_2, noted $\tau_1 \vee \tau_2$, is the *tree pattern transducer* $\tau = (T_1, \ldots, T_m)$.

In summary, our model represents commands by giving a rewriting rule, a tree constraint on the input tree and a tree constraint on the output tree. But since our goal is to compute domains and inverse images of tree transformations (c.f. subsection 1.5.2), we will always express constraints on the input rather than on the output when possible. We will see later in section 3.2 that only the `cp -r` command introduces a constraint on its output, for other commands the output tree constraint will always be $\top$.

Next, for each trace atom, we will give a *tree pattern transducer* which represents its action on the file system. Then we will present a procedure to compute the composition of two *tree pattern transducers.*

Definition 10 We call *identity tree pattern transformation* the transformation (C, R, C') with $C = C' = \top$, its domain is the set of all feature trees $[\![C]\!] = \mathcal{T}$, and the rewriting rule R is the identity rule. We can represent this transformation as follows:

$$\begin{array}{|ccc|}\hline \multicolumn{3}{|c|}{C = \top} \\ \hline \text{root} & \Rightarrow_R & \text{root} \\ \hline \multicolumn{3}{|c|}{C' = \top} \\ \hline \end{array}$$

We call *identity tree pattern transducer* the transducer composed only of the *identity tree pattern transformation.*

We call *empty tree pattern transducer* the transducer described by the empty list, its domain is the empty set $\emptyset$.

3.2 Formalisation of Unix commands

We make a list of the usual script commands interacting with the file system, including commands which modify the file system like `mv` and `cp`, and commands which test things on the file system like `test` and `which`. For each corresponding trace atom (a command either succeeding or failing, c.f. definition 1), we give a tree pattern transducer representing its action of the file system. The precise specifications of those commands is informed by [20] and the base specification[2] provided by the IEEE (Institute of Electrical and Electronics Engineers) and The Open Group.

For all commands other than `cp` with option `-r` there is no output tree constraint (their output tree constraint is $\top$), so we do not represent it. Usually, in path arguments of commands, we distinguish the last feature of the path (the deepest component of the path) from the rest of the path, we do this by noting path arguments p/f where f denotes the last feature and p denotes the rest of the path (path p can be the empty path). In the graphical representation of rewriting rules, a solid edge between two nodes represents their link by a single feature (the bottom node is a daughter of the top node and its filename is the label of the edge), while a dotted edge between two nodes represents their link by a possibly longer path (the top node is an ancestor of the bottom node and the path from the one to the other is the label of the edge).

In most cases the input tree constraint of the failure case is the negation of the input constraint of the success case. In other cases the model of tree constraints is not expressive enough to represent the exact set of file systems on which the command fails or succeeds. In such cases the constraints we use are *overapproximations* of the sets of trees which they should represents. This means that, in the success case of the command `cp -r` p_s/f_s p_d/f_d for example, we give a constraint that allows file systems which would, according to the specification, induce the command to fail. We prefer overapproximation to underapproximation because we prefer to have false positives, i.e. seeing an error when there is none, rather than false negatives, i.e. not seeing an error when there is one.

[2] Specification of Unix commands: `https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/`

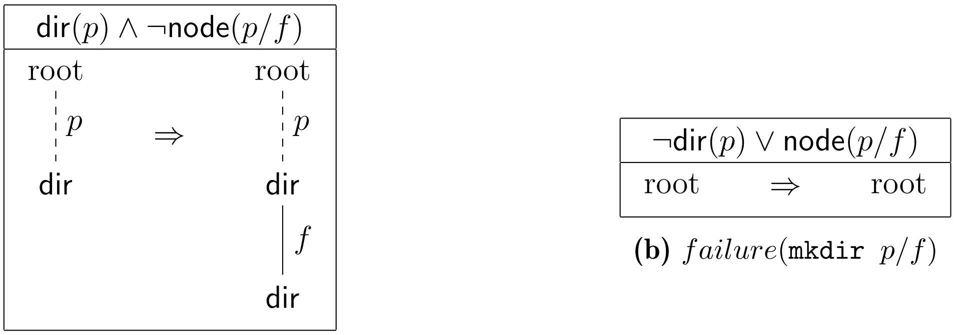
3.2.1 `mkdir`

The `mkdir` command is used to create an empty directory at a given path. First we define its behaviour when used on one argument path.

$$\texttt{mkdir}\ p/f$$

If f is `.` or `..` then the command always fails. We therefore associate, with the success case of this command, the empty tree pattern transducer (its domain of definition is the empty set $\emptyset$). The failure case of this command corresponds to the identity tree pattern transformation.

Otherwise the success and failure cases of `mkdir` p/f perform the transformations shown in Figure 3.5a and Figure 3.5b respectively. The input constraint of $success(\texttt{mkdir}\ p/f)$ is $\mathsf{dir}(p) \wedge \neg\mathsf{node}(p/f)$, it means that, before running the command `mkdir` p/f, the file system should have a directory at path p but should have no node at path p/f. Note that the input constraint of the failure case is the negation of that.



(a) $success(\texttt{mkdir}\ p/f)$

(b) $failure(\texttt{mkdir}\ p/f)$

Figure 3.5 – Transformations representing the behaviour of `mkdir` p/f

The option `-p` changes the action of `mkdir` on the file system. The command
`mkdir -p` $f_1/\ldots/f_n$ acts like the following composition of commands:
`mkdir` $f_1/\ldots/f_i$
`mkdir` $f_1/\ldots/f_i/f_{i+1}$
$\vdots$
`mkdir` $f_1/\ldots\ldots\ldots/f_{n-1}$
`mkdir` $f_1/\ldots\ldots\ldots/f_{n-1}/f_n$

where $f_1/\ldots/f_i$ is the smallest prefix of $f_1/\ldots/f_n$ that is not a directory in the file system. Every such composition of `mkdir` commands can be expressed as a tree pattern transducer (we show in section 3.3 how to compute the composition of tree pattern transducers). We note τ_i the transducer obtained from this composition of commands by replacing the input tree constraint by its conjunction with the constraint $\mathsf{dir}(f_1/\ldots/f_{i-1}) \wedge \neg\mathsf{dir}(f_1/\ldots/f_i)$, this enforces the condition that $f_1/\ldots/f_i$ is the smallest prefix of $f_1/\ldots/f_n$. In the particular case of τ_{n+1}, the input tree constraint is replaced by its conjunction with $\mathsf{dir}(f_1/\ldots/f_n)$ instead. Then the behaviour of the success of command `mkdir -p` $f_1/\ldots/f_n$ is the disjunction of tree pattern transducers:

$$\bigcup_{1\leq i\leq n+1} \tau_i$$

The failure of command `mkdir -p` $f_1/\ldots/f_n$ is associated with the empty tree pattern transducer because the command never fails.

When `mkdir` is run with several target paths, the command behaves as the composition of `mkdir` commands on each individual target path, with the exception that if a `mkdir` fails then the following `mkdir` commands are still run (normally the script would stop at the first failing command). It returns an error if at least one of the individual commands fails. So the success case would simply be the composition of success cases. The failure case would be a list of transformations, each transducer being the composition of `mkdir` commands, each command either succeeding or failing, but such that at least one command fails.
We call this special kind of composition *uninterrupted composition*, in our implementation it is handled at the level of the concrete interpreter (c.f. section 2.2). This means that the `mkdir` trace atoms treated by our symbolic interpreter have only one path argument.

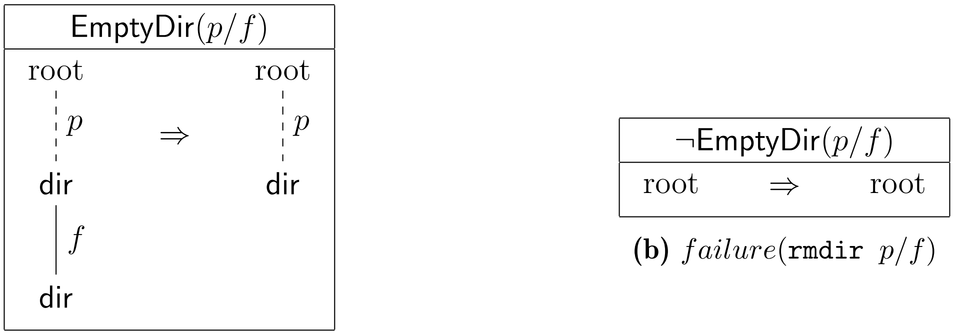
3.2.2 `rmdir`

The `rmdir` command is used to remove a directory at a given path, assuming that this direcory is empty. To remove a directory along with its content one should use the command `rm` with option `-r`. First we define its behaviour when used on one argument path.

$$\texttt{rmdir}\ p/f$$

Like `mkdir` p/f, `rmdir` p/f always fails if f is `.` or `..` , the success case is the empty tree pattern transducer and the failure case is the identity tree pattern transformation:

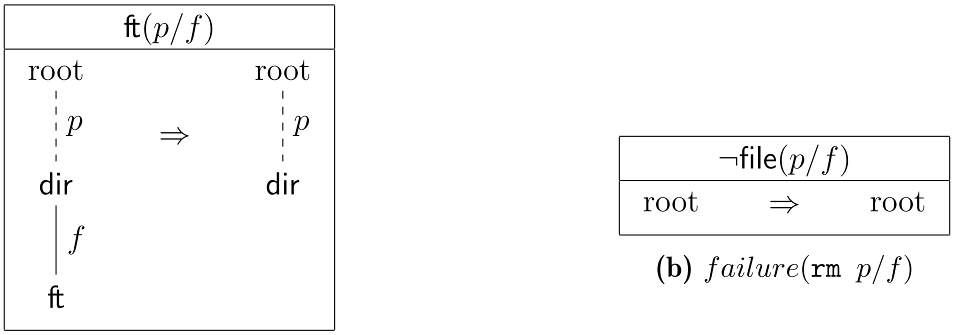
Otherwise the success and failure cases of `rmdir` p/f perform the transformations shown in Figure 3.6a and Figure 3.6b respectively.



(b) $\mathit{failure}(\texttt{rmdir}\ p/f)$

(a) $\mathit{success}(\texttt{rmdir}\ p/f)$

Figure 3.6 – Transformations representing the behaviour of `rmdir` p/f



(a) Transformations for $success($`rm` $p/f)$, one for each ft $\in$ {reg, symlink, fifo, block, char, sock}

(b) $failure($`rm` $p/f)$

Figure 3.7 – Transformations for command `rm` p/f

The option `-p` makes `rmdir` run one-by-one on each prefix of its target path, except the empty path, starting with the longest prefix.

Similarly to `mkdir`, when `rmdir` is run with several target paths, the command behaves as the *uninterrrupted composition* of `rmdir` run one-by-one on each target path.

3.2.3 `rm`

The `rm` command is used to remove a file at a given path, or, when the option `-r` is used, a directory (along with its content). First we define its behaviour when used on one argument path.

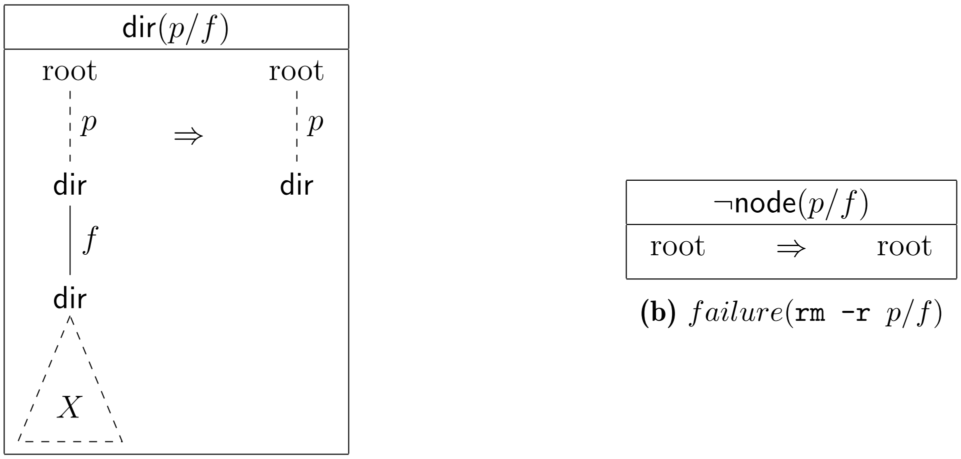
`rm` p/f

If f is `.` or `..` then `rm` p/f always fails, `rm /` also always fails, either way the success case is the empty transducer and the failure case is the identity transducer.

Otherwise the success case has several transformations, one for each type the file at path p/f can have. For each non-directory file type ft $\in$ {reg, symlink, fifo, block, char, sock} we note T_{ft} the corresponding transformation of $success($`rm` $p/f)$. Instead of representing the 6 transformations, we represent only one, but with a symbol ft which can represent any of the six non-directory file type (we only do this to use less paper). The failure case has only one transformation. The transformations for the success and failure cases of `rm` p/f are shown in Figure 3.7a and Figure 3.7b respectively. We recall that file denotes the disjunction of all the non-directory file types:
file(p/f) = reg$(p/f) \vee$ symlink$(p/f) \vee$ fifo$(p/f) \vee$ block$(p/f) \vee$ char$(p/f) \vee$ sock(p/f)

The `-r` option allows the removal of a directory. So, in the success case, we add a 7th transformation to the first 6 used for `rm` p/f. The failure case of `rm -r` p/f has only one transformation. Those are shown in Figure 3.8a and Figure 3.8b respectively. In the graphical representation of rules we omit variables only when they appear at the same path in the input as in the output, this is why for $success($`rm -r` $p/f)$ we make variable X explicit.

The `-f` option prevents the command to return an error. In our setting this means that



(a) 7th transformation of *success*(`rm -r` p/f)

(b) *failure*(`rm -r` p/f)

Figure 3.8 – Transformations for `rm` p/f with option `-r`

the success case becomes the disjunction of the success and failure cases of the corresponding command without the `-f` option, and the failure case becomes the empty transducer.

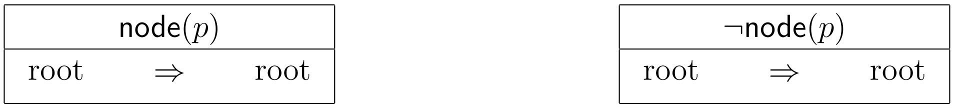
When `rm` is run with several target paths, the command behaves as the *uninterrrupted composition* of `rm` run one-by-one on each target path.

3.2.4 `touch`

The `touch` command is used to make sure that a file exists at a given path, creating an empty file if there is not already one at the given path. First we define its behaviour when used on one argument path.

`touch` p/f

If f is `.` or `..` then the success and failure cases both have one transformation with the identity rule. They are respectively shown in Figure 3.9a and Figure 3.9b.



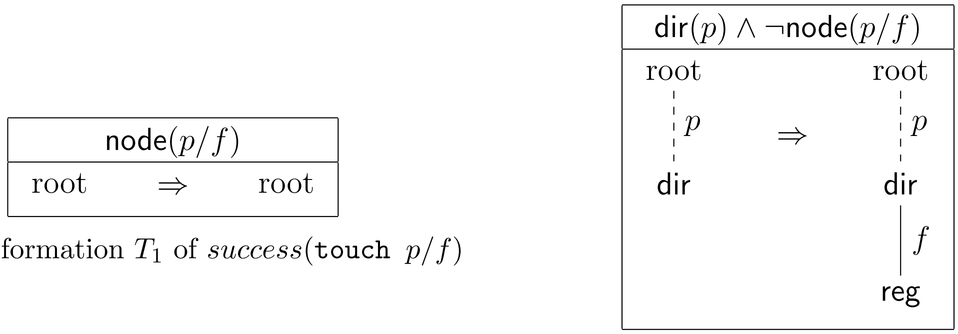
(a) *success*(`touch` p/f)

(b) *failure*(`touch` p/f)

Figure 3.9 – Case of `touch` p/f where f = `.` or f = `..`

Otherwise the success case has two transformations: either the target path already exists or the file has to be created. The transformations for `touch` p/f are shown in Figures 3.10 and Figure 3.11. Here reg is the type of regular files, it is the type created by default by command `touch`.

When `touch` is run with several target paths, the command behaves as the *uninterrrupted composition* of `touch` run one-by-one on each target path.



(a) Transformation T_1 of $success$(`touch` p/f)

(b) Transformation T_2 of $success$(`touch` p/f)

Figure 3.10 – $success$(`touch` p/f)

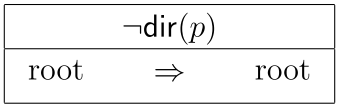


Figure 3.11 – $failure$(`touch` p/f)

3.2.5 `test` and `[ ]`

Commands `touch` and `[` are very similar, the only difference between the two is that `[` requires to add `]` after the arguments of the command. Everything we say about `test` is also true of `[`.

These two commands are mostly used as conditions in conditionnal statements. They can check a variety of things, but here we only care about their interactions with the file system, so we only list their behaviour with the options which check properties of the file system. Because they only check properties of the file system without modifying it, the rewriting rule for each case will be the identity rule. So for each option we only specify the corresponding tree constraints (one in case of success and one in case of failure):

- `test -b` p
  Success: block(p)
  Failure: ¬block(p)

- `test -c` p
  Success: char(p)
  Failure: ¬char(p)

- `test -d` p
  Success: dir(p)
  Failure: ¬dir(p)

- `test -e` p
  Success: node(p)
  Failure: ¬node(p)

- `test -f` p
  Success: $\mathsf{reg}(p)$
  Failure: $\neg\mathsf{reg}(p)$

- `test -h` p
  Success: $\mathsf{symlink}(p)$
  Failure: $\neg\mathsf{symlink}(p)$

- `test -p` p
  Success: $\mathsf{fifo}(p)$
  Failure: $\neg\mathsf{fifo}(p)$

- `test -S` p
  Success: $\mathsf{sock}(p)$
  Failure: $\neg\mathsf{sock}(p)$

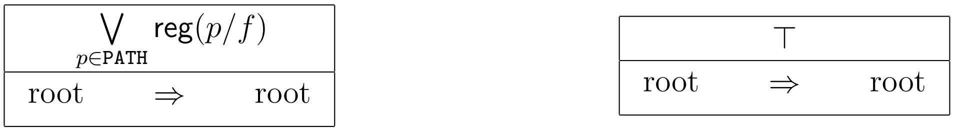
- `test -[GOgkrsuwx]` p
  Each option among `-G`, `-O`, `-g`, `-k`, `-r`, `-s`, `-u`, `-w` and `-x` succeeds if the file at target path verifies some condition not represented in our model. In such case we represent an overapproximation of the behaviour of the commands. If the `test` succeeds then the target path must exist, but in the failure case we cannot deduce anything with our abstraction of file systems. So the constraints are:
  Success: $\mathsf{node}(p)$
  Failure: $\top$

- `test` p_1 `-nt` p_2, `test` p_1 `-ot` p_2
  These options are used to compare the dates of two files, again the input constraints are overapproximations:
  Success: $\mathsf{node}(p_1) \wedge \mathsf{node}(p_2)$
  Failure: $\top$

- `test ! arg`
  Option `!` is used to invert the result of the command, so that `test ! arg` succeeds if `test arg` fails and `test ! arg` fails if `test arg` succeeds. In our framework it means that we take the transducers for the success and failure cases of the command `test arg` and swap them.

- `test` $\texttt{arg}_1$ `-a` $\texttt{arg}_2$
  The `-a` option is used to compute the conjunction of two `test` commands (`-a` stands for `and`). Noting $C_{s1}, C_{f1}, C_{s2}, C_{f2}$ the input tree constraints respectively for the success and failure of `test` $\texttt{arg}_1$ and the success and failure of `test` $\texttt{arg}_2$, the constraints for `test` $\texttt{arg}_1$ `-a` $\texttt{arg}_2$ are:
  Success: $C_{s1} \wedge C_{s2}$
  Failure: $C_{f_1} \vee C_{f_2}$

- `test` $\texttt{arg}_1$ `-o` $\texttt{arg}_2$
  The `-o` option is used to compute the disjunction of two `test` commands (`-o` stands



(a) $success($`which` $f)$ **(b)** $failure($`which` $f)$

Figure 3.12 – Transformations of `which` f

for `or`). Noting $C_{s1}, C_{f1}, C_{s2}, C_{f2}$ the input tree constraints respectively for the success and failure of `test arg`$_1$ and the success and failure of `test arg`$_2$, the constraints for `test arg`$_1$ `-o arg`$_2$ are:
Success: $C_{s1} \vee C_{s2}$
Failure: $C_{f_1} \wedge C_{f_2}$

3.2.6 `which`

We start by defining the behaviour of `which` on one argument.

`which` p/f

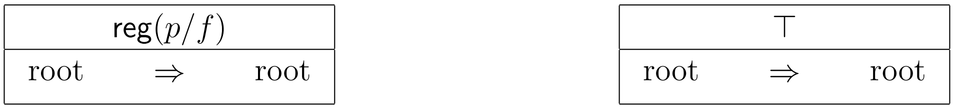
A `which` command you might think of is somehow an extension of the `test -x` command, which checks if the name given as argument appears as an executable file in the directories listed in the `PATH` variable. In the implementation, the concrete interpreter transforms such calls into the corresponding composition of `test -x` commands. Here we specify the `which` utility installed by the `debianutils` package of Debian, which can take a longer path as argument. If the argument of `which` contains at least one `/` then the command does not look in the `PATH`: it just checks that the given path leads to an executable regular file.

If the argument of `which` is a single filename f, we get the transformations shown in Figure 3.12a and Figure 3.12b.

Since we do not represent executability of regular files in our model of file systems, we must approximate the constraints. In the success case that means we only check that there is a regular file, not that it is executable. In the failure case this means that we cannot check anything on the file system, since `which` can fail because the file is either absent, a non-regular file or a regular non-executable file.

When `which`'s argument is not a single filename, but contains some `/`, then `which` does not look in the `PATH` variable but instead checks if the path given as argument is an executable file. In this case, command `which` p/f acts on the file system like `test -f` p/f `-a -x` p/f.
Again we overapproximate its behaviour because we do not model executability of files. The transformations of the success and failure cases are shown in Figure 3.12a and Figure 3.12b respectively.

When `which` is applied to several arguments, it behaves as the *uninterrupted composition* (c.f. section 2.2) of `which` commands run on each argument.



(a) *success*(`which` p/f) **(b)** *failure*(`which` p/f)

Figure 3.13 – Transformations of `which` p/f

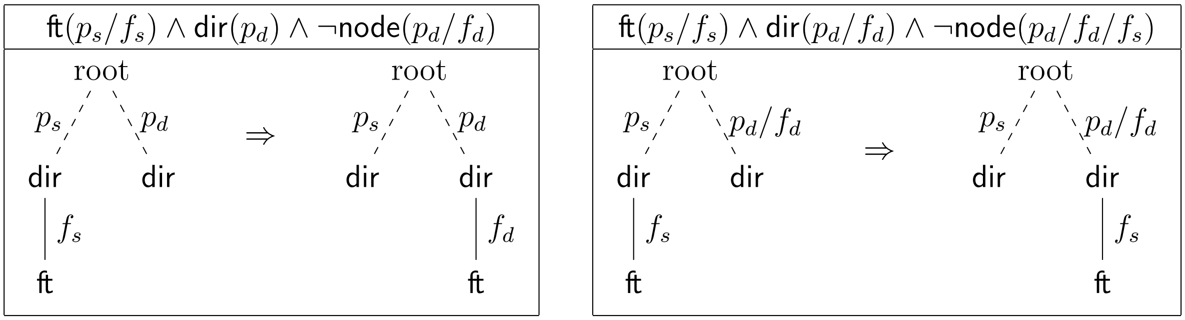
3.2.7 `mv`

The `mv` command is used to move a file or directory from a given source path, noted here p_s/f_s, to a given target path, noted here p_d/f_d.

$$\texttt{mv}\ \ p_s/f_s\ \ p_d/f_d$$

The behaviour of `mv` first depends on whether the target path p_d/f_d is a directory in the file system: if it is not a directory then `mv` takes the content of path p_s/f_s and puts it at path p_d/f_d, otherwise it puts it at path $p_d/f_d/f_s$ instead. We call p_s/f_s the source path and, p_d/f_d or $p_d/f_d/f_s$ (depending on if p_d/f_d is a directory), we call the destination path. The transformations for `mv` will often have two variants depending on the destination path.

If the source path is an ancestor of the destination path then `mv` always fails. Otherwise the success case is composed of the transformations in Figures 3.14, 3.15 and 3.16.



(a) Transformation $T_{1,\mathsf{ft}}$, one for each $\mathsf{ft} \in \texttt{types} \setminus \{\mathsf{dir}\}$

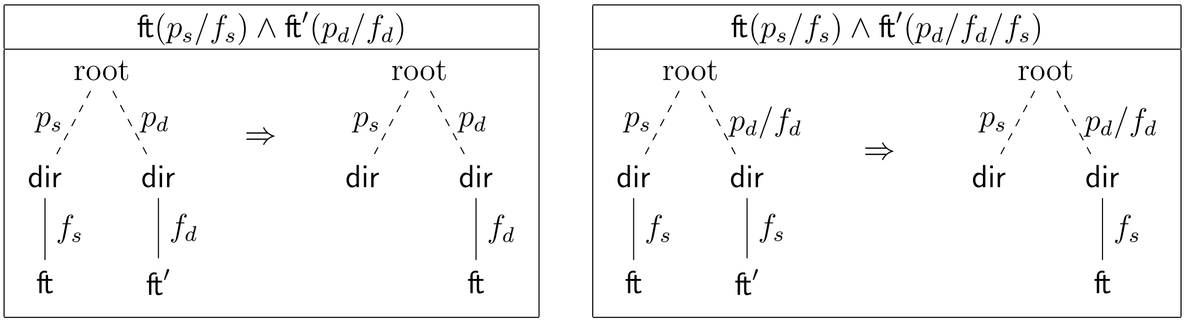
(b) Transformation $T_{2,\mathsf{ft}}$, one for each $\mathsf{ft} \in \texttt{types} \setminus \{\mathsf{dir}\}$

Figure 3.14 – Transformations of *success*(`mv` p_s/f_s p_d/f_d)

First, each non-directory file type $\mathsf{ft} \in \texttt{types} \setminus \{\mathsf{dir}\}$ can be moved. This leads to twelve transformations, or two if we consider them up to the filetype $\mathsf{ft} \in \texttt{types} \setminus \{\mathsf{dir}\}$. We note them $T_{1,\mathsf{ft}}$ and $T_{2,\mathsf{ft}}$, they respectively have destination path p_d/f_d and $p_d/f_d/f_s$. They are shown in Figure 3.14a and Figure 3.14b.

Next, for each non-directory file types $\mathsf{ft}, \mathsf{ft}' \in \texttt{types} \setminus \{\mathsf{dir}\}$, if a file of type ft' is already at path p_d/f_d then it is overwritten. We get the transformations $T_{3,\mathsf{ft},\mathsf{ft}'}$ and $T_{4,\mathsf{ft},\mathsf{ft}'}$ shown in Figure 3.15a and Figure 3.15b.

Finally, if we have a directory at path p_s/f_s then we have transformations T_5, T_6 and T_7 shown in Figure 3.16.

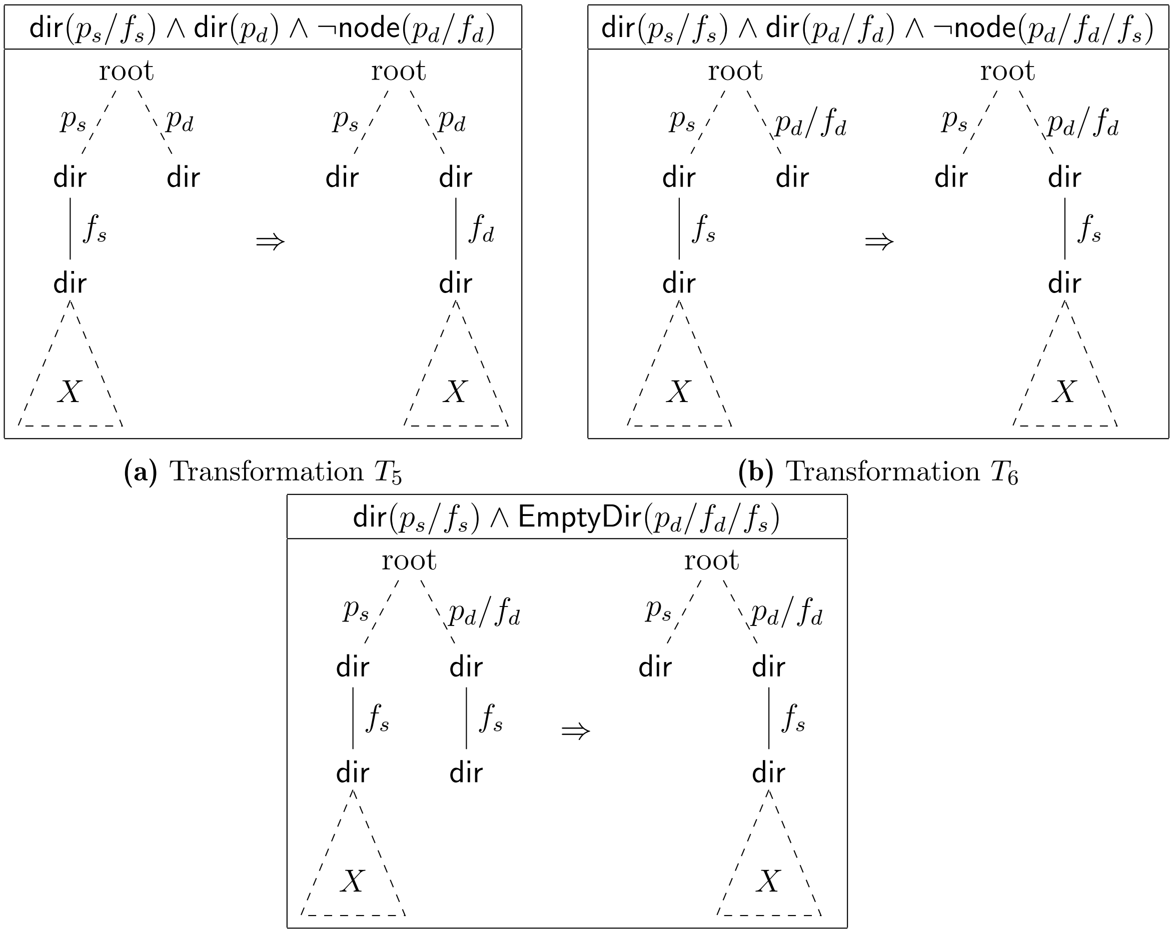


$\mathsf{ft}(p_s/f_s) \wedge \mathsf{ft}'(p_d/f_d)$
root
p_s
p_d
dir
f_s
f_d
ft
ft′
⇒
$\mathsf{ft}(p_s/f_s) \wedge \mathsf{ft}'(p_d/f_d/f_s)$
p_d/f_d

(a) Transformation $T_{3,\mathsf{ft},\mathsf{ft}'}$, one for each $\mathsf{ft}, \mathsf{ft}' \in \texttt{types} \setminus \{\mathsf{dir}\}$

(b) Transformation $T_{4,\mathsf{ft},\mathsf{ft}'}$, one for each $\mathsf{ft}, \mathsf{ft}' \in \texttt{types} \setminus \{\mathsf{dir}\}$

Figure 3.15 – Transformations of $success(\texttt{mv}\ \ p_s/f_s\ \ p_d/f_d)$



$\mathsf{dir}(p_s/f_s) \wedge \mathsf{dir}(p_d) \wedge \neg\mathsf{node}(p_d/f_d)$
$\mathsf{dir}(p_s/f_s) \wedge \mathsf{dir}(p_d/f_d) \wedge \neg\mathsf{node}(p_d/f_d/f_s)$
$\mathsf{dir}(p_s/f_s) \wedge \mathsf{EmptyDir}(p_d/f_d/f_s)$
root
p_s
p_d
p_d/f_d
dir
f_s
f_d
X
⇒

(a) Transformation T_5

(b) Transformation T_6

(c) Transformation T_7

Figure 3.16 – Transformations of $success(\texttt{mv}\ \ p_s/f_s\ \ p_d/f_d)$

Transformation T_7 has no equivalent for when the destination path is p_d/f_d, this is because this case requires the destination path to be an empty directory, whereas the

destination path can only be p_d/f_d if there is no directory at this path.
If the destination path is a non-empty directory then `mv` fails (we will later see that it is not the case for `cp -r`). The failure case is in Figure 3.17.

$$\begin{array}{|c|}\hline \neg\mathsf{node}(p_s/f_s) \vee \neg\mathsf{dir}(p_d) \vee (\mathsf{file}(p_s/f_s) \wedge \mathsf{dir}(p_d/f_d)) \\ \vee(\mathsf{dir}(p_s/f_s) \wedge (\mathsf{file}(p_d/f_d) \vee \mathsf{NonEmptyDir}(p_d/f_d))) \\ \hline \text{root} \quad \Rightarrow \quad \text{root} \\ \hline \end{array}$$

Figure 3.17 – Transformation for $failure($`mv` p_s/f_s $p_d/f_d)$

In the special case where the destination path of the `mv` command ends with a '/' but is not just '/', the command can move a directory only if the target path p_d/f_d is already a directory. In this special case transformation T_5 is removed from the success case, and the constraint for the failure case is replaced with its disjunction with the constraint $\mathsf{dir}(p_s/f_s) \wedge (\mathsf{dir}(p_d) \vee \neg\mathsf{node}(p_d/f_d))$ (i.e. the constraint of the transformation removed from the success case).

When `mv` is used on more than two arguments, it behaves similarly to other commands when they are used on several arguments. The command:
`mv` $p_1/f_1 \ldots p_n/f_n$ p_d/f_d behaves like the *uninterrupted composition* of the commands:
`mv` p_1/f_1 p_d/f_d`;` ... `mv` p_n/f_n p_d/f_d .

3.2.8 `cp`

The `cp` command is used to copy a file or a directory from a given source path, noted here p_s/f_s, to a given target path, noted here p_d/f_d.

`cp` p_s/f_s p_d/f_d

The behaviour of `cp`, similarly to `mv`, depends on whether the target path p_d/f_d is a directory in the file system: if it is not a directory then the destination path is p_d/f_d, otherwise it is $p_d/f_d/f_s$. The transformations for `cp` will often have two variants depending on the destination path.

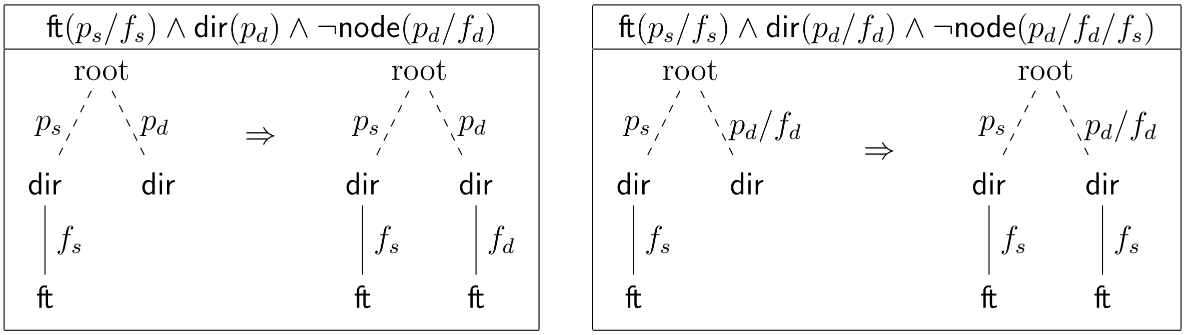
If the source path is an ancestor of the destination path then `cp` always fails. Otherwise the success case is composed of the tree pattern transformations of Figures 3.18 and 3.19.

First, each non-directory file type $\mathsf{ft} \in \mathtt{types} \setminus \{\mathsf{dir}\}$ can be copied. This leads to two transformations per possible file type: Figure 3.18a and Figure 3.18b.

Next, for each non-directory file types $\mathsf{ft}, \mathsf{ft}' \in \mathtt{types} \setminus \{\mathsf{dir}\}$, if a file of type ft' is already at path p_d/f_d then it is overwritten. We get the transformations of Figure 3.19. The failure case has the transformation shown in Figure 3.20

`cp` with option `-r`

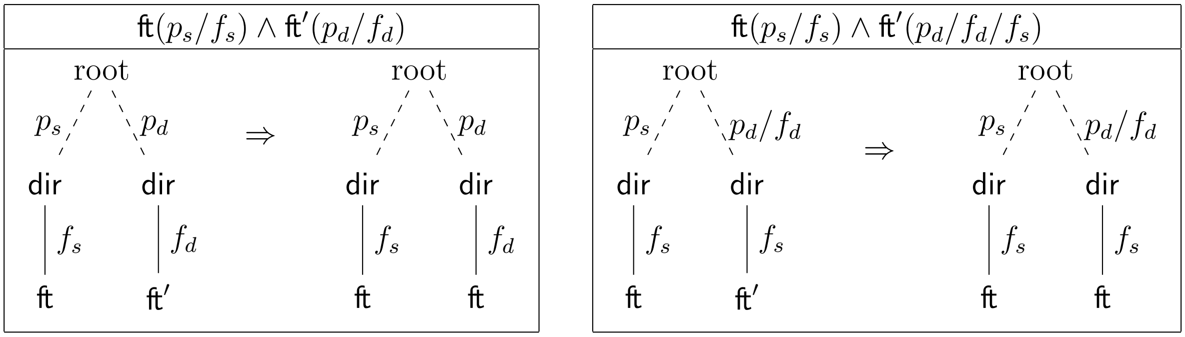
With the option `-r`, `cp` can copy directories. In this setting, for the success case, we keep the four tree pattern transformations defined above but we add some more for the cases where we copy a directory.



(a) Transformation $T_{1,\mathsf{ft}}$, one for each $\mathsf{ft} \in \mathtt{types} \setminus \{\mathsf{dir}\}$

(b) Transformation $T_{2,\mathsf{ft}}$, one for each $\mathsf{ft} \in \mathtt{types} \setminus \{\mathsf{dir}\}$

Figure 3.18 – Transformations of $success$(`cp` p_s/f_s p_d/f_d)



(a) Transformation $T_{3,\mathsf{ft},\mathsf{ft}'}$, one for each $\mathsf{ft}, \mathsf{ft}' \in \mathtt{types} \setminus \{\mathsf{dir}\}$

(b) Transformation $T_{4,\mathsf{ft},\mathsf{ft}'}$, one for each $\mathsf{ft}, \mathsf{ft}' \in \mathtt{types} \setminus \{\mathsf{dir}\}$

Figure 3.19 – Transformations of $success$(`cp` p_s/f_s p_d/f_d)

For the case where we have a directory at path p_s/f_s we add the transformations shown in Figure 3.21 and Figure 3.22.

The transformation shown in Figure 3.22 has no equivalent for when the destination path is p_d/f_d, this is because this case requires the destination path to be an empty directory, whereas the destination path can only be p_d/f_d if there is no directory at this path.

Contrary to `mv`, the `cp` command with option `-r` can succeed when the destination directory is not empty. It will succeed if and only if there are no collisions between filenames present in the copied directory and filenames present in the destination directory. This case could not have been exactly represented in our model. At this point we have to

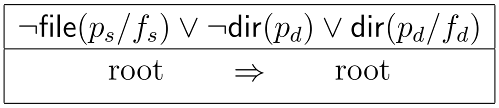
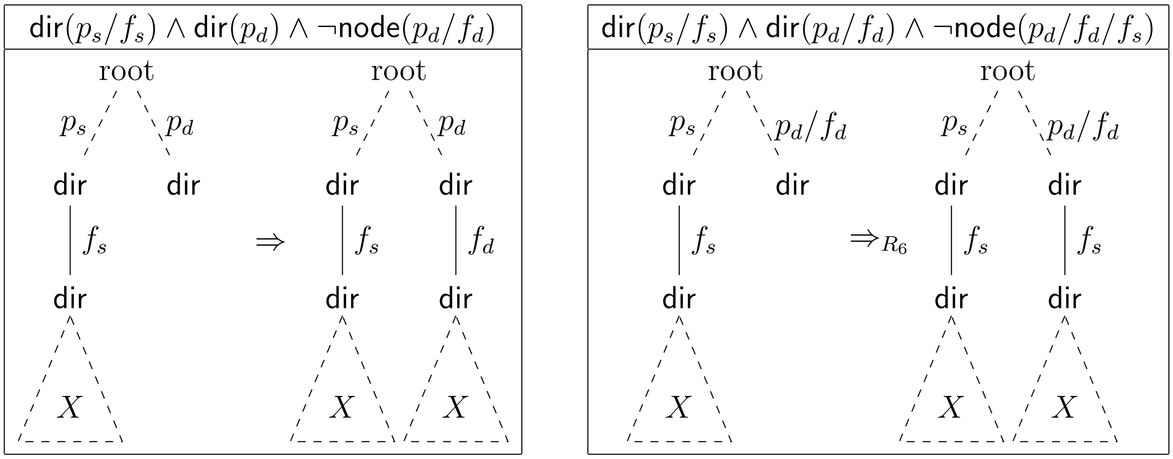


Figure 3.20 – Transformation of $failure$(`cp` p_s/f_s p_d/f_d)


dir(p_s/f_s) ∧ dir(p_d) ∧ ¬node(p_d/f_d)
root
p_s
p_d
dir
f_s
f_d
⇒
X
dir(p_s/f_s) ∧ dir(p_d/f_d) ∧ ¬node(p_d/f_d/f_s)
p_d/f_d
⇒R6

(a) Transformation T_5

(b) Transformation T_6

Figure 3.21 – Transformations of $success($`cp -r` $p_s/f_s\ \ p_d/f_d)$

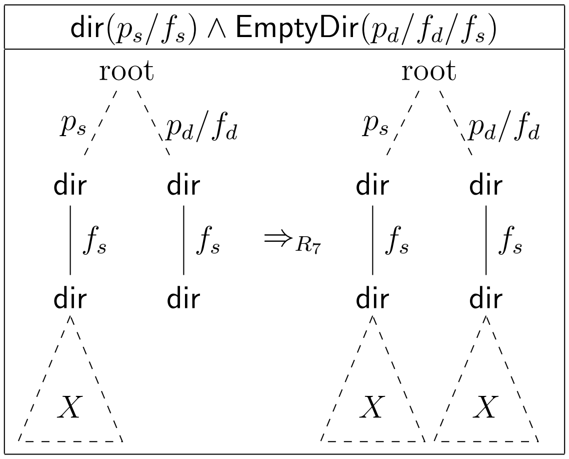


Figure 3.22 – Transformation T_7 of $success($`cp -r` $p_s/f_s\ \ p_d/f_d)$

sacrifice something, either by altering the model or by approximating the behaviour of the command. We choose to do a bit of both: we use an approximation of the behaviour of the command and we allow tree transformations to have tree constraints on their output.

The transformation is shown in Figure 3.23. Note that we chose not to omit variable X although it appears at the same path in the input pattern as in the output pattern (we could have omitted it to simplify the graphical representation). The rule of this transformation forgets all about the relation between variables X, Y and Z. This is an overapproximation of the behaviour of the `cp -r` command, according to its specification the variables X and Y should be mappings of disjoint domains, and their union should be Z. We discuss this approximation later.

The failure case of `cp -r` has two rules. This is because `cp -r` can fail while still modifying the file system. Indeed, when a directory is copied, the files and subdirectories inside it are copied one by one and, when a collision of filenames happens in the destination

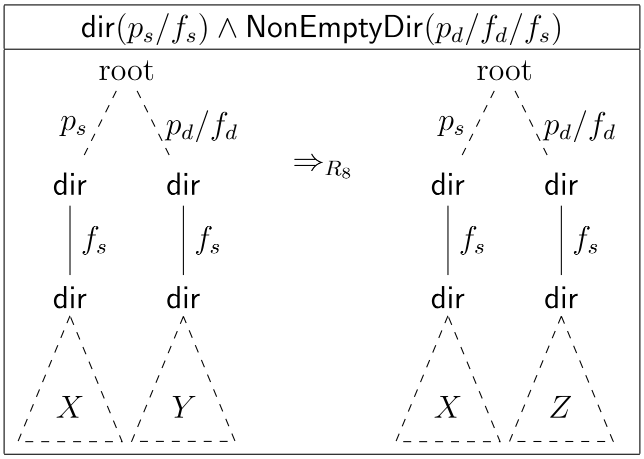
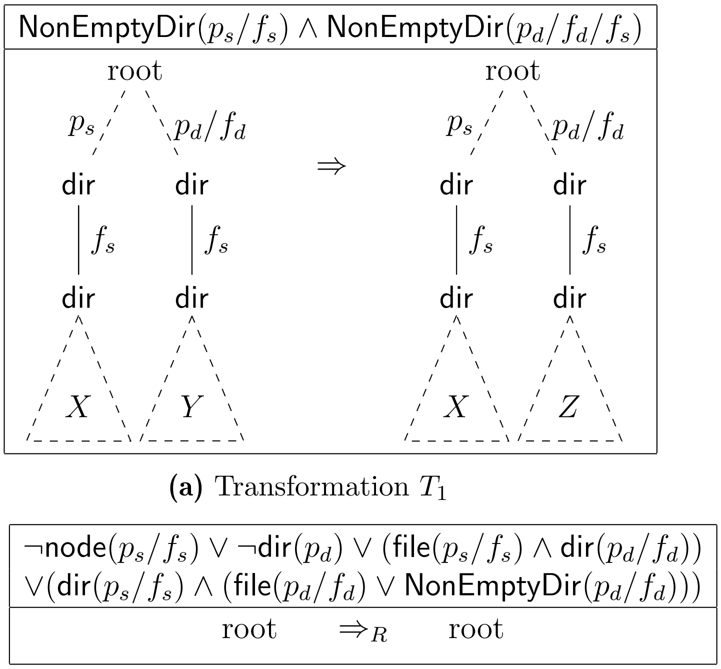


Figure 3.23 – Last transformation T_8 of *success*(`cp -r` p_s/f_s p_d/f_d)

directory, the `cp -r` command stops *without reverting the changes made thus far*. This means that the command can fail and still copy part of a directory. For this case we also use an overapproximation, similar to that of transformation T_8 from the success case (the difference is that the copied directory cannot be empty). Both transformations of $failure$(`cp -r` p_s/f_s p_d/f_d) are represented in Figure 3.24.


NonEmptyDir(p_s/f_s) ∧ NonEmptyDir($p_d/f_d/f_s$)
root
p_s
p_d/f_d
dir
f_s
$\Rightarrow$
X
Y
Z
¬node(p_s/f_s) ∨ ¬dir(p_d) ∨ (file(p_s/f_s) ∧ dir(p_d/f_d))
∨(dir(p_s/f_s) ∧ (file(p_d/f_d) ∨ NonEmptyDir(p_d/f_d)))
root
$\Rightarrow_R$
root

(a) Transformation T_1

(b) Transformation T_2

Figure 3.24 – The 2 transformations of $failure$(`cp -r` p_s/f_s p_d/f_d)

In transformation T_1 we could have made variable X implicit but again we chose not to. Removing the approximation amounts to stating that X and Y are mappings with

non-disjoint domains, and that Z is the union of Y with any part of X disjoint from Y. The part of X which is added to Y represent the files and subdirectories that were copied before the `cp -r` command met a collision of filenames between the source and the destination directories. Since there is no standard for the order in which files and subdirectories should be copied, the part of X which is copied into the destination directory can be any part of X as long as its domain (in the set of filenames) is disjoint from that of Y. This also means that the specification of `cp -r` is not deterministic. Several command specifications are in fact not deterministic, but `cp -r` stays so even with our simplified representation of file systems (as feature trees) and of execution of scripts (no required permissions, no concurrence). This behaviour is even more complex to model than that of the success case of `cp -r`. It is another reason why we use an approximation here.
The other transformation of $failure($`cp` $p_s/f_s\ \ p_d/f_d)$ (T_2 in Figure 3.24b), represents the case where `cp -r` fails without modifying the file system.

When `cp` is used on more than two arguments it behaves similarly to `mv`, with the exception that if there is no directory at path p_d/f_d then the command fails without changing the file system. Otherwise the command: `cp` $p_1/f_1 \dots p_n/f_n\ \ p_d/f_d$ behaves like the *uninterrupted composition* of the commands: `cp` $p_1/f_1\ \ p_d/f_d$; $\dots$ `cp` $p_n/f_n\ \ p_d/f_d$.

About the approximations in `cp -r`

We discuss here the approximation used in the representation of the command
`cp -r` $p_s/f_s\ \ p_d/f_d$, specifically transformation T_8 of *success*(`cp -r` $p_s/f_s\ \ p_d/f_d$) in Figure 3.23 and transformation T_1 of $failure($`cp -r` $p_s/f_s\ \ p_d/f_d)$ in Figure 3.24a. We explain what our choice entails and then we list our alternatives and explain why we decided against them. The argument presented in this discussion are linked to the composition algorithm of section 3.3 and the algorithm to compute inverse images presented in section 4.1, so we advise that you be familiar with these algorithms before reading this discussion.

First we show how the choice we have made, forgetting the link between variables X, Y and Z, forces us to have output contraints in our model of tree pattern transducers. This is because we want to be able to compute compositions of tree pattern transducers. The problem arises when you want to represent the tree pattern transducer representing, for example, the composition of commands: `cp -r /bin /etc; rm /etc/bin/cat`, when there is a non-empty directory at path `/etc/bin/`. In this case command `cp -r /bin /etc` succeeds only with transformation T_8 of *success*(`cp -r` $p_s/f_s\ \ p_d/f_d$) (Figure 3.23). In order to characterize the tree transformation performed by `cp -r /bin /etc; rm /etc/bin/cat`, we still have to express the fact that the output cannot have a file at path `/etc/bin/cat`, which we cannot do with a constraint on the input of the composition `cp -r /bin /etc; rm /etc/bin/cat`. And this is because with our approximation forgets the link between the content of directory `/etc/bin` before and after command `cp -r /bin /etc`.

Without this approximation, all the variables appearing on the right side of rules had to appear on the left side too. This is linked to the fact that commands are somewhat deterministic and their output is always build from the input. This property implied that we could express constraints on the output of rules as constraints on the input. So, because of our choice of approxiamtion, we had to add, in our model of tree pattern transducers, constraints on outputs.

The approximation we should to make may induce our tool to flag as errors some script behaviours which are formally correct. But it is important to remember that this only happens when a scripts attempts to copy a directory into a non-empty directory: not all such occurrences are bugs, but they should not be considered as good practices.

Now we explore the alternatives of this choice and discuss their respective benefits and caveats:

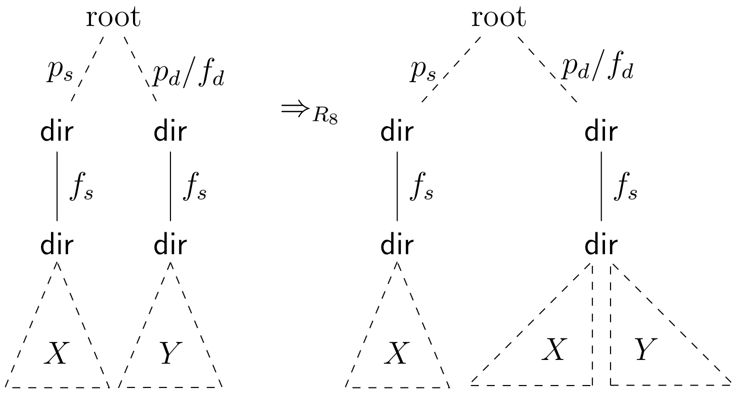


Figure 3.25 – Alternative rule for transformation T_8

1. The first solution is to allow two variables in the same directory. We would get the rule shown in Figure 3.25. This would mean allowing several variables in the same directory, making the model more complex. In particular it induces an exponential blow-up of the time complexity of computing the composition of tree pattern transducers, and of computing the inverse image of a constraint through a rewriting rule (we need to compute inverse images for our implementation, see section 4.1).

   We can see this when on the example of `cp -r /bin /etc; rm /etc/bin/cat`. Since we do not know whether the file at path `/etc/bin/cat` was copied by the `cp -r /bin /etc` command, we have to consider both possibilities. This means that, with each new composition, the number of transformations can increase exponentially.

   This may not seem like a big problem since we already have 88 rules for the success case of `cp -r /bin /etc`. But in section 4.1 we show how to reduce the number of rules. In fact all commands except for `mv` and `cp` can be reduced to one rule for their success case and one rule for their failure case.

   So we reject this alternative because it would incur a big cost to the time complexity of our algorithm.

2. A second alternative would be to keep the rule as it is in Figure 3.23, and add constraints linking the variables X, Y and Z. This approach would yield about the same complexity problem as the previous one. A modification on variable Z would imply a disjunction of cases on X and Y which would worsen the time complexity of computing the composition of tree pattern transducers.

3. A third alternative consists in underapproximating the behaviour of cp by assuming that copying into a non-empty directory always fails. This is better in terms of complexity than what we have chosen to do. But the complexity gain is fairly minor. It does not avoid an exponential blow-up because the success case of `cp -r` still has 4 other transformations. It does not avoid an exponential blow-up when composing with other `mv` or `cp` commands.

   On the other hand, this underapproximation can make some errors invisible. Compared to this underapproximation, the overapproximation we have chosen does not exactly solve this issue, but at least it leaves them visible. It is important to understand that, although we are trying to use formal verification, deciding if a formal error in a script constitutes an actual bug still requires human intervention. For this reason we prefer that our tool flags some type of script behaviour as a potential bug, rather than assuming it is not a bug.

In this section we have defined the transducers associated with most of the usual shell script commands which interact with the file system. We have notably ommited the `find` command. Indeed this command is hard to represent in our model of transducers. But its use in maintainer scripts is rare.

3.3 Composition of tree pattern transducers

Before detailing the procedure used to compute the composition of tree pattern transducers, we illustrate it on a simple example.

3.3.1 Example of composition

We first show how the composition of *tree pattern transducers* is computed on the example script s shown in Figure 3.26. This script succeeds on the script trace
(*success*(`cp -r /bin /tmp`), *success*(`rmdir /tmp/bin/foo`), *success*(`rm /bin/foo`)).

```
    cp -r /bin /tmp
    rmdir /tmp/bin/foo
    rm /bin/foo

```

Figure 3.26 – Example script s

We start by giving a tree pattern transducer for the trace atom *success*(`rm /bin/foo`). There are 6 transformations, one for each non-directory file type
$\mathsf{ft} \in \{\mathsf{reg}, \mathsf{symlink}, \mathsf{fifo}, \mathsf{block}, \mathsf{char}, \mathsf{sock}\}$. For each value of ft the transformation is noted $T_{3,\mathsf{ft}}$ and is shown in Figure 3.27 (the patterns are the same except for the node of type ft. The input constraint ft(`/bin/foo`) indicates that transformation $T_{3,\mathsf{ft}}$ is applied only if the file system has a node of type ft at path `/bin/foo`.

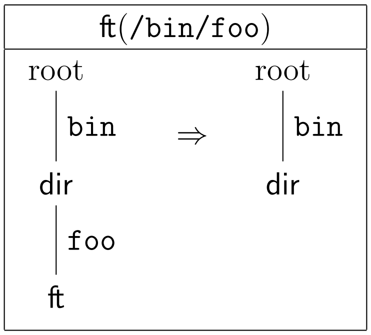


Figure 3.27 – Tree pattern transformation $T_{3,\mathsf{ft}}$ of *success*(`rm /bin/foo`)

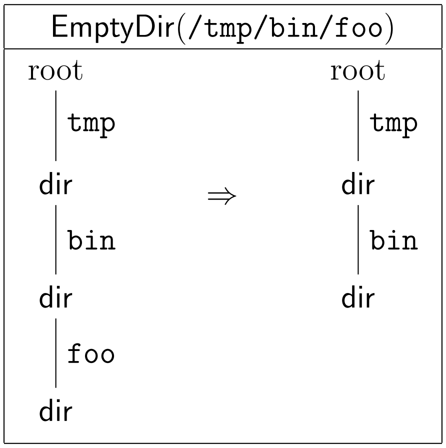


Figure 3.28 – Tree pattern transformation T_2 of *success*(`rmdir /tmp/bin/foo`)

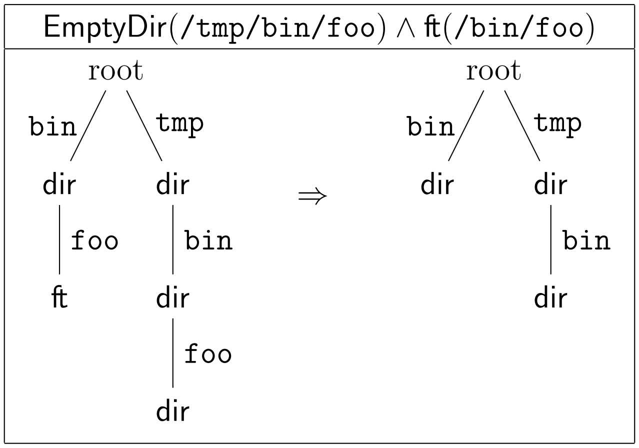


Figure 3.29 – Transformation $T_2T_{3,\mathsf{ft}}$ for ($success$(`rmdir /tmp/bin/foo`), $success$(`rm /bin/foo`))

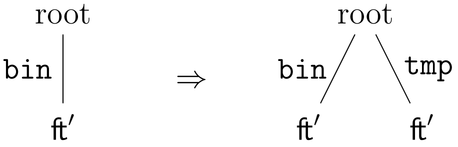


Figure 3.30 – Rule R_1 of $success$(`cp -r /bin /tmp`)

Trace atom $success$(`rmdir /tmp/bin/foo`) yields the transformation T_2 shown in Figure 3.28.

We start by computing the composition of $success$(`rmdir /tmp/bin/foo`) and $success$(`rm /bin/foo`). Since there are 6 transformations for $success$(`rm /bin/foo`), there are 6 compositions to compute (they only differ on filetype ft). Because the composed transformations act on independent parts of the file system, their compositions are straighforward. For each non-directory file type ft ∈ {reg, symlink, fifo, block, char, sock}, the composition of T_2 with $T_{3,\mathsf{ft}}$ is noted $T_2T_{3,\mathsf{ft}}$ and shown in Figure 3.29.

For $success$(`cp -r /bin /tmp`) we have 88 possible transformations, or eight if we consider transformations up to modifications of filetypes. Those are numbered R_1 to R_8 in subsection 3.2.8.
Six of those transformations (considering them up to modifications of filetypes) can be eliminated because their output patterns are incompatible with the tree constraint of $T_2T_{3,\mathsf{ft}}$.

For example, for all non-directory file types ft, ft′ ∈ {reg, symlink, fifo, block, char, sock}, the tree constraint ft(`/bin/foo`) is incompatible with the output pattern of the rule R_1 of $success$(`cp -r /bin /tmp`) represented in Figure 3.30.

Rules R_2, R_3 and R_4 (in subsection 3.2.8) of $success$(`cp -r /bin /tmp`) are also incompatible with tree constraint ft(`/bin/foo`).

Rule R_6 and R_7 are also incompatible with the tree constraint of $T_2T_{3,\mathsf{ft}}$. For example the

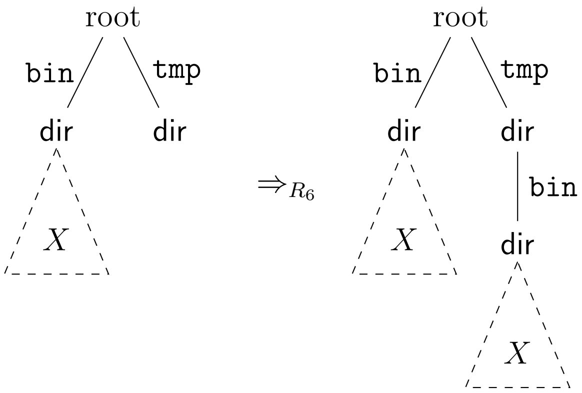


Figure 3.31 – Rule R_6 of *success*(`cp -r /bin /tmp`)

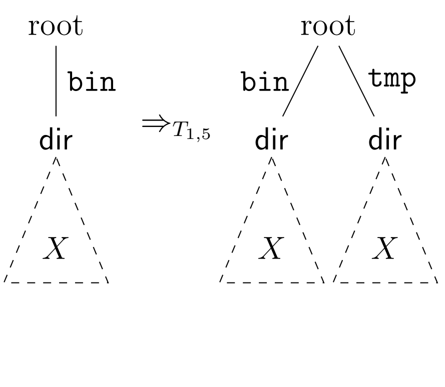
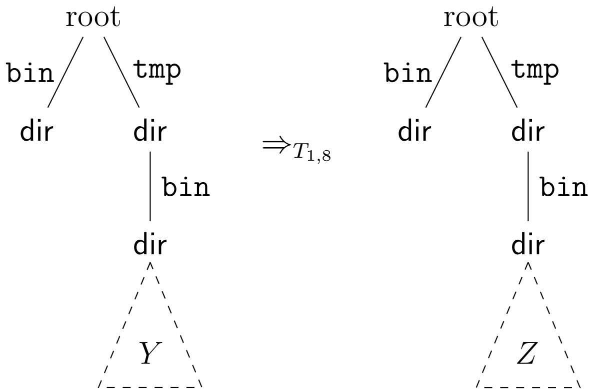
dir(`/bin`) ∧ ¬node(`/tmp`)	dir(`/bin`) ∧ NonEmptyDir(`/tmp/bin`)
	

Figure 3.32 – Transformations $T_{1,5}$ (left) and $T_{1,8}$ (right) of *success*(`cp -r /bin /tmp`)

output pattern of rule R_6 represented in Figure 3.31, combined with the tree constraint EmptyDir(`/tmp/bin/foo`) ∧ ft(`/bin/foo`) of $T_2T_{3,\mathsf{ft}}$, implies that the subtree represented by variable X satisfies the tree constraint EmptyDir(`foo`) ∧ ft(`foo`) which is unsatisfiable because ft $\neq$ dir. Rule R_7 has the same output pattern as R_6, so it entails the same contradiction.

The remaining rules for *success*(`cp -r /bin /tmp`) are rules R_5 and R_8 (from subsection 3.2.8). In the scope of this example we note the corresponding transformations $T_{1,5}$ and $T_{1,8}$, they are represented in Figure 3.32.

We start by composing $T_{1,5}$ with $T_2T_{3,\mathsf{ft}}$, for a fixed parameter ft ∈ {reg, symlink, fifo, block, char, sock}.

First we translate the input constraint EmptyDir(`/tmp/bin/foo`) ∧ ft(`/bin/foo`) of $T_2T_{3,\mathsf{ft}}$, this means:

- applying the input constraint of $T_2T_{3,\mathsf{ft}}$ to the output pattern of $T_{1,5}$ and deducing the constraints satisfied by the variables in the pattern. Here it implies that variable X satisfies the constraint EmptyDir(`bin/foo`) ∧ ft(`foo`),

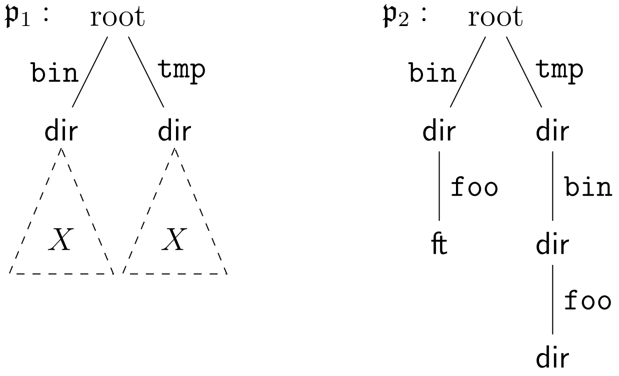


Figure 3.33 – Patterns $\mathfrak{p}_1$ (left) and $\mathfrak{p}_2$ (right)

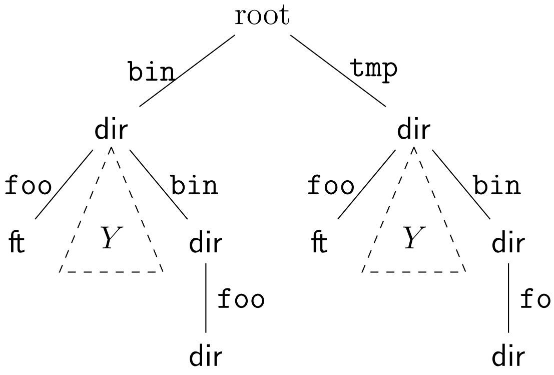


Figure 3.34 – Unification of patterns $\mathfrak{p}_1$ and $\mathfrak{p}_2$

- infering, from the constraint on the variables, the tree constraint satisfied by the input pattern of $T_{1,5}$. Here this means that the input satisfies the constraint EmptyDir(/bin/bin/foo) $\wedge$ ft(/bin/foo).

Then the tree constraint of the composition $T_{1,5}T_2T_{3,\mathsf{ft}}$ is the conjunction of the constraint of $T_{1,5}$ and the translation of the constraint of $T_2T_{3,\mathsf{ft}}$ through the rule of $T_{1,5}$. We get the constraint: dir(/bin) $\wedge \neg$node(/tmp) $\wedge$ EmptyDir(/bin/bin/foo) $\wedge$ ft(/bin/foo). Since dir(/bin) is implied by ft(/bin/foo) we can simplify the constraint, we obtain:

$$C_1 = \neg\mathsf{node}(\texttt{/tmp}) \wedge \mathsf{EmptyDir}(\texttt{/bin/bin/foo}) \wedge \mathsf{ft}(\texttt{/bin/foo})$$

The rewriting rule for $T_{1,5}T_2T_{3,\mathsf{ft}}$ is obtained by first unifying the output pattern of $T_{1,5}$, noted $\mathfrak{p}_1$, with the input pattern of $T_2T_{3,\mathsf{ft}}$, noted $\mathfrak{p}_2$. Patterns $\mathfrak{p}_1$ and $\mathfrak{p}_2$ are represented in Figure 3.33 and their unification is in Figure 3.34.

There the subtree Y is obtained from subtree X by removing file(foo) and dir(bin). We then propagate this substitution of variables to the input pattern of $T_{1,5}$ and the output pattern of $T_2T_{3,\mathsf{ft}}$. We get the patterns $\mathfrak{p}_0$ and $\mathfrak{p}_3$ shown in Figure 3.35. Since we allow ourselves to leave variables implicit only when they are at the same path in the input as in the output pattern, the variable at path /bin/bin was moved so we have to make it

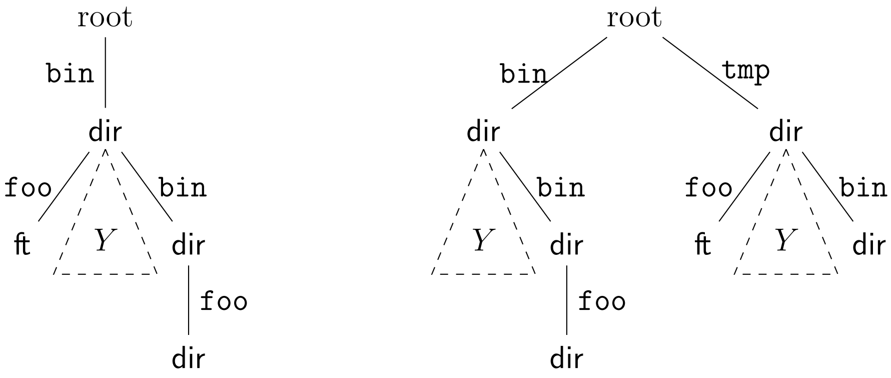


Figure 3.35 – Patterns $\mathfrak{p}_0$ (left) and $\mathfrak{p}_3$ (right)

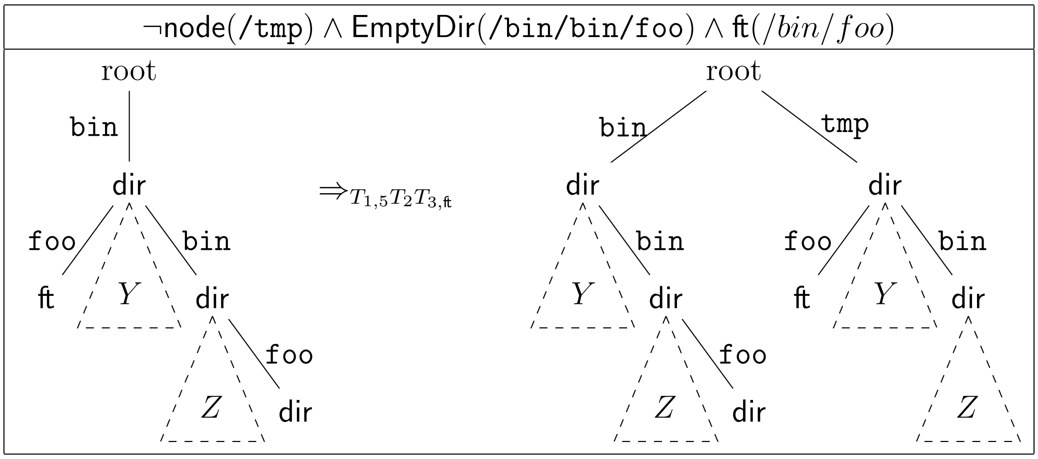


Figure 3.36 – Transformation $T_{1,5}T_2T_{3,\mathsf{ft}}$

explicit. We get the transformation $T_{1,5}T_2T_{3,\mathsf{ft}}$ represented in Figure 3.36. Actually, we have one transformation for each value of ft in $\{\mathsf{reg}, \mathsf{symlink}, \mathsf{fifo}, \mathsf{block}, \mathsf{char}, \mathsf{sock}\}$.

That makes 6 transformations which represent 6 successful executions of the script s shown in Figure 3.26.

In order to get the remaining 6 transformations of the success case of the script we now compose transformation $T_{1,8}$ with $T_2T_{3,\mathsf{ft}}$ for each $\mathsf{ft} \in \{\mathsf{reg}, \mathsf{symlink}, \mathsf{fifo}, \mathsf{block}, \mathsf{char}, \mathsf{sock}\}$. We get the transformation $T_{1,8}T_2T_{3,\mathsf{ft}}$ shown in Figure 3.37 for each ft in $\{\mathsf{reg}, \mathsf{symlink}, \mathsf{fifo}, \mathsf{block}, \mathsf{char}, \mathsf{sock}\}$. We have seen how the composition of tree pattern transducers can work on an example. Next we describe the general procedure for computing the composition of two tree pattern transducers.

3.3.2 Unification of tree patterns

The composition of two tree pattern transducers τ and τ' is the list of *tree pattern transformations* obtained by computing the composition of a *tree pattern transformation* of τ and a *tree pattern tranformation* of τ'. From now on we describe how to compute the

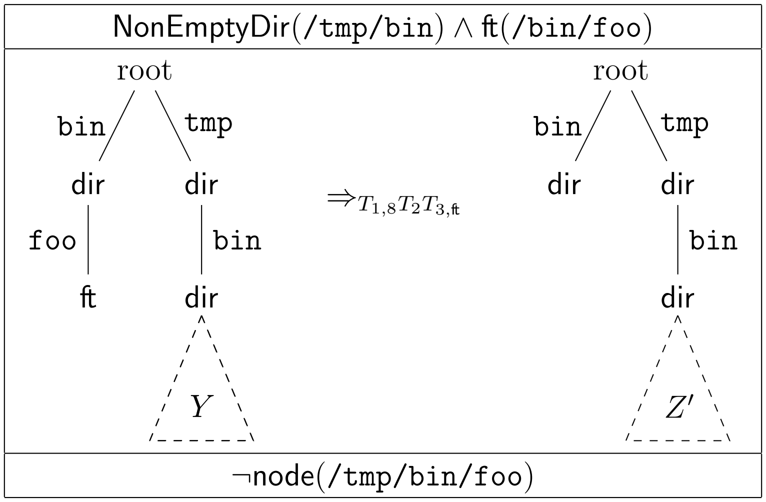


Figure 3.37 – Transformation $T_{1,8}T_2T_{3,\mathsf{ft}}$

composition of two tree pattern transformations T_2 and T_1 noted $T_2 \circ T_1$. Those transformations are of the form: $T_1 = (C_1, (\mathfrak{p}_1, \mathfrak{p}_2), C_2)$ and $T_2 = (C_2', (\mathfrak{p}_2', \mathfrak{p}_3), C_3)$ where C_1, C_2, C_2' and C_3 are tree constraints and $\mathfrak{p}_1, \mathfrak{p}_2, \mathfrak{p}_2'$ and $\mathfrak{p}_3$ are tree patterns.

The first step in computing the composition $T_2 \circ T_1$ consists in unifying the tree patterns $\mathfrak{p}_2$ and $\mathfrak{p}_2'$. In this section we are going to show that any pair of unifyable tree patterns admits a most general unification pattern, we show how to compute the unification and the relation between the variables in tree patterns and their unification. We start by defining those terms:

Definition 11 Two tree patterns $\mathfrak{p}, \mathfrak{p}' \in \mathcal{TP}$ are *unifyable* if there exists a valuation of variables θ such that $\theta(\mathfrak{p}) = \theta(\mathfrak{p}')$.

A set of pairs of tree patterns $S = \{(\mathfrak{p}_i, \mathfrak{p}_i')\}_{i \leq n}$ is unifyable if there exists a valuation of variables θ such that for all $i \leq n$: $\theta(\mathfrak{p}_i) = \theta(\mathfrak{p}_i')$

In order to describe the relationship between variables in patterns and their unification we need a notion of substitution of variables in tree patterns, we call these substitutions *meta-valuations*:

Definition 12 For any two finite sets of variables V_1 and V_2, a *meta-valuation* υ from V_1 to V_2 is a mapping associating with each variable in V_1 a tree pattern with variables in V_2.

Such a meta-valuation υ is *compatible* with a tree pattern $\mathfrak{p}$ if we can replace variables in $\mathfrak{p}$ with their image through υ without collision of filenames, i.e. for each directory in $\mathfrak{p}$ of the form (X, map) with $\upsilon(X) = (Y, \mathsf{map}')$, $X \in V_1$ and $Y \in V_2 \cup \{\bot\}$, the domains of the finite mappings map and map' are disjoint.

If they are compatible then υ associates with $\mathfrak{p}$ the tree pattern obtained from $\mathfrak{p}$ by substituting variables using υ, and substituting $\bot$ with the empty mapping. The resulting tree pattern is noted $\upsilon(\mathfrak{p})$.

A meta-valuation υ from V_1 to V_2 is *compatible* with a meta-valuation υ' from V'_1 to V'_2 if it is compatible with tree patterns $\upsilon'(X)$ for each $X \in V'_1$. In this case we can compose them, the composition is noted $\upsilon \circ \upsilon'$ and is defined by: $\forall X \in V'_1, \upsilon \circ \upsilon'(X) = \upsilon(\upsilon'(X))$.

Now we can express the notion of minimal unification of tree patterns:

Definition 13 For all meta-valuation υ and set $S = \{(\mathfrak{p}_i, \mathfrak{p}'_i)\}_{1 \leq i \leq n}$ of pairs of tree patterns, we say that υ *unifies* S if for all $i \leq n$:

$$\upsilon(\mathfrak{p}_i) = \upsilon(\mathfrak{p}'_i)$$

We say that υ *minimally unifies* S if it unifies S and if, for all meta-valuation υ' unifying S, there is a meta-valuation ω such that $\omega \circ \upsilon = \upsilon'$. We then call υ a *minimal unifier* of S.

For all meta-valuation υ and all set $S = \{(\mathfrak{p}_i, \mathfrak{p}'_i)\}_{1 \leq i \leq n}$ of pairs of tree patterns, we note $\upsilon(S)$ the set $\{(\upsilon(\mathfrak{p}_i), \upsilon(\mathfrak{p}'_i))\}_{1 \leq i \leq n}$. Then we have the following lemma:

Lemma 1 For all meta-valuations υ and υ' and all sets S and S' of pairs of tree patterns, if υ minimally unifies S and υ' minimally unifies $\upsilon(S')$ then $\upsilon' \circ \upsilon$ minimally unifies $S \cup S'$.

Proof
Since υ unifies S and υ' unifies $\upsilon(S')$, $\upsilon' \circ \upsilon$ must unify $S \cup S'$.

Let ω be a meta-valuation unifying $S \cup S'$. Because ω unifies S there exists a meta-valuation ω_0 such that $\omega = \omega_0 \circ \upsilon$. Since $\omega_0 \circ \upsilon$ unifies S', ω_0 must unify $\upsilon(S')$, which implies that there is a meta-valuation ω'_0 such that $\omega_0 = \omega'_0 \circ \upsilon'$. Therefore $\omega = \omega'_0 \circ (\upsilon' \circ \upsilon)$. So $\upsilon' \circ \upsilon$ minimally unifies the set $S \cup S'$. □

Lemma 2 The minimal unifier of a set S of pairs of tree patterns is unique up to renaming of variables.

Proof
We assume that there are two meta-valuations υ and υ' minimally unifying set S of pairs of tree patterns. By definition there must be two meta-valuations ω and ω' such that: $\upsilon = \omega' \circ \upsilon'$ and $\upsilon' = \omega \circ \upsilon$. So $\upsilon = \omega' \circ \omega \circ \upsilon$. We assume that the domain of ω is the set of variables appearing in the image of υ (if it is not then we can take the restriction of ω to that set of variables instead of ω). For all variable $X \in \mathrm{dom}(\omega)$: $\omega' \circ \omega(X) = (X, \emptyset)$. Since $\omega(X)$ is of the form (Z, map), we can deduce here that $\mathsf{map} = \emptyset$ (otherwise $\omega' \circ \omega(X) \neq (X, \emptyset)$). So for all $X \in \mathrm{dom}(\omega)$, $\omega(X)$ is of the form $(Z, \emptyset)$ where Z is a variable, i.e. ω is a renaming of variables. Therefore υ and υ' are equivalent up to renaming of variables. □

Definition 14 A multi-set S of tree patterns is *linear* if each variable appearing in a pattern in S occurs exactly once in S.

A meta-valuation υ is *linear* on a set V of variables if the multi-set $\{\upsilon(X)\}_{X \in V}$ is linear.

In order to prove that the procedure of unification terminates, we define a notion of size on tree patterns and meta-valuations.

Definition 15 The *size* of a tree pattern $\mathfrak{p} = (X, \mathsf{map})$ is inductively defined as $s((X, \mathsf{map})) = 1 + \sum_{f \in \mathrm{dom}(\mathsf{map})} s(\mathsf{map}(f))$ and the *size* of a node $\mathcal{N}$ is $s(\mathfrak{ft}) = 1$ if $\mathcal{N} = \mathfrak{ft} \in \texttt{types} \setminus \{\mathsf{dir}\}$ or $s((\mathsf{dir}, \mathfrak{p})) = s(\mathfrak{p})$ if $\mathcal{N} \in \{\mathsf{dir}\} \times \mathcal{TP}$.

The size of a meta-valuation υ on a set of variables V is $s(\upsilon) = \sum_{X \in V} s(\upsilon(X))$.

We can now describe the procedure to unify unifyable pairs of tree patterns:

Lemma 3 For all unifyable set S of pairs of tree patterns we can compute a minimal unifier υ of S.

Noting V_1 the set of variables appearing in patterns on the left side of pairs of S, if the multi-set of patterns on the right side of pairs in S is linear then υ is linear on V_1.

Proof
We recall that the size of a tree pattern $\mathfrak{p} = (X, \mathsf{map})$ is inductively defined as $s((X, \mathsf{map})) = 1 + \sum_{f \in \mathrm{dom}(\mathsf{map})} s(\mathsf{map}(f))$ and the *size* of a node $\mathcal{N}$ is $s(\mathfrak{ft}) = 1$ if $\mathcal{N} = \mathfrak{ft} \in \texttt{types} \setminus \{\mathsf{dir}\}$ or $s((\mathsf{dir}, \mathfrak{p})) = s(\mathfrak{p})$ if $\mathcal{N} \in \{\mathsf{dir}\} \times \mathcal{TP}$. With each unifyable set S of pairs of patterns we associate its complexity, noted $C(S)$, defined by $C(S) = (n_{var}(S), s(S))$ where $n_{var}(S)$ is the number of distinct variables appearing in patterns in S and $s(S)$ is the sum of the sizes of the patterns in S. We prove the lemma by induction on the complexity $C(S)$ of S, using the lexicographic order on pairs of integers.

If S is the empty set then the empty meta-valuation minimally unifies S.

Let S a unifyable non-empty set of pairs of tree patterns, with $n \geq 1$. Let $(\mathfrak{p}, \mathfrak{p}') \in S$ a pair of tree patterns of the form: $\mathfrak{p} = (X_1, \mathsf{map}_1)$ and $\mathfrak{p}' = (X_2, \mathsf{map}_2)$ where map_1 and map_2 are finite mappings and X_1 and X_2 are either variables or the constant $\bot$. Since S is unifyable $(\mathfrak{p}, \mathfrak{p}')$ is also unifyable. Now we have different cases depending on X_1 and X_2:

Case 1: $X_1 = X_2 = \bot$

In this case the domains of map_1 and map_2 are identical. Indeed, if there were a feature $f \in \mathrm{dom}(\mathsf{map}_1) \setminus \mathrm{dom}(\mathsf{map}_2)$, then, for all valuation θ compatible with $\mathfrak{p}$ and $\mathfrak{p}'$, $\theta(\mathfrak{p})$ would have a node at path f but not $\theta(\mathfrak{p}')$, and so $(\mathfrak{p}, \mathfrak{p}')$ would not be unifyable.

We define the set of pairs of patterns: $S' = \{(\mathsf{map}_1(f), \mathsf{map}_2(f))\}_{f \in \mathrm{dom}(\mathsf{map}_1)} \cup S \setminus \{(\mathfrak{p}, \mathfrak{p}')\}$. So S' and S have the same unifiers, and therefore they have the same minimal unifiers. Furthermore S' and S have the same variables and $s(S) = 1 + s(S')$, so $C(S') < C(S)$. Therefore we can use the induction hypothesis on S' to compute a minimal unifier of S.

Case 2: $X_1 = X_2 \neq \bot$

The domains of map_1 and map_2 are identical (otherwise $(\mathfrak{p}, \mathfrak{p}')$ would not be unifyable).

We define the set of pairs of patterns:
$S' = \{(\mathsf{map}_1(f), \mathsf{map}_2(f))\}_{f \in \mathrm{dom}(\mathsf{map}_1)} \cup S \setminus \{(\mathfrak{p}, \mathfrak{p}')\}$. So S' and S have the same unifiers, and therefore they have the same minimal unifiers. Furthermore S' has at most as many distinct variables as S and $s(S) = 1 + s(S')$, so $C(S') < C(S)$. Therefore we can use the induction hypothesis on S' to compute a minimal unifier of S.

Case 3: $X_1 = \bot$ **and** $X_2 \neq \bot$

The domain of map_2 is included in the domain of map_1 (otherwise $(\mathfrak{p}, \mathfrak{p}')$ would not be unifyable).

We note map'_1 the restriction of the mapping map_1 to the set $\mathrm{dom}(\mathsf{map}_1) \setminus \mathrm{dom}(\mathsf{map}_2)$. So the sets $\{(\mathfrak{p}, \mathfrak{p}')\}$ and $\{((\bot, \mathsf{map}'_1), (X_2, \emptyset))\} \cup \{(\mathsf{map}_1(f), \mathsf{map}_2(f))\}_{f \in \mathrm{dom}(\mathsf{map}_2)}$ have the same unifiers. We define the sets $S_{X_2} = \{((\bot, \mathsf{map}'_1), (X_2, \emptyset))\}$ and
$S' = \{(\mathsf{map}_1(f), \mathsf{map}_2(f))\}_{f \in \mathrm{dom}(\mathsf{map}_2)} \cup S \setminus \{(\mathfrak{p}, \mathfrak{p}')\}$. Then S and $S_{X_2} \cup S'$ have the same unifiers, and therefore they have the same minimal unifiers.

Noting V_2 the set of variables appearing in S_{X_2}, we define the meta-valuation υ so that $\upsilon(X_2) = (\bot, \mathsf{map}'_1)$ and υ is the identity on $V_2 \setminus \{X_2\}$. So υ unifies $S_{X_2} = \{((\bot, \mathsf{map}'_1), (X_2, \emptyset))\}$. We now show that υ minimally unifies S_{X_2}. Let ω be a unifier of S_{X_2}. The variable X_2 can not appear in map'_1 because S_{X_2} is unifyable. So we define the the meta-valuation ω' for all variable $X \in V_2 \setminus \{X_2\}$ by: $\omega'(X) = \omega(X)$. We get:

$$\omega(X_2) = \omega((\bot, \mathsf{map}'_1)) = \omega'((\bot, \mathsf{map}'_1)) = \omega'(\upsilon(X_2))$$

Furthermore, for all variable $X \in V_2 \setminus \{X_2\}$: $\omega(X) = \omega'(X) = \omega'(\upsilon(X))$. Thus $\omega = \omega' \circ \upsilon$. Therefore υ minimally unifies S_{X_2}.

Since $S_{X_2} \cup S'$ is unifyable, there exists a valuation of variables θ unifying $S_{X_2} \cup S'$. Because θ unifies S_{X_2} there is a valuation θ_0 such that $\theta = \theta_0 \circ \upsilon$. Since θ unifies S', θ_0 unifies $\upsilon(S')$, and so $\upsilon(S')$ is unifyable. All variables appearing in $\upsilon(S')$ also appear in S, furthermore X_2 appears in S but not in $\upsilon(S')$. Therefore $\upsilon(S')$ has strictly less variables than S and so we can use the induction hypothesis on $\upsilon(S')$ to compute a minimal unifier υ' of $\upsilon(S')$.

Lemma 1 allows us to conclude that $\upsilon' \circ \upsilon$ minimally unifies $S_{X_2} \cup S'$, so it also minimally unifies S.

If the multi-set of patterns appearing on the right side of pairs in S is linear, then it is also true of $\upsilon(S')$, thus υ' is linear on the set V_1 of the variables appearing on the left side of pairs of $\upsilon(S')$. In this case variable X_2 appears once in $\mathfrak{p}'$ and therefore does not appear in S'. Since υ is linear on $V_2 \setminus \{X_2\}$ and $S_{X_2} \cup S'$ have the same variables except for X_2: $\upsilon' \circ \upsilon$ is linear on the set V_1 of variables appearing on the left side of pairs of $S_{X_2} \cup S'$.

Case 4: X_1 **and** X_2 **are different variables**

In this case we note map'_1 the restriction of map_1 to the set $\mathrm{dom}(\mathsf{map}_1) \setminus \mathrm{dom}(\mathsf{map}_2)$ and we note map'_2 the restriction of map_2 to the set $\mathrm{dom}(\mathsf{map}_2) \setminus \mathrm{dom}(\mathsf{map}_1)$.

We now show that X_1 does not appear in map'_2. In order to do this we assume that X_1 appears in map'_2 and prove a contradiction. Let θ a valuation unifying S. Then

$\theta((X_1, \mathsf{map}_1)) = \theta((X_2, \mathsf{map}_2))$. Since X_1 appears in map'_2 there exists a feature $f \in \mathrm{dom}(\mathsf{map}_2) \setminus \mathrm{dom}(\mathsf{map}_1)$ such that X_1 appears in $\mathsf{map}'_2(f)$. We note $t_1 = \theta(X_1)$. Since $f \notin \mathrm{dom}(\mathsf{map}_1)$: $t_1(f) = \theta(\mathsf{map}'_2(f))$. Yet $\mathsf{map}'_2(f)$ contains an occurrence of X_1, so $t_1(f) = \theta(\mathsf{map}'_2(f))$ contains an occurrence of t_1, which is a contradiction because t_1 is a finite tree.

Similarly we can show that X_2 does not appear in map'_1, and that either X_1 does not appear in map'_1 or X_2 does not appear in map'_2. From now on we assume that X_2 does not appear in map'_2, the other case (where X_1 does not appear in map'_1) is similar.

Noting $Z \notin V$ a fresh variable, we define the meta-valuation υ_2 so that $\upsilon_2(X_2) = (Z, \mathsf{map}'_1)$ and υ_2 is the identity on $V \setminus \{X_2\}$. We also define υ_1 so that $\upsilon_1(X_1) = (Z, \mathsf{map}'_2)$ and υ_1 is the identity on $\{Z\} \cup V \setminus \{X_1\}$. Now we show that any meta-valuation ω unifying S is of the form $\omega = \omega_0 \circ \upsilon_1 \circ \upsilon_2$ where ω_0 is a meta-valuation.

Since ω unifies S, there must be a variable Z' such that $\omega(X_1) = (Z', \mathsf{map}_{X_1})$ and $\omega(X_2) = (Z', \mathsf{map}_{X_2})$ with:

$$\mathsf{map}_{X_1} \cup \omega(\mathsf{map}_1) = \mathsf{map}_{X_2} \cup \omega(\mathsf{map}_2)$$

We introduce the sets of features: $F_{1,2} = \mathrm{dom}(\mathsf{map}_1) \cap \mathrm{dom}(\mathsf{map}_2)$, $F_1 = \mathrm{dom}(\mathsf{map}_1) \setminus \mathrm{dom}(\mathsf{map}_2)$, $F_2 = \mathrm{dom}(\mathsf{map}_2) \setminus \mathrm{dom}(\mathsf{map}_1)$ and $F_0 = \mathcal{F} \setminus (F_1 \cup F_2 \cup F_{1,2})$. Then we can transform the equation above by taking the image of a feature f. Depending on f we get:

- if $f \in F_{1,2}$ then $\omega(\mathsf{map}_1(f)) = \omega(\mathsf{map}_2(f))$
- if $f \in F_1$ then $\mathsf{map}_{X_2}(f) = \omega(\mathsf{map}_1(f)) = \omega(\mathsf{map}'_1(f))$
- if $f \in F_2$ then $\mathsf{map}_{X_1}(f) = \omega(\mathsf{map}_2(f)) = \omega(\mathsf{map}'_2(f))$
- if $f \in F_0$ then either $\mathsf{map}_{X_1}(f) = \mathsf{map}_{X_2}(f)$ or f is in the domain of neither map_{X_1} nor map_{X_2}.

So, noting map_0 the restriction of map_{X_1} to set F_0, we have $\mathsf{map}_{X_1} = \mathsf{map}_0 \cup \omega(\mathsf{map}'_2)$ and $\mathsf{map}_{X_2} = \mathsf{map}_0 \cup \omega(\mathsf{map}'_1)$. Then we define the meta-valuation ω_0 on the set of variables $\{Z\} \cup V \setminus \{X_1, X_2\}$ by $\omega_0(Z) = (Z', \mathsf{map}_0)$ and, for all $X \in V \setminus \{X_1, X_2\}$: $\omega_0(X) = \omega(X)$. Therefore:

$$\begin{aligned}
\omega_0 \circ \upsilon_1 \circ \upsilon_2(X_1) &= \omega_0((Z, \mathsf{map}'_2)) \\
&= (Z', \mathsf{map}_0 \cup \omega_0(\mathsf{map}'_2)) \\
&= (Z', \mathsf{map}_0 \cup \omega(\mathsf{map}'_2)) \\
&= \omega(X_1)
\end{aligned}$$

This works because we assumed that neither X_1 nor X_2 appeared in map'_2. For all $X \in V \setminus \{X_1, X_2\}$ we also have $\omega_0 \circ \upsilon_1 \circ \upsilon_2(X) = \omega_0(X) = \omega(X)$. Then for all $X \in V \setminus \{X_2\}$: $\omega_0 \circ \upsilon_1(X) = \omega_0 \circ \upsilon_1 \circ \upsilon_2(X) = \omega(X)$. In particular it implies that $\omega_0 \circ \upsilon_1(\mathsf{map}'_1) = \omega(\mathsf{map}'_1)$. With this we can finally show that:

$$\begin{aligned}
\omega_0 \circ \upsilon_1 \circ \upsilon_2(X_2) &= \omega_0(\upsilon_1((Z, \mathsf{map}'_1))) \\
&= (Z', \mathsf{map}_0 \cup \omega(\mathsf{map}'_1)) \\
&= \omega(X_2)
\end{aligned}$$

So $\omega_0 \circ \upsilon_1 \circ \upsilon_2 = \omega$. With this we have shown that any meta-valuation ω unifying S is of the form $\omega = \omega_0 \circ \upsilon_1 \circ \upsilon_2$ where ω_0 is a meta-valuation.

We define $S' = \{(\mathsf{map}_1(f), \mathsf{map}_2(f))\}_{f \in F_{1,2}} \cup S \setminus \{(\mathfrak{p}, \mathfrak{p}')\}$. Now we use the induction hypothesis on the set $\upsilon_1 \circ \upsilon_2(S')$. We can do so because the set of variables in $\upsilon_1 \circ \upsilon_2(S')$ is included in the set $\{Z\} \cup V \setminus \{X_1, X_2\}$, and therefore the number of variables is less than that of S (set V of variables). We get a minimal unifier υ of $\upsilon_1 \circ \upsilon_2(S')$. Therefore $\upsilon \circ \upsilon_1 \circ \upsilon_2$ unifies S'. Let ω a unifier of S. Then there is a meta-valuation ω_0 such that $\omega = \omega_0 \circ \upsilon_1 \circ \upsilon_2$. Thus ω_0 unifies $\upsilon_1 \circ \upsilon_2(S)$ and $\upsilon_1 \circ \upsilon_2(S')$. Then, by minimality of υ, there must be ω_0' such that $\omega_0 = \omega_0'$. Therefore $\omega = \omega_0' \circ \upsilon \circ \upsilon_1 \circ \upsilon_2$. So $\upsilon \circ \upsilon_1 \circ \upsilon_2$ minimally unifies S.

If the multi-set of patterns appearing on the right side of pairs in S is linear, then S' contains no occurrence of X_2 and υ_2 changes nothing in S'. Also X_1 appears only in patterns on the left side of pairs in $\upsilon_2(S')$, so the multi-set of patterns appearing on the right side of pairs in $\upsilon_1 \circ \upsilon_2(S')$ is linear. Then υ is linear on the set V_1 of the variables appearing on the left side of pairs of $\upsilon_1 \circ \upsilon_2(S')$. Therefore $\upsilon \circ \upsilon_1 \circ \upsilon_2$ is linear on $\{X_1\} \cup V_1 \setminus \{Z\}$, which is the set of variables appearing on the left side of pairs of S.

We have proven by induction that for all unifyable set S of pairs of tree patterns we can compute a minimal unifier υ of S such that, noting V_1 the set of variables appearing in patterns on the left side of pairs of S, if the multi-set of patterns on the right side of pairs in S is linear then υ is linear on V_1. □

Corollary 1 For all unifyable pair of tree patterns $(\mathfrak{p}, \mathfrak{p}')$ we can compute a minimal unifier υ and a minimal unification $\mathsf{unify}(\mathfrak{p}, \mathfrak{p}')$ such that:

$$\mathsf{unify}(\mathfrak{p}, \mathfrak{p}') = \upsilon(\mathfrak{p}) = \upsilon(\mathfrak{p}')$$

In the composition of $T_1 = (C_1, (\mathfrak{p}_1, \mathfrak{p}_2), C_2)$ and $T_2 = (C_2', (\mathfrak{p}_2', \mathfrak{p}_3), C_3)$ we can use this corollary to compute the unification of $\mathfrak{p}_2$ and $\mathfrak{p}_2'$. We get a meta-valuation υ and a tree pattern $\mathfrak{q}_2 = \mathsf{unify}(\mathfrak{p}_2, \mathfrak{p}_2')$ such that:

$$\mathfrak{q}_2 = \upsilon(\mathfrak{p}_2) = \upsilon(\mathfrak{p}_2')$$

Then the rewriting rule $R = (\mathfrak{q}_1, \mathfrak{q}_3)$ with $\mathfrak{q}_1 = \upsilon(\mathfrak{p}_1)$ and $\mathfrak{q}_3 = \upsilon(\mathfrak{p}_3)$ is the composition of the rules $(\mathfrak{p}_1, \mathfrak{p}_2)$ and $(\mathfrak{p}_2', \mathfrak{p}_3)$. Indeed, using corollary 1 we can prove that for all feature trees $t_1, t_3 \in \mathcal{T}$:

$$t_1 \to_R t_3 \quad \Leftrightarrow \quad \exists t_2 \in \mathcal{T},\ t_1 \to_{R_1} t_2 \to_{R_2} t_3$$

We do not prove this now, but we prove a stronger claim later in lemma 7.

Complexity analysis

We now discuss the complexity of the procedure computing the composition of rewriting rules. It mainly comes from the complexity of computing the unification of patterns as described in the proof of lemma 3. The recursion's termination is ensured by the fact that at each step either the number n of variables in $\mathfrak{p}$ and $\mathfrak{p}'$ decreases, or n stays the same while the size s of $\mathfrak{p}$ and $\mathfrak{p}'$ decreases. So in order to get the complexity we need a bound on how much s can increase on the steps where n decreases: it happens when some variables are replaced with a subtree. We can bound the growth of s by noticing that the sizes of the subtrees with which are replaced variables are inferior to s. Since a variable occurs at most s times, we can say that, on the steps where n decreases, s increases at most to s^2. So the size of the unification pattern during the algorithm grows at most to s^{2^n} where s is the sum of the sizes of the patterns to unify and n is their total number of variables. Each step of the recursion has a time complexity linear in the *current* size of the patterns to unify. So, in all generality, the worst-case time complexity is $s^{2^{n+1}}$ (i.e. double exponential), and the worst-case space complexity is also double-exponential.

But this worst-case scenario is reached only when variables appear in multiple different places in patterns, otherwise the size of patterns to unify does not grow much. Furthermore we can observe that, among the patterns appearing in rewriting rules which correspond to actual script commands (c.f. section 3.2), most variables appear once in each pattern, the only exceptions are rules R_5, R_6 and R_7 of command `cp` with option `-r` where a variable appears once in the input pattern but twice in the output pattern. In fact we can prove that the composition of two rewriting rules whose input patterns are linear (c.f. definition 14) has a linear input pattern, it is a direct consequence of the linearity condition in lemma 3. Hence, assuming that we only compute compositions of rewriting rules which correspond to script commands, the input patterns will always be linear, and so we always use the unification procedure on a pair of patterns $(\mathfrak{p}, \mathfrak{p}')$ where $\mathfrak{p}'$ is linear. In this particular case, the size of the patterns on the right side of pairs always decreases during the recursion. Conversely, in patterns on the left side of pairs, we can only substitute variables with subtrees smaller than the patterns on the right side. So the complexity of unifying a pattern of size s_1 with a linear pattern of size s_2 is in space $O(s_1 * s_2^{s_1+s_2})$ (because the number of variables is bound by $s_1 + s_2$) and in time $O((s_1 * s_2^{s_1+s_2})^2)$ (so space and time complexity are both exponential).

We can improve this again by computing composition of commands in the right order. Given a sequence of script commands, we can start computing their composition starting with the composition of the last two commands, then the composition of the last three commands and so on until we get the composition of the full sequence. This way the pattern which we have to unify with a linear pattern necessarily is the output pattern of a rule from a script command, so it has at most one variable X_1 occurring several times, and it only appears twice in the pattern. This greatly improves the complexity of the unification algorithm because after variable X_1 is substituted in the recursion, the patterns to unify become linear. Furthermore, at the step of unification where variable X_1 is substituted, the size increase on the pattern on the left comes from a subtree removed from the pattern on the right, and since the X_1 occurs at only one other place in the patterns to unify the total size does not increase. So the size of the patterns to unify only decreases during the recursion. In this case the space complexity of computing one unification of patterns is therefore linear in the size of the patterns to unify, and time complexity is quadratic. The size of the computed minimal unifier is then at most the sum of the sizes of the patterns to unify.

We have computed the composition of the rewriting rules, next we compute the input and output tree constraints of the composition.

3.3.3 Tree constraints in composition

We have computed the input and output tree patterns of the composition of two tree pattern transformations and we get:

$$t_1 \to_R t_3 \quad \Leftrightarrow \quad \exists t_2 \in \mathcal{T},\ t_1 \to_{R_1} t_2 \to_{R_2} t_3$$

The last step in computing the composition consists in translating tree constraints C_2 and C_2', which are on the midway tree t_2, into constraints on either the input tree t_1 or the output tree t_3. For this we introduce a notion of constraints on sets of variables:

Definition 16 For all finite set V of variables, a *valuation constraint* on V is a function associating with each variable in V a tree constraint.

The semantics of a valuation constraint $\mathcal{C}$ on V, noted $[\![\mathcal{C}]\!]$ is:

$$[\![\mathcal{C}]\!] = \{\theta \mid \theta \text{ is a valuation over } V \text{ and: } \forall X \in V, \theta(X) \in [\![\mathcal{C}(X)]\!]\}$$

Lemma 4 For all pattern $\mathfrak{p}$ with set of variables V, there exists a valuation constraint $\mathcal{C}_\mathfrak{p}$ on V whose semantics is the set of valuations over V which are *compatible* (c.f. definition 7) with $\mathfrak{p}$.

Proof
By definition, a valuation θ over V is incompatible with the tree pattern $\mathfrak{p}$ if and only if a path in θ collides with an existing path in $\mathfrak{p}$, i.e. there is a variable $X \in V$ and a feature $f \in \mathcal{F}$ such that the directory $\theta(X)$ has a node at path f and there exists a path p such that:

- X appears at path p in the pattern $\mathfrak{p}$,
- there is a node at path p/f in pattern $\mathfrak{p}$.

We note $F_{\mathfrak{p},X}$ the set of all the features f for which there is a path p such that, in pattern $\mathfrak{p}$, there is a node at path p/f and variable X is at path p. Then the formula $C_{\mathfrak{p},X}$ defined by: $C_{\mathfrak{p},X} = \bigwedge_{f \in F_{\mathfrak{p},X}} \neg\mathsf{node}(f)$ characterizes the values of X which avoid collisions with paths in $\mathfrak{p}$. So the valuation constraint $\mathcal{C}_\mathfrak{p} : X \to C_{\mathfrak{p},X}$ characterizes the valuations which are compatible with pattern $\mathfrak{p}$. □

Lemma 5 For all pattern $\mathfrak{p}$ with set of variables V and for all finite conjunction C of *constraint literals* (c.f. definition 6), there is a valuation constraint noted $\mathfrak{p}^{-1}(C)$ such that, for all valuation θ over V:

$$\theta \in [\![\mathfrak{p}^{-1}(C)]\!] \quad \Leftrightarrow \quad \theta(\mathfrak{p}) \in [\![C]\!]$$

Proof

We prove this by induction on the number of literals in C.

According to lemma 4, for all valuation θ over V: $\theta \in [\![\mathcal{C}_{\mathfrak{p}}]\!] \Leftrightarrow \theta(\mathfrak{p}) \in [\![\top]\!]$. Therefore $\mathfrak{p}^{-1}(\top) = \mathcal{C}_{\mathfrak{p}}$ validates the initiation of the induction.

We assume that some conjunction C of constraint literals satisfies the property, and try to show that, for some literal ℓ, the conjunction $C \wedge \ell$ also satisfies that property. So, for all valuation θ over V: $\theta \in [\![\mathfrak{p}^{-1}(C)]\!] \Leftrightarrow \theta(\mathfrak{p}) \in [\![C]\!]$

We have different cases depending on ℓ and $\mathfrak{p}$:

1. if $\ell = \mathsf{ft}(p)$ and pattern $\mathfrak{p}$ has a node of type ft at path p, then for all valuation θ over V: $\theta(\mathfrak{p}) \in [\![\ell]\!]$. So we define $\mathfrak{p}^{-1}(C \wedge \ell) = \mathfrak{p}^{-1}(C)$ and, for all valuation θ over V:
$$\theta(\mathfrak{p}) \in [\![C \wedge \ell]\!] \quad \Leftrightarrow \quad \theta(\mathfrak{p}) \in [\![C]\!] \quad \Leftrightarrow \quad \theta \in [\![\mathfrak{p}^{-1}(C \wedge \ell)]\!]$$

2. if $\ell = \mathsf{ft}(p)$ and pattern $\mathfrak{p}$ has a node of type $\mathsf{ft}' \neq \mathsf{ft}$ at path p, then for all valuation θ over V: $\theta(\mathfrak{p}) \notin [\![\ell]\!]$. So we define $\mathfrak{p}^{-1}(C \wedge \ell) = \bot$ and, for all valuation θ over V:
$$\theta(\mathfrak{p}) \in [\![C \wedge \ell]\!] \quad \Leftrightarrow \quad \theta \in [\![\mathfrak{p}^{-1}(C \wedge \ell)]\!]$$

3. if $\ell = \mathsf{ft}(p)$ and the longest prefix of p leading to a node in pattern $\mathfrak{p}$ is not p. In this case we note p' this prefix and we have two subcases depending on the node at path p' in pattern $\mathfrak{p}$:

   - if the node is not of the form $(\mathsf{dir}, (X, \mathsf{map}))$ for some variable X then $\theta(\mathfrak{p}) \notin [\![\ell]\!]$ for all valuation θ. This case is the same as case 2.
   - if the node at path p' is of the form $(\mathsf{dir}, (X, \mathsf{map}))$ for some variable $X \in V$ then, noting p_X the path such that $p = p'/p_X$, for all valuation θ over V:
   $$\theta(\mathfrak{p}) \in [\![\mathsf{ft}(p)]\!] \Leftrightarrow \theta(X) \in [\![\mathsf{ft}(p_X)]\!]$$
   Therefore we define $\mathfrak{p}^{-1}(C \wedge \mathsf{ft}(p))$ by: $\mathfrak{p}^{-1}(C \wedge \mathsf{ft}(p))(X) = \mathfrak{p}^{-1}(C)(X) \wedge \mathsf{ft}(p_X)$ and, on $V \setminus \{X\}$: $\mathfrak{p}^{-1}(C \wedge \mathsf{ft}(p)) = \mathfrak{p}^{-1}(C)$. So, for all valuation θ over V:
   $$\theta(\mathfrak{p}) \in [\![C \wedge \mathsf{ft}(p)]\!] \quad \Leftrightarrow \quad \theta \in [\![\mathfrak{p}^{-1}(C \wedge \mathsf{ft}(p))]\!]$$

4. if $\ell = \exists(p, E)$ and there is a feature $f \in \mathcal{F}$ such that pattern $\mathfrak{p}$ has a node at path p/f, then for all valuation θ over V: $\theta(\mathfrak{p}) \in [\![\exists(p, E)]\!]$. This case is the same as case 1.

5. if $\ell = \exists(p, E)$ and there is no feature $f \in \mathcal{F}$ such that pattern $\mathfrak{p}$ has a node at path p/f, then we note p' the longest prefix of p such that pattern $\mathfrak{p}$ has a node at path p'. We have two subcases depending on the node at path p' in pattern $\mathfrak{p}$:

   - if the node is not of the form $(\mathsf{dir}, (X, \mathsf{map}))$ for some variable $X \in V$, then $\theta(\mathfrak{p}) \notin [\![\ell]\!]$ for all valuation θ and this case is the same as case 2.

- if the node is of the form $(\mathsf{dir}, (X, \mathsf{map}))$ for some variable $X \in V$, then, noting p_X the path such that $p = p'/p_X$, for all valuation θ over V we have:

  $$\theta(\mathfrak{p}) \in [\![\exists(p, E)]\!] \Leftrightarrow \theta(X) \in [\![\exists(p_X, E)]\!]$$

  Therefore we define $\mathfrak{p}^{-1}(C \wedge \exists(p, E))$ by: $\mathfrak{p}^{-1}(C \wedge \exists(p, E))(X) = \mathfrak{p}^{-1}(C)(X) \wedge \exists(p_X, E)$ and, on $V \setminus \{X\}$: $\mathfrak{p}^{-1}(C \wedge \exists(p, E)) = \mathfrak{p}^{-1}(C)$. So, for all valuation θ over V:

  $$\theta(\mathfrak{p}) \in [\![C \wedge \exists(p, E)]\!] \quad \Leftrightarrow \quad \theta \in [\![\mathfrak{p}^{-1}(C \wedge \exists(p, E))]\!]$$

This ends the proof that for all finite conjunction C of constraint literals there is a valuation constraint noted $\mathfrak{p}^{-1}(C)$ such that, for all valuation θ over V:

$$\theta \in [\![\mathfrak{p}^{-1}(C)]\!] \quad \Leftrightarrow \quad \theta(\mathfrak{p}) \in [\![C]\!]$$

□

The time complexity of computing $\mathfrak{p}^{-1}(C)$ is quasiquadratic in the sizes of C and $\mathfrak{p}$, assuming access time to a node in pattern $\mathfrak{p}$ is linear in the length of the path to the node. In most cases the algorithm is in linear time, but testing case 4. requires to compute the intersection of a possibly cofinite set E with the set of outgoing features of the node at path p in the pattern, which can be done in quasilinear time in the sizes of C and $\mathfrak{p}$. The size of $\mathfrak{p}^{-1}(C)$ is linear in the sizes of C and $\mathfrak{p}$.

Lemma 6 For all tree pattern $\mathfrak{p}$ and for all valuation constraint $\mathcal{C}$ over the set V of variables in $\mathfrak{p}$, there is a tree constraint $\mathfrak{p}(\mathcal{C})$ such that, for all valuation θ compatible with $\mathfrak{p}$:

$$\theta \in [\![\mathcal{C}]\!] \quad \Leftrightarrow \quad \theta(\mathfrak{p}) \in [\![\mathfrak{p}(\mathcal{C})]\!]$$

Proof
For each variable $X \in V$ we build a tree constraint C_X such that, for all valuation $\theta \in [\![\mathcal{C}_\mathfrak{p}]\!]$:

$$\theta(X) \in [\![\mathcal{C}(X)]\!] \quad \Leftrightarrow \quad \theta(\mathfrak{p}) \in [\![C_X]\!]$$

For each $X \in V$ we build C_X and prove this property by induction over the structure of $\mathcal{C}(X)$. By hypothesis X occurs at least once in pattern $\mathfrak{p}$, we note p_X one path in $\mathfrak{p}$ where X appears.

If $\mathcal{C}(X) = \mathsf{ft}(p)$ then we define $C_X = \mathsf{ft}(p_X/p)$ which leads to, for all $\theta \in [\![\mathcal{C}_\mathfrak{p}]\!]$:

$$\theta(X) \in [\![\mathsf{ft}(p)]\!] \ \Leftrightarrow \ \theta(\mathfrak{p}) \in [\![C_X]\!]$$

If $\mathcal{C}(X) = \exists(p, E)$ then we define $C_X = \exists(p_X/p, E)$ which leads to, for all $\theta \in [\![\mathcal{C}_\mathfrak{p}]\!]$:

$$\theta(X) \in [\![\exists(p, E)]\!] \ \Leftrightarrow \ \theta(\mathfrak{p}) \in [\![C_X]\!]$$

Assuming that we have two valuation constraints $\mathcal{C}_1$ and $\mathcal{C}_2$ and two tree constraints $C_{1,X}$ and $C_{2,X}$ such that, for all valuation $\theta \in [\![\mathcal{C}_\mathfrak{p}]\!]$:

$$\theta(X) \in [\![\mathcal{C}_1(X)]\!] \ \Leftrightarrow \ \theta(\mathfrak{p}) \in [\![C_{1,X}]\!] \quad \text{and} \quad \theta(X) \in [\![\mathcal{C}_2(X)]\!] \ \Leftrightarrow \ \theta(\mathfrak{p}) \in [\![C_{2,X}]\!]$$

we give the following definitions of C_X:

- if $\mathcal{C}(X) = \mathcal{C}_1(X) \wedge \mathcal{C}_2(X)$ then $C_X = C_{1,X} \wedge C_{2,X}$
- if $\mathcal{C}(X) = \mathcal{C}_1(X) \vee \mathcal{C}_2(X)$ then $C_X = C_{1,X} \vee C_{2,X}$
- if $\mathcal{C}(X) = \neg\mathcal{C}_1(X)$ then $C_X = \neg C_{1,X}$
- if $\mathcal{C}(X) = \top$ then $C_X = \top$

This ends the proof that for all variable $X \in V$ and for all valuation $\theta \in [\![\mathcal{C}_\mathfrak{p}]\!]$:

$$\theta(X) \in [\![\mathcal{C}(X)]\!] \quad\Leftrightarrow\quad \theta(\mathfrak{p}) \in [\![C_X]\!]$$

Finally we define $\mathfrak{p}(\mathcal{C}) = \bigwedge_{X \in V} C_X$ which entails that, for all valuation $\theta \in [\![\mathcal{C}_\mathfrak{p}]\!]$:

$$\theta \in [\![\mathcal{C}]\!] \quad\Leftrightarrow\quad \theta(\mathfrak{p}) \in [\![\mathfrak{p}(\mathcal{C})]\!]$$

□

With these lemmas we can now translate tree constraints C_2 and C'_2 from tree pattern $\mathfrak{q}_2$ to tree pattern $\mathfrak{q}_1$ or tree pattern $\mathfrak{q}_3$. Some variables of $\mathfrak{q}_2$ may not appear in $\mathfrak{q}_1$, this is why we may need to translate some constraints to the output pattern $\mathfrak{q}_3$.

Among the rules we use to represent the behaviour of commands, only the last rule for the success case of the `cp -r` command has a variable which appears only in its output pattern. If this rule did not exist we could have altered our model of rewriting rules to force that all variables must appear in the input pattern; this would have allowed us to remove tree constraints on the output of rules only to have tree contraints on the input, making the model slightly simpler. This restriction of the model would make sense since a variable appearing only in the output pattern of a rule would imply, in scripts, that a command fills a directory with a number of different files or directories in a non-deterministic way. Although the specification of the `cp -r` command is in fact non-deterministic, we have used a non-deterministic approximation of its behaviour. We discuss this in more detail in subsection 3.2.8.

In order to translate the tree constraints C_2 and C'_2 into tree constraints on the input and the output, we start by translating them to a *valuation constraint* on the variables appearing in pattern $\mathfrak{q}_2$.

Before we use lemma 5, we check that the formula $C_0 = C_2 \wedge C'_2$ is satisfiable. To do this we put C_0 under disjunctive normal form. The normalisation process is the same as in boolean logic but with constraint literals (c.f. definition 6) in place of boolean literals. So there are conjunctions of constraint literals $C_{2,1}, \ldots, C_{2,k}$ such that C_0 is equivalent to $C_{2,1} \vee \cdots \vee C_{2,k}$.

We can decide the satisfiability of each conjunction $C_{2,i}$ by trying to build a tree that satisfies it: all positive literals impose the existence and the file type of a node in the tree, the existence of a node at a path implies that the nodes along the path are all directories. Contradictions can arise in three cases:

Contradiction between two positive literals: when two positive literals entail two different file types for a same node, e.g. `reg(/bin/touch)` and `sock(/bin)` imply that the node at path `/bin` is both a directory and a socket,

Contradiction between a positive and a negative literal: when a positive literal entails a file type for a node which is precisely forbidden by a negative literal, e.g. `reg`(`/bin/touch`) and $\neg$`dir`(`/bin`),

Contradiction with existential constraints: when several literals with constraints of the form $\exists(p, E)$ (c.f. definition 6) for a same path p lead to a contradiction. Checking if such literals produce a contradiction is reducible to the Dual-Horn satisfiability problem on boolean formulas. We recall that an instance of the Dual-Horn satisfiability problem is a conjunction of clauses such that each clause contains at most of negative literal.

The reduction maps each feature $f \in \mathcal{F}$ to a boolean variable noted $\hat{f}$. Each feature tree is associated with the valuation of variables where $\hat{f}$ is true if there is a node at path p/f in the tree. A positive literal $\exists(p, E)$ is associated with the formula $\varphi(p, E) = \bigvee_{f \in E} \hat{f}$. A negative literal $\neg\exists(p, E)$ is associated with the formula $\bigwedge_{f \in E} \neg\hat{f}$. For each feature f such that there are negative literals $\neg\mathsf{ft}(p/f)$ for each file type $f \in$ `types`, we add a formula $\neg\hat{f}$. The conjunction of the formulas cited above is a conjunction of Dual-Horn clauses; we define it as the instance S of the Dual-Horn satisfiability problem in the reduction.

It follows that, if a tree t satisfies these tree constraints, then the corresponding valuation of the boolean variables ($\hat{f}$ is true if there is a node at path p/f in the tree) satisfies formula S. The converse does not work because the set of boolean variables is infinite, and, by definition, a node in a feature tree cannot have an infinite number of child nodes. But we can simplify it by replacing some boolean variables.

In order to do this we note E_j for $j \leq l$ the finite sets appearing in existential constraints at path p and E'_j for $j \leq l'$ the cofinite sets appearing in existential constraint at path p. Then the set $E_0 = \bigcup_{j \leq l} E_j \cup \bigcup_{j \leq l'} \mathcal{F} \setminus E'_j$ is finite. So we can put together the variables from the cofinite set $E_\infty = \mathcal{F} \setminus E_0$. Then for all $j \leq l'$ we have: $E'_j = E_\infty \cup E''_j$ where $E''_j \subseteq E_0$ is finite. Finally we define a new boolean variable $\hat{f}_\infty$ such that $\hat{f}_\infty = \bigvee_{f \in E_\infty} \hat{f}$, and we can rewrite, for all $j \leq l'$:

$$\varphi(p, E'_j) = \hat{f}_\infty \vee \bigvee_{f \in E''_j} \hat{f}$$

Then the set of boolean variables in formula S is $\{\hat{f}_\infty\} \cup \{\hat{f}\}_{f \in E_0}$ which is finite. If a valuation of variables satisfies S then we can deduce a tree which satisfies the tree constraints mentioned above, where there is a node at a path p/f for some $f \in E_\infty$ if and only if the boolean variable $\hat{f}_\infty$ is true.

The complexity of the reduction is quasilinear (because of the computation of a finite union of finite sets for E_0) in the size of the disjunctive normal form of tree constraint $C_{2,i}$.

Complexity of checking satisfiability

We now analyze the complexity of computing tree constraints $C_{2,i}$ for all $i \leq k$ and checking their satisfiability. Assuming that tree constraints C_2 and C'_2 are given in disjunctive normal form, we have:

$$C_2 = \varphi_1 \vee \cdots \vee \varphi_n \quad \text{and} \quad C_2' = \varphi_1' \vee \cdots \vee \varphi_m'$$

where $\varphi_1, \ldots, \varphi_m'$ are conjunctions of literals. In this case:

$$C_2 \vee C_2' = \bigvee_{i \leq n, j \leq m} \varphi_i \wedge \varphi_j'$$

So we can define, for all $i \leq n$ and $j \leq m$: $C_{2,i+n(j-1)} = \varphi_i \wedge \varphi_j'$. The sum of the sizes of $C_{2,i}$ is:

$$\sum_{i \leq n*m} \|C_{2,i}\| = \sum_{i \leq n, j \leq m} \|\varphi_i\| + \left\|\varphi_j'\right\| = m * \|C_2\| + n * \|C_2'\|$$

The time complexity of checking the satisfiability of $C_{2,i}$ is quasilinear in the size of $C_{2,i}$, so the time complexity of checking the satifiability of $C_{2,i}$ for all $i \leq n * m$ is polynomial in $\|C_2\| + \|C_2'\|$, with a polynomial of order 4.

Finally, for each satisfiable tree constraint $C_{2,i}$ we compute the valuation constraint $\mathcal{C}_i = \mathfrak{q}_2{}^{-1}(C_{2,i})$, which gives one tree constraint for each variable X in the set V_2 of the variables in tree pattern $\mathfrak{q}_2$. Noting V_1 and V_3 the sets of variables in patterns $\mathfrak{q}_1$ and $\mathfrak{q}_3$ respectively, we can split the set V_2 into three disjoint sets as follows:

$$V_2 = (V_2 \cap V_1) \uplus (V_2 \cap (V_3 \setminus V_1)) \uplus (V_2 \setminus (V_1 \cup V_3))$$

We note $\mathcal{C}_{1,i}, \mathcal{C}_{3,i}$ and $\mathcal{C}_{2,i}$ the restrictions of $\mathcal{C}_i$ to the sets $(V_2 \cap V_1), (V_2 \cap (V_3 \setminus V_1))$ and $(V_2 \setminus (V_1 \cup V_3))$ respectively. Since $C_{2,i}$ is satisfiable, so are $\mathcal{C}_{1,i}, \mathcal{C}_{3,i}$ and $\mathcal{C}_{2,i}$.

The constraints on variables in $V_2 \cap V_1$ we can translate on the input pattern $\mathfrak{q}_1$, and those on variables in $V_2 \cap (V_3 \setminus V_1)$ we can translate on the output pattern $\mathfrak{q}_3$. For each $i \leq k$ we define the input tree constraint $C_{1,i} = C_1 \wedge \mathfrak{q}_1(\mathcal{C}_{1,i})$, the output tree constraint $C_{3,i} = C_3 \wedge \mathfrak{q}_3(\mathcal{C}_{3,i})$ and the tree pattern transformation $T_i' = (C_{1,i}, R, C_{3,i})$ with the rewriting rule $R = (\mathfrak{q}_1, \mathfrak{q}_3)$. We now prove the final lemma of the procedure of composition.

Lemma 7 We have the following equality of relations:

$$g_{T_2} \circ g_{T_1} = \bigcup_{i \leq k} g_{T_i'}$$

Proof

We start by showing that, for all feature trees $t_1, t_3 \in \mathcal{T}$:

$$t_1 \rightarrow_R t_3 \quad \Leftrightarrow \quad \exists t_2 \in \mathcal{T},\ t_1 \rightarrow_{R_1} t_2 \rightarrow_{R_2} t_3$$

If $t_1 \rightarrow_R t_3$ then there exists a valuation θ_R such that $t_1 = \theta_R(\mathfrak{q}_1)$ and $t_3 = \theta_R(\mathfrak{q}_3)$. By definition of $\mathfrak{q}_1$ and $\mathfrak{q}_3$ we have a valuation of variables υ such that $t_1 = \theta_R(\upsilon(\mathfrak{p}_1))$ and $t_3 = \theta_R(\upsilon(\mathfrak{p}_3))$. Therefore, by definition of $\mathfrak{q}_2$, we have $\theta_R(\mathfrak{q}_2) = \theta_R(\upsilon(\mathfrak{p}_2)) = \theta_R(\upsilon(\mathfrak{p}_2'))$. Noting $t_2 = \theta_R(\mathfrak{q}_2)$ we have $t_1 \rightarrow_{R_1} t_2$ and $t_2 \rightarrow_{R_2} t_3$.

If there exists a feature tree t_2 such that $t_1 \rightarrow_{R_1} t_2 \rightarrow_{R_2} t_3$ then there are two valuations θ and θ' of the variables in $\mathfrak{p}_2$ and $\mathfrak{p}'_2$ respectively such that $t_2 = \theta(\mathfrak{p}_2) = \theta'(\mathfrak{p}'_2)$. Because υ minimally unifies $\mathfrak{p}_2$ and $\mathfrak{p}'_2$, there exists a valuation θ_0 of the variables in $\mathfrak{q}_2$ such that $\theta = \theta_0 \circ \upsilon$. Therefore $t_1 = \theta_0(\upsilon(\mathfrak{p}_1)) = \theta_0(\mathfrak{q}_1)$ and $t_3 = \theta_0(\upsilon(\mathfrak{p}_3)) = \theta_0(\mathfrak{q}_3)$. So $t_1 \rightarrow_R t_3$.

Now, given two feature trees t_1 and t_3 such that $t_1 \rightarrow_R t_3$, we aim to prove that:

$$(t_1, t_3) \in g_{T_2} \circ g_{T_1} \quad \Leftrightarrow \quad \exists i \leq k, (t_1, t_3) \in g_{T'_i}$$

If $(t_1, t_3) \in g_{T'_i}$ for some $i \leq k$, then $t_1 \in [\![C_1 \wedge \mathfrak{q}_1(\mathcal{C}_{1,i})]\!]$ and $t_3 \in [\![C_3 \wedge \mathfrak{q}_3(\mathcal{C}_{3,i})]\!]$. So there exists a valuation θ such that $t_1 = \theta(\mathfrak{q}_1)$, $t_3 = \theta(\mathfrak{q}_3)$ and the restrictions of θ to the sets $V_2 \cap V_1$ and $V_2 \cap (V_3 \setminus V_1)$ respectively satisfy valuation constraints $\mathcal{C}_{1,i}$ and $\mathcal{C}_{3,i}$. Since valuation constraint $\mathcal{C}_{2,i}$ is satisfiable there must be a valuation θ' of the variables in $V_2 \setminus (V_1 \cup V_3)$ which satisfies it. Then we define the valuation θ_0 as equal to θ on $V_1 \cup V_3$ and equal to θ' on $V_2 \setminus (V_1 \cup V_3)$. Then $t_1 = \theta_0(\mathfrak{q}_1)$, $t_3 = \theta_0(\mathfrak{q}_3)$ and θ_0 satisfies the valuation constraint $\mathcal{C}_i$. Because $\mathcal{C}_i = \mathfrak{q}_2^{-1}(C_{2,i})$, the feature tree $t_2 = \theta_0(\mathfrak{q}_2)$ satisfies the tree constraint $C_{2,i}$, and so it also satisfies the tree constraint $C_0 = C_2 \wedge C'_2$. The construction of $\mathfrak{q}_1, \mathfrak{q}_2$ and $\mathfrak{q}_3$ imply that $(t_1, t_2) \in g_{R_1}$ and $(t_2, t_3) \in g_{R_2}$. We have also shown that t_1, t_2 and t_3 respectively satisfy tree constraints $C_1, C_2 \wedge C'_2$ and C_3, so: $(t_1, t_3) \in g_{T_2} \circ g_{T_1}$.

If $(t_1, t_3) \in g_{T_2} \circ g_{T_1}$ then there is a feature tree t_2 such that $(t_1, t_2) \in g_{T_1}$ and $(t_2, t_3) \in g_{T_2}$. So t_1, t_2 and t_3 respectively satisfy tree constraints $C_1, C_2 \wedge C'_2$ and C_3. Since $C_2 \wedge C'_2$ is equivalent to $C_{2,1} \vee \cdots \vee C_{2,k}$, there must exist an $i \leq k$ such that t_2 satisfies $C_{2,i}$. Because υ minimally unifies $\mathfrak{p}_2$ and $\mathfrak{p}'_2$ there must be a valuation θ_0 such that $t_1 = \theta_0(\upsilon(\mathfrak{p}_1)), t_2 = \theta_0(\upsilon(\mathfrak{p}_2)) = \theta_0(\upsilon(\mathfrak{p}'_2))$ and $t_3 = \theta_0(\upsilon(\mathfrak{p}_3))$. Considering $t_2 = \theta_0(\mathfrak{q}_2)$, valuation θ_0 has to satisfy valuation constraint $\mathcal{C}_i$. So t_1 and t_3 respectively satisfy tree constraints $\mathfrak{q}_1(\mathcal{C}_{1,i})$ and $\mathfrak{q}_3(\mathcal{C}_{3,i})$. It then follows from $t_1 = \theta_0(\mathfrak{q}_1) \in [\![C_1]\!]$ and $t_3 = \theta_0(\mathfrak{q}_3) \in [\![C_3]\!]$ that $(t_1, t_3) \in g_{T'_i}$.
□

This lemma proves that the composition of tree pattern transformations T_1 and T_2 can be expressed as the tree pattern transducer $(T'_1, \ldots, T'_k)$. This procedure works for any two tree pattern transformations T_1 and T_2, so we can use it to compute the composition of tree pattern transducers.

3.3.4 Conclusion on composition

We have shown the following theorem:

Theorem 2 *For all two tree pattern transducers τ_1 and τ_2, we can compute a tree pattern transducer τ_3 such that $g_{\tau_3} = g_{\tau_2} \circ g_{\tau_1}$.*

Proof
Tree pattern transducer τ_3 is obtained by applying the composition procedure to each pair composed of a transformation from τ_1 and a transformation from τ_2, and concatenating the resulting lists of transformations. □

Corollary 3 The set of tree-to-tree functions described by tree pattern transducers is closed under composition.

The closure under composition of the model of tree pattern transducers is an important argument in favor of using it to verify properties of scripts and their interaction with file systems.

Complexity of the composition procedure

We now analyse the complexity of the procedure computing the composition of tree pattern transducers. Tree pattern transducers are composed of several tree pattern transformations, so we start with the complexity of computing the composition of two tree pattern transformations $T_1 = (C_1, (\mathfrak{p}_1, \mathfrak{p}_2), C_2)$ and $T_2 = (C'_2, (\mathfrak{p}'_2, \mathfrak{p}_3), C_3)$. We compute this complexity under the assumptions that patterns $\mathfrak{p}_1$ and $\mathfrak{p}'_2$ are linear, that pattern $\mathfrak{p}_2$ has at most one duplication of variable, and that tree constraints C_1, C_2, C'_2 and C_3 are under disjunctive form.

The first step is computing the composition of rewriting rules, this is done in time quadratic in $\|T_1\| + \|T_2\|$. The sizes of the minimal unifier v and of the unification pattern $\mathfrak{q}_2$ are linear in the size of $\mathfrak{p}_2$ and $\mathfrak{p}'_2$.

The next step is computing constraints $C_{2,i}$ for $i \leq k$ and checking their satisfiability. The sum of the sizes of the $C_{2,i}$ for $i \leq k$ is quadratic in the size of constraints C_2 and C'_2, and the $C_{2,i}$ for $i \leq k$ are computed in quadratic time in the size of constraints C_2 and C'_2. The satisfiabilities of $C_{2,i}$ for all $i \leq k$ are checked in time polynomial in the sizes of C_2 and C'_2, with a polynomial of order 4.

Finally we compute $\mathfrak{q}_1(\mathfrak{q}_2{}^{-1}(C_{2,i}))$ and $\mathfrak{q}_3(\mathfrak{q}_2{}^{-1}(C_{2,i}))$ for each $i \leq k$. For each $i \leq n * m$, computing $\mathfrak{q}_2{}^{-1}(C_{2,i})$ is in time quasiquadratic in the sizes of $\mathfrak{q}_2{}^{-1}$ and $C_{2,i}$, so it is quasiquadratic in $\|T_1\| + \|T_2\|$. The size of $\mathfrak{q}_2{}^{-1}(C_{2,i})$ is linear in $\|T_1\| + \|T_2\|$. Therefore $\mathfrak{q}_1(\mathfrak{q}_2{}^{-1}(C_{2,i}))$ and $\mathfrak{q}_3(\mathfrak{q}_2{}^{-1}(C_{2,i}))$ are of size linear in $\|T_1\| + \|T_2\|$, and computing them is in time quasiquadratic in $\|T_1\| + \|T_2\|$. Computing those for each $i \leq n * m$ is polynomial in $\|T_1\| + \|T_2\|$, with a polynomial of order 5.

Summing up, the time complexity of computing the composition of two tree pattern transformations T_1 and T_2 is polynomial in $\|T_1\| + \|T_2\|$, with a polynomial of order 5. It is important to note that the most costly part of the algorithm is the translation of tree constraints.

Finally the computation of the composition of two tree pattern transducers τ_1 and τ_2 has time complexity $O((\|\tau_1\| + \|\tau_2\|)^7)$ because we compute the composition of all pairs composed of a transformation from τ_1 and a transformation from τ_2.

Although we've made efforts to give a lower bound for the composition of our model of transducers, the important result is that the model is closed under composition and represents the behaviour of Unix commands with reasonable accuracy. This is because we do not use this algorithm to verify properties of scripts, instead we only compute inverse images of sets of trees by successive Unix commands.

Computing successive inverse images of the set of all trees through several tree transformations is equivalent to computing the domain of the composition of those transformations.

The domain of tree pattern transformations is characterized by their input constraint, so the procedure of composition already gives an algorithm for computing the domains of compositions of transformations. Furthermore we can cut from the algorithm the computations of the output constraints and the tree patterns. This gives us a first algorithm for computing domains of compositions of transformations.

In chapter 4 we show how we improved this algorithm and present the results of its implementation.

Chapter 4

Implementation

In this chapter we present the algorithm we have implemented in the CoLiS project. Our implementation builds on a tool designed by Nicolas Jeannerod, Benedikt Becker, Yann Régis Gianas, Claude Marché, Mihaela Sighireanu, and Ralf Treinen as part of the CoLiS project. This tool solves the problem of, given a Shell script, determining the file systems on which the script can succeed and the ones on which it can fail. It works in two steps: in a first step it lists the scenarios of execution of the script which are compatible with the structure of the script, each scenario is represented by a sequence of succeeding or failing commands. These sequences are called *script traces*. In a second step it checks, for each *script trace*, whether there is a file system compatible with that script trace. The parts of the tool computing the first and second steps are called respectively the concrete interpreter and the symbolic interpreter. Our implementation reuses the existing concrete interpreter, so our contribution consists only in a new version of the symbolic interpreter.

In section 4.1 we discuss the algorithm we use, which is inspired by the one presented in chapter 3 where we recursively compute constraints put by scripts on the file system. In a second section we present the practical results and compare them with those of the first CoLiS tool. In a last section we discuss the benefits of our approach, for this problem and for verification problems on scripts in general.

4.1 The algorithm

In this section we define the algorithm which computes the sets of file systems on which a given script trace[1] works. We can see each script trace as a function transforming one file system into another. What our algorithm computes is then the domain of the function of a script trace, or the inverse image of the set of all file systems through the function. This function is the composition of the functions associated with the succeeding or failing commands that compose the script trace. Our approach to this algorithm to computing this inverse image is to compute the inverse images successively through each succeeding or failing command in the script trace. In this sense we call our approach backward, as opposed to the forward approach used in the algorithm used by Nicolas

[1] c.f. definition 1.

Jeannerod, Benedikt Becker, Yann Régis Gianas, Claude Marché, Mihaela Sighireanu, and Ralf Treinen in their implementation solving the same problem.

We have shown in section 2.4 that computing the domain of a script trace is NP-hard. This means that our algorithm for this problem cannot be better than NP in the general case. But we can still improve the complexity in some specific cases. In particular, we have argued in section 2.4 that the NP-hardness of the problem came from commands `mv`, `cp` (provided the logical operators used with command `test` are split into distinct trace atoms), and so we give an algorithm with polynomial time complexity in the special case where commands `mv` and `cp` are forbidden, and command `test` is forbidden from using logical operators.

4.1.1 Naive version of the algorithm

In this subsection we describe an early version of our algorithm based on computing compositions of tree pattern transductions, this version is not used in our implementation.

In chapter 3 we have presented how to compute the tree pattern transduction representing the action of a script trace on the file system. The domain of a tree pattern transduction is the union of the domains of the tree pattern transformations composing it. By definition the domain of a tree pattern transformation is characterized by its input tree constraint. So we can compute the tree pattern transduction representing a script trace and get a characterization of its domain.

The fact that we only compute the input tree consraint of a composition of transformations greatly simplifies the algorithm, let us analyze its complexity. Computing the input tree constraint of two tree pattern transformations $T_1 = (C_1, (\mathfrak{p}_1, \mathfrak{p}_2), C_2)$ and $T_2 = (C'_2, (\mathfrak{p}'_2, \mathfrak{p}_3), C_3)$ is then done in three steps:

1. We check that the tree constraint $C_2 \wedge C'_2$ is satisfiable. Assuming that tree constraints are kept under disjunctive normal form, checking this is done in time polynomial in the size of $C_2 \wedge C'_2$, with a polynomial of order 4.

2. We translate the constraint $C_2 \wedge C'_2$ onto the input tree of T_1. Writing $C_2 \wedge C'_2 = \bigvee_{i \leq k} C_{2,i}$ where formulas $C_{2,i}$ are conjunctions of literals, we compute:

$$C'_1 = \bigvee_{i \leq k} \mathfrak{p}_1(\mathfrak{p}_2^{-1}(C_{2,i}))$$

   Then the input constraint of the composition of T_1 and T_2 is $C_1 \wedge C'_1$. The time complexity of this step is quasiquadratic in the sum the sizes of $\mathfrak{p}_1$, $\mathfrak{p}_2$ and the $C_{2,i}$ for each $i \leq k$, so it is polynomial in the sizes of T_1 and T_2, with a polynomial of order 5.

3. We put $C_1 \wedge C'_1$ under disjunctive normal form and check its satisfiability. The time complexity is quasiquadratic in the sum of the sizes of C_1 and C'_1, so polynomial in the sizes of T_1 and T_2 with a polynomial of order 5. The size of the resulting constraint depends linearly on the product of the sizes of C_1, C_2 and C'_2, which is polynomial in the sum of the sizes of T_1 and T_2 with a polynomial of order 3.

In total, the time complexity of computing the input constraint of the composition of two tree pattern transformations is polynomial in the sizes of the transformations, with a polynomial of order 5. But this complexity is only for the composition of two transformations. When computing the inverse image of a script trace we have two complexity hurdles: each script command may have multiple transformations, and when we compose more than two commands the constraints grow with each composition. These two problems: the multiplicity of transformations and the growth of the input tree constraint with each composition, make the time complexity of computing the domain of a script trace doubly-exponential. We make adjustments to this algorithm in order to improve the complexity.

4.1.2 The improved algorithm

Our first goal is to reduce the growth of the representation of the inverse images we successively compute for each command in a script trace. So far we have only used tree constraints to represent sets of feature trees. We could stop putting rules under disjunctive normal form, but then the time complexity of checking their satisfiability would be exponential. Instead we design **a new model** in order to represent sets of feature trees.

Our second goal is to reduce the number of transformations representing the action of script commands. We do this by altering the model of rewriting rules.

These models for representing sets of trees and transformations of trees are specifically designed for the problem we are trying to solve. In particular they make use of the fact that we only compute inverse images. Computing only inverse images entails that we never have constraints stating that two subtrees must be equal or similar in some way. This means that any constraint put on a subtree at a given path in the file system does not propagate outside of the subtree at that path, and so we can check if this constraint leads to a contradiction only by looking at the other constraints on the subtree. In a way constraints become *local*. A constraint on a subtree can have non-local consequences only when a subtree (i.e. a directory) is moved or copied using commands `mv` or `cp -r`. The inverse of the `mv` command only moves a local constraint from one subtree to another. The inverse of the `cp -r` command asserts that two subtrees are the same and removes one of them; this requires to compute the conjunction of the constraints on the two subtrees, but we already compute conjunctions of tree constraints

This leads us to represent constraints in a tree-like structure. This model is mainly inspired by prefix sets of trees, so we name it *prefix feature trees*. We recall that the set of types of files is $\mathtt{types} = \{\mathsf{dir}, \mathsf{reg}, \mathsf{symlink}, \mathsf{fifo}, \mathsf{block}, \mathsf{char}, \mathsf{sock}\}$.

Definition 17 A *local existential constraint* is an existential tree constraint $\exists(\epsilon, E)$ on the empty path ϵ.

A *local existential literal* is either a *local existential constraint* $\exists(\epsilon, E)$ or the negation of a *local existential constraint* $\neg\exists(\epsilon, E)$.

We note $\mathcal{CC}$ the set of conjunctions of *local existential literals*.

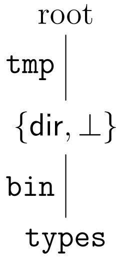


Figure 4.1 – Prefix tree representing the constraint $\neg\mathsf{node}(/\mathtt{tmp}) \vee (\mathsf{dir}(/\mathtt{tmp}) \wedge \mathsf{node}(/\mathtt{tmp/bin}))$

The sets of *prefix feature trees* and *prefix nodes* are inductively defined by, respectively:

$$\mathsf{ptrees} = \mathcal{CC} \times (\mathcal{F} \rightsquigarrow \mathsf{pnodes})$$

$$\mathsf{pnodes} = \mathcal{P}(\mathtt{types} \cup \{\bot\}) \times \mathsf{ptrees}$$

where $\mathcal{P}(\mathtt{types} \cup \{\bot\})$ denotes the set of parts of $\mathtt{types} \cup \{\bot\}$ and $\mathcal{F} \rightsquigarrow \mathsf{pnodes}$ denotes the set of finite mappings from $\mathcal{F}$ to pnodes.

The sets of feature trees associated with *prefix feature trees* and *prefix nodes* are defined by, for all prefix feature tree of the form $(C, \mathsf{map}) \in \mathcal{CC} \times (\mathcal{F} \rightsquigarrow \mathsf{pnodes})$ and prefix node of the form $(S, \mathsf{t}) \in \mathcal{P}(\mathtt{types} \cup \{\bot\}) \times \mathsf{ptrees}$:

$$\begin{aligned} \llbracket (C, \mathsf{map}) \rrbracket = \llbracket C \rrbracket \cap \{\mathsf{map}' \mid{}& \forall f \in \mathrm{dom}(\mathsf{map}) \cap \mathrm{dom}(\mathsf{map}'), \mathsf{map}'(f) \in \llbracket \mathsf{map}(f) \rrbracket, \\ & \forall f \in \mathrm{dom}(\mathsf{map}') \setminus \mathrm{dom}(\mathsf{map}), \qquad \bot \in \llbracket \mathsf{map}'(f) \rrbracket \ \} \\ \llbracket (S, \mathsf{t}) \rrbracket ={}& (S \setminus \{\mathsf{dir}\}) \cup ((S \cap \{\mathsf{dir}\}) \times \llbracket \mathsf{t} \rrbracket) \end{aligned}$$

We call *full prefix feature tree* the prefix feature tree $(\top, \emptyset)$. We note it $p_\top$ and its associated set of feature trees is the set of all feature trees: $\llbracket p_\top \rrbracket = \mathcal{T}$.

The symbol $\bot$ is used as a file type, it represents nodes which are absent from a tree. For example, the command `mkdir /tmp/bin` fails either if directory `/tmp` is absent or if there is already a node at path `/tmp/bin`. The set of feature trees which satisfy this condition is the domain of the failure case of the command. This set is characterized by the prefix feature tree shown in Figure 4.1. In this representation we do not show the local existential constraints because they are all $\top$. The exact prefix feature tree is $\mathsf{t} = (\top, [\mathtt{tmp}/(\{\mathsf{dir}, \bot\}, (\top, [\mathtt{bin}/(\mathtt{types}, \emptyset)]))])$. This prefix feature tree can be read from top to bottom starting with the root: the edge `tmp` leading to the node $\{\mathsf{dir}, \bot\}$ means that a feature tree in the set $\llbracket \mathsf{t} \rrbracket$ has either a directory or no node at path `/tmp`; the edge `bin` leading to the node `types` means that, if there is a directory node at path `/tmp`, then there is a node at path `/tmp/bin` (its filetype is in `types` which is the set of all filetypes).

The two main differences between this model and prefix sets of feature trees are the use of local existential constraints and the fact that each node allows a set of filetypes instead of just one file type.

Local existential constraints are necessary in order to express existential tree constraints which are used in the input constraints of some commands such as `rmdir`, `move` and `cp`.

Each node allowing a set of file types instead of just one file type is useful because it can replace a tree constraint which is a disjunction of literals. For example in section 3.2 we describe the constraints of several transformations using the constraint $\mathsf{node}(p)$ where p is a path, which is defined as $\mathsf{node}(p) = \bigcup_{\mathsf{ft} \in \mathtt{types}} \mathsf{ft}(p)$; this constraint can be expressed in this new model with the prefix feature tree containing the node $(\mathtt{types}, p_\top)$ at path p, and such that the sets of file types of the nodes along path p all are the singleton $\{\mathsf{dir}\}$. The growth of the tree constraints during the computation of inverse images is driven by the disjunctions inside them. By removing disjunctions we improve the informal complexity of our algorithm.

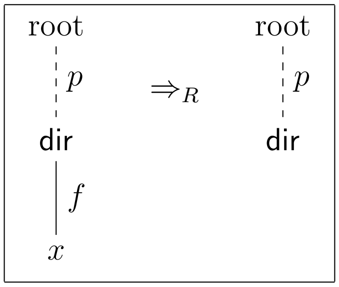
With this representation of sets of trees, we can represent the input and output constraints of the succeeding and failing version of the script commands presented in section 3.2, except for `mv` and `cp`. We give more detail on this later.

The process of computing inverse images of prefix feature trees is similar to that of computing inverse images of tree constraints.

Alterations to rewriting rules

In order to reduce the number of transformations for each command, we modify the model of tree pattern rewriting rules. The new version of the model must allow us to still compute inverse images of prefix feature trees through it. Similarly to prefix feature trees, we allow nodes of tree patterns to represent a disjunction of file types instead of just one file type.

For example, we had 6 different rules for the success of the `rm` command depending on the file type of the file to remove. Instead we can replace these 6 rules with just one. The success of command `rm` p/f then gives the rule:



where x is a *file type variable*, with the constraint $x \in \mathtt{types} \setminus \{\mathsf{dir}\}$. We formally define this new type of tree patterns, we call them *prefix patterns*:

Definition 18 Given a set V of variables and a set U of file type variables, *prefix patterns* with variables in V and file type variables in U are inductively defined (similarly to tree patterns) as:

$$\mathcal{PP} = (V \cup \{\bot\}) \times (\mathcal{F} \rightsquigarrow ((U \cup \mathtt{types}) \times \mathcal{PP}))$$

For all file type variable x, a *file type constraint* on x is a subset of $\mathtt{types} \cup \{\bot\}$.

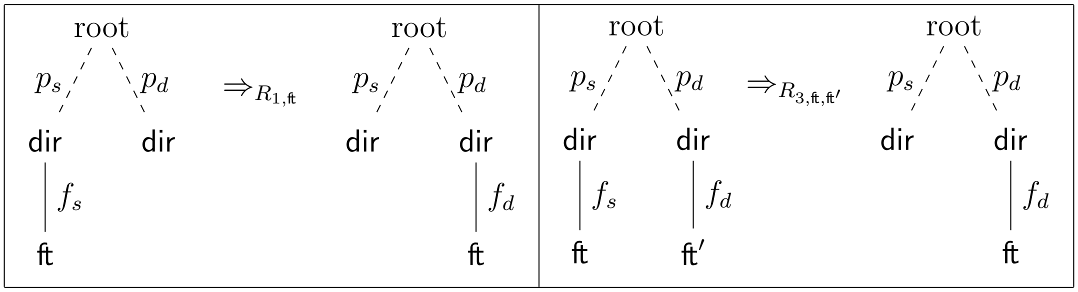
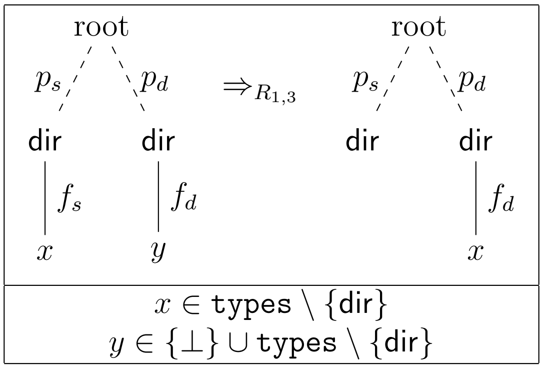

root
p_s
p_d
$\Rightarrow_{R_{1,\mathsf{ft}}}$
dir
f_s
f_d
ft
$\Rightarrow_{R_{3,\mathsf{ft},\mathsf{ft}'}}$
ft′

Figure 4.2 – Rules $R_{1,\mathsf{ft}}$ (left) and $R_{3,\mathsf{ft},\mathsf{ft}'}$ (right) of *success*(`mv` p_s/f_s p_d/f_d)



$x \in \texttt{types} \setminus \{\mathsf{dir}\}$
$y \in \{\bot\} \cup \texttt{types} \setminus \{\mathsf{dir}\}$

Figure 4.3 – Unique rule $R_{1,3}$ merging all versions of rules $R_{1,\mathsf{ft},\mathsf{ft}'}$ and $R_{3,\mathsf{ft},\mathsf{ft}'}$

File type constraints allow us to put restrictions on the file types of nodes. It is useful in the case of the `rm` command where we know that the file to remove is not a directory.

Similarly to prefix feature trees, we allow the value $\bot$ for file types of nodes; a node with file type $\bot$ represents an absent node. This helps us fuse together some transformations of `mv` and `cp`. For example, the rules $R_{1,\mathsf{ft}}$ and $R_{3,\mathsf{ft},\mathsf{ft}'}$ for the success of command `mv` p_s/f_s p_d/f_d shown in Figure 4.2. $R_{1,\mathsf{ft}}$ and $R_{3,\mathsf{ft},\mathsf{ft}'}$ actually represent 42 rules (6 versions of R_1 and 36 versions of R_3), each with different values for the file types $\mathsf{ft}, \mathsf{ft}' \in \texttt{types} \setminus \{\mathsf{dir}\}$. They can all be replaced with the one rule $R_{1,3}$ shown in Figure 4.3.

With the new prefix feature tree model and the alterations we made to the model of rewriting rules we can simplify the representation of script commands as transducers. Using simplifications similar to the one shown in Figures 4.2 and 4.3, we get one transformation for each success and error case of each command presented in section 3.2, except for commands `mv`, `cp` and `touch`. The number of rules for these last 3 commands is still greatly reduced.

The command `touch` is particular because, even if we cannot express its success case with only one rewriting rule, we can still express the inverse image of a prefix feature tree through it as only one prefix feature tree. If we tried to represent the success case of `touch` p/f in one rewriting rule we would have the rule shown in Figure 4.4, but with two possible combinations of constraints for x and y. If $x \in \texttt{types} \setminus \{\mathsf{dir}, \bot\}$ then `touch` p/f

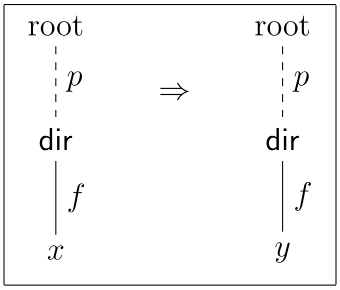


Figure 4.4 – Rule of $success(\texttt{touch}\ p/f)$

does not modify the file system and therefore $y = x$, but if $x = \bot$ then $\texttt{touch}\ p/f$ creates a regular file and so $y = \mathsf{reg}$. We can express the inverse image of a prefix feature tree t (representing a set of trees) through command $\texttt{touch}\ p/f$: noting (S_y, map) the node at path p/f in t, the corresponding node in the inverse image would be $(S_x, \emptyset)$ where the set S_x of possible file types for x is $S_x = (S_y \setminus \{\mathsf{dir}\}) \cup \{\bot\}$ if $\mathsf{reg} \in S_y$, and $S_x = S_y \setminus \{\mathsf{dir}\}$ otherwise. We use this in the implementation in order to compute the inverse image of a prefix feature tree through the success case of command `touch`. This allows us to compute the inverse image of a prefix feature tree through $success(\texttt{touch}\ p/f)$ as one prefix feature tree (instead of a list of prefix feature trees).

Finally, we make a last modification to our model in order to remove approximations we had made on the behaviours of the `which` command and of the `test` command with option `-x`. These commands' success and failure cases depend on whether or not a regular file is executable. Since we already deal with 7 different types of files, we can also split the file type reg of regular files into two new file types exec and $\mathsf{non\text{-}exec}$ which respectively represent regular executable files and regular non-executable files.

The problem with `mv` and `cp`

With the modifications we have brought to our algorithm for computing inverse images of commands efficiently, we still have several rules for commands `mv` and `cp`.

Compared to the initial number of rules in subsection 3.2.7 and subsection 3.2.8, we have merged the following rules:

- In the success case of `mv`, we merge rules $T_{1,\mathsf{ft}}$ and $T_{3,\mathsf{ft},\mathsf{ft}'}$ together into one rule, we merge rules $T_{2,\mathsf{ft}}$ and $T_{4,\mathsf{ft},\mathsf{ft}'}$ together similarly, and we merge rules T_6 and T_7 together. With this we get only 4 rules for the success case of `mv`, down from 87.

- In the success case of `cp`, we merge rules $T_{1,\mathsf{ft}}$ and $T_{3,\mathsf{ft},\mathsf{ft}'}$ for all values of $\mathsf{ft}, \mathsf{ft}' \in \texttt{types} \setminus \{\mathsf{dir}\}$ into one rule, and we merge rules $T_{2,\mathsf{ft}}$ and $T_{4,\mathsf{ft},\mathsf{ft}'}$ together into one rule similarly. This leaves only 2 rules for the success case of `cp`, down from 84.

- In the success case of `cp -r`, the same simplifications make the success case of `cp -r` go from 88 rules down to 5.

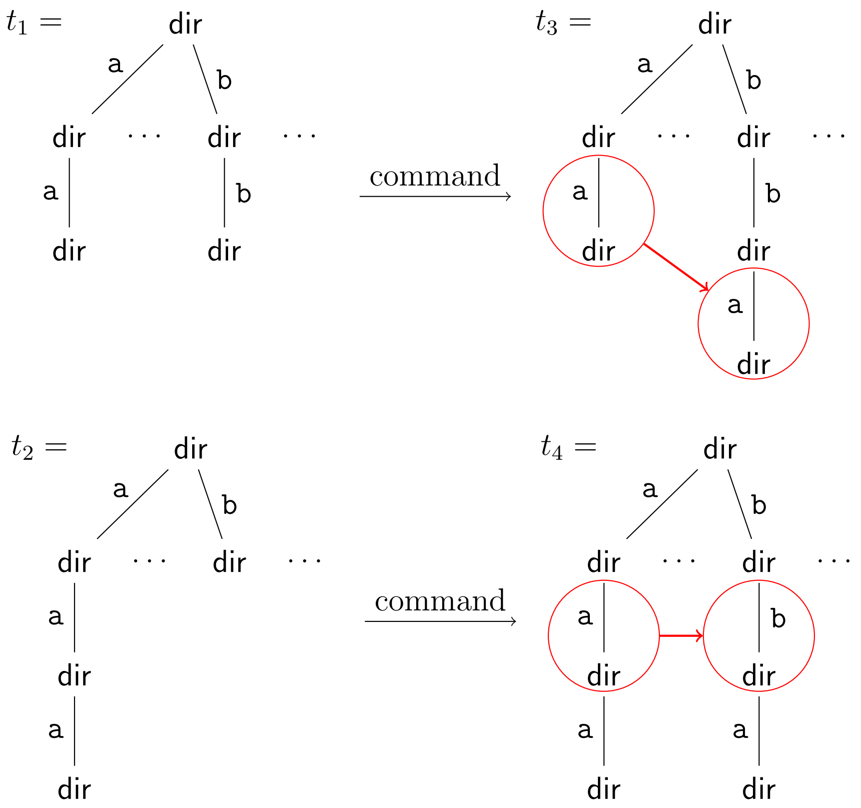


Figure 4.5 – Two trees mapped to the same output by command `cp -r /a/a /b/b`

The inverse image of a prefix feature tree by a trace atom of command `mv` or `cp` cannot always be represented as a prefix feature tree. This is for the same reason that the problem of computing the domains of script traces is NP-hard, as we have shown in section 2.4. That property, which inverse images of prefix feature trees by trace atoms of command `mv` or `cp` have, we call it *prefix-splittability*.

Definition 19 We say that a set S of feature trees is *prefix-splittable* if there exists a feature tree t containing two variables x and y, and 4 feature trees $t_{x,1}$, $t_{x,2}$, $t_{y,1}$ and $t_{y,2}$ such that:

$$t[x/t_{x,1}, y/t_{y,1}] \in S \text{ and } t[x/t_{x,2}, y/t_{y,2}] \in S$$

$$t[x/t_{x,1}, y/t_{y,2}] \notin S \text{ and } t[x/t_{x,2}, y/t_{y,1}] \notin S$$

This property is the one we use in the proof of NP-hardness in section 2.4. We can see this on the example in Figure 4.5 where we show two input trees t_1 and t_2 that are mapped by command `cp -r /a/a /b/b` onto trees t_3 and t_4 respectively. We can see that `cp -r` copies things differently depending of whether its target path `/b/b` exists or not, we highlight in a red circle how the directory is copied in each case. This behaviour is responsible for the inverse image of `cp -r` not being *prefix-splittable*. Indeed we can check that the set $\{t_3, t_4\}$ can be expressed as a prefix feature tree, and that trees t_1 and t_2 can be written as $t_1 = t[x/t_{x,1}, y/t_{y,1}]$ and $t_2 = t[x/t_{x,2}, y/t_{y,2}]$ such that the trees $t[x/t_{x,1}, y/t_{y,2}]$

and $t[x/t_{x,2}, y/t_{y,1}]$ are not in the inverse image of $\{t_3, t_4\}$ by command `cp -r /a/a /b/b`.

This means that we need to use disjunctions of prefix feature trees to represent the inverse images of commands `cp` and `mv`. We use lists of prefix feature trees to represent disjunctions of prefix feature trees. The sets they represent are therefore unions of sets described by prefix feature trees.

Complexity analysis

We now analyze the time complexity of this new algorithm computing the set of file systems compatible with a given script trace. We start with the complexity of computing the inverse image of a prefix feature tree through one prefix pattern transformation. Let $\mathfrak{t} \in \mathsf{ptrees}$ be a prefix feature tree and $T = (\mathfrak{t}_1, (\mathfrak{p}_1, \varphi, \mathfrak{p}_2), \mathfrak{t}_2)$ a prefix pattern transformation. The inverse image of $\mathfrak{t}$ through T is computed in three steps as in subsection 4.1.1:

1. We check that the conjunction $\mathfrak{t}'$ of $\mathfrak{t}$ and $\mathfrak{t}_2$ does not represent the empty set of feature trees. The conjunction of $\mathfrak{t}$ and $\mathfrak{t}_2$ is of size inferior to the sum of the sizes of $\mathfrak{t}$ and $\mathfrak{t}_2$. It is computed in time linear in the sum of the sizes of $\mathfrak{t}$ and $\mathfrak{t}_2$.

   We check that $\mathfrak{t}'$ does not represent the empty set of feature trees node-by-node starting with leaves. For each node, we check that its local existential constraints and the file names of its child nodes are not in contradiction. For this we use the same algorithm as the one for checking the satisfiability of tree constraints presented in subsection 3.3.3. Its time complexity is quasilinear in the sizes of the existential constraints and the number of file names of child nodes.

   In total the time complexity of computing $\mathfrak{t}'$ and checking if its set of feature trees is empty is quasilinear in the size of $\mathfrak{t}'$, so it is quasilinear in the sum of the sizes of $\mathfrak{t}$ and $\mathfrak{t}_2$.

2. We compute the inverse image $\mathfrak{t}''$ of $\mathfrak{t}'$ through the rewriting rule $(\mathfrak{p}_1, \varphi, \mathfrak{p}_2)$.

   Except for the rules of `cp -r` which copy a subtree, this step consists mainly in putting constraints on subtrees or moving subtrees, in such cases the time complexity is linear in the sum of the sizes of $\mathfrak{t}'$ and $(\mathfrak{p}_1, \varphi, \mathfrak{p}_2)$, and the size of $\mathfrak{t}''$ is inferior to the sum of the sizes of $\mathfrak{t}'$ and T.

   In the case where the rule copies a subtree, computing the inverse image also requires to compute the conjunction of the copied subtree and its copy. Then time complexity is still linear, and the size of $\mathfrak{t}''$ is still inferior to the sum of the sizes of $\mathfrak{t}'$ and T.

3. We check that the conjunction of $\mathfrak{t}''$ and $\mathfrak{t}_1$ does not represent the empty set of feature trees. Again the time complexity is quasilinear in the sum of the sizes of $\mathfrak{t}''$ and $\mathfrak{t}_1$, which means quasilinear in the sum of the sizes of $\mathfrak{t}$ and T.

The time complexity of computing the inverse image of $\mathfrak{t}$ through T is therefore quasilinear in the sum the the sizes of $\mathfrak{t}$ and T. The size of the computed inverse image is inferior to the sum of the sizes of $\mathfrak{t}$ and T. The growth of prefix feature trees with each inverse image is crucial because, for each script trace, we compute inverse images successively through each command of the script trace.

Hence, the time complexity of computing the set of feature trees compatible with a script trace is quasiquadratic in the size of the script trace.

4.1.3 Specificity of the backward approach

A simple argument from the tree transducer literature in favor of the backward approach is the fact that top-down tree transducers do not preserve the regularity of tree languages, but their inverse images do. Indeed the inverse image through a top-down tree transducer of a regular tree language is a regular language. This is not true of the direct image of top-down tree transducers because those machines can make copies of a subtree, and it is impossible for bottom-up tree automata (the automata recognizing regular languages) to check the equality between two subtrees of a tree. Although top-down tree transducers work on ranked ordered trees and file systems are more akin to unranked unordered trees, the ability of scripts to copy a directory in a file system is an important hurdle in solving our problem. We have seen this several times in chapter 3, in particular when discussing the complexity of the algorithm of composition of pattern rewriting rules at the end of subsection 3.3.2. There we noticed an apparent disymetry when composing rewriting rules: when we compose more than two rules which copy subtrees, we create mutiple copies of a same variable in the output pattern of the rewriting rule, which greatly increases the complexity of computing the *composition on the right* of this rule but not its *composition on the left*.

One could argue that the complexity of computing the composition of rules in a forward, right-to-left way could be reduced by changing the data-structure of these rules and dynamically aggregating the constraints put on different occurrences of a same variable, but since each occurrence of a variable can be altered, by removing a subtree from it for example, we would still need to distinguish the original and altered versions of a same variable. A purely logical framework would better suit this approach. In fact the original version of the symbolic interpreter of the CoLiS tool relies on a logical framework named *feature tree logics* (presented in [19]). The complexity of this version is tied to the mutiplicity of variables, and the operation of copying subtrees is the generator of this mutiplicity of variables.

By computing inverse images instead, we exempt ourselves of variables. The absence of variable propagation allows us to represent sets of trees using an easy-to-update tree-like structure. It is easy to update in the sense that updating the representation of a set of trees by adding a constraint on the subtree at path p consists in modifying the representation only at path p. Updates to the representation of sets of trees have only *local* consequences. This representation is the model of *prefix feature trees*.

Problem with the backward approach: as stated in our description of the CoLiS toolchain in subsection 1.3.1, the concrete interpreter does not simply compute all script traces of executions of the input script to send them to the symbolic interpreter. It actually computes intermediairy traces representing partial executions of the script, these are sent to the symbolic interpreter which tells the concrete when a partial execution is already makes incoherent use of the file system. This allows the concrete interpreter to cut branches of executions earlier. This is especially useful in loops where, without information from the symbolic interpreter, the concrete interpreter would anticipate more iterations of loops, even if their use of the file system should prevent more iterations. This is efficient when the symbolic interpreter checks script traces incrementally from left to right (from first command to last command), because computations of partial script traces are used to compute the script traces of which they are a prefix. This is exactly how the forward approach works.

In our case however, if the symbolic interpreter is called on script traces of partial executions of scripts, the result cannot easily be used to compute script traces of which it is a prefix. If we check each partial script trace sent by the concrete interpreter we have to compute a lot of different script traces, but if we only check complete script traces we might let the concrete interpreter go down possibly big branches of execution which also means more traces for the symbolic interpreter to check. There are several ways to remedy this problem.

Solution 1 In our implementation we opted for a simple compromise. We only check a portion of partial script traces, specifically only partial traces whose length is a multiple of a fixed integer n. In the case scenario where no branches of executions are cut, this divides the computation time by about n (assuming n is small compared to the length of complete script traces). In the case scenario where a branch can be cut, we do not cut the branch as soon as the execution of the script is impossible, but we cut it at most $n-1$ steps after that point. We have experimented with different values for n, we found the best balance, between checking too many traces and not cutting dead branches early enough, at $n = 5$. This value corresponds to the number of commands we add to a script trace before checking whether it represents a valid execution of the script.

A finer version of this first solution consists in analyzing the script to predict the shape of the tree of executions, which allows us to find the points in the tree where cutting branches would be most efficient.

Solution 2 We could also modify the symbolic interpreter so as to be able to use computations of partial script traces to check longer script traces. A naive way to do this would be to use the model of tree pattern transducers from chapter 3 to model script traces. We would then use the composition procedure to compute script traces recursively from left to right (as in the forward approach). It might be possible to represent script traces in a more concise way, allowing us to compute domains more easily.

Solution 3 We could code a new concreter interpreter which computes execution scenarios starting from the end. The behaviour of control structures like `if` and `while` can be expressed in a backward way. This would lead to represent executions scenarios as paths in a tree, but where the root of the tree is the last command of a script and leaves are the first command. Although this approach would allow us to easily cut branches of iterations of loops while keeping the symbolic interpreter efficient, it would not work on some scripts because of the use of variables. Some scripts define or update variables in `if` statements for example (some even define different versions of a function in an `if` statement), which creates two very different branches of execution, but the choice between the two can be dependent on the state of the file system at the start of the script.

Another problem with the backward approach for finding errors is that when detecting errors in a script, it is important to detect which precise command produces the error. In terms of script traces this is equivalent to finding the smallest *prefix* of a trace which makes it incompatible with configurations of file systems. Because our approach computes recursively backwards in script traces, it can more easily find the smallest suffix, which corresponds finding to the last command which makes a script fail.

However, this problem can be considered minor because the concrete interpreter gives reports on errors which are precise enough that points of failure are easy to spot.

4.2 Practical results

We have implemented the improved algorithm computing the domain of script traces as part of the CoLiS project. The code of the pre-existing CoLiS tool (presented in subsection 1.3.1) was divided into a concrete interpreter computing script traces and a symbolic interpreter computing the domains of these traces. We programmed an alternative version of this symbolic interpreter. We refer to our version as the transducers version, and the other version as the constraints version. We used the same programming language, OCaml, as that of the existing CoLiS tool in order to make interfacing easier.

The full version of our tool is available on a git repository at `https://gitlab.inria.fr/ssalvati/transducers-for-colis.git`, because we reuse the concrete interpreter programmed by our collaborators, you will need an invitation in order to access it (rapporteurs already have an invitation, people that are intersted can contact me by email). It contains the concrete interpreter and both versions of the symbolic interpreter. Our contribution, the symbolic interpreter based on computing inverse images of sets of trees, represents about 1200 lines of code.

It is important to note that our implementation is based on the version of the CoLiS toolchain dating back from March 2020. The toolchain has been updated since, most notably the concrete interpreter can now parse and execute more constructions of maintainer scripts.

Experimentation: We have tested our implementation on Debian maintainer scripts and compared our results with those of the constraints version. We used 4 Intel Core i3-8130U CPU @ 2.20GHz with 16GB of RAM.

Running on a corpus of 28,815 Debian maintainer scripts, from 12,592 packages, extracted from a *sid* Debian distribution, each version of the tool (transducers and constraints version) runs in about 4 minutes.

```
1 #!/bin/sh -eu
2 # Disable the OLD amd package (reinstated in postrm)
3 if test -f /etc/amd/config
4 then
5     echo "Disabling old amd package..."
6     mv -f /etc/amd/config /etc/amd/config.disabled-by-am-utils
7 fi
```

Figure 4.6 – Script `preinst` from package `am-utils_6.2+rc20110530-3.2+b1`

The concrete interpreter runs only partially or fails on 20,739 scripts (72%) and completely runs on 8076 (28%). Both versions of the symbolic interpreter were run on those 8076 scripts and we analyzed some of their reports.

Over those 8076 scripts the constaints version finds at least one failing execution scenario in 1672 scripts (21%), while the transducer version (ours) finds 1711 scripts (21%). The 1711 scripts found by our version include the 1672 scripts found by the constraints version, and 39 other scripts. It is hard to automatically search for the types of errors because the commands making a script fail are not always informative of the type of errors. Instead we had to analyse scripts by hand to find where errors came from. We have done this on a sample of them.

Of the 1672 scripts found by both versions, a big proportion seem to come from the concrete interpreter. For example some scripts do different things depending on their argument using `case "$1" in`, and produce an error when the argument is the empty string.

The remaining 39 scripts detected only by our tool seem to all contain a genuine bug. Some errors come from the use of `cp` or `mv`, which indicates a difference of modelisation of these commands between the two versions.

One such example is the script `preinst` from package `am-utils_6.2+rc20110530-3.2+b1` shown in Figure 4.6, which fails when `/etc/amd/config` is a regular file (so the `test` line 3 succeeds) and `/etc/amd/config.disabled-by-am-utils/config` is a directory (so the `mv` fails). This bug is interesting because it is not too obvious. The condition on the `test` is supposed to make sure that the `mv` is possible, it does check that `/etc/amd/config` is a regular file and `/etc/amd` is a directory, but it forgets to check that the new filename `config.disabled-by-am-utils` is not already used.

Overall this shows that our implementation has similar complexity as that of another implementation solving the same problem, despite the fact that we reuse a part of their code (the concrete interpreter) which was better suited to their approach of the problem. Our implementation provides similar results, but our analysis gives slightly finer results and detects a few more bugs.

In the future we hope to update our implementation to the latest version of the concrete interpreter published in [19], that version supports a greater variety of script commands and constructs.

4.3 Equivalence

On our model of tree pattern transformations we have shown how to compute composition and how to decide whether a transducer represents the empty transformation (i.e. testing if its domain is empty). Testing the equivalence of two transducers is another important verification tool. We have not implemented it, but we can use tools from section 3.3 to give an algorithm for it.

For two tree pattern transducers to be equivalent we need to check three things: first that the input constraints are equivalent (required because the domain of transducers is characterized by them). Second checking that the rewriting rules are equivalent on the domain given by the input constraints, this can be done by computing the most general unifier of the patterns of the rule and checking that the input patterns, through unification, stay compatible with the input constraints. Thirdly we need to check that the output constraints are equivalent. This is enough to test equivalence.

For testing the equivalence of two tree constraints C_1 and C_2, we can test that both $C_1 \wedge \neg C_2$ and $\neg C_1 \wedge C_2$ are unsatisfiable, using the test of satifiability from subsection 3.3.3. Although we suspect that there is an algorithm with better formal complexity. We could also use prefix feature trees (c.f. subsection 4.1.2) to represent sets of feature trees instead of tree constraints, this could improve the average time complexity by removing some disjunctions.

4.4 Conclusion on CoLiS

In this part of the thesis we have presented our work in the scope of the CoLiS ANR project. We have built a formalism of tree transformations able to represent the actions of some script commands on file systems with carefully chosen approximations. We have developed algorithms to efficiently verify their properties, including computing the composition of transformations, testing if a transformation is empty and testing if two transformations are equivalent.

Inspired by the literature of tree transducers, we have designed an algorithm to automatically check for errors in scripts. We have tailored our model and algorithms to the specific problem of checking script traces, and implemented our solution. We have tested this solution and compared it to an existing tool, showing that our approach allowed a slightly more accurate analysis while keeping a similar time complexity.

Finally we have studied the complexities of our algorithms. We have proven the NP-hardness of the problem of checking script traces, and precisely identified the mechanisms inducing this complexity. We have then shown how, in the specific case where those mechanisms are absent, our algorithm has polynomial time complexity.

After this, our most immediate prospect is to adapt our implementation to the latest version of the concrete interpreter which supports a lot more functionalities of scripts than the one we based our work on. This new version is able to not only run maintainer scripts, but it also simulates the combinations of maintainer scripts run by package managers when performing tasks of installation and removal of packages.

Since our implementation is fast enough to run on large corpuses of scripts, we could also try to model file systems and commands more accurately. In particular removing the big approximation of the `cp -r` command on the case where its destination directory is not empty.

A third possibility would be to use a similar tool in order to detect if an uninstallation script puts the file system back as it was before installation. To check this we only need to test if the composition the installation script with the uninstallation script performs the identity function. This would require to implement our algorithms for composition and for checking the equivalence.

Part III

Part 3 : Transduction through functional programming

Chapter 5

Definition of high-order tree transducers

In this part of the thesis we present the model of High-Order Deterministic tree Transducers (HODT) which gives a uniform generalisation of the models of tree transformations presented in section 1.4 of the introduction. They are defined similarly to top-down tree transducers, except that rules can produce λ-terms of arbitrary types.

This class of transducers naturally contains top-down tree transducers, as they are HODT of order 0 (the output of rules are trees), but also MTT, which are HODT of order 1 (outputs are tree contexts).

The *linear* and *almost linear* restrictions of HODT are of special interest to us. In terms of expressiveness, linear HODTR ($\mathrm{HODTR}_{\mathrm{lin}}$) corresponds to the class of MSOT. This links our formalism to other equivalent classes of transducers, such as finite-copying macro-tree transducers [12, 13], with an important difference: the linearity restriction is a simple syntactic restriction, whereas finite-copying, or the equivalent single-use-restricted condition are both global conditions that are harder to enforce. For STT, the linearity condition corresponds to the copyless condition described in [2] and where the authors prove that any STT can be made copyless.

The relationship of $\mathrm{HODTR}_{\mathrm{lin}}$ to MSOT is made via a transformation that *reduces the order* of transducers. We indeed prove that for any $\mathrm{HODTR}_{\mathrm{lin}}$, there exists an equivalent $\mathrm{HODTR}_{\mathrm{lin}}$ whose order is at most 3. This transformation allows us to prove then that $\mathrm{HODTR}_{\mathrm{lin}}$ are equivalent to Attribute Tree Transducers with the single use restriction ($\mathrm{ATT}_{\mathrm{sur}}$). In turn, this shows that $\mathrm{HODTR}_{\mathrm{lin}}$ are equivalent to MSOT [7].

One of the main interests of $\mathrm{HODTR}_{\mathrm{lin}}$ is that λ-calculus also offers a simple composition algorithm. This approach gives an efficient procedure for composing two $\mathrm{HODTR}_{\mathrm{lin}}$. In general, this procedure raises the order of the produced transducer. In comparison, composition in other equivalent classes are either complex or indirect (through MSOT).

The last two results allow us to obtain a composition algorithm for other equivalent classes of tree transducer, such as MTT or STT: compile into $\mathrm{HODTR}_{\mathrm{lin}}$, compose, reduce the order, and compile back into the original model. The advantage of this approach over the existing ones is that the complex composition procedure is decomposed into two simpler steps (the back and forth translations between the formalisms are unsurprising technical procedures). We believe in fact that existing approaches [16, 2] combine in one step the two elements, which is what makes them more complex.

The property of order reduction also applies to a wider class of HODT, *almost linear* HODT ($\mathrm{HODTR}_{\mathrm{al}}$). Again here, this transformation allows us to prove that this class of tree transformations is equivalent to that of Attribute Tree Transducers which is known to be equivalent to MSO tree transformations with unfolding [7], i.e. MSO tree transduction that produce Directed Acyclic Graphs (i.e. trees with shared sub-trees) that are unfolded to produce a resulting tree. We call these transductions Monadic Second Order Transductions with Sharing (MSOTS). Note however that $\mathrm{HODTR}_{\mathrm{al}}$ are not closed under composition.

Structure of part III

In chapter 5 we present the model of High-Order Deterministic tree Transducers (HODT) and discuss some of its important properties.
In chapter 6 we give two algorithms for computing the composition of HODT, one of which preserves linearity. We also discuss the complexities of these algorithms.
In chapter 7 we present the procedure used to reduce the order of $\mathrm{HODTR}_{\mathrm{lin}}$ and $\mathrm{HODTR}_{\mathrm{al}}$, and we show that these models are respectively equivalent to MSOT and MSOTS.

Structure of chapter 5

In section 5.1 we recall some notions of λ-calculus and we define the model HODT and its variants.
In section 5.2 we give an example of HODT and how to compute its output.
In section 5.3 we prove that domains of $\mathrm{HODTR}_{\mathrm{lin}}$ and $\mathrm{HODTR}_{\mathrm{al}}$ are regular sets of trees, we give a decision process for the regular type-checking in those models. We also show that the look-ahead does not increase the expressivity of the model HODT, and we discuss the how the order of λ-terms in HODT increase its expressivity.

5.1 Definitions

In this section we present notions of λ-calculus, then we define the transducer model of High-Order Deterministic top-down tree Transducers, its variant with regular look-ahead, and the linear and almost-linear variants.

5.1.1 Simply-typed lambda calculus

We explain here how to represent trees and functions on trees using simply-typed λ-calculus, and we present the definitions of linear and almost linear λ-terms.

Fix a finite set of atomic types $\mathcal{A}$, we then define the set of types over $\mathcal{A}$, $\mathrm{types}(\mathcal{A})$, as the types that are either an atomic type, i.e. an element of $\mathcal{A}$, or a functional type $(A \to B)$, with A and B being in $\mathrm{types}(\mathcal{A})$. The operator $\to$ is right-associative and $A_1 \to \cdots \to A_n \to B$ denotes the type $(A_1 \to (\cdots \to (A_n \to B)\cdots))$. The order of a type A is inductively defined by $\mathrm{order}(A) = 0$ when $A \in \mathcal{A}$, and $\mathrm{order}(A \to B) = \max(\mathrm{order}(A) + 1, \mathrm{order}(B))$.

Signatures A signature Σ is a triple $(C, \mathcal{A}, \tau)$ with C being a finite set of *constants*, $\mathcal{A}$ a finite set of *atomic types*, and τ a mapping from C to types$(\mathcal{A})$, *the typing function*. We allow ourselves to write types(Σ) to refer to the set types$(\mathcal{A})$.

The order of a signature is the maximal order of a type assigned to a constant (i.e. $\max\{\text{order}(\tau(c)) \mid c \in C\}$). A *tree signature* is a signature of order 1 with a unique atomic type. In this work, we mostly deal with tree signatures. In a tree signature with atomic type o, the types of constants are of the form $o \to \cdots \to o \to o$. We write $o^n \to o$ for an order-1 type which uses $n+1$ occurrences of o, for example, $o^2 \to o$ denotes $o \to o \to o$. When c is a constant of type A, we may write c^A to make explicit that c has type A. Two signatures $\Sigma_1 = (C_1, \mathcal{A}_1, \tau_1)$ and $\Sigma_2 = (C_2, \mathcal{A}_2, \tau_2)$ can be summed if for every c in $C_1 \cap C_2$ we have $\tau_1(c) = \tau_2(c)$; in such a case we write $\Sigma_1 + \Sigma_2$ for the signature $(C_1 \cup C_2, \mathcal{A}_1 \cup \mathcal{A}_2, \tau)$ so that if c is in C_1, $\tau(c) = \tau_1(c)$ and if c is in C_2, $\tau(c) = \tau_2(c)$. The sum operation over signatures being associative and commutative, we write $\Sigma_1 + \cdots + \Sigma_n$ to denote the sum of several signatures.

We assume that for every type A, there is an infinite countable set of variables of type A. When two types are different the set of variables of those types are of course disjoint. As with constants, we may write x^A to make it clear that variable x is of type A.

λ-Terms When Σ is a signature, we define the family of simply typed λ-terms over Σ, denoted $\Lambda(\Sigma) = (\Lambda^A(\Sigma))_{A \in \text{types}(\Sigma)}$, as the smallest family indexed by types(Σ) so that:

- if c^A is a constant in Σ, then c^A is in $\Lambda^A(\Sigma)$,
- any variable x^A is in $\Lambda^A(\Sigma)$,
- if $A = B \to C$ and M is in $\Lambda^C(\Sigma)$, then $(\lambda x^B.M)$ is in $\Lambda^A(\Sigma)$,
- if M is in $\Lambda^{B \to A}(\Sigma)$ and N is in $\Lambda^B(\Sigma)$, then (MN) is in $\Lambda^A(\Sigma)$.

The term M is a *pure* λ-term if it does not contain any constant c^A from Σ. When the type is irrelevant we write $M \in \Lambda(\Sigma)$ instead of $M \in \Lambda^A(\Sigma)$. We drop parentheses when it does not bring ambiguity. In particular, we write $\lambda x_1 \dots x_n.M$ for $(\lambda x_1(\dots(\lambda x_n.M)\dots))$, and $M_0 M_1 \dots M_n$ for $((\dots(M_0 M_1)\dots)M_n)$.

The set $\text{fv}(M)$ of free variables of a term M is inductively defined on the structure of M:

- $\text{fv}(c) = \emptyset$,
- $\text{fv}(x) = \{x\}$,
- $\text{fv}(MN) = \text{fv}(M) \cup \text{fv}(N)$,
- $\text{fv}(\lambda x.M) = \text{fv}(M) \setminus \{x\}$.

Terms which have no free variables are called *closed*. We write $M[x_1, \dots, x_k]$ to emphasize that $\text{fv}(M)$ is included in $\{x_1, \dots, x_k\}$. When doing so, we write $M[N_1, \dots, N_k]$ for the capture avoiding substitution of variables x_1, …, x_k by the terms N_1, …, N_k. In other contexts, we simply use the usual notation $M[x_1/N_1, \ldots, x_k/N_k]$. Moreover given a substitution θ, we write $M.\theta$ for the result of applying this (capture avoiding) substitution and we write $\theta[x_1/N_1, \ldots, x_k/N_k]$ for the substitution that maps the variables x_i to the terms N_i but is otherwise equal to θ. Of course, we authorize such substitutions only when the λ-term N_i has the same type as the variable x_i.

Reduction A *β-redex* is a term of the form $((\lambda x.M)N)$ and its *β-contractum* is $M[x/N]$. A term M *β-contracts* to M' when one of its subterm is a β-redex and is replaced by its β-contractum in M'. We write $M \rightarrow_\beta M'$ when M β-contracts to M'. The reflexive transitive closure of β-contraction is β-reduction and is written $\overset{*}{\rightarrow}_\beta$ and its symmetric reflexive and transitive closure, β-conversion, is written $=_\beta$. A term that does not contain a β-redex is said in β-normal form.

An η-redex is a term of the form $(\lambda x.(M\,x))$ when $x \notin \text{fv}(M)$ and its *η-contractum* is the term M. The relations of η-contraction, $\rightarrow_\eta$, η-reduction, $\overset{*}{\rightarrow}_\eta$, and η-conversion, $=_\eta$, are defined similarly to β-contraction. So as to compare λ-terms, we use the union of β-contraction and η-contration, $\rightarrow_{\beta\eta}$. But this can be done by putting terms in a particular form: the *η-long form*. A term M is said to be in η-long form whenever if N is a subterm of M that has type $A \rightarrow B$ then either N is of the form $\lambda x.N'$, or its occurrence in M is applied to some argument. For every term M there is a term M' in η-long form such that $M =_\eta M'$ and moreover $M =_{\beta\eta} N$ iff given M' and N' that are η-long forms of M and N, $M' =_\beta N'$. From now on, we are always going to work with terms in η-long form.

Consider closed terms of type o that are in β-normal form and that are built on a tree signature, they can only be of the form $a\,t_1 \ldots t_n$ where a is a constant of type $o^n \rightarrow o$ and $t_1, \ldots, t_n$ are closed terms of type o in β-normal form. This is just another notation for ranked trees. So when the type o is meant to represent trees, types of order 1 which have the form $o \rightarrow \cdots \rightarrow o \rightarrow o$ represent functions from trees to trees, or more precisely tree contexts. Types of order 2 are types of trees parametrized by contexts. The notion of order captures the complexity of the operations that terms of a certain type describe.

Linearity and almost linearity A term M is said *linear* if each variable (either bound or free) in M occurs exactly once in M. A term M is said *syntactically almost linear* when each variable in M of non-atomic type occurs exactly once in M. Note that, through β-reduction, linearity is preserved but not syntactic almost linearity.

For example, given a tree signature Σ_1 with one atomic type o and two constants f of type $o^2 \rightarrow o$ and a of type o, the term $M = (\lambda y_1 y_2.f\,y_1\,(f\,a\,y_2))\,a\,(f\,x\,a)$ with free variable x of type o is linear because each variable (y_1, y_2 and x) occurs exactly once in M. The term M contains a β-redex so: $(\lambda y_1 y_2.f\,y_1\,(f\,a\,y_2))\,a\,(f\,x\,a) \rightarrow_\beta (\lambda y_2.f\,a\,(f\,a\,y_2))\,(f\,x\,a) \rightarrow_\beta f\,a\,(f\,a\,(f\,x\,a))$. The term $f\,a\,(f\,a\,(f\,x\,a))$ has no β-redex so it is the β-normal form of M.

Another example: the term $M_2 = (\lambda y.f\,y\,y)\,(x\,a)$ with free variable x of type $o \rightarrow o$ is syntactically almost linear because the variable y which occurs twice in the term is of the atomic type o. It β-reduces to the term $M_2' = f\,(x\,a)\,(x\,a)$ which is not syntactically almost linear, so β-reduction does not preserve syntactical almost linearity.

We call a term *almost linear* when it is β-convertible to a syntactically almost linear term. Almost linear terms are characterized also by typing properties (see [22]).

For any signature $\Sigma = (C, \{o\}, \tau)$ with a single atomic type o and for any type A on this signature, we define a new symbol Ω_A which will serve as an *undetermined* object. We note Σ^Ω the signature obtained this way. Intuitively, it represent the result of a computation which does not produce an output. For example, if there is no transition rule with a state q and a tree $a\,x_1 \dots x_n$, then we can add a rule $q(a\,x_1 \dots x_n) \to \Omega_A$; this allows us to have a complete set of transition rules, while still being a generalisation of other transduction models (such as DTOP and MTT) that do not necessarily produce an output for every input.

5.1.2 High-Order Deterministic top-down tree Transducers

From now on we assume that Σ_i is a tree signature for every number i and that its atomic type is o_i.

A High-Order Deterministic top-down tree Transducer (HODT) T is a tuple $(\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R)$ where:

- $\Sigma_1 = (C_1, \{o_1\}, \tau_1)$ is a first-order tree signature, the *input signature*,
- $\Sigma_2 = (C_2, \{o_2\}, \tau_2)$ is a first-order tree signature, the *output signature*,
- $\Sigma_Q = (Q, \{o_1, o_2\}, \tau_s)$ is the *state signature*, and is so that for every $q \in Q$, $\tau_s(q)$ is of the form $o_1 \to A_q$ where A_q is in types(Σ_2). Constants of Q are called *states*,
- q_0 is the *initial state*,
- R is a finite set of rules of the form

$$q(a\,\overrightarrow{x}) \to M[x_1, \dots, x_n]$$

where:

  - q is a state of Q,
  - a is a constant of Σ_1 with type $o_1^n \to o_1$,
  - $\overrightarrow{x} = x_1, \dots, x_n$ are variables of type o_1 representing the child trees of the root labeled a,
  - M is a term of type A_q that is built on the signature $\Sigma_2^\Omega + \Sigma_Q$,
  - there is exactly one rule in R for each possible left-hand side (determinism).

Notice that we have given states a type of the form $o_1 \to A$ where $A \in$ types(o_2). The reason why we do this is to have a uniform notation. Indeed, a state q is meant to transform, thanks to the rules in R, a tree built in Σ_1 into a λ-term built on Σ_2^Ω with type A_q. So we simply write $q\,M\,N_1 \dots N_n$ when we want to transform M with the state q and pass $N_1, \dots, N_n$ as arguments to the result of the transformation. We write Σ_T for the signature $\Sigma_1 + \Sigma_2 + \Sigma_Q$ (assuming that the constants in Σ_Q, Σ_1 and Σ_2 are distinct from each other). Notice also that the right-hand part of a rule is a term that is built only with constants of Σ_2^Ω, states from Σ_Q and variables of type o_1. Thus, in order for this term to have a type in types(Σ_2), it is necessary that the variables of type o_1 only occur as the first argument of a state in Σ_Q.

Remark that we did not put any requirement on the type of the initial state. So as to restrict our attention to transducers as they are usually understood, it suffices to add the requirement that the initial state is of type $o_1 \to o_2$. However, we consider as well that transducers may produce *programs* instead of first order terms.

The constant Ω

We use the constant Ω_{o_2} of Σ_2^Ω so as to denote undefined outputs. Because of this, we impose that for every pair q, a, there is a rule of the form $q(a\, x_1 \dots x_n) \to_T M$, as if there were no such a rule, we would simply add the rule $q(a\, x_1 \dots x_n) \to_T \Omega_{A_q}$ without changing the behavior of the transducer. Note that this technical choice is not standard in the definitions of transducers where failure is more often represented by blocked computations. This technical choice makes several properties like removal of look-ahead and composition work while they would not if we were sticking to the more traditional definitions.

It is worth noting that if we were implementing a transducer in a programming language we would need to specify what to do when a computation is blocked and that this would amount to signal failure by some mechanism (abort computation, throw an exception,...), thus marking failure with some constant seems to be closer to actual implementations of transducers.

We will later show in subsection 5.3.2 that the addition of a look-ahead to this model allows us to represent *partial* tree-to-tree functions without resorting to the constant Ω, but while still keeping our result on composition of transducers.

5.1.3 Regular look-ahead

Similarly to known models of transducers (like DTOP in section 1.4) we equip HODT with what we call a *regular look-ahead*. It is a deterministic bottom-up tree automaton which is run on the input tree of a High-Order Deterministic top-down tree Transducer (HODT) in order to guide its rules. We first restate the definition of bottom-up tree automata, but with the λ-calculus representation of trees.

Tree automata

A BOT $\mathtt{A}$ is a tuple (Σ_L, Σ, R) where:

- $\Sigma = (C, \{o\}, \tau)$ is a first-order tree signature, the *input signature*,
- $\Sigma_L = (L, \{o\}, \tau_L)$ is the *state signature*, and is such that for every $\ell \in L$, $\tau_L(\ell) = o$. Constants of L are called *states*,
- R is a finite set of rules of the form $a\, \ell_1 \dots \ell_n \to \ell$ where:

- ℓ,$\ell_1, \ldots, \ell_n$ are states of L,
- a is a constant of Σ with type $o^n \to o$.

An automaton is said *deterministic* when there is at most one rule in R for each possible left hand side. It is *non-deterministic* otherwise.

Apart from the notation, our definition differs from the classical one by the fact there are no final states, and hence, the automaton does not describe a language. This is due to the fact that BOT will be used here purely for look-ahead purposes.

A High-Order Deterministic top-down tree Transducer with Regular look-ahead (HODTR) T is a tuple $(\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R, \mathtt{A})$ where Σ_1, Σ_2, Σ_Q and q_0 are as before and:

- $\mathtt{A}$ is a deterministic bottom-up tree automaton, we call it the *look-ahead* automaton of T,
- R is a finite set of rules of the form

  $$q(a\overrightarrow{x})\langle\overrightarrow{\ell}\rangle \to M[x_1, \ldots, x_n]$$

  where:

  - q is a state of Σ_Q,
  - a is a constant of Σ_1 with type $o_1^n \to o_1$,
  - $\overrightarrow{x} = x_1, \ldots, x_n$ are variables of type o_1 representing the child trees of the root labeled a,
  - $\overrightarrow{\ell} = \ell_1, \ldots, \ell_n$ are in L (the set of states of the look-ahead $\mathtt{A}$),
  - M is a term of type A_q that is built on $\Sigma_2^\Omega + \Sigma_Q$.
  - there is exactly one rule in R for each possible left-hand side (determinism).

5.1.4 Linear and Almost-Linear variants

A linear High-Order Deterministic top-down tree Transducer with Regular look-ahead (HODTR$_{\text{lin}}$) is a HODTR where the terms on the right side of rules are linear.

An almost-linear High-Order Deterministic top-down tree Transducer with Regular look-ahead (HODTR$_{\text{al}}$) is a HODTR where the terms on the right side of rules are almost-linear.

The rules of such transducers are of the form

$$q(a\overrightarrow{x})\langle\overrightarrow{\ell}\rangle \to M[x_1, \ldots, x_n]$$

where $\overrightarrow{x} = x_1, \ldots, x_n$ are variables of type o_1 and $\overrightarrow{\ell} = \ell_1, \ldots, \ell_n$ are states of the look-ahead automaton. The linearity or almost linearity constraint on M affects both bound variables and free variables $x_1, \ldots, x_n$, meaning that all of the subtrees $x_1, \ldots, x_n$ are used in computing the output.

That will be important for the composition of two transducers because if the first transducer fails in a branch of its input tree then the second transducer, applied to that tree, must fail too.

This restriction forcing the use of input subtrees does not reduce the model's expressivity because we can always add a state q which visits the subtree but only produces the identity function on type o_2 (this state then has type $o_1 \to A_q = o_1 \to o_2 \to o_2$).

Weakly Deterministic variant

A linear High-Order Weakly Deterministic top-down tree Transducer with Regular look-ahead ($\mathrm{HOWDTR}_{\mathrm{lin}}$) is a $\mathrm{HODTR}_{\mathrm{lin}}$ whose look-ahead automaton is not necessarily deterministic, but the transducer is deterministic in the weaker sense that, when two rules of the transducer are of the form:

$$\begin{aligned} & q(a\,x_1 \dots x_n)\langle \ell_1, \dots, \ell_n\rangle \to_T M[x_1, \dots, x_n] \\ \text{and} \quad & \\ & q(a\,x_1 \dots x_n)\langle \ell'_1, \dots, \ell'_n\rangle \to_T M'[x_1, \dots, x_n] \end{aligned}$$

there must be some i such that no tree is recognized by both states ℓ_i and ℓ'_i of the look-ahead automaton.

This alteration of the model of $\mathrm{HODTR}_{\mathrm{lin}}$ is useful for the composition of transducers. Indeed, when composing $\mathrm{HODTR}_{\mathrm{lin}}$, we will have to determinize the look-ahead automaton so as to obtain a $\mathrm{HODTR}_{\mathrm{lin}}$, which may cause an exponential blow-up of the look-ahead automaton. However if we keep the look-ahead non-deterministic, the transducer stays deterministic because only one rule of the transducer can apply when it is actually run.

Notice that it suffices to determinize the look-ahead so as to obtain a $\mathrm{HODTR}_{\mathrm{lin}}$ from a $\mathrm{HOWDTR}_{\mathrm{lin}}$, and therefore the two models are equally expressive.

5.1.5 Tree transformations associated with transducers

Given a HODT, HODTR or $\mathrm{HOWDTR}_{\mathrm{lin}}$ T, we write $T :: \Sigma_1 \longrightarrow \Sigma_2$ to mean that the input signature of T is Σ_1 and its output signature is Σ_2.

A transducer T induces a notion of reduction on terms. A T-redex is a term of the form $q(a\ M_1 \dots M_n)$ and if

$$q(a\ x_1 \dots x_n) \to_T M[x_1, \dots, x_n]$$

is a rule of T, then its T-contractum is $M[M_1, \dots, M_n]$. In the case we deal with a transducer with look-ahead, $q(a\,M_1 \dots M_n)$ is a contractum for the rule

$$q(a\ x_1 \dots x_n)\langle \ell_1, \dots, \ell_n\rangle \to_T M[x_1, \dots, x_n]$$

only if (the β-normal forms of) $M_1, \dots, M_n$ are respectively accepted by $\mathtt{A}$ from the states $\ell_1, \dots, \ell_n$. In that case, a T-contractum of $q(a\ M_1 \dots M_n)$ is $M[M_1, \dots, M_n]$. Notice that T-contracta are typed terms and that they have the same type as their corresponding T-redices. For technical reasons, we also assume that $q\,\Omega \to_T \Omega_{A_q}$. The relation of T-contraction relates a term M and a term M' when M' is obtained from M by replacing one of its T-redex with a corresponding T-contractum.

We write $M \to_T M'$ when M T-contracts to M', and we write $\to_{\beta,T}$ the union of the relations $\to_T$ and $\to_\beta$. The relation of β-reduction is confluent, and so is the relation of T-reduction as transducers are deterministic, therefore the union of the two relations is terminating. It is not hard to prove that $\to_{\beta,T}$ is also locally confluent. It follows that it is confluent and strongly normalizing. The symmetric reflexive and transitive closure of $\to_{\beta,T}$ is written $=_{\beta,T}$. Given a term M built on Σ_T, we write $|M|_T$ to denote its normal form modulo $=_{\beta,T}$.

Then we write rel(T) for the relation

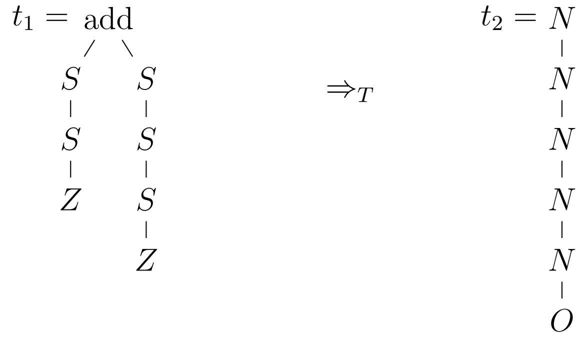
$$\{(M, |q_0 M|_T) \mid \text{M is a closed term of type } o_1 \text{ and } |q_0 M|_T \in \Lambda(\Sigma_2)\}.$$

Given a finite set of trees S_1 on Σ_1 and S_2 included in $\Lambda^{A_{q_0}}$, we respectively write $T(S_1)$ and $T^{-1}(S_2)$ for the image of S_1 by T and the inverse image of S_2 by T. As we explained earlier, we consider that the output of T is valid if its normal form modulo $=_{\beta,T}$ does not contain occurrences of Ω. Note however that the computation of T may succeed even though during the reduction some Ω occurs. Indeed this occurrence of Ω may be deleted afterwards during the computation. When using a head-reduction strategy, i.e. only reducing redices that are top-most in terms, once an Ω occurs, it will never be deleted. This reduction strategy thus detects error only when they occur and also allows for computing normal forms of transducers. Such a strategy can be implemented with a lazy evaluation. Thus this notion of transducer could be directly implemented as Haskell programs.

5.2 Example of high-order tree transducers

We give an example of a HODTR$_{\text{lin}}$ T that computes the result of additions of numeric expressions (numbers being represented in unary notation). For this we use an input tree signature with type o_1, and constants Z^{o_1}, $S^{o_1 \to o_1}$ and add$^{o_1 \to o_1 \to o_1}$ which respectively denote zero, the successor function and addition. The output signature is similar but different to avoid confusion: it uses the type o_2 and constants O^{o_2}, $N^{o_2 \to o_2}$ which respectively denote zero and successor.

For example T associates with the input tree t_1 the output tree t_2 as follows:



We do not really need the look-ahead automaton for this computation, so we omit it for this example. We could have a blank look-ahead automaton A with one state ℓ and rules: $\mathtt{A}(Z) = \ell$, $\mathtt{A}(S\,\ell) = \ell$ and $\mathtt{A}(\text{add}\,\ell\,\ell) = \ell$; which would not change the result of the transducer.

The transducer T has two states: q_0 of type $o_1 \to o_2$ (the initial state), and q_i of type $o_1 \to o_2 \to o_2$. The rules of the transducer are the following:

- $q_0(Z) \to O$,
- $q_0(S\,x) \to N(q_i\,x\,O)$,
- $q_0(\text{add}\,x\,y) \to q_i\,x\,(q_i\,y\,O)$,
- $q_i(Z) \to \lambda x.x$,
- $q_i(S\,x) \to \lambda y.N(q_i\,x\,y)$,
- $q_i(\text{add}\,x\,y) \to \lambda z.q_i\,x\,(q_i\,y\,z)$.

The order of T is $\max\{\text{order}(A_q) \mid q \in Q\} = 1$.

We show how the rules works on an example. We use a graphical representation of terms as trees where constants of atomic type o_1 or o_2 are leaves, constants of order 1 (i.e. S, N, q_0 and q_i) are inner nodes, and subterms of order ≥ 1 in η-long form are inner nodes whose internal structure is represented in a rectangle. As an example, we perform the transduction of the tree t_1 shown before. The computation, shown in Figure 5.1 and Figure 5.2, starts by applying the initial state q_0 to t_1.

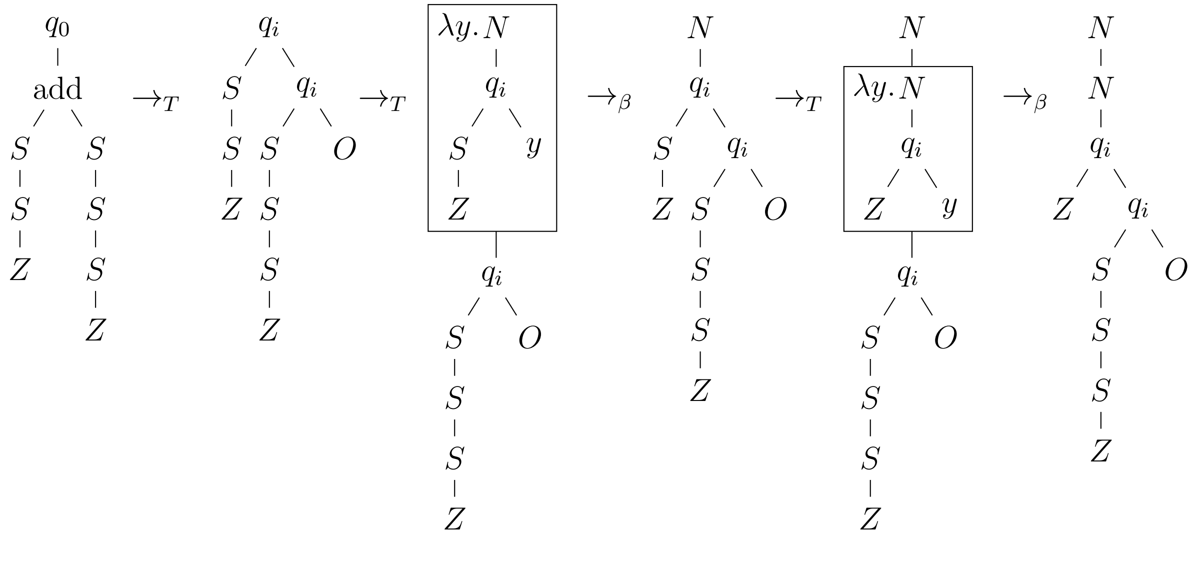


Figure 5.1 – Computation of tree t_1 (part 1)

The state q_i transforms a sequence of n symbols S into a λ-term of the form $\lambda x.N^n(x)$, and the *add* maps both its children into such terms and composes them. The state q_0 simply applies O to the resulting term.

Note that our reduction strategy here has consisted in reducing β-redices as soon as they appear, and computing T-redices only when there are no β-redices. This makes the

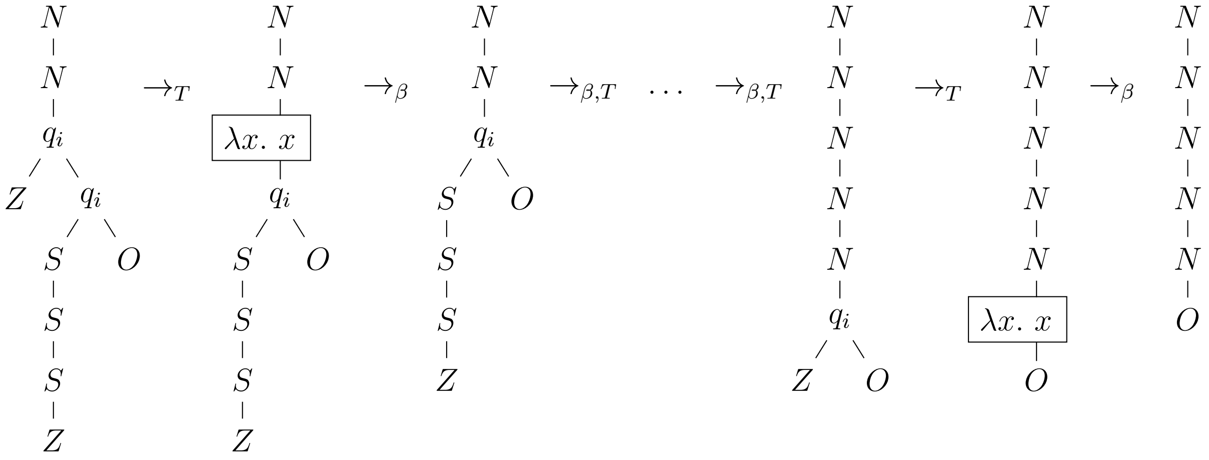


Figure 5.2 – Computation of tree t_1 (part 2)

computation simpler to present. As we mentioned earlier, any reduction strategy would lead to the same result.

5.3 Properties of High-Order Transducers

In this section we present useful properties of the different variants of High-Order Transducers.

In the first section we prove that the inverse image of a regular set of trees by a HODT is a regular set of trees, which implies that the domains of HODT are also regular sets of trees. In the second section we prove that the addition of the look-ahead does not change the expressivity of HODT. In the third section we show how HODT offer a generalisation of known classes of transducers like DTOP and MTT.

5.3.1 Domains

In this part we will show that the domain of a $\mathrm{HODTR}_{\mathrm{lin}}$, a $\mathrm{HODTR}_{\mathrm{al}}$ or a $\mathrm{HOWDTR}_{\mathrm{lin}}$ is a regular set of trees, i.e. is a set of trees recognizable by a deterministic bottom-up tree automaton. Actually, we will prove a more general claim: that the reverse image of any regular set of trees by a $\mathrm{HODTR}_{\mathrm{lin}}$, $\mathrm{HODTR}_{\mathrm{al}}$ or $\mathrm{HOWDTR}_{\mathrm{lin}}$ is a regular set of trees. This result will also give a decision process for the regular type-checking of transducers.

Inverse image preserves regularity

In order to show that the inverse image of a regular tree language by a HODT is a regular language we prove a more general result: that the inverse image of a recognizable set of λ-terms (in the sense of [26]) is a regular set of trees.

The definition of a recognizable set of λ-terms is tied to the notion of model of λ-calculus. Here we are going to use *monotone models of λ-calculus*. Fixing a signature Σ, these models are defined by means of monotone applicative structures, i.e. a family of finite meet semilattices indexed by types(Σ), $(\mathcal{M}_A)_{\text{types}(\Sigma)}$ and so that $\mathcal{M}_{A\to B}$ is the set of monotone functions from $\mathcal{M}_A$ to $\mathcal{M}_B$ i.e. those functions f from $\mathcal{M}_A$ to $\mathcal{M}_B$ so that for $a, a' \in \mathcal{M}_A$, $a \leq a'$ implies that $f(a) \leq f(a')$. $\mathcal{M}_{A\to B}$ is ordered pointwise which means that for every $f, g \in \mathcal{M}_{A\to B}$, $f \leq g$ iff for every $a \in \mathcal{M}_A$, $f(a) \leq g(a)$. We note $\bot_A$ for the minimal element of $\mathcal{M}_A$, or $\bot$ when the type is not relevant. Notice that for a signature Σ, a monotone applicative structure is completely determined by the meet semilattices to which atomic types are associated and also that for every A, $\mathcal{M}_A$ is a finite set. Finally a monotone model of Σ is a pair $\mathbb{M} = (\mathcal{M}, \rho)$ where $\mathcal{M} = (\mathcal{M}_A)_{A\in\text{types}(\Sigma)}$ is a monotone applicative structure on Σ and ρ is a mapping that associates to every c^A of Σ a value in $\mathcal{M}_A$. We naturally extend ρ to Σ^Ω by defining $\rho(\Omega_A)$ to $\bot_A$ for every atomic type A of Σ. We thus make no difference between models of Σ and models of Σ^Ω. Monotone models of Σ give an interpretation to λ-terms built in Σ (and Σ^Ω). So as to deal with free variables given a term M and *valuation* μ that maps every variable x^A to a value in $\mathcal{M}_A$ (only finitely many variables have a value different from $\bot$ and we write $\emptyset$ for the valuation that maps every variable to $\bot$), the interpretation of M with the valuation μ is written $[\![M, \mu]\!]_{\mathbb{M}}$ and is inductively defined as follows:

- $[\![c, \mu]\!]_{\mathbb{M}} = \rho(c)$,
- $[\![x, \mu]\!]_{\mathbb{M}} = \mu(x)$,
- $[\![MN, \mu]\!]_{\mathbb{M}} = [\![M,\, u]\!]([\![N, \mu]\!])$,
- $[\![\lambda x.M, \mu]\!]_{\mathbb{M}}(a) = [\![M, \mu']\!]_{\mathbb{M}}$ where μ' is the valuation that is identical to μ but that maps x to a.

We may write $[\![M, \mu]\!]$ when $\mathbb{M}$ is obvious from the context and $[\![M]\!]$ when μ is a valuation that maps every variable to $\bot$.

Theorem 4 (see [5]) *Whenever $M =_{\beta\eta} N$, we have that for every μ: $[\![M, \mu]\!]_{\mathbb{M}} = [\![N, \mu]\!]_{\mathbb{M}}$.*

A particular kind of monotone models are models induced by finite state automata. Given a tree signature Σ (with atomic type o) and a deterministic bottom-up tree automaton $\mathtt{A} = (L, \Sigma, E, F)$. We may define the monotone applicative structure $\mathcal{M} = (\mathcal{M}_A)_{A\in\text{types}(\Sigma)}$ such that $\mathcal{M}_o$ is the *flat semilattice* $L_\bot$, the element of which are $L \cup \{\bot\}$, so that $\bot \leq \ell$ for every state ℓ in L and two different states ℓ and ℓ' of L are incomparable. We now define the function $\rho :$ for each symbol a of Σ of type $o^r \to o$, and all $\ell, \ell_1, \ell_2, \ldots, \ell_r$ in $\mathcal{M}_o$:

$$\rho(a)\, \ell_1 \ldots \ell_r = \ell \text{ when } \mathtt{A} \text{ has a rule } a(\ell_1, \ldots, \ell_r) \to \ell$$

Proposition 1 (see [27]) *For every closed term M of type o, $\mathtt{A}$ accepts the normal form of M with state ℓ iff $[\![M]\!] = \ell$.*

Thus, monotone models can be seen as a nice generalization to λ-calculus of the algebraic interpretation of recognizability. In particular, fixing a monotone model $(\mathcal{M}, \rho)$, a recognizable set of λ-terms is defined by fixing a type A and a finite subset F of $\mathcal{M}_A$, the *recognizing set*. The elements of the recognizable set is then just the set of closed terms M of $\Lambda^A(\Sigma^\Omega)$ so that $[\![M]\!] \in F$. The previous proposition shows the notion of recognizability for λ-terms is a conservative extension of that of recognizability for trees.

Let's now turn to the construction of an automaton that recognizes precisely the inverse image of a recognizable set of λ-terms by a HODT. We fix a HODT $T = (\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R)$, and a recognizable set of λ-terms defined by the monotone model $(\mathcal{M}, \rho)$ of Σ and the recognizing set F included in $\mathcal{M}_{A_{q_0}}$. We now define a monotone model $\mathbb{T} = (\mathcal{T}, \theta)$ on the signature Σ_T. We define $\mathcal{T}$ so that $\mathcal{T}_{o_2} = \mathcal{M}_{o_2}$, $\mathcal{T}_{o_1} = \mathcal{M}_{A_{q_1}} \times \cdots \times \mathcal{M}_{A_{q_n}}$ when the states of T are $\{q_1, \dots, q_n\}$ and the tuples in $\mathcal{T}_{o_1}$ are ordered component-wise, i.e, $(a_1, \dots, a_n) \leq (b_1, \dots, b_n)$ when $a_1 \leq b_1$, $\dots$, $a_n \leq b_n$. We now define θ as follows:

- $\theta(a) = \rho(a)$ when a is a constant of Σ_o,
- $\theta(a)t_1 \dots t_r = (f_1, \dots, f_n)$ when a is a constant of type ${o_1}^r \to o_1$ and where $f_i = [\![M_i, \mu]\!]$ with $\mu(x_i) = t_i$ and there is a rule $q_i(ax_1 \dots x_r) \to M_i$ in R,
- $\theta(q_i)(f_1, \dots, f_n) = f_i$ for every tuple $(f_1, \dots, f_n)$ of $\mathcal{M}_{A_{q_1}} \times \cdots \times \mathcal{M}_{A_{q_n}}$.

This model defines an automaton on the input signature: its states are the elements of $\mathcal{T}_{o_1}$, its transition function is just defined by the interpretation of the constant of Σ_1 by θ. Its final states are precisely those tuples $(f_1, \dots, f_n)$ of $\mathcal{T}_{o_1}$ so that f_1 is an element of F, the recognizing set.

Theorem 5 *Given a transducer T from Σ_1 to Σ_2 and a recognizable language of λ-term $\mathcal{R}$, $T^{-1}(\mathcal{R})$ is an effectively computable regular set of trees.*

The proof of theorem 5 is based on the idea that given a term M of Σ_T, and a valuation μ, then $[\![M, \mu]\!]_{\mathbb{T}}$ is invariant modulo $=_{\beta,T}$. Thanks to Theorem 4, to prove this it suffices to prove the following lemma:

Lemma 8 Given M and N in $\Lambda(\Sigma_T)$, when $M \to_T N$, for every μ, $[\![M, \mu]\!]_{\mathbb{T}} = [\![N, \mu]\!]_{\mathbb{T}}$.

Proof
The proof is an induction on the structure of M. All the cases are straightforward except the one where $M = q_i(a\, M_1 \dots M_r)$, there is a rule of T of the form $q(a\, x_1 \dots x_n) \to P[x_1, \dots, x_r]$ and $N = P[M_1, \dots, M_r]$. By definition if

$$[\![a\, M_1 \dots M_r, \mu]\!] = (f_1, \dots, f_n) ,$$

then $[\![M, \mu]\!] = f_i$. Again by definition, we have that $f_i = [\![P, \nu]\!]$ with $\nu(x_j) = [\![M_j, \mu]\!]$ for every j in $[1, r]$. But $[\![N, \mu]\!] = [\![(\lambda x_1 \dots x_r.P)M_1 \dots M_r, \mu]\!]$ by Theorem 4. So by definition $[\![N, \mu]\!] = [\![P, \mu']\!]$ where $\mu'(x_j) = [\![M_j, \mu]\!]$ for every j in $[1, r]$. But then, as $\mathrm{fv}(P) = \{x_1, \dots, x_r\}$ we have that $[\![N, \mu]\!] = [\![P, \nu]\!] = [\![M, \mu]\!]$. □

Corollary 6 Given M and N in $\Lambda(\Sigma_T)$, when $M =_{\beta,T} N$, for every μ, $[\![M, \mu]\!]_{\mathbb{T}} = [\![N, \mu]\!]_{\mathbb{T}}$.

A direct consequence of the previous corollary is:

Proposition 2 *Given a tree t built on Σ_1, we have*

$$[\![t]\!]_{\mathbb{T}} = ([\![M_1]\!], \dots, [\![M_n]\!])$$

where $M_i = |q_i\, t|_T$ for every i in $[1, n]$.

Proof
By definition $[\![t]\!]_{\mathbb{T}} = ([\![q_1\, t]\!], \dots, [\![q_n\, t]\!])$, and, by Corollary 6, $[\![M_i]\!] = [\![q_i\, t]\!]$ for every i in $[1, n]$. □

Theorem 5 follows immediately:
Proof
Given a tree t built on Σ_1, the automaton that we have constructed simply computes $[\![t]\!]$. From Proposition 2, when $[\![t]\!] = (f_1, \dots, f_n)$, we know that $f_i = [\![|q_i\, t|_T]\!]$. Thus, $|q_1\, t|_T$ is in $\mathcal{R}$ iff f_1 is in the recognizing set defining $\mathcal{R}$, which is precisely how we have defined the finite states of the automaton. □

Type-checking A consequence of theorem 5 is the decidability of the so-called *regular type-checking* of transducers. The question is the following, given a transducer $T :: \Sigma_1 \longmapsto \Sigma_2$, a regular language of trees R_1 on Σ_1 and a recognizable set of λ-terms $\mathcal{R}_2$ on Σ_2, is $T(\mathcal{R}_1)$ included into $\mathcal{R}_2$. As $T^{-1}(\mathcal{R}_2)$ is a regular tree language, it suffices to check whether R_1 is included in $T^{-1}(\mathcal{R}_2)$.

Proposition 3 *Regular type checking is decidable for HODT.*

The same construction can be easily adapted to HODTR. But the result for HODTR can also be obtained by combining Theorem 5 and Theorem 7.

Domains Theorem 5 implies that the domains of HODT are regular sets of trees. It also shows a way to compute, for each HODT T, a bottom-up tree automaton which accepts exactly the domain of T, this automaton can be used as look-ahead in order to detect failing computations early. Therefore the constant Ω, which we use to represent failing computations, does not change the expressivity of HODTR. This is also true of HODT considering theorem 7.

5.3.2 Look-ahead

In this section, we show that the addition of a look-ahead does not enhance the expressive power of HODT. The construction we use is just a transposition of the one proposed in [14] for MTTs. For this we prove that HODT are able to compute the result of finite state tree automata.

The HODT that computes the result of finite state tree automata will have the tree signature $\Upsilon = (\emptyset, \{o\}, \emptyset)$ (i.e. the tree signature with no constant) as output signature. The idea is that it is possible to represent finite sets and functions from finite sets to finite sets in Υ and this is enough to represent the transition rules of finite state tree automata.

So given a finite set $Q = \{q_1, \dots, q_r\}$ of states, we let $\mathbb{F}_r = o^r \to o$. Now the only closed λ-terms in $\Lambda^{\mathbb{F}_r}(\Upsilon)$ are of the form $\lambda x_1 \dots x_r.x_i$ with i in $[1, r]$. There are exactly r such terms. We use a bijection ind_Q between Q and $\Lambda^{\mathbb{F}_r}(\Upsilon)$, so that $\mathsf{ind}_Q(q_i) = \lambda x_1 \dots x_r.x_i$.

The next lemma generalizes the intuition that we can represent every function from finite sets to finite sets within Υ, by showing that it is true when considering functions from Q^k to sets of λ-terms.

Lemma 9 Let f be a function from Q^k to the set of closed λ-terms in $\Lambda^A(\Upsilon)$. Then there exists a term M_f of type $\mathbb{F}_r^k \to A$ such that, for all $x_1, \dots, x_k \in Q$:

$$f(x_1, \dots, x_k) =_{\beta\eta} M_f\, \mathsf{ind}(x_1) \dots \mathsf{ind}(x_k)$$

Proof
We prove this lemma by induction on the number k.

- The initiation is when $k = 0$, then $f()$ is a λ-term N of type A, so we define $M_f = N$. In that case $f() = M_f$.

- For the induction we assume that for all function $g : Q^k \to \Lambda^A$ there exists M_g with the right property, then we prove the same for a function $f : Q^{k+1} \to \Lambda^A$.

  For all $1 \leq i \leq r$, we define the function g_i by, for all $x_1, \dots, x_k \in Q : g_i(x_1, \dots, x_k) = f(q_i, x_1, \dots, x_k)$. Then since type A is built only on atomic type o, it is necessary of the form $A = A_1 \to \cdots \to A_m \to o$.

  Finally we define :

$$\begin{aligned} M_f \quad = \quad & \lambda q\ x_1 \dots x_k\ y_1 \dots y_m. \\ & \qquad\qquad q\,(M_{g_1}\ x_1 \dots x_k\ y_1 \dots y_m) \quad \dots \quad (M_{g_r}\ x_1 \dots x_k\ y_1 \dots y_m) \end{aligned}$$

  Then we have, for all $q_i \in Q$, $M_f\ (\mathsf{ind}\ q_i) =_{\beta\eta} M_{g_i}$. So, for all $q_i, x_2, \dots, x_{k+1} \in Q$:

$$M_f\ (\mathsf{ind}\ q_i)\ (\mathsf{ind}\ x_2) \dots (\mathsf{ind}\ x_{n+1})\ y_1 \dots y_m \quad =_{\beta\eta} \quad f(q_i, x_2, \dots, x_m)\ y_1 \dots y_m$$

□

Using lemma 9, we can now show that HODT can represent the finite state automata. Given $\mathtt{A}$ a complete deterministic bottom-up finite state tree automaton with states $Q = \{q_1, \dots, q_r\}$, over the tree signature Σ, it is a function from the trees built on Σ to Q. Representing $\mathtt{A}$ with a HODT T then means that $\mathtt{A}$ maps a tree t to q_i iff T maps t to $\lambda x_1 \dots x_r.x_i$.

Lemma 10 Given a finite state automaton $\mathtt{A} = (L, \Sigma, F, R)$ where $L = \{\ell_1, \dots, \ell_r\}$, there is a HODT $T_{\mathtt{A}}$ of order 1 so that for every tree t built on Σ, $T_{\mathtt{A}}(t) = \lambda x_1 \dots x_r.x_i$ iff $\mathtt{A}(t) = \ell_i$.

Proof
Given a constant a of arity n in Σ, we define f_a to be the function so that $f_a(\ell_{j_1}, \dots, \ell_{j_n}) = \ell$ whenever $a\, \ell_{j_1} \dots \ell_{j_n} \to \ell$ is a rule in R.

Since we work with complete deterministic automata, f_a is well-defined. Moreover, when $\mathtt{A}(t_1) = \ell_{j_1}$, ..., $\mathtt{A}(t_n) = \ell_{j_n}$: $\mathtt{A}(a\, t_1 \dots t_r) = f_a(\ell_{j_1}, \dots, \ell_{j_n})$. From Lemma 9, there is a term M_a so that, for every ℓ_{j_1}, ..., ℓ_{j_n} in L with $f_a(\ell_{j_1}, \dots, \ell_{j_n}) = \ell_i$:

$$M_a(\lambda x_1 \dots x_s.x_{j_1}) \dots (\lambda x_1 \dots x_s.x_{j_n}) =_\beta \lambda x_1 \dots x_s.x_i \ .$$

We define the transducer $T_{\mathtt{A}}$ as $(\Sigma_{\tilde{q}}, \Sigma, \Upsilon, \tilde{q}, R_T)$ where $\Sigma_{\tilde{q}}$ is the signature with the one state $\tilde{q}$ and where for every a, $\tilde{q}(a\, x_1 \dots x_n) \to M_a(\tilde{q}\, x_1) \dots (\tilde{q}\, x_n)$ is a rule of R_T. State $\tilde{q}$ is of type $o \to \mathbb{F}_r$ of order 1 (where o is the atomic type of Σ) and so $T_{\mathtt{A}}$ is of order 1.

Now a simple induction on the structure of a tree t proves the conclusion of the lemma.
□

With lemmas 10 and 9 we can finally prove that HODTR has the same expressivity as HODT.

Theorem 7 *For all HODTR T there exists a HODT T' that defines the same relation. Moreover* $\text{order}(T') = \max(1, \text{order}(T))$.

Proof
Let $T = (\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R_T, \mathtt{A})$ be a HODTR where the set of states of $\mathtt{A}$ is $L = \{\ell_1, \dots, \ell_r\}$. We are going to construct a HODT $T' = \{\Sigma_{Q'}, \Sigma_1, \Sigma_2, q_0, R'\}$ that is equivalent to T.

Lemma 10, implies that there is a transducer $T_{\mathtt{A}}$ of order 2 with initial state $\tilde{q}$ so that for every tree t built on Σ_1, $\mathtt{A}(t) = \ell_i$ iff $T_{\mathtt{A}} = \lambda x_1 \dots x_s.x_i$. We let $\Sigma_{Q'}$ be the union of the signatures of T and $T_{\mathtt{A}}$. Since $T_{\mathtt{A}}$ has order 1, the order of the state signature of T' is indeed $\max(1, \text{order}(T))$.

Given a state q of T, and a constant a of Σ_1 of type $o_1^n \to o_1$, we let f_a be the function from Q^n to $\Lambda^{A_q}(\Sigma_2^\Omega)$ defined by $f_a(\ell_{i_1}, \dots, \ell_{i_n}) = M$ whenever there is a rule $q(a\ x_1 \dots x_n)\langle \ell_{i_1}, \dots, \ell_{i_n} \rangle \to M$ is a rule in R_T. Note that when there is no such rule we then consider that $M = \Omega_{A_q}$ so that f_a is a well-defined function. Now Lemma 9 tells us that there is a term M_a such that:

$$M_a(\lambda x_1 \dots x_r.x_{i_1}) \dots (\lambda x_1 \dots x_r.x_{i_n}) = M \quad \Leftrightarrow \quad f_a(\ell_{i_1}, \dots, \ell_{i_n}) = M \ .$$

We then let $q(a\, x_1 \dots x_n) \to M_a(\tilde{q}\, x_1) \dots (\tilde{q}\, x_n)$ be a rule of T'.

Now given t a tree built on Σ_1, we want to show that for every state $q \in Q$, $|q\, t|_T = |q\, t|_{T'}$. This is obtained by a simple induction on t. □

Note that, when we remove the look-ahead from a HODT, the order of the new transducer is neither linear nor almost-linear, and its order is ≥ 1.

5.3.3 The expressivity of the order of transducers

An important feature of HODT is that it generalizes several types of known transducers defined in section 1.4. In doing so it provides greater insight into their differences.

Deterministic TOP-down tree transducers (DTOP) are one of the most natural classes of tree transducers, they can be seen as HODT of order 0 i.e. transducers where the right side of rules are trees. Macro Tree Transducers (MTT) are similar to DTOP but can use trees given as parameters on the right side of their rules. This added layer of abstraction can be seen in the framework of typed λ-calculus as going from terms of type o of order 0 (trees) to terms of types $o^n \rightarrow o$ of order 1 (tree contexts).

Noting $\mathrm{HODT}_{\leq \mathrm{n}}$ the restriction of HODT to order n, the models $\mathrm{HODT}_{\leq \mathrm{n}}$ for $n \in \mathbb{N}$ form a strict hierarchy of classes of tree transformations. This can shown in a similar way as in [16] for high-level tree transducers.

The approach of functional programming for tree transducers is also interesting when we study partial transductions. When we look at MTTs representing partial tree-to-tree functions, a blocked computation can arise when a state of the transducer has no rule for a particular node. However, a rule of the transducer applied to a node does not have to use the results of computations on its child nodes. When a rule ignores a branch in which the computation is blocked, the transducer could still produce an output. We could choose to consider such a computation as valid or invalid, in functional programming this respectively corresponds to the call-by-name and call-by-value methods of evaluating programs. In λ-calculus these methods correspond to different strategies for reducing terms. Again functional programming brings better understanding of the workings of transducers.

Besides order, there are other constraints we can put on computations of transducers. One way to do this is to restrict the ability of transducers to make copies of a subtree. This has already been done on MTTs and Attribute Tree Transducers (ATTs) in the form of the Single-Use Restricted property, it consists in putting a bound on the number of times a subtree can be copied. The resulting classes of transducers have been respectively named MTT_{SUR} and ATT_{SUR}. We prove in chapter 7 that those models represent the same tree-to-tree functions as $\mathrm{HODTR}_{\mathrm{lin}}$, so the linearity constraint on terms corresponds to that Single-Use Restricted property. This class of functions is especially important because it has been shown to also be the class of functions defined by Monadic Second-Order Logic (MSOT).

The almost-linearity constraint is similar to linearity, but is more permissive since it allows the copying of subtrees. We also prove in chapter 7 that the almost-linear variant is equivalent to the ATT model, and to the model of Monadic Second-Order Logic Transductions with Sharing of subtrees (MSOTS). This result is interesting because it removes the bound on copying without reaching the great complexity of MTTs.

These equivalences of models, along with the algorithms translating transducers from one model to the other, are interesting by themselves. But they are also useful because we give in chapter 6 two distinct algorithms for computing the composition of two transducers. These two algorithms rely on models of the λ-calculus, and in doing so they provide a deeper understanding of the algorithms of composition of equivalent classes of transducers.

Chapter 6

Composition of transducers

In this chapter we present two procedures for computing a transducer which computes the composition of the transductions of two transducers. The first one computes the composition of transducers in HODTR using a semantic analysis of λ-terms based on Scott models. The second one computes the composition of transducers in $\mathrm{HODTR}_{\mathrm{lin}}$ using a finer grained analysis based on coherent spaces, as introduced by Girard in [18]. This second procedure can also be used for computing the composition of HODT in general.

6.1 Composition of HODT based on Scott models

Our goal here is to prove that HODT are closed under composition. For this we start by showing how to extend the transformations they define on trees to λ-terms. Given a transducer $T = (\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R)$, what does it mean for example for T to transform a λ-term of type $o_1 \to o_1$ into a λ-term of Σ_2? A λ-term of that type denotes a function from trees to trees, and the question is what should we do of its argument. A simple answer is to consider that it has been transformed by T, then we need to know with which state. As we cannot answer *a priori* we replace that argument with several arguments each of which represent the putative result of the transformation of that argument by one of the states of T. Generalizing this idea to higher orders, we define a transformation $\langle\cdot,\cdot\rangle_T$ that maps a type $A \in \mathrm{types}(\Sigma_1)$ and a state q of Σ_Q, to the type $\langle q, A\rangle_T \in \mathrm{types}(\Sigma_2)$ which will be the type of the result of applying T with state q to a λ-term of type A (we assume that the states of Σ_Q are $\{q_1, \ldots, q_r\}$):

- $\langle q, o_1\rangle_T = A_q$,
- $\langle q, A \to B\rangle_T = \langle q_1, A\rangle_T \to \cdots \to \langle q_r, A\rangle_T \to \langle q, B\rangle_T$.

When T is clear from the context, we simply write $\langle q, A\rangle$.

We now explain how T transforms any λ-term starting with a state q. For this we define an operation $\langle q, M, \mu\rangle_T$ where μ is an injective mapping that maps pairs of variables and states (x^A, q') to variables $y^{A_{q'}}$. The function μ is a way of *renaming* variables x in a λ-abstraction into variables that represent the result of the transformation of x by a given state of T (we thus call such functions renamings). The definition of $\langle q, M, \mu\rangle_T$ is given by an induction on M:

- $\langle q, x^A, \mu\rangle_T = \mu(x^A, q)$,
- $\langle q, MN, \mu\rangle_T = \langle q, M, \mu\rangle_T \langle q_1, N, \mu\rangle_T \dots \langle q_r, N, \mu\rangle_T$,
- $\langle q, \lambda x^A.M, \mu\rangle_T = \lambda x_1^{\langle q_1, A\rangle_T} \dots x_r^{\langle q_r, A\rangle_T}.\langle q, M, \mu'\rangle_T$ where μ' is equal to μ but for the pairs (x^A, q_i) which are mapped to the fresh variables $x_i^{\langle q_i, A\rangle_T}$,
- $\langle q, a, \mu\rangle_T = \lambda x_{1,1} \dots x_{1,r} \dots x_{s,1} \dots x_{s,r}.M'$ when there is a rule $q\,(a x_1 \dots x_s) \to M'[x_{i,j} := q_j\, x_i]_{i\in[s], j\in[r]}$ in T_2 (here we implicitly assume that M' is in β-normal form and that it is in $\Lambda(\Sigma_2^\Omega)$),
- $\langle q, \Omega_{o_1}, \mu\rangle_T = \Omega_{A_q}$.

Similarly to what we did for type transformation, we allow ourselves to write $\langle q, M, \mu\rangle$ and, when μ is irrelevant (e.g. when it is the empty function) or obvious from the context, we simply write $\langle q, M\rangle$. A first thing to remark is that the result of $\langle q, M^A, \mu\rangle$ is a term of $\Lambda^{\langle q, A\rangle}(\Sigma_2^\Omega)$, the *result of applying T to M with state q*.

For the moment we just have given an intuitive meaning to that transformation, we are now going to formalize that meaning so as to be able to use it in the construction that computes the composition of two HODT. For this we use a logical relation.

Logical relation

We define the logical relation $\mathcal{R} = (\mathcal{R}_A)_{A\in\mathrm{types}(\Sigma_1)}$ where $\mathcal{R}_A$ is a subset of $\Lambda^A(\Sigma_1) \times (\Lambda^{\langle A, q_1\rangle}(\Sigma_2^\Omega) \times \dots \times \Lambda^{\langle A, q_r\rangle}(\Sigma_2^\Omega))$. We use the notation $M\ \mathcal{R}_A\ (M_1, \dots, M_r)$ to mean that M and $(M_1, \dots, M_r)$ are related in the relation $\mathcal{R}_A$, i.e. $(M, (M_1, \dots, M_r)) \in \mathcal{R}_A$. The relations $\mathcal{R}_A$ are inductively defined on A as follows:

- $M\ \mathcal{R}_{o_1}\ (M_1, \dots, M_r)$ when M is a closed term of type o_1 and for every i: $M_i =_\beta |q_i\, M|_T$,
- $M\ \mathcal{R}_{A\to B}\ (M_1, \dots, M_r)$ if and only if for every terms $N, N_1, \dots N_r$ such that $N\ \mathcal{R}_A\ (N_1, \dots, N_r)$: $\quad MN\ \mathcal{R}_B\ (M_1 N_1, \dots, M_r N_r)$.

When the type is clear or irrelevant we write $M\ \mathcal{R}\ (M_1, \dots, M_r)$. The logical relation is defined so that it is closed under $=_\beta$ in all its components.

Lemma 11 For every A, if $M =_\beta N$, $M_1 =_\beta N_1$, ..., $M_r =_\beta N_r$ and $M\ \mathcal{R}_A\ (M_1, \dots, M_r)$, then $N\ \mathcal{R}_A\ (N_1, \dots, N_r)$.

Proof
Straightforward induction on A. □

We are now in position to prove the main property about our transformation: the adequacy Lemma.

Lemma 12 Given a term $M \in \Lambda(\Sigma_1)$, a substitution σ, a renaming μ, a substitution θ so that for every x,

$$\sigma(x)\ \mathcal{R}\ (\theta(\mu(x,q_1)),\dots,\theta(\mu(x,q_r)))\ ,$$

then we have:

$$M.\sigma\ \mathcal{R}\ (\langle q_1, M, \mu\rangle.\theta,\dots,\langle q_r, M, \mu\rangle.\theta)\ .$$

Proof

We prove the lemma by induction on M.

In case $M = x$, the conclusion follows from the hypothesis that $\sigma(x)\ \mathcal{R}\ (\theta(\mu(x,q_1)),\dots,\theta(\mu(x,q_r)))$, since $\langle q_i, x, \mu\rangle.\theta = \theta(\mu(x,q_i))$.

In case $M = M_1M_2$, then by induction, we have

$$M_i.\sigma\ \mathcal{R}\ (\langle q_1, M_i, \mu\rangle.\theta,\dots,\langle q_r, M_i, \mu\rangle.\theta)$$

for $i \in [1,2]$. Now by definition of the logical relation we obtain that

$$M_1M_2.\sigma\ \mathcal{R}\ (\langle q_1, M_1, \mu\rangle.\theta\langle Q, M_2, \mu\rangle.\theta,\dots,\langle q_r, M_1, \mu\rangle.\theta\langle Q, M_2, \mu\rangle.\theta)$$

and as $\langle q_i, M_1, \mu\rangle.\theta\langle Q, M_2, \mu\rangle.\theta = (\langle q_1, M_1, \mu\rangle\langle Q, M_2, \mu\rangle).\theta = \langle q_i, M_1M_2, \mu\rangle.\theta$, we obtain that

$$M_1M_2.\sigma\ \mathcal{R}\ (\langle q_1, M_1M_2, \mu\rangle.\theta,\dots,\langle q_r, M_1M_2, \mu\rangle.\theta)\ .$$

In case $M = \lambda x^A.N$, we let μ' be a renaming that extends μ to the pairs (x, q_i), then by induction hypothesis for every $P\mathcal{R}_A(P_1,\dots,P_r)$, letting $\sigma' = \sigma[P/x]$ and $\theta' = \theta[P_1/\mu'(x,q_1),\dots,P_r/\mu'(x,q_r)$, we have that:

$$N.\sigma'\ \mathcal{R}\ (\langle q_1, N, \mu'\rangle.\theta',\dots,\langle q_r, N, \mu'\rangle.\theta')\ .$$

But $N.\sigma' =_\beta (\lambda x.N).\sigma P$ similarly,

$$\begin{array}{rcl}\langle q_i, N, \mu'\rangle.\theta' & =_\beta & ((\lambda y_1\dots y_r.\langle q_i, N, \mu'\rangle).\theta)P_1\dots P_r \\ & = & (\langle q_i, \lambda x.N, \mu\rangle.\theta)P_1\dots P_r\end{array}$$

if $\mu'(x, q_i) = y_i$. From Lemma 11, we then obtain that

$$\begin{array}{rl}(\lambda x.N).\sigma P\ \mathcal{R} & (\langle q_1, \lambda x.N, \mu\rangle.\theta P_1\dots P_r,\dots, \\ & \langle q_r, \lambda x.N, \mu\rangle.\theta P_1\dots P_r)\end{array}$$

The conclusion follows from the definition of the logical relation.

The case where $M = a$ is a simple consequence of the definition of $\langle q_i, a, \mu\rangle$. □

Corollary 8 For every closed $M \in \Lambda(\Sigma_1)$

$$M\ \mathcal{R}\ (\langle q_1, M\rangle,\dots,\langle q_r, M\rangle)\ .$$

This corollary with Lemma 11 proves that for every tree t of Σ_1: $\langle q_i, t\rangle =_\beta |q_i\, t|_T$, which entails that $\langle\cdot,\cdot,\cdot\rangle_T$ is a conservative extension of the action of T on trees.

We now turn to the construction of the HODT that computes the composition of two HODT $T_1 = (\Sigma_Q, \Sigma_1, \Sigma_2, q_1, R_1)$ and $T_2 = (\Sigma_P, \Sigma_2, \Sigma_3, p_1, R_2)$. In a strict view, this composition makes sense only when the output of the first transducer has atomic type, i.e. q_1 has type $o_1 \to o_2$. However, using the logical relation we have just defined we can prove a more general result (see Proposition 4 below).

We let T be the transducer defined by $(\Sigma_S, \Sigma_1, \Sigma_3, \langle p_1, q_1\rangle, R)$ where Σ_S is a state signature where the states are of the form $\langle p, q\rangle$ with p a state of Σ_P, q a state of Σ_Q and the type of $\langle p, q\rangle$ is $o_1 \to \langle p, A_q\rangle_{T_2}$. The intuitive meaning of the state $\langle p, q\rangle$ in T is that it computes the *output* of T_2 with state p from the output of T_1 with state q. Here the notion of output of T_2 is taken in the broader sense we have defined above and that applies to every λ-term.

Before we give the definition of the rules of T, we suppose that the states of T_1 are $\{q_1, \ldots, q_r\}$, while the states of T_2 are $\{p_1, \ldots, p_m\}$. Now, given a rule $q(a\, x_1 \ldots x_s) \to M$ of T_1 and a state p of T_2, we suppose that $M = M'[q_i\, x_j/y_{i,j}]_{i\in[r], j\in[s]}$, and then T has rule $\langle p, q\rangle(a\, x_1 \ldots x_s) \to \langle p, M', \mu\rangle.\theta$ where μ is the renaming that maps $(p_k, y_{i,j})$ to $z_{k,i,j}$ for $k \in [m]$, $i \in [r]$, $j \in [s]$ and θ is the substitution that maps $z_{k,i,j}$ to $\langle p_k, q_i\rangle\, x_j$.

Proposition 4 *For every state p of T_2 and q of T_1 we have, for every tree t of Σ_1:*

$$|\langle p, q\rangle\, t|_T =_\beta \langle p, |q\, t|_{T_1}\rangle_{T_2} \ .$$

So in particular, when T_1 outputs trees, we have that:

Theorem 9 *When q_1 has type $o_1 \to o_2$, for every tree t of Σ_1 we have:*

$$|\langle p_1, q_1\rangle\, t|_T = |p_1\ |q_1\, t|_{T_1}\,|_{T_2} \ .$$

We need to be careful with the interpretation of Theorem 9. Indeed, it is not specified in the theorem hypotheses that $|q_1\, t|_{T_1}$ is in $\Lambda(\Sigma_2)$, there could well be occurrences of Ω in $|q_1\, t|_{T_1}$. As a result, when looking at $\text{rel}(T)$, it may strictly contain $\text{rel}(T_1) \circ \text{rel}(T_2)$. They are equal when T_1 defines a total function. We may also make them coincide by adding an inspection (i.e. a look-ahead that is only used at the beginning of the computation) to T that checks that the input tree t is so that $|q_1\, t|_{T_1}$ (λ-terms that do not contain occurrences of Ω are recognizable and thus using Theorem 5 we can construct such an inspection). We can also modify T with a construct similar to the one we used to remove look-aheads so as to simulate this inspection.

Note also that with these rules for T we have that $\text{order}(T) = \text{order}(T_1) + \text{order}(T_2)$. So when T_1 and T_2 represent DTOPs, their order is 0, and the order of T is then also 0 so that T is also a DTOP. Our construction thus also shows that DTOPs are closed under composition (when considering total DTOPs, or when using inspections). If T_1 and T_2 represent MTTs, then their order is 1 and the order of T is then 2 and is therefore not an MTT in general.

Theorem 10 *Given two HODT, T_1 and T_2 with respective initial state q_1 and p_1 so that T_1 outputs trees, there is an HODT T so that* $\mathrm{order}(T) = \mathrm{order}(T_1) + \mathrm{order}(T_2)$ *and* $\mathrm{rel}(T) = \{(M, |p_1\, |q_1\, M|_{T_1}\, |_{T_2}) \mid M$ *is a closed input tree*$\}$. *Futhermore when T_1 is total* $\mathrm{rel}(T) = \mathrm{rel}(T_1) \circ \mathrm{rel}(T_2)$.

6.2 Composition of HODTR$_{\mathrm{lin}}$ based on coherent spaces

As we are interested in limiting the size of the transducer that is computed, and even though our primary goal is to compose HODTR$_{\mathrm{lin}}$, this section is devoted to the composition of HOWDTR$_{\mathrm{lin}}$. Recall that HOWDTR$_{\mathrm{lin}}$ are HODTR$_{\mathrm{lin}}$ whose look-ahead automaton is not necessarily deterministic, but the rules of the transducer are still deterministic because, despite having several possible look-ahead states for each input tree, only one rule of the transducer is applicable at each step. Working with non-deterministic look-aheads allows us to save the possibly exponential cost of determinizing an automaton.

Example of composition In order to illustrate this algorithm, we will show how it works on an example. We present here two HODTR$_{\mathrm{lin}}$ T_1 and T_2 of which we will compute the composition. For the sake of clarity we have chosen simple and deterministic transducers, but keep in mind that the general procedure works also on more complex, weakly deterministic transducers.

We use a simple tree signature Σ to represent words over an alphabet $\{a, b\}$ of two letters. We define Σ with an atomic type o, and constants $a^{o\to o}$ of type $o \to o$, $b^{o\to o}$ of type $o \to o$, and $\#^o$ of type o. The symbol # is used as an end-of-word symbol. For example the word *baba* is represented by the tree $b(a(b(a\#)))$.

Transducer T_1 computes the reversal of words over $\{a, b\}$. For example the output of T_1 on input tree $b(a(b(a\#)))$ will be $a(b(a(b\#)))$. It has Σ as both input and output tree signature. It contains an initial state q_0 of type $o \to o$, and a state q_1 of type $o \to o \to o$, with rules:

- $q_0(a\ x) \to (q_1\ x)\ (a\ \#)$
- $q_0(b\ x) \to (q_1\ x)\ (b\ \#)$
- $q_0(\#) \to \#$
- $q_1(a\ x) \to \lambda y.(q_1\ x)\ (a\ y)$
- $q_1(b\ x) \to \lambda y.(q_1\ x)\ (b\ y)$
- $q_1(\#) \to \lambda y.y$

This transducer does not use any look-ahead information in its rules, so we can simply define a look-ahead automaton $\mathtt{A}_1$ with a single state ℓ_1 recognizing all trees over signature Σ.

Transducer T_2 takes a word and removes all its initial as and trailing bs. For example the output of T_2 on input tree $a(b(a(b\#)))$ will be $b(a\#)$. It has Σ as both input and output signature.

Its look-ahead automaton $\mathtt{A}_2$ is used to mark the input word's trailing bs. It has two states ℓ_a and ℓ_b with rules:

- $\# \to \ell_b$
- $b(\ell_b) \to \ell_b$
- $a(\ell_b) \to \ell_a$
- $b(\ell_a) \to \ell_a$
- $a(\ell_a) \to \ell_a$

The states of T_2 are the initial state p_0 of type $o \to o$ and state p_1 of type $o \to o$, and the rules are:

- $p_0(\#) \to \#$
- $p_0(a\ x)\langle \_ \rangle \to p_0\ x$
- $p_0(b\ x)\langle \ell_a \rangle \to b\ (p_1\ x)$
- $p_0(b\ x)\langle \ell_b \rangle \to p_1\ x$
- $p_1(\#) \to \#$
- $p_1(a\ x)\langle \_ \rangle \to a\ (p_1\ x)$
- $p_1(b\ x)\langle \ell_a \rangle \to b\ (p_1\ x)$
- $p_1(b\ x)\langle \ell_b \rangle \to p_1\ x$

Here we use _ to mean that the rule is the same for ℓ_a and ℓ_b.

In the rest of the section we will work to build a transducer which computes the composition of T_1 and T_2, noted $T = T_1 \circ T_2$.

6.2.1 Semantic analysis

Let $T_1 = (\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R_1, \mathtt{A}_1)$ and $T_2 = (\Sigma_P, \Sigma_2, \Sigma_3, p_0, R_2, \mathtt{A}_2)$ be two Linear High-Order Weakly Deterministic tree Transducers with Regular look-ahead. The rules of T_1 can be written:

$$q(a\overrightarrow{x})\langle \overrightarrow{\ell} \rangle \to M\,(q_1\ x_1)\dots(q_n\ x_n)$$

where $q, q_1, \dots, q_n \in Q$ are states of T_1, $\overrightarrow{\ell} = \ell_1, \dots, \ell_n$ are states of $\mathtt{A}_1$ and the λ-term M is of type $A_{q_1} \to \dots \to A_{q_n} \to A_q$. Our goal is to build a HOWDTR$_{\text{lin}}$ $T :: \Sigma_1 \to \Sigma_3$ that does the composition of T_1 and T_2, so we want to replace a rule such as that one with a new rule which corresponds to applying T_2 to the term M.

In order to do so, we need, for each tree of type o_2 in M, to know the associated state $\ell \in L_2$ of T_2's look-ahead, and the state $p \in P$ of T_2 which is going to process that node. So with any such tree we associate the pair (p, ℓ). In this case we call (p, ℓ) the *token* which represents the behavior of the tree. In general, we want to associate *tokens* not only with trees, but also with λ-terms of higher order. For example, we map an occurrence of a symbol $a \in \Sigma_2$ of type $o_2 \to o_2 \to o_2$, whose arguments x_1 and x_2 (of type o_2) respectively have look-ahead states ℓ_1 and ℓ_2 and are processed by states p_1 and $p_2 \in P$ of T_2, to the *token* $(p_1, \ell_1) \multimap (p_2, \ell_2) \multimap (p, \ell)$ where (p, ℓ) is the token of the tree $a\ x_1 x_2$ (of type o_2). We formally define *tokens* as follows:

Definition 20 The set of *semantic tokens* $[\![A]\!]$ over a type A built on atomic type o_2 is defined by induction:
$[\![o_2]\!] = \{(p, \ell) \mid p \in P, \ell \in L_2\}$
$[\![A \to B]\!] = \{f \multimap g \mid f \in [\![A]\!], g \in [\![B]\!]\}$

Naturally, the semantic token associated with a λ-term M of type A built on atomic type o_2 will depend on the context where the term M appears. For example a tree of atomic type o_2 can be processed by any state $p \in P$ of T_2, and a term of type $A \to B$ can be applied to any argument of type A. But for any such M taken out of context, there exists a finite set of possible tokens for it. For example, a given tree of type o_2 can be processed by any state $p \in P$ depending on the context, but not with any look-ahead state $\ell \in L_2$.

Example

We can apply this to the example transducer T_2 defined at the beginning of section 6.2. Noting $[\![M]\!]$ the set of possible tokens for a term M, we get:

- $[\![\#]\!] = \{(p_0, \ell_b), (p_1, \ell_b)\}$
- $[\![b\ \#]\!] = \{(p_0, \ell_b), (p_1, \ell_b)\}$
- $[\![a\ (b\ \#)]\!] = \{(p_0, \ell_a), (p_1, \ell_a)\}$

Note that, since the look-ahead is deterministic, each term of type o has only one possible look-ahead state (either ℓ_a or ℓ_b).

Because there is the rule $p_0(b\ x)\langle \ell_a \rangle \to b\ (p_1\ x)$ in T_2 and the rule $b(\ell_a) \to \ell_a$ in $\mathtt{A}_2$, the term b of type $o \to o$ can be associated with token $(p_1, \ell_a) \multimap (p_0, \ell_a)$. This means that, if the argument x of b has behavior (p_1, ℓ_a), then the result of b applied to x (i.e. $b\, x$) may have behavior (p_0, ℓ_a).
This way we can deduce the sets of possible tokens for constants a and b:

$$[\![a]\!] = \{(p_0, \ell_a) \multimap (p_0, \ell_a), (p_0, \ell_b) \multimap (p_0, \ell_a), (p_1, \ell_a) \multimap (p_1, \ell_a), (p_1, \ell_b) \multimap (p_1, \ell_a)\}$$

$$[\![b]\!] = \{(p_1, \ell_a) \multimap (p_0, \ell_a), (p_1, \ell_b) \multimap (p_0, \ell_b), (p_1, \ell_a) \multimap (p_1, \ell_a), (p_1, \ell_b) \multimap (p_1, \ell_b)\}$$

We will later see examples of tokens for more complex terms.

In order to define the set of possible semantic tokens for any term, we use a system of derivation rules. The following derivation rules are used to derive judgments which associate a term with a semantic token. So a judgment $\Gamma \vdash M : f$ associates term M with token f, where Γ is a substitution which maps free variables in M to tokens. The rules are the following:

$$\frac{p(a\ \overrightarrow{x})\langle \ell_1, \ldots, \ell_n \rangle \xrightarrow{T_2} M(p_1\, x_1) \ldots (p_n\, x_n) \qquad \mathtt{A}_2(a\,(\ell_1, \ldots, \ell_n)) = \ell}{\vdash a : (p_1, \ell_1) \multimap \cdots \multimap (p_n, \ell_n) \multimap (p, \ell)}$$

$$\frac{\Gamma_1 \vdash M : f \multimap g \qquad \Gamma_2 \vdash N : f}{\Gamma_1, \Gamma_2 \vdash M\, N : g}$$

$$\frac{\Gamma, x^A : f \vdash M : g}{\Gamma \vdash \lambda x^A . M : f \multimap g} \qquad\qquad \frac{f \in [\![A]\!]}{x^A : f \vdash x^A : f}$$

Using this system we can derive, for any term M^A, all the semantic tokens that correspond to possible behaviors of M^A when it is processed by T_2.

6.2.2 Unicity of derivation for semantic token judgements

We will later show that we can compute the image of M through T_2 from the derivation of the judgement $\vdash M : f$, assuming that f is the token which represents the behavior of T_2 on M. But before that we need to prove that for a given term M and token f the derivation of the judgement $\vdash M : f$ is unique.

Theorem 11 *For every type A, for every term M of type A and every token $f \in [\![A]\!]$, there is at most one derivation $\mathcal{D} :: \vdash M : f$.*

This theorem relies in part on the fact that tokens form a *coherent space*, as first introduced by Girard in [18]. Before proving this we introduce known definitions and properties of coherent spaces under the framework of linear logic.

Coherent spaces

Our main goal now is to indicate that for all term M of type A and all token $f \in [\![A]\!]$ which corresponds to a behavior of M, there is only one possible derivation for the judgement $\vdash M : f$, which will be the key trick to preserve linearity in composition. In order to prove that, we will see that tokens form a coherent space.

First, we define a coherence relation $\mathrel{\overset{\frown}{\smile}}_A \subseteq [\![A]\!] \times [\![A]\!]$ for all type A by induction on A:

Definition 21 For all $p, p' \in P$ and $\ell, \ell' \in L_2$,

$$(p, \ell) \mathrel{\overset{\frown}{\smile}}_{o_2} (p', \ell') \Leftrightarrow \text{there exists a tree recognized by both } \ell \text{ and } \ell'$$

For all type $A, B \in \text{types}(o_2)$, for all $f, f' \in [\![A]\!]$ and $g, g' \in [\![B]\!]$:

$$f \multimap g \mathrel{\overset{\frown}{\smile}}_{A \to B} f' \multimap g' \;\Leftrightarrow\; (f \mathrel{\overset{\frown}{\smile}}_A f' \Rightarrow (g \mathrel{\overset{\frown}{\smile}}_B g' \wedge (f \neq f' \;\Rightarrow\; g \neq g')))$$

Intuitively, two tokens are coherent if they can both be derived from the same term. For tokens of type o_2 for instance, it means that there must be a tree recognized by both look-ahead states. In the special case where the transducer is deterministic, the look-ahead states must be the same: $(p, \ell) \mathrel{\overset{\frown}{\smile}}_{o_2} (p', \ell') \Leftrightarrow \ell = \ell'$.

We also define the corresponding incoherence relation $\asymp_A \in [\![A]\!] \times [\![A]\!]$: intuitively, two tokens are incoherent if they can not both be possible distinct tokens for the same term, so if they are either equal or not coherent with each other.

Definition 22 For all type A built on o_2:

$$f \asymp_A f' \Leftrightarrow \neg(f \mathrel{\overset{\frown}{\smile}}_A f') \vee f = f'$$

The incoherence relation allows us to give a simpler alternative definition of the coherence relation $\mathrel{\overset{\frown}{\smile}}_{A \leftarrow B}$ between tokens in $[\![A \to B]\!]$: for all $f, f' \in [\![A]\!]$ and $g, g' \in [\![B]\!]$,

$$f \multimap g \mathrel{\overset{\frown}{\smile}}_{A \to B} f' \multimap g' \quad\Leftrightarrow\quad (f \mathrel{\overset{\frown}{\smile}}_A f' \;\Rightarrow\; g \mathrel{\overset{\frown}{\smile}}_B g') \wedge (g \asymp_B g' \;\Rightarrow\; f \asymp_A f')$$

Theorem 12 *For all type A and term M^A of type A, if there exists two semantic tokens $f, f' \in [\![A]\!]$ associated with M^A, i.e. the judgments $\vdash M : f$ and $\vdash M : f'$ are derivable, then f and f' are coherent: $f \mathrel{\overset{\frown}{\smile}}_A f'$.*

In order to prove this theorem, we need to prove a stronger theorem, by induction on term M:

Theorem 13 *If there exists two derivations* $\mathcal{D} :: \Gamma \vdash M : f$ *and* $\mathcal{D}' :: \Gamma' \vdash M : f'$ *then* $\Gamma \multimap f \overset{\frown}{\smile} \Gamma' \multimap f'$.

Here, when writing $\Gamma \multimap f$ with $\Gamma = x_1 : f_1, \ldots, x_n : f_n$, we mean by Γ the tensor product $(f_1, \ldots, f_n)$.

Proof

We prove this by induction on term M:

- If $M = a$ is a constant from Σ_2 then the last rules of $\mathcal{D}$ and $\mathcal{D}'$ are:

$$\mathcal{D} :: \frac{p(a\,\overrightarrow{x})\langle \ell_1, \ldots, \ell_n \rangle \xrightarrow{T_2} M(p_1\,x_1)\ldots(p_n\,x_n) \qquad \mathtt{A}_2(a\,(\ell_1, \ldots, \ell_n)) = \ell}{\vdash a : (p_1, \ell_1) \multimap \cdots \multimap (p_n, \ell_n) \multimap (p, \ell)}$$

$$\mathcal{D}' :: \frac{p'(a\,\overrightarrow{x})\langle \ell'_1, \ldots, \ell'_n \rangle \xrightarrow{T_2} M(p'_1\,x_1)\ldots(p'_n\,x_n) \qquad \mathtt{A}_2(a\,(\ell'_1, \ldots, \ell'_n)) = \ell'}{\vdash a : (p'_1, \ell'_1) \multimap \cdots \multimap (p'_n, \ell'_n) \multimap (p', \ell')}$$

  We need to prove that:

$$(p_1, \ell_1) \multimap \cdots \multimap (p_n, \ell_n) \multimap (p, \ell) \overset{\frown}{\smile} (p'_1, \ell'_1) \multimap \cdots \multimap (p'_n, \ell'_n) \multimap (p', \ell')$$

  which is equivalent to:

$$((p_1, \ell_1), \ldots, (p_n, \ell_n)) \multimap (p, \ell) \overset{\frown}{\smile} ((p'_1, \ell'_1), \ldots, (p'_n, \ell'_n)) \multimap (p', \ell')$$

  To prove this we assume that $((p_1, \ell_1), \ldots, (p_n, \ell_n)) \overset{\frown}{\smile} ((p'_1, \ell'_1), \ldots, (p'_n, \ell'_n))$ and we show that:

$$(p, \ell) \overset{\frown}{\smile} (p', \ell') \wedge \big((p, \ell) = (p', \ell') \Rightarrow ((p_1, \ell_1), \ldots, (p_n, \ell_n)) = ((p'_1, \ell'_1), \ldots, (p'_n, \ell'_n))\big)$$

  $((p_1, \ell_1), \ldots, (p_n, \ell_n)) \overset{\frown}{\smile} ((p'_1, \ell'_1), \ldots, (p'_n, \ell'_n))$ means that, for all $i \leq n$, $(p_i, \ell_i) \overset{\frown}{\smile} (p'_i, \ell'_i)$ and so there exists a tree t_i recognized by both look-ahead states ℓ_i and ℓ'_i. Then the tree $a\,t_1 \ldots t_n$ is recognized by both look-ahead states ℓ and ℓ', so $(p, \ell) \overset{\frown}{\smile} (p', \ell')$.

  Note that we assumed transducer T_2 to be weakly deterministic, but in the deterministic case we would have $\ell_i = \ell'_i$ for each i and therefore $\ell = \ell'$.

  Furthermore, if $(p, \ell) = (p', \ell')$ then, because T_2 is weakly deterministic and because there exists, for each $i \leq n$, a tree recognized by both ℓ_i and ℓ'_i, the rules of T_2 appearing in $\mathcal{D}$ and $\mathcal{D}'$ are the same. So $((p_1, \ell_1), \ldots, (p_n, \ell_n)) = ((p'_1, \ell'_1), \ldots, (p'_n, \ell'_n))$.

- If $M = N_1\,N_2$ then the last rules of $\mathcal{D}$ and $\mathcal{D}'$ respectively are of the form:

$$\frac{\Gamma_1 \vdash N_1 : g \multimap f \qquad \Gamma_2 \vdash N_2 : g}{\Gamma_1, \Gamma_2 \vdash N_1\,N_2 : f} \qquad \frac{\Gamma'_1 \vdash N_1 : g' \multimap f' \qquad \Gamma'_2 \vdash N_2 : g'}{\Gamma'_1, \Gamma'_2 \vdash N_1\,N_2 : f'}$$

Through the induction hypothesis, we get that $\Gamma_1 \multimap (g \multimap f) \mathrel{\overset{\frown}{\smile}} \Gamma'_1 \multimap (g' \multimap f')$ and $\Gamma_2 \multimap g \mathrel{\overset{\frown}{\smile}} \Gamma'_2 \multimap g'$. Then $\Gamma_1, \Gamma_2 \mathrel{\overset{\frown}{\smile}} \Gamma'_1, \Gamma'_2$ implies that $\Gamma_1 \mathrel{\overset{\frown}{\smile}} \Gamma'_1$ and $\Gamma_2 \mathrel{\overset{\frown}{\smile}} \Gamma'_2$, which means that $g \multimap f \mathrel{\overset{\frown}{\smile}} g' \multimap f'$ and $g \mathrel{\overset{\frown}{\smile}} g'$, which in turn implies that $f \mathrel{\overset{\frown}{\smile}} f'$. Reciprocally, assuming that $f \asymp f'$, we have two cases depending on whether or not $g \asymp g'$. On the one hand we have that $g \asymp g'$ implies that $\Gamma_2 \asymp \Gamma'_2$ and therefore $\Gamma_1, \Gamma_2 \asymp \Gamma'_1, \Gamma'_2$, on the other hand we have that $f \asymp f'$ and $g \mathrel{\overset{\frown}{\smile}} g'$ imply that $g \multimap f \asymp g' \multimap f'$ and so $\Gamma_1 \asymp \Gamma'_1$ and $\Gamma_1, \Gamma_2 \asymp \Gamma'_1, \Gamma'_2$. In either case $f \asymp f'$ implies that $\Gamma_1, \Gamma_2 \asymp \Gamma'_1, \Gamma'_2$. Finally we can conclude that $\Gamma_1, \Gamma_2 \multimap f \mathrel{\overset{\frown}{\smile}} \Gamma'_1, \Gamma'_2 \multimap f'$

- If $M = \lambda x^B.N$ then $f = g \multimap h$, $f' = g' \multimap h'$ and the last rules of $\mathcal{D}$ and $\mathcal{D}'$ respectively are:

$$\frac{\Gamma, x^B : g \vdash N : h}{\Gamma \vdash \lambda x^B.N : g \multimap h} \qquad\qquad \frac{\Gamma', x^B : g' \vdash N : h'}{\Gamma' \vdash \lambda x^B.N : g' \multimap h'}$$

The induction hypothesis gives $(\Gamma, x^B : g) \multimap h \mathrel{\overset{\frown}{\smile}} (\Gamma', x^B : g') \multimap h'$, which we can write: $(\Gamma, g) \multimap h \mathrel{\overset{\frown}{\smile}} (\Gamma', g') \multimap h'$ using the tensor product, and that is equivalent to $\Gamma \multimap (g \multimap h) \mathrel{\overset{\frown}{\smile}} \Gamma' \multimap (g' \multimap h')$.

- If $M = x^A$ then $f, f' \in \llbracket A \rrbracket$. So $\Gamma = x^A : f$ and $\Gamma' = x^A : f'$ and derivations $\mathcal{D}$ and $\mathcal{D}'$ are:

$$\frac{f \in \llbracket A \rrbracket}{x^A : f \vdash x^A : f} \qquad\qquad \frac{f' \in \llbracket A \rrbracket}{x^A : f' \vdash x^A : f'}$$

Trivially $f \mathrel{\overset{\frown}{\smile}} f' \;\Rightarrow\; f \mathrel{\overset{\frown}{\smile}} f'$ and $f \asymp f' \;\Rightarrow\; f \asymp f'$, therefore $f \multimap f \mathrel{\overset{\frown}{\smile}} f' \multimap f'$. So $\Gamma \multimap f \mathrel{\overset{\frown}{\smile}} \Gamma' \multimap f'$.

We have shown theorem 13, of which theorem 12 is a particular case, by induction on M. Indeed if M is a closed term and Γ and Γ' are empty substitutions then $\Gamma \multimap f$ is f and $\Gamma' \multimap f'$ is f', therefore $f \mathrel{\overset{\frown}{\smile}} f'$. □

We have shown that any two tokens derivable for a same term are coherent. So the set of tokens derivable for a given term M^A form a clique in the coherence graph of $\llbracket A \rrbracket$, we call it the *coherent state* of term M^A in $\llbracket A \rrbracket$.

Now, using the previous theorem, we will be able to prove that there is only one way of deriving any given derivable judgement $\vdash M : f$.

Proof of unicity

We can now prove theorem 11:

Proof

Because subterms of M may have free variables, we add a substitution Γ to the induction hypothesis:

"If there exists two derivations $\mathcal{D} :: \Gamma \vdash M : f$ and $\mathcal{D}' :: \Gamma \vdash M : f$ then $\mathcal{D}$ and $\mathcal{D}'$ are the same."

We prove this by induction on term M, there are several cases:

- If $M = a$ is a constant from Σ_2 or if $M = x$ is a free variable in Γ then derivations $\mathcal{D}$ and $\mathcal{D}'$ are axioms so they must be equal.

- If $M = N_1\, N_2$ then the last rules of $\mathcal{D}$ and $\mathcal{D}'$ respectively are of the form:

$$\frac{\Gamma_1 \vdash N_1 : g \multimap f \qquad \Gamma_2 \vdash N_2 : g}{\Gamma_1, \Gamma_2 \vdash N_1\, N_2 : f} \qquad \frac{\Gamma_1' \vdash N_1 : g' \multimap f \qquad \Gamma_2' \vdash N_2 : g'}{\Gamma_1', \Gamma_2' \vdash N_1\, N_2 : f}$$

  where $\Gamma_1, \Gamma_2 = \Gamma = \Gamma_1', \Gamma_2'$. Since the variables substituted by substitutions Γ_1 and Γ_1' must be the free variables in term N_1, $\Gamma_1 = \Gamma_1'$ (because $\mathsf{dom}(\Gamma_1) = \mathsf{FV}(N_1) = \mathsf{dom}(\Gamma_1')$). Similarly, we deduce that $\Gamma_2 = \Gamma_2'$. Then we can apply theorem 13 to the derivations of $\Gamma_2 \vdash N_2 : g$ and $\Gamma_2' \vdash N_2 : g'$, and to the derivations of $\Gamma_1 \vdash N_1 : g \multimap f$ and $\Gamma_1' \vdash N_1 : g' \multimap f$. The first application yields $g \supset g'$ (since $\Gamma_2 = \Gamma_2'$), the second yields $g \multimap f \supset g' \multimap f$ (because $\Gamma_1 = \Gamma_1'$), together they imply that $g = g'$. Finally we can apply the induction hypothesis to get unicity of a derivation of $\Gamma_1 \vdash N_1 : g \multimap f$ and unicity of a derivation of $\Gamma_2 \vdash N_2 : g$, this implies that derivations $\mathcal{D}$ and $\mathcal{D}'$ are the same.

- If $M = \lambda x^B.N$ then $f = g \multimap h$ and the last rule of $\mathcal{D}$ and $\mathcal{D}'$ is the same:

$$\frac{\Gamma, x^B : g \vdash N : h}{\Gamma \vdash \lambda x^B.N : g \multimap h}$$

  The induction hypothesis implies the unicity of a derivation of $\Gamma, x^B : g \vdash N : h$, which entails the unicity of a derivation of $\Gamma \vdash \lambda x^B.N : g \multimap h$.

□

Now that we have shown that there is only one derivation per judgement $\vdash M : f$, we are going to see how to use that derivation in order to compute the term N that is the image of M by transducer T_2.

6.2.3 Collapsing of token derivations

We define a function (we call it collapsing function) which maps every derivation $\mathcal{D} :: \vdash M : f$ to a term $\overline{\mathcal{D}}$ which corresponds to the output of transducer T_2 on term M assuming that M has behavior f.

Definition 23 Let $\mathcal{D}$ be a derivation. We define $\overline{\mathcal{D}}$ by induction on $\mathcal{D}$, there are different cases depending on the first rule of $\mathcal{D}$:

If $\mathcal{D}$ is of the form:

$$\frac{p(a\,\overrightarrow{x})\langle \ell_1, \ldots, \ell_n\rangle \xrightarrow{T_2} N(p_1\,x_1)\ldots(p_n\,x_n) \qquad \mathtt{A}_2(a\,(\ell_1, \ldots, \ell_n)) = \ell}{\vdash a : (p_1, \ell_1) \multimap \cdots \multimap (p_n, \ell_n) \multimap (p, \ell)}$$

then $\overline{\mathcal{D}} = N$,

if $\mathcal{D}$ is of the form:

$$\frac{\mathcal{D}_1 :: \Gamma_1 \vdash N_1 : f \multimap g \qquad \mathcal{D}_2 :: \Gamma_2 \vdash N_2 : f}{\Gamma_1, \Gamma_2 \vdash N_1\, N_2 : g}$$

then $\overline{\mathcal{D}} = \overline{\mathcal{D}}_1\, \overline{\mathcal{D}}_2$,

if $\mathcal{D}$ is of the form:

$$\frac{\mathcal{D}_1 :: \Gamma, x^A : f \vdash N : g}{\Gamma \vdash \lambda x^A.N : f \multimap g}$$

then $\overline{\mathcal{D}} = \lambda x^{\overline{f}}.\overline{\mathcal{D}_1}$,

if $\mathcal{D}$ is of the form:

$$\frac{f \in [\![A]\!]}{x^A : f \vdash x^A : f}$$

then $\overline{\mathcal{D}} = x^{\overline{f}}$.

We can check that, for all derivation $\mathcal{D} :: \vdash M : f$, the term $\overline{\mathcal{D}}$ is of type $\overline{f}$ given by: $\overline{(p,\ell)} = A_p$ and $\overline{f \multimap g} = \overline{f} \rightarrow \overline{g}$.

Now that we have associated, with any pair (M, f) such that f is a semantic token of term M, a term $N = \overline{\mathcal{D}}$ which represents the image of M by T_2, we need to show that replacing M with N in the computation of transducers leads to the same results. In order to do so we use a logical relation.

Example of derivation collapsing

Transducer T_1 has a rule: $q_0(b\ x) \to (q_1\ x)\,(b\ \#)$ which can also be written $q_0(b\ x) \to M\,(q_1\ x)$ with $M = \lambda y.y\,(b\ \#)$. We show how to associate the token $f = ((p_1,\ell_b) \multimap (p_0,\ell_a)) \multimap (p_0,\ell_a)$ to this term M of type $(o \to o) \to o$, and how to obtain the image of M through transducer T_2 (assuming f describes how M is processed by T_2). We derive the judgment $\vdash M : f$ as follows:

$$\dfrac{\dfrac{\dfrac{(p_1,\ell_b)\multimap(p_0,\ell_a)\in[\![o\to o]\!]}{y^{o\to o}:(p_1,\ell_b)\multimap(p_0,\ell_a)\vdash y^{o\to o}:(p_1,\ell_b)\multimap(p_0,\ell_a)} \qquad \dfrac{\dfrac{p_1(b\,x)\langle\ell_b\rangle\xrightarrow{T_2}p_1\,x \qquad \mathtt{A}_2(b\,(\ell_b))=\ell_b}{\vdash b:(p_1,\ell_b)\multimap(p_1,\ell_b)} \qquad \dfrac{p_1(\#)\xrightarrow{T_2}\# \qquad \mathtt{A}_2(\#)=\ell_b}{\vdash \#:(p_1,\ell_b)}}{\vdash b\,\#:(p_1,\ell_b)}}{y^{o\to o}:(p_1,\ell_b)\multimap(p_0,\ell_a)\vdash y\,(b\,\#):(p_0,\ell_a)}}{\vdash \lambda y.y\,(b\,\#):((p_1,\ell_b)\multimap(p_0,\ell_a))\multimap(p_0,\ell_a)}$$

With this derivation $\mathcal{D}$ we associate the term $\overline{\mathcal{D}}$:

$$\overline{\mathcal{D}} = \lambda x.x((\lambda z.z)(\#)) =_{\beta\eta} \lambda x.x\,\#$$

Intuitively, $\overline{\mathcal{D}}$ represents the image of $M = \lambda y.y\,(b\ \#)$ through T_2. We can see that it removes the trailing b, as T_2 would. Note that the structure of term $\overline{\mathcal{D}}$ is the same as that of the derivation, which is the same as that of the initial term M.

The token information is important because it allows us to transform constants like b and $\#$ into their image through T_2. For example, b is transformed into $\lambda z.z$ here, but if tokens were different then we might use the rule $p_1(b\,x)\langle\ell_a\rangle \to b\,(p_1\,x)$, in that case we would transform b into $\lambda z.b\,z$.

The fact that b is applied to $\#$ in M implies that the token associated with $\#$ has look-ahead ℓ_b, which is why look-ahead state ℓ_b appears in the rule of T_2 used to process b. So applying b to $\#$ indirectly implies that the b should be removed by T_2. Note that variable x has type $A_{p_1} \to A_{p_0}$, which happens to be the same type as variable y, but it will not always be the case. With more complex types of states p_0, p_1 of T_2, variables in $\overline{\mathcal{D}}$ would have more complex types.

Logical relation

Our logical relation is indexed on a type A and a semantic token $f \in [\![A]\!]$, it is defined as follows:

Definition 24 We define the logical relation R^A_f, for all type A built on atomic type o_2 and for all semantic token $f \in [\![A]\!]$, by induction on type A:

$$R^{o_2}_{(p,\ell)} = \{(M,N) \mid p(M\!\downharpoonleft_\beta) \overset{T_2}{=} N\!\downharpoonleft_\beta,\ \mathtt{A}_2(M\!\downharpoonleft_\beta) = \ell\}$$

$$R^{A\to B}_{f\multimap g} = \{(M,N) \mid \forall (M',N') \in R^A_f, (M\,M', N\,N') \in R^B_g\}$$

This logical relation is, by definition, the one we want to have between the terms M which appear on the right side of rules of T_1 and the corresponding N that will appear on the right side of the rules of our new transducer T which computes the composition of T_1 and T_2.

What we want to show now is that it is the same relation as the (M, N) such that N is the collapsing of the unique (unique according to theorem 11) derivation of the judgment $\vdash M : f$, this property is called *adequation*. Formally it means that, for all type $A \in \text{types}(o_2)$, token $f \in [\![A]\!]$ and for any closed terms M and N of respective types A and $\overline{f}$:

$$\exists \mathcal{D} :: \vdash M : f \ \text{ and } \ \overline{\mathcal{D}} =_{\beta\eta} N \quad \Rightarrow \quad (M, N) \in R^A_f$$

We prove a more general claim by induction on term M:

Theorem 14 *For all type $A \in \text{types}(o_2)$, token $f \in [\![A]\!]$, terms M of type A and N of type $\overline{f}$. For all substitutions of variables Γ and σ such that $\Gamma(x) = g \ \Rightarrow \ \sigma(x) \in R^B_g$ and $\mathsf{dom}(\Gamma) = \mathsf{FV}(M)$:*

$$\exists \mathcal{D} :: \ \Gamma \vdash M : f \ \wedge \ \overline{\mathcal{D}} =_{\beta\eta} N \quad \Rightarrow \quad (\, M.(\pi_1 \circ \sigma)\, ,\, N.(\pi_2 \circ \sigma)\,) \in R^A_f$$

where π_1 and π_2 are projections so that $\pi_1((M, N)) = M$ and $\pi_2((M, N)) = N$.

In order to prove this theorem, we first need to show that the logical relation is compatible with β-reduction (and η-expansion):

Lemma 13 For all type A and token $f \in [\![A]\!]$, for all terms M, N, M', N' such that $M =_{\beta\eta} M'$ and $N =_{\beta\eta} N'$: $(M, N) \in R^A_f \ \Rightarrow \ (M', N') \in R^A_f$.

Proof

We prove this lemma by induction on type A. Let M, N, M', N' be terms such that $M =_{\beta\eta} M'$, $N =_{\beta\eta} N'$ and $(M, N) \in R^A_f$.

If $A = o_2$ and $f = (p, \ell)$ then $p(M\!\downharpoonright_\beta) \overset{T_2}{=} N\!\downharpoonright_\beta$ and $\mathtt{A}_2(M\!\downharpoonright_\beta) = \ell$. So $p(M'\!\downharpoonright_\beta) = p(M\!\downharpoonright_\beta) \overset{T_2}{=} N\!\downharpoonright_\beta = N'\!\downharpoonright_\beta$ and $\mathtt{A}_2(M'\!\downharpoonright_\beta) = \mathtt{A}_2(M\!\downharpoonright_\beta) = \ell$. In that case $(M', N') \in R^A_f$.

If $A = B \to C$ and $f = g \multimap h$ then, for all $(M_1, N_1) \in R^B_g$, $(M\, M_1, N\, N_1) \in R^c_h$. Since $M =_{\beta\eta} M'$ and $N =_{\beta\eta} N'$, we have $(M\, M_1, N\, N_1) =_{\beta\eta} (M'\, M_1, N'\, N_1)$ and, by induction hypothesis on type C, $(M'\, M_1, N'\, N_1) \in R^C_h$. So $(M', N') \in R^{B \to C}_{g \multimap h}$. □

We can now prove theorem 14.

Proof

We use an induction on term M.

Let $A \in \text{types}(o_2)$, token $f \in [\![A]\!]$, terms M of type A and N of type $\overline{f}$. Let Γ and σ substitutions of variables such that $\Gamma(x) = g \ \Rightarrow \ \sigma(x) \in R^B_g$ and $\mathsf{dom}(\Gamma) = \mathsf{FV}(M)$. Let $\mathcal{D}$ a derivation of the judgement $\Gamma \vdash M : f$ (unique according to theorem 11). Assume that $\overline{\mathcal{D}} =_{\beta\eta} N$. We want to prove $(\, M.(\pi_1 \circ \sigma)\, ,\, N.(\pi_2 \circ \sigma)\,) \in R^A_f$.

In most cases, we will show that $(M.(\pi_1 \circ \sigma), \overline{\mathcal{D}}.(\pi_2 \circ \sigma)) \in R^A_f$ and conclude using lemma 13. We distinguish four cases depending on M, one for each derivation rule as head of derivation $\mathcal{D}$:

- If $M = x^A$ then the head rule of $\mathcal{D}$ is:

$$\frac{f \in [\![A]\!]}{x^A : f \ \vdash x^A : f}$$

Since $\Gamma(x^A) = f$, we have $\sigma(x^A) \in R_f^A$. So:

$$(M.(\pi_1 \circ \sigma), N.(\pi_2 \circ \sigma)) = (\pi_1(\sigma(x^A)), \pi_2(\sigma(x^A))) \in R_f^A$$

- If $M = M_1\, M_2$ then the head rule of $\mathcal{D}$ is:

$$\frac{\mathcal{D}_1 :: \Gamma_1 \vdash M_1 : f' \multimap f \qquad\qquad \mathcal{D}_2 :: \Gamma_2 \vdash M_2 : f'}{\Gamma_1, \Gamma_2 \vdash M_1\, M_2 : f}$$

  where $\Gamma = \Gamma_1, \Gamma_2$ such that the domains of Γ_1 and Γ_2 are the sets of free variables of M_1 and M_2 respectively. Similarly, we can split substitution σ into σ_1 and σ_2 in order to apply the induction hypothesis on $\mathcal{D}_1$ with σ_1 and on $\mathcal{D}_2$ with σ_2. Noting B the type of M_2 we get:

$$(M_1.(\pi_1 \circ \sigma_1), \overline{\mathcal{D}_1}.(\pi_2 \circ \sigma_1)) \in R_{f' \multimap f}^{B \to A} \qquad\qquad (M_2.(\pi_1 \circ \sigma_2), \overline{\mathcal{D}_2}.(\pi_2 \circ \sigma_2)) \in R_{f'}^{B}$$

  By definition of $R_{f' \multimap f}^{B \to A}$ we get $(M_1.(\pi_1 \circ \sigma_1) M_2.(\pi_1 \circ \sigma_2), \overline{\mathcal{D}_1}.(\pi_2 \circ \sigma_1) \overline{\mathcal{D}_2}.(\pi_2 \circ \sigma_2)) \in R_f^A$. So $((M_1\, M_2).(\pi_1 \circ \sigma), (\overline{\mathcal{D}_1}\, \overline{\mathcal{D}_2}).(\pi_2 \circ \sigma)) \in R_f^A$. Since $\overline{\mathcal{D}} = \overline{\mathcal{D}_1}\, \overline{\mathcal{D}_2}$, we conclude using lemma 13.

- If $M = \lambda x^B.M'$ then the head rule of $\mathcal{D}$ is:

$$\frac{\mathcal{D}' :: \Gamma, x^B : g \vdash M' : f'}{\Gamma \vdash \lambda x^B.M' : g \multimap f'}$$

  where $A = B \to C$ and $f = g \multimap f'$. First we show that $(\lambda x.M'.(\pi_1 \circ \sigma), \lambda x.\overline{\mathcal{D}'}.(\pi_2 \circ \sigma)) \in R_{g \multimap f'}^{B \to C}$. Let $(M_0, N_0) \in R_g^B$. In order to use the induction hypothesis we define $\Gamma' = \Gamma, x^B : g$ and the substitution $\sigma' = \sigma \circ [x \leftarrow (M_0, N_0)]$, then: $(M'.(\pi_1 \circ \sigma'), \overline{\mathcal{D}'}.(\pi_2 \circ \sigma')) \in R_{f'}^C$. Because of the definition of σ' we have: $(\lambda x.M'.(\pi_1 \circ \sigma))\, M_0 =_{\beta\eta} M'.(\pi_1 \circ \sigma')$ and
  $(\lambda x.\overline{\mathcal{D}'}.(\pi_2 \circ \sigma)) =_{\beta\eta} \overline{\mathcal{D}'}.(\pi_2 \circ \sigma')$. Using lemma 13 we deduce that
  $((\lambda x.M'.(\pi_1 \circ \sigma))\, M_0, (\lambda x.\overline{\mathcal{D}'}.(\pi_2 \circ \sigma))\, N_0) \in R_{f'}^C$. This proves that
  $(\lambda x.M'.(\pi_1 \circ \sigma), \lambda x.\overline{\mathcal{D}'}.(\pi_2 \circ \sigma)) \in R_{g \multimap f'}^{B \to C}$. We conclude using lemma 13.

- If $M = a$ then the head rule of $\mathcal{D}$ is:

$$\frac{p(a\, \overrightarrow{x})\langle \ell_1, \ldots, \ell_n \rangle \xrightarrow{T_2} N'\,(p_1\, x_1) \ldots (p_n\, x_n) \qquad\qquad \mathtt{A}_2(a\,(\ell_1, \ldots, \ell_n)) = \ell}{\vdash a : (p_1, \ell_1) \multimap \cdots \multimap (p_n, \ell_n) \multimap (p, \ell)}$$

  Since Γ is the empty substitution, we only need to prove $(M, N) \in R_{(p_1,\ell_1) \multimap \ldots \multimap (p,\ell)}^{o_2 \to \ldots o_2}$. In order to do this we define the property $\mathcal{P}(i)$ for $0 \leq i \leq n$ by:

$$\begin{aligned}\mathcal{P}(i) = \text{"For all } (M_1, N_1) \in R_{(p_1,\ell_1)}^{o_2}, \ldots, (M_i, N_i) \in R_{(p_i,\ell_i)}^{o_2},& \\ \text{we have } (M\, M_1 \ldots M_i, N'\, N_1 \ldots N_i) \in R_{(p_{i+1},\ell_{i+1}) \multimap \ldots\, (p_n,\ell_n) \multimap (p,\ell)}^{o_2 \to \ldots \to o_2}\text{"}&\end{aligned}$$

  We prove $\mathcal{P}(i)$ by downward induction for $0 \leq i \leq n$.

We start by proving $\mathcal{P}(n)$:
let $(M_1, N_1) \in R^{o_2}_{(p_1,\ell_1)}, \ldots, (M_n, N_n) \in R^{o_2}_{(p_n,\ell_n)}$. So for all $i \leq n$, we have $p_i(M_i\lfloor_\beta) \overset{T_2}{=} N_i\lfloor_\beta$ and $\mathtt{A}_2(M_i\lfloor_\beta) = \ell_i$. Now we look at $p(M\, M_1 \ldots M_n\lfloor_\beta)$:

$$\begin{aligned}
p((M\, M_1 \ldots M_n)\lfloor_\beta) &= p(a\,(M_1\lfloor_\beta) \ldots (M_n\lfloor_\beta)) \\
&\overset{T_2}{=} N'\,(p_1(M_1\lfloor_\beta)) \ldots (p_n(M_n\lfloor_\beta)) \\
&\overset{T_2}{=} N'\,(N_1\lfloor_\beta) \ldots (N_n\lfloor_\beta) \\
&\overset{T_2}{=} (N'\, N_1 \ldots N_n)\lfloor_\beta
\end{aligned}$$

Note that we can apply the rule of T_2 because we know that $\mathtt{A}_2(M_i\lfloor_\beta) = \ell_i$ for all $i \leq n$. Then we check $\mathtt{A}_2((M\, M_1 \ldots M_n)\lfloor_\beta)$:

$$\begin{aligned}
\mathtt{A}_2((M\, M_1 \ldots M_n)\lfloor_\beta) &= \mathtt{A}_2(a\,(M_1\lfloor_\beta) \ldots (M_n\lfloor_\beta)) \\
&= \mathtt{A}_2(a\,\ell_1 \ldots \ell_n) \\
&= \ell
\end{aligned}$$

We have shown $\mathcal{P}(n) =$ "$(M\, M_1 \ldots M_n, N'\, N_1 \ldots N_n) \in R^{o_2}_{(p,\ell)}$".

Next we prove the induction step, for $1 \leq j \leq n$, $\mathcal{P}(j) \Rightarrow \mathcal{P}(j-1)$: we assume $\mathcal{P}(j)$ and need to prove $\mathcal{P}(j-1)$.
Let $(M_1, N_1) \in R^{o_2}_{(p_1,\ell_1)}, \ldots, (M_{j-1}, N_{j-1}) \in R^{o_2}_{(p_{j-1},\ell_{j-1})}$. According to $\mathcal{P}(j)$, for all $(M_j, N_j) \in R^{o_2}_{(p_j,\ell_j)}$: $(M\, M_1 \ldots M_j, N'\, N_1 \ldots N_j) \in R^{o_2 \to \ldots \to o_2}_{(p_{j+1},\ell_{j+1}) \multimap \ldots \multimap (p,\ell)}$. So $(M\, M_1 \ldots M_{j-1}, N'\, N_1 \ldots N_{j-1}) \in R^{o_2 \to \ldots \to o_2}_{(p_j,\ell_j) \multimap (p_{j+1},\ell_{j+1}) \multimap \ldots \multimap (p,\ell)}$ and $\mathcal{P}(j-1)$ is true.

Therefore, by induction, $\mathcal{P}(0) =$ "$(M, N') \in R^{o_2 \to \ldots\, o_2}_{(p_1,\ell_1) \multimap \cdots \multimap (p,\ell)}$" is true. Since $N' = \overline{\mathcal{D}} =_{\beta\eta} N$ we can conclude that $(M, N) \in R^{o_2 \to \ldots\, o_2}_{(p_1,\ell_1) \multimap \cdots \multimap (p,\ell)}$ using lemma 13.

This ends the proof of theorem 14. □

As a corollary of theorem 14 we get that if there exists a derivation $\mathcal{D}$ of a judgement $\vdash M : f$ then $(M, \overline{\mathcal{D}}\lfloor_{\beta\eta}) \in R^A_f$.

With this corollary we can build the transducer which performs the composition of two transductions.

6.2.4 Construction of the transducer which realizes the composition

We recall the following notations $T_1 = (\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R_1, \mathtt{A}_1)$ and $T_2 = (\Sigma_P, \Sigma_2, \Sigma_3, p_0, R_2, \mathtt{A}_2)$ are two HOWDTR$_{\text{lin}}$, $Q = \{q_1, \ldots, q_m\}$ is the set of states of T_1 and, for every state $q_i \in Q$, we note A_{q_i} the type of $q_i(t)$ when t is a tree of type o_1. For all type A built on o_2, the set of tokens of terms of type A is noted $[\![A]\!]$ and is finite.

Previously, we saw how to apply transducer T_2 to terms M of type A built on the atomic type o_2, so we can apply T_2 to terms which appear on the left side of rules of T_1:

$$q(a\,\overrightarrow{x})\langle\overrightarrow{\ell}\rangle \to M\,(q_{i_1}\,x_1) \ldots (q_{i_n}\,x_n)$$

In a rule such as this one, in order to replace term M with term $N = \overline{\mathcal{D}}$ where $\mathcal{D}$ is the unique derivation of the judgement $\vdash M : f$, we need to know which token f properly describes the behavior of T_2 on M. The computation of that token is done in the look-ahead automaton $\mathtt{A}$ of T.

We define the set of states of $\mathtt{A}$ as:

$$L = L_1 \times [\![A_{q_1}]\!] \times \cdots \times [\![A_{q_m}]\!]$$

With any tree t (of type o_1) we want to associate the look-ahead of T_1 on t and, for each state $q_i \in Q$ of T_1, a token of $q_i(t)$. The transition function of the look-ahead automaton $\mathtt{A}$ is defined by, for all $(\ell_1, f_{1,1}, \ldots, f_{1,n}), \ldots, (\ell_n, f_{m,1}, \ldots, f_{m,n}) \in L$:

$$a\,(\ell_1, f_{1,1}, \ldots, f_{1,m}) \ldots (\ell_n, f_{n,1}, \ldots, f_{n,m}) \xrightarrow{\mathtt{A}} (\ell, f_1, \ldots, f_m)$$

where $a\,\ell_1 \ldots \ell_n \xrightarrow{\mathtt{A}_1} \ell$ and, for all state $q_i \in Q$, f_i is such that in T_1 there exists a rule $q_i(a\,\overrightarrow{x})\langle \ell_1, \ldots, \ell_n\rangle \xrightarrow{T_1} M\,(q_{i_1}\,x_1) \ldots (q_{i_n}\,x_n)$ and a derivation of the judgement $\vdash M : f_{1,i_1} \multimap \cdots \multimap f_{n,i_n} \multimap f_i$. Note that this look-ahead automaton is non-deterministic in general, but the transducer is weakly deterministic in the sense that, at each step, even if several look-ahead states are possible, only one rule of the transducer can be applied.

We define the set of states Q' of transducer T by:

$$Q' = \{(q, f) \mid q \in Q, f \in [\![A_q]\!]\} \cup \{q'_0\}$$

Then we define the set R of rules of transducer T as the set of rules of the form:

$$(q, f)(a\,\overrightarrow{x})\langle (\ell_1, f_{1,1}, \ldots, f_{1,m}), \ldots\rangle \xrightarrow{T} \overline{\mathcal{D}}\,((q_{i_1}, f_{1,i_1})\,x_1) \ldots ((q_{i_n}, f_{n,i_n})\,x_n)$$

such that there exists in T_1 a rule:

$$q(a\,\overrightarrow{x})\langle \ell_1, \ldots\rangle \xrightarrow{T_1} M\,(q_{i_1}\,x_1) \ldots (q_{i_n}\,x_n)$$

and $\mathcal{D}$ is a derivation of the judgement $\vdash M : f_{1,i_1} \multimap \cdots \multimap f_{n,i_n} \multimap f$.

Because of Theorem 11, that set of rules is weakly deterministic.

To that set R we then add rules for the initial state q'_0, which simply replicate the rules of states of the form $(q_0, (p_0, \ell))$: for all $a \in \Sigma_1$, all $(\ell_1, f_{1,1}, \ldots, f_{1,m}), \ldots, (\ell_n, f_{n,1}, \ldots, f_{n,m}) \in L$ and all rule in R of the form:

$$(q_0, (p_0, l))(a\,\overrightarrow{x})\langle (\ell_1, f_{1,1}, \ldots, f_{1,m}), \ldots\rangle \xrightarrow{T} M\,((q_1, f_1)\,x_1) \ldots ((q_n, f_n)\,x_n)$$

where p_0 is the initial state of T_2 and $l \in L_2$ is a state of the look-ahead automaton of T_2, we add the rule :

$$q'_0(a\,\overrightarrow{x})\langle (\ell_1, f_{1,1}, \ldots, f_{1,m}), \ldots\rangle \xrightarrow{T} M\,((q_1, f_1)\,x_1) \ldots ((q_n, f_n)\,x_n)$$

This set R of rules is still weakly deterministic according to Theorem 11.

We have thus defined the $\mathrm{HOWDTR}_{\mathrm{lin}}$ $T = (\Sigma_{Q'}, \Sigma_1, \Sigma_3, q'_0, R, \mathtt{A})$.

Example: Construction of the composed transducer

We show how to compute the transducer $T = T_2 \circ T_1$. We start with its look-ahead automaton $\mathtt{A}$. Since transducer T_1's look-ahead has only one state, we can ignore it, so we define our set of look-ahead states as: $L = [\![A_{q_0}]\!] \times [\![A_{q_1}]\!]$ with $A_{q_0} = o$ and $A_{q_1} = o \to o$. The set of tokens of type o is $[\![o]\!] = \{p_0, p_1\} \times \{\ell_a, \ell_b\}$, and the set of tokens of type $o \to o$ is $[\![o \to o]\!] = \{f \multimap g \mid f, g \in [\![o]\!]\}$. So we have 64 different look-ahead states.

We will not detail all the transition rules of the look-ahead automaton, but we give one example. Earlier we gave the derivation of the judgment:
$\vdash M : ((p_1, \ell_b) \multimap (p_0, \ell_a)) \multimap (p_0, \ell_a)$ with $M = \lambda y.y\,(b\,\#)$ appearing in the rule $q_0(b\ x) \to M\,(q_1\ x)$ of T_1. We can also derive, for the term $M' = (\lambda zy.z\,(b\,y))$ appearing in the rule $q_1(b\,x) \to M'\,(q_1\,x)$ of T_1, the judgment:
$\vdash M' : ((p_1, \ell_b) \multimap (p_0, \ell_a)) \multimap (p_1, \ell_b) \multimap (p_0, \ell_a)$. So we get, for all token $f \in [\![A_{q_0}]\!]$, the following rule of the look-ahead automaton $\mathtt{A}$ of T:

$$b(f,\,(p_1,\ell_b) \multimap (p_0,\ell_a)\,) \xrightarrow{\mathtt{A}} (\,(p_0,\ell_a)\,,\,(p_1,\ell_b) \multimap (p_0,\ell_a)\,)$$

This rule works for any value of f because, noting t the tree below b, f represents the behavior of $q_0(t)$ when processed by T_2 but, since $q_0(b\,t)$ and $q_1(b\,t)$ only depend on $q_1(t)$, the overall behavior of the tree $b\,t$ only depends on the behavior of $q_1(t)$, not on that of $q_0(t)$.
Some states are not accessible, for example the token $f = (p_0, \ell_a) \multimap (p_0, \ell_b)$ of type $o \to o$. Indeed a linear term of type $o \to o$ must use its argument, meaning its output must be a tree of type o containing the argument as a subtree. With an argument of token (p_0, ℓ_a) and therefore look-ahead ℓ_a, it is impossible that the output has token (p_0, ℓ_b) and look-ahead ℓ_b (due to the rules of $\mathtt{A}_2$). Token $(p_0, \ell_a) \multimap (p_1, \ell_a) \in [\![o \to o]\!]$ is impossible for similar reasons of accessibility between p_0 and p_1. Furthermore, because look-ahead states are q_0-token/q_1-token pairs, some pairs are not accessible. This way we get 24 accessible look-ahead states. This look-ahead automaton is non-deterministic, its determinized version has only 5 accessible states.
Then we have the set of states of T:

$$Q' = (\{q_0\} \times [\![o]\!]) \;\cup\; (\{q_1\} \times [\![o \to o]\!]) \;\cup\; \{q_0'\}$$

for a total of $4 + 16 + 1 = 21$ states.
Again we do not explicitly show each rule of T here, but we give one example. We have previously given the derivation $\mathcal{D} :: \vdash M : ((p_1, \ell_b) \multimap (p_0, \ell_a)) \multimap (p_0, \ell_a)$ with its collapsing $\overline{\mathcal{D}} = \lambda y.y\,\#$, and where M appears in the rule $q_0(b\ x) \to M\,(q_1\ x)$ of T_1. Therefore we have, for all token $f \in [\![o]\!]$, the following rule in T:

$$(q_0, (p_0, \ell_a))\,(b\,x)\langle(f, (p_1, \ell_b) \multimap (p_0, \ell_a)\,)\rangle \xrightarrow{T} (\lambda y.y\,\#)\,((q_1, (p_1, \ell_b) \multimap (p_0, \ell_a))\,x)$$

Again the token f associated with $q_0(b\,x)$ does not matter to the result of the rule, so the rule works for any value of $f \in [\![o]\!]$.

Example: Construction of the composed transducer

Because q_0 and p_0 are the initial states of T_1 and T_2, we also have the following rule for each token $f \in [\![o]\!]$:

$$q'_0\,(b\,x)\langle (f, (p_1, \ell_b) \multimap (p_0, \ell_a)\,)\rangle \xrightarrow{T} (\lambda y. y\,\#)\,((q_1, (p_1, \ell_b) \multimap (p_0, \ell_a))\,x)$$

Within this set of rules, we can remove unaccessible states of the form (q_0, f) because they do not appear on the right side of rules of T_1 and are not initial states of T. If we use the determinized version of the look-ahead automaton, we can also merge some states of the form (q_1, f). In the end we only have 8 states in the set Q' of states of T.

Note that the reductions to the number of states of look-ahead and states of T come from the coherence relations between the tokens. For example, look-ahead states $((p_1, \ell_b), (p_1, \ell_b) \multimap (p_1, \ell_b))$, $((p_1, \ell_b), (p_1, \ell_a) \multimap (p_1, \ell_a))$ and
$((p_0, \ell_b), (p_1, \ell_a) \multimap (p_1, \ell_a))$ can be merged partly because $(p_1, \ell_b) \mathrel{\overset{\frown}{\smile}} (p_0, \ell_b)$ and $(p_1, \ell_b) \multimap (p_1, \ell_b) \mathrel{\overset{\frown}{\smile}} (p_1, \ell_a) \multimap (p_1, \ell_a)$.
In fact, the known procedure for determinizing an automaton which consists in grouping states when they recognize the same tree, can only group together accessible states of the form $(\ell, f_1, \ldots, f_n)$ and $(\ell', f'_1, \ldots, f'_n)$ when, for each $i \leq n$, $f_i \mathrel{\overset{\frown}{\smile}} f'_i$. That is because f_i and f'_i describe the behavior of $q_i(t)$ where t is recognized by $(\ell, f_1, \ldots, f_n)$ and $(\ell', f'_1, \ldots, f'_n)$.

Now we prove the main theorem:

Theorem 15 $T = T_2 \circ T_1$

Proof
We first prove the following statement by induction on a tree t of type o_1:
For all state $q \in Q$ of transducer T_1 and for all token $f \in [\![A_q]\!]$ such that $q(t) \xrightarrow{T_1} M$ and $\vdash M : f$, there exists a term N such that $(q, f)(t) \xrightarrow{T} N$ and $(M, N) \in R_f^{A_q}$.

Let $t = a\,t_1 \ldots t_n$ a tree of type o_1, $q \in Q$ a state of T_1 and $f \in [\![A_q]\!]$ a token such that $q(t) \xrightarrow{T_1} M$ and $\vdash M : f$. Then there is a rule:

$$q(a\,t_1 \ldots t_n) \xrightarrow{T_1} M_0\,(q_1\,t_1) \ldots (q_n\,t_n)$$

If term M_0 forgets one or several of its arguments, then there exists a term M'_0 which uses all its arguments such that $M_0\,(q_1\,t_1) \ldots (q_n\,t_n) =_{\beta\eta} M'_0\,(q_{i_1}\,t_{i_1}) \ldots (q_{i_m}\,t_{i_m})$ where $i_1, \ldots, i_m$ are the indices of the arguments used by M_0. For the sake of clarity we forget this renaming of variables and proceed assuming M_0 uses all of its arguments.

Since the computation of $q(t) \xrightarrow{T_1} M$ terminates and M_0 uses all its arguments: for all $i \leq n$, the computation of $q_i(t_i)$ by T_1 terminates, we note its result M_i (a term of type A_{q_i}). Therefore $M_0\,M_1 \ldots M_n \rightarrow^*_{\beta\eta} M$. So $\vdash M_0\,M_1 \ldots M_n : f$ and there exists $f_1 \in [\![A_{q_1}]\!], \ldots, f_n \in [\![A_{q_n}]\!]$ such that $\vdash M_0 : f_1 \multimap \cdots \multimap f_n \multimap f$ and, for all $i \leq n$,

$\vdash M_i : f_i$. Then we can apply the induction hypothesis to each tree t_i with state q_i and token f_i: for all $i \leq n$, there is a term N_i such that $(q_i, f_i)(t_i) \xrightarrow{T} N_i$ and $(M_i, N_i) \in R^{A_{q_i}}_{f_i}$.

Because of the rule $q(a\,t_1 \dots t_n) \xrightarrow{T_1} M_0\,(q_1 t_1) \dots (q_n t_n)$ in T_1, there must be in T a rule:

$$(q, f)(a\,t_1 \dots t_n) \xrightarrow{T} \overline{\mathcal{D}_0}\,((q_1, f_1)(t_1)) \dots ((q_n, f_n)(t_n))$$

Where $\mathcal{D}_0$ is the derivation of the judgement $\vdash M_0 : f_1 \multimap \cdots \multimap f_n \multimap f$. So $(q, f)(a\,t_1 \dots t_n) \xrightarrow{T} \overline{\mathcal{D}_0}\,N_1 \dots N_n$.

By using theorem 14 (adequation) on $\mathcal{D}_0$ we get $(M_0, \overline{\mathcal{D}_0}) \in R^{A_{q_1} \to \dots A_{q_n} \to A_q}_{f_1 \multimap \dots f_n \multimap f}$. By definition of the logical relation, we obtain $(M_0\,M_1 \dots M_n, \overline{\mathcal{D}_0}\,N_1 \dots N_n) \in R^{A_q}_f$. Finally we apply lemma 13. So, with $N = \overline{\mathcal{D}_0}\,N_1 \dots N_n$, we have $(q, f)(t) \xrightarrow{T} N$ and $(M, N) \in R^{A_q}_f$.

Let t_1 be a tree of type o_1. Assume that $T_2 \circ T_1(t_1) = t_3$. Then there is a term t_2 of type o_2 such that $q_0(t_1) \xrightarrow{T_1} t_2$ and $p_0(t_2) \xrightarrow{T_2} t_3$. Then we can derive the judgement $\vdash t_2 : (p_0, \ell)$ where t_2 is recognized by look-ahead state ℓ of T_2 and p_0 is the initial state of T_2. So there exists a term N such that $(q_0, (p_0, \ell))(t_1) \xrightarrow{T} N$ and $(t_2, N) \in R^{o_2}_{(p_0, \ell)}$. By definition of the logical relation we have: $p_0(t_2) \overset{T_2}{=} N\lfloor_\beta$, so $t_3 = N\lfloor_\beta$ and $(q_0, (p_0, \ell))(t_1) \xrightarrow{T} t_3$. Thanks to the definition of R, we can conclude that $q_0'(t_1) \xrightarrow{T} t_3$. So $T_2 \circ T_1(t_1) = t_3$ implies that $T(t_1) = t_3$.

For the reverse implication, we first show by induction on tree t that, for all state $q \in Q$ and token $f \in \llbracket A_q \rrbracket$, if $(q, f)(t) \xrightarrow{T} N$ then there exists a term M such that $q(t) \xrightarrow{T_1} M$, $\vdash M : f$ and $(M, N) \in R^{A_q}_f$.

Let $t = a\ t_1 \dots t_n$ a tree of type o_1 with $(q, f)(t) \xrightarrow{T} N$. So there is a rule of T such that $(q, f)(t) \xrightarrow{T} N_0\,((q_1, f_1)(t_1)) \dots ((q_n, f_n)(t_n))$. Then there are $N_1, \dots, N_n$ such that $(q, f)(t) \xrightarrow{T} N_0\,N_1 \dots N_n$, $N =_{\beta\eta} N_0\,N_1 \dots N_n$ and, for all $i \leq n$, $(q_i, f_i)(t_i) \xrightarrow{T} N_i$. Then we apply the induction hypothesis and get M_i such that $q_i(t_i) \xrightarrow{T_1} M_i$ and $\vdash M_i : f_i$. There is in T_1 a rule $q(t) \xrightarrow{T_1} M_0\,(q_1 t_1) \dots (q_n t_n)$, so $q(t) \xrightarrow{T_1} M_0\,M_1 \dots M_n$, with $\vdash M_0 : f_1 \multimap \dots f_n \multimap f$. So for $M = M_0\,M_1 \dots M_n$ we have $\vdash M : f$. Finally we deduce that $(M, N) \in R^{A_q}_f$ using the property we proved earlier in this proof and the lemma 13.

Now we try to show that $T(t_1) = t_3 \ \Rightarrow\ T_2 \circ T_1(t_1) = t_3$. Assume that $T(t_1) = t_3$. Then $q_0'(t_1) \xrightarrow{T} t_3$, so there exists a token $(p_0, \ell) \in \llbracket o_2 \rrbracket$ such that $(q_0, (p_0, \ell))(t_1) \xrightarrow{T} t_3$. So there exists a term M such that $q(t_1) \xrightarrow{T_1} M$, $\vdash M : f$ and $(M, t_3) \in R^{o_2}_{(p_0, \ell)}$. Then, by definition of the logical relation: $p_0(M\lfloor_\beta) \xrightarrow{T_2} t_3$. So $T_2 \circ T_1(t_1) = t_3$.

So the transduction of T is the composition of the transductions of T_2 and T_1. □

Complexity analysis

We analyze the complexity of this algorithm and show that using the algorithm on $\text{HOWDTR}_{\text{lin}}$ instead of $\text{HODTR}_{\text{lin}}$ avoids, in the general case, an exponential blow-up of the size of the produced transducer.

First the set of states Q' of T is of size $|Q'| = 1+\Sigma_{q\in Q}|[\![A_q]\!]|$ where $|[\![A_q]\!]|$ is the number of tokens of type A_q. $|[\![A_q]\!]| = (|P|\,|L_2|)^{|A_q|}$ where $|P|$ is the number of states of transducer T_2, $|L_2|$ is the number of states of the look-ahead automaton of transducer T_2 and $|A_q|$ is the size of the type A_q. So the size of Q' is $O(\Sigma_{q\in Q}(|P|\,|L_2|)^{|A_q|})$, that is a polynomial in the size of T_2 to the power of the size of types of states of T_1.

It is important to note that the set $[\![A_q]\!]$ of tokens of type A_q is where HOWDTR$_{\text{lin}}$ and HODTR$_{\text{lin}}$ differ in their complexity: the deterministic alternative to the weakly deterministic T would require to store with the state not a single token, but a set of two-by-two coherent tokens, that would bring the size of Q' to $1 + \Sigma_{q\in Q}2^{|[\![A_q]\!]|}$ which would be exponential in the size of T_2 and doubly exponential in the size of types of T_1.

Then there is the look-ahead automaton: its set of states is $L = L_1 \times [\![A_{q_1}]\!] \times \cdots \times [\![A_{q_m}]\!]$. So the number of states is in $O(|L_1|\,(|P|\,|L_2|)^{\Sigma_{q\in Q}|A_q|})$. The size of the set of rules of the look-ahead automaton is in $O(\Sigma_{a^{(n)}\in\Sigma_1}|L|^{n+1})$ where n is the arity of the constant $a^{(n)}$.

Finally there is the set R of rules of T. For every judgement $\vdash M : f_{1,i_1} \multimap \cdots \multimap f_{n,i_n} \multimap f$, finding a derivation $\mathcal{D}$ of that judgement and computing the corresponding $\overline{\mathcal{D}}$ is in $O(|M|^2)$ time where $|M|$ is the size of M. The number of possible rules is in $O(\Sigma_{a^{(n)}\in\Sigma_1}(|Q'|)^{n+1})$. So computing R is done in time $O(|R|^2\,\Sigma_{a^{(n)}\in\Sigma_1}(|Q'|)^{n+1})$ where R is the set of rules of T_1. With a fixed input signature Σ_1, the time complexity of the algorithm computing T is a polynomial in the sizes of T_1 and T_2, with only the sizes of types of states of T_1 as exponents.

In total, the composition procedure for HOWDTR$_{\text{lin}}$ is exponential in the sizes of types of T_1 and the arities of input trees of T_1. The corresponding procedure on HODTR$_{\text{lin}}$ differs because it is doubly exponential in the sizes of types of T_1.

This complexity analysis has focused on the general case, but it is important to note that, as in the example we have presented, the number of states of both the look-ahead and the transducer can be greatly reduced in practice. Furthermore we have done so by using standard automata procedures like determinizing the look-ahead, removing unaccessible states, and minimizing by merging equivalent states. The minimization by merging equivalent states is particularly useful here because we tend to create a lot of rules whose right-hand side only differ in the tokens stored in the states. Although we have not quantified this reduction in the general case, we can see why our construction based on the coherence spaces of tokens can often lead to reductions. These reductions can reveal underlying properties of transducers, and they are only permitted by our approach which uses coherence spaces.

Composition for equivalent models of transducers

As the HODTR$_{\text{lin}}$ model generalizes other classes of transducers, namely MTT$^{R}_{sur}$, ATT$_{sur}$, STT and MSOT, it is possible to perform their composition in our setting. Thanks to results of Theorem 17, it is then possible to reduce the order of the result of the composition, and obtain a HODTR$_{\text{lin}}$ that can be converted back in those other models. This methods gives an important insight on the composition procedure for those other formalisms.

In comparison, the composition algorithms for equivalent classes of transductions are either not direct or very complex as they essentially perform composition and order reduction at once. For instance, composition of single used restricted MTT is obtained through MSO ([14]). The composition algorithm for Streaming Tree Transducers described in [2] is direct, but made complex by the fact that the algorithm hides this reduction of order.

Comparison of composition procedures for HODT

We have seen that the composition procedure for $\text{HOWDTR}_{\text{lin}}$ also works on $\text{HODTR}_{\text{lin}}$ by putting sets of two-by-two coherent tokens, instead of tokens, as states of the look-ahead automaton. This look-ahead automaton computes a flow analysis and predicts by which state an input tree will be processed. This can also be used in the composition of HODT in general. In this composition algorithm, the composition of two HODT would use the flow analysis in order to only compute the terms that are used to build the output. In a way a HODT built with this procedure would compute its output in a *call by name* fashion, while a HODT built using the procedure of subsection 6.1 computes in a *call by value* fashion. With the exception that this *call by name* transducer would still not have the complexity drawback of a classic *call by name* algorithm: the flow analysis of this transducer would allow it to know which values are needed and compute those only once.

This new composition procedure for HODT would have worse time complexity than the procedure of subsection 6.1, but the computed transducer could itself compute its output faster, i.e. by using less rules, than the transducer computed using the procedure of subsection 6.1. This gain would come at the cost of having a bigger look-ahead automaton.

Either of the composition procedures for HODT can also be compared to that of the similar (but not equivalent) model of High-level tree transducers [16], which is less direct as it goes through a reduction to iterated pushdown tree transducers and back.

Chapter 7

Equivalence with existing models

In this chapter we prove that $\mathrm{HODTR}_{\mathrm{lin}}$ and $\mathrm{HODTR}_{\mathrm{al}}$ are respectively equivalent to Monadic Second Order Transductions from trees to trees (MSOT) and to Monadic Second Order Transductions from trees to terms (i.e. trees with sharing) (MSOTS). In order to do so we prove several properties of $\mathrm{HODTR}_{\mathrm{lin}}$ and $\mathrm{HODTR}_{\mathrm{al}}$.

In section 7.1 we give a decomposition of simply-typed linear and almost linear terms which allows us to represent terms of high order with terms of order ≤ 3. In section 7.2 we outline a construction that transforms a transducer of $\mathrm{HODTR}_{\mathrm{lin}}$ or $\mathrm{HODTR}_{\mathrm{al}}$ into an equivalent linear or almost linear transducer of order ≤ 3. In section 7.3 we show that there are translations between $\mathrm{HODTR}_{\mathrm{lin}}$ of order 3 and attribute tree transducers with the *single use restriction* and between $\mathrm{HODTR}_{\mathrm{al}}$ of order 3 and attribute tree transducers. These two models are known to be respectively equivalent to MSOT and MSOTS [7], which allows us to conclude that $\mathrm{HODTR}_{\mathrm{lin}}$ and $\mathrm{HODTR}_{\mathrm{al}}$ are respectively equivalent to MSOT and to MSOTS.

7.1 Template decomposition

The central idea in the construction consists in decomposing λ-terms M into pairs $\langle M', \sigma\rangle$ where M' is a pure λ-term and σ is a substitution of variables with the following properties:

- $M =_\beta M'.\sigma$,
- the free variables of M' have at most order 1,
- for every variable x, $\sigma(x)$ is a closed λ-term,
- the number of free variables in M' is minimal.

In such a decomposition, we call the term M' a *template*.

In case M is of type A, linear or almost linear, it can be proven that M' can be taken from a finite set [21]. The linear case is rather simple, but the almost linear case requires some precaution as one needs first to put M in syntactically almost linear form and then make the decomposition. Though the almost linear case is more technical the finiteness argument is the same in both cases and is based on proof theoretical arguments in multiplicative linear logic which involves polarities in a straightforward way.

The linear case conveys the intuition of decompositions in a clear manner. One takes the normal form of M and then delineates the largest contexts of M, i.e. first order terms that are made only with constants and that are as large as possible. These contexts are then replaced by variables and the substitution σ is built accordingly. The fact that the contexts are chosen as large as possible makes it so that no introduced variable can have as argument a term of the form $x\, M_1 \dots M_n$ where x is another variable introduced in the process. Therefore, the new variables introduced in the process bring one negative atom and several (possibly 0) positive ones and all of them need to be matched with positive and negative atoms in the type of M as, under these conditions, they cannot be matched together. This explains why there are only finitely many possible templates for a fixed type.

Theorem 16 *For all type A built on tree signature Σ, the set of templates of closed linear (or almost linear) terms of type A is finite.*

We prove this theorem first on linear terms, then we extend the proof to almost linear terms.

7.1.1 Finiteness of linear templates

In order to show that the set of linear templates of a given type A is finite, we use notions and properties defined in [22]: the definitions of positive and negative subtype occurrences and subpremises in A and what it entails in the structure of terms of type A.

For any type A, we can label occurrences of subtypes in A as *positive* or *negative* using the following rules:

- A is positive, we note it A^+,
- if $B \rightarrow C$ is a positive subtype of A then B is negative and C is positive, we note it $(B^- \rightarrow C^+)^+$,
- if $B \rightarrow C$ is a negative subtype of A then B is positive and C is negative, we note it $(B^+ \rightarrow C^-)^-$.

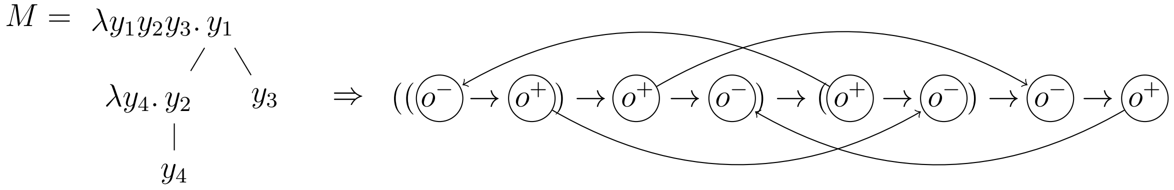
For example, if $A = ((o \rightarrow o) \rightarrow (o \rightarrow o)) \rightarrow ((o \rightarrow o) \rightarrow (o \rightarrow o))$ is a type built on the atomic tree type o, then we can label occurrences of subtypes of A as follows:

$$A = ((o^- \rightarrow o^+)^+ \rightarrow (o^+ \rightarrow o^-)^-)^- \rightarrow ((o^+ \rightarrow o^-)^- \rightarrow (o^- \rightarrow o^+)^+)^+$$

So, for all subtype occurrence $A' = A_1 \rightarrow \dots A_n \rightarrow o$, if A' is positive then $A_1^- \rightarrow \dots A_n^- \rightarrow o^+$, if A' is negative then $A_1^+ \rightarrow \dots A_n^+ \rightarrow o^-$.

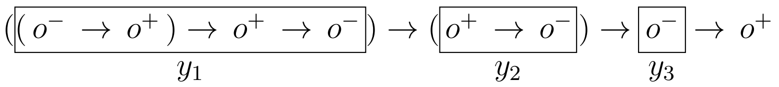
With any closed linear term M in β-normal form of type A we associate a bijection from the set of positive occurrences of the atomic type o in A to the set of negative occurrences of the atomic type o in A, we call it the *trace* of M and note it $\Theta(M)$.

We show how to compute $\Theta(M)$ on an example. To a term $M = \lambda y_1 y_2 y_3 . y_1\, (\lambda y_4 . y_2\, y_4)\, y_3$ of type $A = ((o^- \rightarrow o^+) \rightarrow o^+ \rightarrow o^-) \rightarrow (o^+ \rightarrow o^-) \rightarrow o^- \rightarrow o^+$ we have:



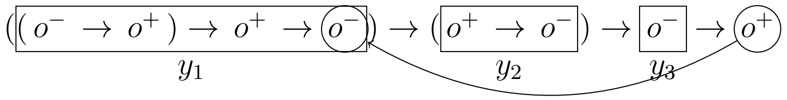
$M =$
$\lambda y_1 y_2 y_3.\, y_1$
$\lambda y_4.\, y_2$
y_3
y_4
$\Rightarrow$
$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$

The trace is computed by induction on M:
First M introduces y_1,y_2 and y_3:



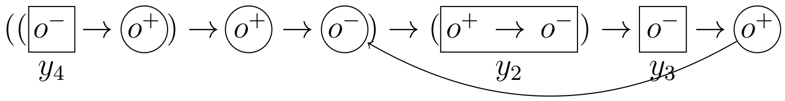
$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$
y_1
y_2
y_3

Then, because y_1 is the head variable of M, the output type of M corresponds to the output type of y_1:



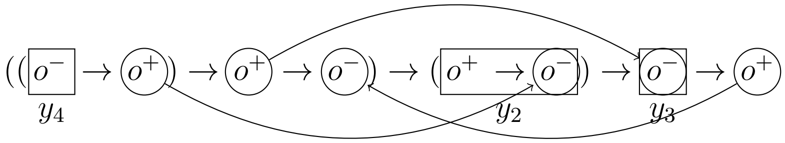
$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$
y_1
y_2
y_3

Then in the arguments of y_1 we introduce y_4 and we have two terms of type o^+ to match with output types o^- of variables:



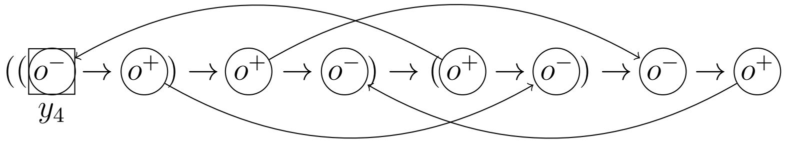
$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$
y_4
y_2
y_3

Those are mapped to y_2 and y_3:



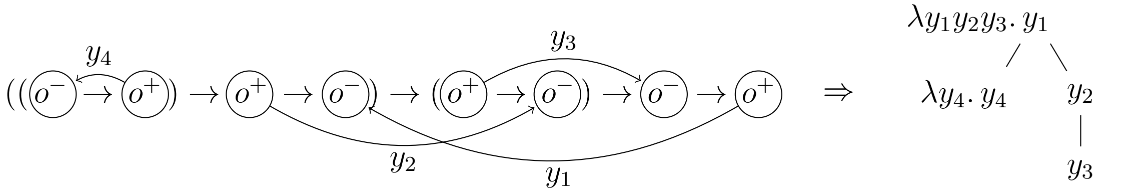
$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$
y_4
y_2
y_3

Finally the argument of y_2 is y_4:

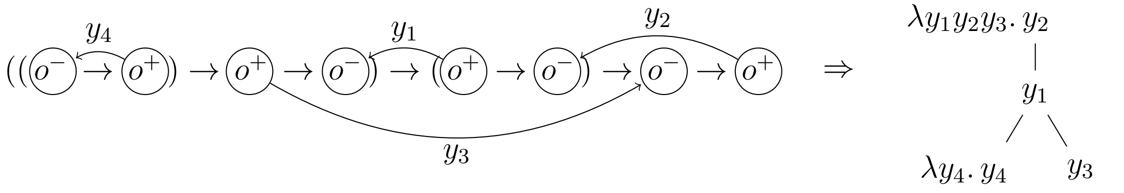


$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$
y_4

This is how we compute the trace of a linear term in linear normal form. The function which associates a trace with any linear term in linear normal form is injective, and it is possible, given a trace $\Theta(M)$, to compute the term M. For example:

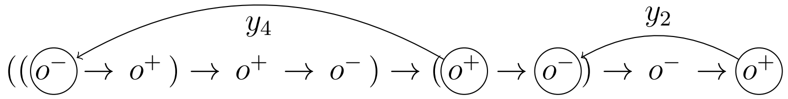


y_4
y_3
$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$
y_2
y_1
$\Rightarrow$
$\lambda y_1 y_2 y_3.\, y_1$
$\lambda y_4.\, y_4$
y_2
y_3

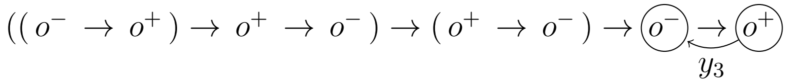


y_4
y_1
y_2
$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$
y_3
$\Rightarrow$
$\lambda y_1 y_2 y_3.\, y_2$
y_1
$\lambda y_4.\, y_4$
y_3

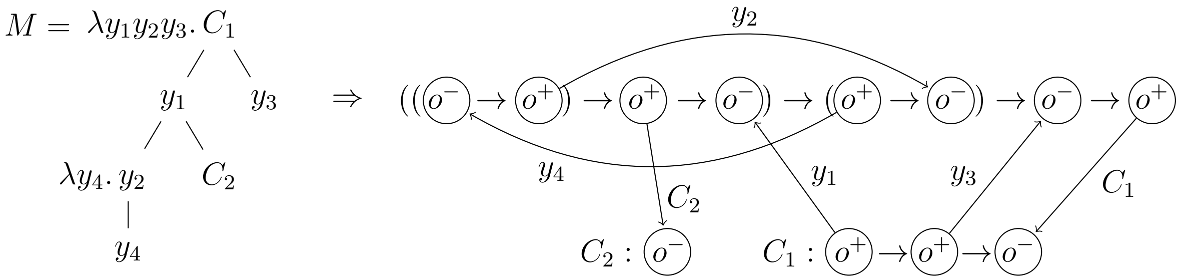
However injective, the Θ function is not surjective in general, meaning there are bijections from positive to negative atomic subtype occurrences that do not correspond to any term. For example, for type $A = ((o \to o) \to o \to o) \to (o \to o) \to o \to o$, there are only 3 terms in linear normal form of type A, and only 3 corresponding traces (the three examples we have shown so far). Any other bijection between positive and negative atomic subtype occurrences is not a trace either because it binds variable y_4 outside of its scope:


y_4
y_2
$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$

or because some variable would not appear in the term:


$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$
y_3

The consequence of this is that the number of closed linear terms in linear normal form of a given type A is bounded by the number of bijections between A's sets of positive and negative atomic subtype occurrences. In order to have a bound on the number of linear templates of a type, we extend the trace function from closed linear terms to linear terms with free variables which represent tree contexts, i.e. with type of the form $o^n \to o$. Again we show how it works on an example: the template $M = \lambda y_1 y_2 y_3 . C_1 \, (y_1 \, (\lambda y_4 . y_2 \, y_4) \, C_2) \, y_3$ with tree contexts C_1 and C_2 of respective types $o \to o \to o$ and o,


$M = \lambda y_1 y_2 y_3 . C_1$
y_1
y_3
$\lambda y_4 . y_2$
C_2
y_4
$\Rightarrow$
y_2
$((o^- \to o^+) \to o^+ \to o^-) \to (o^+ \to o^-) \to o^- \to o^+$
y_4
C_2
y_1
y_3
C_1
$C_2 : o^-$
$C_1 : o^+ \to o^+ \to o^-$

Naturally, the free variables provide new atomic subtype occurrences and the positivity and negativity of those are computed as if C_1 and C_2 were variables like y_2 and y_3. If a tree context is of the form $o^n \to o$ then it has 1 negative and n positive atomic subtype occurrences.

In order to show that the set of linear templates of a type is finite, we use the fact that templates are *minimal* decompositions: it means that there can not be a tree context that is directly applied to another tree context. This implies that, in the trace of a template, a positive atomic subtype occurrence of a tree context can not be mapped to a negative atomic subtype occurrence in a tree context. Since there is exactly one negative atomic subtype occurrence per tree context, the number of tree contexts in a template of type A is bounded by the number of positive atomic subtype occurrences in A. On the other hand, the number of positive atomic subtype occurrences in the tree contexts is bounded by the number of negative atomic subtype occurrences in A. So, for any given type A, the number of tree contexts of a linear template is bounded, the arity n of these tree contexts is bounded and, for each tree contexts setting, the number of traces (and therefore the number of templates) is bounded. Consequently, for all type A the number of linear templates of type A is bounded (by n^n where n is the size of type A).

7.1.2 Finiteness of almost linear templates

In the case of almost linear templates, we first define an almost linear normal form for terms that are equivalent to almost linear terms. For this we use results by M. Kanazawa [22] (2012) on almost affine lambda terms. Note that these results are applicable to both almost affine and almost linear terms. This report characterizes almost linear terms as terms that have the *negatively non-duplicated* property, consequently almost linear terms are terms that are both non-erasing (each bound variable is used at least once) and have the *negatively non-duplicated* property.

The other result of that paper we are using is a lemma (Lemma 8 page 13), which, for every negatively non-duplicated term M in η-long β-normal form, builds, through a deterministic procedure, an almost affine term M' that β-reduces to M. The way M' is computed from M is by successively factorizing variables y that are not of atomic type but occur at several places in M. For any such variable y, the *negatively non-duplicated* property implies that there are terms $N_1, \dots, N_m$ such that y always occurs in a term $y\, N_1 \dots N_m$ of atomic type in M; then there is a subterm M_y of M containing all occurrences of $y\, N_1 \dots N_m$, that term M_y is β-equivalent to the term $(\lambda y'.M'_y)\,(y\, N_1 \dots N_m)$ where $M'_y = M_y[y\, N_1 \dots N_m/y']$. By replacing M_y with $(\lambda y'.M'_y)\,(y\, N_1 \dots N_m)$ in M we remove the copying of the non atomic variable y and instead have the copying of variable y' which is of atomic type. By applying this process to every copied variable of non-atomic type in M we get the almost linear term M' β-equivalent to M.

With any term M equivalent to an almost linear term, we associate the almost linear term M' obtained by applying that process to the η-long β-normal form of M. Since two equivalent terms M_1 and M_2 have the same η-long β-normal form, they are associated with the same almost linear term M'. Therefore we have a normal form for all term that is equivalent to an almost linear term, we call it the almost linear normal form.

Once we have the almost linear normal form, we can apply the same reasoning as the one for linear templates. Because of the process of factorizing copied non-atomic variables, almost linear templates can be more complex than linear ones. But since the number of distinct non-atomic variables in a term M is bounded by the size of the type of M, the number of almost linear templates of a type A is bounded by $n_{templates} * (n_{fact})^{n_{var}}$ where $n_{templates}$ is the number of linear templates of type A, n_{fact} is a bound on the number of templatewise distinct possible factorizations of a non-atomic variable (i.e. two factorizations are templatewise distinct only if the templates of the factorized terms are distinct) and n_{var} is a bound on the number of non-atomic variables. We saw before that $n_{templates} \leq n^n$ where n is the size of the type A. The number of non-atomic variables is bounded by the size n of the type A. The template of a factorized term only depends on at which subterm M_y of M the factorization happens, and the number of templatewise distinct such M_y is bounded by the size of the template, so $n_{fact} \leq 2n$. Therefore the number of almost linear templates of a given type A of size n is bounded by n^{3n}.

We have proven that, for all type A, the sets of templates of linear and almost linear terms of type A are finite. Using this we can show how to compute, for each transducer of $\mathrm{HODTR}_{\mathrm{lin}}$ or $\mathrm{HODTR}_{\mathrm{al}}$, an equivalent transducer of order ≤ 3.

7.2 Order reduction

The template associated with a λ-term can be computed compositionally (i.e. from the templates of its parts). Since the sets of linear and almost linear templates are finite, templates can be computed by the look-ahead of a $\mathrm{HODTR}_{\mathrm{lin}}$ or of a $\mathrm{HODTR}_{\mathrm{al}}$. When reducing the order, we enrich the look-ahead with template information while the substitution that is needed to reconstruct the produced term is computed by the new transducer. The substitution is then performed by the initial state used at the root of the input tree which then outputs the same result as the former transducer. The substitution can be seen as a tuple of order 1 terms. It is represented as a tuple using Church encoding, i.e. a continuation. This makes the transducer we construct be of order 3.

Theorem 17 *Any $HODTR_{lin}$ (resp. $HODTR_{al}$) has an equivalent $HODTR_{lin}$ (resp. $HODTR_{al}$) of order* 3.

For this proof we will use the following notations: if a λ-term M is associated to the decomposition $\langle M', \sigma\rangle$ where M' is a template and σ a substitution of the free variables in M', then we note $\mathfrak{T}(M) = (M', (\sigma(y_1), \ldots, \sigma(y_n)))$ where $y_1, \ldots, y_n$ are the free variables in M. In this case we allow $=$ to mean *equal up to renaming of free variables*. For all type A we note $\mathfrak{T}_{lin}\langle A\rangle$ and $\mathfrak{T}_{al}\langle A\rangle$ the sets of linear and almost linear templates of terms of type A.

We prove this first for $\mathrm{HODTR}_{\mathrm{lin}}$ and then for $\mathrm{HODTR}_{\mathrm{al}}$.

7.2.1 Linear case of the order reduction

First we prove a lemma showing how to compute the template of a term from the templates of its components:

Lemma 14 Let $M[x_1, \ldots, x_n]$ be a linear term built on signature Σ_1 with typed free variables $x_1^{A_1}, \ldots, x_n^{A_n}$, let $t_1, \ldots, t_n$ be linear templates (for variables $x_1, \ldots, x_n$). Then there is a linear template t and tree contexts $C_1, \ldots, C_\ell$ with free variables $y_{1,1}, \ldots, y_{1,\ell_1}, \ldots, y_{n,1}, \ldots, y_{n,\ell_n}$ such that, for all linear terms $N_1, \ldots, N_n$ with $\mathfrak{T}(N_i) = (t_i, (C_{i,1}, \ldots, C_{i,\ell_i}))$ for all i :

$$\mathfrak{T}(M[x_1/N_1, \ldots, x_n/N_n]) = (t, (C_1, \ldots, C_\ell)[y_{i,j}/C_{i,j}]_{i\leq n, j\leq \ell_i})$$

Proof
For all $i \leq n$: $N_i =_{\beta\eta} t_i[y_{i,1}/C_{i,1}, \ldots, y_{i,\ell_i}/C_{i,\ell_i}]$, where $y_{i,1}, \ldots, y_{i,\ell_i}$ are the free variables of t_i, because $\mathfrak{T}(N_i) = (t_i, (C_{i,1}, \ldots, C_{i,\ell_i}))$. Then we define t and $(C_1, \ldots, C_\ell)$ as the template and tree-contexts of the λ-term $M[x_1/t_1, \ldots, x_n/t_n]$ on the signature $\Sigma_1 \cup \{y_{i,j}\}_{i\leq n, j\leq \ell_i}$ (it

is a tree signature because variables $y_{i,j}$ are tree-contexts and therefore of order at most 1). Consequently :

$$\begin{aligned} M[x_1/N_1, \ldots, x_n/N_n] &= M[x_1/t_1, \ldots, x_n/t_n][y_{i,1}/C_{i,1}, \ldots, y_{i,\ell_i}/C_{i,\ell_i}] \\ &= t[z_1/C_1, \ldots, z_\ell/C_\ell][y_{1,1}/C_{1,1}, \ldots, y_{n,\ell_n}/C_{n,\ell_n}] \end{aligned}$$

and so :

$$\mathfrak{T}(M[x_1/N_1, \ldots, x_n/N_n]) = (t, (C_1, \ldots, C_\ell)[y_{i,j}/C_{i,j}]_{i \leq n, j \leq \ell_i})$$

□

Now we can prove theorem 17 in the linear case:

Proof

Let $T = (\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R, \mathtt{A})$ be a HODTR$_{\text{lin}}$. We note L the set of states of $\mathtt{A}$. We want to define a HODTR$_{\text{lin}}$ $T' = (\Sigma_{Q'}, \Sigma_1, \Sigma_2, q'_0, R', \mathtt{A}')$ of order 3 equivalent to T.

We start by defining the look-ahead automaton $\mathtt{A}'$ and its set of states $L' = L \times \mathfrak{T}_{lin}\langle A_{q_0}\rangle \ldots \mathfrak{T}_{lin}\langle A_{q_m}\rangle$ where $A_{q_0}, \ldots, A_{q_m}$ are the output type of the states in Q and $\mathfrak{T}_{lin}\langle A\rangle$ is the set of linear templates of type A. So this look-ahead associates, with every input tree N, the look-ahead $\mathtt{A}$ on tree N and, for each state q_i, the template of $q_i(N)$. Lemma 14 shows how to compute the template of a term $M[x_1, \ldots, x_n]$ using the templates of $x_1, \ldots, x_n$, then we define the rules of $\mathtt{A}'$ accordingly so that, for all input tree N, the state of the look-ahead $\mathtt{A}'$ on tree N is $(l, t_0, t_1, \ldots, t_m)$ where l is the look-ahead of $\mathtt{A}$ on N and, for all $i \leq m$, t_i is the template of $q_i(N)$. We prove this by induction on the input tree, the induction step is a direct application of lemma 14.

Then we define the set of states Q' of T' : $Q' = \{(q_i, t) \mid q_i \in Q, t \in \mathfrak{T}_{lin}\langle A_{q_i}\rangle\} \cup \{q'_0\}$. We will now define the rules in R' so that, for all $q_i \in Q, t \in \mathfrak{T}_{lin}\langle A_{q_i}\rangle$ and for all input tree N : $(q_i, t)(N) = (C_1, \ldots, C_\ell)$ (using continuations to represent the tuple) such that $\mathfrak{T}(q_i(N)) = (t, (C_1, \ldots, C_\ell))$. For all state $(q_i, t) \in Q'$, input tree constant f of arity n, input tree variables $x_1, \ldots, x_n$ and their look-ahead states $l_1, \ldots, l_n$ in L and $l'_1, \ldots, l'_n$ in L', and for all rule in R of the form : $q_i(f\, x_1 \ldots x_n)\langle l_1, \ldots, l_n\rangle \to M[x_1, \ldots, x_n]$ where variable x_1 is processed by state q_{i_1}, x_2 by q_{i_2} and so on, we add the following rule in R' :

$$\begin{aligned} &(q_i, t)\,(f\, x_1 \ldots x_n)\langle l'_1, \ldots, l'_n\rangle \to \\ &\quad \lambda k.(q_{i_1}, t_1)\, x_1\, (\lambda y_{1,1}, \ldots y_{1,\ell_1}. \ \ldots (q_{i_n}, t_n)\, x_n\, (\lambda y_{n,1} \ldots y_{n,\ell_n}.k\, C_1 \ldots C_\ell) \ldots) \end{aligned}$$

This is a way of setting variables $y_{1,1}, \ldots, y_{1,\ell_1}$ to the tree contexts $(C_{1,1}, \ldots, C_{1,\ell_1}) = (q_{i_1}, t_1)\,(x_1)$, it is necessary because the alternative, i.e. using a projection on the tuple every time a tree context $C_{1,i}$ is used, would break linearity.

The output type of such a state (q_i, t) is $(A_1 \to \ldots A_\ell \to o) \to o$ where o is the atomic output tree type and A_i is the type of the i-th free variable of t, then, since the order of one of the A_i is at most 1, the order of the output type of (q_i, t) is at most 3. So the order of T' is at most 3.

Note that if state q_0 has output type o, the only template for that type is the term x where x is a free variable of type o. Then for the initial state q'_0 of output type o, we add special rules in R'. For all rule already in R' of the form : $(q_0, t)(f\, x_1 \ldots x_n)\langle \overrightarrow{\ell}\rangle \to (C_1)$

where (C_1) is the unary tuple of type $(o \to o) \to o$ containing the tree C_1 of type o, we add the rule : $q'_0(f\, x_1 \dots x_n)\langle \overrightarrow{\ell} \rangle \to C_1$.

For all $q_i \in Q, t \in \mathfrak{T}_{lin}\langle A_{q_i}\rangle$ and for all input tree N such that $\mathfrak{T}(q_i(N)) = (t, (C_1, \dots, C_\ell))$: $(q_i, t)(N) \to^*_{R'} (C_1, \dots, C_\ell)$; we prove this by induction on the input tree N. Again the induction is a direct application of Lemma 14.

Finally we conclude by applying this property to state $q_0 \in Q$ and template $x \in \mathfrak{T}_{lin}\langle o \rangle$, and replacing the first rule applied to (q_0, x) by the corresponding rule on q'_0. □

7.2.2 Almost linear case of the order reduction

We first prove the analog of lemma 14 for the almost linear case :

Lemma 15 Let $M[x_1, \dots, x_n]$ be an almost linear term on signature Σ_1 with typed free variables $x_1^{A_1}, \dots, x_n^{A_n}$, let $t_1, \dots, t_n$ be almost linear templates of $x_1, \dots, x_n$. Then there is an almost linear template t and tree contexts $C_1, \dots, C_\ell$ with free variables $y_{1,1}, \dots, y_{n,\ell_n}$ such that, for all almost linear terms $N_1, \dots, N_n$ with $\mathfrak{T}(N_i) = (t_i, (C_{i,1}, \dots, C_{i,\ell_i}))$ for all i :

$$\mathfrak{T}(M[x_1/N_1, \dots, x_n/N_n]) = (t, (C_1, \dots, C_\ell)[y_{i,j}/C_{i,j}]_{i \le n, j \le \ell_i})$$

Proof
The key to this proof is to notice that the property of being an almost linear λ-term is preserved by substitution of variables with almost linear λ-terms and by $\beta\eta$-equivalence. It ensures that the term $M[x_1/N_1, \dots, x_n/N_n]$ is $\beta\eta$-equivalent to an almost linear λ-term.

For all $i \le n$, $N_i =_{\beta\eta} t_i[y_{i,1}/C_{i,1}, \dots, y_{i,\ell_i}/C_{i,\ell_i}]$, where $y_{i,1}, \dots, y_{i,\ell_i}$ are the free variables of t_i, because $\mathfrak{T}(N_i) = (t_i, (C_{i,1}, \dots, C_{i,\ell_i}))$. Then we define t and $(C_1, \dots, C_\ell)$ as the template and tree-contexts of the λ-term $M[x_1/t_1, \dots, x_n/t_n]$ on the signature $\Sigma_1 \cup \{y_{i,j}\}_{i \le n, j \le \ell_i}$ (it is a tree signature because variables $y_{i,j}$ are tree-contexts and therefore of order at most 1). Consequently :

$$\begin{aligned} M[x_1/N_1, \dots, x_n/N_n] &= M[x_1/t_1, \dots, x_n/t_n][y_{i,1}/C_{i,1}, dots, y_{i,\ell_i}/C_{i,\ell_i}] \\ &= t[z_1/C_1, \dots, z_\ell/C_\ell][y_{1,1}/C_{1,1}, \dots, y_{n,\ell_n}/C_{n,\ell_n}] \end{aligned}$$

and so :

$$\mathfrak{T}(M[x_1/N_1, \dots, x_n/N_n]) = (t, (C_1, \dots, C_\ell)[y_{i,j}/C_{i,j}]_{i \le n, j \le \ell_i})$$

□

Then the order reduction theorem for almost linear transducers (theorem 17) is proven similarly to the linear case, but using lemma 15 as the almost linear extension of lemma 14.

7.3 Expressiveness of HODTR$_{\text{lin}}$ and HODTR$_{\text{al}}$

The proof of the order reduction theorem (theorem 17) shows that every HODTR$_{\text{lin}}$ (or HODTR$_{\text{al}}$) can be seen as mapping trees to tuples of contexts and combining these contexts in a linear (respectively almost linear) way. This understanding of $\mathrm{HODTR_{lin}}$ and $\mathrm{HODTR_{al}}$ transducers allows us to translate them into Attribute Tree Transducers with Single Use Restriction ($\mathrm{ATT_{sur}}$), and Attribute Tree Transducers (ATT) respectively. Then, using [7], we can conclude with the following expressivity result:

Theorem 18 *$HODTR_{lin}$ are equivalent to MSOT and $HODTR_{al}$ are equivalent to MSOTS.*

Note that the equivalence between $\mathrm{HODTR_{lin}}$ and MSOT could be obtained more easily through MTT^{R}_{sur} than $\mathrm{ATT_{sur}}$, but this way we would not get the equivalence between $\mathrm{HODTR_{al}}$ and MSOTS.

In subsection 7.3.1 we give a definition of ATT and $\mathrm{ATT_{sur}}$, and we give the result from [7] of equivalence with MSOT and MSOTS. In subsection 7.3.2 we show how to translate Attribute Tree Transducers into $\mathrm{HODTR_{al}}$, and we show that the Single-Use Restricted property on ATT leads to the linearity on $\mathrm{HODTR_{al}}$. In subsection 7.3.3 we prove the converse translation, conclude and discuss the implications of this characterization of the expressiveness of $\mathrm{HODTR_{lin}}$ and $\mathrm{HODTR_{al}}$.

7.3.1 Definition of ATT

Attribute grammars [17] are ways to formalize a class of syntax directed translations based on context-free grammar. They amount to equip a context-free grammar with semantics attributes that propagate along the abstract syntax tree. These semantics attributes are *synthesized* when their value is propagated bottom-up and *inherited* when they are propagated top-down.

Attribute tree transducers, as defined by [7, 17], correspond to the combination of a relabeling attribute grammar (REL) and an attribute grammar (ATT) whose attributes are trees. The relabeling simulates both the finite state control and the look-ahead automaton of usual transducers. In our setting, they can be seen as HODT with look-ahead whose rules are of the form $q(a\,x_1 \dots x_n) \to b\,q_1(x_1) \dots q_n(x_n)$, where $a \in \Sigma$, $b \in \Delta$ and a and b have the same arity. We call REL the class of transductions defined this way.

Although ATT were defined as attribute grammar by [7, 17], in this piece we will call them transducers and we give them a slightly different but equivalent definition. Formally, an ATT from the input alphabet Σ to the output alphabet Δ is a tuple $(\Sigma, \Delta, S, I, out, R, root)$ where:

- Σ is a ranked alphabet,
- Δ is a ranked alphabet,
- S and I are the finite sets of respectively *synthesized* and *inherited* attributes,
- $out \in S$ is the *meaning attribute*,
- R, the *rules*, is a function that maps elements a of Σ of arity n to equations of the form $(\alpha, i) = M(\alpha_1, i_1) \dots (\alpha_k, i_k)$ for every (α, i) in $(S \times \{0\} \cup I \times [1, n])$ where M is a linear λ-term of type $o^k \to o$ built on the signature Δ and where (α_j, i_j) are

pairwise distinct constants that have atomic type so that α_j is in $S \cup I$ and i_j is in $[0, n]$.

- *root*, the *initialization of inherited attributes* which maps elements a of Σ to equations of the form $(\alpha, 0) = M(\alpha_1, 0) \dots (\alpha_k, 0)$ for every α in I, where M is a linear λ-term of type $o^k \rightarrow o$ built on the signature Δ and, for all $j \leq k$, $(\alpha_j, 0)$ is a constant of atomic type and α_j is in $S \cup I$.

Example of ATT

We use the ranked alphabet $\Sigma = \{f^{(2)}, a^{(0)}, b^{(0)}\}$ with constant f of arity 2, and constants a and b of arity 0. We define the ATT $T = (\Sigma, \Sigma, S, I, m, R, \mathit{root})$, with:

- Σ as both input and output alphabet,
- Sets $S = \{s, m\}$ and $I = \{i\}$ of synthesized attributes and inherited attributes, with m as meaning attribute,
- The *rules* function R defined for each symbol in Σ.
  The equations of $R(a)$ are:
  - $s_0 = a$
  - $m_0 = i_0$

  For $R(b)$ we have:
  - $s_0 = b$
  - $m_0 = i_0$

  And for $R(f)$ we have:
  - $s_0 = s_1$
  - $m_0 = f\ m_1\, m_2$
  - $i_1 = s_2$
  - $i_2 = i_0$
- *root*, the initiation of the inherited attribute i is:
  - $i = s$

Intuitively, this transducer does a permutation of the leaves of its input tree without modifying the rest of the tree. We will later see how it works on an example.

Now given an input tree N built on signature Σ, we let V_N be the set of paths of N that is inductively defined by, for $N = a\, N_1 \dots N_n$: $V_N = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{i.u \mid u \in V_{N_i}\}$. For u in V_N, we write $N\ \lfloor_u$ for the subterm of N that is at path u and which is defined as

$N \downharpoonright_{\epsilon}= N$, $(a\, N_1 \ldots N_n) \downharpoonright_{iu}= N_i \downharpoonright_{u}$. For u in V_N, we let $lab_N(u)$ be the constant a in Σ such that $N \downharpoonright_{u}= a\, N_1 \ldots N_n$. Consider v in $V_{N\downharpoonright_u}$, we have that $(N \downharpoonright_{u}) \downharpoonright_{v}= N \downharpoonright_{uv}$. Therefore the operation that appends u in front of an element of $V_{N\downharpoonright_u}$ defines an injection from $V_{N\downharpoonright_u}$ into V_N that preserves the designated term.

The ATT associates a set of attributes with each element of V_N. Formally, it builds a set of equations whose left-hand side belong to $A(N) = (S \cup I) \times V_N$. We call the elements of $A(N)$ *attribute instances* or simply *attributes* of N when the context is clear. For $u \in V_N$, the subset $A_u(N) = \{(\alpha, u) \mid \alpha \in S \cup I\}$ is the set of attributes associated with N at path u. We use 0 to mean the empty path component ϵ, so attributes of the root are noted $(\alpha, 0)$ instead of (α, ϵ). For each attribute $(\alpha, v) \in A(N \downharpoonright_{u})$ we define $u.(\alpha, v)$ as the attribute $(\alpha, uv) \in A(N)$. Given a set of attribute instances S, we write $u.S$ for the set $\{(\alpha, uv) \mid (\alpha, v) \in S\}$. Then the following identity holds $u.A_v(N \downharpoonright_{u}) = A_{uv}(N)$.

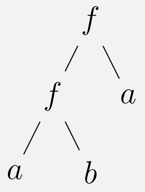
The ATT associates an equation with every attribute (α, u) of $A(N)$ as follows. If an equation $E_{(\alpha,i)} \in R(a)$ is of the form $(\alpha, i) = M(\alpha_1, i_1) \ldots (\alpha_n, i_n)$ then, for all path $u \in V_N$ such that $lab_N(u) = a$, the equation $(\alpha, u.i) = M(\alpha_1, u.i_1) \ldots (\alpha_n, u.i_n)$ is the equation for the attribute $(\alpha, u.i)$ and is noted $u.E_{(\alpha,i)}$. The operation $u.$ on equations naturally extends to sets of equations. We note $Eq_u(N)$ the set of equations $u.R(lab_N(u))$, and $Eq_{u\downarrow}(N)$ the set of equations $\bigcup_{v \in V_{N\downharpoonright_u}} Eq_{uv}(N)$. Then the set of equations associated with N (noted $Eq(N)$) is $Eq(N) = Eq_{\epsilon\downarrow}(N) = \bigcup_{u \in V_N} Eq_u(N)$. The *complete* set of equations of N (noted $CEq(N)$) is $CEq(N) = root(lab_N(\epsilon)) \cup Eq(N)$. We will also use the notation $CEq_{u\uparrow}(N)$ for the set $CEq(N) \setminus Eq_{u\downarrow}(N)$ for all $u \in V_N$.

We represent the way attributes depend on each other using graph as follows. With an equation $E_{(\alpha,i)} \in R(a)$ of the form $(\alpha, i) = M(\alpha_1, i_1) \ldots (\alpha_n, i_n)$ we associate the directed graph $G(E_{(\alpha,i)})$ whose set of vertices is $V = \{(\alpha, i), (\alpha_1, i_1), \ldots, (\alpha_n, i_n)\}$ and set of edges is $E = \{((\alpha, i), (\alpha_j, i_j)) \mid j \in [1, n]\}$. Define the operation of non-disjoint union of graphs whose sets of vertices are not necessarity disjoint as follows: for all graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. For all set Eq of equations we define the graph $G(Eq)$ associated with the set of equations Eq as $\bigcup_{E \in Eq} G(E)$. We extend the operation $u.$ on such graphs by: $u.G(Eq) = G(u.Eq)$. The *dependency graph* $D(N)$ of N and *complete dependency graph* $CD(N)$ of N are $G(Eq(N))$ and $G(CEq(N))$ respectively. Similarly, we will use the notations $D_{u\downarrow}(N)$ for the graph $G(Eq_{u\downarrow}(N))$ and $CD_{u\uparrow}(N)$ for the graph $G(Eq_{u\uparrow}(N))$.

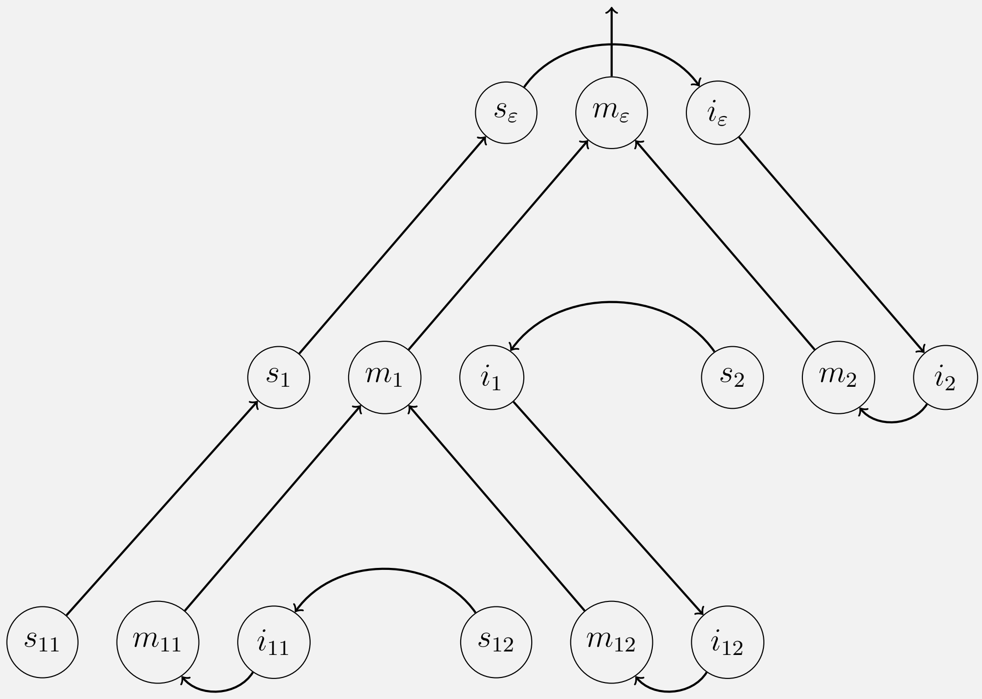
Note that in $D(N)$, there are no edges pointing to inherited attributes of the root node of N (attributes in $I \times \{\epsilon\}$).

Example: Running an ATT on an input

We run transducer T on the following input tree N:



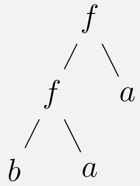
The paths of nodes in N are ε (the root) with label f, 1 with label f, 11 with label a, 12 with label b and 2 with label a.
In order to compute the output of T on the input N, we display the attributes of each node in N, and we represent each dependency between two attributes as an arrow (this is the graph noted $CD(N)$):



The meaning attribute is m so the output of the transducer is the value of m_ε. The value of m_ε is computed using the *rules* function R: since the label of the node at path ε in N is f, we apply $R(f)$. So $m_\varepsilon = f\, m_1\, m_2$.

To compute m_2 we look at the label of the node at path 2 in N, which is a. So we apply $R(a)$ and we get $m_2 = i_2$.
To compute i_2, since i is an *inherited* attribute, we should apply the rule of the parent node of the node at path 2, which is path ε. So we use $R(f)$ and get $i_2 = i_\varepsilon$.
To compute i_ε, the inherited attribute of the root, we use the *root* rule which gives $i_\varepsilon = s_\varepsilon$.

By following the sequence of equations we finally get to $s_{11} = a$. So $m_2 = a$. Similarly, we get $m_1 = f\, m_{11}\, m_{12}$ with $m_{11} = s_{12} = b$ and $m_{12} = s_2 = a$. Finally we obtain the output tree:



Note that the dependency relation between attributes always goes upward in the tree for synthesized attributes, but downward for inherited attributes.
The graph is acyclic and we can see why, when it is not acyclic, the output is not defined.

When $CD(N)$ is acyclic, the ATT is said *non-circular* on N and we note $\mathrm{Ord}(CD(N))$ the set of its topological sorts (i.e. the total orders which embed into the partial order on nodes induced by the acyclic graph $CD(N)$). In that case, we can associate with every attribute of N a tree built on Δ by applying the equations in $CEq(N)$. Indeed, a topological sort of the acyclic graph $CD(N)$ gives an order in which we can evaluate the attributes of N, i.e. associate them with a term built on Δ. The term associated to (out, ϵ) is the *output* of the ATT. An ATT is said *non-circular* when for every N, $CD(N)$ is acyclic. We note *ATT* the class of transductions that are defined by ATT. When for every N the dependency graph is a tree, the ATT is said *single use restricted*. We note $\mathrm{ATT}_{\mathrm{sur}}$ the class of transductions that are defined by single use restricted ATT.

Theorem 19 *[7] We have the following equivalences:*

- $\mathrm{REL} \circ ATT = MSOTS$,
- $\mathrm{REL} \circ ATT_{sur} = MSOT$,

7.3.2 REL ∘ ATT ⊆ HODTR$_{\mathbf{al}}$ and REL ∘ ATT$_{sur}$ ⊆ HODTR$_{\mathbf{lin}}$

In this section we prove that the composition of a relabeling attribute grammar with an ATT can be modeled by a $\mathrm{HODTR}_{\mathrm{al}}$, and that if the ATT is single use restricted then the translated $\mathrm{HODTR}_{\mathrm{al}}$ is a $\mathrm{HODTR}_{\mathrm{lin}}$.

For this proof we need to translate a set of equations on attributes into a computation of λ-terms. In a sense this is the step where we transform a logical framework of computation into a sequential framework of computation. The core of this proof consists in analyzing the structure of dependency graphs of attributes to deduce an order in which to compute the attributes. For this we require a few more definitions.

Definitions and notations

For each term N, we note $CD^{\top}(N)$ the graph obtained from $CD(N)$ by adding a fresh vertex noted $\top$ and an edge $((out, \epsilon), \top)$. Similarly, given $u \in V_N$, $CD^{\top}_{u\uparrow}(N)$ denotes the graph obtained from $CD_{u\uparrow}(N)$ by adding a fresh vertex $\top$ and an edge $((out, \epsilon), \top)$. We note $A^{\top}(N) = \{\top\} \cup A(N)$ and, for all path $u \in V_N$, $A^{\top}_u(N) = \{\top\} \cup A_u(N)$. Then the attributes we have to compute in order to compute the output of the ATT are the attributes connected to the vertex $\top$ in $CD^{\top}(N)$.

We use the convention that $u.\top = u^{-1}.\top = \top$.

Definition 25 For all graph $G = (V, E)$ and set V', we call *trace* of G on V', noted $\mathrm{tr}(G)|_{V'}$, the subgraph of the transitive closure of G induced by $V' \cap V$.

Lemma 16 For all graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and set V, if $V_1 \cap V_2 \subseteq V$, then $\mathrm{tr}(G_1 \cup G_2)|_V = \mathrm{tr}(\mathrm{tr}(G_1)|_V \cup G_2)|_V$.

Proof
The set of vertices of both $\mathrm{tr}(G_1 \cup G_2)|_V$ and $\mathrm{tr}(\mathrm{tr}(G_1)|_V \cup G_2)|_V$ is $V \cap (V_1 \cup V_2)$.

For all vertices x and y, if there is in $\mathrm{tr}(G_1)|_V$ a path from x to y there exists a path from x to y in G_1. Then for all path in $\mathrm{tr}(G_1)|_V \cup G_2$ from x to y there is a path from x to y in $G_1 \cup G_2$. So, for all edge (x, y) in the graph $\mathrm{tr}(\mathrm{tr}(G_1)|_V \cup G_2)|_V$ there is an edge (x, y) in $\mathrm{tr}(G_1 \cup G_2)|_V$.

Let $(x, y) \in V^2$ be an edge of the graph $\mathrm{tr}(G_1 \cup G_2)|_V$, then there is a path w from x to y in $G_1 \cup G_2$. This path can be written $w = w_1 \dots w_n$ where $w_1, \dots, w_n$ are paths in either G_1 or G_2 and, for all $i \leq n-1$, if w_i is a path in G_1 then w_{i+1} is a path in G_2 and if w_i is a path in G_2 then w_{i+1} is a path in G_1. Then, for all $i \leq n-1$, the end vertex of path w_i is in $V_1 \cap V_2$. Since $V_1 \cap V_2 \subseteq V$ and x and y are in V, all start and end vertices of paths $w_1, \dots, w_n$ are in V. Then for all path w_i in $\mathrm{tr}(G_1)|_V$ there is a path w'_i with same start and end vertices in the graph G_1. Therefore, noting $w'_i = w_i$ if w_i is a path in G_2 but not G_1 for all $i \leq n$, $w'_1, \dots, w'_n$ is a path from x to y in $\mathrm{tr}(G_1)|_V \cup G_2$. So there is an edge (x, y) in the graph $\mathrm{tr}(\mathrm{tr}(G_1)|_V \cup G_2)|_V$. □

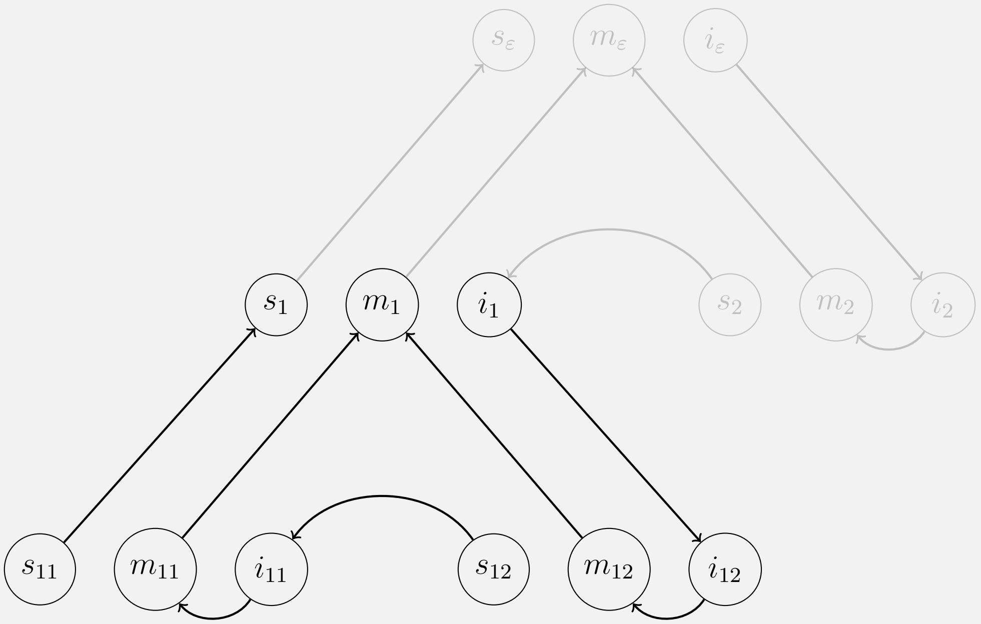
With this we can define the important notions of *synthesis graph* and *inheritance graph.*

Definition 26 For all tree path $u \in V_N$, we define the *synthesis graph* of path u in N, noted $GS_u(N)$, as the graph $u^{-1}.(\mathrm{tr}(D_{u\downarrow}(N))|_{A_u(N)})$.

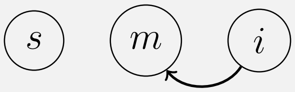
For all tree path $u \in V_N$, we call the *inheritance graph* of path u in N, noted $GI_u(N)$, the graph $u^{-1}.(\mathrm{tr}(CD^{\top}_{u\uparrow}(N))|_{V'})$ where V' is the subset of $A^{\top}_u(N)$ of vertices connected to the vertex $\top$ in the graph $CD^{\top}_{u\uparrow}(N)$.

Example of synthesis graph

Following up on the previous example of transducer T processing tree N, on the node at path 1 in N, we compute its synthesis and inheritance graphs.
To compute the synthesis graph $GS_1(N)$, we start by taking the restriction of the graph $D(N)$ to the attributes and edges below the node at path 1:



Then we simplify it by removing attributes of other nodes, but remembering the dependencies between attributes of our node at path 1, we obtain the synthesis graph $GS_1(N)$:



This is our way of remembering that, because of the dependencies through m_{12} and i_{12}, attribute m_1 indirectly depends on attribute i_1.

For any path u, the synthesis graph at path u represents the dependencies between attributes incurred from the equations *below* the node at path u. Next we try and show that synthesis graphs can be computed by a bottom-up tree automaton.

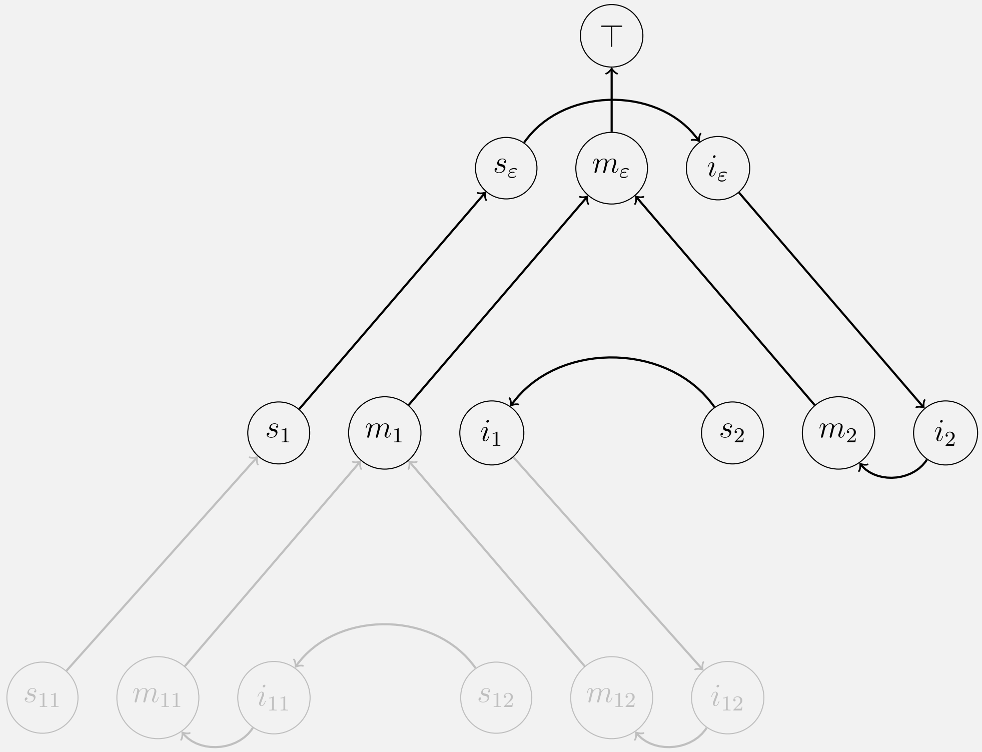
For every tree N and path $u \in V_N$, the sets of nodes of $GS_u(N)$ and $GI_u(N)$ are $A_\epsilon(N\restriction_u)$ and $A_\epsilon^\top(N\restriction_u)$ respectively, since these sets are not dependent on the tree N or the path u we simply note them $A_\epsilon = (S \cup I) \times \{\epsilon\}$ and $A_\epsilon^\top = \{\top\} \cup A_\epsilon$ respectively.

Lemma 17 For all $u \in V_N$, the edges of the graph $GS_u(N)$ are of the form $((\alpha, \epsilon), (\gamma, \epsilon))$ with $\alpha \in S \cup I$ and $\gamma \in S$.

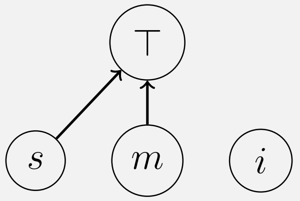
Lemma 18 For all $u \in V_N$, $GS_u(N) = \mathrm{tr}(G(R(a)) \bigcup_{1 \leq i \leq n} i.GS_{ui}(N))|_{A_\epsilon^\top}$ where n is the arity of the node at path u in N.

Example of inheritance graph

Similarly to the synthesis graph, for the inheritance graph $GI_1(N)$ we take the restriction of $CD^\top(N)$ to the attributes and edges which are not strictly below the node at path 1:



Again we simplify it by removing attributes of other nodes, but remembering the dependencies between attributes of our node at path 1, we obtain the inheritance graph $GI_1(N)$:



The goal of the inheritance graph is to remember not only how the inherited attributes depend on the synthesized attributes, but also how the output depends on the synthesized attributes.

Proof
We note G_0 the graph $\mathrm{tr}(G(R(a)) \cup \bigcup_{1\leq i\leq n} i.GS_{ui}(N))|_{A^\top_\epsilon}$. The graphs G_0 and $GS_u(N)$ have the same set of vertices A_ϵ.

Let (x, y) be an edge of the graph G_0, then, by definition of G_0, there is a path from $u.x$ to $u.y$ in the graph $\bigcup_{i\leq n} ui.GS_{ui}(N) \cup u.G(R(a))$ (this works because $u.$ is only a renaming of the attributes). By definition, any edge in $u.G(R(a))$ is in $D_{u\downarrow}(N)$. For all $i \leq n$ and for all edge (x_i, y_i) in $ui.GS_{ui}(N)$ there is a path in $D_{ui\downarrow}(N)$ from x_i to y_i, then this path also exists in the graph $D_{u\downarrow}(N)$. Then there is in the graph $D_{u\downarrow}(N)$ a path from $u.a$ to $u.b$. So the set of edges of G_0 is included in the set of edges of $GS_u(N)$.

Let (x, y) be an edge of $GS_u(N)$, then there is in the graph $D_{u\downarrow}(N)$ a path from $u.x$ to $u.y$. This path is of the form $w_1 e_1 w_2 \dots w_m e_m w_{m+1}$ where $e_1, \dots, e_m$ are edges in $u.G(R(a))$ and $w_1, \dots, w_{m+1}$ are paths with no edges in $u.G(R(a))$. Since $D_{u\downarrow}(N) = u.G(R(a)) \cup \bigcup_{1 \leq i \leq n} D_{ui\downarrow}(N)$ and the graphs $D_{ui\downarrow}(N)$ have disjoint sets of vertices, for all $j \leq m+1$ there is an index $i_j \leq n$ such that the path w_j is in the graph $D_{ui_j\downarrow}(N)$. Then for all $j \leq m+1$, noting x_j and y_j the respective start and end of path w_j, there is an edge (x_j, y_j) in the graph $ui_j.GS_{ui_j}(N)$. Then the path $(x_1, y_1)e_1 \dots e_m(x_{m+1}, y_{m+1})$ is in the graph $\bigcup_{i \leq n} ui.GS_{ui}(N) \cup u.G(R(a))$, with $u.x = x_1$ and $u.y = y_{m+1}$. So there is a path from x to y in the graph $\bigcup_{i \leq n} i.GS_{ui}(N) \cup G(R(a))$, therefore there is an edge (x, y) in the graph $\mathrm{tr}(\bigcup_{i \leq n} i.GS_{ui}(N) \cup G(R(a)))|_{A_\epsilon}$, so this edge is in G_0.

So $G_0 = GS_u(N)$. □

The synthesized and inheritance graph are important because they allow us to understand how attributes depends on each other, which is important when we want to sequentialize the computation of attributes. Since the synthesized graph at a given node only depends of the graph below that node, and because it is a graph with a fixed set A_ϵ of vertices, we can compute a bottom-up tree automaton which computes synthesis graphs recursively:

Lemma 19 There exists a bottom-up tree automaton $\mathtt{A}$, whose set of states is the set of directed acyclic graphs with set of vertices A_ϵ, which associates with any node in a tree N the graph $GS_u(N)$.

Proof
We define the bottom-up tree automaton $\mathtt{A} = (\Sigma_P, \Sigma_1, R_{\mathtt{A}})$ where P is the set of states of the form p_G where $G = (V, E)$ is a directed acyclic graph with $V = A_\epsilon$ and $E \subseteq \{((\alpha, \epsilon), (\gamma, \epsilon)) \mid \alpha \in S \cup I, \gamma \in S\}$, i.e. potential synthesis graphs according to lemma 17; and $R_{\mathtt{A}}$ is the set of rules of the form $a(p_{G_1} \dots p_{G_n}) \rightarrow p_{G_0}$ where a is a tree constant in Σ_1 of arity n, and G_0 is the graph $\mathrm{tr}(\bigcup_{i \leq n} i.G_i \cup G(R(a)))|_{A_\epsilon}$ where $G(R(a))$ is the graph induced by the equations of the attribute transducer associated with the tree constant a.

Lemma 18 implies by induction that automaton $\mathtt{A}$ indeed associates with any node at path u in N the synthesis graph $GS_u(N)$ of N at path u. □

Definition 27 For all tree path $u \in V_N$, The *interface graph* of N at path u (noted $G_u(N)$) is the directed acyclic graph $u^{-1}.(\,\mathrm{tr}(CD^\top(N))|_{V'})$ where V' is the subset of $A_u^\top(N)$ of vertices connected to the vertex $\top$ in the graph $CD^\top(N)$.

The interface graph at a given path u represents how the attributes of the node at path u depend on each other. The interface graph gives us the order (or orders) in which we can compute the attributes of a given node. We show that we can compute the interface graph from the synthesis graph and the inheritance graph:

Lemma 20 For all path $u \in V_N$, $G_u(N) = \text{tr}(GS_u(N) \cup GI_u(N))|_{V'}$ where V' is the subset of $A^\top_\epsilon$ of vertices connected to the vertex $\top$ in the graph $GS_u(N) \cup GI_u(N)$.

Proof
We note $G = \text{tr}(GS_u(N) \cup GI_u(N))|_{V'}$. We first prove the following claim:

Claim 1 For all $x, y \in A^\top_\epsilon$, there is a path from $u.x$ to $u.y$ in the graph $CD^\top(N)$ if and only if there is a path from x to y in the graph $GS_u(N) \cup GI_u(N)$.

Proof
Assume there is a path from $u.x$ to $u.y$ in $CD^\top(N)$. Since $CD^\top(N) = CD^\top_{u\uparrow}(N) \cup D_{u\downarrow}(N)$, this path can be seen as a sequence of paths $w_1 \ldots w_m$ alternating between graphs $CD^\top_{u\uparrow}(N)$ and $D_{u\downarrow}(N)$ (if w_i is a path in the graph $CD^\top_{u\uparrow}(N)$ then w_{i+1} is a path in $D_{u\downarrow}(N)$ and conversely). We note x_i and y_i the respective start and end of path w_i for all $i \leq m$. For all $i \leq m-1$, since the vertex $y_i = x_{i+1}$ is in both graphs $CD^\top_{u\uparrow}(N)$ and $D_{u\downarrow}(N)$, it must be in the set $A_u(N)$. Then there is an edge (x_i, y_i) in either $\text{tr}(CD^\top_{u\uparrow}(N))|_{A_u(N)}$ or $\text{tr}(D_{u\downarrow}(N))|_{A_u(N)}$ for all $i \leq m$. Because $x = u^{-1}.x_1$ and $y = u^{-1}.y_m$, there is in the graph $GS_u(N) \cup GI_u(N)$ a path from x to y.

Assume there is a path from x to y in $CD^\top(N)$. That path is of the form $(x_1, x_2)(x_2, x_3) \ldots (x_m, x_{m+1})$ where, for each $i \leq m$, (x_i, x_{i+1}) is an edge of either $GS_u(N)$ or $GI_u(N)$. So, for all $i \leq m$, there is either in $CD^\top_{u\uparrow}(N)$ or in $D_{u\downarrow}(N)$ a path from $u.x_i$ to $u.x_{i+1}$. Therefore we have in the graph $CD^\top(N)$ a path from $u.x = u.x_1$ to $u.y = u.x_{m+1}$.
□

This claim applied with $y = \top$ implies that G and $G_u(N)$ have the same sets of vertices.

The claim also implies that (x, y) is an edge of G if and only if (x, y) is an edge of $G_u(N)$.

So $G = G_u(N)$ for all path $u \in V_N$. □

With this, for each node of the input tree, we can compute its *interface graph* which expresses the dependencies between its attributes. Now we need to express precisely how the synthesized attributes are computed from the inherited attributes, and how those expressions are transmitted from a node to its parent. For this we will need to compute a larger *interface graph* which includes the dependencies between a node's attributes and its child nodes' attributes. We call this graph the *local dependency graph*.

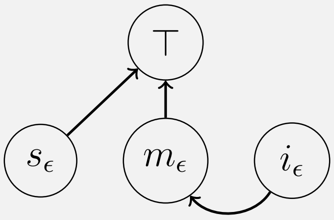
We will use the notations $A_{[1,n]} = \bigcup_{1 \leq j \leq n}(S \cup I) \times \{j\}$, $A_{[0,n]} = \bigcup_{0 \leq j \leq n}(S \cup I) \times \{j\}$ (with the convention that $0 = \epsilon$), $A^\top_{[1,n]} = \{\top\} \cup A_{[1,n]}$ and $A^\top_{[0,n]} = \{\top\} \cup A_{[0,n]}$.

Definition 28 For all path $u \in V_N$, we define the *local dependency graph* of N at path u, noted $G_{u.[0,n]}(N)$ where n is the arity of $lab_N(u)$, as the graph $u^{-1}.\text{tr}(CD^\top(N))|_{V'}$ where V' is the set of vertices in $u.A^\top_{[0,n]}$ that are connected to the vertex $\top$ in the graph $CD^\top(N)$.

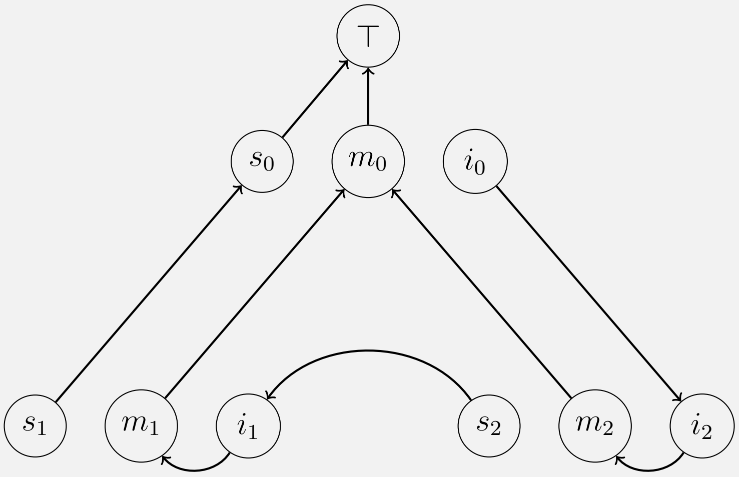
This *local dependency graph* can be computed from the synthesis graph at path u and the inheritance graphs at paths $u1, \ldots, un$.

Example of interface graph and local dependency graph

Following up on our previous example, the interface graph $G_1(N)$ of N at path 1 is:



We can see that it is the union of the synthesis graph and the inheritance graph, as proven in lemma 20.
The local dependency graph $G_{1.[0,2]}(N)$ of N at path 1 is:



Lemma 21 For all tree N and path $u \in V_N$, noting $a = lab_N(u)$ the constant of the node at path u in N and n its arity, the local dependency graph $G_{u.[0,n]}(N)$ of N at path u is $\mathrm{tr}(GI_u(N) \cup G(R(a)) \cup \bigcup_{1\leq j\leq n} j.GS_{uj}(N))|_{V'}$ where V' is the set of vertices in $A^\top_{[0,n]}$ that are connected to the vertex $\top$ in the graph $GI_u(N) \cup G(R(a)) \cup \bigcup_{1\leq j\leq n} j.GS_{uj}(N)$.

Proof
We first prove the following claim:

Claim 2 For all vertices $x, y \in A^\top_{[0,n]}$, there is in the graph $CD^\top(N)$ a path from $u.x$ to $u.y$ if and only if there is a path from $u.x$ to $u.y$ in the graph $G = \mathrm{tr}(CD^\top_{u\uparrow}(N))|_{A^\top_u(N)} \cup u.G(R(a)) \cup \bigcup_{1\leq i\leq n} \mathrm{tr}(D_{ui\downarrow}(N))|_{A_{uj}(N)}$.

Proof
If there is a path from $u.x$ to $u.y$ in G then, because $CD^\top(N) = CD^\top_{u\uparrow}(N) \cup u.G(R(a)) \cup \bigcup_{1\leq i\leq n} D_{ui\downarrow}(N)$, there must be a path from $u.x$ to $u.y$ in $CD^\top(N)$.

If there is a path from $u.x$ to $u.y$ in $CD^\top(N)$, then this path can be seen as a sequence $w_1 \dots w_m$ of paths where each w_j with $j \leq m$ is a path in either one of the following $n+2$

graphs: $CD^{\top}_{u\uparrow}(N), D_{u1\downarrow}(N), \dots D_{un\downarrow}(N), u.G(R(a))$, and, for each $j \leq m-1$, w_{j+1} is a path in a different graph than w_j. Noting x_j the end of path w_j or start of path w_{j+1}, since w_j and w_{j+1} are paths of a different graph among $CD^{\top}_{u\uparrow}(N), D_{u1\downarrow}(N), \dots D_{un\downarrow}(N)$ and $u.G(R(a))$, x_j is in the intersection of the sets of vertices of these two graphs, which is necessarily included in the set $u.A^{\top}_{[0,n]}(N)$. $x_0 = u.x$ and $x_m = u.y$ are also in the set $u.A^{\top}_{[0,n]}(N)$. This implies that if w_j is a path in $CD^{\top}_{u\uparrow}(N)$ then there is in $\mathrm{tr}(CD^{\top}_{u\uparrow}(N))|_{A^{\top}_{u}(N)}$ a path w'_j from x_{j-1} to x_j. Also if w_j is a path in $D_{ui\downarrow}(N)$ then there is in $\mathrm{tr}(D_{ui\downarrow}(N))|_{A_{uj}(N)}$ a path w'_j from x_{j-1} to x_j. So there is in the graph $G = \mathrm{tr}(CD^{\top}_{u\uparrow}(N))|_{A^{\top}_{u}(N)} \cup u.G(R(a)) \cup \bigcup_{1\leq i\leq n} \mathrm{tr}(D_{ui\downarrow}(N))|_{A_{uj}(N)}$ a path from $u.x$ to $u.y$.
□

Since $G_{u.[0,n]}(N) = u^{-1}.\mathrm{tr}(CD^{\top}(N))|_{u.A^{\top}_{[0,n]}}$ and $\mathrm{tr}(GI_u(N) \cup G(R(a)) \cup \bigcup_{1\leq j\leq n} j.GS_{uj}(N))|_{A^{\top}_{[0,n]}} = u^{-1}.\mathrm{tr}(G)|_{V'}$, the claim implies that the set of vertices of the graph $G_{u.[0,n]}(N)$ is the set V' of vertices in $A^{\top}_{[0,n]}$ that are connected to the vertex $\top$ in the graph $GI_u(N) \cup G(R(a)) \cup \bigcup_{1\leq j\leq n} j.GS_{uj}(N)$.

It also entails that, for all vertices $x, y \in V'$, there is in the graph $G_{u.[0,n]}(N)$ an edge (x, y) if and only if (x, y) is an edge in the graph $\mathrm{tr}(GI_u(N) \cup G(R(a)) \cup \bigcup_{1\leq j\leq n} j.GS_{uj}(N))|_{V'}$.

Therefore $G_{u.[0,n]}(N) = \mathrm{tr}(GI_u(N) \cup G(R(a)) \cup \bigcup_{1\leq j\leq n} j.GS_{uj}(N))|_{V'}$. □

Corollary 20 The local dependency graph $G_{u.[0,n]}(N)$ can be computed using only the constant $lab_N(u)$, the inheritance graph of N at path u and the synthesis graphs of N at paths $u1, \dots, un$.

Furthermore, we can show that, when the ATT is *single use restricted*, the local dependency graphs are trees:

Lemma 22 If $CD^{\top}(N)$ is a tree then, for all path $u \in V_N$, $G_{u.[0,n]}(N)$ is a tree.

Proof
We use *ad absurdum* reasoning. We assume that $G_{u.[0,n]}(N)$ is not a tree, so there exists two nodes x, y and two distinct paths from x to y in $G_{u.[0,n]}(N) = u^{-1}.\mathrm{tr}(CD^{\top}(N))|_{V'}$. Then there are two distinct paths from $u.x$ to $u.y$ in $CD^{\top}(N)$, then $CD^{\top}(N)$ is not a tree.
□

Corollary 21 If the ATT is *single use restricted* then for all input tree N and path $u \in V_N$, the graph $G_{u.[0,n]}(N)$ is a tree.

Topological sorts

In order to sequentialize the computation of attributes, we use topological sorts of the graphs of dependency prevously defined, specifically *interface graphs* and *local dependency graphs*. We will later need to use induction on the sorted attributes, in order to facilitate that, we define our topological sorts as sequences of attributes:

Definition 29 We call a total order $<$ on a finite set V *compatible with* a directed acyclic graph $G = (V, E)$ if, for all edge $(v, v') \in E$, $v < v'$.

Noting n the size of the set V, for all sequence $\tau = v_1 \dots v_n \in V^*$ of length n such that $i \neq j \Rightarrow v_i \neq v_j$ for all $1 \leq i, j \leq n$, we associate with τ the unique total order $<$ on V such that $i < j \Leftrightarrow v_i < v_j$ for all $1 \leq i, j \leq n$.

We call a sequence $\tau \in V^*$ a *topological sort* of a directed acyclic graph $G = (V, E)$ if it is of length n and the total order $<$ associated with it is compatible with G.

Lemma 23 For all directed acyclic graph G we can build a topological sort τ of G.

Proof
We build τ inductively. We note $G = (V, E)$.

Since G is acyclic there exists a vertex x of G which has no incoming edges. We use induction and assume we can build a topological sort τ' of the subgraph of G induced by the set $V \setminus \{x\}$ of vertices. Then $\tau = x\tau'$ is a topological sort of G. □

Lemma 24 For all path $u \in V_N$ any topological sort τ of $G_u(N)$ is of the form $\tau = \tau'(\alpha, \epsilon)\top$ with $\alpha \in S$.

Proof
By definition of $G_u(N)$, from any vertex of $G_u(N)$ there is a path to $\top$, so a topological sort of $G_u(N)$ must end with $\top$. The form of the rules of the attribute transducer imply that if there is a path in $G_u(N)$ from (γ, ϵ) to $\top$ with $\gamma \in I$ then there must exists $\alpha \in S$ and a path in the graph $G_u(N)$ from (α, ϵ) to (γ, ϵ). So any topological sort of $G_u(N)$ ends with $(\alpha, \epsilon)\top$ for some $\alpha \in S$. □

Each *local dependency graph* contains several *interface graphs*. We now express the relations between *local dependency graphs*, *interface graphs* and their topological sorts.

Definition 30 For all sets V and V' such that $V' \subset V$, for all graph $G = (V, E)$ and topological sort τ of G, we call *topological subsort induced* by the subset V', and we note $\tau|_{V'}$, the biggest subsequence of τ included in V'^*.

Lemma 25 For all directed acyclic graph $G = (V, E)$, topological sort τ of G and subset V' of V, $\tau|_{V'}$ is a topological sort of $\mathrm{tr}(G)|_{V'}$.

Proof
We note $G' = \mathrm{tr}(G)|_{V'} = (V', E')$. Let (a, b) be an edge in E', then there is a path in the graph G from a to b of the form $a\, v_1 \dots v_m\, b$. So, noting $<_\tau$ the total order on V associated with τ, $a <_\tau v_1 <_\tau \cdots <_\tau v_m <_\tau b$. Therefore $a <_\tau b$, and a appears in the sequence τ strictly before b. Then a appears in the sequence $\tau|_{V'}$ strictly before b, and $a <_{\tau'} b$ where $<_{\tau'}$ is the total order on V' associated with $\tau|_{V'}$.

We have shown that $<_{\tau'}$ is compatible with G', so $\tau|_{V'}$ is a topological sort of $\mathrm{tr}(G)|_{V'}$. □

Lemma 26 For all directed acyclic graph $G = (V, E)$, subset V' of V and topological sort τ' of $\mathrm{tr}(G)|_{V'}$, there exists a topological sort τ of G such that $\tau' = \tau|_{V'}$.

To prove this we need the following lemma:

Lemma 27 For all directed acyclic graph $G = (V, E)$, and subset $V' \subseteq V$ of vertices, and for all two vertices $x, y \in V'$, noting $\mathrm{tr}(G)|_{V'} = (V', E')$ the subgraph of the transitive closure of G induced by the subset V' of vertices, if the graph $(V', E' \cup \{(x, y)\})$ is acyclic then the graph $(V, E \cup \{(x, y)\})$ is also acyclic.

Proof
We use ad absurdum reasoning. We assume that the graph $(V', E' \cup \{(x, y)\})$ is acyclic and that there is a cycle in the graph $(V, E \cup \{(x, y)\})$. Since (V, E) is acyclic the edge (x, y) is part of the cycle, so the cycle is of the form $(x, y)(y, x_1) \dots (x_n, x)$ with vertices $x_1, \dots, x_n \in V$. Then there is a path from y to x in G, therefore there is an edge (y, x) in $\mathrm{tr}(G)|_{V'}$, so $(y, x) \in E'$. Then $(V', E' \cup \{(x, y)\})$ is not acyclic, which leads to a contradiction. □

Now we can prove lemma 26:

Proof
We note $x_1, \dots, x_n$ the vertices in V' such that $\tau' = x_1 \dots x_n$, and $\mathrm{tr}(G)|_{V'} = (V', E')$. We note $E_{\tau'}$ the set of edges $E_{\tau'} = \{(x_i, x_j)\}_{1 \leq i < j \leq n}$, then we show that the graph $G' = (V', E' \cup E_{\tau'})$ is acyclic.

If G' contained a cycle, it would imply that there was in (V', E') a path from x_j to x_i with $i < j$, which is contradicts the fact that $\tau' = x_1 \dots x_n$ is a topological sort of (V', E').

Since $(V', E' \cup E_{\tau'})$ is acyclic, we can use lemma 27 and deduce that $(V, E \cup E_{\tau'})$ is also acyclic. Then there exists a topological sort τ of $(V, E \cup E_{\tau'})$. Because $E_{\tau'} = \{(x_i, x_j)\}_{1 \leq i < j \leq n}$ and by definition of topological sorts: $\tau|_{V'} = \tau'$. Also τ is a topological sort of G. □

Definition 31 For all graphs G and $\tilde{G}$ with the same set of vertices, we say that $\tilde{G}$ is an *over-specification* of G, and we note $\tilde{G} \trianglerighteq G$, if all topological sort of $\tilde{G}$ is a topological sort of G.

Lemma 28 The relation $\trianglerighteq$ has the following properties:

1. for all graphs G_1, G_2 and G_3: $G_1 \trianglerighteq G_2 \trianglerighteq G_3 \Rightarrow G_1 \trianglerighteq G_3$ (transitivity),
2. for all graphs $G = (V, E)$ and $\tilde{G} = (V, \tilde{E})$: $E \subseteq \tilde{E} \Rightarrow \tilde{G} \trianglerighteq G$,
3. for all graph $G = (V, E)$: $G \trianglerighteq \mathrm{tr}(G)|_V \trianglerighteq G$,
4. for all graphs G and $\tilde{G}$ and set V': $\tilde{G} \trianglerighteq G \Rightarrow \mathrm{tr}(\tilde{G})|_{V'} \trianglerighteq \mathrm{tr}(G)|_{V'}$
5. for all graphs $G_1, G_2, \tilde{G}_1$ and $\tilde{G}_2$, if $\tilde{G}_1 \trianglerighteq G_1$ and $\tilde{G}_2 \trianglerighteq G_2$ then $\tilde{G}_1 \cup \tilde{G}_2 \trianglerighteq G_1 \cup G_2$

Proof

1. Implied by the definition of $\trianglerighteq$.

2. Implied by the definition of topological sorts.

3. The previous point implies that $\mathrm{tr}(G)|_V \trianglerighteq G$. For all topological sort τ of G, by transitivity of the order associated with τ, τ is also a topological order of the transitive closure $\mathrm{tr}(G)|_V$ of G. So $G \trianglerighteq \mathrm{tr}(G)|_V$.

4. For all topological sort τ' of $\mathrm{tr}(\tilde{G})|_{V'}$, according to lemma 26, there is a topological sort τ of $\tilde{G}$ such that $\tau|_{V'} = \tau'$. Then τ is also a topological sort of G and, according to lemma 25, $\tau' = \tau|_{V'}$ is a topological sort of $\mathrm{tr}(G)|_{V'}$.

5. Let us assume that $\tilde{G}_1 \trianglerighteq G_1$ and $\tilde{G}_2 \trianglerighteq G_2$ with $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. For all topological sort τ of $\tilde{G}_1 \cup \tilde{G}_2$, according to lemma 25, $\tau|_{V_1}$ and $\tau|_{V_2}$ are topological sorts of $\tilde{G}_1$ and $\tilde{G}_2$ respectively. So $\tau|_{V_1}$ and $\tau|_{V_2}$ respectively are topological sorts of G_1 and G_2. So τ is a topological sort of $G_1 \cup G_2$. Therefore $\tilde{G}_1 \cup \tilde{G}_2 \trianglerighteq G_1 \cup G_2$.

□

Lemma 29 For all graphs G_1 and G_2 such that $G_2 \trianglerighteq G_1$ and G_2 is closed by transitivity, then G_2 can be obtained from G_1 by adding edges.

Proof

We note $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$. Then G_2 can be obtained from G_1 by adding edges if and only if $E_1 \subseteq E_2$. We use *ad absurdum* reasoning and assume there is an edge $(x, y) \in E_1 \setminus E_2$. We note $V_{y\downarrow} = \{z \mid z \in V, (z, y) \in E_2\}$ and $V_x = V \setminus (\{y\} \cup V_{y\downarrow})$. So $x \in V_x$. Let $\tau_{y\downarrow}$ and τ_x be topological sorts of the acyclic graphs $\mathrm{tr}(G_2)|_{V_{y\downarrow}}$ and $\mathrm{tr}(G_2)|_{V_x}$ respectively. We now prove that $\tau = \tau_{y\downarrow} y \tau_x$ is a topological sort of G_2: for all $z_1, z_2 \in E_2$,

- if $(z_1, z_2) \in V_{y\downarrow} \times \{y\}$ then $z_1 <_\tau z_2$ because we put $\tau_{y\downarrow}$ before y in τ,
- if $(z_1, z_2) \in V_{y\downarrow} \times V_x$ then $z_1 <_\tau z_2$ because we put $\tau_{y\downarrow}$ before τ_x in τ,
- if $(z_1, z_2) \in \{y\} \times V_x$ then $z_1 <_\tau z_2$ because we put y before τ_x in τ,
- if $(z_1, z_2) \in V_{y\downarrow}^2$ then $z_1 <_{\tau_{y\downarrow}} z_2$ entails $z_1 <_\tau z_2$,
- the case $(z_1, z_2) \in \{y\}^2$ is impossible because G_2 is acyclic,
- if $(z_1, z_2) \in V_x^2$ then $z_1 <_{\tau_x} z_2$ entails $z_1 <_\tau z_2$.
- the case $(z_1, z_2) \in \{y\} \times V_{y\downarrow}$ is impossible because $z_2 \in V_{y\downarrow} \Rightarrow (z_2, y) \in E_2$ and G_2 is acyclic,
- the case $(z_1, z_2) \in V_x \times V_{y\downarrow}$ is impossible because the transitivity of G_2 would imply that $(z_1, y) \in E_2$, which contradicts the fact that $z_1 \notin V_{y\downarrow}$,
- the case $(z_1, z_2) \in V_x \times \{y\}$ also contradicts $z_1 \notin V_{y\downarrow}$.

So τ is a topological sort of G_2. But since $(x, y) \in E_1$ and $y <_\tau x$, τ is not a topological sort of G_1. That is in contradiction with the fact that $G_2 \trianglerighteq G_1$. □

We have previously shown that synthesis graphs can be computed, in a bottom-up way, by a tree automaton. From these synthesis graphs we show how to compute topological sorts of the corresponding interface graphs in a top-down way.

Lemma 30 We can build a function f such that, for all path $u \in V_N$ where the tree constant $a = lab_N(u)$ is of arity n and for all topological sort τ_0 of $G_u(N)$:
$\tau = f(a, \tau_0, (GS_{u1}(N), \dots, GS_{un}(N)))$ is a topological sort of $G_{u.[0,n]}(N)$ and, for each $j \leq n$, $j^{-1}.(\tau|_{A_j^\top})$ is a topological sort of $G_{uj}(N)$.

Proof
We note V' the set of vertices of the graph $G_u(N)$. For all tree constant a of arity n, for all topological sort τ_0 over a subset of $A_\epsilon^\top$ and for all synthesis graphs $G_1, \dots, G_n$ (acyclic graphs with vertices in $A_\epsilon^\top$ and edges included in $((S \cup I) \times \{\epsilon\}) \times (S \times \{\epsilon\})$ as per lemma 17), we define $f(a, \tau_0, (G_1, \dots, G_n))$ as the topological sort τ, obtained using lemma 23, of the graph $G = \mathrm{tr}(\bigcup_{1 \leq i \leq n} i.G_i \cup G(R(a)) \cup G_{\tau_0})|_{V''}$ where G_{τ_0} is the graph with set of vertices V' and set of edges $E_{\tau_0} = \{(x, y) \mid x <_{\tau_0} y\}$, and V'' is the set of vertices in $A_{[0,n]}^\top$ that are connected to the vertex $\top$ in the graph $\bigcup_{1 \leq i \leq n} i.G_i \cup G(R(a)) \cup G_{\tau_0}$.

In order to use lemma 23 we need to prove that G is acyclic. By construction, τ_0 is the only topological sort of G_{τ_0}. Since τ_0 is a topological sort of $G_u(N)$, $G_{\tau_0} \trianglerighteq G_u(N)$. According to lemma 20 $G_u(N) = \mathrm{tr}(GS_u(N) \cup GI_u(N))|_{A_\epsilon^\top}$, so $G_u(N) \trianglerighteq GS_u(N) \cup GI_u(N) \trianglerighteq GI_u(N)$. Then, according to lemma 29, G_{τ_0} can be obtained from $GI_u(N)$ by adding edges. So G can be obtained from $\bigcup_{1 \leq i \leq n} i.G_i \cup G(R(a)) \cup GI_u(N)$ by adding edges. By adding these same edges to $G_u(N) = \mathrm{tr}(\bigcup_{1 \leq i \leq n} i.G_i \cup G(R(a)) \cup GI_u(N))|_{A_\epsilon^\top}$ we get G_{τ_0}, which is acyclic, so according to lemma 27 G is acyclic too.

Since $\top$ is not a vertex in the graph $\bigcup_{1 \leq i \leq n} i.G_i \cup G(R(a))$ and the vertices of G_{τ_0} are the vertices of $G_u(N)$, the set V'' of vertices connected to $\top$ in $\bigcup_{1 \leq i \leq n} i.G_i \cup G(R(a)) \cup G_{\tau_0}$ is also the set of vertices connected to $\top$ in the graph $\bigcup_{1 \leq i \leq n} i.G_i \cup G(R(a)) \cup GI_u(N)$. Then, according to lemma 21, $G_{u.[0,n]}(N) = \mathrm{tr}(\bigcup_{1 \leq i \leq n} i.G_i \cup G(R(a)) \cup GI_u(N))|_{V''}$. So G can be obtained from $G_{u.[0,n]}(N)$ by adding edges, therefore $G \trianglerighteq G_{u.[0,n]}(N)$. So τ is a topological sort of $G_{u.[0,n]}(N)$. □

Lemma 31 For all path $u \in V_N$ any topological sort τ of $G_{u.[0,n]}(N)$ is of the form $\tau = \tau'(\alpha, \epsilon)\top$ with $\alpha \in S$.

Proof
Similar to proof of lemma 24. □

From now on, when we introduce a topological sort τ over a subset of $A_\epsilon^\top$ or $A_{[0,n]}^\top$, we assume it is of the form described in lemmas 24 and 31.

Sequentializing the computation of attributes

For all input tree N and path $u \in V_N$, a topological sort of the interface graph $G_u(N)$ gives an order in which the attibutes can be computed. This order determines the type of the λ-term that will be the image of the subtree $N \downharpoonleft_u$. That type is defined as follows:

Definition 32 For all topological sort τ over a subset of $A_\epsilon^\top$ (of the form described in lemma 24), we associate with τ the type $t(\tau)$ inductively defined by:

- if τ is of the form $(\alpha, \epsilon)\, \tau'$ with $\alpha \in S$ then $t(\tau) \triangleq o \times t(\tau')$,
- if τ is of the form $(\alpha, \epsilon)\, \tau'$ with $\alpha \in I$ then $t(\tau) \triangleq o \rightarrow t(\tau')$,
- if $\tau = (\alpha, \epsilon) \top$ where $\alpha \in S$, then $t(\tau) \triangleq o$.

For all input tree N and path $u \in V_N$, we want to associate a λ-term with the subtree $N \downharpoonleft_u$ of N which sequentializes the computation of the attributes of the node at path u. In order to do so we use a topological sort of the interface graph at path u in N, with the following semantics:

Definition 33 For all topological sort τ over the set $A_\epsilon^\top$, term N and path $u \in V_N$, noting $\mathit{Att}(N, (\alpha, u))$ the tree associated with the attribute (α, u) in the ATT, we define $\mathcal{R}_\tau(N, u)$ by induction on τ:

- $\mathcal{R}_{(\alpha,u)\tau'}(N, u) \triangleq \{(M_1, M_2) \mid M_1 \rightarrow^*_{\beta\eta} \mathit{Att}(N, (\alpha, u)), M_2 \in \mathcal{R}_{\tau'}(N, u)\} \quad$ if $\alpha \in S$,
- $\mathcal{R}_{(\alpha,u)\tau'}(N, u) \triangleq \{M \mid M(\mathit{Att}(N, (\alpha, u))) \in \mathcal{R}_{\tau'}(N, u)\} \quad$ if $\alpha \in I$.
- $\mathcal{R}_{(\alpha,\epsilon)\top}(N, u) \triangleq \{M \mid M \rightarrow^*_{\beta\eta} \mathit{Att}(N, (\alpha, u))\} \quad$ where $\alpha \in S$.

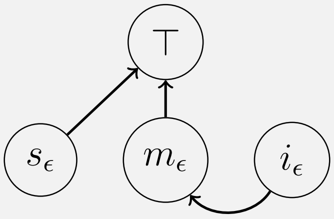
Intuitively, $\mathcal{R}_\tau(N, u)$ is the set of terms which perform the part of the computation done inside the subtree of N at path u.

Note that terms in $\mathcal{R}_\tau(N, u)$ have type $t(\tau)$.

In this section we use the tuple notation for λ-terms as syntactic sugar. This notation can be expressed in the simply-typed λ-calculus using continuation-passing. So the use of tuples does not makes this result less general.

Example of topological sort and associated λ-term

Following up on our previous example, we had the interface graph $G_1(N)$:



We use the topological sort $\tau = i_\epsilon\, m_\epsilon\, s_\epsilon\, \top$ which is compatible with $G_1(N)$. It represents a valid order in which to compute the attributes. The corresponding type is:

$$t(\tau) = o \to (o \times o)$$

An example of term performing the part of the computation which happens inside the subtree of N at path 1 is:

$$M = \lambda i.(f\, b\, i\, , a) \in \mathcal{R}_{(\alpha,\epsilon)\top}(N, u)$$

Term M represents the computation done by the subtree $f\, a\, b$: it takes as input a tree i which represents the inherited attribute i_ϵ, it produces a tree $f\, b\, i$ which represents the synthesized attribute m_ϵ, and a tree a which represents the synthesized attribute s_ϵ.

Note that $\mathcal{R}_{(\alpha,\epsilon)\top}(N, u)$ is a set of terms because we consider λ-terms up-to $\beta\eta$-equivalence.

Lemma 32 For all terms M and M' that are $\beta\eta$-equivalent,

$$M \in \mathcal{R}_\tau(N, u) \Leftrightarrow M' \in \mathcal{R}_\tau(N, u)$$

Proof

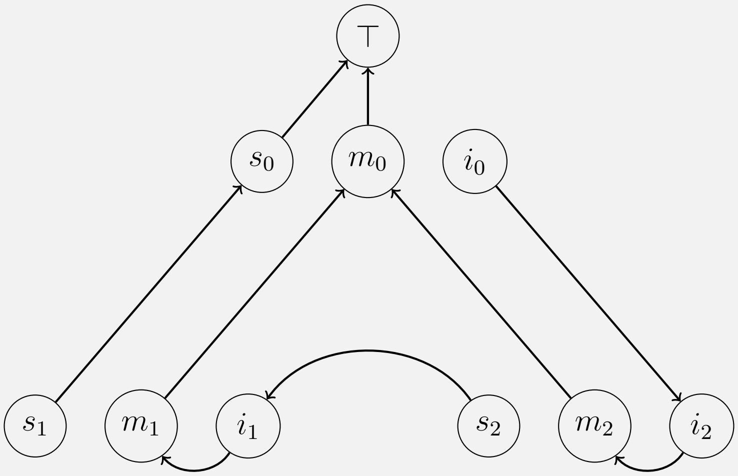
Straightforward induction on τ. □

For the purpose of clarity, we will use a special notation for the binding of variables: for binding a variable x to a term M inside a term M', in place of $(\lambda x.M')\, M$ we will write $\mathsf{let}\ x = M\ \mathsf{in}\ M'$.

Now we want to build a λ-term which computes the term associated with a node depending on the terms associated with its child nodes. This term depends on a topological sort of the local dependency graph which gives an order to compute the attributes of the nodes and its child nodes.

Example of topological sort of the local dependency graph and corresponding λ-term

We start with the local dependency graph of N at path 1:



From this point onwards we consider paths relatively to the node at path 1. So relative path 0 means absolute path 1, relative path 1 means absolute path 11 and relative path 2 means absolute path 12.

We choose the topological sort $\tau = i_0\, i_2\, m_2\, s_2\, i_1\, m_1\, s_1\, s_0\, m_0 \top$ which is compatible with the local dependency graph. We deduce the induced subsorts for each node: $\tau_0 = i_0\, s_0\, m_0 \top$, $\tau_1 = i_1\, m_1\, s_1 \top$ and $\tau_2 = i_2\, m_2\, s_2 \top$.

From τ_1 and τ_2 we deduce the types of the terms computed by the child nodes 1 and 2: $t(\tau_1) = t(\tau_2) = o \to (o \times o)$. From τ_0 we get the type of the term M we want to build. This term M takes the terms computed by child nodes 1 and 2 to compute the result at path 0. So $t(\tau_0) = o \to (o \times o)$.

Noting X_1 and X_2 the terms computed by child nodes 1 and 2 respectively, we get:

$$M = \lambda i_0.\mathsf{let}(m_2, s_2) = X_2\, i_0\, \mathsf{in}\ \mathsf{let}(m_1, s_1) = X_1\, s_2\, \mathsf{in}\, (s_1, f\, m_1\, m_2)$$

We get this by reducing the term obtained from definition 34 with the topological sort τ and with the continuation function $Cont : i \to X_i$.

For example, instead of $\mathsf{let}\,(m_2, s_2) = X_2\, i_0\, \mathsf{in}$, definition 34 would produce:

$$\mathsf{let}\, i_2 = i_0\, \mathsf{and}\, X_2' = X_2\, i_2\, \mathsf{in}\ \mathsf{let}\,(m_2, X_2'') = X_2'\, \mathsf{in}\ \mathsf{let}\, s_2 = X_2''\, \mathsf{in}$$

which can be simplified into $\mathsf{let}\,(m_2, s_2) = X_2\, i_0\, \mathsf{in}$. A similar simplification was applied with $\mathsf{let}(m_1, s_1) = X_1\, s_2\, \mathsf{in}$.

It is important to note that M depends entirely on our chosen topological sort τ of the local dependency graph.

Definition 34 For all tree constant a of arity n in Σ, for all topological sort τ over a subset of $A^{\top}_{[0,n]}$, injective substitution var which associates variables of type o with attributes, and injective substitution $Cont$ which associates variables with indices between 1 and n such that for all $i \in [1,n]$, $Cont(i)$ is of type $t(\tau|_{A^{\top}_i})$, we define the term $\mathbb{M}_a(\tau, var, Cont)$ by induction on τ as follows:

- if $\tau = (\alpha, 0)\,\top$ with $\alpha \in S$ then : $\mathbb{M}_a(\tau, var, Cont) \triangleq var(R(a)((\alpha, 0)))$
- if $\tau = (\alpha, 0)\,\tau'$ with $\alpha \in S$ and $\tau' \neq \top$ then :
$\mathbb{M}_a(\tau, var, Cont) \triangleq \mathsf{let}\ y_{(\alpha,0)} = var(R(a)((\alpha, 0)))\ \mathsf{in}$
$$(y_{(\alpha,0)}, \mathbb{M}_a(\tau', var \uplus [(\alpha, 0) \to y_{(\alpha,0)}], Cont))$$
- if $\tau = (\gamma, 0)\,\tau'$ with $\gamma \in I$ then :
$\mathbb{M}_a(\tau, var, Cont) \triangleq \lambda y_{(\gamma,0)}.\mathbb{M}_a(\tau', var \uplus [(\gamma, 0) \to y_{(\gamma,0)}], Cont)$
- if $\tau = (\alpha, i)\,\tau'$ with $\alpha \in S$, $i \neq 0$ and $\tau|_{A^{\top}_i} \neq (\alpha, i)\,\top$ then :
$\mathbb{M}_a(\tau, var, Cont) \triangleq \mathsf{let}\ (y_{(\alpha,i)}, X'_i) = Cont(i)\ \mathsf{in}$
$$\mathbb{M}_a(\tau', var \uplus [(\alpha, i) \to y_{(\alpha,i)}], Cont \circ [i \to X'_i])$$
with X'_i a fresh variable of type $t(\tau'|_{A^{\top}_i})$.
- if $\tau = (\alpha, i)\,\tau'$ with $\alpha \in S$, $i \neq 0$ and $\tau|_{A^{\top}_i} = (\alpha, i)\,\top$ then :
$\mathbb{M}_a(\tau, var, Cont) \triangleq \mathsf{let}\ y_{(\alpha,i)} = Cont(i)\ \mathsf{in}\ \mathbb{M}_a(\tau', var \uplus [(\alpha, i) \to y_{(\alpha,i)}], Cont')$
where $Cont'$ is $Cont$ from which we removed the association $[i \to Cont(i)]$.
- if $\tau = (\gamma, i)\,\tau'$ with $\gamma \in I$ and $i \neq 0$ then :
$\mathbb{M}_a(\tau, var, Cont) \triangleq \mathsf{let}\ y_{(\gamma,i)} = var(R(a)((\gamma, i)))\ \mathsf{and}\ X'_i = Cont(i)\, y_{(\gamma,i)}\ \mathsf{in}$
$$\mathbb{M}_a(\tau', var \uplus [(\gamma, i) \to y_{(\gamma,i)}], Cont \circ [i \to X'_i])$$
where X'_i is a fresh variable of type $t(\tau'|_{A^{\top}_i})$.

Then we prove that $\mathbb{M}_a$ fits the semantics we have chosen:

Lemma 33 For all constant a of arity n in Σ, and for all topological sort τ over a subset of $A^{\top}_{[0,n]}$, noting $\tau_i = \tau|_{A^{\top}_i}$ for $i \leq n$, noting $M = \mathbb{M}_a(\tau, var, Cont)$ where var is the empty substitution and for all $i \in [1,n]$, $Cont(i) = X_i$ with X_i a free variable of type $t(\tau_i)$, then M is of type $t(\tau_0)$ and, for all tree N and path $u \in V_N$ such that $lab_N(u) = a$ and τ is a topological sort of $G_{u.[0,n]}(N)$, for all terms $M_1 \in \mathcal{R}_{\tau_1}(N, u1), \ldots, M_n \in \mathcal{R}_{\tau_n}(N, un)$:

$$M[X_1/M_1, \ldots, X_n/M_n] \in \mathcal{R}_{\tau_0}(N, u)$$

Proof
We first prove a more general claim by induction on τ:

Claim 3 For all topological sort τ over a subset of $A^{\top}_{[0,n]}$, for all tree N and path $u \in V_N$ such that $lab_N(u) = a$ and τ is a topological sort of $G_{u.[0,n]}(N)$, for all injective mapping var from attributes to variables such that, for all $(\alpha, i) \in \tau$, all attribute appearing in $R(a)((\alpha, i))$ is either in τ or in the domain of var, for all function $Cont$ associating variables with indices $i \in [1, n]$ and for all substitution σ of the variables in $Cont$ such that $\forall i \in [1, n], \sigma(Cont(i)) \in \mathcal{R}_{\tau_i}(N, ui)$ with $\tau_i = \tau|_{A^{\top}_i}$:

$$\sigma \circ \nu(\mathbb{M}_a(\tau, var, Cont)) \in \mathcal{R}_{\tau_0}(N, u)$$

where ν is the variable substitution such that for all attribute (α, i) in $\mathsf{dom}(var)$:
$\nu(var((\alpha, i))) = Att(N, (\alpha, ui))$.

Proof
We fix a topological sort τ over a subset of $A^{\top}_{[0,n]}$, an input tree N, a path $u \in V_N$ such that $lab_N(u) = a$ and τ is a topological sort of $G_{u.[0,n]}(N)$, an injective mapping var from attributes to variables such that, noting $\mathsf{dom}(var)$ its domain, for all $(\alpha, i) \in \tau$, all attribute appearing in $R(a)((\alpha, i))$ is either in τ or in $\mathsf{dom}(var)$. We note ν the variable substitution such that for all attribute $(\alpha, i) \in \mathsf{dom}(var)$, $\nu(var((\alpha, i))) = Att(N, (\alpha, ui))$ (exists because var is injective), we also fix a function $Cont$ associating variables with indices in $[1, n]$, and a substitution σ of the free variables in $Cont$ such that $\forall i \in [1, n], \sigma(Cont(i)) \in \mathcal{R}_{\tau_i}(N, ui)$ where $\tau_i = \tau|_{A^{\top}_i}$.

We assume the induction hypothesis for all topological sort τ' shorter (with a smaller number of elements) than τ.

As in the definition of $\mathbb{M}_a$ we have 6 cases:

- if $\tau = (\alpha, 0)\top$ with $\alpha \in S$ then $\mathbb{M}_a(\tau, var, Cont) \triangleq var(R(a)((\alpha, 0)))$. In this case $\sigma \circ \nu(\mathbb{M}_a(\tau, var, Cont)) = \nu \circ var(R(a)((\alpha, 0)))$. Since all attributes appearing in $R(a)((\alpha, 0))$ are in $\mathsf{dom}(var)$, and $\forall(\alpha, i) \in \mathsf{dom}(var)$, $\nu(var((\alpha, i))) = Att(N, (\alpha, ui))$. Then by definition of $Att(N, (\alpha, u))$ with $lab_N(u) = a$:
$\sigma \circ \nu(\mathbb{M}_a(\tau, var, Cont)) = Att(N, (\alpha, u)) \in \mathcal{R}_{(\alpha,0)\top}(N, u)$.

- if $\tau = (\alpha, 0)\tau'$ with $\alpha \in S$ and $\tau' \neq \top$ then $\mathbb{M}_a(\tau, var, Cont) \triangleq$
$\mathsf{let}\ y_{(\alpha,0)} = var(R(a)((\alpha, 0)))\ \mathsf{in}\ (y_{(\alpha,0)}, \mathbb{M}_a(\tau', var \uplus [(\alpha, 0) \to y_{(\alpha,0)}], Cont))$. The induction hypothesis implies that $\sigma \circ \nu'(\mathbb{M}_a(\tau', var, Cont)) \in \mathcal{R}_{\tau'_0}(N, u)$ where
$\nu' = \nu \uplus [y_{(\alpha,0)} \to Att(N, (\alpha, u))]$. Similarly to the case
$\tau = (\alpha, 0)\top$: $\nu(var(R(a)((\alpha, 0)))) = Att(N, (\alpha, u))$. Therefore
$\sigma \circ \nu(\mathbb{M}_a((\alpha, 0)\tau', var, Cont)) \in \mathcal{R}_{(\alpha,0)\tau'_0}(N, u)$.

- if $\tau = (\gamma, 0)\tau'$ with $\gamma \in I$ then $\mathbb{M}_a(\tau, var, Cont) \triangleq$
$\lambda y_{(\gamma,0)}.\mathbb{M}_a(\tau', var \uplus [(\gamma, 0) \to y_{(\gamma,0)}], Cont)$. The induction hypothesis entails that $\sigma \circ \nu'(\mathbb{M}_a(\tau', var \uplus [(\gamma, 0) \to y_{(\gamma,0)}], Cont)) \in \mathcal{R}_{\tau'_0}(N, u)$ where
$\nu' = \nu \uplus [y_{(\gamma,0)} \to Att(N, (\gamma, u))]$. Then, by definition of $\mathcal{R}_{(\gamma,0)\tau'_0}(N, u)$ for $\gamma \in I$,
$\sigma \circ \nu(\mathbb{M}_a(\tau, var, Cont)) \in \mathcal{R}_{(\gamma,0)\tau'_0}(N, u)$.

- if $\tau = (\alpha, i)\tau'$ with $\alpha \in S$, $i \neq 0$ and $\tau_i \neq (\alpha, i)\top$ then $\mathbb{M}_a(\tau, var, Cont) \triangleq$
$\mathsf{let}\ (y_{(\alpha,i)}, X'_i) = Cont(i)\ \mathsf{in}\ \mathbb{M}_a(\tau', var \uplus [(\alpha, i) \to y_{(\alpha,i)}], Cont \circ [i \to X'_i])$ where X'_i

is a fresh variable of type $t(\tau'_i)$. Noting $(M_1, M_2) = \sigma(\mathit{Cont}(i)) \in \mathcal{R}_{(\alpha,i)\tau'_i}(N, ui)$, we have $M_1 \rightarrow^*_{\beta\eta} \mathit{Att}(N, (\alpha, ui))$ and $M_2 \in \mathcal{R}_{\tau'_i}(N, ui)$. So we apply the induction hypothesis on $\sigma' \circ \nu'(\mathbb{M}_a(\tau', \mathit{var} \uplus [(\alpha, i) \rightarrow y_{(\alpha,i)}], \mathit{Cont} \circ [i \rightarrow X'_i]))$ where $\nu' = \nu \uplus [y_{(\alpha,i)} \rightarrow \mathit{Att}(N, (\alpha, ui))]$ and σ' is obtained from σ by removing the association $[\mathit{Cont}(i) \rightarrow \sigma(\mathit{Cont}(i))]$ and adding $[X'_i \rightarrow M_2]$. So $\sigma \circ \nu(\mathbb{M}_a(\tau, \mathit{var}, \mathit{Cont})) =_{\beta\eta} \sigma' \circ \nu'(\mathbb{M}_a(\tau', \mathit{var} \uplus [(\alpha, i) \rightarrow y_{(\alpha,i)}], \mathit{Cont} \circ [i \rightarrow X'_i]))$ and therefore $\sigma \circ \nu(\mathbb{M}_a(\tau, \mathit{var}, \mathit{Cont})) \in \mathcal{R}_{\tau_0}(N, u)$.

- if $\tau = (\alpha, i)\,\tau'$ with $\alpha \in S$, $i \neq 0$ and $\tau_i = (\alpha, i)\,\top$ then $\mathbb{M}_a(\tau, \mathit{var}, \mathit{Cont}) \triangleq$ $\mathsf{let}\ y_{(\alpha,i)} = \mathit{Cont}(i)\ \mathsf{in}\ \mathbb{M}_a(\tau', \mathit{var} \uplus [(\alpha, i) \rightarrow y_{(\alpha,i)}], \mathit{Cont}')$ where Cont' is Cont from which we removed the association $i \rightarrow \mathit{Cont}(i)$. This case is analogous to the previous one, and with the same arguments we reach the conclusion that $\sigma \circ \nu(\mathbb{M}_a(\tau, \mathit{var}, \mathit{Cont})) \in \mathcal{R}_{\tau_0}(N, u)$.

- if $\tau = (\gamma, i)\,\tau'$ with $\gamma \in I$ and $i \neq 0$ then $\mathbb{M}_a(\tau, \mathit{var}, \mathit{Cont}) \triangleq\ \mathsf{let}\ y_{(\gamma,i)} = \mathit{var}(R(a)((\gamma, i)))\ \mathsf{and}\ X'_i = \mathit{Cont}(i)\, y_{(\gamma,i)}\ \mathsf{in}$
$$\mathbb{M}_a(\tau', \mathit{var} \uplus [(\gamma, i) \rightarrow y_{(\gamma,i)}], \mathit{Cont} \circ [i \rightarrow X'_i])$$
where X'_i is a fresh variable of type $t(\tau'_i)$. We have $\sigma(\mathit{Cont}(i)) \in \mathcal{R}_{(\gamma,i)\tau'_i}(N, ui)$ and $\nu(\mathit{var}(R(a)((\gamma, i)))) =_{\beta\eta} \mathit{Att}(N, (\gamma, ui))$, then $\sigma(\mathit{Cont}(i))\, \nu(\mathit{var}(R(a)((\gamma, i)))) \in \mathcal{R}_{\tau'_i}(N, ui)$. We apply the induction hypothesis on $\sigma' \circ \nu'(\mathbb{M}_a(\tau', \mathit{var} \uplus [(\gamma, i) \rightarrow y_{(\gamma,i)}], \mathit{Cont} \circ [i \rightarrow X'_i]))$ where $\nu' = \nu \uplus [y_{(\gamma,i)} \rightarrow \mathit{Att}(N, (\gamma, ui))]$ and σ' is obtained from σ by removing the association $[\mathit{Cont}(i) \rightarrow \sigma(\mathit{Cont}(i))]$ and adding $[X'_i \rightarrow \sigma(\mathit{Cont}(i))\ \nu(\mathit{var}(R(a)((\gamma, i))))]$. Therefore $\sigma \circ \nu(\mathbb{M}_a(\tau, \mathit{var}, \mathit{Cont})) = \sigma' \circ \nu'(\mathbb{M}_a(\tau', \mathit{var} \uplus [(\gamma, i) \rightarrow y_{(\gamma,i)}], \mathit{Cont} \circ [i \rightarrow X'_i])) \in \mathcal{R}_{\tau_0}(N, u)$.

This ends the inductive proof of the claim. □

Since τ is a topological sort of the graph $G_{u.[0,n]}(N)$, for all $(\alpha, i) \in \tau$, all attribute appearing in $R(a)((\alpha, i))$ is in τ. Therefore we can apply the claim on τ with $\mathit{Cont}(i)$ the substitution such that $\mathit{Cont}(i) = X_i$ for $i \in [1, n]$, σ the substitution such that $\sigma(X_i) = M_i$ for $i \in [1, n]$ and var and ν empty substitutions. So :

$$\mathbb{M}_a(\tau, \mathit{var}, \mathit{Cont})[X_1/M_1, \ldots, X_n/M_n] = \sigma(\mathbb{M}_a(\tau, \mathit{var}, \mathit{Cont})) \in \mathcal{R}_{\tau_0}(N, u)$$

□

Now that we have shown that $\mathbb{M}$ computes terms correctly, we need to prove that it is *almost linear* in general, and *linear* if our ATT is *single use restricted.*

Lemma 34 For all tree constant a of arity n in Σ, for all topological sort τ over a subset of $A^{\top}_{[0,n]}$, injective substitution var which associates variables of type o with attributes and injective substitution Cont which associates variables with indices between 1 and n such that, for all $i \in [1, n]$, $\mathit{Cont}(i)$ is of type $t(\tau|_{A^{\top}_i})$, the term $\mathbb{M}_a(\tau, \mathit{var}, \mathit{Cont})$ is *almost linear.*

Proof
In the inductive definition of $\mathbb{M}_a(\tau, var, Cont)$, the variables we use are either in *var* or in *Cont*. Variables in *var* are of atomic type so copying them does not prevent almost linearity. Each time a variable of *Cont* is used, it occurs once and is removed from *Cont* in the inductive call to $\mathbb{M}_a(\tau', var', Cont')$. So $\mathbb{M}_a(\tau, var, Cont)$ is *almost linear.* □

Lemma 35 Assumming the ATT is *single use restricted*, for all tree constant a of arity n in Σ, for all topological sort τ over a subset of $A^{\top}_{[0,n]}$, injective substitution *var* which associates variables of type o with attributes and injective substitution *Cont* which associates variables with indices between 1 and n such that, for all $i \in [1, n]$, $Cont(i)$ is of type $t(\tau|_{A^{\top}_i})$, the term $\mathbb{M}_a(\tau, var, Cont)$ is *linear.*

Proof
As we saw in the previous lemma, variables in *Cont* are never copied, so we only need to prove that variables in *var* are not copied.

According to corollary 21, since the ATT is *single use restricted*, the graph $G_{u.[0,n]}(N)$ is a tree. For all attribute (α, i) in $G_{u.[0,n]}(N)$ there exists a unique attribute x in $G_{u.[0,n]}(N)$ such that there is an edge $((\alpha, i), x)$ in $G_{u.[0,n]}(N)$. So x is the only attribute in $G_{u.[0,n]}(N)$ such that (α, i) occurs in $R(a)(x)$.

A straightforward induction on τ proves that for all τ, var and *Cont* such that $var((\alpha, i)) = y_{(\alpha,i)}$, the number of occurrences of $y_{(\alpha,i)}$ in $\mathbb{M}_a(\tau, var, Cont)$ is 1 if x is in τ and 0 otherwise.

Therefore the term $\mathbb{M}_a(\tau, var, Cont)$ is *linear.* □

Then we define the term that will compute the inherited attributes of the root node by applying the *root* equations:

Definition 35 With $G(root)$ the graph whose set of vertices is $A^{\top}_{\epsilon}$ an edges represent dependencies in the *root* equations; for all subsort τ of a topological sort of $G(root)$, injective substitution *var* which associates variables of type o with attributes, and variable X_0 of type $t(\tau)$, we define the term $\mathbb{M}_{root}(\tau, var, X_0)$ of type o by induction on τ as follows:

- if $\tau = (\alpha, 0)\, \top$ with $\alpha \in S$ then : $\mathbb{M}_{root}(\tau, var, X_0) \triangleq X_0$
- if $\tau = (\alpha, 0)\, \tau'$ with $\alpha \in S$ and $\tau' \neq \top$ then :
$\mathbb{M}_{root}(\tau, var, X_0) \triangleq \mathsf{let}\ (y_{(\alpha,0)}, X'_0) = X_0\ \mathsf{in}\ \mathbb{M}_{root}(\tau', var \uplus [(\alpha, 0) \to y_{(\alpha,0)}], X'_0)$
- if $\tau = (\gamma, 0)\, \tau'$ with $\gamma \in I$ then : $\mathbb{M}_{root}(\tau, var, X_0) \triangleq$
$\mathsf{let}\ y_{(\gamma,0)} = var(root((\gamma, 0)))\ \mathsf{and}\ X'_0 = X_0\ y_{(\gamma,0)}\ \mathsf{in}\ \mathbb{M}_{root}(\tau', var \uplus [(\gamma, 0) \to y_{(\gamma,0)}], X'_0)$
where X'_0 is a fresh variable of type $t(\tau')$.

For all subsort τ of a topological sort of $G(root)$ we define the term $\mathbb{M}_{root}(\tau)$ as the term $\lambda X_0.\mathbb{M}_{root}(\tau, var, X_0)$ where *var* is the empty substitution and X_0 is a free variable of type $t(\tau)$.

Then we prove that $\mathbb{M}_{root}$ computes the right output:

Lemma 36 For all subsort τ of a topological sort of $G(root)$, for all tree N such that τ is a topological sort of $G_\epsilon(N)$ and for all term $M_0 \in \mathcal{R}_\tau(N, \epsilon)$, the term $\mathbb{M}_{root}(\tau)\, M_0$ β-reduces to the output of the ATT on input N.

Proof
Similar to lemma 33. □

Lemma 37 For all subsort τ of a topological sort of $G(root)$, injective substitution var which associates variables of type o with attributes and variable X_0 of type $t(\tau)$, the term $\mathbb{M}_{root}(\tau, var, X_0)$ is *almost linear* in general and *linear* if the ATT is *single use restricted.*

Proof
Similar to lemmas 34 and 35. □

Finally we can define a HODTR$_{\text{al}}$ which is equivalent to an ATT:

Definition 36 Let $T = (\Sigma_1, \Sigma_2, S, I, out, R, root)$ be an ATT.
We define the HODTR$_{\text{al}}$ $\mathbb{HO}(T)$ as $(\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R', \mathtt{A})$ with:

- $\mathtt{A}$, the look-ahead automaton, is the bottom-up tree automaton given by lemma 19,
- Σ_Q, the signature of the set of states, is defined as:

  $$Q \triangleq \{q_0\} \cup \{q_{\tau(\alpha,\epsilon)\top} \mid \tau\,(\alpha, \epsilon)\top \text{ is a topological sort on a subset of } A_\epsilon^\top \text{ and } \alpha \in S\}$$

  The number of states is $|Q| = |S \cup I|!$. The type of a state q_τ is $o_1 \to t(\tau)$, where $t(\tau)$ is defined in definition 32,
- Σ_1 and Σ_2 are respectively the input and output tree signatures from the ATT T,
- R' is the set of rules, it includes the rules the form:

  $$q_{\tau_0}(a\,\overrightarrow{x})\langle\,\overrightarrow{\ell}\,\rangle \to M(q_{\tau_1}x_1)\ldots(q_{\tau_n}x_n)$$

  where $\overrightarrow{\ell} = \ell_1, \ldots, \ell_n$ are the states of look-ahead associated with the subtrees $\overrightarrow{x} = x_1, \ldots, x_n$ respectively and, noting $\tau = f(a, \tau_0, (\ell_1, \ldots, \ell_n))$ the topological sort computed in lemma 30, for all $1 \leq j \leq n$: τ_j is the topological sort $\tau_j = j^{-1}.(\tau|_{A_j^\top})$. And with $M = \mathsf{let}\ X_1 = q_{\tau_1}(x_1)\ \mathsf{and}\ \ldots X_n = q_{\tau_n}(x_n)\ \mathsf{in}\ \mathbb{M}_a(\tau, var, Cont)$ where var is the empty substitution, $Cont = [i \to X_i]_{i\in[1,n]}$ and $\mathbb{M}_a$ is defined in definition 34.

  To that first set of rules we add special rules for the initial state q_0 : for all rule already in R' of the form $q_{\tau_0}(a\,\overrightarrow{x})\langle\,\overrightarrow{\ell}\,\rangle \to M$ where τ_0 is a subsort of a topological sort of $G(root)$, we add to R' the rule:

  $$q_0(a\,\overrightarrow{x})\langle\,\overrightarrow{\ell}\,\rangle \to \mathbb{M}_{root}(\tau_0)\, M$$

A complexity analysis on the size of $\mathbb{HO}(T)$ reveals that, noting $m = |S| + |I|$ the number of attributes, n the maximum arity of a symbol in Σ_1 and p the number of symbols in Σ_1, the number of states in the look-ahead automaton of $\mathbb{HO}(T)$ grows in e^{m^2} (graphs with attributes as vertices), the number of states of $\mathbb{HO}(T)$ grows with $m!$ (orderings on the set of attributes). Then the number of rules of $\mathbb{HO}(T)$ grows in $m! * p * e^{m^2 * n}$ and the size of these rules grows linearly with the size of the rules of T and the number m of attributes. Note that the only non-linear factor is $m! * e^{m^2 * n}$ and comes from the potentially big numbers of accessible synthesis graphs and topological sorts of synthesis graphs, which could be smaller in practical cases.

Theorem 22 *For all ATT T, the HODTR$_{al}$ $T' = \mathbb{HO}(T)$ is equivalent to T, and T' is linear if T is* single use restricted*.*

Proof
Let N be an input tree of T'.

For all path $u \in V_N$, according to lemma 19, the look-ahead state associated with the node at path u in N is the synthesis graph $GS_u(N)$ of N at path u.

Then a straighforward downward induction using lemma 30 shows that for all non-ϵ path $u \in V_N$ the node at path u in N is processed by a state of the form q_τ where τ is a topological sort of $G_u(N)$.

A straighforward upward induction using lemma 33 proves that for all non-ϵ path $u \in V_N$ the result of the computation of $q_\tau(N\downharpoonleft_u)$ is a term in $\mathcal{R}_\tau(N, u)$.

Finally, using lemma 36, we conclude that $q_0(N)$ computes exactly the output of the ATT T on the input tree N. Thus we have shown that T' computes the same transduction as T.

Furthermore, lemmas 34, 35 and 37 imply that T' is almost linear in general and linear if T is *single use restricted*. □

Theorem 23 *For all ATT T and relabeling attribute grammar P there exists a HODTR$_{al}$ T' equivalent to $P \circ T$ and if T is* single use restricted *then T' is linear.*

Proof
The relabeling P can be modeled by a simple HODTR$_{\text{lin}}$. Then we can compose it with $\mathbb{HO}(T)$ in order to obtain a HODTR$_{\text{al}}$ T' equivalent to $P \circ T$ such that if T is *single use restricted* then $\mathbb{HO}(T)$ is linear and therefore T' is also linear. □

Corollary 24 The class MSOT is included in the class HODTR$_{\text{lin}}$ and the class MSOTS is included in the class HODTR$_{\text{al}}$.

7.3.3 HODTR$_{\text{al}}$ $\subseteq$ REL $\circ$ ATT and HODTR$_{\text{lin}}$ $\subseteq$ REL $\circ$ ATT$_{sur}$

Theorem 25 *For all HODTR$_{al}$ $T = (\Sigma_Q, \Sigma_1, \Sigma_2, q_0, R, \mathtt{A})$ there exists a relabeling attribute grammar P and an ATT T' such that T is equivalent to $P \circ T'$ and, if T is linear, then T' is* single use restricted*.*

Proof
First we assume that T is the result of the order reduction procedure described in the proof of theorem 17, so the result of applying a state $q \in Q$ to an input tree N is a tuple of tree contexts: $q(N) \rightarrow_T (C_1, \ldots, C_n)$.

The relabeling attribute grammar framework is powerful enough to simulate the bottom-up look-ahead automaton and the top-down finite state structure of T. Therefore we can build a relabeling attribute grammar P that computes, for each node of an input tree N, which rule of T would be applied to it. Then T' will compute the actual results of applying these rules.

Since each state q of T computes a tuple of contexts, we need attributes to simulate tree contexts. We can do this by mapping the free variables of a tree context to inherited attributes, and mapping the tree context to a synthesized attribute. For example a tree context $C_1 = f\, y_1\, y_2$, where f is a tree constant of arity 2 and y_1 and y_2 are free variables, will be represented by one synthesized attribute α_1 linked to two inherited attributes β_1 and β_2 by the equation: $(\alpha_1, \epsilon) = f\,(\beta_1, \epsilon)\,(\beta_2, \epsilon)$. This way we can build an ATT T' such that $P \circ T'$ is equivalent to T.

Furthermore, if T is linear, then each tree context is used exactly once, so attributes are never used twice and T' is *single use restricted.* □

Corollary 26 $\mathrm{HODTR}_{\mathrm{al}} \subseteq \mathrm{REL} \circ \mathrm{ATT}$ and $\mathrm{HODTR}_{\mathrm{lin}} \subseteq \mathrm{REL} \circ \mathrm{ATT}_{sur}$.

Finally we can conclude, using to theorem 19:

Theorem 18 $\mathrm{HODTR}_{\mathrm{lin}}$ are equivalent to MSOT and $\mathrm{HODTR}_{\mathrm{al}}$ are equivalent to MSOTS.

Part IV

Part 4 : Expressivity of MSO transductions, case of the MIX languages

Chapter 8

The MIX languages and MSO transductions

In this chapter we present a contribution in the research field of languages of words. Specifically we show that the language $MIX_2 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$, also known as the two-sided Dyck language D_2^*, is not a EDT0L language of finite index. We then deduce that this language is not an EDT0L language using a theorem from Latteux [23, 24]. This result, which solves an open problem, has notably been asked because of its implications in computational group theory, those implications are not discussed here.

8.1 Preliminaries

In this section we present the MIX languages, and introduce some classes of word languages, most importantly EDT0L languages. We also give some of its properties which are known from the literature and therefore are not part of our contribution.

The MIX languages

We are especially interested in the MIX languages, those are defined by, for all integer $k \leq 2$: $MIX_k = \{w \in \{a_1, \ldots, a_k\}^* \mid \forall i,\, j \in [1,k],\, |w|_{a_i} = |w|_{a_j}\}$. These languages have attracted the attention of at least two communities: mathematical linguistics and computational group theory. The first one is interested in these languages as being completely unstructured and should not be in the class of languages that are used to define natural languages. The second community is interested in classifying groups in terms of the formal properties of their word languages. The languages MIX_k are the word languages of particular presentations of the groups $\mathbb{Z}^{k-1}$. A long standing open problem in computational linguistics was whether MIX_3 is a Tree Adjoining Language. This question was solved in the negative proving that MIX_3 is not a Tree Adjoining Language. Another long standing problem in computational group theory is whether MIX_3 is an indexed language. This problem is still open. On the other hand it has been showed that for every k, MIX_k is a Multiple Context Free Language (MCFL). And even more precisely that MIX_k is a k-Multiple Context-Free Language (k-MCFL), i.e. an MCFL that is only allowed to handle tuples of at most k strings.

EDT0L systems

The EDT0L acronym stands for Extended Deterministic Table 0 Lindenmayer systems. They are defined as follows.

Definition 37 An *EDT0L system* is a 4-tuple $G = (V, \Sigma, w_0, H)$ where V (called the working alphabet of G) and $\Sigma \subseteq V$ (called the terminal alphabet) are finite alphabets, $w_0 \in V$ is the initial word of G. H is a finite set of finite substitutions from V to $(V \cup \Sigma)^*$.

A finite sequence $\tau_1, \ldots, \tau_n$ of substitutions in H generates the word $u = \tau_1 \circ \tau_2 \circ \cdots \circ \tau_n(w_0)$.

The language $L(G)$ generated by G is the set of words in Σ^* that are generated by sequences of substitutions in H.

An *EDT0L system* G is of *finite index* if there is an integer n such that all word generated by sequences of substitutions in H contains at most n occurrences of letters from the working alphabet V.

The set of languages generated by EDT0L systems is noted EDT0L.

EDT0L are closed under sequential transductions (i.e. deterministic rational transductions).

Theorem 27 ([6, 15]) *The class of EDT0L languages is closed under sequential transductions.*

Non-branching Multiple Context-Free Grammars

Definition 38 A non-branching Multiple Context-Free Grammar (MCFG(1)) is a 4-tuple $G = (V, \Sigma, S, R)$ where V is a finite ranked alphabet (called the working alphabet), Σ is a finite alphabet (called the terminal alphabet) and $S \in V$ is a symbol of arity 1 (called the initial symbol). R is a finite set of rules, a rule in R is either a trivial rule of the form:

$$J(u_1, \ldots, u_n) \leftarrow$$

where $J \in V$ is of arity n and $u_1, \ldots, u_n$ are words over the alphabet Σ, or it an internal rule of the form:

$$J(u_1, \ldots, u_n) \leftarrow K(x_1, \ldots, x_m)$$

where $J, K \in V$ are symbols of arity n and m respectively, $x_1, \ldots, x_m$ are variables and $u_1, \ldots, u_n$ are words over the alphabet $\Sigma \cup \{x_1, \ldots, x_m\}$ such that each variable x_i appears at most once in the word $u_1 u_2 \ldots u_n$.

The grammar G is non-permuting if, in each internal rule, variables $x_1, \ldots, x_m$ appear in the word $u_1 u_2 \ldots u_n$ in order.

The grammar G is non-erasing if, in each internal rule, each variable x_i appears exactly once in the word $u_1 u_2 \ldots u_n$.

Rules of G recursively define a set of derivable tuples of words on each symbol $J \in V$. A symbol $J \in V$ of arity n derives a n-tuple $(u_1, \ldots, u_n)$ of words over Σ either if

- there is a trivial rule $J(u_1, \dots, u_n) \leftarrow$, or
- there is an internal rule $J(w_1, \dots, w_n) \leftarrow K(x_1, \dots, x_m)$ and a substitution σ of the variables $x_1, \dots, x_m$ such that $u_i = \sigma(w_i)$ for all $i \leq n$ and the symbol K derives the m-tuple $(\sigma(x_1, \dots, \sigma(x_m))$.

The language generated by grammar G, noted $L(G)$, is the set of words derivable on the initial symbol S of arity 1.

The set of non-branching Multiple Context-Free Languages (noted MCFL(1)) is the set of languages generated by non-branching Multiple Context-Free Grammar.

Theorem 28 *Non-permuting non-erasing MCFG(1) generate the same languages as general MCFG(1).*

We now state an equivalence between classes of languages of the literature. The main result of chapter 8 shows that MIX_2 is not a language of this class.

Theorem 29 *The following classes of languages are equivalent:*

- *non-branching Multiple Context-Free Languages (MCFL(1)),*
- *output languages of Streaming String Transducers (out(SST)) [3],*
- *output languages of word-to-word Monadic Second Order Transductions (out(MSOT)),*
- *output languages of deterministic two-way transducers (out(2WST)),*
- *EDT0L of finite index.*

Proof
The equivalence of MCFL(1) and out(SST) is clear from the definitions of the two formalisms. The equivalence of out(SST), out(MSOT) and out(2WST) is stated in [4]. Finally the equivalence of out(2WST) and EDT0L of finite index is proved in [23]. □

The following theorem is due to Latteux [23, 24], its statement requires that we define the shuffle operation between two languages L_1 and L_2. We write $L_1 \uparrow L_2$, *the shuffle of L_1 and L_2*, for the language $\{u_1v_1 \dots v_nu_n \mid u_1 \dots u_n \in L_1 \wedge v_1 \dots v_n \in L_2\}$ (note that the u_i and the v_i can be the empty string in the definition of $L_1 \uparrow L_2$).

Theorem 30 ([23, 24]) *Given $L \subseteq \Sigma^*$ and c so that $c \notin \Sigma$, if $L \uparrow c^*$ is an EDT0L, then L is an EDT0L of finite index.*

This Theorem allows us to prove that MIX_2 is not an EDT0L language as a corollary of the fact that MIX_2 is not generated by an EDT0L of finite index.

8.2 MIX_2 is not an EDT0L language

In this section we present our contribution for this chapter: the proof that MIX_2 is not an EDT0L (Theorem 31). Recall that MIX_2 is the language $\{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$.

Theorem 31 *MIX_2 is not an EDT0L of finite index.*

Corollary 32 MIX_2 is not an EDT0L.

Proof
Theorem 31 and Theorem 29 prove that MIX_2 is not an EDT0L of finite index. Now the sequential transduction of Figure 8.1 that maps MIX_2 to $MIX_2 \uparrow c^*$. This transducer just maps every factor ab in a word to c. Theorem 27 implies then that if MIX_2 is an EDT0L then so is $MIX_2 \uparrow c^*$. Then Theorem 30 would imply that MIX_2 is an EDT0L of finite index, a contradiction.

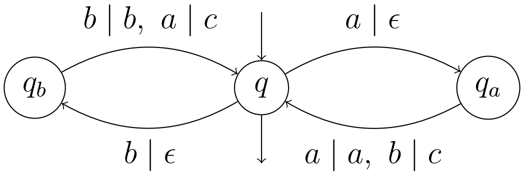


Figure 8.1 – Transducer that maps MIX_2 to $MIX_2 \uparrow c^*$

□

Definition 39 Given a word w in $\{a,b\}^*$ we write $\Delta(w)$ for $|w|_a - |w|_b$. Given a tuple $\mathbf{w} = (w_1, \dots, w_n)$ of words in $\{a,b\}^*$, we let $\Delta(\mathbf{w})$ be $\Delta(w_1 \dots w_n)$. We note $|\Delta(\mathbf{w})|$ the absolute value of $\Delta(\mathbf{w})$.

To prove that MIX_2 is not an EDT0L of finite index, we prove that it is not a non-branching Multiple Context-Free Language (MCFL(1)) and conclude using theorem 29 stating that those classes of languages are the same. Instead of directly working on the derivations of non-branching Multiple Context-Free Grammars (MCFG(1)) we introduce a notion of *k-derivation* which captures the main mechanisms of the MCFG(1) we consider when they deal with MIX_2.

Definition 40 A *k-derivation* is a sequence of k-tuples $\mathbf{u} = (u_1, \dots, u_k)$ so that:

- for every i in $[1,k]$, u_i is in $\{a,b\}^*$, and
- $|\Delta(\mathbf{u})| \leq k$.

In a k-derivation $\mathbf{u}_1 \dots \mathbf{u}_n$, for every p in $[1, n-1]$, we have that $\mathbf{u}_{p+1} = (u'_1, \dots, u'_k)$ is the result of applying one of the following atomic operations to $\mathbf{u}_p = (u_1, \dots, u_k)$:

- for an index i in $[1,k]$:

letter erasure $u_i = xu'_i$ or $u_i = u'_i x$ for $x \in \{a, b\}$, and for every j in $[1, k] - \{i\}$, $u'_j = u_j$

left word splitting $u_{i-1} = \epsilon$ and $u_i = u'_{i-1}u'_i$, and for every j in $[1, k] - \{i - 1, i\}$, $u'_j = u_j$

right word splitting $u_{i+1} = \epsilon$ and $u_i = u'_i u'_{i+1}$ and for every j in $[1, k] - \{i, i + 1\}$, $u'_j = u_j$.

When $\mathbf{u_1} \dots \mathbf{u_n}$ is a k-derivation, we say that *there is a k-derivation from (or starting at) $\mathbf{u_1}$ to (or ending at) $\mathbf{u_n}$*. A tuple $\mathbf{u}$ *admits a k-derivation* if there is a k-derivation from $\mathbf{u}$ to $(\epsilon, \dots, \epsilon)$. By extension a word w *admits a k-derivation* when $(w, \epsilon, \dots, \epsilon)$ admits a k-derivation.

We are now going to relate k-derivations to MCFG(1).

Proposition 5 *If there is a MCFG(1) G so that $\mathcal{L}(G) = MIX_2$, then there is k so that every word w in MIX_2 admits a k-derivation.*

Proof
Let $G = (V, \Sigma, S, R)$ be a MCFG(1) such that $\mathcal{L}(G) = MIX_2$. According to theorem 29 we can assume that G is non-erasing and non-permuting. We also assume that each non-terminal $J \in V$ is used in at least one derivation of a word $w \in MIX_2$ (otherwise it could be removed from G without changing $\mathcal{L}(G)$).

First we prove the following claim for each non-terminal $J \in V$:

Claim 4 There is an integer Δ_J such that, for all tuple $(u_1, \dots, u_n)$ derivable on J:

$$\Delta_J = \Delta(u_1, \dots, u_n)$$

Proof
Assume that two tuples $(u_1, \dots, u_n)$ and $(u'_1, \dots, u'_n)$ are derivable on J (if not then the claim is trivial). Since J is used in at least one derivation of a word $w \in MIX_2$, there must be a sequence of rules such that $S(w) \leftarrow^* J(u_1, \dots, u_n)$. In such a case there exists some words $w_1, \dots, w_{n+1}$ so that $w = w_1u_1w_2u_2 \dots u_nw_{n+1}$. By applying the same sequence of rules to $J(u'_1, \dots, u'_n)$ we get that the word $w' = w_1u'_1w_2u'_2 \dots u'_nw_{n+1}$ is derivable on S. So $w' \in MIX_2$.

Therefore we have $\Delta(w) = 0 = \Delta(w')$. By subtracting $\sum_{1 \le i \le n+1} \Delta(w_i)$ we get:

$$\sum_{1 \le i \le n} \Delta(u_i) = \sum_{1 \le i \le n} \Delta(u'_i) \ .$$

□

We note r the maximum arity of a non-terminal in V.

Each rule of G can be expressed as a sequence of atomic operations as defined for r-derivations: letter erasure, left word splitting and right word splitting (for non-terminals of arity $i < r$ we add empty words at the end, so $u_{i+1} = \epsilon, \dots, u_r = \epsilon$). We note d the maximum number of atomic operations into which rules of G can be expressed, and Δ_V the maximum of Δ_J for $J \in V$

Then we define the integer $k = \max(r, \Delta_V + d)$. Now we prove that any word $u \in MIX_2$ admits a k-derivation.

By definition of $\mathcal{L}(G)$ and because $\mathcal{L}(G) = MIX_2$, there exists a derivation of u starting with $S(u)$. Then there is sequence of derivable tuples of the form:

$$S(u) \leftarrow J_1(u_{1,1}, \ldots, u_{1,r_1}) \quad \ldots \quad J_n(u_{n,1}, \ldots, u_{n,r_n}) \leftarrow$$

where $J_1, \ldots, J_n$ are non-terminals of respective arities $r_1, \ldots, r_n$. By definition of Δ_V we have that, for each $i \leq n$:

$$|\Delta(u_{i,1}, \ldots, u_{i,r_i})| \leq \Delta_V$$

Finally, for each $i \leq n-1$, we can find a k-derivation from $(u_{i,1}, \ldots, u_{i,r_i})$ to $(u_{i+1,1}, \ldots, u_{i+1,r_{i+1}})$. Since the rules of G can be expressed as sequences of less than d atomic operations and because an atomic operation removes at most one letter, any tuple $\mathbf{u}$ in the k-derivation respects the condition:

$$|\Delta(\mathbf{u})| \leq \Delta_V + d \leq k\,.$$

The same reasoning applies to the rules $S(u) \leftarrow J_1(u_{1,1}, \ldots, u_{1,r_1})$ and $J_n(u_{n,1}, \ldots, u_{n,r_n}) \leftarrow$. Then we get a k-derivation of $u \in MIX_2$. □

Now proving Theorem 31 boils down to show that for every k there is some word s_k of MIX_2 which does not admit any k-derivation.

8.2.1 The counter example word s_k

In this section, we fix $k > 0$ and we then let $m = 16k + 4$. We now define the following words:

Definition 41 For every n we inductively define the words v_n and u_n as follows:

- $v_n = a^{m^n}$, i.e. $v_{n+1} = v_n^m$ and $v_0 = a$,
- $u_0 = b^2$ and $u_{n+1} = v_{n+1} u_n^{2m} v_{n+1}$.

We then let:

$$s_k = v_{2k} u_{2k} v_{2k}\;.$$

Our main goal is to show that s_k does not admit any k-derivation. We start by giving some qualitative properties about the words u_n.

Lemma 38 For every n, $\Delta(u_n) = -2m^n$.

Proof
We start by proving this for $n = 0$, we have that $u_0 = b^2$ and thus $|u_0|_a - |u_0|_b = -2m^0$ which is as expected.

Now suppose that $n = p + 1$, by induction we have that $|u_p|_a - |u_p|_b = -2m^p$. Now $u_n = v_n u_p^{2m} v_n$ so $|u_n|_a - |u_n|_b = -2m^p \times 2m + 2m^n = -4m^{p+1} + 2m^n = -4m^n + 2m^n = -2m^n$ since $n = p + 1$. □

Lemma 39 For every n, $|u_n|_b = 2(2m)^n$.

Proof
Induction on n. When $n = 0$ we have that $|u_0|_b = 2$. Now $|u_{n+1}|_b = 2m|u_n|_b = 2m \times 2(2m)^n = 2(2m)^{n+1}$. □

Lemma 40 For every n, $|u_n|_a = 2(2m)^n - 2m^n$ and $\Delta(s_k) = 0$.

Proof
Consequence of the previous lemmas. □

Definition 42 For each word $u \in \{a, b\}^*$, we let left(u) and right(u) be the longest prefix and longest suffix of u in a^*. The function strip(u) returns the string u where its longest prefix and suffix made with a's have been removed.

Lemma 41 For every n, the longest factor of u_n in a^* is left(u_n) and has length $\frac{m^{n+1}-m}{m-1}$.

Proof
We proceed by induction on n. In case $n = 0$, the conclusion is trivial. In case $n = p + 1$, then the induction hypothesis tells us that the largest sequence of a's in u_p is left(u_p) and its length is $\frac{m^{p+1}-m}{m-1} = \frac{m^n-m}{m-1}$. As $u_n = v_n u_p^{2m} v_n$, the longest sequence of a's in u_n can only be:

- either left(u_n) which is $v_n$left(u_p),
- or the sequence a's on each side of two consecutive u_p, i.e., left(u_p)left(u_p).

The word $v_n$left(u_p) contains $m^n + \frac{m^n-m}{m-1}$ a's that is $\frac{m^{n+1}-m}{m-1}$ a's while left(u_p)left(u_p) has as length $2\frac{m^n-m}{m-1}$. We have that

$$\begin{aligned}|v_n\text{left}(u_p)| - |\text{left}(u_p)\text{left}(u_p)| &= \frac{m^{n+1}-m}{m-1} - 2\frac{m^n-m}{m-1} \\ &= \frac{m^n(m-2)+m}{m-1}\end{aligned}$$

As $k > 0$ and $m = 16k + 4$, we have that $m - 2$ is positive. Therefore $|v_n\text{left}(u_p)| - |\text{left}(u_p)\text{left}(u_p)| \geq 0$ and left(u_n) is indeed the longest sequence of a's in u_n. □

Lemma 42 For every n, we have :

- $\Delta(\text{strip}(u_n)) = -2m^n - 2\frac{m^{n+1}-m}{m-1}$ and,
- for every factor u of u_n, we have that $|\Delta(u)| \leq |\Delta(\text{strip}(u_n)|$.

Proof
From Lemma 38, $\Delta(u_n) = -2m^n$. The definition of strip(u_n) entails that $\Delta(\text{strip}(u_n)) = \Delta(u_n) - 2|\text{left}(u_n)| = -2m^n - 2\frac{m^{n+1}-m}{m-1}$ as expected.

We prove the second statement of the lemma by induction on n.

When $n = 0$, we have that $u_n = abba$ and strip$(u_n) = bb$ and the conclusion is clear.

When $n = p + 1$, we have that $u_n = v_n u_p^{2m} v_n$. There are several possible cases: either u contains some b or u contains only a's.

In case u contains only a's, Lemma 41 tells us that $|u| \leq |\text{left}(u_n)| = \frac{m^{n+1}-m}{m-1} \leq 2m^n + 2\frac{m^{n+1}-m}{m-1} = |\Delta(\text{strip}(u_n))|$.

In case u contains some b's, and u is a factor of some u_p, the induction hypothesis tells use that $|\Delta(u)| \leq |\Delta(\text{strip}(u_p)| < |\Delta(\text{strip}(u_n))|$. We now consider that u contains one or more occurrences of u_p and is of the form $v_1 u_p^l v_2$ where v_1 is a suffix of u_p and v_2 is a prefix of u_p. So as to make $|\Delta(u)|$ maximal, the induction hypothesis tells us that it must be the case that $v_1 = \text{strip}(u_p)\text{right}(u_p)$ and $v_2 = \text{left}(u_p)\text{strip}(u_p)$. In that case, $|\Delta(u)| = 2|\Delta(u_p')| + l|\Delta(u_p)| - 2|\text{left}(u_p)|$. Then $|\Delta(u)|$ is maximal when l is, i.e. when $l = 2m - 2$ and in that case $u = \text{strip}(u_n)$. Hence the conclusion. □

Definition 43 We let:

- $\Lambda_n = m^{2k-n+1}$,
- $B_n = |\text{left}(u_n)| = \frac{m^{2k-n+1}-m}{m-1}$,
- $\sigma_n = |\Delta(\text{strip}(u_{2k-n}))| = 2m^{2k-n} + 2\frac{m^{2k-n+1}-m}{m-1} = 2\Lambda_{n+1} + 2B_n$.

Lemma 43 For every $0 < n < 2k$, we have the following identities:

- $\sigma_n > 2\Lambda_{n+1}$,
- $\sigma_n > 2B_n$,
- $B_n > \Lambda_{n+1}$,
- $2B_n > \sigma_{n+1}$.

Proof
The two first identities come from the fact that $\sigma_n = 2\Lambda_{n+1} + 2B_n$.

We prove that $B_n > \Lambda_{n+1}$:

$$\begin{aligned}
\frac{m^{2k-n+1}-m}{m-1} - m^{2k-n} &= \frac{m^{2k-n+1} - m - m^{2k-n+1} + m^{2k-n}}{m-1} \\
&= \frac{m^{2k-n-1}(m-1)}{m-1} \\
&= m^{2k-n-1} \\
&> 0
\end{aligned}$$

Let's now prove that $2B_n > \sigma_{n+1}$:

$$\begin{aligned} 2\frac{m^{2k-n+1}-m}{m-1} - 2m^{2k-n-1} - 2\frac{m^{2k-n}-m}{m-1} &= \frac{2}{m-1}(m^{2k-n+1} - m - 2m^{2k-n} + m + m^{2k-n-1}) \\ &= \frac{2m^{2k-n-1}}{m-1}(m^2 - 2m + 1) \\ &= \frac{2m^{2k-n-1}}{m-1}(m-1)^2 \\ &> 0 \end{aligned}$$

The last line comes from the fact that $m > 0$ as $m = 16k + 4$ and $k > 0$. □

Lemma 44 For all $n \in [0, 2k]$, if u is a factor of $w = \text{left}(u_{2k-n})u_{2k-n}\text{right}(u_{2k-n})$, then $|\Delta(u)| \leq \sigma_n$.

Proof
If u is a factor of u_{2k-n} then Lemma 42 tells us that indeed $|\Delta(u)| \leq \sigma_n$. So as to conclude, we only need to show that $|\text{left}(w)| = |\text{left}(u_{2k-n})\text{left}(u_{2k-n})| \leq \sigma_n$. This is clear since $\sigma_n - |\text{left}(u_{2k-n})\text{left}(u_{2k-n})| = 2m^{2k-n}$. □

Lemma 45 Given $0 < n < 2k$, if u is a factor of s_k so that u does not have a factor in a^l with $l \geq 2\Lambda_{n+1}$, then we have that $|\Delta(u)| \leq \sigma_n$.

Proof
A first case is when u does not contain any occurrence of b. Then, by hypothesis, $|\Delta(u)| = |u| < 2\Lambda_{n+1} < \sigma_n$ (Lemma 43).

In the rest of the proof, we let $w_p = \text{left}(u_{2k-p})u_{2k-p}\text{right}(u_{2k-p})$.

For the case where u contains occurrences of b, we show that u is a factor of w_n. For this we prove that if u is a factor of w_{p+1} with $p \geq n$, then u is a factor of w_p. Indeed, by definition,

$$w_{p+1} = \text{left}(u_{2k-p+1})v_{2k-p+1}u_{2k-p}^{2m}v_{2k-p+1}\text{right}(u_{2k-p+1}) \ .$$

If all the b's in u come from a unique u_{2k-p} then u is a factor of some word of the form $a^i u_{2k-p} a^j$. Now as, $\text{left}(w_p) = \text{left}(u_{2k-p})\text{left}(u_{2k-p})$ which contains $2B_p$ a's and $2B_p > 2\Lambda_{n+1}$ whenever $p \geq n$ (Lemma 43), w_p contains longer factors of a^* than what is authorized for u by the hypothesis and thus, in this case, u is a factor of $a^i u_{2k-p} a^j$ when $i = j = 2B_p$, i.e. u is a factor of w_p.

Now, in case u contains b's coming from different occurrences of u_{2k-p}, it must be the case that u has $\text{left}(u_{2k-p})\text{left}(u_{2k-p})$ as a factor, i.e. a sequence of a's larger than $2\Lambda_{n+1}$ as we have just seen in the previous case. So this case is impossible.

Now as u is a factor of $s_k = v_{2k}u_{2k}v_{2k}$ and v_{2k} is a prefix of $\text{left}(u_{2k})$, we have that s_k is a factor of w_0. Therefore, from what precedes, we know that u is a factor of w_n. Lemma 44 implies $|\Delta(u)| \leq \sigma_n$. □

Our goal is to define an invariant that is satisfied at each step of a k-derivation starting with $(s_k, \epsilon, \ldots, \epsilon)$ and that every tuples in which such a k-derivation ends must be different form $(\epsilon, \ldots, \epsilon)$. In other words, that s_k admits no k-derivation. Informally, this invariant is related to the fact that when there is a derivation from $(s_k, \epsilon, \ldots, \epsilon)$ to $\mathbf{w}$, then, for $0 < n \leq 2k$, there are n borders (prefixes or suffixes) of the components of w that contain a *large* number of a's. Here "*large*" depends on n and means "*more than* $3\Lambda_{n+1}$". The intuition is that during a k-derivation, the number of borders of the k words that are handled become occupied by a large number of a's is increasing. More precisely, when n borders have a number of a's that is larger than $3\Lambda_{n+1}$, when the derivation goes on, the number of a's on these borders may decrease below $3\Lambda_{n+1}$. But then, it must be the case that a border which did not have a large number of a's so far is now occupied with $3\Lambda_{n+2}$ a's. And now there are $n+1$ borders with a large number of a's; "*large*" now meaning with "*more than* $3\Lambda_{n+2}$".

We need now a number of concepts so as to formalize this idea. We call $B = [1,k] \times \{\mathbf{l}, \mathbf{r}\}$ the set of *borders* of a k-tuple, where $(i, \mathbf{l})$ means the *left border* of the i-th component, and $(i, \mathbf{r})$ means the *right border* of the i-th component.

Definition 44 Given a tuple $\mathbf{w} = (w_1, \ldots, w_k)$ and a *border* $\mathfrak{b} \in B$, we define the *edge-affix* of $\mathfrak{b}$ in $\mathbf{w}$, noted $\mathrm{affix}(\mathbf{w}, \mathfrak{b})$, by:

1. if $\mathfrak{b} = (i, \mathbf{l})$, then $\mathrm{affix}(\mathbf{w}, \mathfrak{b}) = \mathrm{left}(w_i)$,
2. if $\mathfrak{b} = (i, \mathbf{r})$, then $\mathrm{affix}(\mathbf{w}, \mathfrak{b}) = \mathrm{right}(w_i)$.

and the *edge-length* of $\mathfrak{b}$ in $\mathbf{w}$ as: $\mathrm{lg}(\mathbf{w}, \mathfrak{b}) = |\mathrm{affix}(\mathbf{w}, \mathfrak{b})|$.

From now on, when the context is clear, we use the expression *edge of* $\mathbf{w}$ to refer to either an element of the set B of borders, or to the corresponding edge-affix of $\mathbf{w}$. Notice that when w_i is in a^* we have that edge-affixes of $(i, \mathbf{l})$ and of $(i, \mathbf{r})$ overlap and are equal to w_i.

Definition 45 Given an integer n so that $1 \leq n \leq 2k$, we say that w in Σ^* is *n-heavy* when $a^{2\Lambda_{n+1}}$ is one of its factors. Otherwise u is *n-light*. For all tuple $\mathbf{w} = (u_1, \ldots, u_k)$ and all border $\mathfrak{b} = (i, \mathbf{l})$ or $\mathfrak{b} = (i, \mathbf{r})$, we say that $\mathfrak{b}$ is n-heavy (respectively n-light) for $\mathbf{w}$ if u_i is n-heavy (respectively n-light).

For all n-heavy border $\mathfrak{b}$ of $\mathbf{w}$, we define the *n-heavy segment* of $\mathbf{w}$ at border $\mathfrak{b}$, noted $\mathrm{seg}_h(n, \mathbf{w}, \mathfrak{b})$ as:

- the shortest word w such that $w\, a^{2\Lambda_{n+1}}$ is a prefix of u_i when $\mathfrak{o} = (i, \mathbf{l})$,
- the shortest word w such that $a^{2\Lambda_{n+1}}\, w$ is a suffix of u_i when $\mathfrak{o} = (i, \mathbf{r})$.

When $\mathfrak{b} = (i, \mathbf{l})$ or $\mathfrak{b} = (i, \mathbf{r})$ is an n-light border of $\mathbf{w}$, we define the *n-light segment* of $\mathbf{w}$ at border $\mathfrak{b}$ to simply be u_i.

Contrary to the case of n-heavy borders which are associated to segments that are always non-overlapping factors of a component, n-light borders are associated to the very same segment which is the component itself.

Definition 46 Given a $\mathbf{w}$ in $(\Sigma^*)^k$, we define an *n-pre-decomposition* of $\mathbf{w}$ as a tuple (E, H, L) where:

- $E \subseteq B$ (the set of *edges*) is a set of n borders.
- $H \subseteq B \setminus E$ (the set of *heavy borders*) is the set of all n-heavy borders of $\mathbf{w}$ that are not in E.
- $L \subseteq \{1, \ldots, k\}$ (the set of *light components*) is the set of all n-light components of $\mathbf{w}$.

An n-pre-decomposition is an *n-decomposition* when, for every border $\mathfrak{e} \in E$:

$$\lg(\mathbf{w}, \mathfrak{e}) \geq \Lambda_n - k + \Delta(\mathbf{w}) - (|H| + |L|)\sigma_n - \Delta_h(H) - \Delta_l(L) \qquad (\xi^n)$$

with $\Delta_h(H) = \Sigma_{\mathfrak{b} \in H} \Delta(\mathrm{seg}_h(n, \mathbf{w}, \mathfrak{b}))$ and $\Delta_l(L) = \Sigma_{i \in L} \Delta(w_i)$.

An n-decomposition of $\mathbf{w}$ is *maximal* if there exists no n'-decomposition of $\mathbf{w}$ with $n < n'$.

Equation (ξ^n) helps us to prove that edges remain large during k-derivations starting at s_k. In particular, as shown by the lemma below, an edge $\mathfrak{e}$ verifies equation (ξ^n) only if it is longer than $3\Lambda_{n+1}$. In particular, it entails that in an n-decomposition, every border is either in E, H or is the border of a component in L.

Lemma 46 Given a n-decomposition (E, H, L) of a tuple $\mathbf{w}$ with $|\Delta(w)| \leq k$, we have that for every $\mathfrak{e}$ in E, $\lg(\mathbf{w}, \mathfrak{e}) > 3\Lambda_{n+1}$.

Proof
As (E, H, L) is an n-decomposition, we have that every $\mathfrak{e}$ in E satisfies equation (ξ^n). In the term $\Lambda_n - k + \Delta(\mathbf{w}) - (|H| + |L|)\sigma_n - \Delta_h(H) - \Delta_l(L)$ we have that $|H| + |L| \leq 2k$. Moreover as for every $\mathfrak{b}$ in H, we have that $\mathrm{seg}_h(n, \mathbf{w}, \mathfrak{o})$ does not contain $a^{2\Lambda_{n+1}}$ as a factor and thus, from Lemma 45, we know that $|\Delta(\mathrm{seg}_h(n, \mathbf{w}, \mathfrak{b}))| \leq \sigma_n$. Similarly, for every i in L, $\Delta(w_i) \leq \sigma_n$. Therefore, $|\Delta_h(H) + \Delta_l(L)| + (|H| + |L|)\sigma_n \leq 4k\sigma_n$. Finally since $|\Delta(\mathbf{w})| \leq k$ we have:

$$\Lambda_n - k + \Delta(\mathbf{w}) - (|H| + |L|)\sigma_n - \Delta_h(H) - \Delta_l(L) \geq \Lambda_n - 2k - 4k\sigma_n$$

We now prove that $\Lambda_n - 2k - 4k\sigma_n - 3\Lambda_{n+1} > 0$:

$$\begin{aligned}
\Lambda_n - 2k - 4k\sigma_n - 3\Lambda_{n+1} &= m^{2k-n+1} - 2k - 4k\left(2m^{2k-n} + 2\frac{m^{2k-n+1} - m}{m-1}\right) - 3m^{2k-n} \\
&= m^{2k-n}\left(m - 8k - 3 - \frac{8km}{m-1}\right) + \frac{8km}{m-1} - 2k \\
&= m^{2k-n}\frac{m^2 - m - 8km + 8k - 3m + 3 - 8km}{m-1} + \frac{8km - 2km + 2k}{m-1} \\
&= m^{2k-n}\frac{m(m - 4 - 16k) + 3 + 8k}{m-1} + \frac{6km + 2k}{m-1}
\end{aligned}$$

As $m = 16k + 4$, we obtain that

$$\Lambda_n - 2k - 4k\sigma_n - 3\Lambda_{n+1} = m^{2k-n}\frac{3+8k}{m-1} + \frac{6km+2k}{m-1}$$

so that, since $k > 0$, $\Lambda_n - 2k - 4k\sigma_n - 3\Lambda_{n+1} > 0$. Hence the conclusion. □

Lemma 47 Given (E, H, L) an n-pre-decomposition of $\mathbf{w}$, if for $\mathfrak{e}$ in E we have that $\lg(\mathbf{w}, \mathfrak{e}) \geq \Lambda_n$ then $\mathfrak{e}$ verifies (ξ^n).

Proof
In the term

$$\Lambda_n - k + \Delta(\mathbf{w}) - (|H| + |L|)\sigma_n - \Delta_h(H) - \Delta_l(L)$$

we have $|\Delta(\mathbf{w})| \leq k$, and $|\Delta_h(H) + \Delta_l(L)| \leq (|H| + |L|)\sigma_n$ and thus we have that:

$$\Lambda_n \geq \Lambda_n - k + \Delta(\mathbf{w}) - (|H| + |L|)\sigma_n - \Delta_h(H) - \Delta_l(L)$$

□

Lemma 48 Let (E, H, L) an n-decomposition of $\mathbf{w}$. If that decomposition is maximal then, for all $\mathfrak{b} \in H$, $\mathrm{seg}_h(n, \mathbf{w}, \mathfrak{b}) \neq \varepsilon$.

Proof
Ad absurdum, assume that there exists $\mathfrak{b}$ in H with $\mathrm{seg}_h(n, \mathbf{w}, \mathfrak{b}) = \varepsilon$ and show that there is an $n+1$-decomposition of $\mathbf{w}$.

Because $\mathrm{seg}_h(n, \mathbf{w}, \mathfrak{b}) = \varepsilon$, $\lg(\mathbf{w}, \mathfrak{b}) \geq 2\Lambda_{n+1}$. So we define a new set $E' = E \cup \{\mathfrak{b}\}$ of edges. Now we show that (E', H', L') is an $(n+1)$-decomposition of $\mathbf{w}$, where H' is the set of borders in $B \setminus E'$ that are $(n+1)$-heavy in $\mathbf{w}$, and L' is the set of $(n+1)$-light components of $\mathbf{w}$.

In order to check equation (ξ^{n+1}), it is enough to show that borders in E' have edge-length at least Λ_{n+1}. So (ξ^{n+1}) holds for $\mathfrak{o}$. Since (E, H, L) is an n-decomposition, for all $\mathfrak{e} \in E$, $\lg(\mathbf{w}, \mathfrak{e}) \geq 3\Lambda_{n+1}$. Thus equation (ξ^{n+1}) holds for all borders in E', so (E', H', L') is an $(n+1)$-decomposition of $\mathbf{w}$.

□

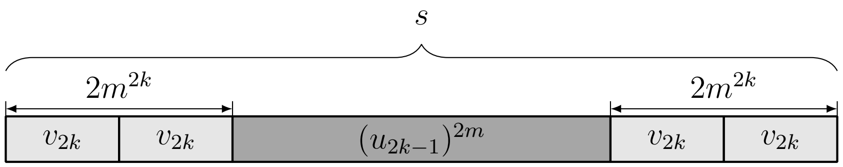
Now, for $0 < n \leq 2k$, when a tuple admits an n-decomposition, not all its components can be the empty string as it has n borders with a number of a's larger than $3\Lambda_{n+1} > 0$. Thus, if every tuple that occurs in a k-derivation starting at s_k has an n-decomposition for some $n > 0$, then it proves that the word s_k admits no k-derivation. We are thus going to prove the following lemma.

Lemma 49 If there is a k-derivation from $(s_k, \epsilon, \ldots, \epsilon)$ to $\mathbf{w}$, then there is an n-decomposition of $\mathbf{w}$ for some n in $[1, 2k]$.

The rest of the section is devoted to the proof of that lemma. We first prove that $(s_k, \epsilon, \ldots, \epsilon)$ has a 2-decomposition and that if $\mathbf{ww}'$ is a k-derivation so that $\mathbf{w}$ has an n-decomposition, then $\mathbf{w}'$ has an m-decomposition for some m.

Base case: $(s_k, \epsilon, \ldots, \epsilon)$ has a 2-decomposition.

We have that $s_k = v_{2k}v_{2k}u_{2k-1}^{2m}v_{2k}v_{2k}$. So $v_{2k}\, v_{2k} = a^{2m^{2k}}$ is both a prefix and a suffix of s_k, and that we represent pictorially by:



As a convention for representing words, we use a light gray for factors that are in a^*, for those we usually add an indication about their length (the number of occurrences of a in them). That way we see edges and their length more clearly. It also makes clearer the difference between n-heavy and n-light words: n-heavy words have a factor in a^* of length $2\Lambda_{n+1}$ but n-light words don't. We use a darker gray to represent any other kind of factor. We will use this convention in all subsequent representations of words as illustrations in the proof.

Here we have two edges $(1, \mathbf{l})$ and $(1, \mathbf{r})$ of length greater than $2m^{2k}$ in $\mathbf{w}$. So we define $E = \{(1, \mathbf{l}), (1, \mathbf{r})\}$, $H = \emptyset$ and $L = \{2, 3, \ldots, k\}$. In order to prove that (E, H, L) is a 2-decomposition of $\mathbf{w}$ we need to show that equation (ξ^2) holds for all edge $\mathfrak{e} \in E$. This is simply because we have that $2m^{2k}$ is greater than $\Lambda_2 = m^{2k-1}$ and thus by Lemma 47 the edges in E satisfy (ξ^2).

So there is a 2-decomposition of $(s_k, \epsilon, \ldots, \epsilon)$.

Inductive step: $\mathbf{w}\mathbf{w}'$ is a k-derivation with $\mathbf{w} = (w_1, \ldots, w_k)$ and $\mathbf{w} = (w'_1, \ldots, w'_k)$.

We assume is that $\mathbf{w}$ has an n-decomposition for some n in $[1, 2k]$. We also assume that this n-decomposition (E, H, L) is maximal (for n) which implies, according to Lemma 48, that heavy segments of $\mathbf{w}$ in H are not the empty string ε.

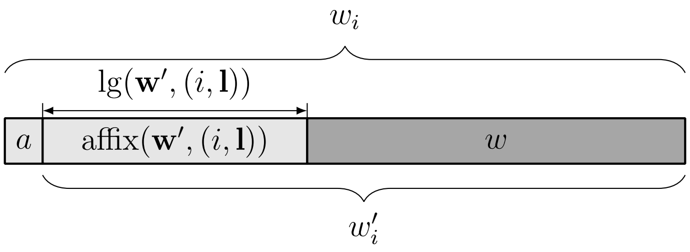
There are two basic cases based on which derivation step transforms $\mathbf{w}$ into $\mathbf{w}'$: whether it is a letter erasure or a (left or right) word splitting. Then each case is split into several cases depending on where the derivation step occurs in $\mathbf{w}$.

Case 1 : erasure of a letter

First we consider the case where the derivation step erases a letter on one side of a w_i. The cases where the letter is erased on the right or on the left of w_i are symetric. Without loss of generality, we thus assume that it is erased on the left. There are three different subcases depending on whether $(i, \mathbf{l})$ is in E, H or i is in L. In each of these subcases we show that (E, H, L) is still an n-decomposition of $\mathbf{w}'$.

Case 1.1 : $(i, \mathbf{l})$ is in E

The letter erased must be an a:


w_i
$\mathrm{lg}(\mathbf{w}', (i, \mathbf{l}))$
a
$\mathrm{affix}(\mathbf{w}', (i, \mathbf{l}))$
w
w'_i

In order to prove that (E, H, L) is an n-decomposition of $\mathbf{w}'$, we first need to check that H and L are sets of n-heavy borders and n-light components of $\mathbf{w}_2$. The only component that is changing from $\mathbf{w}$ to $\mathbf{w}'$ is u_i which loses one a, we need to prove that if $(i, \mathbf{r})$ is n-heavy in $\mathbf{w}$ then it is also n-heavy in $\mathbf{w}'$. It is the case because equation (ξ^n) holds for edge $(i, \mathbf{l})$ in $\mathbf{w}$ which implies, according to Lemma 46, that $\mathrm{lg}(\mathbf{w}, (i, \mathbf{l})) \geq 3\Lambda_{n+1}$ and so w'_i has $a^{3\Lambda_{n+1}} - 1$ as a factor.

Now we check (ξ^n), for all edge $\mathfrak{e} \in E$:

$$\mathrm{lg}(\mathbf{w}', \mathfrak{e}) \geq \Lambda_n - k + \Delta(\mathbf{w}') - (|H| + |L|)\Delta_n - \Delta_h(H) - \Delta_l(L) \qquad (\xi^n)$$

Compared to the identities concerning $\mathbf{w}$, only two terms in these inequalities have changed: $\mathrm{lg}(\mathbf{w}', (i, \mathbf{l}))$ (and $\mathrm{lg}(\mathbf{w}', (i, \mathbf{r}))$ if $(i, \mathbf{r})$ is in E and $w_i \in a^*$), and $\Delta(\mathbf{w}')$. The term $\mathrm{lg}(\mathbf{w}', (i, \mathbf{l}))$ is decremented (same for $\mathrm{lg}(\mathbf{w}', (i, \mathbf{r}))$ if $(i, \mathbf{r})$ is in E and $w_i \in a^*$) and $\Delta(\mathbf{w})$ is incremented. So (ξ^n) must hold for $(i, \mathbf{l})$ and all other edges in E on tuple $\mathbf{w}'$.

So (E, H, L) is indeed an n-decomposition of $\mathbf{w}'$.

Case 1.2 : $(i, \mathbf{l})$ is in H

If $(i, \mathbf{l}) \in H$ then, according to Lemma 48, the corresponding n-heavy segment of $\mathbf{w}'$ is not the empty string ε. So, after erasing one letter, it is still a n-heavy segment of $\mathbf{w}'$. In order to prove that (E, H, L) is an n-decomposition of $\mathbf{w}'$ we only need to check equation (ξ^n). The only terms that change in (ξ^n) are $\Delta(\mathbf{w})$ and $\Delta_h(H)$; they change so that $\Delta_h(H) - \Delta(\mathbf{w}')$ remains constant, consequently (ξ^n) still holds. Therefore (E, H, L) is an n-decomposition of $\mathbf{w}'$.

Case 1.3 : i is in L

Here again, it is enough to show (ξ^n) still hold for $\mathbf{w}'$ to prove that (E, H, L) is an n-decomposition. The only terms that change in (ξ^n) are $\Delta(\mathbf{w}')$ and $\Delta_l(L)$, and they change so that $\Delta_l(L) - \Delta(\mathbf{w}')$ remains constant, so equation (ξ^n) still holds. Therefore (E, H, L) is an n-decomposition of $\mathbf{w}'$.

We have shown that if there exists an n-decomposition (E, H, L) of $\mathbf{w}$ and if $\mathbf{w}\mathbf{w}'$ is a k-derivation erases a letter, then there exists an n-decomposition of $\mathbf{w}'$.

Case 2 : split of a component w_i of $\mathbf{w}$

The cases where w_i is split between w'_i and w'_{i+1} and between w'_{i-1} and w'_i are symetric and we will thus only treat the case where w_i is split between w'_i and w'_{i+1}. In that case $w_i = w'_i\, w'_{i+1}$ and $w_{i+1} = \varepsilon$. We already know that every $\mathfrak{e}$ in E has length at least $3\Lambda_{n+1}$, so no border of w_{i+1} can be in E and, since they cannot be in H either: $i + 1 \in L$. Then there are different subcases depending on how w_i is split and on whether $(i, \mathbf{l})$ and $(i, \mathbf{r})$ are in L, E, H or if one is in E and the other is in H. For each of these subcases, in order to find a decomposition of $\mathbf{w}'$, we use one of the two following methods :

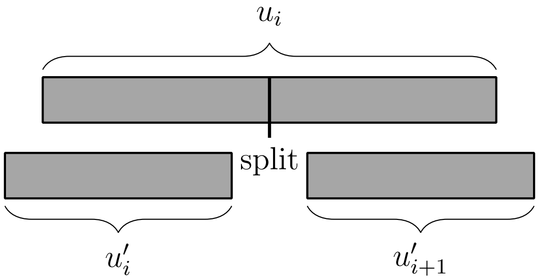
- we use the fact that (E, H, L) is an n-decomposition of $\mathbf{w}$, make a few changes to it in order to obtain an n-decomposition of $\mathbf{w}'$. In order to prove that equation (ξ^n) still holds for the new n-decomposition we rely on the fact that (ξ^n) holds for (E, H, L) and analyze the changes in (ξ^n) introduced by the derivation step.

- We find an edge $\mathfrak{e}$ of $\mathbf{w}'$ and add it to E in order to get an $(n+1)$-decomposition of $\mathbf{w}'$. Note that with this method, the sets of $(n+1)$-heavy and $(n+1)$-light segments can be very different from the n-heavy and n-light segments of $\mathbf{w}$. So we have a different way of proving equation (ξ^{n+1}) for this new decomposition. The fact that for every edge that a in E for the n-decomposition of $\mathbf{w}$ have length at least $3\Lambda_{n+1}$ implies that (ξ^{n+1}) holds for these edges for any sets H' and L' of $(n+1)$-heavy and $(n+1)$-light segments. It therefore only remains to prove that $\mathfrak{e}$ also verifies equation (ξ^{n+1}). For this we use the fact that $\lg(\mathbf{w}, \mathfrak{e}) \geq \Lambda_{n+1}$ implies (ξ^{n+1}) for any H' and L'. Therefore, in order to get an $(n+1)$-decomposition using this method, we only need to find a new edge $\mathfrak{e}$ of length at least Λ_{n+1}.

In every case, we allow ourselves to assume that the n-decomposition (E, H, L) of $\mathbf{w}$ is maximal (for n). Let us now go into the details of the different cases.

A first easy case is when u_i is n-light.

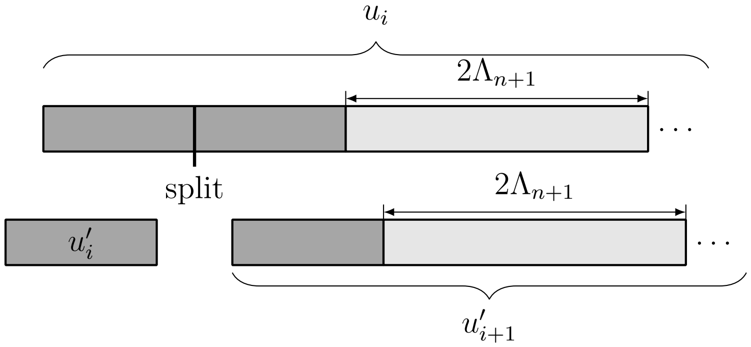
Case 2.1 : $i \in L$



Since u_i does not have $a^{2\Lambda_{n+1}}$ as a factor, neither do u'_i and u'_{i+1}. Then u'_i and u'_{i+1} are n-light segments of $\mathbf{w}'$, so L is the set of n-light components of $\mathbf{w}'$. We show that (E, H, L) is an n-decomposition of $\mathbf{w}'$. In equation (ξ^n) we have $|L'| = |L|$ and since $u_i = u'_i u'_{i+1}$, we also have $\Delta_l(L') = \Delta_l(L)$, so (ξ^n) still holds for all the edges in E. Therefore (E, H, L) is an n-decomposition of $\mathbf{w}'$.

In all the remaining cases, because u_i contains $a^{2\Lambda_{n+1}}$ as a factor, each border of u_i is either in E or in H. A first case is when a border of u_i is in H and the split of u_i splits the corresponding n-heavy segment:

Case 2.2 : $(i, \mathbf{l}) \in H$ **and** u'_i **is a prefix of** $\mathrm{seg}_h(n, \mathbf{w}, (i, \mathbf{l}))$



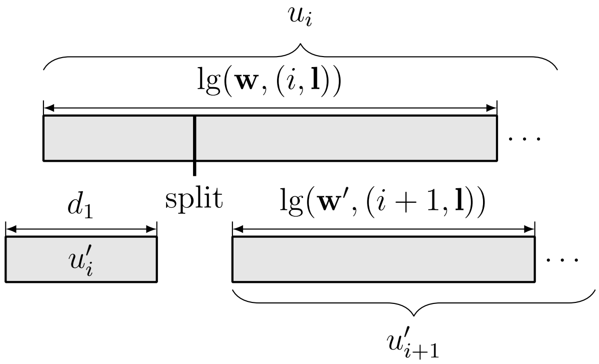
In this case u'_i is n-light and u'_{i+1} has $a^{2\Lambda_{n+1}}$ as a factor. The set of n-light components of $\mathbf{w}'$ is $L' = L \setminus \{i+1\} \cup \{i\}$. If $(i, \mathbf{r})$ is an edge in E, then we define E' and H' by $E' = E \setminus \{(i, \mathbf{r})\} \cup \{(i+1, \mathbf{r})\}$ and $H' = H \setminus \{(i, \mathbf{l})\} \cup \{(i+1, \mathbf{l})\}$. Otherwise, when $(i, \mathbf{r}) \in H$, we let $E' = E$ and $H' = H \setminus \{(i, \mathbf{l}), (i, \mathbf{r})\} \cup \{(i+1, \mathbf{l}), (i+1, \mathbf{r})\}$. Now, in order to prove that (E', H', L') is an n-decomposition of $\mathbf{w}'$, we only need to show that equation (ξ^n) holds for all edges in E'.

The only terms that are changing in (ξ^n) are: $\Delta_h(H') = \Delta_h(H) - \Delta(u'_i)$ and $\Delta_l(L') = \Delta_l(L) + \Delta(u'_i)$. Therefore, $\Delta_h(H') + \Delta_l(L') = \Delta_h(H) + \Delta_l(L)$ and (ξ^n) still holds for all edges in E' and (E', H', L') is a n-decomposition of $\mathbf{w}'$.

Here we assumed that the split n-heavy segment is on the left of u_i, but the case where it is on the right of u_i is symmetric.

Another similar case is when a border of u_i is an edge and a small piece of it is split from the rest of u_i. Here a small piece means shorter than $2\Lambda_{n+1}$, so that the small piece is n-light. Again we have two symetric cases for the side of u_i on which the split happens, we will only deal with the case where the split happens on the left side of u_i:

Case 2.3 : $(i, \mathbf{l}) \in E$ **and** u'_i **is shorter than** $2\Lambda_{n+1}$


u_i
$\lg(\mathbf{w}, (i, \mathbf{l}))$
$\dots$
d_1
split
$\lg(\mathbf{w}', (i+1, \mathbf{l}))$
u'_i
$\dots$
u'_{i+1}

In this case the length $\lg(\mathbf{w}, (i, \mathbf{l}))$ of the edge $(i, \mathbf{l})$ of $\mathbf{w}$ is the sum of the length $d_1 \leq 2\Lambda_{n+1}$ of u'_i and the length $\lg(\mathbf{w}', (i+1, \mathbf{l}))$ of the edge $(i+1, \mathbf{l})$ of $\mathbf{w}'$. The new sets E', H' and L' are simply obtained from E, H, L as follows:

- $L' = L - \{i+1\} \cup \{i\}$,

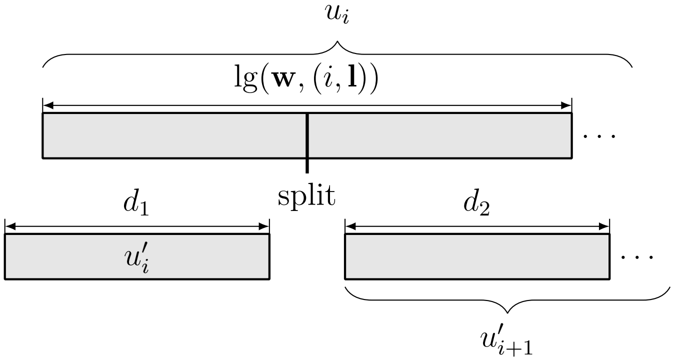
- $E' = \left\{ \begin{array}{l} E - \{(i,\mathbf{l}),(i,\mathbf{r})\} \cup \{(i+1,\mathbf{l}),(i+1,\mathbf{r})\} \text{ when } (i,\mathbf{r}) \in E \\ E - \{(i,\mathbf{l})\} \cup \{(i+1,\mathbf{l})\} \text{ otherwise} \end{array} \right.$,
- $H' = \left\{ \begin{array}{l} H - \{(i,\mathbf{r})\} \cup \{(i+1,\mathbf{r})\} \text{ when } (i,\mathbf{r}) \in H \\ H \cup \text{ otherwise} \end{array} \right.$

Notice that, by hypothesis, $(i,\mathbf{r})$ is either in H or in E.

Now, in order to prove that (E',H',L') is a n-decomposition we first need to check equation (ξ^n). The only terms that can change in (ξ^n) are: $\Delta_l(L') = \Delta_l(L)+d_1$, $\lg(\mathbf{w}',(i+1,\mathbf{l})) = \lg(\mathbf{w},(i,\mathbf{l}))-d_1$ and, if $(i,\mathbf{r}) \in E$ and $u_i \in a^*$, then $\lg(\mathbf{w}',(i+1,\mathbf{r})) = \lg(\mathbf{w},(i,\mathbf{l}))-d_1$. From this we deduce that equation (ξ^n) holds for edge $(i+1,\mathbf{l})$ and also for the other edges in E'. So (E',H',L') is an n-decomposition of $\mathbf{w}'$.

The next case is when a longer part (longer than $2\Lambda_{n+1}$) of an edge is split. Again we consider only the case where the edge being split is $(i,\mathbf{l})$ because the case where it is $(i,\mathbf{r})$ is symetric.

Case 2.4: $(i,\mathbf{l}) \in E$ and $u'_i \in a^*$ is longer than $2\Lambda_{n+1}$



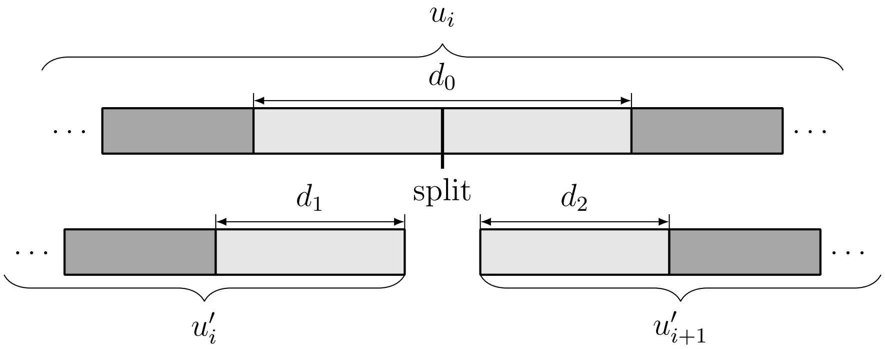
In this case we have two new edges $(i,\mathbf{l})$ and $(i,\mathbf{r})$ of $\mathbf{w}'$ coming from the split of one edge $(i,\mathbf{l})$ of $\mathbf{w}$, so we have an $(n+1)$-decomposition of $\mathbf{w}'$. Remember that, in order for (ξ^{n+1}) to hold, we only need to check that we have $n+1$ edges of length at least Λ_{n+1}. It is the case for edges $(i,\mathbf{l})$ and $(i,\mathbf{r})$ because the length d_1 of u'_i is greater than $2\Lambda_{n+1}$. Since (ξ^n) implies that edges are longer than $3\Lambda_{n+1}$, it is also the case for the other edges in E untouched by the split.

In the event that $(i,\mathbf{r})$ was an edge of $\mathbf{w}$ and that its length changed after the split, we still need to check that the corresponding edge $(i+1,\mathbf{r})$ of $\mathbf{w}'$ is longer than Λ_{n+1}. This event occurs only if u_i was in a^* and so $u'_{i+1} = a^{d_2}$. We have already dealt with the case where a small piece (shorter than $2\Lambda_{n+1}$) is split from the edge $(i,\mathbf{r})$ of $\mathbf{w}$: it is the symetric of case 2.4. We can then consider w.l.o.g. the piece split from $(i,\mathbf{r})$, i.e. u'_{i+1}, is longer than $2\Lambda_{n+1}$. Therefore (ξ^{n+1}) holds for $(i+1,\mathbf{r})$ in $\mathbf{w}'$.

So we have a $(n+1)$-decomposition of $\mathbf{w}'$.

Now we have seen all the cases where the split cuts an edge-affix or a heavy segment into two. The cases we have left are when the split happens further away from the borders of u_i. The next case is when the split cuts a factor $a^{2\Lambda_{n+1}}$ of u_i which is not in an edge-affix in two:

Case 2.5: $u'_i = w_1\, b\, a^{d_1}$**,** $u'_{i+1} = a^{d_2}\, b\, w_2$ **with** $d_1 + d_2 \geq 2\Lambda_{n+1}$

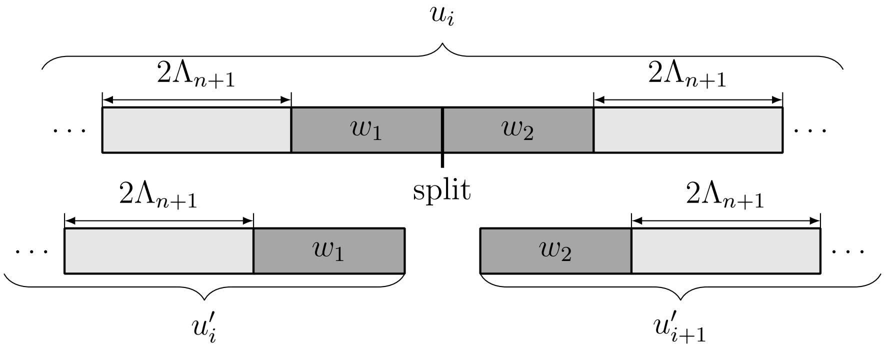


Here $2\Lambda_{n+1} \leq d_0$ and $d_0 = d_1 + d_2$. Then either $d_1 \geq \Lambda_{n+1}$ or $d_2 \geq \Lambda_{n+1}$, and in either of these cases, we have a new edge ($(i, \mathbf{r})$ or $(i+1, \mathbf{l})$) of $\mathbf{w}'$ of length greater than Λ_{n+1}. So, using the same method as in the previous case, we get an $(n+1)$-decomposition of $\mathbf{w}'$.

The last case is when the split happens in neither a heavy segment, an edge-affix nor a factor $a^{2\Lambda_{n+1}}$ of u_i.

Case 2.6: u_i **is** n**-heavy is split neither in one of its affix nor in a factor** $a^{2\Lambda_{n+1}}$

As u_i is not split in a factor $a^{2\Lambda_{n+1}}$, if there is no occurrence of $a^{2\Lambda_{n+1}}$ to the right, then it must be the case that the split is in the right-affix of u_i. Thus by hypothesis, there must be an occurrence of $a^{2\Lambda_{n+1}}$ in u'_{i+1}. Similarly there must be an occurrence of $a^{2\Lambda_{n+1}}$ in u'_i.



In this case u'_i and u'_{i+1} are n-heavy, so we define the words $w_1 = \mathrm{seg}_h(n, \mathbf{w}', (i, \mathbf{r}))$ and $w_2 = \mathrm{seg}_h(n, \mathbf{w}', (i{+}1, \mathbf{l}))$. We construct an n-decomposition (E', H', L') of $\mathbf{w}'$. If $(i, \mathbf{r}) \in E$ then the edge $(i, \mathbf{r})$ of $\mathbf{w}$ becomes the edge $(i+1, \mathbf{r})$ of $\mathbf{w}'$: $E' = E \setminus \{(i, \mathbf{r})\} \cup \{(i+1, \mathbf{r})\}$ and $H' = H \cup \{(i, \mathbf{r}), (i+1, \mathbf{l})\}$, or else $E' = E$ and $H' = H \cup \{(i+1, \mathbf{l}), (i+1, \mathbf{r})\}$. In any case u_{i+1} is n-light but u'_i and u'_{i+1} are n-heavy so $L' = L \setminus \{i+1\}$. Now we only need to check equation (ξ^n).

In every cases, we have:

$$\begin{aligned}
|L'| &= |L| - 1 && (8.1)\\
|H'| &= |H| + 2 && (8.2)\\
\Delta(H') &= \Delta(H) + \Delta(w_1) + \Delta(w_2) && (8.3)
\end{aligned}$$

Now, for all edge $\mathfrak{e}' \in E'$, there is an element of $\mathfrak{e}$ so that $\lg(\mathbf{w}', \mathfrak{e}') = \lg(\mathbf{w}, \mathfrak{e})$, we have:

$$\lg(\mathbf{w}', \mathfrak{e}) \geq \Lambda_n - k + \Delta(\mathbf{w}) - (|H| + |L|)\sigma_n - \Delta_h(H) - \Delta_l(L) \ .$$

Furthermore, using equations (8.1), we obtain:

$$\begin{aligned}
\lg(\mathbf{w}', \mathfrak{e}) &\geq \Lambda_n - k + \Delta(\mathbf{w}') - (|H'| + |L'| - 1)\sigma_n - \Delta_h(H') - \Delta(w_1) - \Delta(w_2) - \Delta_l(L')\\
&\geq \Lambda_n - k + \Delta(\mathbf{w}') - (|H'| + |L'|)\sigma_n + \Delta_h(H') + \Delta_l(L') - (\sigma_n + \Delta(w_1) + \Delta(w_2))
\end{aligned}$$

Since $w_1 w_2$ does not contain an occurrence of $a^{2\Lambda_{n+1}}$, Lemma 45 tells us that

$$|\Delta(w_1 w_2)| = |\Delta(w_1) + \Delta(w_2)| \leq \sigma_n \ .$$

So $\sigma_n + \Delta(w_1) + \Delta(w_2) \geq 0$ and thus:

$$\lg(\mathbf{w}', \mathfrak{e}) \geq \Lambda_n - k + \Delta(\mathbf{w}') - (|H'| + |L'|)\sigma_n + \Delta_h(H') + \Delta_l(L')$$

which finally proves that (E', H', L') is a n-decomposition of $\mathbf{w}'$.

We have shown that, in any case, if there exists an n-decomposition (E, H, L) of $\mathbf{w}$ and $\mathbf{w}\mathbf{w}'$ is a k-derivation then there is an m-decomposition of $\mathbf{w}'$.

From this we can conclude that if there is a k-derivation from $(s_k, \epsilon, \ldots, \epsilon)$ to $\mathbf{w}$, then $\mathbf{w}$ has an n-decomposition for n in $[1, 2k]$. A consequence is that this n-decomposition must contain n edges whose length is greater than $3\Lambda_{n+1} = 3m^{2k-n}$ which is strictly positive and $\mathbf{w}$ cannot be equal to $(\epsilon, \ldots, \epsilon)$. This implies the following proposition.

Proposition 6 *For every $k > 0$, s_k admits no k-derivation.*

Now Theorem 31 is a consequence of this proposition and Proposition 5.

Part V

Conclusion

Chapter 9

Conclusion

9.1 Contributions

In this thesis, we presented our work on the formal modelisation of tree transformations, first on a practical level as part of the CoLiS project, then on a more theoretical level with the study of tree transducers and their properties using tools of functional programming. Finally we have proven an interesting result on classes of word languages.

Our first contribution is the implementation of a tool checking Debian installation scripts for bugs in their interaction with the file system, and the design of its algorithm. This contribution includes the design of a model of tree transformations and its optimization for the specific use of checking executions of Shell scripts, the design of algorithms on both the general and optimized version of this model and a complexity analysis showing the sources of NP-hardness in the problem.

Our main theoretical contribution is the design of the model of High-Order Deterministic tree transducers (HODT) which generalizes several known classes of transducers. Our work includes the design of algorithms for computing the composition of HODT and translating HODT into other models of transducers. We also shown interesting properties of HODT, notably that linear and almost-linear HODT are as expressive as transductions defined by Monadic Second-Order logic. With this work we have shown that a functional programming approach to tree transducers provides an insightful view on those models.

Our last contribution consists in a proof that the language MIX, which is the commutative closure of the Dyck language, is not included in several classes of languages on words, including the class of EDT0L languages and the class of output languages of word-to-word transductions defined by Monadic Second-Order logic, which was an open problem.

9.2 Perspectives

9.2.1 Our implementation in CoLiS

One of our short term goals would to adapt our implementation to the latest version of the concrete interpreter. This supports a lot more functionalities of scripts than the one we based our work on. In addition to improving the coverage of corpuses of maintainer scripts, it also tests full installation scenarios instead of testing maintainer scripts in isolation. Once this is done we could again compare our results with those of our collaborators in the project, and maybe detect pertinent bugs to report.

Since our implementation is fast enough to run on large corpuses of scripts, we could also model file systems and commands more accurately. In particular removing the big approximation of the `cp -r` command on the case where its destination directory is not empty. This would require to enlarge our model for representing tree transformations.

A third possibility would be to use a similar tool in order to detect if an uninstallation script puts the file system back as it was before installation. To check this we only need to test if the composition the installation script with the uninstallation script performs the identity function. This would require to implement our algorithms for composition and for checking if a tree transformation is the identity.

9.2.2 Implementing $\mathrm{HODTR}_{\mathrm{lin}}$ in CoLiS

Our $\mathrm{HODTR}_{\mathrm{lin}}$ model could be used to represent the behaviour of Unix commands more accurately for the CoLiS project. The first hurdle in this endeavour would be to adapt the $\mathrm{HODTR}_{\mathrm{lin}}$ model to the formalism of feature trees i.e. unranked unordered trees. *Unranked* trees can be represented as *ranked* binary trees. *Unordered* trees can be modelized using *ordered* trees with a commmutativity property of the children as in [8], but our case is not that simple because children are distinguished by labels on edges (filenames). In this way file systems are more akin to ordered trees than unordered trees. We would need a model where child nodes are either accessed directly using their filenames, or accessed commutatively compared to other child nodes as in [8].

9.2.3 Testing equivalence of HODTR

Testing the equivalence between two transducers would be a very useful verification tool. Since the equivalence of MSOT transducers is decidable [11], the equivalent model of $\mathrm{HODTR}_{\mathrm{lin}}$ also has an algorithm for checking the equivalence of two transducers. If the equivalence is too hard to check, we could start by checking *origin* equivalence as defined in [9]. Origin equivalence is a downgraded version of equivalence where transducers, in addition to associating an output tree with an input tree, specify which nodes of the output tree come from which node of the input. Two transducers are origin-equivalent when they compute the same output nodes from the same input nodes. This notion of equivalence is easier to check because two origin-equivalent transducers must have similar ways of computing their output.

Bibliography

[1] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

[2] R. Alur and L. D'Antoni. Streaming tree transducers. *J. ACM*, 64(5):31:1–31:55, 2017.

[3] Rajeev Alur. Streaming string transducers. In Lev D. Beklemishev and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information and Computation - 18th International Workshop, WoLLIC 2011, Philadelphia, PA, USA, May 18-20, 2011. Proceedings*, volume 6642 of *Lecture Notes in Computer Science*, page 1. Springer, 2011. URL: `http://dx.doi.org/10.1007/978-3-642-20920-8_1`, `doi:10.1007/978-3-642-20920-8_1`.

[4] Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. In *FSTTCS*, pages 1–12, 2010.

[5] R. M. Amadio and P-L. Curien. *Domains and Lambda-Calculi.* Cambridge Tracts in Theoretical Computer Science. Cambridge Univ. Press, 1998.

[6] Peter R.J. Asveld. Controlled iteration grammars and full hyper-afl's. *Information and Control*, 34(3):248 – 269, 1977. URL: `http://www.sciencedirect.com/science/article/pii/S0019995877903084`, `doi:https://doi.org/10.1016/S0019-9958(77)90308-4`.

[7] Roderick Bloem and Joost Engelfriet. A comparison of tree transductions defined by monadic second order logic and by attribute grammars. *J. Comput. Syst. Sci.*, 61(1):1–50, 2000. `doi:10.1006/jcss.1999.1684`.

[8] Adrien Boiret, Vincent Hugot, Joachim Niehren, and Ralf Treinen. Deterministic automata for unordered trees. *arXiv preprint arXiv:1408.5966*, 2014.

[9] Mikoλaj Bojańczyk. Transducers with origin information. In *International Colloquium on Automata, Languages, and Programming*, pages 26–37. Springer, 2014.

[10] B. Courcelle. Monadic second-order definable graph transductions: a survey. *Theoretical Computer Science*, 126(1):53–75, 1994.

[11] B. Courcelle. Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. In Rozenberg, editor, *Handbook of Graph Grammars*, 1997.

[12] J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and mso definable tree translations. *Information and Computation*, 154(1):34 – 91, 1999.

[13] J. Engelfriet and S. Maneth. The equivalence problem for deterministic MSO tree transducers is decidable. *Inf. Process. Lett.*, 100(5):206–212, 2006.

[14] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31(1):71–146, 1985.

[15] Joost Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical systems theory*, 10(1):289–303, Dec 1976. `doi:10.1007/BF01683280`.

[16] Joost Engelfriet and Heiko Vogler. High level tree transducers and iterated pushdown tree transducers. *Acta Informatica*, 26(1):131–192, Oct 1988. `doi:10.1007/BF02915449`.

[17] Z. Fulop. On attributed tree transducers. *Acta Cybernet.*, 5:261–279, 1981.

[18] J. Y. Girard. Linear logic. *TCS*, 50:1–102, 1987.

[19] Nicolas Jeannerod. *Verification of Shell Scripts Performing File Hierarchy Transformations.* Theses, Université de Paris, March 2021. URL: `https://hal.archives-ouvertes.fr/tel-03369452`.

[20] Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, and Ralf Treinen. Specification of UNIX Utilities. Technical report, ANR, October 2019. URL: `https://hal.inria.fr/hal-02321691`.

[21] M Kanazawa and R Yoshinaka. Distributional learning and context/substructure enumerability in nonlinear tree grammars. In *Formal Grammar*, pages 94–111. Springer, 2016.

[22] Makoto Kanazawa. *Almost affine lambda terms.* National Institute of Informatics, 2012.

[23] Michel Latteux. Edt0l, systèmes ultralinéaires et opérations associées. Technical report, Université de Lille, 1977.

[24] Michel Latteux. Sur les générateurs algébriques et linéaires. *Acta Informatica*, 13(4):347–363, May 1980. `doi:10.1007/BF00288769`.

[25] 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'06)*, pages 199–208. IEEE, 2006.

[26] S. Salvati. Recognizability in the simply typed lambda-calculus. In *WoLLIC*, volume 5514 of *LNCS*, pages 48–60, 2009.

[27] S. Salvati and I. Walukiewicz. Using models to model-check recursive schemes. *Logical Methods In Computer Science*, 2015.

[28] J. Thatcher and J. Wright. Generalized Finite Automata Theory With an Application to a Decision Problem of Second-Order Logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.