A mechanized theory of regular trees in dependent type theory

Régis Spadotti

▶ To cite this version:

Régis Spadotti. A mechanized theory of regular trees in dependent type theory. Mathematical Software [cs.MS]. Université Paul Sabatier - Toulouse III, 2016. English. NNT : 2016TOU30178 . tel-01589656

HAL Id: tel-01589656

https://theses.hal.science/tel-01589656v1

Submitted on 18 Sep 2017

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

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

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *19 mai 2016* **par :**

Régis Spadotti

Une théorie mécanisée des arbres réguliers en théorie des types dépendants

JURY

Tarmo Uustalu	Professeur des universités	Institute of Cybernetics, Tallinn
Alan Schmitt	Chargé de recherche, HDR	INRIA, Rennes
David Chemouil	Chercheur	ONERA, Toulouse
Nicolas Tabareau	Chargé de recherche	INRIA, Nantes
Jean-Paul Bodeveix	Professeur des universités	IRIT/UPS, Toulouse
Mamoun Filali	Chargé de recherche	IRIT/CNRS, Toulouse

École doctorale et spécialité :
MITT : Domaine STIC : Sûreté du logiciel et calcul de haute performance
Unité de Recherche :
Institut de Recherche en Informatique de Toulouse
Directeurs de Thèse :
Jean-Paul Bodeveix et *Mamoun Filali*
Rapporteurs :
Tarmo Uustalu et *Alan Schmitt*

A MECHANIZED THEORY OF REGULAR TREES IN DEPENDENT TYPE THEORY

RÉGIS SPADOTTI

Abstract

Proof assistants are tools developed by computer scientists in order to ease formal reasoning. In this sense, they provide a framework to express statements and properties. Then, by using the proof rules of the underlying logic, theorems are proved and mechanically checked by the machine.

Dependent type theory is a formalism which can serve as an alternative to set theory as a foundation for all mathematics. Type theory provides a unified framework for defining programs along with their data structures, and for expressing their properties. Concretely, this means that the same language is used to define programs, state their specifications, and express the proof of their soundness. Moreover, when the underlying logic is constructive, it is possible to extract a program from the proof term of its specification. Some type theories offer powerful reasoning principles such as induction for reasoning about finite objects, or coinduction for reasoning about infinite objects.

Graphs are a ubiquitous data structure in computer science. They are used for giving semantics to various logic, for modeling computations, or for expressing relations between objects. The problem of representing graphs in dependent type theory can be quite challenging. Indeed, the main obstacle is that, in full generality, graphs can be circular structures. However, induction—the main reasoning principle in dependent type theory—fails to capture naturally such circularity. Indeed, inductive types are based around the notion of well-foundedness. Nevertheless, it is well-known that coalgebraic approaches are better-suited in order to reason about non well-founded structures, *i.e.*, structures embedding some forms of circularity. As such, coinductive types may be used to define and reason about these infinite objects. The key idea is that circular structures can be thought of as infinite trees when cycles are unfolded infinitely.

We are interested in the problem of mechanizing a theory of regular trees in dependent type theory. Informally, regular trees are characterized as the subset of infinite trees having the property that the set of their distinct subtrees is finite. As such, regular trees can be thought of as finite cyclic structures.

In this thesis, we propose two formalizations of regular trees. The first one, based on coinduction, defines regular trees as a restriction of a coinductive type. The second one follows a syntactic approach, in the sense that regular trees are characterized as inductively defined cyclic terms, *i.e.*, terms with back-pointers. We prove that these two representations are isomorphic.

Then, we study the problem of defining transformations on trees that preserve the regularity property. To this end, we leverage the formalism of tree transducers as a tool to obtain a synctactic characterization of a subset of corecursive function definitions. Tree transducers are then interpreted back as tree morphisms preserving this regularity property.

Finally, we study various decidability results through a mechanization of a coalgebraic μ-calculus interpreted on regular trees. In particular, we prove that the bisimilarity relation on regular trees is decidable through a reduction to a model-checking problem.

Résumé

Les assistants de preuve sont des outils développés par les informaticiens dans le but de faciliter le raisonnement formel. En ce sens, ces outils fournissent un cadre formel permettant l'expression d'énoncés ainsi que de leurs propriétés. Ensuite, en utilisant les règles de preuve de la logique sous-jacente, les preuves des théorèmes sont données à l'ordinateur pour vérification.

La théorie des types dépendants est un formalisme pouvant servir comme fondation alternative à la théorie des ensembles pour exprimer les mathématiques. Plus généralement, de telles théories des types offrent un cadre unifié permettant la définition de structures de données, des programmes manipulant ces structures et l'expression de leurs propriétés. Concrètement, cela signifie que le même langage est utilisé à la fois pour définir les programmes, pour énoncer leurs spécifications, et enfin exprimer la preuve de leur correction. La notion de constructivité peut être exploitée afin d'extraire un programme à partir du terme de preuve de sa spécification. Certaines théories intègrent de puissants principes de raisonnement tels que l'induction pour raisonner sur des objets finis, ou la co-induction pour raisonner sur les objets infinis.

Les graphes sont une structure de données omniprésente en informatique. Ils sont utilisés pour donner une sémantique aux logiques, pour modéliser des calculs, ou encore pour décrire des relations entre objets. Le problème consistant à représenter la structure de graphes en théorie des types dépendants peut s'avérer difficile. En effet, le principal obstacle est que, dans leur forme la plus générale, les structures de graphe peuvent être cycliques. Cependant, le principe d'induction (principe de base de raisonnement) échoue à capturer une telle circularité. En effet, les types inductifs sont basés sur le principe de bonne-fondation. Néanmoins, il est bien connu que les approches basées sur les coalgèbres sont plus adaptées pour raisonner sur des structures non bien-fondées, c'est-à-dire, des structures infinies ou présentant une forme de circularité. Dans ce contexte, les types co-inductifs offrent un cadre naturel pour définir et raisonner sur ce type d'objets. L'idée principale réside dans le fait que les structures cycliques peuvent être vues comme des arbres infinis obtenus par dépliage infini des cycles.

Nous nous intéressons au problème consistant à mécaniser une théorie des arbres réguliers en théorie des types dépendants. Informellement, les arbres réguliers sont caractérisés comme le sous-ensemble des arbres infinis tels que l'ensemble de leurs sous-arbres distincts est fini. Ainsi, les arbres réguliers peuvent être vus comme des structures cycliques finies.

Dans le cadre de cette thèse, nous proposons deux formalisations des arbres réguliers. La première, basée sur la co-induction, définit les arbres réguliers comme la restriction d'un type co-inductif. La seconde formalisation suit une approche syntaxique, dans le sens où les arbres réguliers sont caractérisés par des termes cycliques définis inductivement, c'est-à-dire, par des termes intégrant des pointeurs de retour. Nous prouvons que ces deux représentations sont isomorphes.

Dans un second temps, nous étudions le problème consistant à définir des transformations d'arbres qui préservent la propriété de régularité. Dans ce but, nous exploitons le formalisme des transducteurs d'arbres comme un outil visant à obtenir une caractérisation syntaxique d'un sous-ensemble de fonctions co-récursives. Les transducteurs d'arbres sont ensuite interprétés comme des morphismes d'arbres préservant la régularité.

Enfin, nous étudions des résultats de décidabilité via une mécanisation du μ-calcul coalgébrique interprété sur les arbres réguliers. En particulier, nous prouvons la décidabilité de la relation de bissimilarité entre arbres réguliers via d'une réduction vers un problème de vérification de modèles.

Acknowledgements

First and foremost, I wish to express my gratitude to my supervisors Jean-Paul Bodeveix and Mamoun Filali for all the support and guidance. In particular, I appreciated the constant feedback about my work and the freedom to work on a wide variety of topics.

I would like to thank the rest of my thesis committee. First of all, Alan Schmitt and Tarmo Uustalu for reviewing the manuscript, providing me with insightful comments and remarks, and for suggesting improvements and clarifications on the manuscript. Furthermore, I would like to thank Nicolas Tabareau and David Chemouil for their feedback and ideas upon my work.

I am indebted to Benedikt Ahrens for introducing me to homotopy type theory. I had the pleasure to collaborate with him on various insightful topics that helped me deepen my understanding of type theory. I am also grateful to Herman Geuvers for making it possible to visit his group in Nijmegen.

During my PhD, I had the opportunity to teach. I would like to thank all the teachers of IUT Paul Sabatier I had the pleasure to work with: Jean-Paul Carrara, Max Chevalier, Christine Julien, Hervé Leblanc, Patrick Magnaud, Christian Percebois, Florence Sedes, Fabienne Viallet.

I had the privilege to be part of the ACADIE team at IRIT. I would like to thank the members of the group (current or past): Andres, Arnaud, Aurélie, Bertrand, Benedikt, Célia, Érik, Florent, Jan-Georg, Jean-Baptiste, Jean-Paul, Mamoun, Manuel, Marc, Martin, Mathias, Meriem, Philippe, Ralph, Sergei, Wilmer, Xavier, Yamine, Zhibin. Last but not least, a special thanks to Guillaume Verdier for his invaluable daily support.

Finally, I would like to thank my parents and friends for all the support they gave me throughout my studies.

CONTENTS

Introduction 1

Chapter 1 — Preliminaries 5

1.1 Dependent Type Theory . . . 5
 1.1.1 Base Types and Type Operators . . . 5
 1.1.2 Inductive and Coinductives Types . . . 8
 1.1.3 Equality, Identity Type, and Setoids . . . 15
1.2 Finite Types . . . 17
 1.2.1 Canonical Finite Types . . . 18
 1.2.2 Finite Setoids . . . 20
 1.2.3 Finitely Indexed Setoids . . . 22
 1.2.4 Weakly Finite Indexed Setoids . . . 23
 1.2.5 Exploration Function . . . 26
 1.2.6 Streamless Setoid . . . 27
1.3 Signatures and Indexed Signatures . . . 28
 1.3.1 Signatures . . . 28
 1.3.2 Indexed Signatures . . . 30
 1.3.3 Term Algebra . . . 31
1.4 Coalgebras and Coinductive Types . . . 33
 1.4.1 Semantics of Coinductive Types . . . 33
 1.4.2 Equality and Coinductive Types . . . 35
1.5 Conclusion . . . 37

Chapter 2 — Regular Trees 39

2.1 A Coinductive Characterization . . . 39
 2.1.1 Infinite Trees . . . 40
 2.1.2 Unranked Coalgebras . . . 42
 2.1.3 Paths and Successors in Unranked Coalgebras . . . 44
 2.1.4 Finite Type of Successors . . . 54
 2.1.5 Regular Trees . . . 58
2.2 A Syntax for Regular Trees . . . 62
 2.2.1 Cyclic Terms . . . 62
 2.2.2 Syntactic Properties on Cyclic Terms . . . 66
 2.2.3 Semantics of Cyclic Terms . . . 67
 2.2.4 Subterms of Cyclic Terms . . . 72
 2.2.5 Soundness and Completeness of Syntactic Representation . . . 76
2.3 Conclusion . . . 90

Chapter 3 — Tree Transducers 91

3.1 Motivating Example . . . 91
3.2 Top-down Tree Transducers . . . 92
  3.2.1 Definitions and Semantics . . . 92
  3.2.2 Tree Transducers in Dependent Type Theory . . . 94
  3.2.3 Towards Tree Transducers on Infinite Trees . . . 96
3.3 Guarded Top-Down Tree Transducers . . . 100
3.4 Guarded Tree Transducers with ε-rules . . . 101
3.5 Tree Transducers with Finite Look-ahead . . . 103
3.6 Binary Top-Down Tree Transducers . . . 108
3.7 Conclusion . . . 110

Chapter 4 — Applications 111

4.1 Sequentialization of Processes . . . 111
4.2 Model-Checking on Regular Trees . . . 113
  4.2.1 Syntax . . . 113
  4.2.2 Denotational Semantics . . . 115
  4.2.3 Decidability of Bisimilarity . . . 120
4.3 Conclusion . . . 123

Conclusion 125

Bibliography 129

APPENDICES

Appendix A — Definitions and Notations 137

A.1 Identity Type . . . 138
A.2 Signature . . . 139
A.3 Free Monad . . . 140
A.4 Maybe . . . 141
A.5 Canonical Finite Set . . . 143
A.6 Vector . . . 144
A.7 List . . . 146

Appendix B — Cyclic Terms Properties 149

Appendix C — Transitive Closure Computation 159

Appendix D — Extended abstract 167

INTRODUCTION

Proof assistants are tools developed by computer scientists in order to ease formal reasoning. In this sense, they provide a framework to express statements and properties. Then, by using the proof rules of the underlying logic, theorems are proved and mechanically checked by the machine. Dependent type theory is a formalism which can serve as an alternative to set theory as a foundation for all mathematics. Type theory provides a unified framework for defining programs along with their data structures, and for expressing their properties. Concretely, this means that the same language is used to define programs, state their specifications, and express the proof of their soundness. Moreover, when the underlying logic is constructive, it is possible to extract a program from the proof term of its specification. Some type theories offer powerful reasoning principles such as induction for reasoning about finite objects, or coinduction for reasoning about infinite objects.

Graphs are a ubiquitous data structure in computer science. They are used for giving semantics to various logic, for modeling computations, or for expressing relations between objects. The problem of representing graphs in dependent type theory can be quite challenging. Indeed, the main obstacle is that, in full generality, graphs can be circular structures. Consequently, induction—the main reasoning principle in dependent type theory—fails to capture naturally such circularity. Indeed, inductive types are based around the notion of well-foundedness. Nevertheless, it is well-known that coalgebraic approaches are better-suited in order to reason about non well-founded structures [Rut00], *i.e.*, structures embedding some forms of circularity. As such, coinductive types may be used to define and reason about these infinite objects. The key idea is that circular structures can be thought of as infinite trees when cycles are unfolded infinitely. Such an approach was followed, for instance, in [Pic12] which dealt with the problem of representing graphs in the Coq proof assistant.

However, when one wants to reason about finite circular structures, things are not as well-behaved. Indeed, the problem is that when a finite circular structure is represented through a coinductive type, we lose some information, namely the finiteness property. Consequently, coinductive types fail to capture the distinction between finite and infinite circular structures. Though, it is not entirely obvious how to characterize the subset of coinductive terms embedding such finiteness property and how to preserve it. One of the problems is that, in a constructive setting, various non-equivalent characterization of finiteness exist [SC10, FU15]. On the one hand, some characterizations require the underlying equality to be decidable, *i.e.*, such that there exists an effective procedure checking equality. On the other hand, it could be interesting to consider weaker characterizations of finiteness in order to capture an increasingly larger subset of finite circular structures but at the same time, still strong enough to preserve computability. Regular trees [Cou83] are a fundamental example of infinite trees embedding a finiteness property. Intuitively, regular trees are the subset of infinite trees having the property that the set of their distinct subtrees is finite. For example, they arise as solutions of finite system of equations.

Finite circular structures also arise in inductively-generated structures, *e.g.*, in the syntax of languages containing an iteration operator. For instance, they are found in functional languages containing let rec style definitions to express general recursion, in imperative programming with while loops, or in the form of μ operator in process algebras (resp. in type theory) denoting recursive processes (resp. recursive types). Semantically, such terms are generally *identified with their unfolding*. Consequently, reasoning over these terms has to take into account this identification. Furthermore, operations defined on these terms ought to be semantically invariant under unfolding but it is not entirely clear how to extend inductive reasoning to such an identification principle. However, when terms are viewed as infinite terms, the unfolding operator becomes *transparent*. In this context, it could be interesting to reuse the framework of coinductive types, which naturally identifies terms with their unfolding. For instance, this framework could be used to define operations on terms by coiteration or to reason about such operations by coinduction.

The aim in this thesis is to devise a mechanized theory of regular trees in dependent type theory. In particular, we are interested in giving a formal definition of regular trees taking into account the specificity of the dependent type theory we are working with, while remaining fully constructive and axiom-free.

OUTLINE AND SUMMARY OF CONTRIBUTIONS

In this section, we give an overview of each chapter contained this thesis along with a summary of contributions.

Chapter 1 [Preliminaries]. This chapter contains a short introduction to the dependent type theory, namely the calculus of (co)inductive constructions, used as the mathematical foundation of all results found in the subsequent chapters of this thesis. In particular, we introduce inductive and coinductive types along with their reasoning principles. Furthermore, coinductive types are presented through a categorical semantics, *i.e.*, as (weakly) final coalgebras. Next, we introduce various—non equivalent—definitions of finite setoids. These definitions are straightforward generalizations of finite type definitions found in the literature. This generalization to setoids is motivated by the fact that the bisimilarity relation on coinductive types is not a congruence. As a result, we cannot substitute bisimilar terms in arbitrary contexts. Moreover, the underlying theory lacks proper quotient types.

The main contribution of this chapter is a mechanized library of finite setoids. In particular, we consider weak forms of finite setoids in the sense that the underlying setoid equivalences are not assumed to be decidable.

The presentation of coinductive types is partially based upon the following previous work:

[AS14] Benedikt Ahrens and Régis Spadotti. Terminal semantics for codata types in intensional Martin-Löf type theory. In *TYPES*, volume 39 of *LIPIcs*, pages 1–26. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2014

[ACS15] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In *TLCA*, volume 38 of *LIPIcs*, pages 17–30. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2015

Chapter 2 [Regular Trees]. This chapter is divided in two main parts. First, we formalize the type of regular trees as a restriction of coinductive types over an arbitrary signature. This formalization of regular trees is based upon the hierarchy of finite setoids introduced in Chapter 2. In particular, we do not assume the type of function symbols (as given by the signature) to have a decidable equality. Yet, it is constructive enough to prove that regular trees are closed under the subtree order. Next, we define a syntax for regular trees by means of cyclic terms. This syntax is then shown to be sound and complete with respect to the coinductive characterization of regular trees. In addition, we give a sound and complete axiomatization of cyclic term equivalence. Finally, we prove that the function mapping cyclic terms to regular trees is a (setoid) isomorphism.

The main contributions of this chapter are summarized in the following table:

Type of regular trees $\textsc{Reg}_S$	Definition 2.14
Closure under the subtree order	Theorem 2.5
Type of cyclic terms $\mathbb{C}_S$	Definition 2.15
Soundness of syntactic representation	Theorem 2.7
Completeness of syntactic representation	Theorem 2.9
$[\![_]\!] : \mathbb{C}_S \to \textsc{Reg}_S$ is a setoid isomorphism	Theorem 2.12

Chapter 3 [Tree Transducers]. In this chapter, we study the problem of defining regularity preserving trees morphisms by means of (co)recursive schemes. To this end, we use the formalism of top-down tree transducers as a tool to model the abstract syntax of corecursive function definitions as found in proof assistants such as Coq or Agda. In particular, we study various sorts of top-down tree transducers of increasing (syntactic) expressiveness. Finally, this abstract syntax is reified as regularity preserving tree morphisms.

The main contributions of this chapter are summarized in the table below:

Guarded top-down tree transducers	Definition 3.9
Regularity-preserving induced tree morphism	Theorem 3.2
Top-down tree transducers with ϵ-rules	Definition 3.11
Regularity-preserving induced tree morphism	Theorem 3.3
Top-down tree transducers with finite look-ahead	Definition 3.13
Regularity-preserving induced tree morphism	Theorem 3.4
Binary top-down tree transducers	Definition 3.16
Regularity-preserving induced tree morphism	Theorem 3.5

Chapter 4 [Applications]. In this chapter, we present examples using the mechanized theory of regular trees developed in the previous chapters. Through the computation of a parallel composition operator on a CSP-like process algebra, we highlight how a termination problem can be transformed to a productivity problem. The computation of the parallel composition operator is realized by means of a top-down tree transducer. Next, we consider decidability problems on the type of regular trees. Indeed, regular trees are characterized through a finiteness property. Consequently, various properties that are generally undecidable on infinite trees become decidable on the subset of regular trees. To this end, we give an interpretation of the propositional modal μ-calculus over the type regular trees and prove the decidability of satisfiability. Then,

we show how some decidability problems can be translated to a model-checking problem. In particular, we show that bisimilarity is decidable on regular trees.

The main contributions of this chapter are detailed in the following table:

Parallel composition operator defined as a tree transducer	Section 4.1
Interpretation of the μ-calculus over regular trees	Section 4.2
Decidability of μ-calculus formula satisfiability (model-checking)	Theorem 4.1
Decidability of bisimilarity	Theorem 4.2

A subset of the results presented from Chapter 2 to 4 has been published in

[Spa15] Régis Spadotti. A mechanized theory of regular trees in dependent type theory. In *ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 405–420. Springer, 2015

CHAPTER ONE

PRELIMINARIES

In this chapter, we introduce the underlying theory used throughout the remaining of this thesis. First, we give a short introduction to the dependent type theory that serves as the theoretical foundation of all definitions and results presented in the subsequent chapters. Then, we study various notions of finite types and setoids in a constructive setting. Finite types are the main ingredient to characterize the regularity property on infinite trees. Moreover, we introduce signatures as a way to abstract away structural properties of trees. Finally, we present a semantic study of coinductive types through coalgebras in category theory.

1.1. DEPENDENT TYPE THEORY

This section introduces the underlying theory used in the formalization of the various results presented within this thesis. *Set theory* and all its derivatives (ZF, ZFC, ...) is a well-known and widely used foundation for mathematics. Type theory is another formal system that may be used for the same purpose. One difference between set theory and type theory is that set theory is built on top of a logic while type theory is a logic in itself. Some of these logics are intuitionistic and as such, support *constructive reasoning*. Another interesting property of type theory is that it inherits the *computational model* of the λ-calculus. This last aspect is particularly important in the context of computer science.

Throughout the years, various sorts of type theories have emerged with increasing complexity and expressiveness. Examples include the Simply-Typed Lambda Calculus [Chu40], System F [Gir72, Rey74], Intuitionistic Martin-Löf Type Theory (IMLTT) [ML75], Calculus of Constructions (CoC) [CH88], etc. Among these type systems, the last ones, namely IMLTT and CoC, are of particular interest to us because they allow types to *depend* upon values. Such type theories are also known as *dependent type theories*. This somewhat subtle change yields a powerful and expressive logic. In particular, through the Curry-Howard isomorphism, programs/proofs (terms) and specifications/statements (types) may be expressed within a *unique formalism*.

In the remaining of this section, we give an overview of the syntax of the dependent type theory we work with and introduce various notations. Then, we introduce the base type primitives along with their powerful reasoning principles. Finally, we discuss the problem of dealing with equality between terms.

1.1.1. Base Types and Type Operators

We assume the reader to be somewhat familiar with dependent type theory as implemented in proof assistants such as AGDA [Nor07] (based on IMLTT) or COQ [Coq16] (based on CoC). Here, our goal is not to give an in-depth treatment of the underlying type theory. Instead, we review

Name	Type	Constructor(s)
Arrow	$A : \textsc{Type}, B : \textsc{Type} \vdash A \to B : \textsc{Type}$	$\lambda(a : A).\ b$
Product	$A : \textsc{Type}, B : \textsc{Type} \vdash A \times B : \textsc{Type}$	(a, b)
Disjoint sum	$A : \textsc{Type}, B : \textsc{Type} \vdash A \uplus B : \textsc{Type}$	$\mathsf{inl}\, a, \mathsf{inr}\, b$
Empty	$\mathbb{0} : \textsc{Type}$	
Unit	$\mathbb{1} : \textsc{Type}$	tt
And	$A : \textsc{Prop}, B : \textsc{Prop} \vdash A \wedge B : \textsc{Prop}$	(a, b)
Or	$A : \textsc{Prop}, B : \textsc{Prop} \vdash A \vee B : \textsc{Prop}$	$\mathsf{left}\, a, \mathsf{right}\, b$
False	$\bot : \textsc{Prop}$	
True	$\top : \textsc{Prop}$	$\star$

Table 1.1. Base types along with their respective constructors.

all notations that will be used since they are slightly different in the mechanization. Various introductions to the theories and the proof assistants include [Chl13, PdAC+16, BC04, BC00].

The type theory used in this thesis is the Calculus of Coinductive Constructions [CH88] as implemented within the COQ proof assistant with some slight syntactic modifications aiming to improve readability. We begin by reviewing the base types along with their respective inhabitants found in this dependent type theory.

First, we assume the existence of an infinite cumulative hierarchy of *predicative universes* $(\textsc{Type}_i)_{i \in \mathbb{N}}$ such that $\textsc{Type}_i$ is a subtype[1] of $\textsc{Type}_{i+1}$, for all i. At the bottom of the hierarchy, there is an *impredicative universe of logical values*, denoted PROP, such that PROP is a subtype of $\textsc{Type}_0$. Informally, the impredicativity of PROP means that a statement quantifying over all inhabitants of PROP can be instantiated with itself whereas predicativity restricts the type of statement to live in a strictly larger universe than the universe being quantified upon. For example, consider the id function defined in both PROP and in TYPE:

$$\mathsf{id}_{\textsc{Type}} : \forall (A : \textsc{Type}_i).\ A \to A \qquad \mathsf{id}_{\textsc{Prop}} : \forall (A : \textsc{Prop}).\ A \to A$$
$$\mathsf{id}_{\textsc{Type}}\ x \equiv x. \qquad \mathsf{id}_{\textsc{Prop}}\ x \equiv x.$$

Since PROP is impredicative the term $\mathsf{id}_{\textsc{Prop}}(\mathsf{id}_{\textsc{Prop}})$ type-checks whereas $\mathsf{id}_{\textsc{Type}}(\mathsf{id}_{\textsc{Type}})$ leads to a universe inconsistency which is due to the fact that the type of $\mathsf{id}_{\textsc{Type}}$ is $\textsc{Type}_{i+1}$. From now on, we omit the universe level and simply write TYPE instead of $\textsc{Type}_i$. Generally, the universe levels can be inferred as it is the case in COQ. Inference is also possible when quantification upon universes is allowed [ST14], a feature known as universe polymorphism.

The universe PROP is used in the Calculus of Constructions to isolate terms that are considered as logical statements—without computational content—from terms considered to be programs. However, a term living in the sort PROP *should not* be confused with the type of mere propositions defined in Section 1.1.3. Nonetheless, it is common to refer to inhabitants of PROP as propositions.

In order to support the separation between program and logical values, typing rules prevent the elimination (case-analysis) of values in PROP to produce values in TYPE. This ensures that a computationally relevant term cannot be obtained from a computationally irrelevant one. This clear separation allows an extraction procedure [Pau89] to optimize away logical statements.

We assume the existence of some base types and operators on types which are listed in Table 1.1. All types except the arrow type may be introduced as inductive type definitions (see

[1] When S is a subtype of T, a term of type S can be used in a context where the type T is expected.

Section 1.1.2). The product and disjoint sum types are redefined in the PROP universe and are called *and* and *or* respectively. The negation of a proposition is defined as follows:

$$\neg_ : \text{PROP} \to \text{PROP}$$
$$\neg P \equiv P \to \bot.$$

Now, we introduce the two important type operators that allow dependencies between values and types to be expressed. The first one, called *dependent product* (also known as pi-type), is a generalization of the arrow operator:

$$\boxed{\text{Type}} \quad \frac{A : \text{TYPE} \qquad B : A \to \text{TYPE}}{\prod_{a:A} B(a) : \text{TYPE}}. \qquad \boxed{\text{Constructor}} \quad \lambda(a : A).\ b : \prod_{a:A} B(a).$$

When the right-hand side does not depend on the left-hand side, we drop the $\prod$ symbol and simply write $A \to B$ instead of $\prod_{a:A} B$. To save space, it is often convenient to drop the $\prod$ symbol altogether and simply write $(a : A) \to B(a)$ instead of $\prod_{a:A} B(a)$. In addition, it is also common to group together formal parameters of the same type. Thus, we may abbreviate $(a : A)(a' : A)$ into $(a\ a' : A)$.

Successive use of dependent product can introduce some sort of redundancy. For instance, consider the following type expressing the functoriality of the type LIST:

$$\text{map} : (A\ B : \text{TYPE}) \to (A \to B) \to \text{LIST}\ A \to \text{LIST}\ B.$$

Here A and B are repeated twice. This means that whenever the function map is used, both types have to be explicitly supplied even though both parameters could be inferred from the type of the function. One solution to avoid this redundancy is to mark that A and B as *implicit arguments*. They are usually written with curly braces as follows:

$$\text{map} : \{A\ B : \text{TYPE}\} \to (A \to B) \to \text{LIST}\ A \to \text{LIST}\ B.$$

Consequently, instead of writing map $A\ B\ f\ l$, we can write map $f\ l$, leaving A and B to be inferred from the type of f or l. Implicit arguments are supported in both AGDA and COQ and their usage is quite common in languages based on dependent types. We follow the same convention in the remaining of this thesis.

Another convenient feature used extensively is known as *implicit generalization*. This allows free variables found in the type to be implicitly quantified by pi-types. As a result, with implicit generalization, the type of the map function can be abbreviated as:

$$\text{map} : (A \to B) \to \text{LIST}\ A \to \text{LIST}\ B$$

where A and B occur freely in the type of map.

The second operator specific to dependent type theory is called *dependent sum* (also known as sigma-type or dependent pair) and is a generalization of the product operator:

$$\boxed{\text{Type}} \quad \frac{A : \text{TYPE} \qquad B : A \to \text{TYPE}}{\sum_{a:A} B(a) : \text{TYPE}}. \qquad \boxed{\text{Constructor}} \quad (a, b) : \sum_{a:A} B(a).$$

Both of theses operators, namely dependent product and dependent sum, have a counterpart in the universe of propositions, called *forall* and *exists* respectively:

$$\boxed{\text{Type}} \quad \frac{A : \text{TYPE} \qquad B : A \to \text{PROP}}{\forall (a : A).\ B(a) : \text{PROP}}. \qquad \boxed{\text{Constructor}} \quad \lambda(a : A).\ b : \forall (a : A).\ B(a).$$

$$\boxed{\text{Type}} \quad \frac{A : \text{TYPE} \qquad B : A \to \text{PROP}}{\exists (a : A).\ B(a) : \text{PROP}}. \qquad \boxed{\text{Constructor}} \quad (a, b) : \exists (a : A).\ B(a).$$

Note that in both cases, the constructors are overloaded.

Finally, we may write the subset type as $\{ a : A \mid P(a) \} : \text{TYPE}$ where $P : A \to \text{PROP}$ instead of $\sum_{a:A} P(a)$. We use this type in order to emphasize that $P(a)$ is a logical proposition. This type can be thought of as a "constructive exists" in the sense that the witness a is computationally relevant whereas the proof $P(a)$ is not.

1.1.2. Inductive and Coinductives Types

Inductive Types

Inductive types are a powerful construction that can be used to extend the theory with new types. For instance, both the product and the disjoint sum can be defined as an inductive type as follows:

```
inductive _×_ (A B : Type) : Type
  | (_,_) : A → B → A × B.

inductive _⊎_ (A B : Type) : Type
  | inl : A → A ⊎ B
  | inr : B → A ⊎ B.
```

Here the type is introduced with the keyword **inductive** followed by an identifier. We allow types to be defined with mixfix notations [DN08], following the conventions of AGDA. The underscore (_) denotes a placeholder for an argument to be picked (in-order) from the parameters immediately following the underscore. Arguments of inductive type definitions may appear before or after the colon (:) operator. In the first case, it denotes polymorphic arguments whereas in the second case, it represents indices of inductive families. In the latter case, the values of indices are given by the type of constructor definitions.

Every inductive type definition yields an *induction principle*[1] (dependent eliminator) where the returned type may depend upon the input term. For example, the induction principle for the product type is given by:

$$\times\text{-rect}\ \frac{P : A \times B \to \text{TYPE} \qquad \forall (a : A).\ \forall (b : B).\ P(a, b)}{\forall (p : A \times B).\ P(p)}$$

while its definition is given by:

```
×-rect : (P : A × B → Type) → (∀(a : A). ∀(b : B). P(a, b)) → ∀p. P(p)
×-rect P f (a, b) ≡ f a b.
```

or with case-analysis performed in the right-hand side:

```
×-rect : (P : A × B → Type) → (∀(a : A). ∀(b : B). P(a, b)) → ∀p. P(p)
×-rect P f p ≡ match p with
                 | (a, b) ⇒ f a b
               end
```

The definition of ×-rect illustrates the fact that values of inductive types can be eliminated through a construction called *pattern-matching*. Pattern-matching may appear in the left-hand side of equations (as in the example above) or through the construction **match** ... **with** ... **end** which allows case-analysis to be perform on the right-hand side of equations.

[1] The induction principle is automatically generated in COQ but not in AGDA.

Inductive type definitions may also be recursive. For instance, consider the type of natural numbers defined as follows:

$$\textbf{inductive}\ \mathbb{N} : \textsc{Type}$$
$$\mid \mathsf{zero} : \mathbb{N}$$
$$\mid \mathsf{suc} : \mathbb{N} \to \mathbb{N}.$$

The inductive definition of $\mathbb{N}$ captures the well-known induction principle over natural numbers:

$$\mathbb{N}\text{-rect}\ \frac{P : \mathbb{N} \to \textsc{Type} \qquad P(\mathsf{zero}) \qquad \forall (n : \mathbb{N}).\ P(n) \to P(\mathsf{suc}\, n)}{\forall (n : \mathbb{N}).\ P(n)}.$$

The function $\mathbb{N}$-rect is defined recursively[1] as follows:

$$\mathbb{N}\text{-rect} : (P : \mathbb{N} \to \textsc{Type}) \to P(\mathsf{zero}) \to (\forall (n : \mathbb{N}).\ P(n) \to P(\mathsf{suc}\, n)) \to \forall (n : \mathbb{N}).\ P(n)$$
$$\mathbb{N}\text{-rect}\ P\ f_z\ f_s\ \mathsf{zero} \equiv f_z$$
$$\mathbb{N}\text{-rect}\ P\ f_z\ f_s\ (\mathsf{suc}\, n) \equiv f_s\ n\ (\mathbb{N}\text{-rect}\ P\ f_z\ f_s\ n).$$

Such a recursive definition is well-defined because of a syntactic criterion ensuring termination, namely that n is a subterm of $\mathsf{suc}\, n$. We say that the function $\mathbb{N}$-rect is defined by *recursion* because the definition of $\mathbb{N}$-rect consists of a series of equations (or rewrite rules) where the function $\mathbb{N}$-rect is allowed to appear on the right-hand side—provided that the function is well-defined. When a function over natural numbers is defined with the function $\mathbb{N}$-rect, we say that it is defined by *induction*. For example, we can define the function that computes the sum of two natural numbers by induction on *the first argument*:

$$_+_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$_+_ \equiv \mathbb{N}\text{-rect}\ (\lambda _.\ \mathbb{N} \to \mathbb{N})\ \mathsf{id}_{\mathbb{N}}\ (\lambda _.\ \lambda p_m.\ \mathsf{suc} \circ p_m).$$

Inductive families

Inductive types may also be indexed by other types. In this case, they are called *inductive families*. For example, the type of fixed-length lists, also known as *vectors*, can be defined as an inductive family indexed by a natural number:

$$\textbf{inductive}\ \textsc{Vec}\ (A : \textsc{Type}) : \mathbb{N} \to \textsc{Type}$$
$$\mid [\,] : \textsc{Vec}\ A\ \mathsf{zero}$$
$$\mid _{::}_ : \{n : \mathbb{N}\} \to A \to \textsc{Vec}\ A\ n \to \textsc{Vec}\ A\ (\mathsf{suc}\, n).$$

Another example of a $\mathbb{N}$-indexed type is the type representing canonical finite sets (see also Section 1.2 for a more detailed presentation):

$$\textbf{inductive}\ \textsc{Fin} : \mathbb{N} \to \textsc{Type}$$
$$\mid \mathsf{zero} : \{n : \mathbb{N}\} \to \textsc{Fin}(\mathsf{suc}\, n)$$
$$\mid \mathsf{suc} : \{n : \mathbb{N}\} \to \textsc{Fin}\, n \to \textsc{Fin}(\mathsf{suc}\, n).$$

Inductive families are also particularly useful in order to define logical predicates or relations. For instance, the binary relation "lower-or-equal" on natural numbers can be introduced as follows:

$$\textbf{inductive}\ _\leq_ : \mathbb{N} \to \mathbb{N} \to \textsc{Prop}$$
$$\mid \leq_{\mathsf{zero}} : \{n : \mathbb{N}\} \to \mathsf{zero} \leq n$$
$$\mid \leq_{\mathsf{suc}} : \{m\ n : \mathbb{N}\} \to m \leq n \to \mathsf{suc}\, m \leq \mathsf{suc}\, n.$$

[1] Recursive function is a primitive notion in Coq and Agda from which (dependent) eliminators are derived.

In the context of dependent types and more precisely of inductive families, pattern-matching can be complicated by the fact that case-analysis on a term may have consequences over the types depending on it. Proof assistants such as COQ deal with this issue by extending the **match** construction with annotations. In full generality, these annotations take the form of:

$$\textbf{match}\ t\ \textbf{as}\ t'\ \textbf{in}\ T\ i_1\ \dots\ i_k\ \textbf{return}\ R(t', i_1, \dots, i_k)\ \textbf{with}$$
$$\begin{array}{|l} p_1 \Rightarrow \dots \\ \dots \Rightarrow \dots \\ p_n \Rightarrow \dots \end{array}$$
$$\textbf{end}$$

Here t' binds the concrete case of t, $i_1, \dots, i_k$ bind indices in the type of t and R is the type of the expression returned on the right-hand sides for each case. The return type may depend upon t' and the indices. Fortunately, these annotations are usually inferred and thus may be left implicit. In the case when inference fails, these annotations have to be explicitly provided. However, these annotations can be quite complex and ultimately hinder readability.

Another approach consists in extending pattern-matching with a dependent pattern-matching compiler. In particular, this allows case-analysis to be performed simultaneously on multiple terms. Traditionally, dependent-pattern matching compilers often assume that Leibniz equality (defined in Section 1.1.3) satisfies the principle of *Uniqueness of Identity Proofs* (UIP) [GMM06]. However, this is not the case in COQ. Recent work on dependent pattern-matching for AGDA [CDP14] lifts this restriction and some forms of dependent pattern-matching can be performed without this axiom. The same line of work is pursued for the COQ proof assistant through a plugin called Equations [Soz10]. For the remaining of this thesis, we perform dependent pattern-matching as equations (as done in AGDA or with the Equations plugin in COQ). The main benefit is that it allows for very concise formulations of definitions as opposed to definitions based on the more primitive **match**. . .**with**. . .**end** construction, possibly extended with annotations. Though, we do not assume equality to satisfy UIP and all these equations have been elaborated into **match** expressions—possibly extended with type annotations—in the COQ development accompanying this thesis. As an illustration, consider the definition of the lookup function returning the element of a vector at a given position:

$$\begin{array}{l} _[_] : \textsc{Vec}\ A\ n \to \textsc{Fin}\, n \to A \\ (x\ ::\ _)\ \ [\ \mathsf{zero}\ ]\ \equiv\ x \\ (_\ ::\ xs)\ \ [\ \mathsf{suc}\, i\ ]\ \equiv\ xs[i]. \end{array}$$

From the pattern-matching of the index $i : \textsc{Fin}\, n$, we can deduce that n ought to be a successor natural number. Consequently, there exists $n' : \mathbb{N}$ such that

$$n = \mathsf{suc}\, n'. \qquad (*)$$

Furthermore, we can deduce that the vector cannot be empty because it ought to contain at least one element. This explains why the empty vector ($[\,]$) is not considered as valid pattern because it would contradict the equation $(*)$. Making all implicit arguments explicit would lead to the following definition, highlighting the usage of *inaccessible patterns* (enclosed in $\lfloor _ \rfloor$):

$$\begin{array}{l} _[_] : \{A : \textsc{Type}\} \to \{n : \mathbb{N}\} \to \textsc{Vec}\ A\ n \to \textsc{Fin}\, n \to A \\ _[_]\ \ A\ \ \lfloor \mathsf{suc}\, n \rfloor\ \ (x\ ::\ _)\ \ \mathsf{zero}\ \ \ \equiv\ x \\ _[_]\ \ A\ \ \lfloor \mathsf{suc}\, n \rfloor\ \ (_\ ::\ xs)\ \ (\mathsf{suc}\, i)\ \equiv\ xs[i]. \end{array}$$

Inaccessible patterns, $\lfloor \mathsf{suc}\, n \rfloor$ in this example, are a way to indicate that the terms enclosed in $\lfloor _ \rfloor$ are the only ones that would make the equations type-check. Note that, inaccessible patterns can—in most cases—be inferred from the context and thus can be left implicit.

Dependent records

Inductive type definitions can also be dependent on their polymorphic arguments. Consider the definition of the dependent sum as an inductive type:

$$\begin{array}{l} \textbf{inductive}\ \Sigma\ (A : \mathrm{TYPE})\ (B : A \to \mathrm{TYPE}) : \mathrm{TYPE} \\ \quad \mid\ (_,_) : (a : A) \to (b : B(a)) \to \Sigma\ A\ B. \end{array}$$

Given a dependent pair, we can define, by case-analysis, two (dependent) functions projecting respectively, the first and second element of a pair:

$$\begin{array}{ll} \mathsf{proj}_1 : \Sigma\ A\ B \to A & \mathsf{proj}_2 : (s : \Sigma\ A\ B) \to B(\mathsf{proj}_1\ s) \\ \mathsf{proj}_1\ \ (a, _)\ \equiv\ a. & \mathsf{proj}_2\ \ (_, b)\ \equiv\ b. \end{array}$$

Defining functions projecting arguments out of constructors of inductive definition is so common that there is a special syntax known as *record* definitions. For instance, we can introduce the dependent sum as a *dependent record*—declared with the **record** keyword—as follows:

$$\begin{array}{l} \textbf{record}\ \Sigma\ (A : \mathrm{TYPE})\ (B : A \to \mathrm{TYPE}) : \mathrm{TYPE} \\ \quad \textbf{constructor}\ (_,_) \\ \qquad \left[\begin{array}{l} \mathsf{proj}_1\ : A \\ \mathsf{proj}_2\ : B(\mathsf{proj}_1). \end{array} \right. \end{array}$$

The record syntax could be "desugared" into the declarations presented above. However, there exist some subtleties regarding the status of record definitions. For instance, AGDA (and more recently COQ) implements the η-rule for record. This rule states that a term t of a given record type is definitionally equal (defined in Section 1.1.3) to the term obtained by instantiating its constructor with all projection functions applied to t. Concretely, if we consider the record type Σ and a term $t : \Sigma\ A\ B$, given types A and B, the η-rule means that the following equality holds:

$$t \equiv (\mathsf{proj}_1\, t, \mathsf{proj}_2\, t).$$

Consequently, because of the η-rule, the definition of Σ as an inductive type and as a record type are not entirely equivalent. From now on, we assume dependent records to satisfy the η-rule.

Strictly positive types

Syntactic restrictions are imposed upon inductive definitions. In particular, recursive occurrences of the type being defined shall not appear *negatively*—to the left of an arrow—in the arguments of constructors. For instance, consider the following inductive definition:

$$\begin{array}{l} \textbf{inductive}\ T : \mathrm{TYPE} \\ \quad \mid\ c : (\underline{T} \to T) \to T. \end{array}$$

Here T (underlined) appears negatively in the type of the constructor argument c. If such definition would be accepted it would be possible to write a term which could loop forever:

$$\begin{array}{ll} \mathsf{app} : T \to T & \mathsf{forever} : T \\ \mathsf{app}\ \ (c(f))\ \equiv\ f(c(f)). & \mathsf{forever}\ \equiv\ \mathsf{app}(c(\mathsf{app})). \end{array}$$

We refer to [CP88] for a study of general positive occurrences in the context of the calculus of constructions. To summarize, inductive definitions, as found in the COQ proof assistant, may only characterized the so-called *strictly positive types*.

Well-founded induction

Well-founded induction is a generalization of structural induction schemes presented thus far. Intuitively, it abstracts the notion of terminating computations.

For example, we may want to define a function on natural numbers, following a general recursive scheme given as follows:

$$
\begin{aligned}
&f : \mathbb{N} \to \mathbb{N} \\
&f \;\; n \;\equiv\; \textbf{if } p(n) \\
&\qquad\quad \textbf{then } n \\
&\qquad\quad \textbf{else } f(\mathsf{suc}\, n).
\end{aligned}
$$

where p is a function from natural numbers to booleans. Such definition is not accepted by the termination checker because of the problematic call $f(\mathsf{suc}\, n)$, which is clearly not structural.

In order to show that the computation does indeed terminate, we have to find some sort of *measure*. Intuitively, this measure indicates how far the current computation step is from the end result. For instance, this measure could take the form of a function $\mu : \mathbb{N} \to \mathbb{N}$. Then, given an input $n : \mathbb{N}$, $\mu(n)$ would represent the number of steps remaining in the computation. Next, we could, for instance, annotate the function f with the values given by μ. Thus, if the measure decreases by one each time the function is called recursively, we would obtain back a structural recursive scheme. This time, however, the function f would not be defined by structural recursion on the input parameter n but on its measure $\mu(n)$.

This notion of measure can be generalized further by considering a binary relation $\succ$ over the input parameters of the function. For example, if we consider the value of the input parameters of the function f defined above, we obtain a chain:

$$n_0 \succ n_1 \succ n_2 \succ \cdots$$

where each n_i represents the value of the input parameter after i steps of computation. Thus, the computation ends if and only if the chain eventually stops. When the chain ends for all numbers n, we say that it is *well-founded*. Traditionally, the chain is presented backwards as follows:

$$\cdots \prec n_2 \prec n_1 \prec n_0.$$

An element n is said to be $\prec$*-accessible* when all its $\prec$-predecessors are themselves accessible. Formally, the accessibility relation $\textsc{Acc}$ is defined as:

$$\textsf{acc-intro}\ \frac{\forall m.\ m \prec n \to \textsc{Acc}\ (_\prec_)\ m}{\textsc{Acc}\ (_\prec_)\ n}.$$

More concretely, the accessibility relation can be introduced as an inductive type as follows:

$$
\begin{aligned}
&\textbf{inductive } \textsc{Acc}\ \{A : \textsc{Type}\}\ (_\prec_ : A \to A \to \textsc{Prop})\ (x : A) : \textsc{Prop} \\
&\quad \mid \ \textsf{acc-intro} : (\{y : A\} \to y \prec x \to \textsc{Acc}\ (_\prec_)\ y) \to \textsc{Acc}\ (_\prec_)\ x.
\end{aligned}
$$

Note that, there is no base constructor.

A binary relation $_\prec_$ is said to be *well-founded* when all elements are $\prec$-accessible:

$$
\begin{aligned}
&\textsc{Well-founded} : \{A : \textsc{Type}\} \to (\prec : A \to A \to \textsc{Prop}) \to \textsc{Prop} \\
&\textsc{Well-founded}\ \prec\ \equiv\ \forall (a : A).\ \textsc{Acc} \prec a.
\end{aligned}
$$

Finally, a well-founded binary relation induces an induction principle given by:

$$\textsf{wf-rect}\ \frac{P : A \to \textsc{Type} \qquad \forall (a : A).\ (\forall (a' : A).\ a' \prec a \to P(a')) \to P(a)}{\forall (a : A).\ P(a)},$$

defined by structural induction over the accessibility proof:

$\mathsf{wf\text{-}rect} : \{A : \textsc{Type}\} \to (P : A \to \textsc{Type}) \to (_\prec_ : A \to A \to \textsc{Prop})$
$\quad \to \textsc{Well-founded}(_\prec_) \to (\forall a.\ (\forall a'.\ a' \prec a \to P(a')) \to P(a)) \to \forall (a : A).\ P(a)$
$\mathsf{wf\text{-}rect}\ \ P\ \ (_\prec_)\ \ wf_{\prec}\ \ P_f\ \ a\ \equiv\ \mathsf{wfAcc}\ a\ (wf_{\prec}(a))$
$\quad$ **where** $\mathsf{wfAcc} : \forall (a : A).\ \textsc{Acc}\ (_\prec_)\ a \to P(a)$
$\qquad \mathsf{wfAcc}\ \ a\ \ (\mathsf{acc\text{-}intro}\ acc)\ \equiv\ P_f\ a\ (\lambda a'.\ \mathsf{wfAcc}(a) \circ acc(a')).$

Intuitively, in order to prove a property $P(a)$, we may assume that the property holds for any $\prec$-predecessor of a.

Remark. There is one subtlety in the type of the well-founded induction principle wf-rect. Here, the type of P is $A \to$ TYPE and not $A \to$ PROP. Recall that the sort PROP is used as a universe to represent logical values and that elimination in TYPE is explicitly forbidden. However, under certain circumstances, inductively-defined PROP values may have a recursor allowing elimination on any sort. The precise conditions and justification are described in COQ reference manual. Intuitively, the elimination on type is allowed whenever the inductive type has at most one constructor such that each of its arguments is in the sort PROP. Following the terminology of the COQ reference manual, when such conditions are met, we refer to the PROP universe as PROP-extended.

Empty type

The empty type $\bot$ presented in Section 1.1.1 may also be defined as an inductive type but *without any constructor*:

inductive $\bot : \textsc{Prop}.$

The induction principle[1] associated to the empty type is given by:

$$\bot\text{-rect}\ \frac{P : \textsc{Type}}{\bot \to P}.$$

Essentially, the induction principle $\bot$-rect states that whenever the context contains a term inhabiting $\bot$, it can be used to produce a term of any type. Since $\bot$ has no constructor, the only way to obtain a term $p : \bot$ is by proving the existence of a contradiction.

We introduce the () notation as an extension to the grammar of patterns. This notation deals with the elimination of $\bot$, and more generally, of any type provable to be empty by an automatic procedure. As an illustration, consider the following inductive type definitions:

inductive $T : \textsc{Type}$
$\quad |\ A : T$
$\quad |\ B : T.$

inductive $F : T \to \textsc{Type}$
$\quad |\ constr_F : \mathbb{N} \to F(A).$

Then, we define a function over the type family F as such:

$\mathsf{get\text{-}val} : (t : T) \to F(t) \to \mathbb{N}$
$\mathsf{get\text{-}val}\ \ A\ \ (constr_F(n))\ \equiv\ n$
$\mathsf{get\text{-}val}\ \ B\ \ ().$

Here, in the second equation, the type of the second argument of get-val is *empty*. Consequently, we can use the () pattern notation. Note that, the second equation does not have a right-hand side. This is due to the fact that implicitly, the () notation embeds the induction principle of

[1] The empty type $\bot$ qualifies as PROP-extended.

the empty type. Indeed, this induction principle can be used to produce a value of any type, $\mathbb{N}$ in this case. Without such syntax extension, the proof that the type $F(B)$ is empty would have to be explicitly provided and then eliminated through $\bot$-rect.

Decidability

Within the calculus of constructions, we cannot prove that the law of excluded-middle (LEM) holds for an arbitrary proposition:

$$\forall (P : \textsc{Prop}).\ P \vee \neg P. \tag{LEM}$$

Even though LEM does not hold for *all* propositions, it can still be *proven* for *some* propositions. A computational version of LEM can be introduced as an inductive type definition as follows:

```
inductive Dec (P : Prop) : Type
  | yes :  P → Dec P
  | no  : ¬P → Dec P.
```

When the type $\textsc{Dec}\,P$ is inhabited, it captures the fact that it is *decidable* whether the property P holds or not. Furthermore, because $\textsc{Dec}\,P$ lives in the sort $\textsc{Type}$, it is computationally relevant. As a result, it is admissible to define functions by case-analysis which depends upon the value of $\textsc{Dec}\,P$. For example, the statement:

> "equality is decidable on natural numbers"

can be formalized as:

$$\mathbb{N}\text{-dec} : \forall (m, n : \mathbb{N}).\ \textsc{Dec}(m = n).$$

The function $\mathbb{N}$-dec is called a *decision procedure*. Through extraction, when all logical content is removed, the type $\textsc{Dec}\,P$ is isomorphic to $\mathbb{B}$ (bool). Though, the type $\textsc{Dec}\,P$ is more informative than $\mathbb{B}$ as it comes with a justification (with a proof). Given a predicate $P : A \to \textsc{Prop}$ over some type A, we say that P is *decidable* when, for all $a : A$, the type $\textsc{Dec}(P(a))$ is inhabited. Formally,

```
Decidable : (P : A → Prop) → Type
Decidable  P  ≡  ∀(a : A). Dec(P(a)).
```

Coinductive type

Coinductive types are the dual of inductive types. Indeed, they allow terms to be defined by an *infinite* application of constructors. By opposition, terms of inductive types are constructed by a *finite* application of constructors. Coinductive type definitions are introduced by the keyword **coinductive**. The syntax is the same as inductive type definitions and the same positivity constraints apply. For example, the type of streams over a base type A is defined as follows:

```
coinductive Stream (A : Type) : Type
  | _::_ : A → Stream A → Stream A.
```

As a concrete application, the type of streams could then be used to model the infinite behavior of reactive systems. Coinductive types can also be used to represent cyclic structures such as automata or graphs [Rut00, Pic12]. Finally, coinductive types and their associated reasoning principle are discussed in greater detail in Section 1.4.

1.1.3. Equality, Identity Type, and Setoids

In order to allow equational reasoning to be used, we present two equalities as found in the type theory we consider. The first one lives purely at the meta-level while the second one is expressible as a type. Then, we discuss how equational reasoning is extended to equivalence reasoning by means of setoids.

Definitional equality

The first equality, called *definitional equality*, identifies terms which are syntactically equal up-to reductions[1] (β, η, δ, ...). When two terms t_1 and t_2 are definitionally equal, we write $t_1 \equiv t_2$. Note that definitional equality lives purely at the meta-level of the type theory. Consequently, we cannot talk about definitional equality within the type theory itself.

Leibniz equality

The second equality, called *Leibniz equality*, is a relation that is introduced as an inductive family:

$$\textbf{inductive}\ _{=}_\ (A : \textsc{Type})\ (x : A) : A \to \textsc{Prop}$$
$$\mid\ \mathsf{refl} : x = x.$$

There is exactly *one* constructor refl which stands for reflexivity.

Contrary to definitional equality, Leibniz equality is a type and is often called the *identity type* or sometimes *propositional equality*. The induction principle[2] induced from the definition of $_{=}_$ is given by:

$$=\text{-rect}\ \frac{P : \forall\{x, y : A\}.\ x = y \to \textsc{Type} \qquad \forall(x : A).\ P(\mathsf{refl}_x)}{\forall\{x, y : A\}.\ \forall(p : x = y).\ P(p)}.$$

Given a term $p : x = y$ witnessing the equality of two terms x and y, a context P abstracting over x and y, and an equality proof p ; in order to produce a value $P(p)$, we can assume that y can be substituted by x and p by refl_x in P.

With this induction principle, we can prove that Leibniz equality is substitutive as follows:

$$\mathsf{subst} : (P : A \to \textsc{Type}) \to x = y \to P(x) \to P(y)$$
$$\mathsf{subst}\ \ P\ \equiv\ =\text{-rect}\ (\lambda x.\ \lambda y.\ \lambda_.\ P(x) \to P(y))\ (\lambda x.\ \mathsf{id}).$$

Or, with dependent pattern-matching:

$$\mathsf{subst} : (P : A \to \textsc{Type}) \to x = y \to P(x) \to P(y)$$
$$\mathsf{subst}\ \ P\ \ \mathsf{refl}\ \equiv\ \mathsf{id}.$$

As a result, given a proof $p : x = y$, the function subst allows x to be substituted by y in *any* context P abstracting over x. This substitution operation is quite common. Therefore, we introduce the following convenient notation:

$$_{*} : \{A : \textsc{Type}\} \to \{P : A \to \textsc{Type}\} \to \{x\ y : A\} \to x = y \to P(x) \to P(y)$$
$$p_* \ \equiv\ \mathsf{subst}\ P\ p$$

where the context P is left to be inferred. Thus, subst $P\ p\ t$ may be abbreviated as $p_*(t)$.

It is important to note that, even though two definitionally equal terms are propositionally equal, the converse does not hold in general. For instance, in Coq, we can prove that $n + 0 = n$

[1] See Coq reference manual for a detailed description of each reduction rule.

[2] The identity type $_{=}_$ qualifies as Prop-extended.

holds but not $n+0 \not\equiv n$. Leibniz equality is an *intensional equality* and not an *extensional equality* since it does not satisfy the *reflection principle*:

$$\textit{reflection rule}\ \frac{A = B}{A \equiv B}.$$

One benefit of intensional equality is to ensure that the type-checking problem remains decidable. This is not the case with an extensional equality because the reflection rule could make the type-checking procedure loop forever.

Mere propositions

A type P is called a *mere proposition* when it satisfies the following property:

$$\forall (p, q : P).\ p = q.$$

Note that, being a mere proposition is a property over types and must not be confused with the sort PROP which is a universe of *logical propositions*.

Setoids

Sometimes Leibniz equality is too restrictive but it is often convenient to be able to identify terms *up-to equivalence*—for a suitable notion of equivalence. For example, one may want to consider equality on functions up-to extensionality. In set theory, we can achieve this by considering the quotient set $A/\!\sim$ over an equivalence relation $\sim$ on A. However, since we are lacking proper quotient type [Coh13], we will use a slightly weaker definition by considering *setoids* [BCP03].

Formally, the type of setoids is defined as a pair $(A, \sim)$ where A is a type and $\sim$ is a binary relation over A which is an *equivalence relation*, *i.e.*, a relation that is:

Reflexive. $\forall (x : A).\ x \sim x$,
Symmetric. $\forall (x, y : A).\ x \sim y \to y \sim x$,
Transitive. $\forall (x, y, z : A).\ x \sim y \to y \sim z \to x \sim z$.

It is often convenient to identify a setoid $(A, \sim)$ to its carrier A in a context where a type is expected. In this case, we say that is there is a *coercion* from setoids to TYPE.

Setoid morphisms generalize the notion of function on setoids. Formally, a setoid morphism from two setoids $(A, \sim_A)$ and $(B, \sim_B)$, denoted $A \longrightarrow B$, is defined as a function $f : A \to B$ on the underlying carriers such that f preserves the equivalence relations:

$$\forall a.\ \forall a'.\ a \sim_A a' \to f(a) \sim_B f(a').$$

Moreover, we overload the notations of functions to denote the identity setoid morphism (id) and the composition of setoid morphisms ($_\circ_$). Setoid morphisms can be generalized further to dependent arrows by considering indexed setoids.

Let $_\approx_$ be a binary relation over A and $P : A \to \text{TYPE}$ be a type family. We say that P *respects* the relation $_\approx_$ when:

$$\forall (a, a' : A).\ a \approx a' \to P(a) \to P(a'). \qquad (P \text{ RESPECTS } \approx)$$

A setoid morphism $f : A \longrightarrow B$ is called a *split epimorphism* when it has a right-inverse, *i.e.*, when there exists a setoid morphism $g : B \longrightarrow A$ such that the composite $f \circ g$ yields the identity setoid morphism. In this case, the morphism g is called the *section* of f.

Conversely, when the morphism f is left-invertible, it is called a *split monomorphism* and g is known as the *retraction* of f. Finally, we say that A is a *retract* of B whenever there exists a split monomorphism $A \longrightarrow B$.

A setoid morphism $f : A \longrightarrow B$ that has an inverse is called a setoid *isomorphism*. We write $A \cong B$ when there exists an isomorphism from A to B.

Type classes

Within the Coq proof assistant, setoids and setoid rewriting [Soz09] is achieved by means of *type classes* [SO08]. Contrary to languages such as Haskell, type classes in dependent type theory are not necessarily a primitive construction. As it is the case for Agda or Coq, type classes are simply defined as a syntactic sugar on top of dependent records with some additions in the implicit arguments inference algorithm to perform instance resolution. Type classes are particularly useful as they introduce some form of overloading, thus supporting a form of ad hoc polymorphism. Thus, it possible to use a common interface (or notation) but with different definitions depending on the instance considered. Finally, thanks to the instance resolution procedure, we can leave much of the details implicit. We follow the same approach in the rest of this thesis.

As an illustration of the use of type classes, we introduce a class for types with a decidable equality:

class $\textsc{EqDec}\ (A : \textsc{Type}) : \textsc{Type}$
  $_ \overset{?}{=} _ : \forall (x, y : A).\ \textsc{Dec}(x = y).$

The function $_ \overset{?}{=} _$ has an implicit parameter of type $\textsc{EqDec}\, A$ for some type A. This parameter is automatically inferred when a valid instance exists in the context. Thus, it is possible to use the same notation and write $n \overset{?}{=} n'$ or $b \overset{?}{=} b'$ with $n, n' : \mathbb{N}$ and $b, b' : \mathbb{B}$, provided that an instance exists for both $\mathbb{N}$ and $\mathbb{B}$.

We conclude this section by defining various setoids on most of the base types presented in Table 1.1.

Canonical setoid. For any type A, we can form the setoid $(A, _=_)$.
Function setoid. For any type A and setoid $(B, _\sim_B_)$, we can form the setoid $(A, _\sim_{\mathsf{fext}}_)$ where $_\sim_{\mathsf{fext}}_$ denotes function extensionality:

$$f \sim_{\mathsf{fext}} g \overset{\text{def}}{\iff} \forall (x : A).\ f(x) \sim_B g(x).$$

Sigma-type setoid. For any setoid $(A, _\sim_A_)$ and type family $B : A \to \textsc{Type}$ respecting the equivalence relation $_\sim_A_$, we can form the setoid $(\Sigma\ A\ B, _\sim_\Sigma_)$ where $_\sim_\Sigma_$ is defined as:

$$p \sim_\Sigma p' \overset{\text{def}}{\iff} \mathsf{proj}_1\, p \sim_A \mathsf{proj}_1\, p'.$$

1.2. FINITE TYPES

Finite types are used extensively throughout this thesis. In particular, they are used to define the regularity property of infinite trees. In this section, we introduce various definitions of finite types. This variety of definitions allows one to use the most suited definition depending on context. Next, we study the relationship between all of these definitions. Note that, in a classical setting, all of these definitions are equivalent. Most of the material presented in this section is a straightforward generalization to setoids of the results found in [FU15, SC10, GP15].

1.2.1. Canonical Finite Types

Canonical finite sets are represented as a type indexed by their cardinal. It is the representation, in type theory, of the set $\{m : m < n\}$.

Definition 1 **Canonical finite set**

The type of canonical finite sets is defined as:

$$\textsc{FiniteSet}\, n := \{\, m : \mathbb{N} \mid m < n \,\}\,.$$

Equivalently, canonical finite sets can be represented as an inductive family as follows:

$$\begin{aligned}&\textbf{inductive}\ \textsc{Fin} : \mathbb{N} \to \textsc{Type}\\ &\quad \mathsf{zero} : \{n : \mathbb{N}\} \to \textsc{Fin}(\mathsf{suc}\, n)\\ &\quad \mathsf{suc} : \{n : \mathbb{N}\} \to \textsc{Fin}\, n \to \textsc{Fin}(\mathsf{suc}\, n).\end{aligned}$$

Here, the constructors zero and suc should be thought of as a way to index elements—in a canonical way—within a finite set. Consequently, the constructor zero indexes the first element, while suc is used to get to the next index.

Proposition 1 **Canonical finite set isomorphism**

For any natural number n, we have:

$$\textsc{FiniteSet}\, n \cong \textsc{Fin}\, n.$$

Proof. In order to prove the isomorphism, we have to give two maps:

$$\begin{aligned}&\mathsf{from} : \{n : \mathbb{N}\} \to \textsc{FiniteSet}\, n \to \textsc{Fin}\, n\\ &\mathsf{from}\ \{\mathsf{zero}\}\ \ (_, ())\\ &\mathsf{from}\ \{\mathsf{suc}\, n\}\ \ (\mathsf{zero}, _)\ \equiv\ \mathsf{zero}\\ &\mathsf{from}\ \{\mathsf{suc}\, n\}\ \ (\mathsf{suc}\, m, p)\ \equiv\ \mathsf{suc}(\mathsf{from}(m, p'))\end{aligned}$$

where p' is a proof of $m < n$ deduced from the proof $p : \mathsf{suc}\, m < n$. The second map is given by:

$$\begin{aligned}&\mathsf{to} : \{n : \mathbb{N}\} \to \textsc{Fin}\, n \to \textsc{FiniteSet}\, n\\ &\mathsf{to}\ \ \mathsf{zero}\ \ \equiv\ (\mathsf{zero}, p)\\ &\mathsf{to}\ \ (\mathsf{suc}\, n)\ \equiv\ (\mathsf{suc}\, m, q')\\ &\qquad \textbf{where}\ \ (m, q)\ \equiv\ \mathsf{to}(n)\end{aligned}$$

where p is a proof that $0 < \mathsf{suc}\, n$ and q' is a proof that $\mathsf{suc}\, m < \mathsf{suc}\, n$ deduced from the proof $q : m < n$. Proving that to is the inverse of from is immediate by induction. □

Now, we prove various closure properties on finite sets such as the closure by sum, product and exponentiation. These results are shown by using the inductive representation of finite sets.

Proposition 2 **Empty and singleton sets**

We have the following isomorphisms:

$$\textsc{Fin}\, 0 \cong \mathbb{0} \ \ \textit{and}\ \ \textsc{Fin}\, 1 \cong \mathbb{1}.$$

Proof. Immediate. □

Remark. The type $\textsc{Fin}\, 0$ represents the *empty set* while $\textsc{Fin}\, 1$ denotes the canonical *singleton set*.

Proposition 3 **Decomposition of a finite set**

Every finite set with at least one element can be decomposed as the union of two finite sets:

$$\textsc{Fin}(\mathsf{suc}\, n) \cong \mathbb{1} \uplus \textsc{Fin}\, n.$$

Proof. Immediate. □

Proposition 4 **Closure properties**

Canonical finite sets are closed under sum, product and exponentation.

Proof.

Sum. $\forall (n, m : \mathbb{N}).\ \textsc{Fin}(n+m) \cong \textsc{Fin}\, n \uplus \textsc{Fin}\, m.$

Let n, m be two natural numbers. We proceed by induction on n:

- **Base step.** Assume that $n = 0$.

$$\begin{aligned}
\textsc{Fin}(0+m) &\equiv \textsc{Fin}\, m \\
&\cong \mathbb{0} \uplus \textsc{Fin}\, m && (\mathbb{0} \text{ neutral for } \uplus) \\
&\cong \textsc{Fin}\, \mathsf{zero} \uplus \textsc{Fin}\, m. && (\text{by Prop. 1.2 } [\textsc{Fin}\, 0 \cong \mathbb{0}])
\end{aligned}$$

- **Inductive step.** Assume that $n = \mathsf{suc}\, n'$ where $n' : \mathbb{N}$.
The inductive hypothesis is given by

$$\forall (m : \mathbb{N}).\ \textsc{Fin}(n' + m) \cong \textsc{Fin}\, n' \uplus \textsc{Fin}\, m. \tag{IH}$$

$$\begin{aligned}
\textsc{Fin}(\mathsf{suc}\, n' + m) &\equiv \textsc{Fin}(\mathsf{suc}(n' + m)) \\
&\cong \mathbb{1} \uplus \textsc{Fin}(n' + m) && (\text{by Prop. 1.3 } [\textsc{Fin}(\mathsf{suc}\, n) \cong \mathbb{1} \uplus \textsc{Fin}\, n]) \\
&\cong \mathbb{1} \uplus (\textsc{Fin}\, n' \uplus \textsc{Fin}\, m) && (\text{by induction hypothesis (IH)}) \\
&\cong \textsc{Fin}(\mathsf{suc}\, n') \uplus \textsc{Fin}\, m. && (\text{by Prop. 1.3 and associativity of } \uplus)
\end{aligned}$$

Product. $\forall (n, m : \mathbb{N}).\ \textsc{Fin}(n \times m) \cong \textsc{Fin}\, n \times \textsc{Fin}\, m.$

Let n, m be two natural numbers. We proceed by induction on n:

- **Base step.** Assume that $n = 0$.

$$\begin{aligned}
\textsc{Fin}(0 \times m) &\equiv \textsc{Fin}\, \mathsf{zero} \\
&\cong \mathbb{0} \times \textsc{Fin}\, m && (\mathbb{0} \text{ zero for } \times \text{ and Prop. 1.2 } [\textsc{Fin}\, 0 \cong \mathbb{0}]) \\
&\cong \textsc{Fin}\, \mathsf{zero} \times \textsc{Fin}\, m. && (\text{by Prop. 1.2})
\end{aligned}$$

- **Inductive step.** Assume that $n = \mathsf{suc}\, n'$ where $n' : \mathbb{N}$.
The induction hypothesis is given by

$$\forall (m : \mathbb{N}).\ \textsc{Fin}(n' \times m) \cong \textsc{Fin}\, n' \times \textsc{Fin}\, m. \tag{IH}$$

$$\begin{aligned}
\textsc{Fin}(\mathsf{suc}\, n' \times m) &\equiv \textsc{Fin}(m + n' \times m) \\
&\cong \textsc{Fin}\, m \uplus \textsc{Fin}(n' \times m) && (\text{closure of sum}) \\
&\cong \textsc{Fin}\, m \uplus (\textsc{Fin}\, n' \times \textsc{Fin}\, m) && (\text{by induction hypothesis (IH)}) \\
&\cong (\mathbb{1} \times \textsc{Fin}\, m) \uplus (\textsc{Fin}\, n' \times \textsc{Fin}\, m) && (\mathbb{1} \text{ neutral for } \times) \\
&\cong \textsc{Fin}(\mathsf{suc}\, n') \times \textsc{Fin}\, m. && (\text{distributivity of } \times \text{ over } \uplus)
\end{aligned}$$

Exponentiation. $\forall(n, m : \mathbb{N})$. $\textsc{Fin}(n^m) \cong \textsc{Fin}\, m \longrightarrow \textsc{Fin}\, n$.

Let n, m be two natural numbers. We proceed by induction on m:

- **Base step.** Assume that $m = 0$.

$$\begin{aligned} \textsc{Fin}\, n^0 &\equiv \textsc{Fin}\, 1 \\ &\cong \mathbb{0} \longrightarrow \textsc{Fin}\, n && \text{(exactly one morphism from } \mathbb{0} \text{ to any setoid)} \\ &\cong \textsc{Fin}\, \mathsf{zero} \longrightarrow \textsc{Fin}\, n. && \text{(by Prop. 1.2 } [\textsc{Fin}\, 0 \cong \mathbb{0}]) \end{aligned}$$

- **Inductive step.** Assume that $m = \mathsf{suc}\, m'$.

  The induction hypothesis is given by

$$\textsc{Fin}\, n^{m'} \cong \textsc{Fin}\, m' \longrightarrow \textsc{Fin}\, n. \tag{IH}$$

$$\begin{aligned} \textsc{Fin}\, n^{\mathsf{suc}\, m'} &\equiv \textsc{Fin}\left(n \times n^{m'}\right) \\ &\cong \textsc{Fin}\, n \times \textsc{Fin}\, n^{m'} && \text{(closure of finite sets under product)} \\ &\cong (\mathbb{1} \uplus \textsc{Fin}\, m') \longrightarrow \textsc{Fin}\, n && (A \times (B \longrightarrow A) \cong (\mathbb{1} \uplus B) \longrightarrow A) \\ &\cong \textsc{Fin}(\mathsf{suc}\, m') \longrightarrow \textsc{Fin}\, n. && \text{(by Prop. 1.3 } [\textsc{Fin}(\mathsf{suc}\, n) \cong \mathbb{1} \uplus \textsc{Fin}\, n]) \end{aligned}$$

□

1.2.2. Finite Setoids

Definition 2 **Finite setoid**

A setoid A is finite if there exists a natural number n such that A is isomorphic to $\textsc{Fin}\, n$. Formally,

$$\textsc{Finite}\, A := \sum_{n:\mathbb{N}} A \cong \textsc{Fin}\, n.$$

The number n is called the cardinal of A. When A is finite, we write $\#A$ to denote its cardinal.

Remark. When a type satisfied Definition 1.2, it is also called *finitely enumerable* or *Bishop-finite.*

Another common definition of finiteness of a type/setoid is characterized by the ability to list all of its inhabitants.

Definition 3 **DF-Listable / Listable**

A setoid A is listable without duplicates if there exists a list $l : \textsc{List}\, A$ such that:

(i) $\forall(x : A).\ x\ [\in]\ l$,

(ii) *l is duplicate-free.*

When the list may contain duplicates, we say that A is listable.

Remark. In the definition of listable, the membership relation $_[\in]_$ (see Definition A.36) designates the proof-relevant variant of $_\in_$. This means that from a proof of $x\ [\in]\ l$ we can extract the position of x within the list l. This is not (always) possible from a proof of $x \in l$ since it lives in the sort $\textsc{Prop}$.

Proposition 5 **Finite and listability**

A setoid A is finite if and only if it is listable without duplicates.

Proof. In order to make the proof more tractable, we use the following decomposition:

$$\begin{aligned}\textsf{FINITE}\,A &\overset{(i)}{\iff} \sum_{n:\mathbb{N}}\ \sum_{v:\textsf{VEC}\,A\,n}(\forall x.\ x\,[\in]\,v) \times \textsf{NODUP}\,v\\ &\overset{(ii)}{\iff} \sum_{l:\textsf{LIST}\,A}(\forall x.\ x\,[\in]\,l) \times \textsf{NODUP}\,l.\end{aligned}$$

The definitions of the membership relation along with the predicate NODUP can be found in Appendix A, for both vectors and lists. By definition, FINITE A is an isomorphism between a canonical finite set FIN $\#A$ and A. We call index the map $A \to \textsf{FIN}\,\#A$ and $\mathsf{index}^{-1} : \textsf{FIN}\,\#A \to A$ the map in opposite direction. Now, we prove both equivalences:

(i) FINITE $A \iff \sum_{n:\mathbb{N}} \sum_{v:\textsf{VEC}\,A\,n}(\forall x.\ x\,[\in]\,v) \times \textsf{NODUP}\,v$

($\Longrightarrow$) Assume that A is finite.
We must show that there exists a duplicate-free vector containing all elements of A. We pick $n := \#A$ (the cardinal of A) and $v := \mathsf{tabulate}\ \mathsf{index}^{-1}$. Now, we prove that v contains every inhabitant of A. Let $a : A$. To show that $a\,[\in]\,v$, we pick the first projection, called i, to be the index given by $i := \mathsf{index}\,a$. It remains to show that $v(i) \approx a$. By equational reasoning:

$$\begin{aligned} v(i) &\equiv \left(\mathsf{tabulate}\ \mathsf{index}^{-1}\right)(\mathsf{index}\,a) & \\ &= \mathsf{index}^{-1}(\mathsf{index}\,a) & &(\text{by Prop. A.8 } [\mathsf{lookup}(\mathsf{tabulate}\,f) \doteq f])\\ &\approx a. & &(\mathsf{index}^{-1} \text{ is the inverse of } \mathsf{index})\end{aligned}$$

Finally, proving that v is duplicate-free consists in establishing the injectivity of the map $\mathsf{lookup}\,v : \textsf{FIN}\,n \to A$. By definition of v and Proposition A.9, we have $\mathsf{lookup}\,v = \mathsf{lookup}\left(\mathsf{tabulate}\ \mathsf{index}^{-1}\right)$ that is extensionally equivalent to index^{-1}. In particular, since index^{-1} is an isomorphism, it is also injective.

($\Longleftarrow$) Assume that we have $n : \mathbb{N}$ and $v : \textsf{VEC}\ A\ n$ such that $M : \forall x.\ x[\in]v$ and $D : \textsf{NODUP}\,v$. To show that A is finite, we have to construct an isomorphism from A to a canonical finite set. We define the cardinal of A to be n. It remains to give two setoid morphisms $f : \textsf{FIN}\,n \longrightarrow A$ and $g : A \longrightarrow \textsf{FIN}\,n$ and prove that they are inverse of one another. The map f is defined as $f := \mathsf{lookup}\,v$ while the map g is given by $g := \mathsf{proj}_1 \circ M$. Showing that f is a setoid morphism is trivial. For g, it is a direct consequence of the fact that the vector does not contain duplicates. Finally, We show that g is the inverse of f:

- Let $i : \textsf{FIN}\,n$. To prove that $g(f(i)) = \mathsf{proj}_1(M(v(i))) = i$, we use the hypothesis D stating that lookup is injective. It remains to show that $v(\mathsf{proj}_1(M(v(i)))) = v(i)$ which is actually given by $\mathsf{proj}_2(M(v(i)))$.
- Let $a : A$. The proof of $f(g(a)) = v(\mathsf{proj}_1(M(a))) = a$ is given by $\mathsf{proj}_2(M(v(i)))$.

(ii) $\sum_{n:\mathbb{N}} \sum_{v:\textsf{VEC}\,A\,n}(\forall x.\ x\,[\in]\,v) \times \textsf{NODUP}\,v \iff \sum_{l:\textsf{LIST}\,A}(\forall x.\ x\,[\in]\,l) \times \textsf{NODUP}\,l.$

Straightforward by induction on vectors from left to right, and by induction on lists from right to left. The maps converting a vector to a list and back are defined in Appendix A. ☐

Proposition 6 — Finite closure

Finite setoids are closed under sum, product and exponentiation.

Proof. Consequence of Proposition 1.4 [FIN closure]. ☐

1.2.3. Finitely Indexed Setoids

We consider another definition of finiteness commonly known as *Kuratowski-finiteness*.

Definition 4 **Finitely indexed**

A setoid A is said to be finitely indexed if there exists a split monomorphism from A into a canonical finite type. Formally,

$$\textsc{FinitelyIndexed}\, A := \sum_{n:\mathbb{N}} \textsc{Retract}(A, \textsc{Fin}\, n).$$

Remark. In Definition 1.4, the natural number n denotes an upper bound of the cardinal A. Finitely indexed setoids are sometimes called *finitely generated setoids*.

Proposition 7 **Finitely indexed and listability**

A setoid A is said to be finitely indexed if and only if there exists a list $l : \textsc{List}\, A$ and a map $M : \forall(x : A).\ x\, [\in]\, l$ such that $\forall(x, y : A).\ x \approx y \to \mathsf{proj}_1(M(x)) = \mathsf{proj}_1(M(y))$.

Proof. The proof is similar to the one given in Prop. 1.5 [FINITE ⇔ DF-LISTABLE]. □

Remark. In Proposition 1.7, in order to construct the split monomorphism from the list l, it is not enough to specify that l contains every inhabitant of A. It also mandatory to require that the membership proofs respects the underlying setoid equality. In Proposition 1.5, it is not necessary because we have proof that the list l is duplicate-free. From this fact, we can derive that each membership proof ought to be unique and thus respects the underlying setoid equality.

Proposition 8 **Finite equivalent to finitely indexed**

A setoid is finite if and only if it is finitely indexed.

Proof. Let A be a setoid.

($\Longrightarrow$) Assume that A is finite.
Thus, there exists an isomorphism $A \longrightarrow \textsc{Fin}\, \#A$. In particular, the map $A \longrightarrow \textsc{Fin}\, \#A$ is a split monomorphism.

($\Longleftarrow$) Assume that A is finitely indexed.
Then, by definition there exists a split epimorphism f from an initial segment of $\mathbb{N}$. To prove that A is finite it suffices to show that the underlying setoid equality on A is decidable. Indeed, with Proposition 1.7 [FINITELYINDEXED listability], we obtain a list containing all inhabitants of A. Then, with the decidability of the underlying A equality, we can remove any duplicate. Thus, it remains to prove the following statement:

$$\forall(a, a' : A).\ \textsc{Dec}(a \approx a').$$

We call $\mathsf{index} : A \longrightarrow \textsc{Fin}\, n$ and $\mathsf{value} : \textsc{Fin}\, n \longrightarrow A$ the retraction of index. Since equality is decidable on canonical finite sets (Proposition A.6), we can compare the values of the indices a and a' to deduce whether $a \approx a'$. Consequently, we consider two cases:

- **Case 1.** Assume that $\mathsf{index}\, a = \mathsf{index}\, a'$.
  We can conclude that $a \approx a'$ since the function index is injective.

- **Case 2.** Assume that $\mathsf{index}\, a \neq \mathsf{index}\, a'$.
  We must show that $a \not\approx a'$. Assume that $a \approx a'$. Since the function index is a setoid morphism, it is a congruence, hence $\mathsf{index}\, a \approx \mathsf{index}\, a'$. Contradiction. □

Proposition 9 **Closure properties**

Finitely indexed types are closed under sum, product and exponentiation.

Proof. This is a consequence of Prop. 1.8 establishing the equivalence between finite setoids and finitely indexed setoids along with the proof of the closure property on finite setoids(Prop. 1.4). □

1.2.4. Weakly Finite Indexed Setoids

In the previous section, finitely indexed setoids were defined by the existence of a split epimorphism from an initial segment of $\mathbb{N}$. Now, we consider a slightly weaker assumption in the sense that the indexing map is not required to be a setoid morphism. The main consequence is that equivalent elements in the setoid are not required to be indexed by the same element.

Definition 5 **Weakly Finitely Indexed Setoids**

A setoid A is said to be weakly finitely indexed when there exist a natural number n and two maps $f : \textsc{Fin}\, n \to A$ and $g : A \to \textsc{Fin}\, n$ such that g is a right-inverse of f. Formally, we define it as a dependent record as follows:

record $\textsc{WFI}\ (A : \textsc{Setoid}) : \textsc{Type}$
  constructor $\langle_,_,_,_\rangle$

$$\left[\begin{array}{l} \mathsf{ucard} : \mathbb{N} \\ \mathsf{index} : A \to \textsc{Fin}\, \mathsf{ucard} \\ \mathsf{value} : \textsc{Fin}\, \mathsf{ucard} \to A \\ \quad \mathsf{rinv} : \forall (a : A).\ \mathsf{value}(\mathsf{index}\, a) \approx_A a. \end{array}\right.$$

Remark. Given a weakly finitely indexed setoid A, we may write $\#A$ instead of $\mathsf{ucard}\, A$ to denote the upper bound of the cardinal of A.

Proposition 10 **Finitely indexed to weakly finitely indexed**

A finitely indexed setoid A is weakly finitely indexed.

Proof. Immediate. □

Proposition 11 **Weakly finitely indexed to finitely indexed**

Intuitionistically, there is no map turning a weakly finitely indexed setoid A into a finitely indexed setoid.

Proof. Let A be a setoid such that A is weakly finitely indexed. We prove that if a map:

$$\phi : \forall A.\ \textsc{WFI}\, A \to \textsc{FinitelyIndexed}\, A$$

were to exist, then it would entail the law of excluded-middle (LEM). Assume that the function ϕ exists and let $P : \textsc{Prop}$ be an arbitrary truth value. We can form the type defined as:

$S : \textsc{Type}$
$S \equiv \sum_{p:\textsc{Prop}} (p = \top \vee p = P)$.

The equivalence relation considered on S is given by logical equivalence on propositions:

$$s \approx_S s' \overset{\text{def}}{\iff} \mathsf{proj}_1\, s \leftrightarrow \mathsf{proj}_1\, s'.$$

Clearly, the type S is weakly finitely indexed as it is listable (Proposition 1.12) where the list is given by $[(\top, \text{left refl}); (P, \text{right refl})]$. Thus, by ϕ we have that S is finitely indexed and by Proposition 1.8, S is also finite. As a result, we can now use the fact that the underlying setoid equality on S is decidable. By case-analysis on $(\top, \text{left refl}) \overset{?}{\approx} (P, \text{right refl})$, we consider two cases:

- **Case 1.** Assume that $e : \top \leftrightarrow P$.
  From e, we can conclude that P holds.
- **Case 2.** Assume that $e : \neg(\top \leftrightarrow P)$.
  From e, we can conclude that $\neg P$ holds.

To summarize, we have shown that either P or $\neg P$ hold, thus proving LEM. □

Proposition 12 **Weakly finitely indexed equivalent to listable**

A setoid A is weakly finitely indexed if and only if A is listable.

Proof. The proof is similar to the one given in Prop. 1.7 [FINITELYINDEXED listability]. □

The following proposition establishes that weakly finitely indexed setoids are preserved by a "weak" form of split epimorphism. Here, by weak, we mean that the section is not required to be a setoid morphism. An immediate corollary is that weakly finitely indexed are preserved by isomorphisms.

Proposition 13 **Weakly finitely indexed preserved by "weak" split epimorphism**

Let A be a weakly finitely indexed setoid and B a setoid. If there exists two maps $f : A \longrightarrow B$ and $g : B \to A$ such that g is a right-inverse of f then B is weakly finitely indexed.

Proof. Let A and B be two setoids. Moreover, let $f : A \longrightarrow B$ be a setoid morphism and $g : B \to A$ such that g is a right-inverse of f. Furthermore, assume that A is weakly finitely indexed. Thus, by definition there exist two maps $\text{index}_A : A \to \text{FIN}\,\#A$ and $\text{value}_A : \text{FIN}\,\#A \to A$ such that index is a right-inverse of value. First, to prove that B is weakly finitely indexed, we have to give an upper bound on its cardinal. We pick $\#B := \#A$. Then, we define the two maps as:

$$\text{index}_B : B \to \text{FIN}\,\#B \qquad \text{value}_B : \text{FIN}\,\#B \to B$$
$$\text{index}_B \equiv \text{index}_A \circ g. \qquad \text{value}_B \equiv f \circ \text{value}_A.$$

Finally, it remains to show that index_B is a right-inverse of value_B. Let $b : B$:

$$\begin{aligned} \text{value}_B(\text{index}_B\, b) &\equiv f(\text{value}_A(\text{index}_A(g(b)))) \\ &\approx_B f(g(b)) && (\text{index}_A \text{ right-inverse of } \text{value}_A) \\ &\approx_B b. && (g \text{ right-inverse of } f) \end{aligned}$$
□

In the following, we show that weakly finitely indexed setoids are enough to prove the *pigeon hole principle*. This proof is based on well-founded induction over $\succ$-accessible lists (Def. A.38).

Lemma 1 **Pigeon hole principle on finite types**

Let A be a type with a decidable Leibniz equality. Let l be a list of elements of A such that the list l is $\succ$-accessible. For any stream $s : \mathbb{N} \to A$, if for all $k : \text{FIN}(\text{length}\, l)$, we have $l(k) = s(\text{length}\, l - \text{suc}(\text{to}\mathbb{N}\, k))$, then there exist two distinct positions i, j such that $s(i) = s(j)$.

Proof. Let A be a type and $_\overset{?}{=}_ : \forall a.\, \forall a'.\, \text{DEC}(a = a')$ be the map deciding Leibniz equality on A. Moreover, let $s : \mathbb{N} \to A$ denote a stream of elements of A. Formally, we have to prove the following statement:

$$\forall (l : \text{LIST}\, A).\ \lceil l \rceil \to (\forall k.\ l(k) = s(\text{length}\, l - \text{suc}(\text{to}\mathbb{N}\, k))) \to \sum_{i,j:\mathbb{N}} i \neq j \wedge s(i) = s(j).$$

We proceed by induction on the proof of $\lceil l \rceil$ ($\succ$-accessibility of l):

- **Base step.** Inductive type without a base constructor.
- **Inductive step.** Assume that $l : \text{List}\, A$, $B : \lceil l \rceil$ and $Inv : \forall k.\ l(k) = s(\mathsf{length}\, l - \mathsf{suc}(\mathsf{to}\mathbb{N}\, i))$. The inductive hypothesis is given by:

$$\forall l'.\ l' \prec l \to (\forall k.\ l'(k) = s(\mathsf{length}\, l' - \mathsf{suc}(\mathsf{to}\mathbb{N}\, k)) \to \sum_{i,j:\mathbb{N}} i \neq j \wedge s(i) = s(j). \qquad \text{(IH)}$$

By case-analysis on the result of membership decision procedure $s(\mathsf{length}\, l) \in^? l$:

  - **Case 1.** Assume that $E : s(\mathsf{length}\, l) \in l$.
    From E, we can extract an index i such that $l(i) = s(\mathsf{length}\, l)$. Consequently, we have found two positions i and $\mathsf{length}\, l$ such that $s(i) = s(\mathsf{length}\, l)$. Moreover, it is straightforward to check that both indices are distinct because and index in a list ought to be strictly smaller than its length.

  - **Case 2.** Assume that $E : s(\mathsf{length}\, l) \notin l$.
    We use the induction hypothesis (IH) with the list l' picked to be $s(\mathsf{length}\, l) :: l$. The proof of $l' \prec l$, *i.e.*, that $l \subset s(\mathsf{length}\, l) :: l$, is straightforward because of assumption E. Finally, it remains to prove:

$$\forall (k : \text{Fin}(\mathsf{length}\, l)).\ l'(k) = s(\mathsf{length}\, l - \mathsf{suc}(\mathsf{to}\mathbb{N}\, k)).$$

    By case-analysis on k, we consider two cases:

    - **Case 2.1.** Assume that $k = \mathsf{zero}$.
      We have $l'(\mathsf{zero}) = s(\mathsf{length}\, l) = s(\mathsf{length}\, l - 0) = s(\mathsf{suc}(\mathsf{length}\, l) - \mathsf{suc}\,\mathsf{zero})$. This holds by reflexivity, provided that $\mathsf{length}\, l - 0$ has been rewritten into $\mathsf{length}\, l$.

    - **Case 2.2.** Assume that $k = \mathsf{suc}\, k'$ where $k' : \text{Fin}(\mathsf{length}\, l)$:
      We conclude immediately with hypothesis Inv. □

Theorem 1 — Pigeon hole principle

Let A be a setoid such that A is weakly finitely indexed. For any stream $s : \mathbb{N} \to A$, there exists two distinct positions $i, j : \mathbb{N}$ such that $s(i) \approx s(j)$.

Proof. Let $(A, \approx)$ be a weakly finitely indexed setoid. Thus, by definition there exist two maps $\mathsf{index} : A \to \text{Fin}\,\#A$ and $\mathsf{value} : \text{Fin}\,\#A \to A$ such that $\forall a.\ \mathsf{value}(\mathsf{index}\, a) \approx a$. Let $s' : \mathbb{N} \to A$ denote the stream of elements of A. We instantiate Lemma 1.1 with $l := []$ and $s := \mathsf{index} \circ s'$. The proof of $\lceil [] \rceil$ is given by the well-foundedness of $_\succ_$ (Prop. A.15) while the proof of $\forall k.\ l(k) = s(\mathsf{length}\, l - \mathsf{suc}(\mathsf{to}\mathbb{N}\, k))$ is given by elimination of the empty type on $k : \text{Fin}(\mathsf{length}\, []) \equiv \text{Fin}\,\mathsf{zero}$. Thus, we obtain two distinct positions i, j such that $E : s(i) = s(j)$.

Finally, it remains to find two distinct positions i', j' such that $s'(i) \approx s'(j')$. We pick $i' := i$ and $j' := j$. By equational reasoning:

$$\begin{aligned} s'(i') &\approx \mathsf{value}(\mathsf{index}(s'(i))) && (\mathsf{index} \text{ right-inverse of } \mathsf{value}) \\ &\approx \mathsf{value}(\mathsf{index}(s'(j))) && (\text{by rewriting hypothesis } E) \\ &\approx s'(j'). && (\mathsf{index} \text{ right-inverse of } \mathsf{value}) \end{aligned}$$

□

1.2.5. Exploration Function

In this section, we introduce yet another characterization of finiteness based on exploration functions. Intuitively, an exploration function can be thought of as the operation of mapping a function to each element of a type and then combine all intermediate results.

One interesting use-cases of exploration functions is to derive big-operators such as sum and product. For instance, if we have a map $f : I \to \mathbb{N}$, we may want to define an operation representing the sum of all $f(i)$, for all $i : I$. This operation is usually written as:

$$\sum_{i:I} f(i).$$

Then, if we assume that I is listable, *i.e.*, there exists a list $l = [i_0; i_1; \ldots; i_n]$ containing all inhabitants of I, the operation can be computed as follows:

$$\sum_{i:I} f(i) = f(i_0) + f(i_1) + \ldots + f(i_n).$$

Here, we clearly see the scheme describe above. The type I is explored by mapping the function f on each of its inhabitants. Then, the intermediate computations are combined through the sum operation. Exploration functions are studied more extensively in [GP15].

Definition 6 Exploration of a type

The type of exploration functions on a type A is defined as follows:

$$\textsc{Explore}\, A := \forall (M : \textsc{Type}).\ \forall (\epsilon : M).\ \forall (_\oplus_ : M \to M \to M).\ (A \to M) \to M.$$

Moreover, any exploration function expl : $\textsc{Explore}\, A$ *shall satisfy, for all type families* $P : M \to \textsc{Type}$*, the following introduction rule*

$$\mathsf{expl\text{-}intro}\ \frac{P(\epsilon) \qquad P(m) \times P(m)' \to P(m \oplus m') \qquad \forall a.\ P(f(a))}{P(\mathsf{expl}\ \epsilon\ (_\oplus_)\ f)},$$

and elimination rule

$$\mathsf{expl\text{-}elim}\ \frac{P(m \oplus m') \to P(m) \times P(m') \qquad P(\mathsf{expl}\ \epsilon\ (_\oplus_)\ f)}{\forall a.\ P(f(a))}.$$

Remark. Definition 1.6 is given for a plain type M. However, when considering setoids, we add a side-condition on the type family $P : M \to \textsc{Type}$, namely that $P \circ f$ respects the underlying setoid equality:

$$\forall (m, m' : M).\ m \approx m' \to P(f(m)) \to P(f(m')).$$

Here, we use implication instead of an equivalence because the other direction can be derived.

Proposition 14 Explorable and listability

A setoid A is listable if and only if there exists an exploration function.

Proof. Let $(A, \approx)$ be a setoid.

$(\Longrightarrow)$ Assume that A is listable.

Thus, there exists a list l such that $\forall (a : A).\ a\ [\in]\ l$. Then, the exploration function on A is derived as follows:

$$\begin{aligned} &\mathsf{expl} : \{M : \textsc{Type}\} \to M \to (M \to M \to M) \to (A \to M) \to M \\ &\mathsf{expl}\ \ \epsilon\ \ _\oplus_\ \ f\ \ \equiv\ \ \mathsf{fold}\ \epsilon\ (\lambda a.\ \lambda r.\ f(a) \oplus r)\ l. \end{aligned}$$

Finally, the introduction and elimination rules of expl are proven by a straightforward induction over l.

($\Longleftarrow$) Assume that there exists an exploration function expl on A.
The list l enumerating the elements of A is constructed as follows:

$$\begin{aligned} &l : \textsc{List}\ A \\ &l\ \equiv \mathsf{expl}\ [\,]\ (_{+\!\!+}_)\ [_]. \end{aligned}$$

It remains to prove that every element of A is in the list l, *i.e.*, $\forall(a : A).\ a\ [\in]\ l$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{l_1{+\!\!+}l_2\ [\subseteq]\ l \to l_1\ [\subseteq]\ l}(1) \quad \dfrac{}{l_1{+\!\!+}l_2\ [\subseteq]\ l_2\ [\subseteq]\ l}(2)}{l_1{+\!\!+}l_2\ [\subseteq]\ l \to l_1\ [\subseteq]\ l \land l_2\ [\subseteq]\ l} \qquad \dfrac{}{l\ [\subseteq]\ l}}{[a]\ [\subseteq]\ l}(3)}{a\ [\in]\ l}(4)$$

where (1) and (2) are derived from the introduction rules of $_{+\!\!+}_$, (3) is given by expl-elim (with $P(m) := m \subseteq l$) and (4) is by definition of subset. $\square$

Remark. Given an exploration function on A, an upper bound for the cardinal of the type A can be computed as follows:

$$\mathsf{ucard}_A\ \equiv\ \mathsf{expl}\ \mathsf{zero}\ (_+_)\ (\lambda_.\ 1).$$

1.2.6. Streamless Setoid

Streamless types are introduced in [SC10] and some closure properties are studied in [Par14]. In this section, we generalize slightly the definition of streamless types to setoids and show the connection between streamless setoids and other notions of finiteness presented thus far. Streamless setoids will be used mainly as a tool to obtain a sufficient condition to compute fixpoints of monotonic functions on (complete) lattices.

Definition 7 **Streamless setoid**

A setoid A is said to be streamless if every stream over A has duplicates. Formally,

$$\textsc{Streamless}\ A := \prod_{s:\mathbb{N}\to A} \sum_{i,j:\mathbb{N}} i < j \land s(i) \approx_A s(j).$$

Remark. Definition 1.7 is a reformulation of the statement: "there is no injection from $\mathbb{N}$ to A". This is also similar to the pigeon hole principle introduced in the previous section. In particular, note that the stream position i, j are computationally-relevant.

Proposition 15 **Weakly finitely indexed to streamless**

Let A be a setoid. If A is weakly finitely indexed, then it is streamless.

Proof. Let A be a weakly finitely indexed setoid. Moreover, let $s : \mathbb{N} \to A$ denote a stream of elements of A. By Theorem 1.1, there exist two distinct positions i, j such that $s(i) \approx s(j)$. Clearly, since $\leq$ is total and $i \neq j$, we have either $i < j$ or $j > i$. Assume the first case holds, *i.e.*, $i < j$. As a result, we have two positions i, j such that $i < j$ and $s(i) \approx s(j)$. Consequently, we can conclude that A is streamless. The other direction holds by symmetry. $\square$

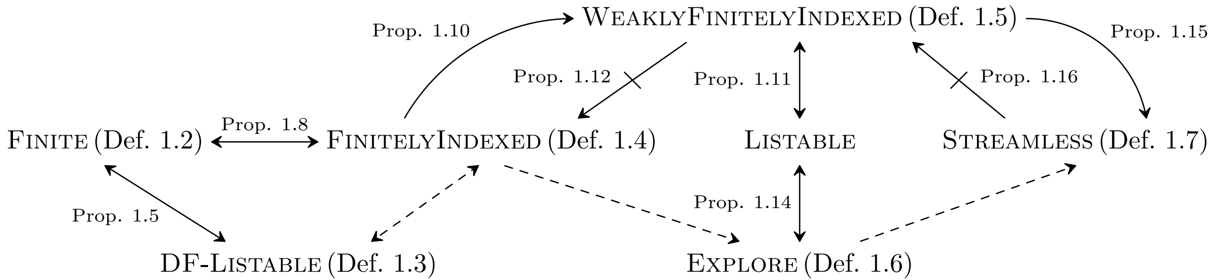


Figure 1.1. Relationships between various definitions of finite setoids.

Proposition 16 **Streamless to weakly finitely indexed**

Intuitionistically, there is no map from streamless setoids to weakly finitely indexed setoids.

Proof. To prove that no such map exists, we show that the law of excluded-middle (LEM) is derivable. Assume that there is map from streamless setoids to weakly finitely indexed setoids:

$$\phi : \forall (A : \textsc{Setoid}).\ \textsc{Streamless}\, A \to \textsf{WFI}\, A.$$

Let $P : \textsc{Prop}$ be an arbitrary proposition. We construct the setoid $S := (P, \approx_P)$ where

$$p \approx_P p' \overset{\text{def}}{\iff} \top.$$

Clearly, the setoid S is streamless: for any stream $s : \mathbb{N} \to P$, we have $s(0) \approx_P s(1) \iff \top$. By ϕ, the setoid S is also weakly finitely indexed. Thus, by definition, there exists an upper bound $\#S$ for the cardinal of S and there exist two maps, $\mathsf{index} : S \to \textsc{Fin}\, \#S$ and $\mathsf{value} : \textsc{Fin}\, \#S \to S$. By case-analysis on $\#S$, we consider two cases:

- **Case 1.** Assume that $\#A = 0$.
  We prove that $\neg P$ holds. Assume that we have a proof $p : P$. With $\mathsf{index}\, p : \textsc{Fin}\, \#S \equiv \textsc{Fin}\, 0$, we obtain a term with empty type. Contradiction.
- **Case 2.** Assume that $\#A = \mathsf{suc}\, n$ where $n : \mathbb{N}$.
  We prove that P holds. This proof is given by $\mathsf{value}\ \mathsf{zero} : P$.

As a result, we have proven that $P \vee \neg P$ holds. □

Figure 1.1 summarizes the various notions of finiteness presented in this section along with their relationships. Some relationships are the result of theorems and are annotated accordingly. Other relationships are represented with dashed lines to emphasize that they are a logical consequence of the transitivity of implication.

1.3. SIGNATURES AND INDEXED SIGNATURES

In this section, we introduce the definitions of signatures and indexed signatures. Then, we define the free term algebra (resp. coalgebra) induced by signatures. Finally, we give examples showing how signatures are used to represent (co)inductive type definitions within the type theory itself.

1.3.1. Signatures

A *signature* (or ranked alphabet) is defined to be a set of symbols or *operators*, along with a *ranking* function assigning to each operator an arity. The type of signature is represented as dependent record as follows:

record SIG : TYPE
 constructor _ ◁ _
  Op : TYPE
  Ar : Op $\to \mathbb{N}$.

The field Op is a type representing the set of operators or function symbols. Note that, traditionally the set of operators is required to be finite but we do not make this assumption in the definition of SIG. However, shall we need such assumption, we would rather add the proof of finiteness as a parameter of the definition or proposition that require it. This has the benefit of emphasizing the parts of the mechanization that actually rely on such assumption. The field Ar is a map assigning a rank to each operator. We use $\{\, \alpha_1^{(a_1)}, \ldots, \alpha_n^{(a_n)} \,\}$ as a notation to define a finite ranked alphabet: the set of operators is given by $\{\, \alpha_1, \ldots, \alpha_n \,\}$ while the rank of α_i is given by the superscript a_i, for each i. A ranked alphabet such that each operator has a rank of 1 is called a *unary ranked alphabet*.

Example. We consider a signature describing a tree-like structure with operators $\{\, \bot, ::, \times \,\}$ of arity 0, 1, and 2 respectively. This ranked alphabet is thus represented as follows:

Tree-like : SIG
Tree-like $\equiv \{\, \bot^{(0)}, ::^{(1)}, \times^{(2)} \,\}$.

Example. The signature for the type of lists is described as follows:

List-Sig : $(A :$ TYPE$) \to$ SIG
List-Sig $A \equiv \{\, [\,]^{(0)}, ::\, {}_a{}^{(1)} \,\}_{(a:A)}$.

Extension functor of a ranked alphabet. A signature S induces an endofunctor on TYPE, called the *extension functor*, where the action on types is given by:

ext : SIG → TYPE → TYPE
ext S X $\equiv \sum_{o:S.\mathsf{Op}}$ VEC X $(S.\mathsf{Ar}\, o)$.
Coercion ext : SIG ↣ SORTCLASS.

and the action on morphisms is given by:

$\mathsf{map}_{\text{SIG}} : (S : \text{SIG}) \to (X \to Y) \to S(X) \to S(Y)$
$\mathsf{map}_{\text{SIG}}\ S\ f\ (o, os) \equiv (o, \mathsf{map}_{\text{VEC}}\ f\ os)$.

Note that the function ext is left implicit in the definition of $\mathsf{map}_{\text{SIG}}$. Here, we use a feature available in the COQ proof assistant known as implicit coercions [Saï97]. Declaring ext as a coercion from signatures to sorts implies that a signature may be used in place of a sort. The conversion from signatures to sorts is done implicitly—through ext—during type-checking. This notion of *extension* of a signature will be particularly important in the next section when we define the term algebra over a signature.

Combinators. It is possible to define various combinators on signatures and establish their soundness with respect to their underlying extension functor. We give a few examples:

N-ary combinator.

$\langle _ \rangle^{-}$: TYPE → $\mathbb{N}$ → SIG
$\langle A \rangle^{n} \equiv A \lhd \mathsf{const}\, n$.

Sum combinator.

$$_\langle\uplus\rangle_ : \textsc{Sig} \to \textsc{Sig} \to \textsc{Sig}$$
$$S_1 \ \langle\uplus\rangle\ S_2 \ \equiv\ (S_1.\mathsf{Op} \uplus S_2.\mathsf{Op}) \lhd [\, S_1.\,\mathsf{Ar}, S_2.\,\mathsf{Ar}\,].$$

Product combinator.

$$_\langle\times\rangle_ : \textsc{Sig} \to \textsc{Sig} \to \textsc{Sig}$$
$$S_1 \ \langle\times\rangle\ S_2 \ \equiv\ (S_1.\mathsf{Op} \times S_2.\mathsf{Op}) \lhd \mathsf{uncurry}(\lambda o1.\ \lambda o2.\ S_1.\,\mathsf{Ar}\, o_1 + S_2.\,\mathsf{Ar}\, o_2).$$

The function $[_,_]$ denotes the dependent eliminator of the coproduct:

$$\frac{P : A \uplus B \qquad l : \forall(a : A).\ P(\mathsf{inl}\, a) \qquad r : \forall(b : B).\ P(\mathsf{inr}\, b)}{[l, r] : \forall(x : A \uplus B).\ P(x)}$$

while uncurry is the dependent eliminator of the product:

$$\frac{P : A \times B \to \textsc{Type} \qquad f : \forall(a : A).\ \forall(b : B).\ P(a, b)}{\mathsf{uncurry}\, f : \forall(x : A \times B).\ P(x)}.$$

Finally, to etablish the soundness, for instance, of the combinator $_\langle\uplus\rangle_$ we need to show how the combinator is related to its counterpart in the semantics, *i.e.*, on its extension functor. Concretely, this consists in proving the following statement:

$$\forall(X : \textsc{Type}).\ (S_1\langle\uplus\rangle S_2)(X) \cong S_2(X) \uplus S_2(X).$$

1.3.2. Indexed Signatures

Many-sorted signatures are a generalization of signatures adding typing constraints (sorts) to the arguments of operators. The type of many-sorted signatures is defined as a dependent record:

```
record MSig (S : Type)
  constructor _◁_
  [ Op : Type
    Typ : Op → List S × S.
```

The main difference with signatures is that the arity function Ar of the type Sig has been replaced by a typing function Typ which is used to assign a sort to operators and their arguments. Here, this mapping is expressed by means of a non-empty list of sorts. The non-empty list is encoded as a pair (l, r) composed of a possibly empty list of sorts and a sort. Intuitively, the list l assigns a sort to each argument of an operator while the sort r denotes the sort of the operator itself. Note that the arity of the operator is now implicit and may be recovered by computing the length of the list.

An alternative representation of many-sorted signatures can be introduced as follows:

```
record ISig (I : Type)
  constructor _◁_
  [ Op : I → Type
    Typ : (i : I) → Op i → List I.
```

Here, the difference with the type MSig is that the type of operators is indexed over a base type I. Both representations, namely MSig and ISig, are in fact provably isomorphic but working with indexed signatures seems more appropriate. Indeed, in the context of inductive families, it is relatively straightforward to derive their corresponding indexed signatures, as illustrated in the following example.

Remark. When the indexed type I is defined to be the unit type, the type of signatures Sig is isomorphic to the type of indexed signatures ISig $\top$.

Example. In the previous section, we showed how the signature for polymorphic lists can be defined. Now, we extend the signature to lists with length constraints in order to represent the signature of *vectors*, *i.e.*, fixed-length lists. First, we recall the inductive definition of vectors:

```
inductive Vec (A : Type) : ℕ → Type
  | [] : Vec A zero
  | _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n).
```

The type Vec is indexed by a natural number accounting for the length of the list. Thus, the term [] constructs an empty vector, *i.e.*, a list of length equal to 0. The constructor _::_ extends a vector of length n with one element, yielding a vector of length $n+1$. The signature corresponding to the inductive definition of the type Vec can be defined as follows:

```
VecSig (A : Type) : ISig ℕ
VecSig A ≡ OpVec ◁ ArVec
    where inductive OpVec : ℕ → Type
              []   : OpVec zero
              cons : (a : A) → (n : ℕ) → OpVec(suc n)
          ArVec : {n : ℕ} → OpVec n → List ℕ
          ArVec ⌊zero⌋  []          ≡ []
          ArVec ⌊suc m⌋ (cons a m)  ≡ [m].
```

Here, we use the convenience of inductive families in order to define the set of operators OpVec. The index is a natural number accounting for the number of elements within the vector. Note that, it is the same type indexing Vec and its signature. This is in general the case when we want to derive the index signature from an inductive family definition. Unlike the definition of Vec, the type OpVec is not recursive. The function ArVec defines both the arity of operators and the type of their arguments. As such, the operator [] has no argument while the operator cons m has one argument whose type (sort) is given by m.

1.3.3. Term Algebra

Given a signature S, the type of all terms constructed from the operators of S is called the *term algebra induced by* S. It is inductively defined as follows:

```
inductive 𝕋_ (S : Sig) : Type
  | _◁_ : (o : S.Op) → (Fin(S.Ar o) → 𝕋_S) → 𝕋_S.
```

The term algebra $\mathbb{T}_S$ has only one constructor $_\triangleleft_$ with two parameters, namely an operator and a list of arguments. This list of arguments is modeled as a finite map. This representation has several benefits. First, the induced inductive principle is easily derived:

$$\frac{P : \mathbb{T}_S \to \textsc{Type} \qquad \forall (o : S.\mathsf{Op}).\ \forall (f : \textsc{Fin}(S.\mathsf{Ar}\, o) \to \mathbb{T}_S).\ (\forall k.\ P(f(k))) \to P(o \triangleleft f)}{\forall (t : \mathbb{T}_S).\ P(t)}.$$

Second, arguments are easily accessed through function application given an index of a subterm. However, the main drawback of the functional representation concerns equality between terms. The problem is that Leibniz equality does not satisfy functional extensionality. Thus, given two terms $o \triangleleft f$ and $o \triangleleft f'$, we cannot conclude, in general, that $o \triangleleft f = o \triangleleft f'$ when $\forall (k : \textsc{Fin}(S.\mathsf{Ar}\, o)).\ f(k) = f'(k)$.

Alternatively, arguments of operators can be represented as a vector. The choice of a vector instead of a list is motivated by the fact that we want to ensure—by construction—that the number of arguments matches the specification given by the arity function.

$$\textbf{inductive}\ \mathbb{T}_{_}\ (S : \textsc{Sig}) : \textsc{Type}$$
$$\mid\ _ \lhd _ : (o : S.\mathsf{Op}) \to \textsc{Vec}\ \mathbb{T}_S\ (S.\mathsf{Ar}\, o) \to \mathbb{T}_S.$$

This vector-based representation of arguments circumvents the main drawback of the functional representation in the sense that, extensionally equal vectors are provably Leibniz equal. However, the induction principle associated to this new representation is complicated by the fact that the type $\mathbb{T}_S$ is nested[1] with the type VEC:

$$\frac{\begin{array}{ll} P : \mathbb{T}_S \to \textsc{Type} & \forall (o : S.\mathsf{Op}).\ \forall (v : \textsc{Vec}\ \mathbb{T}_S\ (S.\mathsf{Ar}\, o)).\ Q(v) \to P(o \lhd v) \\ Q : \forall n.\ \textsc{Vec}\ \mathbb{T}_S\ n \to \textsc{Type} & Q([\,]) \quad \forall n.\ \forall (t : \mathbb{T}_S).\ \forall (v : \textsc{Vec}\ \mathbb{T}_S\ n).\ P(t) \to Q(v) \to Q(t :: v) \end{array}}{\forall (t : \mathbb{T}_S).\ P(t)}$$

With that induction principle, we can prove an alternative principle which is closer to the one induced by the functional representation:

$$\frac{P : \mathbb{T}_S \to \textsc{Type} \qquad \forall (o : S.\mathsf{Op}).\ \forall (v : \textsc{Vec}\ \mathbb{T}_S\ n).\ (\forall (k : \textsc{Fin}(S.\mathsf{Ar}\, o)).\ P(v(k))) \to P(o \lhd v)}{\forall (t : \mathbb{T}_S).\ P(t)}.$$

Free Monad. Given a signature $S : \textsc{Sig}$ and a type $X : \textsc{Type}$ denoting variables, we can extend the signature S by injecting variables into S as follows:

$$\langle _ \rangle_{_} : \textsc{Sig} \to \textsc{Type} \to \textsc{Sig}$$
$$\langle S \rangle_X \ \equiv\ S \ \langle \uplus \rangle\ \langle X \rangle^0.$$

where $\langle X \rangle^0$ denotes the nullary signature. The term algebra $\mathbb{T}_{\langle S \rangle_X}$ induced by the signature $\langle S \rangle_X$ is isomorphic to the following inductive definition:

$$\textbf{inductive}\ \mathbb{T}_S\ (X : \textsc{Type}) : \textsc{Type}$$
$$\mid\ \mathsf{var} : X \to \mathbb{T}_S(X)$$
$$\mid\ _ \lhd _ : (o : S.\mathsf{Op}) \to \textsc{Vec}\ (\mathbb{T}_S(X))\ (S.\mathsf{Ar}\, o) \to \mathbb{T}_S(X).$$

The type $\mathbb{T}_S(X)$ is called the *free monad* of S. Indeed, it carries the structure of monad. Concretely, we can define the parallel substitution operation on it:

$$_ \gg\!= _ : \mathbb{T}_S(X) \to (X \to \mathbb{T}_S(Y)) \to \mathbb{T}_S(Y)$$
$$(\mathsf{var}\, x) \gg\!= f \ \equiv\ f(x)$$
$$(o \lhd v) \gg\!= f \ \equiv\ o \lhd \mathsf{map}_{\textsc{Vec}}\ (_ \gg\!= f)\ v.$$

The unit operation of the monad is given by var. It is straightforward to check, by induction, that all laws of monads are satisfied by the parallel substitution function:

Left-identity. $\mathsf{var}\, x \gg\!= f = f(x)$,
Right-identity. $t \gg\!= \mathsf{var} = t$,
Associativity. $(t \gg\!= f) \gg\!= g = t \gg\!= (\lambda x.\ f(x) \gg\!= g)$.

[1] The induction principle generated by COQ does not account for the nesting of VEC with $\mathbb{T}_S$.

1.4. COALGEBRAS AND COINDUCTIVE TYPES

In this section, we review the relationship between coalgebraic structures and coinductive types. First, we recall some formal definitions about coalgebras in a type-theoretic context, that is, where definitions are expressed as types. To keep the presentation simple, we specialize the definitions to the category of types. Nevertheless, these definitions can be generalized to an arbitrary category. Finally, we show how coalgebras are used to give a semantics to coinductive types.

Definition 8 **Coalgebra**

An F-coalgebra for a functor $F : \textsc{Type} \to \textsc{Type}$ is a dependent pair:

$$\textsc{Coalg}(F) : \textsc{Type} := \sum_{X:\textsc{Type}} X \to F(X).$$

The first component X is called the carrier of the coalgebra while the second component $X \to F(X)$ is called the destructor. We denote by destr, *the second component of* $\textsc{Coalg}(F)$.

Definition 9 **Morphism of coalgebras**

Let $\mathcal{X} = (X, \mathsf{destr}_X)$ and $\mathcal{Y} = (Y, \mathsf{destr}_Y)$ be two F-coalgebra for a functor F. A morphism of coalgebras is given by a function $f : X \to Y$ on the underlying carriers such that the following diagram commutes:

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ \uparrow{\scriptstyle \mathsf{destr}_X} & & \uparrow{\scriptstyle \mathsf{destr}_Y} \\ X & \xrightarrow[f]{} & Y \end{array} \qquad \mathsf{destr}_Y \circ f = F(f) \circ \mathsf{destr}_X$$

The type of coalgebra morphisms is thus defined as a dependent pair where the first component is the function between the carriers and the second component is a coherence condition:

$$\mathcal{X} \to \mathcal{Y} := \{\, f : X \to Y \mid F(f) \circ \mathsf{destr}_X = \mathsf{destr}_Y \circ f \,\}.$$

Definition 10 **Final coalgebra**

A coalgebra $\mathcal{X}$ is said to be the final coalgebra when, for all coalgebras $\mathcal{Y}$, there exists a unique coalgebra morphism into $\mathcal{X}$. Formally:

$$\textsc{Final}(\mathcal{X}) := \forall (\mathcal{Y} : \textsc{Coalg}(F)).\ \exists!(f : \mathcal{X} \to \mathcal{Y}).\ \sum_{f:\mathcal{X}\to\mathcal{Y}} \forall f'.\ f = f'.$$

We say that $\mathcal{X}$ is weakly final when the morphism exists but is not necessarily unique.

1.4.1. Semantics of Coinductive Types

In the previous section, we introduced coalgebras in a category-theoretic setting. Now, we show how coinductive types can be semantically described as weakly final coalgebras.

Coinductive datatypes are introduced by the keyword **coinductive**. For example, given a signature S, the coterm algebra can be defined as a codatatype as follows:

```
coinductive coT_ (S : Sig) : Type
  | _◀_ : (o : S.Op) → (Fin(S. Ar o) → coT_S) → coT_S.
```

This type definition is very similar to its dual, the term algebra over S defined in Section 1.3.3. The main difference is that terms are allowed to be built by infinite applications of $_ \blacktriangleleft _$. Given a term $t : \text{co}\mathbb{T}_S$, we can observe its root function symbol and its subterms:

$$\begin{array}{ll} \mathsf{root} : \text{co}\mathbb{T} \to S & \mathsf{subterms} : (t : \text{co}\mathbb{T}_S) \to \text{FIN}(S.\mathsf{Ar}(\mathsf{root}\, t)) \to \text{co}\mathbb{T}_S \\ \mathsf{root}\ \ (o \blacktriangleleft _) \ \equiv\ o. & \mathsf{subterms}\ \ (o \blacktriangleleft os) \ \equiv\ os. \end{array}$$

Note that there is a dependence between the type of nodes and its arity. Consequently, the observation function subterms depends on root.

We can give to the type $\text{co}\mathbb{T}_S$ the structure of an S-coalgebra as follows:

$$\begin{array}{l} \mathsf{coalg}_{\text{co}\mathbb{T}} : \text{COALG}(S) \\ \mathsf{coalg}_{\text{co}\mathbb{T}} \ \equiv\ (\text{co}\mathbb{T}_S, \mathsf{destr}) \\ \quad \textbf{where}\ \ \mathsf{destr} : \text{co}\mathbb{T}_S \to S(\text{co}\mathbb{T}_S) \\ \qquad\qquad \mathsf{destr}\ \ t \ \equiv\ (\mathsf{root}\, t, \mathsf{tabulate}(\mathsf{subterms}\, t)). \end{array}$$

The destructor is given by the two observation functions namely root and subterms. Next, we prove that the type $\text{co}\mathbb{T}_S$ gives rise to a weakly-final coalgebra. To do that, we have to show that for any coalgebra (X, destr_X) there is an arrow into $\text{co}\mathbb{T}_S$:

$$\begin{array}{l} \mathsf{coiter} : (destr_X : X \to S(X)) \to X \to \text{co}\mathbb{T}_S \\ \mathsf{coiter}\ \ destr_X\ \ x \ \equiv\ o \blacktriangleleft \mathsf{coiter}\, destr_X \circ f \\ \quad \textbf{where}\ \ o : S.\mathsf{Op} \\ \qquad\qquad o \ \equiv\ \mathsf{proj}_1(destr_X(x)) \\ \qquad\qquad f : \text{FIN}(S.\mathsf{Ar}\, o) \to X \\ \qquad\qquad f \ \equiv\ \mathsf{lookup}(\mathsf{proj}_2(destr_X(x))). \end{array}$$

The function coiter is defined by *corecursion*. In the context of inductive types, recursive definitions are accepted provided that the recursive call is applied to a subterm of the inductive term. This ensures that the recursive function definition viewed as a rewrite system will eventually terminate.

By duality, for coinductive types, a corecursive call is accepted provided that it is guarded by a constructor—in this case, the constructor is $_ \blacktriangleleft _$. For the calculus of coinductive constructions, the rules specifying whether a corecursive call is in guarded form can be found in [Coq93, Gim96]. This condition ensures that constructors appearing on the left-hand side of equations cannot be consumed infinitely often without producing a value. Note that, for both recursive or corecursive definitions, the acceptance condition is based upon a *syntactic criterion*. Nonetheless, this criterion is not complete (see Section 1.1.2 for well-founded induction). Therefore, valid (semantically) (co)recursive definitions may fail to satisfy this syntactic criterion. For instance, the stream of natural numbers $\omega = 0, 1, 2, \ldots$ could be defined as follows:

$$\begin{array}{ll} \mathsf{map} : (A \to B) \to \text{STREAM}\, A \to \text{STREAM}\, B & \omega' : \text{STREAM}\, \mathbb{N} \\ \mathsf{map}\ \ f \ \equiv\ \mathsf{coiter}\langle f \circ \mathsf{head}, \mathsf{tail}\rangle. & \omega' \ \equiv\ 0 :: \mathsf{map}\ \mathsf{suc}\ \omega'. \end{array}$$

$$\begin{array}{ccc} B \times \text{STREAM}\, A & \xrightarrow{\text{STREAM}(\mathsf{coiter})} & B \times \text{STREAM}\, B \\ \uparrow{\scriptstyle \langle f \circ \mathsf{head}\rangle} & & \uparrow{\scriptstyle \langle \mathsf{head}, \mathsf{tail}\rangle} \\ \text{STREAM}\, A & \xrightarrow[\mathsf{coiter}]{} & \text{STREAM}\, B \end{array}$$

Here, the function map is defined by *coiteration*. Since the type of streams is a weakly final coalgebra, in order to produce a map $\text{STREAM}\, A \to \text{STREAM}\, B$, it suffises to give to the type $\text{STREAM}\, A$ the structure of a $\text{STREAM}\, B$ coalgebra. However, the problem with the corecursive

definition of ω' is that it violates the guard condition. Indeed, the corecursive call occurs under a function which is not a constructor. Nevertheless, this definition is productive. This fact can be shown by considering another definition of ω, this time syntactically guarded:

$$\begin{array}{ll} \omega_ : \mathbb{N} \to \textsc{Stream}\ \mathbb{N} & \qquad\qquad \omega : \textsc{Stream}\ \mathbb{N} \\ \omega_n \ \equiv\ n :: \omega_{\mathsf{suc}\, n}. & \qquad\qquad \omega \ \equiv\ \omega_0. \end{array}$$

Then, we can show that the following laws are satisfied—up-to bisimilarity—by this definition:

$$\mathsf{head}\, w \equiv 0, \qquad\qquad \mathsf{tail}\, \omega \sim \mathsf{map}\ \mathsf{suc}\ \omega.$$

This example illustrates the incompleteness of the syntactic criterion for corecursive definitions.

Consequently, it is sometimes convenient to rely on "more primitive" notions to define functions. We can use (dependent) iterators or well-founded induction for inductive types and coiterators for coinductive types. Various techniques have been proposed to ease the production of corecursive definitions [BPT15, AM13] or to circumvent the guard condition [EHB13, Dan10].

1.4.2. Equality and Coinductive Types

So far, we showed that the type $\text{co}\mathbb{T}_S$ is weakly final. However, it is interesting to discuss why it fails to be the final coalgebra. This is mainly due to the fact that Leibniz equality fails to identify extensionally equal functions. In order to be the final coalgebra, equality between inhabitants of the type $\text{co}\mathbb{T}_S$ has to satisfy an extensionality principle known as *bisimilarity*. Before defining formally bisimilarity, we recall the definition of a bisimulation relation expressed coalgebraically.

Bisimulation. Let $(\mathcal{X}, destr_{\mathcal{X}})$ be an S-coalgebra for a signature S and let $\mathcal{R} : \mathcal{X} \to \mathcal{X} \to \textsc{Prop}$ be a binary relation. Define the sigma-closure of $\mathcal{R}$ to be

$$\overline{\mathcal{R}} := \exists x.\ \exists y.\ x\, \mathcal{R}\, y,$$

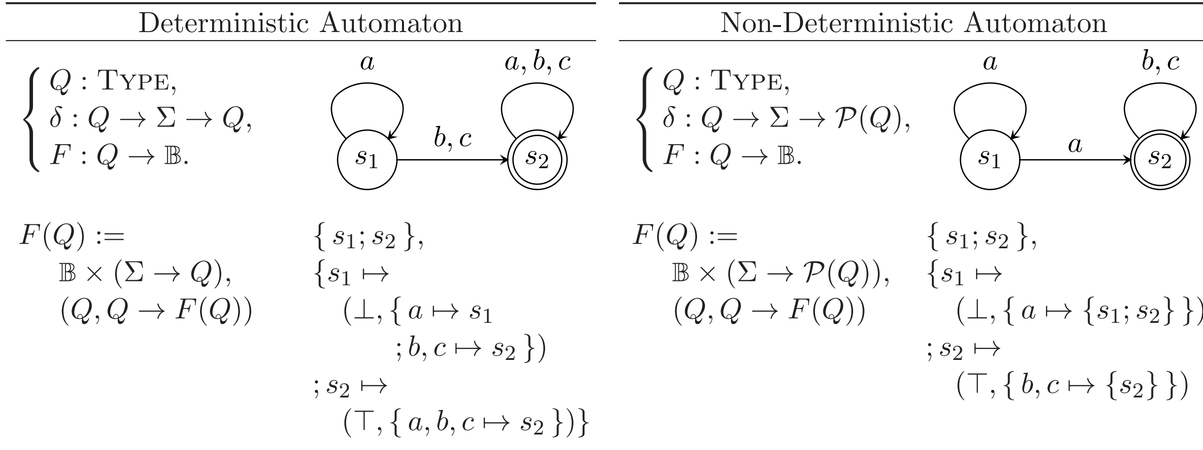
along with two projections $\pi_1^{\overline{\mathcal{R}}}(x, y, r) := x$ and $\pi_2^{\overline{\mathcal{R}}}(x, y, r) := y$. An *$S$-bisimulation* is a relation $\mathcal{R}$ together with a map $destr_{\mathcal{R}} : \overline{\mathcal{R}} \to S(\overline{\mathcal{R}})$ such that both $\pi_1^{\overline{\mathcal{R}}}$ and $\pi_2^{\overline{\mathcal{R}}}$ are S-coalgebra morphisms:

$$\begin{array}{ccccc} S(\mathcal{X}) & \xleftarrow{S\left(\pi_1^{\overline{\mathcal{R}}}\right)} & S(\overline{\mathcal{R}}) & \xrightarrow{S\left(\pi_2^{\overline{\mathcal{R}}}\right)} & S(\mathcal{X}) \\ \uparrow{\scriptstyle destr_{\mathcal{X}}} & & \uparrow{\scriptstyle destr_{\mathcal{R}}} & & \uparrow{\scriptstyle destr_{\mathcal{X}}} \\ \mathcal{X} & \xleftarrow[\pi_1^{\overline{\mathcal{R}}}]{} & \overline{\mathcal{R}} & \xrightarrow[\pi_2^{\overline{\mathcal{R}}}]{} & \mathcal{X} \end{array}$$

That is, such that the following equations are satisfied:

$$S\left(\pi_i^{\overline{\mathcal{R}}}\right) \circ destr_{\mathcal{R}} = destr_{\mathcal{X}} \circ \pi_i^{\overline{\mathcal{R}}}, \quad i \in \{1, 2\}$$

We say that a bisimulation relation is an *equivalence relation* when the underlying relation is an equivalence relation.



Deterministic Top-Down Tree Automaton

$$\begin{cases} Q : \textsc{Type}, \\ \Delta : Q \to (\sigma : \Sigma.\mathsf{Op}) \to \textsc{Vec}\ Q\ (\Sigma.\,\mathsf{Ar}\,\sigma), \\ F : Q \to \mathbb{B}. \end{cases}$$

$$\begin{aligned} q_1(f(x_1, x_2)) &\to f(q_1(x_1), q_2(x_2)), \\ q_1(g(x)) &\to g(q_2(x)), \\ q_2(f(x_1, x_2)) &\to f(q_1(x_1), q_1(x_2)), \\ q_2(g(x)) &\to g(q_2(x)). \end{aligned}$$

$$\begin{aligned} F(Q) &:= \\ &\mathbb{B} \times ((\sigma : \Sigma) \to \textsc{Vec}\ Q\ (S.\,\mathsf{Ar}\,\sigma)), \\ &(Q, Q \to F(Q)) \end{aligned}$$

$$\begin{aligned} &\{\, q_1; q_2 \,\}, \\ &\quad \{q_1 \mapsto \bot, \{f \mapsto [q_1, q_2] \\ &\qquad\qquad ;\ g \mapsto [q_2]\} \\ &\quad ; q_2 \mapsto \top, \{f \mapsto [q_1, q_1] \\ &\qquad\qquad ;\ g \mapsto [q_2]\}\} \end{aligned}$$

Figure 1.2. Transition systems along with their respective functors viewed as coalgebras

In the context of labeled transition systems $(S, \mathcal{A}, \longrightarrow)$, a bisimulation relation is defined to be a relation $\mathcal{R}$ over states such that:

$$s\,\mathcal{R}\,s' \Rightarrow \begin{cases} \forall s \xrightarrow{a} t.\ \exists s' \xrightarrow{a} t'.\ t\,\mathcal{R}\,t', \\ \forall s' \xrightarrow{a} t'.\ \exists s \xrightarrow{a} t.\ t\,\mathcal{R}\,t'. \end{cases}$$

Note that, when such transitions systems are represented as coalgebras (see Figure 1.2), the bisimulation relation definition coincides with the categorical one.

For example, when the functor S is the functor of streams, we obtain the classical definition of bisimulation given by:

$$\frac{\mathsf{head}\ s_1 = \mathsf{head}\ s_2 \qquad \mathsf{tail}\ s_1\ \mathcal{R}\ \mathsf{tail}\ s_2}{s_1\ \mathcal{R}\ s_2}.$$

Bisimilarity. The *bisimilarity* relation on the term coalgebra $\text{co}\mathbb{T}_S$ for a given signature S is defined coinductively as follows:

```
coinductive _~_ : coT_S → coT_S → Prop
  | ~-intro : (e : root t₁ = root t₂) → (∀k. subterms t₁ k ~ subterms t₂ e_*(k)) → t₁ ~ t₂.
```

One remarkable property of bisimilarity is that it is the greatest bisimulation. This means that that for any S-bisimulation $\mathcal{R}$, we have $\mathcal{R} \subseteq _\!\sim\!_$, *i.e.*, $\forall x.\ \forall y.\ x\,\mathcal{R}\,y \to x \sim y$. The proof of that statement is straightforward (by corecursion) and follows from these two simple facts:

$$\frac{x\,\mathcal{R}\,y}{e : \mathsf{root}\,x = \mathsf{root}\,y}, \qquad \frac{x\,\mathcal{R}\,y \qquad k : \textsc{Fin}(S.\,\mathsf{Ar}(\mathsf{root}\,x))}{(\mathsf{subterms}\ x\ k)\ \mathcal{R}\ (\mathsf{subterms}\ y\ e_*(k))},$$

which are direct consequences of the definition of bisimulation.

Coinduction proof principle. The *coinduction proof principle* provides a way to prove equality between two terms t_1 and t_2 by giving a bisimulation relation $\mathcal{R}$ such that both t_1 and t_2 are related by $\mathcal{R}$. Formally,

$$\forall(\mathcal{R} : \text{co}\mathbb{T}_S \to \text{co}\mathbb{T}_S \to \textsc{Prop}).\ \textsc{bisim}(\mathcal{R}) \to \forall(t_1, t_2 : \text{co}\mathbb{T}_S).\ t_1\,\mathcal{R}\,t_2 \to t_1 = t_2.$$

Here, BISIM is a predicate on binary relations that characterizes bisimulations. However, the coinduction proof principle is not available for the coinductive types of Coq.

1.5. CONCLUSION

In this chapter, we have introduced the type theory—calculus of coinductive constructions—used to formalize all results contained in this thesis. In particular, we have presented inductive and coinductive type definitions along with their respective reasoning principles. We have also discussed how quotient of types can be approximated through setoids. This is particularly important in the context of coinductive types since they fail to satisfy the coinductive proof principle. Consequently, most of the results based on coinductive type definitions have to be dealt with modulo bisimilarity. Furthermore, we introduced signatures and indexed signatures as a tool to reason about strictly positive (co)inductive types. Finally, we have presented a hierarchy of finite setoid definitions and studied their relationships.

CHAPTER TWO

REGULAR TREES

In this chapter, we introduce formally the type of regular trees [Cou83] over an arbitrary signature. First, we give a characterization of regular trees based on a restriction of a coinductive representation of trees. Intuitively, this restriction is expressed as a property upon the set of subtrees of an infinite tree, namely that this type is finite. To this end, we reuse a lot of the material about finite types introduced in the previous chapter. Moreover, we aim to define a notion of subtrees that is as independent as possible of a particular tree representation. Although we use a weak notion of finiteness[1] to characterize the set of subtrees, we show that it is still sufficient to obtain, constructively, a closure property: a subtree of a regular tree is again regular.

Next, we introduce a syntax aiming to characterize all regular trees. This syntax is based on an inductive representation of infinite trees as cyclic terms [GHUV06]. Here, cycles within terms are encoded by means of binders. Contrary to the coinductive characterization, cyclic terms allow cycles to be represented explicitly. The main advantage of a syntax-based approach is that trees obtained through the syntax are regular *by construction*. Finally, we show that this syntax is sound and complete with respects to the coinductive characterization of regular trees.

2.1. A COINDUCTIVE CHARACTERIZATION

In this section, we aim to give a coinductive characterization of regular trees [Cou83]. Informally, a potentially infinite tree is said to be regular whenever the set of its subtrees is finite—up-to tree isomorphism. In order to formally define this property, we proceed as follows.

First, we introduce, as a coinductive datatype definition, the type of infinite trees over a signature. Then, we discuss the notion of tree isomorphism and introduce the bisimulation-based proof method. In particular, we show that the binary relation defined to be the chain of approximations of tree isomorphisms—up-to a given depth—is a bisimulation.

Next, we define the set of subtrees of an infinite tree through *unranked coalgebras*. These coalgebras are used to abstract away from a particular representation of infinite trees. To this end, we work on coalgebras induced by a specific signature, namely the *unranked signature*. Intuitively, this signature allows nodes of a tree to be represented by its arity. In addition, the carrier of these coalgebras are taken to be setoids rather than types. Starting from a given coalgebra viewed as a transition system, we define the notions of paths and successors. Then, this abstract framework is instantiated on the type of infinite trees yielding a formal definition of regular trees.

Finally, we prove a closure property on regular trees, namely that a subtree of a regular tree is again regular. In particular, we show that this result is not trivial because of the relatively weak definition of finiteness used in the characterization of the regularity property.

[1] For instance, we do not assume decidability of equality of the set of operators of the signature.

2.1.1. Infinite Trees

In the previous chapter, we introduced the coterm algebra induced by a signature. From this type definition, we can derive a type characterizing infinite trees. Here, we recall briefly its definition slightly adapted to the tree terminology (as opposed to the term terminology used previously).

Definition 1 **Type of infinite trees**

Let S : SIG be a signature. The type of infinite trees over the signature S, denoted $\text{co}\mathbb{T}_S$, is defined coinductively as follows:

$$\textbf{coinductive } \text{co}\mathbb{T}\ (S : \text{SIG}) : \text{TYPE}$$
$$\mid\ _ \blacktriangleleft _ : (o : S.\mathsf{Op}) \to (\text{FIN}(S.\mathsf{Ar}\, o) \to \text{co}\mathbb{T}_S) \to \text{co}\mathbb{T}_S.$$

Given an infinite tree, we may observe its root:

$$\mathsf{root} : \{S : \text{SIG}\} \to \text{co}\mathbb{T}_S \to S.\mathsf{Op}$$
$$\mathsf{root}\ \ (o \blacktriangleleft _)\ \equiv\ o$$

and define a function indexing its immediate subtrees:

$$\mathsf{br} : \{S : \text{SIG}\} \to (t : \text{co}\mathbb{T}_S) \to \text{FIN}(S.\mathsf{Ar}(\mathsf{root}\, t)) \to \text{co}\mathbb{T}_S$$
$$\mathsf{br}\ \ (_ \blacktriangleleft b)\ \equiv\ b.$$
$$\textbf{coercion } \mathsf{br} : \text{co}\mathbb{T} \rightarrowtail \text{FUNCLASS}.$$

The map br is used as a coercion from infinite trees to functions. Consequently, given an index k of a subtree of t, we may obtain the subtree at position k by writing $t(k)$ instead of br t k.

Equality between infinite trees

As Leibniz equality fails to identify all isomorphic trees, we define a coinductive extensional equality on trees, known as bisimilarity:

$$\textbf{coinductive } _\sim_\ \{S : \text{SIG}\} : \text{co}\mathbb{T}_S \to \text{co}\mathbb{T}_S \to \text{PROP}$$
$$\mid\ \sim\text{-}\mathsf{intro} : \forall\{t_1, t_2\}.\ (e : \mathsf{root}\, t_1 = \mathsf{root}\, t_2) \to (\forall k.\ t_1(k) \sim t_2(e_*(k))) \to t_1 \sim t_2.$$

When two trees t_1 and t_2 are related by $\sim$ we say that they are (tree) isomorphic or bisimilar. Here, because of the dependence between an operator and its arity we have to transport the equality proof when the kth subtree of t_2 is accessed. Nevertheless, we can prove a strong elimination principle on bisimilarity, namely that we may use *any* proof of equality between roots:

Proposition 1 **Elimination principle on $_\sim_$**

Let S be a signature and t_1, t_2 be two infinite trees over S. If $t_1 \sim t_2$ then, for any equality proof e : $\mathsf{root}\, t_1 = \mathsf{root}\, t_2$ and index k : $\text{FIN}(S.\mathsf{Ar}(\mathsf{root}\, t_1))$, we have $t_1(k) \sim t_2(e_*(k))$.

Proof. Let S : SIG, $t_1, t_2 : \text{co}\mathbb{T}_S$ and $B : t_1 \sim t_2$ a proof that t_1 is bisimilar to t_2. Moreover, let $e : \mathsf{root}\, t_1 = \mathsf{root}\, t_2$ be an equality proof and $k : \text{FIN}(S.\mathsf{Ar}(\mathsf{root}\, t_1))$ be an index of an immediate subtree of t_1. By case-analysis on B, there exists an equality $e' : \mathsf{root}\, t_1 = \mathsf{root}\, t_2$. Furthermore, there exists a proof $H : \forall k.\ t_1(k) \sim t_2(e'_*(k))$. By equational reasoning:

$$
\begin{aligned}
t_1(k) \sim\ & t_2(e'_*(k)) && \text{(by hypothesis } H(k))\\
\equiv\ & t_2(\mathsf{subst}\ (\text{FIN} \circ S.\mathsf{Ar})\ e'\ k) && \text{(by definition of } _{}_*)\\
=\ & t_2(\mathsf{subst}\ \text{FIN}\ (\mathsf{ap}\ S.\mathsf{Ar}\ e')\ k) && \text{(by Prop. A.1 [}\mathsf{subst}\ (P \circ f)\ p \doteq \mathsf{subst}\ P\ (\mathsf{ap}\ f\ p)])\\
=\ & t_2(\mathsf{subst}\ \text{FIN}\ (\mathsf{ap}\ S.\mathsf{Ar}\ e)\ k) && \text{(by UIP on } \mathbb{N},\ \mathsf{ap}\ S.\mathsf{Ar}\ e' = \mathsf{ap}\ S.\mathsf{Ar}\ e)\\
=\ & t_2(\mathsf{subst}\ (\text{FIN} \circ S.\mathsf{Ar})\ e\ k) && \text{(by Prop. A.1 [}\mathsf{subst}\ (P \circ f)\ p \doteq \mathsf{subst}\ P\ (\mathsf{ap}\ f\ p)])\\
\equiv\ & t_2(e_*(k)). && \text{(by definition of } _{}_*)
\end{aligned}
$$

□

Remark. Prop. 2.1 is important since it allows *any proof of equality* to be used regardless of the one used in the initial proof. However, this does not entail that $_\sim_$ is a mere proposition.

In the following, we show that bisimilarity is the largest bisimulation relation. A bisimulation relation on infinite trees is defined as follows:

Definition 2 Bisimulation relation on infinite trees

Let S be a signature. A binary relation $_\mathcal{R}_ : \text{co}\mathbb{T}_S \to \text{co}\mathbb{T}_S \to \text{PROP}$ is said to be a bisimulation relation when it satisfies:

(i) $\forall (t_1, t_2 : \text{co}\mathbb{T}_S).\ t_1\,\mathcal{R}\,t_2 \to \mathsf{root}\,t_1 = \mathsf{root}\,t_2$,
(ii) $\forall (t_1, t_2 : \text{co}\mathbb{T}_S).\ \forall (e : \mathsf{root}\,t_1 = \mathsf{root}\,t_2).\ \forall (k : \text{FIN}(S.\mathsf{Ar}(\mathsf{root}\,t_1))).\ t_1(k)\,\mathcal{R}\,t_2(e_*(k))$.

We denote by $\text{BISIM}(\mathcal{R})$, the type of binary relation satisfying $(i\!-\!ii)$.

Remark. As for the definition of bisimilarity, there should be a dependence between the conditions (i) and (ii). Indeed, the definition should have been written as follows:

$$\forall (t_1, t_2 : \text{co}\mathbb{T}_S).\ t_1\,\mathcal{R}\,t_2 \to \exists (e : \mathsf{root}\,t_1 = \mathsf{root}\,t_2).\ \forall k.\ t_1(k)\,\mathcal{R}\,t_2(e_*(k)). \qquad (\mathsf{bisimulation\text{-}alt})$$

However, in Proposition 2.1 [$\sim$ elim.], we observed that the choice of the equality proof is not important. This justifies why we used universal quantification in (ii). Finally, it is possible to prove (as in Proposition 2.1) that Definition 2.2 and (bisimulation-alt) are in fact equivalent.

Theorem 1 Coinduction proof principle

Let S be a signature. The bisimilarity relation $_\sim_S_$ is the largest bisimulation relation.

Proof. Let $S : \text{SIG}$ be a signature and $\mathcal{R} : \text{co}\mathbb{T}_S \to \text{co}\mathbb{T}_S \to \text{PROP}$ be a binary relation. Moreover, assume that there is a proof $B : \text{BISIM}(\mathcal{R})$. We have to show that:

$$\forall (t_1, t_2 : \text{co}\mathbb{T}_S).\ t_1\,\mathcal{R}\,t_2 \to t_1 \sim t_2. \qquad (\text{cofix})$$

We prove it by corecursion, thus we may assume (cofix). Let t_1, t_2 be two infinite trees and let $r : t_1\,\mathcal{R}\,t_2$. From B and by definition of BISIM, there exists a proof e such that $\mathsf{root}\,t_1 = \mathsf{root}\,t_2$. Furthermore, there exists a proof B' such that $\forall e.\ \forall k.\ t_1(k)\,\mathcal{R}\,t_2(e_*(k))$. Consequently, to prove that $t_1 \sim t_2$, we use the constructor $\sim$-intro. It remains to show that:

- $\mathsf{root}\,t_1 = \mathsf{root}\,t_2$, which is given by hypothesis e,
- $\forall k.\ t_1(k) \sim t_2(e_*(k))$, which is proven by applying $B'\ e\ k : t_1(k)\,\mathcal{R}\,t_2(e_*(k))$ to (cofix). □

We conclude this section by defining the bisimilarity relation up-to a given depth. Then, we show that this binary relation is a bisimulation. Consequently, it may be used to show that two infinite trees are bisimilar (Theorem 2.1). This bisimulation relation is important because it allows bisimilarity proofs to be produced by induction on the depth, free of guard conditions.

Definition 3 Bisimilarity up-to a given depth

Let S be a signature. The binary relation $_\sim_n_ : \text{co}\mathbb{T}_S \to \text{co}\mathbb{T}_S \to \text{PROP}$, denoting bisimilarity up-to depth n, is defined inductively as follows:

inductive $_\sim__\ \{S : \text{SIG}\} : \text{co}\mathbb{T}_S \to \mathbb{N} \to \text{co}\mathbb{T}_S \to \text{PROP}$
  $\sim_0$-intro : $\forall \{t_1, t_2\}.\ t_1 \sim_0 t_2$
  $\sim_{\mathsf{suc}}$-intro : $\forall \{t_1, t_2\}.\ \forall \{n\}.\ \forall (e : \mathsf{root}\,t_1 = \mathsf{root}\,t_2).\ (\forall k.\ t_1(k) \sim_n t_2(e_*(k))) \to t_1 \sim_{\mathsf{suc}\,n} t_2$.

Now, we prove that the bisimilarity up-to a given depth induces a chain whose limit coincides with the bisimilarity relation.

Theorem 2 **Bisimilarity up-to a given depth induces a bisimulation relation**

Let S be a signature. The binary relation $\mathcal{D}$ defined as:

$$t_1 \,\mathcal{D}\, t_2 \overset{\text{def}}{\iff} \forall(n : \mathbb{N}).\ t_1 \sim_n t_2,$$

is a bisimulation relation.

Proof. Let S be a signature and $t_1, t_2 : \mathrm{co}\mathbb{T}_S$ be two trees. Assume that $d : t_1 \,\mathcal{D}\, t_2$, *i.e.*, that we have $d : \forall n.\ t_1 \sim_n t_2$. In order to show that $\mathcal{D}$ is a bisimulation, we prove (*i*–*ii*) of Definition 2.2:

(*i*) $\mathsf{root}\, t_1 = \mathsf{root}\, t_2$.
By case-analysis on $d(1) : t_1 \sim_1 t_2$, we obtain a proof $e : \mathsf{root}\, t_1 = \mathsf{root}\, t_2$.

(*ii*) $\forall(e : \mathsf{root}\, t_1 = \mathsf{root}\, t_2).\ \forall(k : \mathrm{FIN}(S.\,\mathsf{Ar}(\mathsf{root}\, t_1))).\ \forall(n : \mathbb{N}).\ t_1(k) \sim_n t_2(e_*(k))$.
Let $e : \mathsf{root}\, t_1 = \mathsf{root}\, t_2$ be an equality proof, k be an index and n be a natural number. By case-analysis on $d(\mathsf{suc}\, n) : t_1 \sim_{\mathsf{suc}\, n} t_2$, there exists an equality proof $e' : \mathsf{root}\, t_1 = \mathsf{root}\, t_2$ along with a proof H such that $\forall k.\ t_1(k) \sim_n t_2(e'_*(k))$. By the same arguments used in Prop. 2.1, we can substitute e' by e in H, thus we have that $t_1(k) \sim_n t_1(e_*(k))$ by $H(k)$. □

Remark. We could also prove the other direction, namely that the bisimilarity relation is included in $\mathcal{D}$. This proof is immediate by induction on the depth. Intuitively, when two infinite trees are bisimilar, they are bisimilar up-to *any* arbitrary depth.

An important consequence of Theorem 2.2 is that proofs of bisimilarity between infinite trees can be obtained by induction rather than by coinduction. Indeed, one of the problem of coinductively-defined proofs is the syntactic guard condition. This restriction can hinder compositional reasoning. For example, the use of transitivity in coinductive proofs can lead to syntactically non guarded corecursive calls. However, when the proof is obtained by induction on the depth of trees, we are freed from these syntactic considerations and thus, we may recover some forms of compositionality.

2.1.2. Unranked Coalgebras

In this section, we define the type of unranked coalgebras induced by the unranked signature. Here, the goal is to obtain an abstract structure used to define the notion of paths and successors independently of a particular infinite tree representation. Note that, carriers of coalgebras are considered here to be setoids rather than plain types.

Definition 4 **Unranked signature**

The unranked signature, denoted U, is defined as:

$$\mathsf{U} := \mathbb{N} \lhd \mathsf{id}.$$

Remark. The unranked signature is introduced here as a simplification. Without it, every subsequent result of this section would have to be polymorphic over signatures.

As shown in Section 1.3.1, a signature induces a functor on TYPE called the extension function. In the following, we show that this functor can be extended to setoids.

Proposition 2 **Induced functor on setoids**

The unranked signature U induces an endofunctor on setoids denoted F_{U}.

Proof. The action on setoids is defined as a dependent pair:

$$F_{\mathsf{U}}(X, \approx_X) := \left(\sum_{n:\mathbb{N}} \textsc{Vec}\ X\ n \right), \approx$$

where the binary relation $\approx$ is given by:

$$(n, v) \approx (n', v') \overset{\text{def}}{\iff} \exists (e : n = n').\ \forall (k : \textsc{Fin}\, n).\ v(k) \approx_X v'(e_*(k)).$$

It is straightforward to check that $\approx$ is an equivalence relation. Note that, the type $\sum_{n:\mathbb{N}} \textsc{Vec}\ X\ n$ is isomorphic to the type of lists over X.

Given two setoids $(X, \approx_X), (Y, \approx_Y)$ and a setoid morphism $h : X \longrightarrow Y$, the action on h is given by:

$$\mathsf{map}^{F_{\mathsf{U}}}\ h\ (n, v) := (n, \mathsf{map}^{\textsc{Vec}}\ h\ v).$$

Finally, it remains to show that the function $\mathsf{map}^{F_{\mathsf{U}}}$ respects the underlying setoid equivalences:

$$\forall (n, v).\ \forall (n', v').\ (n, v) \approx (n', v') \to \mathsf{map}^{F_{\mathsf{U}}}\ h\ (n, v) \approx \mathsf{map}^{F_{\mathsf{U}}}\ h\ (n', v').$$

Let $(n, v), (n', v') : F_{\mathsf{U}}(X, \approx_X)$ such that $(n, v) \approx (n', v')$. By definition of $\approx$, there exists a proof $e : n = n'$ such that $\forall k.\ v(k) \approx_X v'(e_*(k))$. By induction on e, we can substitute n' by n and e by refl_n. Consequently, we have $\forall k.\ v(k) \approx_X v'(\mathsf{refl}_*(k)) = v'(k)$. Moreover, since h is a setoid morphism and that $\forall v.\ \forall k.\ (\mathsf{map}^{\textsc{Vec}}\ h\ v)(k) = h(v(k))$ (Prop. A.11), we obtain that $\forall k.\ (\mathsf{map}^{\textsc{Vec}}\ h\ v)(k) = h(v(k)) \approx_Y h(v'(k)) = (\mathsf{map}^{\textsc{Vec}}\ h\ v')(k)$. Thus, we have proved that both vectors, namely $\mathsf{map}^{F_{\mathsf{U}}}\ h\ (n, v)$ and $\mathsf{map}^{F_{\mathsf{U}}}\ h\ (n, v')$, are extensionally equivalent. By Proposition A.10, this is a sufficient condition to show that they are equal. ☐

Definition 5 **Type of U-coalgebra**

The type of U*-coalgebra, denoted* $\mathcal{U}$*, is defined as a dependent pair:*

$$\mathcal{U} := \sum_{(X, \approx)} X \longrightarrow F_{\mathsf{U}}(X).$$

The two projections function of a $\mathcal{U}$*-coalgebra, denoted* rank *and* next*, are defined as follows:*

$$\mathsf{rank} : \forall \{X : \mathcal{U}\}.\ X \to \mathbb{N} \qquad \mathsf{next} : \forall \{X : \mathcal{U}\}.\ \forall (s : X).\ \textsc{Fin}(\mathsf{rank}\, s) \to X$$

$$\mathsf{rank}\ \{X, out_X\} \equiv \mathsf{proj}_1 \circ out_X. \qquad \mathsf{next}\ \{X, out_X\} \equiv \mathsf{proj}_2 \circ out_X.$$

Remark. Note that, given a coalgebra (X, out_X), both projection functions, namely rank and next, preserve setoid equivalences. Indeed, since out_X is a setoid morphism, it is straightforward to prove that the following congruence laws hold:

(*i*) $\forall (s, s' : X).\ s \approx_X s' \to \mathsf{rank}\, s = \mathsf{rank}\, s'$,

(*ii*) $\forall s, s'.\ \forall (k : \textsc{Fin}(\mathsf{rank}\, s)).\ \forall (k' : \textsc{Fin}(\mathsf{rank}\, s')).\ s \approx_X s' \to k \simeq k' \to \mathsf{next}\ s\ k \approx_X \mathsf{next}\ s'\ k.$

Here, $k \simeq k'$ denotes an heterogeneous equality over the type of finite sets (see Definition A.24).

Terminology. Let $T : \mathcal{U}$ be a coalgebra. We may refer to T as a *transition system* (TS), where the carrier of T is called the *type of states* and next is the *transition function*. We say that s' is an *immediate successor state* of s when there exists an index $k : \textsc{Fin}(\mathsf{rank}\, s)$ such that $s' = \mathsf{next}\ s\ k$.

2.1.3. Paths and Successors in Unranked Coalgebras

For the remaining of this section, we fix a transition system $T : \mathcal{U}$. We call S the type of states of T. In addition, for a given state $s : S$ and index $k : \textsc{Fin}(\mathsf{rank}\, s)$, we abbreviate $\mathsf{next}\ s\ k$ as $s[k]$.

Paths

A path within a transition system could be represented as a list of natural numbers. To account for the dependency between a state and its rank, we represent elements of a path as a dependent pair (n, k) with n being the rank.

Definition 6 **Path**

The type of paths is defined as a list of pairs:

$\textsc{Path} : \textsc{Type}$

$\textsc{Path} \;\equiv\; \textsc{List}\,(\sum_{n:\mathbb{N}} \textsc{Fin}\, n)$

where the first element of the pair is a natural number $n : \mathbb{N}$ and the second element is an index $k : \textsc{Fin}\, n$ taken from the finite set induced by n. Moreover, we denote by $\varepsilon : \textsc{Path}$, the empty path, i.e., the path defined as $\varepsilon := [\,]$.

However, not all paths are valid for a given transition system. Indeed, we have to ensure that a path can be effectively "mapped" onto it. Figure 2.1 contains an example of a valid (resp. invalid) path in the transition system T starting from the state s_1. The problem with the invalid path $p = [(2, 1); (3, 2); (4, 3)]$ is the last element, namely $(4, 3)$. Here, the rank of the state s_3 is $4 \nleq 3$. Consequently, there is no out-going transition in T that can be indexed by 3.

Definition 7 **Set of s-paths**

Let $s : S$ be a state. The set of s-paths, denoted $\mathscr{P}(s)$, is defined inductively as follows:

$$\mathscr{P}_\varepsilon\ \frac{}{\varepsilon \in \mathscr{P}(s)}, \qquad \mathscr{P}_{::}\ \frac{e : n = \mathsf{rank}\, s \qquad p \in \mathscr{P}(s[e_*(k)])}{(n, k) :: p \in \mathscr{P}(s)}.$$

Moreover, the type of all s-paths, denoted $\mathbb{P}(s)$, is defined as follows:

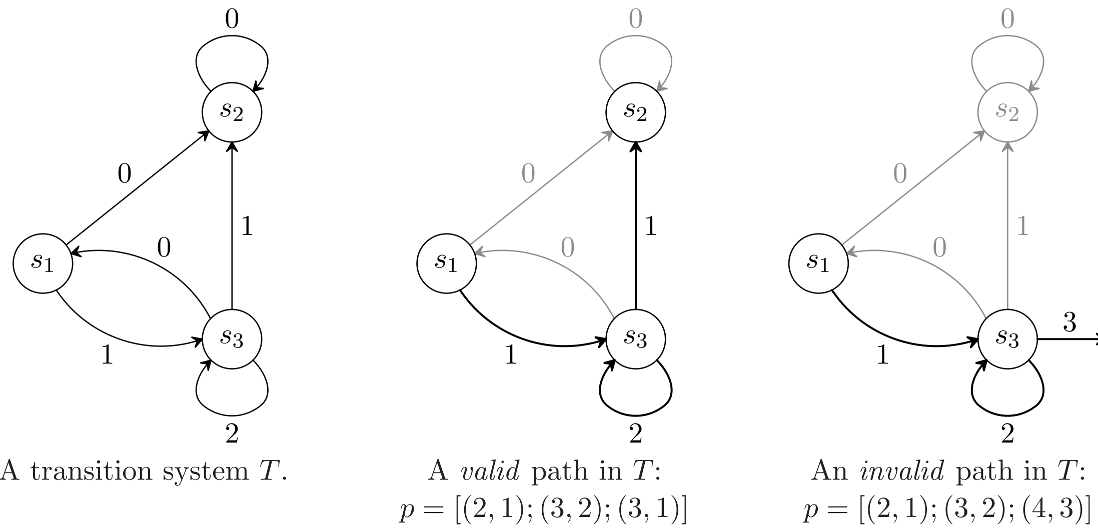
$$\mathbb{P}(s) := \{\, p : \textsc{Path} \mid p \in \mathscr{P}(s)\,\}\,.$$

Notation. Here, we use the membership notation with predicates which can be thought of an encoding of sets in type theory. Thus, $a \in P$ is a notation for $P(a)$, for $P : A \to \textsc{Prop}$.

Remark. The specification of $\mathscr{P}$ could be generalized further, the drawback being that it adds a slightly stronger dependency between the rank of a state and the indices of its out-going transitions. For instance, consider the valid path $p = [(2, 1); (3, 2); (3, 1)]$ of Figure 2.1. This path could have been also expressed as $p' = [(2, 1); (3, 2); (2, 1)]$. The difference concerns the last element $(2, 1)$. With this new definition, we have a weaker specification of the rank of s_3. Nevertheless, it is possible to lift the index 1 of type $\textsc{Fin}\, 2$ to the type $\textsc{Fin}\, 3$. Consequently, an alternative definition of the rule $\mathscr{P}_{::}$ could be given as follows:

$$\mathscr{P}_{::}\ \frac{e : n \leq \mathsf{rank}\, s \qquad p \in \mathscr{P}(s[\mathsf{lift}_e(k)])}{(n, k) :: p \in \mathscr{P}(s)}$$

where the type of the function lift is given by: $\forall (m, n : \mathbb{N}).\ \textsc{Fin}\, m \to m \leq n \to \textsc{Fin}\, n$.



A transition system T.

A *valid* path in T: $p = [(2,1);(3,2);(3,1)]$

An *invalid* path in T: $p = [(2,1);(3,2);(4,3)]$

Figure 2.1. Examples of a valid/invalid paths within a transition system.

Next, we prove two properties on the type of s-paths, namely that it is a mere proposition and that path membership is decidable, regardless of whether the underlying setoid equality on states is decidable or not.

Proposition 3 $\mathscr{P}(s)$ **is a mere proposition**

Let $s : S$ be a state and p be a path. Then, the type $p \in \mathscr{P}(s)$ is a mere proposition.

Proof. Let $p : \textsc{Path}$ and $s : S$. To show that the type $p \in \mathscr{P}(s)$ is a mere proposition, we have to prove that:

$$\forall (P, Q : p \in \mathscr{P}(s)).\ P = Q.$$

Let $P, Q : p \in \mathscr{P}(s)$. We proceed by induction on the path p:

- **Base step.** Assume that $p = \varepsilon$.
  Both P and Q can only be obtained from the rule $\mathscr{P}_\varepsilon$ and are thus equal.
- **Inductive step.** Assume that $p = (n, k) :: p'$ where $n : \mathbb{N}$, $k : \textsc{Fin}\, n$ and $p' : \textsc{Path}$.
  The induction hypothesis is given by:

$$\forall (s : S).\ \forall (P, Q : p' \in \mathscr{P}(s)).\ P = Q. \tag{IH}$$

By case-analysis on $P : (n, k) :: p' \in \mathscr{P}(s)$, there exist an equality proof $e : n = \mathsf{rank}\, s$ and a proof $P' : p' \in \mathscr{P}(s[e_*(k)])$ such that $P' = \mathscr{P}_{::}\ e\ P'$. Likewise, by elimination of Q, there exist an equality proof $e' : n = \mathsf{rank}\, s$ and a proof $Q' : p' \in \mathscr{P}(s[e'_*(k)])$ such that $Q = \mathscr{P}_{::}\ e'\ Q'$. Because $\mathbb{N}$ satisfies UIP, we have that $e = e'$. Altogether, we establish that $P = Q$ as follows:

$$\begin{aligned} P &= \mathscr{P}_{::}\ e\ P' && \text{(by case-analysis)} \\ &= \mathscr{P}_{::}\ e'\ Q' && (e = e' \text{ by UIP on } \mathbb{N} \text{ and } P' = Q' \text{ by (IH)}) \\ &= Q. && \text{(by case-analysis)} \end{aligned}$$

□

Proposition 4 **Decidability of** $\mathscr{P}(s)$

For all states $s : S$, the predicate $\mathscr{P}(s)$ is decidable.

Proof. We have to show that:

$$\forall(p : \textsc{Path}).\ \forall(s : S).\ \textsc{Dec}(p \in \mathscr{P}(s)).$$

Let p : PATH and $s : S$. We proceed by induction on the path p:

- **Base step.** Assume that $p = \varepsilon$.
  In this case, $\textsc{Dec}(\varepsilon \in \mathscr{P}(s))$ is obtained by $\mathsf{yes}\ \mathscr{P}_\varepsilon$.
- **Inductive step.** Assume that $p = (n, k) :: p'$ where $n : \mathbb{N}$, $k : \textsc{Fin}\, n$ and p' : PATH.
  The induction hypothesis is given by:

  $$\forall(s : S).\ \textsc{Dec}(p' \in \mathscr{P}(s)). \tag{IH}$$

  Furthermore, we consider two cases by elimination of $n \stackrel{?}{=} \mathsf{rank}\, s$:

  - **Case 1.** Assume that $e : n = \mathsf{rank}\, s$.
    From the induction hypothesis (IH), with s picked to be $s[e_*(k)]$, we obtain a term $D : \textsc{Dec}(p' \in \mathscr{P}(s[e_*(k)]))$. By case-analysis on D:
    - **Case 1.1.** Assume that $D = \mathsf{yes}\, P'$ where $P' : p' \in \mathscr{P}(s[e_*(k)])$.
      In this case, $\textsc{Dec}(p \in \mathscr{P}(s))$ is obtained by $\mathsf{yes}(\mathscr{P}_{::}\ e\ P')$.
    - **Case 1.2.** Assume that $D = \mathsf{no}\, P'$ where $P' : p' \notin \mathscr{P}(s[e_*(k)])$.
      From hypothesis P', it is straightforward to show that there exists a proof P of $p \notin \mathscr{P}(s)$. Finally, we conclude that $\textsc{Dec}(p \in \mathscr{P}(s))$ with $\mathsf{no}\, P$.
  - **Case 2.** Assume that $e : n \neq \mathsf{rank}\, s$.
    In this case, we prove that there exists a proof P such that $p \notin \mathscr{P}(t)$:
    Assume that $p \in \mathscr{P}(t)$. Then, by case-analysis (using rule $\mathscr{P}_{::}$), there exists a proof e' such that $n = \mathsf{rank}\, s$. Contradiction.
    Finally, from P, we conclude that $\textsc{Dec}(p \in \mathscr{P}(t))$ with $\mathsf{no}\, P$. □

Successors

In this section, we define formally the type of successors of a given state. We follow a bottom-up approach in the sense that we start from the elementary notion of path defined previously.

Definition 8 **Successor state indexed by a path**

Given a state s and a path p, we compute the successor of s ending at p as follows:

$$\begin{aligned}
&_|_ : (s : S) \to \textsc{Path} \to \textsc{Maybe}\, S \\
&s\,|_\varepsilon \equiv \mathsf{just}\, s \\
&s\,|_{(n,k)::p} \equiv \textbf{match}\ n \stackrel{?}{=} \mathsf{rank}\, s\ \textbf{with} \\
&\qquad \mathsf{yes}\, e \Rightarrow s[e_*(k)]|_p \\
&\qquad \mathsf{no}\, _ \Rightarrow \mathsf{nothing} \\
&\quad \textbf{end}
\end{aligned}$$

Note that the return type of the function computing the successor of a state s is not S but MAYBE S. This is due to the fact that we do not know whether the path used in the computation of the successor is an s-path (see Figure 2.1 for an example of an invalid path). In essence, MAYBE is used here to provide a representation of partial functions in a total setting.

Alternatively, we could leverage dependent types in order to strengthen the specification of the function on the input parameters:

$$_|_ : (s : S) \to (p : \textsc{Path}) \to p \in \mathscr{P}(s) \to S.$$

Or, we could provide a justification explaining why the computation failed:

$$_|_ : (s : S) \to (p : \textsc{Path}) \to S \uplus p \notin \mathscr{P}(s).$$

Here, we choose to decompose the problem of defining the function computing the successor state and proving its specification:

Proposition 5 **Successor function specification**

Let s be a state of T and p be a path. We have,

$$\text{is-just}\left(s|_p\right) \iff p \in \mathscr{P}(s).$$

Proof. Let $s : S$ and $p : \textsc{Path}$. We proceed by induction on the path p:

▶ **Base step.** Assume that $p = \varepsilon$.

$$\begin{aligned} \text{is-just}(s|_\varepsilon) &\equiv \text{is-just}(\text{just}\, s) \\ &\Leftrightarrow \top && \text{(by definition of is-just)} \\ &\Leftrightarrow \varepsilon \in \mathscr{P}(s). && \text{(by rule } \mathscr{P}_\varepsilon) \end{aligned}$$

▶ **Inductive step.** Assume that $p = (n, k) :: p'$ where $n : \mathbb{N}$, $k : \textsc{Fin}\, n$ and $p' : \textsc{Path}$. The induction hypothesis is given by:

$$\forall (s : S).\ \text{is-just}\left(s|_{p'}\right) \iff p' \in \mathscr{P}(s). \tag{IH}$$

Finally, we have:

$$\begin{aligned} \text{is-just}\left(s|_p\right) &\Leftrightarrow \exists (e : n = \text{rank}\, s).\ \text{is-just}(s[e_*(k)]|_{p'}) && \text{(by case-analysis on } n \stackrel{?}{=} \text{rank}\, s) \\ &\Leftrightarrow \exists (e : n = \text{rank}\, s).\ p' \in \mathscr{P}(s[e_*(k)]) && \text{(by induction hypothesis (IH))} \\ &\Leftrightarrow p \in \mathscr{P}(s). && \text{(by rule } \mathscr{P}_{::}) \end{aligned}$$

□

Proposition 6 **Alternative successor specification**

Let $s : S$ be state of T and p a path. We have,

$$\text{is-nothing}\left(s|_p\right) \iff p \notin \mathscr{P}(s).$$

Proof. Immediate consequence of Proposition 2.5 and by definition of is-nothing. □

Definition 9 **Setoid structure on $\mathbb{P}(s)$**

Let $s : S$ be a state of T. We say that two s-paths, $p_1, p_2 : \mathbb{P}(s)$, are s-equivalent if and only if the successor states, respectively indexed by p_1 and p_2, are equivalent. Formally,

$$p_1 \approx_s p_2 \stackrel{\text{def}}{\iff} s|_{p_1} \approx_S s|_{p_2}.$$

We could explain the setoid structure on $\mathbb{P}(s)$ by analogy with pointer structures found in languages such as C or OCaml. In this context, a state s could be seen as a memory area and a path could be interpreted as a pointer within this memory. Thus, elements of $\mathbb{P}(s)$ are considered equal if they lead to isomorphic states, or said otherwise, if they point to memory cells representing equal values. To summarize, we are not interested in equality between paths (pointers) but rather, we are interested in equality between states (values).

One important property of the function computing the successor state indexed by a path is that it preserves the underlying setoid equality on states:

Proposition 7 Successor function respects setoid equality

Let $s, s' : S$ be two states of T and p be a path. If $s \approx_S s'$ then $s|_p \approx s'|_p$.

Proof. Let $s, s' : S$, $p : \textsc{Path}$ and $H : s \approx_S s'$. We proceed by induction on the path p:

- **Base step.** Assume that $p = \varepsilon$.
  In this case, we have $s|_\varepsilon = \mathsf{just}\, s$ and $s'|_\varepsilon = \mathsf{just}\, s'$. We have to prove that $\mathsf{just}\, s \approx \mathsf{just}\, s'$. By definition of the setoid equivalence, just is a congruence. Thus, it suffices to show that $s \approx s'$ which holds by assumption H.

- **Inductive step.** Assume that $p = (n, k) :: p'$ where $n : \mathbb{N}$, $k : \textsc{Fin}\, n$ and $p' : \textsc{Path}$.
  The induction hypothesis is given by:

$$\forall (s, s' : S).\ s \approx_S s' \to s|_{p'} \approx s'|_{p'}. \tag{IH}$$

  From hypothesis H, it is straightforward to obtain a proof r such that $\mathsf{rank}\, s = \mathsf{rank}\, s'$.

$$\begin{array}{l} \textbf{match}\ n \stackrel{?}{=} \mathsf{rank}\, s\ \textbf{with} \\ \quad \mathsf{yes}\, e \Rightarrow s[e_*(k)]|_{p'} \\ \quad \mathsf{no}\, _ \Rightarrow \mathsf{nothing} \\ \textbf{end} \end{array} \approx \begin{array}{l} \textbf{match}\ n \stackrel{?}{=} \mathsf{rank}\, s'\ \textbf{with} \\ \quad \mathsf{yes}\, e \Rightarrow s'[e_*(k)]|_{p'} \\ \quad \mathsf{no}\, _ \Rightarrow \mathsf{nothing} \\ \textbf{end} \end{array}$$

$\iff$

  By case-analysis on $n \stackrel{?}{=} \mathsf{rank}\, s$:

  - **Case 1.** Assume that $e : n = \mathsf{rank}\, s$. In addition, since we have $\mathsf{rank}\, s = \mathsf{rank}\, s'$, we deduce that there exists a proof e' of $n = \mathsf{rank}\, s'$. Therefore, we can simplify $n \stackrel{?}{=} \mathsf{rank}\, s'$ into $\mathsf{yes}\, e'$.

$$\begin{array}{ll} & s[e_*(k)]|_{p'} \approx s'[e'_*(k)]|_{p'} \\ \iff & \{ \text{ induction hypothesis (IH) } \} \\ & s[e_*(k)] \approx_S s'[e'_*(k)] \\ \iff & \{ \text{ by rewriting } e' \text{ into } e \text{ with } \textsc{uip} \} \\ & \top. \end{array}$$

  - **Case 2.** Assume that $e : n \neq \mathsf{rank}\, s$.
    From e, we can deduce that there exists a proof e' of $n \neq \mathsf{rank}\, s'$. Thus, $n \stackrel{?}{=} \mathsf{rank}\, s'$ can be simplified into $\mathsf{no}\, e'$. Finally, we conclude that $\mathsf{nothing} \approx \mathsf{nothing}$ by reflexivity. □

Now, we formalize the successor[1] relation between states within a transition system.

[1] The successor relation is indeed the transitive closure of the one-step relation induced by next.

Definition 10 Successor relation (transitive)

Let $s, s' : S$ be two states of T. We say that s' is a successor of s, denoted $s \rightsquigarrow s'$, if there exists a path p such that the successor state of s indexed by p is s'. Formally,

$$_\rightsquigarrow_ : S \to S \to \textsc{Type}$$
$$s \rightsquigarrow s' \;\equiv\; \left\{ p : \textsc{Path} \mid s|_p \in \lfloor _\sim_ \; s' \rfloor \right\}.$$

Moreover, the set of successors starting from a state s, denoted $s\rightsquigarrow$, is defined as:

$$_\rightsquigarrow : S \to \textsc{Type}$$
$$s\rightsquigarrow \;\equiv\; \textstyle\sum_{s':S} s \rightsquigarrow s'.$$

Here, the notation $\lfloor _\approx_S_ \; s' \rfloor$ denotes the lifting of the predicate $_\approx_S_ \; s' : S \to \textsc{Prop}$ to elements on $\textsc{Maybe}(S)$. Thus, we have $\lfloor _\approx_S_ \; s' \rfloor : \textsc{Maybe}(S) \to \textsc{Prop}$. We refer to Appendix A for the formal definition.

Remark. Even though we say that $_\rightsquigarrow_$ is a relation, it is important to note that $s \rightsquigarrow s'$ lives in $\textsc{Type}$ and not $\textsc{Prop}$. This means that $s \rightsquigarrow s'$ is computationally-relevant. Concretely, when we say that s' is a successor of s, we are not just interested in this logical fact, but we are also interested in the path p starting at s and leading to s'.

As for the type $\mathbb{P}(s)$, we give a setoid structure to the type $s\rightsquigarrow$.

Definition 11 Setoid structure on $s\rightsquigarrow$

Let $s : S$ be a state of T. We say that two successor states $s_1, s_2 : s\rightsquigarrow$ are equivalent if and only if $\mathsf{proj}_1\, s_1 \approx_S \mathsf{proj}_1\, s_2$.

The following proposition shows that the type of paths induces an induction principle over the successor relation:

Proposition 8 Induction principle for the successor relation

Let $P : S \to S \to \textsc{Type}$ be a relation on states of T. If,

(i) $\forall (s, s' : S).\; s \approx s' \to P\; s\; s'$,

(ii) $\forall (s, s' : S).\; \forall (k : \textsc{Fin}(\mathsf{rank}\, s)).\; s[k] \rightsquigarrow s' \to P\; (s[k])\; s' \to P\; s\; s'$,

then,

$$\forall (s, s' : S).\; s \rightsquigarrow s' \to P\; s\; s'.$$

Proof. Let $P : S \to S \to \textsc{Type}$. Suppose that both (i) and (ii) holds. Moreover, let $s, s' : S$ and $H = (p, H') : s \rightsquigarrow s'$ where $p : \textsc{Path}$ and $H' : s|_p \in \lfloor _\approx_S_ \; s' \rfloor$. We proceed by induction on the path p:

- **Base step.** Assume that $p = \varepsilon$.
  When we substitute p by ε in H', we have a proof that $\mathsf{just}\, s \in \lfloor _\approx_S_ \; s' \rfloor$. Furthermore, by case-analysis on H', we deduce that $s \approx_S s'$. Finally, we use this fact and hypothesis (i) to conclude that $P\; s\; s'$ holds.
- **Inductive step.** Assume that $p = (n, k) :: p'$ with $n : \mathbb{N}$, $k : \textsc{Fin}\, n$ and $p' : \textsc{Path}$.
  The induction hypothesis is given by:

$$\forall (s, s' : S).\; s|_{p'} \in \lfloor _\approx_S_ \; s' \rfloor \to P\; s\; s'. \tag{IH}$$

By case-analysis on $n \stackrel{?}{=} \mathsf{rank}\, s$ we consider two cases:

- **Case 1.** Assume that $e : n = \mathsf{rank}\, s$.
  In this case, the computation $s|_p$ reduces to $s[e_*(k)]|_{p'}$. In addition, the type of the hypothesis H' is now $s[e_*(k)]|_{p'} \in \lfloor _ \approx_S _\ s' \rfloor$. To prove $P\ s\ s'$, we use hypothesis (ii) with $k := e_*(k)$. Thus, it remains to show that $P\ (s[e_*(k)])\ s'$ holds. This follows directly from the induction hypothesis (IH) instantiated with hypothesis H'.

- **Case 2.** Assume that $e : n \neq \mathsf{rank}\, s$.
  We show that this case leads to a contradiction. The computation $s|_p$ reduces to $\mathsf{nothing}$. Consequently, the type of H' is now $\mathsf{nothing} \in \lfloor _ \approx_S _\ s' \rfloor$ which is clearly empty. Contradiction. □

Remark. Here, in order to define the successor relation, we followed a bottom-up approach. First, we started with the elementary notion of paths, which is then restricted to the set of paths within a transition system—through the predicate $\mathscr{P}$. In addition, starting from a state s and a path p, we defined the (partial) function computing the state ending at p. Altogether, this allowed us to derive the definition of the successor relation. Furthermore, we prove an induction principle on it, derived from the induction principle on paths.

Alternatively, we could have followed a top-down approach. In this case, the successor relation could be defined directly as an inductive type:

$$\rightsquigarrow_{\varepsilon}\ \frac{s \approx_S s'}{s \rightsquigarrow s'}, \qquad \rightsquigarrow_{::}\ \frac{k : \textsc{Fin}(\mathsf{rank}\, s) \qquad s[k] \rightsquigarrow s'}{s \rightsquigarrow s'}.$$

On the one hand, the constructors of the successor relation are in fact the witnesses of the existence of a path, starting from a state s and ending at a state s'. The names of the constructors are thus chosen accordingly to emphasize this fact.

On the other hand, the indices of the successor relation ensure—through typing—that all paths built from the constructors are always within the transition system. Consequently, we eliminate *by construction* any path which is not valid in the transition system.

Even though the latter approach seems more appealing, we argue that the first one, namely the bottom-up approach, is still preferable. Indeed, it allows for better compositional reasoning as we can define computations on "untyped" paths. Then, these computations can be proven to preserve typing, if required. Another benefit is that paths are defined as plain lists. As such, we can reuse the underlying theory of lists to derive functions or properties on paths. Finally, Theorem 2.3 highlights precisely that the path is the only computationally-relevant part in the successor relation. We think that this less obvious when starting from the inductive definition.

Proposition 9 Sequentialization of successor computation

Let $s : S$ be an state of T and p, p' two paths. Then, the following equality holds:

$$s|_{p +\!\!+ p'} = s|_p \gg\!= _|_{p'}.$$

Proof. Trivial by induction on p. □

Proposition 10 **Successor relation is a preorder**

The subtree relation $_\rightsquigarrow_$ *is a preorder:*

(i) $\forall(s:T).\ s \rightsquigarrow s$,

(ii) $\forall(s,s',s'':S).\ s \rightsquigarrow s' \to s' \rightsquigarrow s'' \to s \rightsquigarrow s''$.

We overload the notations on paths denoting $\varepsilon : s \rightsquigarrow s$ *and* $_{+\!\!+}_ : s \rightsquigarrow s' \to s' \rightsquigarrow s'' \to s \rightsquigarrow s''$.

Proof.

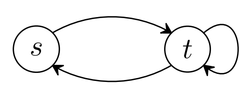
(i) Let $s : S$. The first component of $s \rightsquigarrow s$ is defined to be the empty path ε. Thus it remains to show that $s|_\varepsilon \in \lfloor _\approx_S_\ s \rfloor$. This follows directly from the fact that $s|_\varepsilon = \mathsf{just}\, s$ and that the setoid equivalence $_\approx_S_$ is reflexive.

(ii) Let $s, s', s'' : S$, $H = (p, H_p) : s \rightsquigarrow s'$ and $H' = (p', H_{p'}) : s' \rightsquigarrow s''$ where $p, p' : \textsc{Path}$, $H_p : s|_p \in \lfloor _\approx_S_\ s' \rfloor$ and $H_{p'} : s'|_{p'} \in \lfloor _\approx_S_\ s'' \rfloor$. The first component of $s \rightsquigarrow s''$ is defined to be $p +\!\!+ p'$. Thus, it remains to prove that $s|_{p+\!\!+p'} \in \lfloor _\approx_S_\ s'' \rfloor$. By case-analysis on H_p, there exist a state $u : S$, a proof $e : s|_p = \mathsf{just}\, u$ and a proof that $b : s' \approx_S u$:

$$
\begin{aligned}
& s|_{p+\!\!+p'} \in \lfloor _\approx_S_\ s'' \rfloor \\
\iff & \quad \{ \text{ by Prop. 2.9 [Seq. succ. computation] } \} \\
& s|_p \gg\!= \ -|_{p'} \in \lfloor _\approx_S_\ s'' \rfloor \\
\iff & \quad \{ \text{ by Prop. A.5 } [x \gg\!= f \in \lfloor P \rfloor \Leftrightarrow x \in \lfloor \lfloor P \rfloor \circ f \rfloor] \ \} \\
& s|_p \in \Big\lfloor \lfloor _\approx_S_\ s'' \rfloor \circ -|_{p'} \Big\rfloor \\
\iff & \quad \{ \text{ by rewriting equation } e \ \} \\
& \mathsf{just}\, u \in \Big\lfloor \lfloor _\approx_S_\ s'' \rfloor \circ -|_{p'} \Big\rfloor \\
\iff & \quad \{ \text{ by Prop. A.3 } [m \in \lfloor P \rfloor \Leftrightarrow \exists(a:A).\ m = \mathsf{just}\, a \wedge P(a)] \ \} \\
& u|_{p'} \in \lfloor _\approx_S_\ s'' \rfloor \\
\iff & \quad \{ \text{ by rewriting equation } b \ \} \\
& s'|_{p'} \in \lfloor _\approx_S_\ s'' \rfloor \\
\iff & \quad \{ \text{ by assumption } H_{p'} \ \} \\
& \top.
\end{aligned}
$$

□

Remark. Consider the transition system on the right composed of two states s and t. The rank of s is 1 while the rank of t is 2. Clearly, we have $s \rightsquigarrow t$ and conversely, we have $t \rightsquigarrow s$. However, both states s and t are not equivalent. Consequently, the relation $\rightsquigarrow$ is not antisymmetric.



The following proposition highlights the fact that the subtree relation inherits properties of paths:

Proposition 11 **Associativity of successor concatenation**

The concatenation operation on successor states is associative.

Proof. Let $s_1, s_2, s_3, s_4 : S$ and $p_1 : s_1 \rightsquigarrow s_2$, $p_2 : s_2 \rightsquigarrow s_3$ and $p_3 : s_3 \rightsquigarrow s_4$. Then, we show that:

$$p_1 +\!\!+ p_2 +\!\!+ p_3 \approx (p_1 +\!\!+ p_2) +\!\!+ p_3$$

which by definition of the underlying setoid equivalence on the successor relation is the same as:

$$\mathsf{proj}_1\, p_1 \mathbin{+\!\!+} \mathsf{proj}_1\, p_2 \mathbin{+\!\!+} \mathsf{proj}_1\, p_3 = \big(\mathsf{proj}_1\, p_1 \mathbin{+\!\!+} \mathsf{proj}_1\, p_2\big) \mathbin{+\!\!+} \mathsf{proj}_1\, p_3.$$

Here, the concatenation operator ($_\mathbin{+\!\!+}_$) operates on path which is clearly associative. □

Remark. Similarly, it is immediate to check that, for all states $s_1, s_2 : S$ and $p : s_1 \rightsquigarrow s_2$, we have $\varepsilon \mathbin{+\!\!+} p \approx p$ and $p \mathbin{+\!\!+} \varepsilon \approx p$.

The following theorem emphasizes the fact that the only relevant information is the path and not the successor state indexed by it. Indeed, the type of s-path is already indexed by s and thus given a path the successor state indexed by it can easily be computed.

Theorem 3 **s-path / s-successor isomorphism**

Let $s : S$ be a state of T. We have the following setoid isomorphism:

$$s{\rightsquigarrow} \cong \mathbb{P}(s).$$

Proof. Let s be a state. In order to show that $s{\rightsquigarrow}$ is isomorphic to $\mathbb{P}(s)$, we have to construct two morphisms $f : s{\rightsquigarrow} \longrightarrow \mathbb{P}(s)$ and $g : \mathbb{P}(s) \longrightarrow s{\rightsquigarrow}$ and show that they are inverse of one another.

First, we start by defining the setoid morphisms f and g. The domains and codomains of both f and g are sigma-types, therefore we begin with their first components:

$$f_1 : s{\rightsquigarrow} \to \textsc{Path} \qquad\qquad g_1 : \mathbb{P}(s) \to S$$
$$f_1\ (_, p, _) \equiv p \qquad\qquad g_1\ (p, H) \equiv \mathsf{from\text{-}just}\ \big(s|_p\big)\ H'$$

where H' is a proof that $\mathsf{is\text{-}just}(s|_p)$. It is obtained from $H : p \in \mathscr{P}(s)$ applied to Proposition 2.5 [Succ. function spec.].

The second components of f and g are obtained as follows:

- $f_2 : \forall (q : s{\rightsquigarrow}).\ f_1(q) \in \mathscr{P}(s)$.
  Let $q = (q', p, H) : s{\rightsquigarrow}$ where $q' : S$, $p : \textsc{Path}$ and $H : s|_p \in \lfloor _{\approx_S}_\ q' \rfloor$. By elimination of H, there exists $u : S$ such that $\mathsf{just}\, u = s|_p$ and $u \approx_S q'$. From the fact that $\mathsf{just}\, u = s|_p$, we know that $\mathsf{is\text{-}just}(s|_p)$ holds. Finally, with Proposition 2.5, we conclude that $f_1(q) \in \mathscr{P}(s)$.

- $g_2 : \forall (p : \mathbb{P}(s)).\ s \rightsquigarrow g_1(p)$.
  Let $p = (p', H) : \mathbb{P}(s)$ where $p' : \textsc{Path}$ and $H : p' \in \mathscr{P}(s)$. We pick p' to be the first component of $s \rightsquigarrow g_1(p)$. It remains to prove that $s|_{p'} \in \lfloor _{\approx_S}_\ g_1(p) \rfloor$. By case-analysis on $s|_{p'}$, we consider two cases:

  - **Case 1.** Assume that $e : s|_{p'} = \mathsf{just}\, u$ where $u : S$.

  $$\begin{aligned} & s|_{p'} \in \lfloor _{\approx_S}_\ g_1(p) \rfloor \\ = \quad & \{ \text{ by rewriting e and by definition of } g_1 \} \\ & \mathsf{just}\, u \in \lfloor _{\approx_S}_\ u \rfloor \\ \iff \quad & \{\ _{\approx_S}_ \text{ is reflexive} \} \\ & \top. \end{aligned}$$

  - **Case 2.** Assume that $e : s|_{p'} = \mathsf{nothing}$.
  With Proposition 2.6 and hypothesis e, we obtain a proof that $p' \notin \mathscr{P}(s)$ which contradicts the hypothesis $H : p' \in \mathscr{P}(s)$.

Now that both components are defined, we define both f and g as follows:

$$f : s{\rightsquigarrow} \to \mathbb{P}(t) \qquad\qquad g : \mathbb{P}(s) \to s{\rightsquigarrow}$$
$$f\ \ q\ \equiv\ (f_1(q), f_2(q)). \qquad\qquad g\ \ p\ \equiv\ (g_1(p), g_2(p)).$$

Here, we have yet to show that both f and g preserve the underlying setoid equivalences:

- $\forall (q_1, q_2 : s{\rightsquigarrow}).\ q_1 \approx q_2 \to f(q_1) \approx f(q_2)$.

Let $q_i = (q_i', p_i, H_i)$ where $q_i' : S$, p_i : PATH and $H_i : s|_{p_i} \in \lfloor _ \approx_S _\ q_i' \rfloor$ for $i \in \{1, 2\}$. Moreover, let $E : q_1 \approx q_2$ which by Def. 2.11 is the same as $E : q_1' \approx_S q_2'$

$$
\begin{aligned}
& f(q_1) \\
\approx\ & \quad \{ \text{ by definition of setoid equivalence } \} \\
& s|_{f_1(q_1', p_1, H_2)} \\
\equiv\ & \quad \{ \text{ by definition of } f_1 \} \\
& s|_{p_1} \\
\approx\ & \quad \left\{ \begin{array}{l} \text{By case-analysis on } H_i \text{, there exists two states } u_1, u_2 : S \text{ such that we have} \\ \textsf{just}\, u_i = s|_{p_i} \text{ and } u_i \approx_S q_i'. \text{ Along with hypothesis } E \text{, this allows us to} \\ \text{conclude that } u_1 \approx_S q_1' \approx_S q_2' \approx_S u_2. \end{array} \right\} \\
& s|_{p_2} \\
\equiv\ & \quad \{ \text{ by definition of } f_1 \} \\
& s|_{f_1(q_2', p_2, H_2)} \\
\approx\ & \quad \{ \text{ by definition of setoid equivalence } \} \\
& f(q_2).
\end{aligned}
$$

- $\forall (p_1, p_2 : \mathbb{P}(s)).\ p_1 \approx p_2 \to g(p_1) \approx g(p_2)$.

Let $p_i = (p_i', H)$ where p_i' : PATH and $H_i : p_i \in \mathscr{P}(s)$ for $i \in \{1, 2\}$. Moreover, let $E : p_1 \approx p_2$ which by Def. 2.9 is equivalent to $E : s|_{p_1'} \approx s|_{p_2'}$.

$$
\begin{aligned}
& g(p_1) \\
\approx\ & \quad \left\{ \begin{array}{l} \text{By definition of setoid equivalence and by definition of } g_1. \text{ Moreover, the } 2^{\text{nd}} \\ \text{part of } g(p_1) \text{ is generalized yielding a proof } J_1 : \textsf{is-just}\big(t|_{p_1'}\big). \end{array} \right\} \\
& \textsf{from-just}\ \left(s|_{p_1'}\right)\ J_1 \\
\approx\ & \quad \left\{ \begin{array}{l} \text{By case-analysis on } E \text{, there exists a proof } u_i \text{ such that } s|_{p_i'} = \textsf{just}\, u_i \text{ and} \\ \text{a proof } e : u_1 \approx_S u_2 \text{ ; cases where the case-analysis of } s|_{p_i'} \text{ yields } \textsf{nothing} \\ \text{are omitted since they contradict the hypotheses } H_i \text{ (see Prop. 2.6). Finally,} \\ \text{since we have } \textsf{from-just}\ (s|_{p_i'})\ J_i = u_i \text{,} \\ \text{we can conclude with } e. \end{array} \right\} \\
& \textsf{from-just}\ \left(s|_{p_2'}\right)\ J_2 \\
\approx\ & \quad \left\{ \begin{array}{l} \text{By definition of setoid equivalence and by definition of } g_1. \text{ Moreover, the } 2^{\text{nd}} \\ \text{part of } g_1(p_2') \text{ is generalized yielding a proof } J_2 : \textsf{is-just}(s|_{p_2'}). \end{array} \right\} \\
& g(p_2').
\end{aligned}
$$

Finally, we show that f and g are inverse of one another:

(i) $\forall (q : s{\rightsquigarrow}).\ g(f(q)) \approx q$.

Let $q = (q', p, H) : s{\rightsquigarrow}$ with $q' : S$, p : PATH and $H : s|_p \in \lfloor _ \approx_S _\ q' \rfloor$.

$$
\begin{aligned}
g(f(q)) &\approx g_1(f(q)) && \text{(by definition of setoid equivalence and of } g\text{)} \\
&\equiv \textsf{from-just}\ \left(s|_{f_1(p)}\right)\ (f_2(q)) && \text{(by definition of } g_1 \text{ and by definition of } f\text{)} \\
&\approx q' && (f_1(p) = p \text{ and by case-analysis on } H\text{)} \\
&\approx q. && \text{(by definition of setoid equivalence)}
\end{aligned}
$$

(ii) $\forall (p : \mathbb{P}(t)).\ f(g(s)) \approx p$.

Let $p = (p', H) : \mathbb{P}(s)$ with p' : PATH and $H : p' \in \mathscr{P}(s)$.

$$
\begin{aligned}
f(g(p)) &\approx s|_{f_1(g(p))} && \text{(by definition of setoid equivalence and of } f\text{)} \\
&\approx s|_{p'} && \text{(by definition of } g_2 \text{ we have } f_1(g_2(p)) = p'\text{)} \\
&\approx p. && \text{(by definition of setoid equivalence)}
\end{aligned}
$$

□

2.1.4. Finite Type of Successors

In this section, we define the type of states having the property that the set of reachable states (successors) is finite. This notion is similar to the regularity property of infinite trees. Then, we prove a closure property on such states: any reachable state has a finite set of reachable states.

In a classical setting, this property is immediate. Indeed, Figure 2.2 clearly shows that, from a state s with a finite set of reachable states $s{\leadsto}$, the reachable set of states of any successor state t is necessarily contained in $s{\leadsto}$. Since finite sets are closed under inclusion, we can deduce that the set of reachable states of t is also finite.

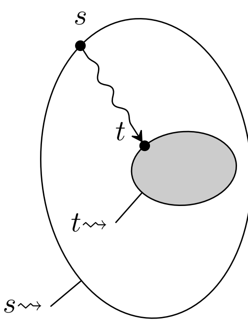


Figure 2.2. Reachable states

However, things are not as well-behaved in a constructive setting: in general, finite types (setoids) are not closed under inclusion. For instance, assume that it is the case for weakly finitely indexed setoids. Then, given a weakly finitely indexed setoid B and an inclusion map $i : A \hookrightarrow B$ from a setoid A, we obtain a proof that A is weakly finitely indexed. From this, we can show that the law of excluded-middle is provable. Let P : PROP be an arbitrary proposition. We pick for B the type defined as $B :=$ UNIT. Furthermore, we define the subset type $A := \{ b : B \mid P \}$ where the underlying setoid equality is given by:

$$(b, p) \approx_A (b', p') \overset{\text{def}}{\Longleftrightarrow} b = b' \wedge p \leftrightarrow p'.$$

Clearly, we have an inclusion map $i : A \hookrightarrow B$ defined to be $i := \lambda(_ : A).\ \mathtt{tt}$. Since B is weakly finitely indexed, then so is A. Finally, by observing the upper bound of the cardinal of A, we consider two cases: if $\#A = 0$, then A is empty and $\neg P$ holds, whereas if $\#A \geq 1$, then A is inhabited and P holds. Consequently, we have shown that either P or $\neg P$ hold.

As illustrated above, the inclusion map is not enough—constructively—for finite types to be closed under inclusion. The missing component is a function deciding which elements of B are in the range of the inclusion map. From this, we can partition B and thus, prove that A is finite.

In the following, we define formally, in an $\mathcal{U}$-coalgebra, the type characterizing states such that the set of their reachable states is finite.

Definition 12 **Has finite successors**

The predicate SUCCFINITE *characterizing states that have a finite set of successors is defined as follows:*

$$\begin{aligned}&\textsc{SuccFinite} : S \to \textsc{Type}\\&\textsc{SuccFinite}\ s \ \equiv\ \mathrm{WFI}(s{\leadsto}).\end{aligned}$$

Remark. For a given state $s : S$ of a coalgebra T, we may write $\textsc{SuccFinite}_T(s)$ to make explicit which coalgebra is considered.

Definition 13 **Finite Successors**

Given a transition system $T : \mathcal{U}$, the type of states having a finite set of successors is defined as follows:

$$\begin{aligned}&\textsc{FSucc}_ : \mathcal{U} \to \textsc{Type}\\&\textsc{FSucc}_{(S, out_S)} \ \equiv\ \textstyle\sum_{s:S} \textsc{SuccFinite}(s).\end{aligned}$$

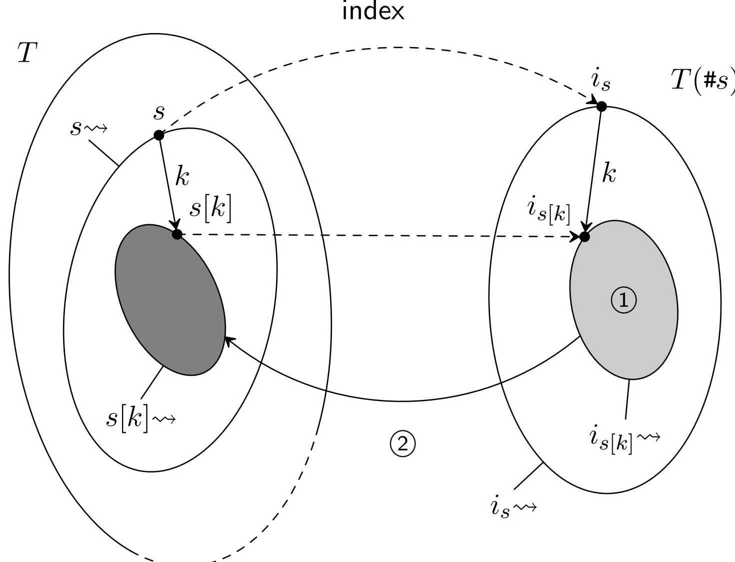


Figure 2.3. Overview of the proof of finiteness of the set of reachable states.

In the remaining of this section, we establish the following closure property: "given a state s such that the set of its successors is finite then, for any of its successors s', the set of its successors is again finite". As explained in the introduction of this section and illustrated in Figure 2.2, if we want to prove the closure property constructively from an inclusion map $i : t{\rightsquigarrow} \hookrightarrow s{\rightsquigarrow}$, we have to somehow be able to filter out the successor of s that are not in the reachable states from t. Though, it is not obvious whether it is possible without decidable equality on states. Indeed, the definition of finiteness chosen in Definition 2.12 is relatively weak and does not entail, in general, decidability of the underlying state equality.

Nevertheless, this version of finiteness still gives us an indexing map to a domain with a decidable equality. Let us recall that this indexing map *is not* a setoid morphism, *i.e.*, equivalent states are not necessarily mapped to equivalent indices.

Figure 2.3 gives an overview of the main steps of the proof of the closure property. Here, T represents an arbitrary $\mathcal{U}$-coalgebra, s denotes a state of T such that the set of its reachable states is finite. We consider an immediate successor state $s[k]$, the goal is then to show that the type $s[k]{\rightsquigarrow}$ is finite. To this end, we first show that the state s induces an $\mathcal{U}$-coalgebra, denoted $T(\#s)$, on the type of indices of $s{\rightsquigarrow}$. Next, we show how to *compute* the set of reachable sets from the state $i_{s[k]}$, thus proving the finiteness of the type $i_{s[k]}{\rightsquigarrow}$. This corresponds to the step ① in Figure 2.3. Finally, in step ②, we show how the type $i_{s[k]}{\rightsquigarrow}$ and $s[k]{\rightsquigarrow}$ are related. Essentially, we prove that there exist two maps $f : i_{s[k]}{\rightsquigarrow} \to s[k]{\rightsquigarrow}$ and $g : s[k]{\rightsquigarrow} \to i_{s[k]}{\rightsquigarrow}$ such that $\forall x.\ f(g(x)) \approx_T x$. Then, we prove that the finiteness of $s[k]{\rightsquigarrow}$ follows from the finiteness of $i_{s[k]}{\rightsquigarrow}$ shown at step ①.

Transitive closure computation

Let $T : \mathcal{U}$ be a transition system. We call S the carrier of T. Moreover, we assume that the type S has a decidable Leibniz equality and that every list l of states is $\succ$-accessible, *i.e.*, we have a proof W : Well-founded(_≻_).

First, we define the following binary relation _⊏_ on pairs of list of states:

$$(l_1, l_2) \sqsubset (l'_1, l'_2) \overset{\text{def}}{\iff} l_1 \succ l'_1 \vee (l_1 = l'_1 \wedge \mathsf{length}\, l_2 < \mathsf{length}\, l'_2).$$

The relation $_\sqsubset_$ is the lexicographic order induced by $_<_$ on $\mathbb{N}$ and $_\succ_$ on lists (see Def. A.38). Since both relations are well-founded, so is their lexicographic product (see [Pau86]).

We compute the reflexive-transitive closure induced by the Next transition function as follows:

$$\begin{aligned}
&\mathsf{succs} : (M : \textsc{List}\, S) \to (P : \textsc{List}\, S) \to \textsc{Acc}\ (_\sqsubset_)\ (P, M) \to \textsc{List}\, S\\
&\mathsf{succs}\ \ M\ \ []\ \ _\ \equiv\ M\\
&\mathsf{succs}\ \ M\ \ (p :: P)\ \ (\mathsf{acc\text{-}intro} f)\ \equiv\ \textbf{match}\ p \in^{?} M\ \textbf{with}\\
&\qquad \mid \mathsf{yes}\, m\ \Rightarrow \mathsf{succs}\ M\ P\ (f\ H)\\
&\qquad \mid \mathsf{no}\, nm \Rightarrow \mathsf{succs}\ (p :: M)\ (\mathsf{Next}(p) +\!\!+ P)\ (f(H'))\\
&\qquad \textbf{end}\\
&\quad \textbf{where}\ \ H : (M, P) \sqsubset (M, p :: P)\\
&\qquad\qquad H' : (p :: M, \mathsf{Next}\, p +\!\!+ P) \sqsubset (M, p :: P).
\end{aligned}$$

The terms for both H and H' are omitted but the proofs are straightforward. Indeed, for H it suffices to show that $\mathsf{length}(P) < 1 + \mathsf{length}(P) = \mathsf{length}(p :: P)$. For H', the proof of $\lceil p :: M \rceil$ is derived from nm.

In the function succs, the list M represents a set of *marked* states, *i.e.*, states that have already been visited. The list P is a set of states to be *processed*, that is, successor states that have yet to be visited. The function Next used above is defined as the list of immediate successors of a state:

$$\begin{aligned}
&\mathsf{Next} : S \to \textsc{List}\, S\\
&\mathsf{Next}\ \ s\ \ \equiv\ \ \mathsf{tabulate}(\mathsf{Next}\, s).
\end{aligned}$$

Finally, the reflexive-transitive closure of reachable states from a state s is computed as follows:

$$\begin{aligned}
&[_]^{*} : S \to \textsc{List}\, S\\
&[s]^{*}\ \equiv\ \mathsf{succs}\ []\ [\, s\,]\ m
\end{aligned}$$

where m is a proof that $\textsc{Acc}\ (_\sqsubset_)\ m$ which is derived from W. The soundness and completeness of $[_]^{*}$ is established in Appendix C.

Induced unranked coalgebras

Proposition 12 Coalgebra structure on successor states

Let $s : S$ be a designated state of T. The type $s{\leadsto}$ of successors of s has an $\mathcal{U}$-coalgebra structure. This induced coalgebra is denoted by $T(s) : \mathcal{U}$.

Proof. Let $s : S$ be a state. We have to construct a coalgebra for the carrier setoid $s{\leadsto}$. To this end, it suffices two give to maps rank, and next that respects the underlying setoid equivalences:

$$\begin{aligned}
&\mathsf{rank} : s{\leadsto} \to \mathbb{N}\\
&\mathsf{rank}\ \equiv\ \mathsf{rank}_T \circ \mathsf{proj}_1
\end{aligned}$$

$$\begin{aligned}
&\mathsf{next} : \forall (s' : s{\leadsto}).\ \textsc{Fin}(\mathsf{rank}\, s') \to s{\leadsto}\\
&\mathsf{next}\ \ (s', H)\ \ k\ \equiv\ (\mathsf{next}_T\ s'\ k, H')\\
&\qquad \textbf{where}\ \ H' : s \leadsto s[k]
\end{aligned}$$

where H' is a proof obtain by transitivity of $_\leadsto_$ (Prop. 2.10) and $H : s \leadsto s'$. The proof that both functions, namely rank and next, respect the underlying equality are straightforward. Indeed, they are a consequence of the fact that rank_T and next_T are already setoid morphisms. □

The following proposition corresponds to step ① of Figure 2.3.

Proposition 13 Coalgebra structure induced by the indexing function

Let σ : FSUCC$_T$ be a state of T such that the set of its successors is finite. The type FIN $\#\sigma$, *has an $\mathcal{U}$-coalgebra structure, denoted by $T(\#\sigma)$. Furthermore, the set of successors of any state of $T(\#\sigma)$ is also finite.*

Proof. Let σ : FSUCC$_T$ be a state of T. We denote by s the state defined to be $\mathsf{proj}_1\,\sigma$. First, we give to the type FIN $\#\sigma$ a structure of an $\mathcal{U}$-coalgebra. As was the case with the previous proposition, we define the two observation functions:

$$\mathsf{rank} : \text{FIN}\,\#\sigma \to \mathbb{N} \qquad \mathsf{next} : \forall(i : \text{FIN}\,\#\sigma).\ \text{FIN}(\mathsf{rank}\,i) \to \text{FIN}\,\#\sigma$$
$$\mathsf{rank} \;\equiv\; \mathsf{rank}_{T(\sigma)} \circ \mathsf{value}, \qquad \mathsf{next}\;\; i\;\; k \;\equiv\; \mathsf{index}(\mathsf{next}_{T(\sigma)}\;(\mathsf{value}\,i)\;k).$$

The proofs that both rank and next respects the underlying setoid equivalences are straightforward. Finally, we have to show that for a state i of $T(\#\sigma)$, its set of successors is finite. Formally, we have to prove:

$$\forall(i : \text{FIN}\,\#\sigma).\ \text{SUCCFINITE}(i).$$

Let i : FIN $\#\sigma$ be a state. We have to show that the type $i{\rightsquigarrow}$ is weakly finitely indexed. By Proposition 1.12, it suffices to show that the type is listable, *i.e.*, that there exists a list containing all elements of $i{\rightsquigarrow}$. The reflexive-transitive closure $l := [i]^*$ computes such a list. However, in order to use the function $[_]^*$, we have yet to show that:

(i) the type FIN $\#\sigma$ has a decidable equality,

(ii) every list of elements of FIN $\#\sigma$ is $\succ$-accessible.

The proof of (i) is given by Proposition A.6. The proof of (ii) is given by Proposition A.15 since the type FIN $\#\sigma$ is listable. By Theorems C.1 and C.2, the list l contains all successors of i. □

Now, we prove the result illustrated by step ② in Figure 2.3.

Lemma 1 $\mathbb{P}_{T(\#\sigma)}(-)$ **weak retract of** $\mathbb{P}_T(-)$

Let σ : FSUCC$_T$ be a state and $s : \sigma{\rightsquigarrow}$ be a successor state of σ. Then, the type $\mathbb{P}_{T(\#\sigma)}(\mathsf{index}\,s)$ is a weak retract of $\mathbb{P}_T(s)$.

Proof. Let σ : FSUCC$_T$ and $s : \sigma{\rightsquigarrow}$ be a successor state. We have to show that there exist two maps $r : \mathbb{P}(\mathsf{index}\,s) \longrightarrow \mathbb{P}(s)$ and $i : \mathbb{P}(s) \to \mathbb{P}(\mathsf{index}\,s)$ such that $\forall s'.\ r(i(s')) \approx s'$.

- $r : \mathbb{P}(\mathsf{index}\,s) \longrightarrow \mathbb{P}(s)$.

  We define the map r as follows:

$$r : \mathbb{P}(\mathsf{index}\,s) \to \mathbb{P}(s)$$
$$r\;\;(p, H) \;\equiv\; (p, f(H))$$
$$\textbf{where}\;\; f : p \in \mathscr{P}(\mathsf{index}\,s) \to p \in \mathscr{P}(s).$$

  The function f is obtained by path induction on p and by exploiting the fact that index is a section of value. It remains to show that r is a setoid morphism. Let $P, P' : \mathbb{P}(\mathsf{index}\,s)$ and $E : P \approx P'$, we have to show that $r(P) \approx r(P')$:

$$\begin{aligned}
r(P) &\equiv s|_{\mathsf{proj}_1\,P} && \text{(by Def. 2.9 [Setoid structure on } \mathbb{P}])\\
&\approx \mathsf{map}^{\text{MAYBE}}\ \mathsf{value}\ \left(\mathsf{index}(s)|_{\mathsf{proj}_1\,P}\right) && \text{(by Equation } (*))\\
&\approx \mathsf{map}^{\text{MAYBE}}\ \mathsf{value}\ \left(\mathsf{index}(s)|_{\mathsf{proj}_1\,P'}\right) && \text{(by rewriting } E)\\
&\approx s|_{\mathsf{proj}_1\,P'} && \text{(by Equation } (*))\\
&\equiv r(P') && \text{(by Def. 2.9 [Setoid structure on } \mathbb{P}])
\end{aligned}$$

where Equation $(*)$ is given by:

$$\forall (p : \text{PATH}).\ \forall s.\ \mathsf{map}^{\text{MAYBE}}\ \mathsf{value}\ \left(\mathsf{index}(s)|_p\right) \approx s|_p. \qquad (*)$$

The proof of $(*)$ is straightforward by path induction on p and by exploiting the fact that index is a section of value.

- $i : \mathbb{P}(s) \to \mathbb{P}(\mathsf{index}\, s)$. We define the map i as follows:

  $i : \mathbb{P}(s) \to \mathbb{P}(\mathsf{index}\, s)$
  $i\ \ (p, H)\ \equiv\ (p, g(H))$
  where $g : p \in \mathscr{P}(s) \to p \in \mathscr{P}(\mathsf{index}\, s)$.

  The function g is obtained by path induction on p by exploiting the fact that index is a section of value.

Finally, it remains to show that i is a right-inverse of r. Let $P : \mathbb{P}(s)$. Clearly, we have $r(i(P)) \approx s|_{\mathsf{proj}_1\, P} \approx P$ by reflexivity. □

We conclude with a proof akin to the closure property mentioned in the introduction.

Theorem 4 **Coalgebra structure on FSUCC**

Let $T : \mathcal{U}$ be a transition system. The type FSUCC_T has an $\mathcal{U}$-coalgebra structure.

Proof. We define the two projection functions as follows:

$\mathsf{rank} : \text{FSUCC}_T \to \mathbb{N}$
$\mathsf{rank}\ \equiv\ \mathsf{rank}_T \circ \mathsf{proj}_1,$

$\mathsf{next} : \forall (s : \text{FSUCC}_T).\ \text{FIN}(\mathsf{rank}\, s) \to \text{FSUCC}_T$
$\mathsf{next}\ \ (s, F)\ \ k\ \equiv\ \mathsf{next}_T\ s\ k, F'$
where $F' : \text{SUCCFINITE}(\mathsf{next}_T\ s\ k)$.

It remains to give the proof F' that the successor closure of a successor state is finite. Let $\sigma : \text{FSUCC}_T$ be a designated state such that we have a proof $F : \text{SUCCFINITE}\ \sigma$. We call $s' := \mathsf{next}_T\ \sigma\ k$, the underlying successor state of σ for the kth transition. To prove that the successor closure of s' is finite, we proceed as follows:

(*i*) by Prop. 2.13, we have $\text{SUCCFINITE}(\mathsf{index}\, s')$ and since $\mathbb{P}(\mathsf{index}\, s') \cong (\mathsf{index}\, s')_{\rightsquigarrow}$ by Thm. 2.3, we obtain that $\text{WFI}(\mathbb{P}(\mathsf{index}\, s'))$,

(*ii*) similarly, we have the isomorphism $\mathbb{P}(s') \cong \text{WFI}(s'_{\rightsquigarrow})$, thus to prove $\text{SUCCFINITE}(s')$, it suffices to show that $\text{WFI}(\mathbb{P}(s'))$,

(*iii*) finally, since $\mathbb{P}(\mathsf{index}\, s')$ is a weak retract of $\mathbb{P}(s')$ by Lem. 2.1 and that $\text{WFI}(\mathbb{P}(\mathsf{index}\, s'))$, we conclude that $\text{WFI}(\mathbb{P}(s'))$.

To summarize,

$$\text{WFI}(\mathsf{index}\, s') \xrightarrow{(i)} \text{WFI}(\mathbb{P}(\mathsf{index}\, s')) \xrightarrow{(iii)} \text{WFI}(\mathbb{P}(s')) \xrightarrow{(ii)} \text{WFI}(s'_{\rightsquigarrow}). \qquad \square$$

2.1.5. Regular Trees

In this section, we instantiate the previous definitions to the type of infinite trees. In addition, we use a slightly different terminology better suited to trees. From these, we derive a formal definition of the type of regular trees and prove the closure property.

Proposition 14 **Infinite tree $\mathcal{U}$-coalgebra**

Let S be a signature. The type $\text{co}\mathbb{T}_S$ of infinite trees has an $\mathcal{U}$-coalgebra structure.

Proof. We consider the setoid $(\text{co}\mathbb{T}_S, _\sim_)$, where $_\sim_$ denotes the bisimilarity relation. We define the two projection functions:

$$\begin{aligned} \mathsf{rank} &: \text{co}\mathbb{T}_S \to \mathbb{N} \\ \mathsf{rank} &\equiv S.\mathsf{Ar} \circ \mathsf{root}, \end{aligned} \qquad \begin{aligned} \mathsf{next} &: \forall (t : \text{co}\mathbb{T}_S).\ \text{FIN}(\mathsf{rank}\, t) \to \text{co}\mathbb{T}_S \\ \mathsf{next} &\equiv \mathsf{br}. \end{aligned}$$

The proofs that both rank and next respect bisimilarity are straightforward. □

Definition 14 **Regular tree**

Let S be a signature. A tree $t : \text{co}\mathbb{T}_S$ is said to be regular when the type of its subtrees is finite. Formally,

$$\text{REGULAR}(t) := \text{SUCCFINITE}(t).$$

The type of all regular trees over the signature S, denoted REG_S, is given by:

$$\text{REG}_S := \text{FSUCC}_{\text{co}\mathbb{T}_S}.$$

The following theorem is important since it establishes that the regularity property is preserved by observations on infinite trees.

Theorem 5 **Regularity property closure**

Let S be a signature and $t : \text{REG}_S$ be a regular tree. For any tree s, if s is a subtree of t, then s is regular.

Proof. Let $t : \text{REG}_S$ be a regular tree, $s : \text{co}\mathbb{T}_S$ be a tree and $P : t \rightsquigarrow s$ a proof that s is a subtree of t. We proceed by induction on P (Prop. 2.8):

▸ **Base step.** Assume that $e : t \sim s$.
It straightforward to show that $\text{REGULAR}(_)$ preserves the underlying setoid equality.

▸ **Inductive step.** Assume that $p : t[k] \rightsquigarrow s$ where $k : \text{FIN}(\mathsf{rank}\, t)$.
The induction hypothesis is given by:

$$\text{REGULAR}(t[k]) \to \text{REGULAR}(s). \tag{IH}$$

From induction hypothesis (IH), to prove $\text{REGULAR}(s)$, it suffices to show that $\text{REGULAR}(t[k])$. From Theorem 2.4 [Coalg. on FSUCC], we can conclude that $\text{REGULAR}(t[k])$ holds since $t[k]$ is a successor state of t which is regular by assumption. □

Now, we consider two examples. The first one is a proof that an infinite tree is regular. In particular, this shows that all definitions given thus far can be instantiated with concrete values. The second example is a proof that the stream ω of the sequence of natural numbers is not regular.

Example of an infinite regular tree. Here, we show how to that an infinite tree is regular. First, we consider the following signature:

$$S := \{\, \bot^{(0)}; \triangle^{(1)}; \square^{(2)} \,\}$$

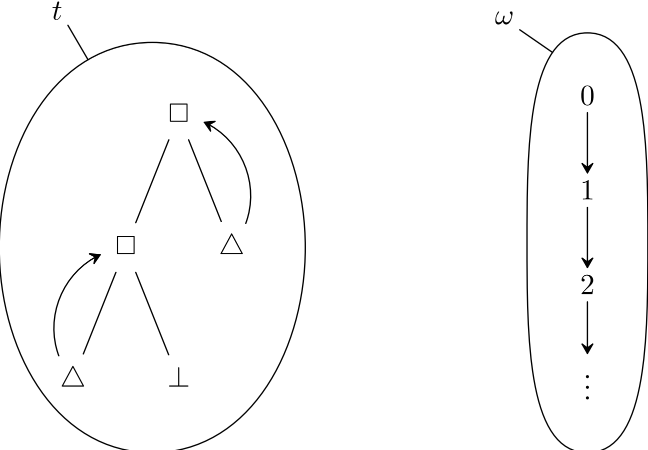


Figure 2.4. Examples of a regular tree t and of a non-regular stream ω

where $\bot$ designates a leaf constructor, $\triangle$ is a unary constructor and $\square$ is a binary constructor. We follow the convention that the left-argument of $\square$ is indexed by zero and the right-argument is indexed by suc zero.

Next, we consider the following infinite tree (Figure 2.4) defined corecursively as follows:

$$t : \text{CO}\mathbb{T}_S$$
$$t \equiv \square\ t'\ (\triangle\ t)$$
$$\quad\textbf{where}\ \ t' : \text{CO}\mathbb{T}$$
$$\qquad\quad t' \equiv \square\ (\triangle\ t')\ \bot.$$

In order to show that t is regular, we how to prove that the set of subtrees of t can be indexed by a finite type. All of these subtrees are represented in Figure 2.5. Since the tree t has two cycles, we proceed compositionally. First, we show that $\text{SuccFinite}(t')$. By Theorem 2.3, it suffices to show that the setoid $\mathbb{P}(t')$ is weakly finitely indexed. To this end, we define the indexing map:

$$\mathsf{index}_{t'} : \text{Path} \to \text{Fin}\,3$$
$$\mathsf{index}_{t'}\ \ \varepsilon \equiv \mathsf{zero}$$
$$\mathsf{index}_{t'}\ \ ((2,\mathsf{zero}) :: \varepsilon) \equiv \mathsf{suc\ zero}$$
$$\mathsf{index}_{t'}\ \ ((2,\mathsf{zero}) :: (1,\mathsf{zero}) :: p') \equiv \mathsf{index}_{t'}(p')$$
$$\mathsf{index}_{t'}\ \ _ \equiv \mathsf{suc\ zero}$$

while the value map is given by:

$$\mathsf{value}_{t'} : \text{Fin}\,3 \to \mathbb{P}(t')$$
$$\mathsf{value}_{t'}\ \ \mathsf{zero} \equiv \varepsilon, _$$
$$\mathsf{value}_{t'}\ \ (\mathsf{suc\ zero}) \equiv (2,\mathsf{zero}) :: \varepsilon, _$$
$$\mathsf{value}_{t'}\ \ _ \equiv (2,\mathsf{suc\ zero}) :: \varepsilon, _.$$

Note that, the domain of $\mathsf{index}_{t'}$ is the type Path rather than $\mathbb{P}(t')$. This is due to the fact that the proof that the path is in t' is not required to produce an index, thus yielding a slightly simpler definition. In the definition of $\mathsf{value}_{t'}$, we have omitted the proofs witnessing the fact that the paths are in the tree t' because they can be inferred automatically. Finally, it remains to show that $\mathsf{index}_{t'}$ is a section of $\mathsf{value}_{t'}$:

$$\forall (p : \mathbb{P}(t')).\ \mathsf{value}_{t'}(\mathsf{index}_{t'}(\mathsf{proj}_1\, p)) \approx p.$$

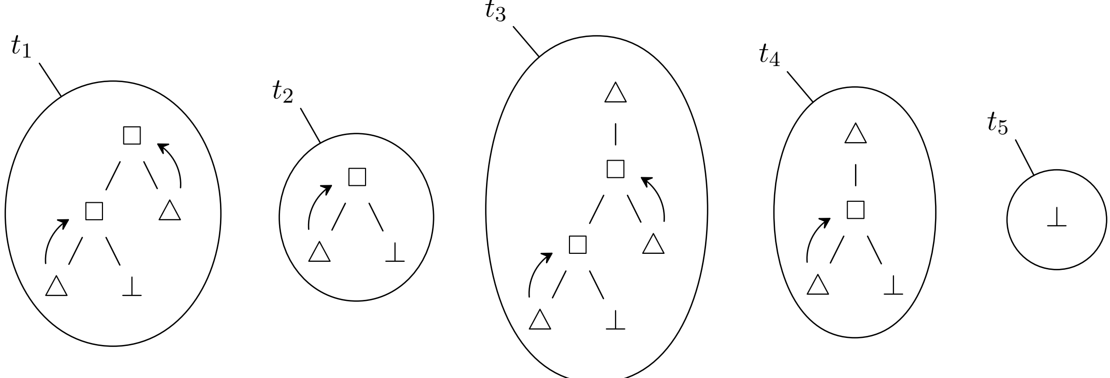


Figure 2.5. Subtrees of the infinite tree t up-to bisimilarity

By Definition 2.9, this is equivalent to:

$$\forall (p : \text{PATH}).\ p \in \mathscr{P}(t') \to t'|_{\text{val-idx}\,p} \approx t'|_p. \qquad (\diamond)$$

where $\text{val-idx} := \text{proj}_1 \circ \text{value}_{t'} \circ \text{index}_{t'}$. We prove Equation $(\diamond)$ by *recursion* on the path p:

- **Case 1.** Assume that $p = \varepsilon$.

  We have $t'|_{\text{val-idx}\,\varepsilon} \equiv t'|_{\text{proj}_1(\text{value}_{t'}\ \text{zero})} \approx t'|_\varepsilon$ by reflexivity.

- **Case 2.** Assume that $p = (2, \text{zero}) :: \varepsilon$.

  We have $t'|_{\text{val-idx}((2,\text{zero}) :: \varepsilon)} \equiv t'|_{\text{proj}_1(\text{value}_{t'}(\text{suc zero}))} \approx t'|_{(2,\text{zero}) :: \varepsilon}$ by reflexivity.

- **Case 3.** Assume that $p = (2, \text{zero}) :: (1, \text{zero}) :: p'$ and that $H : p \in \mathscr{P}(t')$.

  We have $t'|_{\text{val-idx}((2,\text{zero}) :: (1,\text{zero}) :: p')} \equiv t'|_{\text{val-idx}\,p'}$. In addition, it is straightforward to show that from H, we can derive a proof $H' : p' \in \mathscr{P}(t')$. Then, by recursion on p', we obtain a proof that $t'|_{\text{val-idx}\,p'} \approx t'|_{p'}$. Consequently, we have $t'|_{p'} \approx t'|_{(2,\text{zero}) :: (1,\text{zero}) :: p'}$ by reflexivity.

- **Case 4.** Assume that p is none of the above cases and that $H : p \in \mathscr{P}(t')$.

  From the proof H, it is straightforward to show that $p = (1, \text{zero}) :: \varepsilon$. Intuitively, $(1, \text{zero})$ is the only *valid* prefix of the path that we have not seen thus far. Then, we have $t'|_{\text{val-idx}((1,\text{zero}) :: \varepsilon)} \equiv t'|_{\text{proj}_1(\text{value}_{t'}(\text{suc zero}))} \approx \bot \approx t'|_{(1,\text{zero}) :: \varepsilon}$.

Finally, we prove that t is regular. The proof follows the same structure than the previous one, showing that t' is regular. We define the indexing map as follows:

$$\begin{array}{lll}
\text{index}_t : \text{PATH} \to \text{FIN}\,5 & & \\
\text{index}_t\ \ ((2, \text{zero}) :: p) & \equiv & \text{suc}(\text{suc}(\text{index}_{t'}(p))) \\
\text{index}_t\ \ ((2, \text{suc zero}) :: \varepsilon) & \equiv & \text{suc zero} \\
\text{index}_t\ \ ((2, \text{suc zero}) :: (1, \text{zero}) :: p') & \equiv & \text{index}_t(p') \\
\text{index}_t\ \ _ & \equiv & \text{zero}
\end{array}$$

while the value map is given by:

$$\begin{array}{lll}
\text{value}_t : \text{FIN}\,5 \to \mathbb{P}(t) & & \\
\text{value}_t\ \ \text{zero} & \equiv & \varepsilon, _ \\
\text{value}_t\ \ (\text{suc zero}) & \equiv & (2, \text{suc zero}) :: \varepsilon, _ \\
\text{value}_t\ \ (\text{suc}(\text{suc}\,i)) & \equiv & (2, \text{zero}) :: \text{proj}_1(\text{value}_{t'}(i)), _.
\end{array}$$

The proof that index_t is a section of value_t is omitted since it follows the same structure than the proof establishing that $\mathsf{index}_{t'}$ is a section of $\mathsf{value}_{t'}$.

Example of an infinite non-regular tree. In the previous paragraph, we gave an example of a constructive proof of the regularity property of a tree. Now, we consider the opposite problem, *i.e.*, proving that an infinite tree is not regular. For that, we consider a simple case—the stream ω. This stream can be defined corecursively as follows:

$f : \mathbb{N} \to \textsc{Stream}\ \mathbb{N}$ $\qquad\qquad$ $\omega : \textsc{Stream}\ \mathbb{N}$
$f \;\equiv\; n :: f(\mathsf{suc}\, n),$ $\qquad\qquad$ $\omega \equiv f_0.$

To prove that ω is not regular, we assume that this is the case, and derive a contradiction. Assume that we have a proof $H : \textsc{Regular}(\omega)$. By Proposition 1.15, the type $\omega\!\rightsquigarrow$ is also streamless. Then, we define the following stream of ω-successors:

$h : \mathbb{N} \to \omega\!\rightsquigarrow$
$h\ \ n \;\equiv\; f(n), \mathsf{lemma}\, n$
$\quad$ **where** $\mathsf{lemma} : \forall (n : \mathbb{N}).\ \omega \rightsquigarrow f(n).$

The proof of lemma is given by induction on n:

- **Base step.** Assume that $n = \mathsf{zero}$.
  We have $\omega \equiv f(0) \rightsquigarrow f(0)$ by ε (Prop. 2.10, reflexivity of $\rightsquigarrow$).
- **Inductive step.** Assume that $n = \mathsf{suc}\, n'$ where $n' : \mathbb{N}$.
  The induction hypothesis is given by

$$\omega \rightsquigarrow f(n). \tag{IH}$$

  We have to show that $\omega \rightsquigarrow f(\mathsf{suc}\, n)$. We use the transitivity of $\rightsquigarrow$ (Prop. 2.10) with $f(n)$. Thus, we have $\omega \rightsquigarrow f(n)$ by the induction hypothesis (IH) and $f(n) \rightsquigarrow f(\mathsf{suc}\, n)$ by next.

Since $\omega\!\rightsquigarrow$ is streamless, there exist two positions i, j such that $i \neq j$ and $h(i) \approx h(j)$. By definition of the setoid equality (see Definition 2.11), we have $f(i) \sim f(j)$, where $\sim$ denotes bisimilarity. By elimination of $f(i) \sim f(j)$, we have, in particular, that $\mathsf{root}(f(i)) = i = j = \mathsf{root}(f(j))$. This contradicts the assumption $i \neq j$.

2.2. A SYNTAX FOR REGULAR TREES

In the previous section, we have showed with an example how to prove that an infinite tree is regular. However, such proof can be quite involved since it consists in listing every subtree of given tree. Instead of following this approach, we aim at defining a syntax that will characterize *all regular trees by construction.*

2.2.1. Cyclic Terms

The key idea leading to the definition of the syntax for regular trees is illustrated in Figure 2.6. On the left, observe that the tree contains an *explicit* representation of cycles. These cycles were represented semantically through corecursive equations (see Section 1.4), thus exploiting the ability of coinductive datatypes to describe infinite terms. Note that, if we ignore these cycles, we are left with a spanning tree which could be represented *inductively*. Consequently, we have to find a way to represent these cycles within this inductive representation. To accomplish this, we follow the approach developed in [GHUV06] which implements implicit cycles by means of binders. In Figure 2.6, the additions made to the right tree to account for the representation of cycles are highlighted. Here, both rec_x and rec_y denote binders while x and y represent the bound variables.

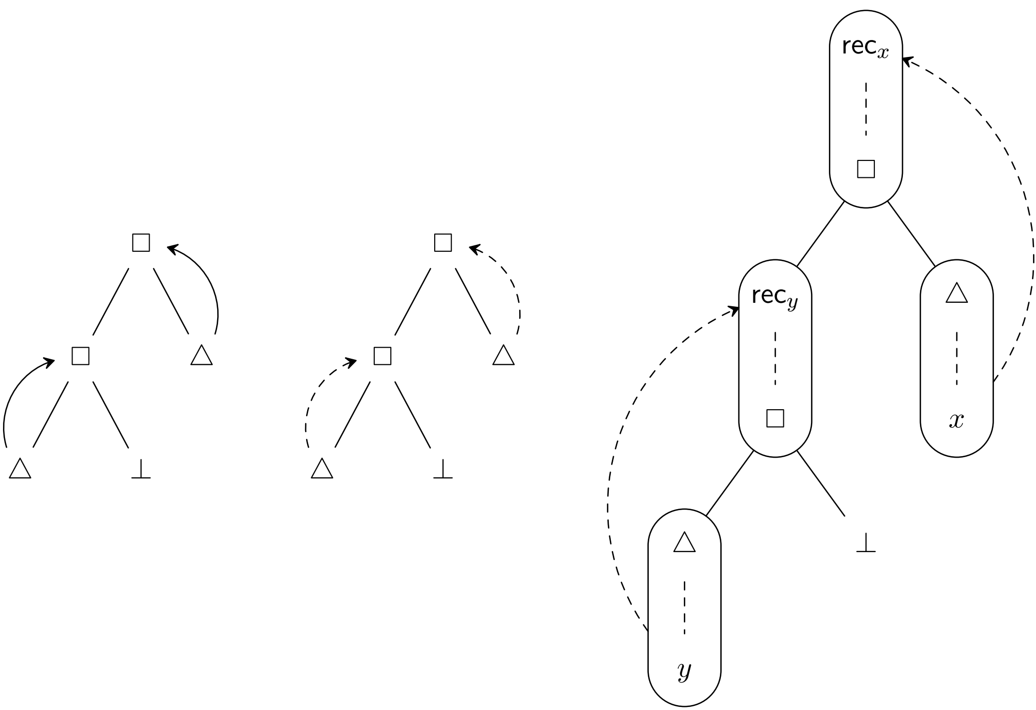


Figure 2.6. Various representation of cycles for a regular tree.

Binder representation

Mechanizing the meta-theory of programming languages is still an active subject of research. One of the main topic concerns the representation of binder operators. This problem led to the POPLMARK challenge. Various solutions were proposed such as Higher-Order Abstract Syntax [PE88], Nominal logics [Pit01], nameless representation with de Bruijn indices [dB72] or hybrid approaches using both named and nameless representation such as the locally nameless representation [ACP+08].

Binders encoding based on a nameless representation are important because there are by essence *canonical* and thus, α-equivalence on terms coincides with Leibniz equality. Another reason is that nameless representations can be conveniently encoded as a *heterogeneous type*, *i.e.*, as a type family indexed by the number of free variables [AR99]. One important property of such encoding is that this type family possesses the structure of a monad, and thus is equipped with a substitution operation with good algebraic properties. For example, the untyped lambda-calculus can be a represented as a heterogeneous inductive type (as presented in [AR99]) as follows:

inductive $\mathrm{LC}\ (n : \mathbb{N}) : \mathrm{TYPE}$
 var $: \mathrm{FIN}\, n \to \mathrm{LC}(n)$
 $_\cdot_ : \mathrm{LC}(n) \to \mathrm{LC}(n) \to \mathrm{LC}(n)$
 $\lambda : \mathrm{LC}(\mathsf{suc}\, n) \to \mathrm{LC}(n).$

The type LC is indexed by the number of free variables. The heterogeneity of LC is observed on the λ-abstraction constructor since the type of its argument is given by $\mathrm{LC}(\mathsf{suc}\, n)$—a lambda-term with an additional free variable. Given a lambda term t with $n+1$ variables constructing the term λt reduces the number of free variables by one. This has the effect of turning a *free variable* in t into a *bound variable*. Following the convention of de Bruijn indices, bound variables are thus indexed by the number of enclosing λ-abstractions.

Remark. The lambda term $\lambda f.\ \lambda g.\ g(f(x))$ is represented as $\lambda\lambda(\mathsf{var}\,0 \cdot (\mathsf{var}\,1 \cdot \mathsf{var}\,2))$. In a locally nameless style—where both free and bound variables have a distinct encoding—the term is represented as $\lambda\lambda\,\mathsf{bvar}\,0 \cdot (\mathsf{bvar}\,1 \cdot \mathsf{fvar}\,x)$, where bvar are bound variables and fvar are free variables.

Inductive Definition

Now, we introduce the type of cyclic terms over a base signature. Here, we follow the encoding presented in [GHUV06] as a mean to represent cycles in a spanning tree (defined inductively).

Definition 15 **Cyclic term**

The type of cyclic term, denoted $\mathbb{C}$, is defined inductively as an heterogeneous datatype as follows:

$$\textbf{inductive}\ \mathbb{C}_\ (S : \textsc{Sig})\ (n : \mathbb{N}) : \textsc{Type}$$
$$\mid\ \mathsf{var} : \textsc{Fin}\,n \to \mathbb{C}_S(n)$$
$$\mid\ \mathsf{rec}\,_ \lhd _ : (o : S.\mathsf{Op}) \to \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\,n))\ (\textsc{Fin}(S.\,\mathsf{Ar}\,o)) \to \mathbb{C}_S(n).$$

Cyclic terms are inductively defined by only two constructors. The constructor $\mathsf{rec}\,o \lhd os$ builds a term with a function symbol o and a vector of subterms given by os. Here rec marks a position within the cyclic term that can be later referenced by a bound variable—introduced with the var constructor. A cyclic term does not contain free variables is said to be *closed*. In addition, for closed terms, we may write $\mathbb{C}_S$ instead of $\mathbb{C}_S(0)$. For instance, the closed cyclic term represented in Figure 2.6 is defined as follows:

$$\begin{aligned}
&\mathsf{rec}\,\Box \lhd [\,\mathsf{rec}\,\Box \lhd\ [\,\mathsf{rec}\,\triangle \lhd [\,\mathsf{var}(\mathsf{suc\ zero})\,]\\
&\qquad\qquad\qquad ;\ \mathsf{rec}\,\bot \lhd [\,]\,]\\
&\qquad ;\ \mathsf{rec}\,\triangle \lhd\ [\,\mathsf{var}(\mathsf{suc\ zero})\,]\,] : \mathbb{C}_{\{\,\bot^{(0)};\triangle^{(1)};\Box^{(2)}\,\}}.
\end{aligned}$$

In this example, the signature is written "inline", that is, the operators are given by $\bot$, $\triangle$ and $\Box$ with an arity of 0, 1 and 2 respectively. In order to avoid redundant usage of rec when it is not further referenced by a bound variable, we introduce the following function:

$$\begin{aligned}
&_ \lhd _ : \{n : \mathbb{N}\} \to (o : S.\mathsf{Op}) \to \textsc{Vec}\ (\mathbb{C}_S(n))\ (S.\,\mathsf{Ar}\,o) \to \mathbb{C}_S(n)\\
&o\ \lhd\ os\ \equiv\ \mathsf{rec}\,o \lhd \mathsf{map}^{\textsc{Vec}}\ \mathsf{weaken}\ os.
\end{aligned}$$

The difference between $\mathsf{rec}\,_ \lhd _$ and $_ \lhd _$ is the number of free variables appearing in the vector of arguments. In the first case, since $\mathsf{rec}\,_ \lhd _$ is a binder, the vector of arguments contains an additional free variable. However, in the second case, the number of free variables in the same for the vector of arguments and the resulting term. Thus, in order to define $_ \lhd _$ from $\mathsf{rec}\,_ \lhd _$ each element contained in the vector of arguments *has to be weakened*. Intuitively, weakening is an operation on terms consisting in shifting every variable by one. This operation is formally defined in the next section. With this function, the previous term can be written more compactly as:

$$\begin{aligned}
&\mathsf{rec}\,\Box \lhd [\,\mathsf{rec}\,\Box \lhd [\,\triangle \lhd [\,\mathsf{var\ zero}\,]\\
&\qquad\qquad\qquad ;\ \bot \lhd [\,]\,]\\
&\qquad ;\ \triangle \lhd\ [\,\mathsf{var\ zero}\,]\,] : \mathbb{C}_{\{\,\bot^{(0)};\triangle^{(1)};\Box^{(2)}\,\}}.
\end{aligned}$$

In the inductive type definition of cyclic terms, $\mathbb{C}_S(n)$ is nested within Vec—type of the constructor $\mathsf{rec}\,_ \lhd _$. However, Coq fails to infer an adequate induction principle that takes into account the nesting of inductive types. Consequently, we prove a more general induction principle.

Proposition 15 Induction principle on cyclic term

Let $P : \forall(n : \mathbb{N}).\ \mathbb{C}_S(n) \to \textsc{Type}$ be a type family on cyclic terms such that:

$$P_{\mathsf{var}}\ \frac{n : \mathbb{N} \qquad x : \textsc{Fin}\, n}{P(\mathsf{var}\, x)}, \qquad P_{\mathsf{rec}}\ \frac{n : \mathbb{N} \qquad o : S.\mathsf{Op} \qquad os : \textsc{Vec}\ (\mathbb{C}(\mathsf{suc}\, n))\ (S.\mathsf{Ar}\, o) \qquad Q(os)}{P(\mathsf{rec}\, o \lhd os)},$$

and let $Q : \forall\{m, n : \mathbb{N}\}.\ \textsc{Vec}\ (\mathbb{C}_S(m))\ n \to \textsc{Type}$ be a type family such that:

$$Q_{[]}\ \frac{m : \mathbb{N}}{Q\ \{m\}\ []}, \qquad Q_{::}\ \frac{m, n : \mathbb{N} \qquad t : \mathbb{C}_S(m) \qquad ts : \textsc{Vec}\ (\mathbb{C}_S(m))\ n \qquad P(t) \qquad Q(ts)}{Q(t :: ts)}.$$

Then, we have:

$$\mathbb{C}_{\mathsf{rec}}\ \frac{n : \mathbb{N} \qquad t : \mathbb{C}_S(n)}{P(t)}.$$

Proof. The proof is given by the following recursive function:

$$\begin{array}{lll}
\mathbb{C}_{\mathsf{rec}} : \{n : \mathbb{N}\} \to (t : \mathbb{C}_S(n)) \to P(t) & & \\
\mathbb{C}_{\mathsf{rec}}\ \ (\mathsf{var}\, x) & \equiv & P_{\mathsf{var}}(x) \\
\mathbb{C}_{\mathsf{rec}}\ \ (\mathsf{rec}\, o \lhd os) & \equiv & P_{\mathsf{rec}}(\textsc{Vec}^{\mathbb{C}}_{\mathsf{rec}}(os)) \\
\quad \textbf{where}\ \ \textsc{Vec}^{\mathbb{C}}_{\mathsf{rec}} : \forall(m, n : \mathbb{N}).\ \forall(v : \textsc{Vec}\ (\mathbb{C}_S(m))\ n).\ Q(v) & & \\
\quad\quad \textsc{Vec}^{\mathbb{C}}_{\mathsf{rec}}\ \ [] & \equiv & Q_{[]} \\
\quad\quad \textsc{Vec}^{\mathbb{C}}_{\mathsf{rec}}\ \ (t :: ts) & \equiv & Q_{::}\ (\mathbb{C}_{\mathsf{rec}}(t))\ (\textsc{Vec}^{\mathbb{C}}_{\mathsf{rec}}(ts)).
\end{array}$$

Note that each recursive call is on a subterm of the input. □

In the following proposition, we prove an induction principle akin to the one that would have been generated by Coq if the argument of the constructor $\mathsf{rec}\, _ \lhd _$ was as a finite map rather than a vector.

Proposition 16 Alternative induction principle on cyclic terms

Let $P : \forall(n : \mathbb{N}).\ \mathbb{C}_S(n) \to \textsc{Type}$ be a type family on cyclic terms such that:

$$P_{\mathsf{var}}\ \frac{n : \mathbb{N} \qquad x : \textsc{Fin}\, n}{P(\mathsf{var}\, x)}, \qquad P_{\mathsf{rec}}\ \frac{n : \mathbb{N} \qquad o : S.\mathsf{Op} \qquad os : \textsc{Vec}\ (\mathbb{C}(\mathsf{suc}\, n))\ (S.\mathsf{Ar}\, o) \qquad \forall k.\ P(os(k))}{P(\mathsf{rec}\, o \lhd os)}.$$

Then, we have:

$$\mathbb{C}_{\mathsf{rect}}\ \frac{n : \mathbb{N} \qquad t : \mathbb{C}_S(n)}{P(t)}.$$

Proof. We use the induction principle of Prop. 2.15 with $Q := \lambda(m, n : \mathbb{N}).\ \lambda v.\ \forall k.\ P(v(k))$. The first two premises for P are actually true by assumption. Then, it remains to show the ones for Q:

- $\forall(m : \mathbb{N}).\ \forall(k : \textsc{Fin}\, \mathsf{zero}).\ P([](k))$.

  The type $\textsc{Fin}\, \mathsf{zero}$ is empty. Contradiction.

- $\forall m, n.\ \forall(t : \mathbb{C}_S(m)).\ \forall(os : \textsc{Vec}\ _\ n).\ P(t) \to (\forall k.\ P(os(k))) \to \forall k.\ P((t :: os)(k))$.

  Let $H : P(t), H' : \forall k.\ P(os(k))$ and $k : \textsc{Fin}(\mathsf{suc}\, n)$. By case-analysis on k:

  - **Case 1.** Assume that $k = \mathsf{zero}$.

    We have to show that $P\ (t :: os)\ \mathsf{zero}$ which reduces to $P(t)$. Then, we conclude with H.

  - **Case 2.** Assume that $k = \mathsf{suc}\, k'$ where $k' : \textsc{Fin}\, n$.

    We have to that $P\ (t :: os)\ (\mathsf{suc}\, k')$ which reduces to $P(os(k'))$. We conclude with H'. □

2.2.2. Syntactic Properties on Cyclic Terms

In this section, we prove some technical results about cyclic terms. Ultimately, this consists in proving that the type $\mathbb{C}_S : \mathbb{N} \to \text{TYPE}$ is a monad. A categorical treatment of syntax with binders can be found in [Ahr15]. In the following, we summarize the various syntactic properties on cyclic terms. The proofs of all these results are detailed in Appendix B. For the remaining of this section, we fix an arbitrary signature S.

Renaming operation

The first operation on cyclic terms, called *renaming*, establishes the functoriality of $\mathbb{C}_S : \mathbb{N} \to \text{TYPE}$.

$$\begin{aligned}
&\mathsf{rename} : (\text{FIN}\, m \to \text{FIN}\, n) \to \mathbb{C}_S(m) \to \mathbb{C}_S(n)\\
&\mathsf{rename}\ \ \rho\ \ (\mathsf{var}\, x) \quad\ \ \equiv\ \mathsf{var}(\rho(x))\\
&\mathsf{rename}\ \ \rho\ \ (\mathsf{rec}\, o \lhd os)\ \equiv\ \mathsf{rec}\, o \lhd \mathsf{map}^{\text{VEC}}\ (\mathsf{rename}(\mathsf{shift}\, \rho))\ os.
\end{aligned}$$

Due to the heterogeneity in the type of the constructor $\mathsf{rec}_ \lhd _$, the renaming function f needs to be shifted, in the following way:

$$\begin{aligned}
&\mathsf{shift} : (\text{FIN}\, m \to \text{FIN}\, n) \to \text{FIN}(\mathsf{suc}\, m) \to \text{FIN}(\mathsf{suc}\, n)\\
&\mathsf{shift}\ \ f\ \ \mathsf{zero} \quad\ \ \equiv\ \mathsf{zero}\\
&\mathsf{shift}\ \ f\ \ (\mathsf{suc}\, m)\ \equiv\ \mathsf{suc}(f(m)).
\end{aligned}$$

Identity.

$$\forall t.\ \mathsf{rename}\ \mathsf{id}\ t = t. \qquad \text{(RENAME-ID)}$$

Composition.

$$\begin{aligned}
&\forall (f : \text{FIN}\, m \to \text{FIN}\, n).\ \forall (g : \text{FIN}\, n \to \text{FIN}\, p).\ \forall (t : \mathbb{C}_S(m)).\\
&\qquad \mathsf{rename}\ (g \circ f)\ t = \mathsf{rename}\ g\ (\mathsf{rename}\ f\ t).
\end{aligned} \qquad \text{(RENAME-}\circ\text{)}$$

Congruence.

$$\begin{aligned}
&\forall (f, g : \text{FIN}\, m \to \text{FIN}\, n).\ \forall (t : \mathbb{C}_S(m)).\\
&\qquad (\forall x.\ f(x) = g(x)) \to \mathsf{rename}\ f\ t = \mathsf{rename}\ g\ t.
\end{aligned} \qquad \text{(RENAME-CONG)}$$

Finally, we define the weakening operation as follows:

$$\begin{aligned}
&\mathsf{weaken} : \mathbb{C}_S(n) \to \mathbb{C}_S(\mathsf{suc}\, n)\\
&\mathsf{weaken}\ \equiv\ \mathsf{rename}\ \mathsf{suc}\,.
\end{aligned}$$

Substitution operation

The second operation on cyclic terms, called *substitution*, establishes that $\mathbb{C}_S$ is a monad.

$$\begin{aligned}
&\mathsf{subst} : (\text{FIN}\, m \to \mathbb{C}_S(n)) \to \mathbb{C}_S(m) \to \mathbb{C}_S(n)\\
&\mathsf{subst}\ \ \sigma\ \ (\mathsf{var}\, x) \quad\ \ \equiv\ \sigma(x)\\
&\mathsf{subst}\ \ \sigma\ \ (\mathsf{rec}\, o \lhd os)\ \equiv\ \mathsf{rec}\, o \lhd \mathsf{map}^{\text{VEC}}\ (\mathsf{subst}(\mathsf{lift}\, \sigma))\ os.
\end{aligned}$$

Due to the heterogeneity in the type of the constructor $\mathsf{rec}_ \lhd _$, the substitution function σ needs to be lifted, in the following way:

$$\begin{aligned}
&\mathsf{lift} : (\text{FIN}\, m \to \mathbb{C}_S(n)) \to \text{FIN}(\mathsf{suc}\, m) \to \mathbb{C}_S(\mathsf{suc}\, n)\\
&\mathsf{lift}\ \ f\ \ \mathsf{zero} \quad\ \ \equiv\ \mathsf{var}\,\mathsf{zero}\\
&\mathsf{lift}\ \ f\ \ (\mathsf{suc}\, m)\ \equiv\ \mathsf{rename}\ \mathsf{suc}\ (f(m)).
\end{aligned}$$

The substitution operation satisfies the following monad laws:

Identity.

$$\forall(f : \text{Fin}\, m \to \mathbb{C}_S(n)).\ \forall(x : \text{Fin}\, m).\ \mathsf{subst}\ f\ (\mathsf{var}\, x) = f(x). \qquad (\text{subst-var})$$

Right-identity.

$$\forall(t : \mathbb{C}_S(m)).\ \mathsf{subst}\ \mathsf{var}\ t = t. \qquad (\text{subst-}\eta)$$

Associativity.

$$\begin{aligned}&\forall(f : \text{Fin}\, m \to \mathbb{C}_S(n)).\ \forall(g : \text{Fin}\, n \to \mathbb{C}_S(p)).\ \forall(t : \mathbb{C}_S(m)).\\ &\quad \mathsf{subst}\ g\ (\mathsf{subst}\ f\ t) = \mathsf{subst}\ (\mathsf{subst}\, g \circ f)\ t.\end{aligned} \qquad (\text{subst-subst})$$

Congruence.

$$\begin{aligned}&\forall(f, g : \text{Fin}\, m \to \mathbb{C}_S(n)).\ \forall(t : \mathbb{C}_S(m)).\\ &\quad (\forall x.\ f(x) = g(x)) \to \mathsf{subst}\ f\ t = \mathsf{subst}\ g\ t.\end{aligned} \qquad \text{subst-cong}$$

The one-variable substitution operation is defined as follows:

```
_[∗ := _] : ℂ_S(suc m) → ℂ_S(m) → ℂ_S(m)
 t[∗ := u]  ≡  subst σ_u t
     where  σ_ : ℂ_S(m) → Fin(suc m) → ℂ_S(m)
            σ_u  zero      ≡  u
            σ_u  (suc x)   ≡  var x.
```

This substitution operation is particularly useful to substitute a newly introduced free variable in a term. Moreover, the one-variable substitution satisfies the following laws:

Weakening and one-variable substitution.

$$\forall(t, u : \mathbb{C}_S(m)).\ \mathsf{weaken}(t)[* := u] = t. \qquad (\text{weaken-subst}_*)$$

Substitution and one-variable substitution commute.

$$\forall(t : \mathbb{C}_S(m)).\ \mathsf{subst}\ f\ (t[* := u]) = (\mathsf{subst}\ (\mathsf{lift}\, f)\ t)[* := \mathsf{subst}\ f\ u]. \qquad (\text{subst-subst}_*)$$

2.2.3. Semantics of Cyclic Terms

Cyclic terms will be interpreted as infinite trees. To this end, we introduce the type of closures:

```
inductive Closure (S : Sig) : ℕ → Type
  | [] : Closure_S(zero)
  | _::_ : ∀(n : ℕ). ℂ_S(n) → Closure_S(n) → Closure_S(suc n).
```

The type Closure is defined in such a way as to ensure that each new free variable may only refer to terms that are already in the closure. As a result, this guarantees that each free variable is *ultimately* defined by a closed term:

```
var-def : {n : ℕ} → Closure_S(n) → Fin n → ℂ_S
var-def  (t :: ρ)  zero      ≡  subst (var-def ρ) t
var-def  (t :: ρ)  (suc n)   ≡  var-def ρ n.
```

Note that the function var-def terminates since the number n of free variables decreases in-between each recursive call. Given a term and a closure, we obtain a closed term where each free variable has been substituted with its definition:

```
close-term : {n : ℕ} → ℂ_S(n) → Closure_S(n) → ℂ_S
close-term  t  ρ  ≡  subst (var-def ρ) t.
```

Now, we give two interpretation functions of cyclic terms as infinite trees based on two implementation strategies. The first one is based on substitution and operates on closed terms:

$$
\begin{aligned}
&[\![_]\!] : \mathbb{C}_S \to \text{co}\mathbb{T}_S \\
&[\![\ \ \mathsf{var}()\ \ ]\!] \\
&[\![\ \mathsf{rec}\, o \lhd os\ ]\!] \equiv o \blacktriangleleft \lambda k.\, [\![\, os(k)[* := \mathsf{rec}\, o \lhd os]\,]\!].
\end{aligned}
$$

The second interpretation function is defined for terms that may contain free variables while carrying their definitions within a closure. However, we have to deal with one technicality in order to define the function by guarded corecursion. When the term is a free variable, its definition has to be looked up within the closure in order to extract a function symbol to ensure productivity. Moreover, note that, nothing prevents the definition of a free variable to be again a free variable. Consequently, the process of looking up the definition of the variable has to be repeated until a term in guarded form is obtained. Though, this process is guaranteed to terminate eventually since closures are inductively defined terms. As a result, we define the interpretation function by *coiteration*, that is, by mapping S-coalgebras into the (weakly) final coalgebra $\text{co}\mathbb{T}_S$. The carrier of the S-coalgebra consists of pairs of terms in guarded form associated with a closure:

$$
\begin{aligned}
&\mathcal{G}_ : \textsc{Sig} \to \textsc{Type} \\
&\mathcal{G}_S \equiv \textstyle\sum_{n:\mathbb{N}} \sum_{o:S.\mathsf{Op}} \textsc{Vec}\ (\mathbb{C}_S(n))\ (S.\,\mathsf{Ar}\, o) \times \textsc{Closure}_S(n).
\end{aligned}
$$

while the morphism is defined as to inductively compute the next guarded term, by first looking up the definition of a free variable in the closure:

$$
\begin{aligned}
&\mathsf{lookup} : \{n : \mathbb{N}\} \to \textsc{Fin}\, n \to \textsc{Closure}_S(n) \to \mathcal{G}_S \\
&\mathsf{lookup}\ \ \mathsf{zero}\ \ \ (\mathsf{var}\, k :: \rho) \equiv \mathsf{lookup}\ k\ \rho \\
&\mathsf{lookup}\ \ \mathsf{zero}\ \ \ (\mathsf{rec}\, o \lhd os :: \rho) \equiv (_,\, o,\, os,\, \mathsf{rec}\, o \lhd os :: \rho) \\
&\mathsf{lookup}\ \ (\mathsf{suc}\, i)\ \ (_ :: \rho) \equiv \mathsf{lookup}\ i\ \rho.
\end{aligned}
$$

Finally, any term associated with a closure can be turned into a term in guarded form:

$$
\begin{aligned}
&\mathsf{to}\mathcal{G} : \{n : \mathbb{N}\} \to \mathbb{C}_S(n) \to \textsc{Closure}_S(n) \to \mathcal{G}_S \\
&\mathsf{to}\mathcal{G}\ \ (\mathsf{var}\, k)\ \ \ \rho \equiv \mathsf{lookup}\ k\ \rho \\
&\mathsf{to}\mathcal{G}\ \ (\mathsf{rec}\, o \lhd os)\ \ \rho \equiv (_,\, o,\, os,\, \mathsf{rec}\, o \lhd os :: \rho).
\end{aligned}
$$

The semantics of cyclic term as an infinite tree is thus obtained as follows:

$$
\begin{aligned}
&\mathcal{G}\text{-}\mathsf{coalg} : \mathcal{G}_S \to S(\mathcal{G}_S) \\
&\mathcal{G}\text{-}\mathsf{coalg}\ \ (m,\, o,\, os,\, \rho) \equiv (o,\, \lambda i.\, \mathsf{to}\mathcal{G}\ (os(i))\ \rho).
\end{aligned}
$$

$$
\begin{aligned}
&[\![_]\!]_ : \{n : \mathbb{N}\} \to \mathbb{C}_S(n) \to \textsc{Closure}_S(n) \to \text{co}\mathbb{T}_S \\
&[\![\, t\,]\!]_\rho \equiv \mathsf{coiterator}\ \mathcal{G}\text{-}\mathsf{coalg}\ (\mathsf{to}\mathcal{G}\ t\ \rho).
\end{aligned}
$$

Remark. With a more powerful termination checker such as the one implemented in the AGDA prover, the interpretation function could be written in a more natural style as follows:

$$
\begin{aligned}
&[\![_]\!]_ : \{n : \mathbb{N}\} \to \mathbb{C}_S(n) \to \textsc{Closure}_S(n) \to \text{co}\mathbb{T}_S \\
&[\![\ \mathsf{var}\ \mathsf{zero}\ ]\!]_{(t :: \rho)} \equiv [\![\, t\,]\!]_\rho \\
&[\![\ \mathsf{var}(\mathsf{suc}\, n)\ ]\!]_{(t :: \rho)} \equiv [\![\ \mathsf{var}\, n\,]\!]_\rho \\
&[\![\ \mathsf{rec}\, o \lhd os\ ]\!]_\rho \equiv o \blacktriangleleft \lambda k.\, [\![\, os(k)\,]\!]_{(\mathsf{rec}\, o \lhd os :: \rho)}.
\end{aligned}
$$

The subtlety is that this recursive definition is mixing induction and coinduction:

- When the term is a free variable, then its definition is looked up within the closure. This search is performed by induction on the closure.
- When the term is not a free variable, then the definition is coinductive and we have to produce a value to satisfy the productivity criterion.

In the following, we prove some technical results about the relationship of the close-term function and the substitution operation. Then, we show that the two semantics defined previously are equivalent, *i.e.*, they yield bisimilar infinite trees.

Proposition 17 **One-variable substitution in close-term**

Let $t : \mathbb{C}_S(\mathsf{suc}\, n)$, $u : \mathbb{C}_S(n)$ *be two cyclic terms and* $\rho : \textsc{Closure}_S(n)$ *be a closure. Then,*

$$\mathsf{close\text{-}term}\ (t[* := u])\ \rho = \mathsf{close\text{-}term}\ t\ (u :: \rho).$$

Proof. Let $t : \mathbb{C}_S(\mathsf{suc}\, n)$, $u : \mathbb{C}_S(n)$ and $\rho : \textsc{Closure}_S(n)$.

$$\begin{aligned}
\mathsf{close\text{-}term}(t[* := u]) &\equiv \mathsf{subst}\ (\mathsf{var\text{-}def}\ \rho)\ (\mathsf{subst}\ \sigma_u\ t) && \text{(by definition)}\\
&= \mathsf{subst}\ (\mathsf{subst}(\mathsf{var\text{-}def}\ \rho) \circ \sigma_u)\ t && \text{(by law SUBST-SUBST)}\\
&= \mathsf{subst}\ (\mathsf{var\text{-}def}(u :: \rho))\ t && (1)\\
&\equiv \mathsf{close\text{-}term}\ t\ (u :: \rho) && \text{(by definition)}
\end{aligned}$$

where (1) is proved with the law SUBST-CONG and by case-analysis on variables on the function $\mathsf{subst}(\mathsf{var\text{-}def}\ \rho) \circ \sigma_u$. □

Proposition 18 **Weakening in close-term**

Let $t, u : \mathbb{C}_S(n)$ *be two cyclic terms and* $\rho : \textsc{Closure}_S(n)$ *be a closure. Then, we have*

$$\mathsf{close\text{-}term}\ (\mathsf{weaken}\, t)\ (u :: \rho) = \mathsf{close\text{-}term}\ t\ \rho.$$

Proof. Let $t, u : \mathbb{C}_S(n)$ and $\rho : \textsc{Closure}_S(n)$.

$$\begin{aligned}
&\mathsf{close\text{-}term}\ (\mathsf{weaken}(t))\ (u :: \rho)\\
= \quad &\{\ \text{by Prop. 2.17}\ [\mathsf{close\text{-}term}\ (t[* := u])\ \rho = \mathsf{close\text{-}term}\ t\ (u :: \rho)]\ \}\\
&\mathsf{close\text{-}term}\ ((\mathsf{weaken}\, t)[* := u])\ \rho\\
= \quad &\{\ \text{by law WEAKEN-SUBST}_*\ \}\\
&\mathsf{close\text{-}term}\ t\ \rho.
\end{aligned}$$

□

Proposition 19 **Empty closure in close-term**

Let $t : \mathbb{C}_S$ *be a closed cyclic term. Then, we have*

$$\mathsf{close\text{-}term}\ t\ [\,] = t.$$

Proof. Let $t : \mathbb{C}_S$.

$$\begin{aligned}
\mathsf{close\text{-}term}\ t\ [\,] &\equiv \mathsf{subst}\ (\mathsf{var\text{-}def}\ [\,])\ t && \text{(by definition)}\\
&= \mathsf{subst}\ \mathsf{var}\ t && (1)\\
&= t && \text{(by law SUBST-}\eta\text{)}
\end{aligned}$$

where (1) is proved with SUBST-CONG and by case-analysis on x in $\mathsf{var\text{-}def}\ [\,]\ x = \mathsf{var}\, x$. □

Proposition 20 **Propositional computation rule**

Let $n : \mathbb{N}$, $o : S.\mathsf{Op}$ be an operator, $os : \text{VEC}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\mathsf{Ar}\, o)$ be a vector of arguments and $\rho : \text{CLOSURE}_S(n)$ be a closure. Then, for all indices $k : \text{FIN}(S.\mathsf{Ar}\, o)$, we have

$$[\![\, \mathsf{close\text{-}term}\ (\mathsf{rec}\, o \lhd os)\ \rho \,]\!]\ k = [\![\, \mathsf{close\text{-}term}\ (os(k))\ (\mathsf{rec}\, o \lhd os :: \rho) \,]\!].$$

Proof. Let $n : \mathbb{N}$, $o : S.\mathsf{Op}$, $os : \text{VEC}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\mathsf{Ar}\, o)$, $\rho : \text{CLOSURE}_S(n)$ and $k : \text{FIN}(S.\mathsf{Ar}\, o)$.

$$\begin{aligned}
& [\![\, \mathsf{close\text{-}term}\ (\mathsf{rec}\, o \lhd os)\ \rho \,]\!]\ k \\
&\quad \equiv [\![\, \mathsf{rec}\, o \lhd \mathsf{map}^{\text{VEC}}\ (\mathsf{subst}(\mathsf{lift}(\mathsf{var\text{-}def}\, \rho)))\ os \,]\!]\ k \\
&\quad \equiv [\![\, (\mathsf{map}^{\text{VEC}}\ (\mathsf{subst}(\mathsf{lift}(\mathsf{var\text{-}def}\, \rho)))\ os)(k)[* := \mathsf{close\text{-}term}\ (\mathsf{rec}\, o \lhd os)\ \rho] \,]\!] \\
&\quad = [\![\, (\mathsf{subst}\ (\mathsf{lift}(\mathsf{var\text{-}def}\, \rho))\ (os(k)))[* := \mathsf{close\text{-}term}\ (\mathsf{rec}\, o \lhd os)\rho] \,]\!] && (1) \\
&\quad = [\![\, (\mathsf{subst}\ (\mathsf{var\text{-}def}\, \rho)\ (os(k)))[* := \mathsf{rec}\, o \lhd os] \,]\!] && (2) \\
&\quad = [\![\, \mathsf{close\text{-}term}\ (os(k))\ (\mathsf{rec}\, o \lhd os :: \rho) \,]\!] && (3)
\end{aligned}$$

where (1) is given by Prop. A.11 $[(\mathsf{map}^{\text{VEC}}\ f\ v)(k) = f(v(k))]$, (2) is justified with the law SUBST-SUBST$_*$, and (3) is due to Prop. 2.17 $[\mathsf{close\text{-}term}\ (t[* := u])\ \rho = \mathsf{close\text{-}term}\ t\ (u :: \rho)]$.

Remark. In the AGDA definition of the semantics function mentioned previously, this law above holds definitionally.

Theorem 6 **Equivalence of semantics**

Let $n : \mathbb{N}$, $t : \mathbb{C}_S(n)$ be a cyclic term and $\rho : \text{CLOSURE}_S(n)$ be a closure. Then, we have

$$[\![\, \mathsf{close\text{-}term}\ t\ \rho \,]\!] \sim [\![\, t \,]\!]_\rho.$$

Proof. We define the binary relation $\mathcal{R}$ as follows:

$$s\, \mathcal{R}\, s' \overset{\text{def}}{\iff} \exists(n : \mathbb{N}).\ \exists(t : \mathbb{C}_S(n)).\ \exists(\rho : \text{CLOSURE}_S(n)).\ s = [\![\, \mathsf{close\text{-}term}\ t\ \rho \,]\!] \wedge s' = [\![\, t \,]\!]_\rho$$

and show that it is a bisimulation relation, that is:

$$\forall(s, s' : \text{co}\mathbb{T}_S).\ s\, \mathcal{R}\, s' \to s\, \overline{\mathcal{R}}\, s'.$$

where $s\, \overline{\mathcal{R}}\, s'$ is defined to be $\exists(e : \mathsf{root}\, s = \mathsf{root}\, s').\ s(k)\, \mathcal{R}\, s'(e_*(k))$. We assume $s\, \mathcal{R}\, s'$ and show that $s\, \overline{\mathcal{R}}\, s'$. By case-analysis on $t : \mathbb{C}_S(n)$, we consider two cases:

- **Case 1.** Assume that $t = \mathsf{var}\, x$ where $x : \text{FIN}\, n$. Here, we have to show that:

$$[\![\, \mathsf{close\text{-}term}\ (\mathsf{var}\, x)\ \rho \,]\!]\ \overline{\mathcal{R}}\ [\![\, \mathsf{var}\, x \,]\!]_\rho.$$

  By induction on the number of free variables n:

  - **Base step.** Assume that $n = 0$ and $x : \text{FIN}\, 0$.
    This case is trivial since the type of x is empty.
  - **Inductive step.** Assume that $n = \mathsf{suc}\, n'$ where $n' : \mathbb{N}$ and $x : \text{FIN}(\mathsf{suc}\, n')$.
    The induction hypothesis is given by:

$$\forall(x : \text{FIN}\, n').\ \forall(\rho : \text{CLOSURE}_S(n')).\ [\![\, \mathsf{close\text{-}term}\ (\mathsf{var}\, x)\ \rho \,]\!]\ \overline{\mathcal{R}}\ [\![\, \mathsf{var}\, x \,]\!]_\rho. \qquad \text{(IH)}$$

By case-analysis on the variable $x : \textsc{Fin}(\mathsf{suc}\, n')$ and the closure $\rho : \textsc{Closure}_S(\mathsf{suc}\, n')$:

- **Case 1.1.** Assume that $x = \mathsf{zero}$ and $\rho = t' :: \rho'$, $t' : \mathbb{C}_S(\mathsf{suc}\, n')$ and $\rho' : \textsc{Closure}_s(n')$. We have, by definition:

$$[\![\, \mathsf{close\text{-}term}\ (\mathsf{var\,zero})\ \rho\,]\!]\ \overline{\mathcal{R}}\ [\![\ \mathsf{var\,zero}\ ]\!]_\rho \iff [\![\,\mathsf{subst}\ (\mathsf{var\text{-}def}\ \rho')\ t'\,]\!]\ \overline{\mathcal{R}}\ [\![\, t'\,]\!]_{\rho'}.$$

  Furthermore, by case-analysis on the term t':

  - **Case 1.1.1.** Assume that $t' = \mathsf{var}\, x'$ where $x' : \textsc{Fin}\, n'$. The goal is given by

$$[\![\,\mathsf{subst}\ (\mathsf{var\text{-}def}\ \rho')\ (\mathsf{var}\, x')\,]\!]\ \overline{\mathcal{R}}\ [\![\ \mathsf{var}\, x'\,]\!]_{\rho'}$$

    and is discharged by the induction hypothesis (IH).

  - **Case 1.1.2.** Assume that $t' = \mathsf{rec}\, o \lhd os$ where $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, n'))\ (S.\,\mathsf{Ar}\, o)$.

$$\begin{aligned}
&[\![\,\mathsf{subst}\ (\mathsf{var\text{-}def}\ \rho')\ (\mathsf{rec}\, o \lhd os)\,]\!]\ \overline{\mathcal{R}}\ [\![\ \mathsf{rec}\, o \lhd os\,]\!]_{\rho'}\\
&\equiv [\![\ \mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{subst}(\mathsf{lift}(\mathsf{var\text{-}def}\ \rho')))\ os\,]\!]\\
&\qquad \overline{\mathcal{R}}\ (o \blacktriangleleft \lambda k.\ [\![\, os(k)\,]\!]_{(\mathsf{rec}\, o \lhd os\, ::\, \rho')})\\
&\Leftrightarrow \forall k.\ ([\![\,\mathsf{close\text{-}term}\ (\mathsf{rec}\, o \lhd os)\ \rho'\,]\!]\ k)\ \mathcal{R}\ ([\![\, os(k)\,]\!]_{(\mathsf{rec}\, o \lhd os\, ::\, \rho')}) && (1)\\
&\Leftrightarrow \forall k.\ ([\![\,\mathsf{close\text{-}term}\ (os(k))\ (\mathsf{rec}\, o \lhd os :: \rho')\,]\!])\\
&\qquad \mathcal{R}\ ([\![\, os(k)\,]\!]_{(\mathsf{rec}\, o \lhd os\, ::\, \rho')}) && (2)\\
&\Leftrightarrow \top && (3)
\end{aligned}$$

    where (1) holds since both roots are equal by reflexity, (2) is given by Prop. 2.20, and (3) is justified by picking $t := os(k)$ and $\rho := \mathsf{rec}\, o \lhd os :: \rho'$.

- **Case 1.2.** Assume that $x = \mathsf{suc}\, x'$ where $x' : \textsc{Fin}\, n'$ and $\rho = t' :: \rho'$.

$$\begin{aligned}
&[\![\,\mathsf{close\text{-}term}\ (\mathsf{var}(\mathsf{suc}\, x'))\ (t' :: \rho')\,]\!]\ \overline{\mathcal{R}}\ [\![\ \mathsf{var}(\mathsf{suc}\, x')\,]\!]_{t' :: \rho'}\\
\equiv\quad &\{\ \text{by definition}\ \}\\
&[\![\,\mathsf{close\text{-}term}\ (\mathsf{var}\, x')\ \rho'\,]\!]\ \overline{\mathcal{R}}\ [\![\ \mathsf{var}\, x'\,]\!]_{\rho'}\\
\iff\quad &\{\ \text{by induction hypothesis (IH)}\ \}\\
&\top.
\end{aligned}$$

- **Case 2.** Assume that $t = \mathsf{rec}\, o \lhd os$ where $o : S.\,\mathsf{Ar}\, o$, $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\,\mathsf{Ar}\, o)$.

$$\begin{aligned}
&[\![\,\mathsf{close\text{-}term}\ (\mathsf{rec}\, o \lhd os)\ \rho\,]\!]\ \overline{\mathcal{R}}\ [\![\ \mathsf{rec}\, o \lhd os\,]\!]_\rho\\
\iff\quad &\{\ \text{both roots are equal by reflexivity}\ \}\\
&\forall k.\ [\![\,\mathsf{close\text{-}term}\ (\mathsf{rec}\, o \lhd os)\ \rho\,]\!]\ k\ \mathcal{R}\ [\![\, os(k)\,]\!]_{(\mathsf{rec}\, o \lhd os\, ::\, \rho)}\\
\iff\quad &\{\ \text{by rewriting with Proposition 2.20}\ \}\\
&\forall k.\ [\![\,\mathsf{close\text{-}term}\ (os(k))\ (\mathsf{rec}\, o \lhd os :: \rho)\,]\!]\ \mathcal{R}\ [\![\, os(k)\,]\!]_{(\mathsf{rec}\, o \lhd os\, ::\, \rho)}\\
\iff\quad &\left\{\begin{array}{l}\text{we pick } t := os(k) \text{ and } \rho := \mathsf{rec}\, o \lhd os :: \rho,\\ \text{then, clearly both terms are related by } \mathcal{R}.\end{array}\right\}\\
&\top.
\end{aligned}$$

$\square$

Proposition 21 **Weakening in $[\![_]\!]_$**

Let $n : \mathbb{N}$, $t : \mathbb{C}_S(\mathsf{suc}\, n)$ be a cyclic term and $\rho : \mathrm{CLOSURE}_S(\mathsf{suc}\, n)$ be a closure. Then, we have

$$[\![\ \mathsf{weaken}\, t\ ]\!]_\rho \sim [\![\, t\,]\!]_{(\mathsf{tail}\, \rho)}.$$

Proof. Let $n : \mathbb{N}$, $t : \mathbb{C}_S(\mathsf{suc}\, n)$ and $\rho : \mathrm{CLOSURE}_S(\mathsf{suc}\, n)$.

$$\begin{aligned} [\![\ \mathsf{weaken}\, t\ ]\!]_\rho &= [\![\, \mathsf{close\text{-}term}\ (\mathsf{weaken}\, t)\ \rho\,]\!] && (1)\\ &= [\![\, \mathsf{close\text{-}term}\ t\ (\mathsf{tail}\, \rho)\ ]\!] && (2)\\ &= [\![\, t\,]\!]_{(\mathsf{tail}\, \rho)} && (3) \end{aligned}$$ □

where equation (1) is given by Theorem 2.6 [$[\![\, \mathsf{close\text{-}term}\ t\ \rho\,]\!] \sim [\![\, t\,]\!]_\rho$], (2) is justified with Proposition 2.18 [$\mathsf{close\text{-}term}\ (\mathsf{weaken}\, t)\ (u :: \rho) = \mathsf{close\text{-}term}\ t\ \rho$], and finally (3) by Theorem 2.6.

2.2.4. Subterms of Cyclic Terms

So far, we have defined two equivalent functions mapping cyclic terms to infinite trees but we have yet to show that these infinite trees are actually regular. To this end, we will show that the type of subterms of a cyclic term is finite. It is interesting to remark that the following definitions and theorems are a generalization—to an arbitrary signature—of the results of [BH97], which deals with the problem of giving a complete axiomatization of the subtyping relation of iso-recursive types. In the remaining of this section, we show that the type of cyclic terms has an unranked coalgebra structure. Then, we prove some technical results about paths in cyclic terms in the context of substitutions.

Proposition 22 **Coalgebra structure on cyclic terms**

Let n be a natural number and S be a signature. The type of cyclic terms with n free variables, $\mathbb{C}_S(n)$, has an $\mathcal{U}$-coalgebra structure.

Proof. Let $n : \mathbb{N}$ and $S : \mathrm{SIG}$ be a signature. We have to construct a coalgebra for the carrier setoid $(\mathbb{C}_S(n), _{=}_)$. To this end, it suffices to give two maps rank, and next:

$$\begin{aligned} &\mathsf{rank} : \mathbb{C}_S(n) \to \mathbb{N}\\ &\mathsf{rank}\ \ (\mathsf{var}\, _) \qquad \equiv\ \mathsf{zero}\\ &\mathsf{rank}\ \ (\mathsf{rec}\, o \lhd os)\ \equiv\ S.\mathsf{Ar}\, o, \end{aligned}$$

$$\begin{aligned} &\mathsf{next} : \forall (c : \mathbb{C}_S(n)).\ \mathrm{FIN}(\mathsf{rank}\, c) \to \mathbb{C}_S(n)\\ &\mathsf{next}\ \ (\mathsf{var}\, _) \qquad ()\\ &\mathsf{next}\ \ (\mathsf{rec}\, o \lhd os)\ \ k\ \ \equiv\ os(k)[* := \mathsf{rec}\, o \lhd os]. \end{aligned}$$

The proof that both functions, namely rank and next, respects the underlying equality are not required since Leibniz equality is substitutive. □

With Proposition 2.22, we can reuse all the abstract material defined in Section 2.1.2. Let us recall briefly the main definitions:

- $\mathscr{P}(t) : \mathrm{PATH} \to \mathrm{PROP}$, set of *paths* in a cyclic term $t : \mathbb{C}_S(n)$,
- $\rightsquigarrow : \mathbb{C}_S(n) \to \mathbb{C}_S(n) \to \mathrm{TYPE}$, successor relation called in this context the *subterm relation*,
- $t|_p : \mathrm{MAYBE}(\mathbb{C}_S(n))$, subterm of t *indexed by a path* p.

In the remaining of this section, we prove some specific results related to the substitution operation on cyclic terms. We fix an arbitrary signature $S : \mathrm{SIG}$.

Proposition 23 **Subterm substitution decomposition**

Let $t : \mathbb{C}_S(m)$ be a cyclic term and $f : \mathrm{FIN}\, m \to \mathbb{C}_S(n)$ be a substitution function. Then, for all $p \in \mathscr{P}(t)$, we have

$$(\mathsf{subst}\ f\ t)|_p = \mathsf{map}^{\mathrm{MAYBE}}\ (\mathsf{subst}\, f)\ (t|_p).$$

Proof. Let $t : \mathbb{C}_S(m)$, $f : \textsc{Fin}\, m \to \mathbb{C}_S(n)$ and $H : p \in \mathscr{P}(t)$. We proceed by induction on p:

▶ **Base step.** Assume that $p = \varepsilon$.

$$\begin{aligned} \mathsf{subst}\ f\ t|_\varepsilon &\equiv \mathsf{just}(\mathsf{subst}\ f\ t) \\ &\equiv \mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\ f)\ (t|_\varepsilon). \end{aligned}$$

▶ **Inductive step.** Assume that $p = (i,k) :: p'$ where $i : \mathbb{N}$, $k : \textsc{Fin}\, i$ and $p' : \textsc{Path}$. The induction hypothesis is given by:

$$\begin{gathered} \forall (m, n : \mathbb{N}).\ \forall (t : \mathbb{C}_S(m)).\ \forall (f : \textsc{Fin}(m) \to \mathbb{C}_S(n)). \\ p' \in \mathscr{P}(t) \to \mathsf{subst}\ f\ t|_{p'} = \mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\ f)\ (t|_{p'}). \end{gathered} \tag{IH}$$

By case-analysis on t, we consider two cases:

- **Case 1.** Assume that $t = \mathsf{var}\, x$ where $x : \textsc{Fin}\, m$.

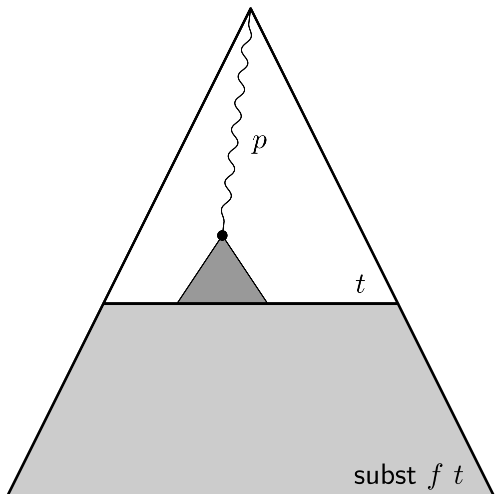
  The type $p \in \mathscr{P}(\mathsf{var}\, x)$ is empty. Contradiction.

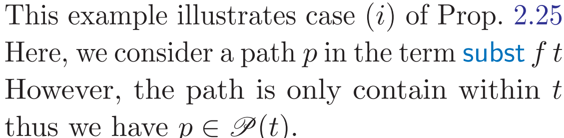
- **Case 2.** Assume that $t = \mathsf{rec}\, o \lhd os$ where $o : S.\mathsf{Op}$ and $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, m))\ (S.\mathsf{Ar}(o))$.

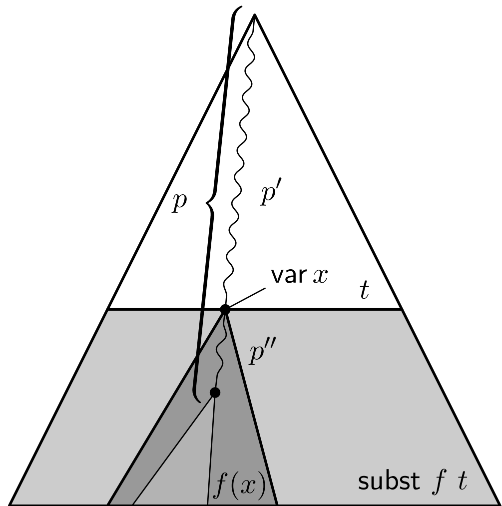
  From elimination of the hypothesis $p \in \mathscr{P}(\mathsf{rec}\, o \lhd os)$, we deduce that there exist a proof $e : n = \mathsf{rank}(\mathsf{rec}\, o \lhd os) = S.\mathsf{Ar}\, o$ and a proof that $p' \in \mathscr{P}(os(e_*(k))[* := \mathsf{rec}\, o \lhd os])$.

$$\begin{aligned}
& \mathsf{subst}\ f\ (\mathsf{rec}\, o \lhd os)|_{(i,k)::p'} \\
\equiv\ & \quad \{ \text{ by definition } \} \\
& \textbf{match}\ i \overset{?}{=} S.\mathsf{Ar}\, o\ \textbf{with} \\
& \quad \mid \mathsf{yes}\, e \Rightarrow (\mathsf{map}^{\textsc{Vec}}\ (\mathsf{subst}(\mathsf{lift}\, f))\ os\ (e_*(k)))[* := \mathsf{subst}\ f\ (\mathsf{rec}\, o \lhd os)]|_{p'} \\
& \quad \mid \mathsf{no}\ _ \Rightarrow \mathsf{nothing} \\
& \textbf{end} \\
=\ & \quad \left\{ \begin{array}{l} \text{by simplification of the } \textbf{match} \text{ expression with } e \\ \text{and by Prop. A.11 } [(\mathsf{map}^{\textsc{Vec}}\ f\ v)(k) = f(v(k))] \end{array} \right\} \\
& \mathsf{subst}\ (\mathsf{lift}\, f)\ (os(e_*(k)))[* := \mathsf{subst}\ f\ (\mathsf{rec}\, o \lhd os)]|_{p'} \\
=\ & \quad \{ \text{ by the law } \textsc{subst-subst}_* \} \\
& \mathsf{subst}\ f\ (os(e_*(k))[* := \mathsf{rec}\, o \lhd os])|_{p'} \\
=\ & \quad \{ \text{ by the induction hypothesis (IH) } \} \\
& \mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\, f)\ \left(os(e_*(k))[* := \mathsf{rec}\, o \lhd os]|_{p'} \right) \\
=\ & \quad \{ \text{ by simplification of the } \textbf{match} \text{ expression with } e \} \\
& \mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\, f) \left(\begin{array}{l} \textbf{match}\ i \overset{?}{=} S.\mathsf{Ar}\, o\ \textbf{with} \\ \quad \mid \mathsf{yes}\, e \Rightarrow os(e_*(k))[* := \mathsf{rec}\, o \lhd os]|_{p'} \\ \quad \mid \mathsf{no}\ _ \Rightarrow \mathsf{nothing} \\ \textbf{end} \end{array} \right) \\
\equiv\ & \quad \{ \text{ by definition } \} \\
& \mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\, f)\ (\mathsf{rec}\, o \lhd os|_{(i,k)::p'}).
\end{aligned}$$

□




This example illustrates case (i) of Prop. 2.25. Here, we consider a path p in the term $\mathsf{subst}\ f\ t$. However, the path is only contain within t, thus we have $p \in \mathscr{P}(t)$.



This example illustrates case (ii) of Prop. 2.25. Here, we consider a path p in the term $\mathsf{subst}\ f\ t$. Note that, in this case, the path goes beyond t. Thus, necessarily, the path ought to go through a variable $\mathsf{var}\ x$ of t. As a result, the path p can be decomposed in two parts: p' which is strictly contained within t and p'' which is strictly contained in the substituted term $f(x)$.

Figure 2.7. Illustration of the two cases described in Proposition 2.25.

In the following, we prove that a path in a cyclic term can be extended to a path in term containing a substitution.

Proposition 24 **Path in substitution introduction rule**

Let $t : \mathbb{C}_S(m)$ be a cyclic term, $f : \textsc{Fin}\, m \to \mathbb{C}_S(n)$ be a substitution. For all paths p, if $p \in \mathscr{P}(t)$ then $p \in \mathscr{P}(\mathsf{subst}\ f\ t)$.

Proof. Let $m, n : \mathbb{N}$, $t : \mathbb{C}_S(m)$, $f : \textsc{Fin}\, m \to \mathbb{C}_S(n)$, $p : \textsc{Path}$ and $H : p \in \mathscr{P}(t)$.

$$
\begin{aligned}
& p \in \mathscr{P}(\mathsf{subst}\ f\ t) \\
\Longleftrightarrow \quad & \{ \text{ by Prop. 2.5 [Succ. function spec.] } \} \\
& \mathsf{is\text{-}just}(\mathsf{subst}\ f\ t|_p) \\
= \quad & \left\{ \text{ by Prop. 2.23 } [(\mathsf{subst}\ f\ t)|_p = \mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\, f)\ (t|_p)] \right\} \\
& \mathsf{is\text{-}just}\left(\mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\, f)\ (t|_p)\right) \\
\Longleftrightarrow \quad & \{ \text{ by Prop. A.4 } [\mathsf{is\text{-}just}(\mathsf{map}^{\textsc{Maybe}}\ f\ m) \Leftrightarrow \mathsf{is\text{-}just}\, m] \} \\
& \mathsf{is\text{-}just}(t|_p) \\
\Longleftrightarrow \quad & \{ \text{ by Prop. 2.5 [Succ. function spec.] } \} \\
& p \in \mathscr{P}(t) \\
\Longleftrightarrow \quad & \{ \text{ by assumption } H \} \\
& \top .
\end{aligned}
$$

□

Next, we prove how a path within a cyclic term applied to a substitution can be decomposed.

Proposition 25 **Path in substitution elimination rule**

Let $t : \mathbb{C}_S(m)$ be a cyclic term and $f : \textsc{Fin}\, m \to \mathbb{C}_S(n)$ be a substitution function. For all paths p such that $p \in \mathscr{P}(\mathsf{subst}\ f\ t)$, then one of the following holds:

(i) $p \in \mathscr{P}(t)$,

(ii) there exist a variable $x : \textsc{Fin}\, m$ and a decomposition of p as two paths p_1 and p_2, such that $t|_{p_1} = \mathsf{just}(\mathsf{var}\, x)$ and $p_2 \in \mathscr{P}(f(x))$.

Proof. Let $m, n : \mathbb{N}$, $t : \mathbb{C}_S(m)$, $f : \textsc{Fin}\, m \to \mathbb{C}_S(n)$, $p : \textsc{Path}$ and $H : p \in \mathscr{P}(\mathsf{subst}\ f\ t)$. We proceed by path induction on p:

▶ **Base step.** Assume that $p = \varepsilon$.
In this case, (i) is trivially true.

▶ **Inductive step.** Assume that $p = (i, k) :: p'$ where $i : \mathbb{N}$, $k : \textsc{Fin}\, i$ and $p' : \textsc{Path}$.
The induction hypothesis is given by:

$$\forall (m, n : \mathbb{N}).\ \forall (t : \mathbb{C}_S(m)).\ \forall (f : \textsc{Fin}\, m \to \mathbb{C}_S(n)).\ H : p' \in \mathscr{P}(\mathsf{subst}\ f\ t)$$
$$p' \in \mathscr{P}(t) \uplus \sum_{x : \textsc{Fin}\, m} \left\{ (p_1, p_2) \;\middle|\; p = p_1 +\!\!+ p_2 \wedge t|_{p_1} = \mathsf{just}(\mathsf{var}\, x) \wedge p_2 \in \mathscr{P}(f(x)) \right\}. \quad \text{(IH)}$$

By case analysis on t, we consider two cases:

- **Case 1.** Assume that $t = \mathsf{var}\, x'$ where $x' : \textsc{Fin}\, m$.
  We show that (ii) holds. We pick $x := x'$ as the variable. Obviously, taking both $p_1 := \varepsilon$ and $p_2 := (i, k) :: p'$ yields a decomposition of p. Finally, $\mathsf{var}\, x|_{p_1} = \mathsf{just}(\mathsf{var}\, x)$ is true by reflexivity and $p_2 \in \mathscr{P}(f(x))$ holds by assumption.

- **Case 2.** Assume that $t = \mathsf{rec}\, o \lhd os$ where $o : S.\mathsf{Op}$ and $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, m))\ (S.\mathsf{Ar}\, o)$.
  From the hypothesis $(i, k) :: p' \in \mathscr{P}(\mathsf{subst}\ f\ (\mathsf{rec}\, o \lhd os))$, we deduce that there exist a proof $e : n = S.\mathsf{Ar}\, o$ and a proof $H : p' \in \mathscr{P}\big(\mathsf{subst}\ f\ (os(e_*(k))[* := \mathsf{rec}\, o \lhd os])\big)$. With H applied to the induction hypothesis (IH), we consider two cases:

  - **Case 2.1.** Assume that $H' : p' \in \mathscr{P}\big(os(e_*(k))[* := \mathsf{rec}\, o \lhd os]\big)$.
    We show that (i) holds. That is, $p' \in \mathscr{P}(\mathsf{rec}\, o \lhd os)$. By applying the rule $\mathscr{P}_{::}$, it remains to show that there is an equality proof $e : n = S.\mathsf{Ar}\, o$ such that $p' \in \mathscr{P}\Big(os(e_*(k))[* := \mathsf{rec}\, o \lhd os]\Big)$. The first proof is given by e while the second is given by hypothesis H'.

  - **Case 2.2.** Assume that $x' : \textsc{Fin}\, m$ and $p'_1, p'_2 : \textsc{Path}$ such that:

$$p' = p'_1 +\!\!+ p'_2, \tag{1}$$
$$os(e_*(k))[* := \mathsf{rec}\, o \lhd os] = \mathsf{just}(\mathsf{var}\, x'), \tag{2}$$
$$p'_2 \in \mathscr{P}(f(x')). \tag{3}$$

    We show that (ii) holds. We pick $x := x'$, $p_1 := k :: p'_1$ and $p_2 := p'_2$. Finally,
    - $k :: p' = (k :: p_1) +\!\!+ p_2$ by (1),
    - $os(e_*(k))[* := \mathsf{rec}\, o \lhd os] = \mathsf{just}(\mathsf{var}\, x)$ by (2),
    - $p'_2 \in \mathscr{P}(f(x))$ by (3). □

Proposition 26 **Path in one-variable substitution elimination rule**

Let $t : \mathbb{C}_S(\mathsf{suc}\, n)$ and $u : \mathbb{C}_S(n)$ be two cyclic terms. For all paths p such that $p \in \mathscr{P}(t[* := u])$, then one of the following holds:

(i) $p \in \mathscr{P}(t)$,
(ii) there exists a path $p' \neq p$ such that p' is a suffix of p and $p' \in \mathscr{P}(u)$.

Proof. Consequence of Proposition 2.25. □

2.2.5. Soundness and Completeness of Syntactic Representation

In this section, we justify that the type of cyclic terms is a syntax for regular trees. Consequently, we prove that the representation is sound, in the sense that each cyclic term induces a regular tree. Furthermore, we are also interested in the converse problem, that is, proving that each regular tree has indeed a representation as a cyclic term.

Soundness of syntactic representation

Proving that the cyclic term representation is sound with respects to the semantic function $[\![_]\!]$ consists in proving that the type of subterms is weakly finitely indexed. To this end, we proceed compositionally. First, we show that the type of paths $\mathbb{P}(\mathsf{var}\, x)$ from a variable is finite. Then, we show that the same holds for the type of paths $\mathbb{P}(\mathsf{rec}\, o \lhd os)$, assuming that the type of paths is finite for all subterms of $\mathsf{rec}\, o \lhd os$. Finally, we prove that the type of subterms $t\!\rightsquigarrow$ is weakly finitely indexed by exploiting the isomorphism between the type $t\!\rightsquigarrow$ and the type of paths $\mathbb{P}(t)$.

Lemma 2 **Finiteness of $\mathbb{P}(\mathsf{var}\, x)$**

Let $x : \mathrm{F{\scriptsize IN}}\, n$ be a free variable. Then, the type $\mathbb{P}(\mathsf{var}\, x)$ is weakly finitely indexed.

Proof. Let $x : \mathrm{F{\scriptsize IN}}\, n$. We have to show that there exist a number c and a map $f : \mathrm{F{\scriptsize IN}}\, c \to \mathbb{P}(\mathsf{var}\, x)$ which has a section. We define $c := 1$ and f as follows:

$f : \mathrm{F{\scriptsize IN}}\, c \to \mathbb{P}(\mathsf{var}\, x)$
$f \;\; \mathsf{zero} \quad \equiv \; (\varepsilon, \mathscr{P}_\varepsilon)$
$f \;\; (\mathsf{suc}())$.

It remains to show that f has a section which consists in proving that:

$$\forall (p : \mathbb{P}(\mathsf{var}\, x)).\ \{\, i : \mathrm{F{\scriptsize IN}}\, c \mid f(i) \approx p \,\}\,.$$

We map any p to the index $\mathsf{zero} : \mathrm{F{\scriptsize IN}}\, c$. Let $(p, H) : \mathbb{P}(\mathsf{var}\, x)$, it remains to show that:

$f(i)$

$=$ { by definition }

$(\varepsilon, \mathscr{P}_\varepsilon)$

$\approx$ By case-analysis on p:

- **Case 1.** Assume that $p = \varepsilon$.
  Clearly, we have $\mathsf{var}\, x|_\varepsilon = \mathsf{var}\, x|_p = \mathsf{var}\, x|_\varepsilon$.
- **Case 2.** Assume that $p = (i, k) :: p'$ where $i : \mathbb{N}$, $k : \mathrm{F{\scriptsize IN}}\, i$ and $p' : \mathrm{P{\scriptsize ATH}}$.
  The type of $H : (i, k) :: p' \in \mathscr{P}(\mathsf{var}\, x)$ is empty. Contradiction.

(p, H). □

Lemma 3 **Finiteness of $\mathbb{P}(\mathsf{rec}\, o \lhd os)$**

Let $o : S.\mathsf{Op}$ be an operator and $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\,\mathsf{Ar}\, o)$ be a vector of cyclic terms. If, for all $k : \textsc{Fin}(S.\,\mathsf{Ar}\, o)$, the type $\mathbb{P}(os(k))$ is weakly finitely indexed, then so is $\mathbb{P}(\mathsf{rec}\, o \lhd os)$.

Proof. Let $o : S.\mathsf{Op}$, $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\,\mathsf{Ar}\, o)$ and $k : \textsc{Fin}(S.\,\mathsf{Ar}\, o)$. By assumption, for all $k : \textsc{Fin}(S.\,\mathsf{Ar}\, o)$, the type $\mathbb{P}(os(k))$ is weakly finitely indexed. Thus, for all k, there exists a natural number c_k along with two functions $f_k : \textsc{Fin}\, c_k \to \mathbb{P}(os(k))$ and $g_k : \mathbb{P}(os(k)) \to \textsc{Fin}\, c_k$ such that g_k is a section of f_k.

In order to prove that $\mathbb{P}(\mathsf{rec}\, o \lhd os)$ is weakly finitely indexed, we have to show that there exists a function from a finite set to $\mathbb{P}(\mathsf{rec}\, o \lhd os)$ that has a right inverse. First, we show that the type $\mathbb{1} \uplus \sum_{k:\textsc{Fin}(S.\,\mathsf{Ar}\, o)} \textsc{Fin}\, c_k$ is finite, *i.e.*, it is isomorphic to a finite type:

$$\begin{array}{ll} & \mathbb{1} \uplus \sum_{k:\textsc{Fin}(S.\,\mathsf{Ar}\, o)} \textsc{Fin}\, c_k \\ \cong & \quad \{ \text{ by Prop. 1.2 } [\textsc{Fin}\, 1 \cong \mathbb{1}] \text{ and Prop. A.7 } [\mathsf{sum}_k\, f_k \text{ finite}] \ \} \\ & \textsc{Fin}\, 1 \uplus \textsc{Fin} \left(\underset{k}{\mathsf{sum}}\, c_k \right) \\ \cong & \quad \{ \text{ by Prop. 1.3 } [\textsc{Fin}(\mathsf{suc}\, n) \cong \mathbb{1} \uplus \textsc{Fin}\, n] \ \} \\ & \textsc{Fin} \left(\mathsf{suc} \left(\underset{k}{\mathsf{sum}}\, c_k \right) \right). \end{array}$$

Next, we define a map from elements in $\mathbb{1} \uplus \sum_{k:\textsc{Fin}(S.\,\mathsf{Ar}\, o)} \textsc{Fin}\, c_k$ to elements in $\mathbb{P}(\mathsf{rec}\, o \lhd os)$:

$$\begin{array}{lll} f : \mathbb{1} \uplus \sum_{k:\textsc{Fin}(S.\,\mathsf{Ar}\, o)} \textsc{Fin}\, c_k \to \mathbb{P}(\mathsf{rec}\, o \lhd os) \\ f\ \ (\mathsf{inl}\, _) & \equiv & (\varepsilon, \mathscr{P}_\varepsilon) \\ f\ \ (\mathsf{inr}(k, i_k)) & \equiv & ((_, k) :: \mathsf{proj}_1(f_k(i_k)), H) \\ \quad \textbf{where}\ \ H : f_k(i_k) \in \mathbb{P}(os(k)[* := \mathsf{rec}\, o \lhd os]) \end{array}$$

where the proof H is obtained from the second projection of $f_k(i_k)$, yielding a path in $\mathscr{P}(os(k))$, along with Proposition 2.24 lifting that path in a term with a substitution.

Finally, we show that f has a right-inverse, *i.e.*, we prove the following:

$$\forall (P : \mathbb{P}(\mathsf{rec}\, o \lhd os)).\ \left\{ i : \mathbb{1} \uplus \textstyle\sum_{k:\textsc{Fin}(S.\,\mathsf{Ar}\, o)} \textsc{Fin}\, c_k \;\middle|\; f(i) \approx P \right\}.$$

Let $P := (p, M) : \mathbb{P}(\mathsf{rec}\, o \lhd os)$ where $p : \textsc{Path}$ and $M : p \in \mathscr{P}(\mathsf{rec}\, o \lhd os)$. We proceed by well-founded induction on the length of the path p. The induction hypothesis is given by:

$$\begin{array}{c} \forall (P' : \mathbb{P}(\mathsf{rec}\, o \lhd os)).\ \forall (L : \mathsf{length}(\mathsf{proj}_1\, P') < \mathsf{length}\, p). \\ \left\{ i : \mathbb{1} \uplus \textstyle\sum_{k:\textsc{Fin}(S.\,\mathsf{Ar}\, o)} \textsc{Fin}\, c_k \;\middle|\; f(i) \approx P' \right\}. \end{array} \tag{IH}$$

By case-analysis on the path p:

- **Case 1.** Assume that $p = \varepsilon$.

  We pick $i := \mathsf{inl}\ \mathsf{tt}$. It remains to prove that $f(i) \approx P$. By definition of setoid equality (Def. 2.9), it suffices to show that:

$$\mathsf{rec}\, o \lhd os|_{\mathsf{proj}_1(f(i))} = \mathsf{rec}\, o \lhd os|_\varepsilon = \mathsf{rec}\, o \lhd os|_p.$$

  Clearly, this holds by reflexivity.

- **Case 2.** Assume that $p = (m, k) :: p'$ where $m : \mathbb{N}$, $k : \textsc{Fin}\, m$ and $p' : \textsc{Path}$.
  By applying Proposition 2.26 to H, we consider two cases:

  - **Case 2.1.** Assume that $H' : p' \in \mathscr{P}\big(os(e_*(k))\big)$.
    We pick $i := \mathsf{inr}(e_*(k), g_k(p', H'))$. Let $q := \mathsf{proj}_1(f_k(g_k(p', H'))) : \textsc{Path}$. It remains to prove that $f(i) \approx P$:

$$
\begin{aligned}
& f(i) \\
\approx \quad & \{ \text{ by definition of setoid equality (Def. 2.9) } \} \\
& \mathsf{rec}\, o \lhd os|_{\mathsf{proj}_1(f(i))} \\
\equiv \quad & \{ \text{ by definition } \} \\
& \textbf{match } S.\mathsf{Ar}\, o \overset{?}{=} S.\mathsf{Ar}\, o \textbf{ with} \\
& \quad \mathsf{yes}\, e' \Rightarrow os(e'_*(e_*(k)))[* := \mathsf{rec}\, o \lhd os]|_q \\
& \quad \mathsf{no}\ _ \ \Rightarrow \mathsf{nothing} \\
& \textbf{end} \\
\approx \quad & \left\{ \text{ by substituting } S.\mathsf{Ar}\, o \overset{?}{=} S.\mathsf{Ar}\, o \text{ with } \mathsf{yes\ refl} \text{ since } \mathbb{N} \text{ satisfies UIP } \right\} \\
& os(e_*(k))[* := \mathsf{rec}\, o \lhd os]|_q \\
\approx \quad & \left\{ \text{ by Prop. 2.23 } [(\mathsf{subst}\ f\ t)|_p = \mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\ f)\ (t|_p)] \right\} \\
& \mathsf{map}^{\textsc{Maybe}}\ (\lambda t.\ t[* := \mathsf{rec}\, o \lhd])\ \Big(os(e_*(k))|_{\mathsf{proj}_1(f_k(g_k(p', H')))}\Big) \\
\approx \quad & \{\ g_k \text{ is a right-inverse of } f_k \text{ by assumption } \} \\
& \mathsf{map}^{\textsc{Maybe}}\ (\lambda t.\ t[* := \mathsf{rec}\, o \lhd])\ \Big(os(e_*(k))|_{p'}\Big) \\
\approx \quad & \left\{ \text{ by Prop. 2.23 } [(\mathsf{subst}\ f\ t)|_p = \mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\ f)\ (t|_p)] \right\} \\
& os(e_*(k))[* := \mathsf{rec}\, o \lhd os]|_{p'} \\
\approx \quad & \left\{ \text{ by substituting } m \overset{?}{=} S.\mathsf{Ar}\, o \text{ with } \mathsf{yes}\, e \text{ since } \mathbb{N} \text{ satisfied UIP } \right\} \\
& \textbf{match } m \overset{?}{=} S.\mathsf{Ar}\, o \textbf{ with} \\
& \quad \mathsf{yes}\, e \Rightarrow os(e_*(k))[* := \mathsf{rec}\, o \lhd os]|_{p'} \\
& \quad \mathsf{no}\ _ \Rightarrow \mathsf{nothing} \\
& \textbf{end} \\
\equiv \quad & \{ \text{ by definition } \} \\
& \mathsf{rec}\, o \lhd os|_{(m,k)\, ::\, p'} \\
\approx \quad & \{ \text{ by definition of setoid equality (Def. 2.9) } \} \\
& ((m, k) :: p', \mathscr{P}_{::}\ e\ H).
\end{aligned}
$$

  - **Case 2.2.** Assume that $p_1, p_2 : \textsc{Path}$, $E : p' = p_1 +\!\!+ p_2$, $H : os(e_*(k))|_{p_1} = \mathsf{just}(\mathsf{var\ zero})$ and $H' : p_2 \in \mathscr{P}(\mathsf{rec}\, o \lhd os)$.
    Since we obtain another path p_2 in $\mathscr{P}(\mathsf{rec}\, o \lhd os)$, we can use the induction hypothesis (IH). Clearly, we have $\mathsf{length}(p_2) < \mathsf{length}(\pi_1(P))$ because p_2 is a decomposition of p'

which is already smaller than $\mathsf{proj}_1\ P = p$. Consequently, we obtain from H' applied to (IH), an index $i' : \mathbb{1} \uplus \sum_{k:\textsc{Fin}(S.\,\mathsf{Ar}\,o)} \textsc{Fin}\, c_k$ such that $H'' : f(i) \approx P'$ where $P' := (p_2, H')$. We pick $i := i'$, it remains to prove that:

$$
\begin{aligned}
& f(i) \\
\approx \quad & \{ \text{ by hypothesis } H'' \text{ and definition of setoid equality (Def. 2.9) } \} \\
& \mathsf{rec}\, o \lhd os|_{p_2} \\
\equiv \quad & \{ \text{ by definition } \} \\
& \left(\mathsf{map}^{\textsc{Maybe}}\ (\lambda t.\ t[* := \mathsf{rec}\, o \lhd os])\ (\mathsf{just}(\mathsf{var}\,\mathsf{zero}))\right) \ggg= \lambda t.\ t|_{p_2} \\
= \quad & \{ \text{ by rewriting hypothesis } H \ \} \\
& \left(\mathsf{map}^{\textsc{Maybe}}\ (\lambda t.\ t[* := \mathsf{rec}\, o \lhd os])\ (os(e_*(k))|_{p_1})\right) \ggg= \lambda t.\ t|_{p_2} \\
\approx \quad & \left\{ \text{ by Prop. 2.23 } [(\mathsf{subst}\ f\ t)|_p = \mathsf{map}^{\textsc{Maybe}}\ (\mathsf{subst}\, f)\ (t|_p)] \right\} \\
& os(e_*(k))[* := \mathsf{rec}\, o \lhd os]|_{p_1 +\!\!+ p_2} \\
= \quad & \{ \text{ by rewriting hypothesis } E \ \} \\
& \mathsf{rec}\, o \lhd |_{(n,k)\, ::\, p'} \\
\approx \quad & \{ \text{ by definition of setoid equality (Def. 2.9) } \} \\
& P.
\end{aligned}
$$

□

Lemma 4 **Finiteness of $\mathbb{P}(t)$**

Let $t : \mathbb{C}_S(n)$ be a cyclic term with n free variables. Then, the type $\mathbb{P}(t)$ is weakly finitely indexed.

Proof. We proceed by induction on t:

- **Base step.** Assume that $t = \mathsf{var}\, x$ with $x : \textsc{Fin}\, n$.
  This a consequence of Lem. 2.2 [$\mathbb{P}(\mathsf{var}\, x)$ finite].
- **Inductive step.** Assume that $t = \mathsf{rec}\, o \lhd os$ with $o : S.\mathsf{Op}$ and $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\,\mathsf{Ar}\, o)$.
  The induction hypothesis is given by:

$$\forall (k : \textsc{Fin}(S.\,\mathsf{Ar}\, o)).\ \mathrm{WFI}(\mathbb{P}(os(k))). \tag{IH}$$

We conclude with the induction hypothesis (IH) applied to Lemma 2.3 [$\mathbb{P}(\mathsf{rec}\, o \lhd os)$ finite]. □

Theorem 7 **Finiteness of $t_{\rightsquigarrow}$**

Let $n : \mathbb{N}$ and $t : \mathbb{C}_S(n)$ be a cyclic term, then the type of subterms of t, namely $t_{\rightsquigarrow}$, is weakly finitely indexed.

Proof. Let n and $t : \mathbb{C}_S(n)$. This is a direct consequence of Thm. 2.3 [$s_{\rightsquigarrow} \cong \mathbb{P}(s)$] and Lem. 2.4. □

In the remaining of this section, we prove that the semantics function $[\![\,_\,]\!]$ maps cyclic terms to regular trees. We show that this result follows from the finiteness of the type of subterms of cyclic terms and the proposition below.

Proposition 27 **Path-membership equivalence**

For all $t : \mathbb{C}_S$ and p : PATH, we have:

$$p \in \mathscr{P}_{\mathbb{C}}(t) \iff p \in \mathscr{P}_{\text{COT}}[\![\, t \,]\!].$$

Proof. Let $t : \mathbb{C}_S$ and p : PATH. We proceed by induction on p:

▸ **Base step.** Assume that $p = \varepsilon$.
This case is trivially true since the empty path ε is valid in any unranked coalgebra.

▸ **Inductive step.** Assume that $p = (n, k) :: p'$ with $n : \mathbb{N}$, k : FIN n and p' : PATH.
The induction hypothesis is given by:

$$\forall (t : \mathbb{C}_S).\ p' \in \mathscr{P}_{\mathbb{C}}(t) \iff p' \in \mathscr{P}_{\text{COT}}[\![\, t \,]\!]. \tag{IH}$$

Furthermore, by case-analysis on t, we consider two cases:

- **Case 1.** Assume that $t = \mathsf{var}\, x$ with x : FIN zero.
Obviously, this case leads to a contradiction since the type of x is empty.
- **Case 2.** Assume that $t = \mathsf{rec}\, o \lhd os$ with $o : S.\mathsf{Op}$ and os : VEC $(\mathbb{C}_S(\mathsf{suc\,zero}))\ (S.\mathsf{Ar}(o))$.

$$\begin{aligned}
&(n, k) :: p' \in \mathscr{P}_{\mathbb{C}}(\mathsf{rec}\, o \lhd os) \\
&\quad \Leftrightarrow \exists (e : n = S.\mathsf{Ar}(o)).\ p' \in \mathscr{P}_{\mathbb{C}}(os(e_*(k))[* := \mathsf{rec}\, o \lhd os]) && (1)\\
&\quad \Leftrightarrow \exists (e : n = S.\mathsf{Ar}(o)).\ p' \in \mathscr{P}_{\text{COT}}([\![\, os(e_*(k))[* := \mathsf{rec}\, o \lhd os] \,]\!]) && (2)\\
&\quad \Leftrightarrow (n, k) :: p' \in \mathscr{P}_{\text{COT}}([\![\, \mathsf{rec}\, o \lhd os \,]\!]) && (3)
\end{aligned}$$

where (1) and (3) are given by the rule $\mathscr{P}_{::}$, and (2) is a consequence of (IH). □

Proposition 28 **Successor in semantics decomposition**

Let $n : \mathbb{N}$, p be a path and $t : \mathbb{C}_S$ be a cyclic term. Then, we have

$$[\![\, t \,]\!]|_p = \mathsf{map}^{\text{MAYBE}}\ [\![\, _ \,]\!]\ (t|_p).$$

Proof. Straightforward by induction on the path p. □

Theorem 8 **Soundness of cyclic term representation**

Let $t : \mathbb{C}_S$ be a closed cyclic term. Then, the infinite tree $[\![\, t \,]\!]$ is regular.

Proof. Let $t : \mathbb{C}(S)$. In order to show that $[\![\, t \,]\!]$ is regular, we have to prove that the type $[\![\, t \,]\!]\rightsquigarrow$ is weakly finitely indexed. By Theorem 2.3, it suffices to show that the type $\mathbb{P}[\![\, t \,]\!]$ is weakly finitely indexed. Also, by Proposition 1.13, it suffices to show that there exists a weak retract from $\mathbb{P}[\![\, t \,]\!]$ to $\mathbb{P}(t)$. Consequently, it remains to show that:

- $f : \mathbb{P}(t) \longrightarrow \mathbb{P}[\![\, t \,]\!]$.
Let $p : \mathbb{P}(t)$. We pick the first projection to be $\mathsf{proj}_1\, p$. The proof that $\mathsf{proj}_1 \in \mathscr{P}([\![\, t \,]\!])$ is given by applying $\mathsf{proj}_2\, p$ to Proposition 2.27. Finally, it is straightforward to check, with Proposition 2.28, that we have a setoid morphism.
- $g : \mathbb{P}[\![\, t \,]\!] \to \mathbb{P}(t)$.
Let $p : \mathbb{P}(t)$. We pick the first projection to be $\mathsf{proj}_1\, p$. The proof that $\mathsf{proj}_1 \in \mathscr{P}(t)$ is given by applying $\mathsf{proj}_2\, p$ to Proposition 2.27.
- $\forall (p : \mathbb{P}[\![\, t \,]\!]).\ f(g(p)) \approx p$.
By Definition 2.9, this holds by reflexivity. □

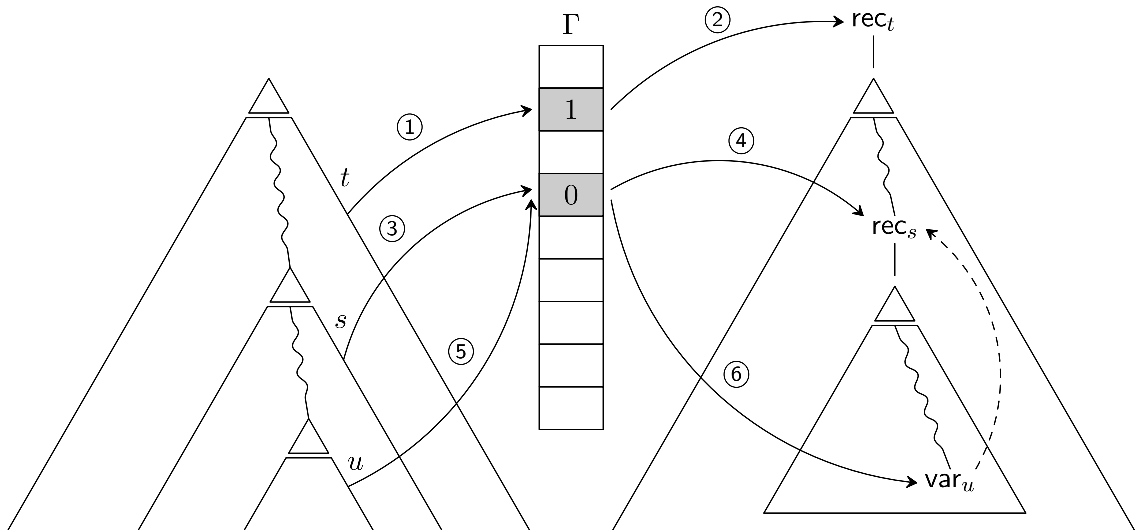


Figure 2.8. Computation of a cyclic term from a regular tree.

Completeness of syntactic representation

Proving that the cyclic term representation is complete with respects to the semantics function $[\![_]\!]$ means that we have to show that there exists a function mapping regular trees to cyclic terms. The algorithm computing such a cyclic term is quite similar to the ones found in graph theory. Intuitively, it consists in traversing in the regular tree until we encounter a leaf node or a node that has already been visited. Consequently, by marking each visited node, we know that this process will eventually end since a regular tree contains only finitely many distinct subtrees. However, we have to deal with some technicalities because, in general, the bisimilarity relation is not decidable on trees. As such, it is not clear how to test whether a node has already been visited. Nevertheless, we will show that having an indexing function is enough to ensure termination and establish the completeness result.

In Figure 2.8, we illustrate how the algorithm computes a cyclic term from a regular tree. On the left, we have an infinite tree t, assumed to be regular, along with an arbitrary subtree s. In addition, we consider yet another subtree u of s, and by transitivity of t. In the middle, we represent an environment that is used to keep track of the subtrees of t that have already been visited. Visited subtrees are denoted by grayed out cells in the environment Γ. In addition, we associate to each visited state a fresh free variable. Following the de Bruijn typing discipline, the free variables are canonical and are represented by natural numbers. On the right, we have represented the cyclic term obtained from t, in six main steps detailed in the following:

① The tree t has not been visited yet. Thus, we mark it in Γ and associate a free variable to t.

② We build the cyclic term corresponding to t by using the constructor rec $_ \lhd _$. The root of t is mapped as-is in the corresponding cyclic term.

③ The tree s has not been visited yet. Thus, we mark it in Γ and associate a free variable to s.

④ As we did for t, we build the cyclic term corresponding to the subtree s of t.

⑤ The tree u is mapped to the same index as s. Here, we are in the case where a visited subtree has already been marked.

⑥ We indicate that u has already been visited by using the constructor var along with the variable associated with the index $\Gamma(u)$. As a result, we obtain a back-pointer into rec$_s$.

Now, we define the type of environments and show how they are extended in a way that preserves de Bruijn typing discipline.

Definition 16 **Environment**

Let $n : \mathbb{N}$ be a natural number. We define the type of environments over S, denoted $\textsc{Env}_S(n)$, as follows:

$$\textsc{Env}_S(n) := \textsc{Fin}\, n \to \mathbb{C}_S(n).$$

The operation extending an environment with an additional element is given by:

$$
\begin{array}{l}
_;_ : \{n : \mathbb{N}\} \to \mathbb{C}_S(\mathsf{suc}\, n) \to \textsc{Env}_S(n) \to \textsc{Env}_S(\mathsf{suc}\, n) \\
(t\,;\Gamma)\ \ \mathsf{zero} \qquad \equiv\ t \\
(t\,;\Gamma)\ \ (\mathsf{suc}\, x)\ \equiv\ \mathsf{weaken}(\Gamma(x)).
\end{array}
$$

Note that, an environment is extended similarly to a list, *i.e.*, by adding the new element in front. As a result, the weaken function is applied to all remaining elements. This is mandatory, because we have to take into account the dependency between the number of free variables in a term and the number of elements in the environment.

We fix an arbitrary regular tree $\chi : \textsc{Reg}_S$. Now, we define the function sem-inv-aux that computes the cyclic term representation of an arbitrary subtree of χ as follows:

$$
\begin{array}{l}
\mathsf{sem\text{-}inv\text{-}aux} : \{l : \textsc{List}\, \#\chi\} \to \lceil l \rceil \to \chi\!\rightsquigarrow\ \to \textsc{Env}_S(\mathsf{length}\, l) \to \mathbb{C}_S(\mathsf{length}\, l) \\
\mathsf{sem\text{-}inv\text{-}aux}\ \ B\ \ t\ \ \Gamma\ \equiv \\
\qquad \textbf{match}\ s \in^{\#}?\ l\ \textbf{with} \\
\qquad\quad \mathsf{yes}(k, _)\ \Rightarrow \Gamma(k) \\
\qquad\quad \mathsf{no}\, M \qquad \Rightarrow \mathsf{rec}\, (\mathsf{root}\, s) \lhd k \mapsto \mathsf{sem\text{-}inv\text{-}aux}\ (M\lceil{::}\rceil B)\ (\mathsf{next}\ t\ k)\ (\mathsf{var}\ \mathsf{zero}; \Gamma) \\
\qquad \textbf{end}
\end{array}
$$

Here, the l is used to mark the indices of the subtrees of χ that have already been visited. The term of type $\lceil l \rceil$ denotes a proof that the list is $\succ$-accessible and is used to witness termination of the recursion function (see Appendix A for further details). Moreover, $t : \chi\!\rightsquigarrow$ represents an arbitrary subtree of χ and $\Gamma : \textsc{Env}_S(\mathsf{length}\, l)$ is used to assign a variable to each visited subtree a corresponding cyclic term.

Next, we establish the soundness of the function sem-inv-aux with respects to the semantic function $[\![_]\!]$. Intuitively, we have to show that given a subtree $t : \chi\!\rightsquigarrow$, the computed cyclic term $\mathsf{sem\text{-}inv\text{-}aux}(t)$ is semantically bisimilar to t. First, we prove a weakening principle on the bisimilarity up-to a given depth relation.

Proposition 29 **Weakening principal on $_\sim_n_$**

Let n be a natural number and $t, s : \textsc{co}\mathbb{T}$ be two infinite trees. If $t \sim_{\mathsf{suc}\, n} s$, then $t \sim_n s$.

Proof. Let $n : \mathbb{N}$, $t, s : \textsc{co}\mathbb{T}$ and $H : t \sim_{\mathsf{suc}\, n} s$. By induction on n, we consider two cases:

- **Base step.** Assume that $n = \mathsf{zero}$.
  In this, case, $t \sim_{\mathsf{zero}} s$ holds trivially by constructor $\sim_0$-intro.
- **Inductive step.** Assume that $n = \mathsf{suc}\, n'$ where $n' : \mathbb{N}$.
  The induction hypothesis is given by:

$$\forall (t, s : \textsc{co}\mathbb{T}_S).\ t \sim_{\mathsf{suc}\, n'} s \to t \sim_n s. \tag{IH}$$

We have to show that $t \sim_{\mathsf{suc}\, n'} s$. From elimination of the hypothesis H, we obtain a proof $e : \mathsf{root}\, t = \mathsf{root}\, s$ and a proof $H' : \forall k.\ t(k) \sim_{\mathsf{suc}\, n'} s(e_*(k))$. We use the constructor $\sim_{\mathsf{suc}}$-intro.

Thus, it remains to prove that $\forall k.\ t(k) \sim_{n'} s(e_*(k))$ which is a consequence of the application of $H(k)$ to the induction hypothesis (IH). □

Here, we prove that the image by $[\![_]\!]$ of the computed cyclic term is bisimilar up-to a given depth to the original regular tree. In the following, we fix an arbitrary regular tree $\chi : \textsc{Reg}_S$.

Lemma 5 Soundness of sem-inv-aux

Let n be a natural number, l be a list of indices $\#\chi$, $t : \chi\rightsquigarrow$ be a subtree of χ, Γ be a environment and ρ a closure. Then,

$$\forall (B : \lceil l \rceil).\ (\forall i.\ l[i] \sim_n [\![\Gamma(i)]\!]_\rho) \to t \sim_n [\![\mathsf{sem\text{-}inv\text{-}aux}\ B\ t\ \Gamma]\!]_\rho.$$

Proof. Let $n : \mathbb{N}$, $l : \textsc{List}(\#\chi)$, $t : \chi\rightsquigarrow$, $\Gamma : \textsc{Env}_S(\mathsf{length}\, l)$, $\rho : \textsc{Closure}_S(\mathsf{length}\, l)$, $B : \lceil l \rceil$ and $I : \forall i.\ l[i] \sim_n [\![\Gamma(i)]\!]_\rho$. We proceed by *recursion* on n, thus we may assume the following function:

$$\begin{aligned} &\forall (n : \mathbb{N}).\ \forall (l : \textsc{List}\,\#\chi).\ \forall (B : \lceil l \rceil).\ \forall (t : \chi\rightsquigarrow).\\ &\quad \forall (\Gamma : \textsc{Env}_S(\mathsf{length}\, l)).\ \forall (\rho : \textsc{Closure}_S(\mathsf{length}\, l)).\\ &\qquad (\forall i.\ l[i] \sim_n [\![\Gamma(i)]\!]_\rho) \to t \sim_n [\![\mathsf{sem\text{-}inv\text{-}aux}\ B\ t\ \Gamma]\!]_\rho. \end{aligned} \tag{fix}$$

By case-analysis on $n : \mathbb{N}$, we consider two cases:

- **Case 1.** Assume that $n = \mathsf{zero}$.
  $t \sim_0 [\![\mathsf{sem\text{-}inv\text{-}aux}\ B\ t\ \Gamma]\!]_\rho$ holds by $\sim_0$-intro.
- **Case 2.** Assume that $n = \mathsf{suc}\, n'$ with $n' : \mathbb{N}$.
  By case-analysis on $t \in^{\#}?\ l$, we consider two cases:
  - **Case 2.1.** Assume that $i : \textsc{Fin}(\mathsf{length}\, l)$ and $e : l[i] \approx t$.

$$\begin{aligned} t &\sim_{\mathsf{suc}\, n'} l[i] && \text{(by rewriting } e\text{)}\\ &\sim_{\mathsf{suc}\, n'} [\![\Gamma(i)]\!]_\rho. && \text{(by } I(i)\text{)} \end{aligned}$$

  - **Case 2.2.** Assume that $M : t \notin l$.
    We have to prove that:

$$t \sim_{\mathsf{suc}\, n'} [\![\ \mathsf{rec}\,(\mathsf{root}\, t) \lhd k \mapsto \mathsf{sem\text{-}inv\text{-}aux}\ (M\lceil :: \rceil B)\ (\mathsf{next}\ t\ k)\ (\mathsf{var\,zero}\,;\Gamma)\,]\!]_\rho.$$

    We define $t_{\mathsf{inv}} := \mathsf{rec}\,(\mathsf{root}\, t) \lhd k \mapsto \mathsf{sem\text{-}inv\text{-}aux}\ (M\lceil :: \rceil B)\ (\mathsf{next}\ t\ k)\ (\mathsf{var\,zero}\,;\Gamma)$. Also, we generalize $\mathsf{suc}\, n'$ in both the goal and the assumption I. Thus, the new goal is:

$$\forall (m : \mathbb{N}).\ (\forall i.\ l[i] \sim_m [\![\Gamma(i)]\!]_\rho) \to t \sim_m [\![t_{\mathsf{inv}}]\!]_\rho.$$

    Let $m : \mathbb{N}$ and $I : \forall i.\ l[i] \sim_m [\![\Gamma(i)]\!]_\rho$. We proceed by induction on m:

    - **Base step.** Assume that $m = \mathsf{zero}$.
      $t \sim_0 [\![t_{\mathsf{inv}}]\!]_\rho$ holds trivially by $\sim_0$-intro.
    - **Inductive step.** Assume that $m = \mathsf{suc}\, m'$ with $m' : \mathbb{N}$.
      The induction hypothesis is given by:

$$(\forall i.\ l[i] \sim_{m'} [\![\Gamma(i)]\!]_\rho) \to t \sim_{m'} [\![t_{\mathsf{inv}}]\!]_\rho. \tag{IH}$$

      Clearly, we have $e : \mathsf{root}\, t = \mathsf{root}\, [\![t_{\mathsf{inv}}]\!]_\rho$ by reflexivity. Consequently, we use the constructor $\sim_{\mathsf{suc}}$-intro with e. Thus, it remains to show that:

$$\forall k.\ t(k) \sim_{m'} [\![\mathsf{sem\text{-}inv\text{-}aux}\ (M\lceil :: \rceil B)\ (\mathsf{next}\ t\ k)\ (\mathsf{var\,zero}\,;\Gamma)]\!]_{(t_{\mathsf{inv}} :: \rho)}.$$

Let $k : \text{FIN}(S.\,\mathsf{Ar}(\mathsf{root}\,t))$. We use the recursive function (fix). Here, m' is a subterm of $\mathsf{suc}\,n'$ since m was generalized from $\mathsf{suc}\,n'$. It remains to prove:

$$\forall i.\ (\mathsf{index}\,t :: l)[i] \sim m' [\![(\mathsf{var\ zero}\ ; \Gamma)(i)]\!]_{(t_{\mathsf{inv}} :: \rho)}.$$

Let $i : \text{FIN}(\mathsf{suc}(\mathsf{length}\,l))$. By case-analysis on i, we consider two cases:

- **Case 2.2.1.** Assume that $i = \mathsf{zero}$.

$$\begin{aligned}(\mathsf{index}\,t :: l)[\mathsf{zero}] \equiv\ & \mathsf{value}(\mathsf{index}\,t) \\ \sim_{m'}\ & t && (\mathsf{index} \text{ is a section of } \mathsf{value}) \\ \sim_{m'}\ & [\![t_{\mathsf{inv}}]\!]_{\rho} && (1) \\ \equiv\ & [\![\mathsf{var\ zero}]\!]_{t_{\mathsf{inv}} :: p} \\ \equiv\ & [\![(\mathsf{var\ zero}\ ; \Gamma)(\mathsf{zero})]\!]_{t_{\mathsf{inv}} :: p}.\end{aligned}$$

  where (1) is a consequence of the induction hypothesis (IH) instantiated with hypothesis I (weaken by Prop. 2.29).

- **Case 2.2.2.** Assume that $i = \mathsf{suc}\,i'$ with $i' : \text{FIN}(\mathsf{length}\,l)$.

$$\begin{aligned} & (\mathsf{index}\,t :: l)[\mathsf{suc}\,i'] \\ \equiv\ & \{ \text{ by definition } \} \\ & l[i'] \\ \sim_{m'}\ & \{ \text{ by applying } I(i') \text{ to Prop. 2.29 [Weakening of } \sim_n] \} \\ & [\![\Gamma(i')]\!]_{\rho} \\ \sim\ & \{ \text{ by Prop. 2.21 } [[\![\mathsf{weaken}\,t]\!]_{\rho} \sim [\![t]\!]_{\mathsf{tail}\,\rho}] \} \\ & [\![\mathsf{weaken}(\Gamma(i'))]\!]_{t_{\mathsf{inv}} :: \rho} \\ \equiv\ & \{ \text{ by definition } \} \\ & [\![(\mathsf{var\ zero}\ ; \Gamma)(\mathsf{suc}\,i')]\!]_{t_{\mathsf{inv}} :: \rho}. \end{aligned}$$

□

Remark. The proof of Lemma 2.5 is somewhat subtle because of nested recursion. The outer recursive function is used to show that the subtree $t : \chi{\rightsquigarrow}$ is actually bisimilar up-to depth n to the semantics of the computed cyclic term $\mathsf{sem\text{-}inv\text{-}aux}(t)$. The inner recursive function is used to prove that the invariant, namely $\forall i.\ l[i] \sim_n [\![\Gamma(i)]\!]$, is preserved. Indeed, the semantics of cyclic terms is defined as the infinite unfolding of its cycles. As such, in the proof of preservation, we are presented with a goal that is identical to the main theorem we are trying to prove in the first place. Nevertheless, this works because the depth up-to which the trees have to be proven bisimilar decreases in each invariant preservation proofs and thus, is guaranteed to end eventually.

Finally, we define the function computing the cyclic term of a regular tree:

```
sem-inv : REG_S → ℂ_S
sem-inv  χ  ≡  sem-inv-aux χ B (χ, ε) Γ
    where  B : [[]]
           Γ : FIN zero → ℂ_S
           Γ  ().
```

The proof B is obtained from Proposition A.15.

Now, we prove the completeness result, namely that the semantics function $[\![\,_\,]\!]$ has a section:

Theorem 9 — Completeness of cyclic term representation

For any regular tree $t : \mathrm{REG}_S$, there exists a semantically equivalent cyclic term.

Proof. Let $t : \mathrm{REG}_S$. The cyclic term is obtained by $c := \mathsf{sem\text{-}inv}\, t$. It remains to show that:

$$t \sim [\![\, c \,]\!].$$

We use coinduction proof principle with the bisimulation up-to a given depth. Thus, it suffices to prove that:

$$\forall (n : \mathbb{N}).\ t \sim_n [\![\, \mathsf{sem\text{-}inv}\, t \,]\!].$$

This follows from Lemma 2.5 [Soundness of sem-inv-aux]. The premises are straightforward to prove since the list is empty. □

The theorem above justifies that the semantics function on cyclic terms $[\![\,_\,]\!] : \mathbb{C}_S \to \mathrm{REG}_S$ has an inverse $\mathsf{sem\text{-}inv} : \mathrm{REG}_S \to \mathbb{C}_S$ which we denote by $[\![\,_\,]\!]^{-1}$.

Inductive axiomatization of cyclic term equivalence

Previously, we were interested in the problem of characterizing syntactically regular trees. Let us recall that two closed cyclic terms are considered to be equivalent when their image by $[\![\,_\,]\!]$ yield bisimilar infinite trees. In this section, we show that equivalence of (closed) cyclic terms admits an inductive axiomatization. Ultimately, we obtain a purely inductive characterization of the type of regular trees. For the remaining of this section, we fix an arbitrary signature $S : \mathrm{SIG}$.

In order to prove that such axiomatization exists for closed cyclic terms, we first define an axiom system for cyclic terms with free variables. Let $n : \mathbb{N}$ and $\Gamma : \mathrm{LIST}(\mathbb{C}_S(n) \times \mathbb{C}_S(n))$ be a context composed of pairs of cyclic terms. We say that two cyclic terms t and s are equivalent in Γ, denoted $\Gamma \vdash t \sim^{\mathsf{i}} s$, whenever a proof is derivable from the inductively generated set of inference rules given by:

$$\sim^{\mathsf{i}}_{\in}\ \frac{(t, s) \in \Gamma}{\Gamma \vdash t \sim^{\mathsf{i}} s}, \qquad \sim^{\mathsf{i}}_{\mathsf{var}}\ \frac{x = y}{\Gamma \vdash \mathsf{var}\, x \sim^{\mathsf{i}} \mathsf{var}\, y},$$

$$\sim^{\mathsf{i}}_{\mathsf{rec}}\ \frac{\forall k.\ (\mathsf{rec}\, o \lhd os, \mathsf{rec}\, o' \lhd os') :: \Gamma \vdash os(k)[* := \mathsf{rec}\, o \lhd os] \sim^{\mathsf{i}} os'(e_*(k))[* := \mathsf{rec}\, o' \lhd os']}{\Gamma \vdash \mathsf{rec}\, o \lhd os \sim^{\mathsf{i}} \mathsf{rec}\, o' \lhd os'}.$$

Definition 17 — Closed cyclic term identification relation

Let $t, s : \mathbb{C}_S$ be two cyclic terms. We say that t is identified to s, denoted $t \sim^{\mathsf{i}} s$, if and only if $[\,] \vdash t \sim^{\mathsf{i}} s$ is derivable from the empty context. Formally,

$$t \sim^{\mathsf{i}} s \overset{\text{def}}{\iff} [\,] \vdash t \sim^{\mathsf{i}} s.$$

In the remaining, we prove that the relation $\sim^{\mathsf{i}}$ is sound and complete with respects to the semantically defined equivalence relation on closed cyclic terms.

Soundness proof

A pair of cyclic terms $(t_1, t_2) : \mathbb{C}_S$ is a valid assumption up-to depth n, when:

$$\mathrm{VALID} : \mathbb{N} \to (\mathbb{C}_S \times \mathbb{C}_S) \to \mathrm{PROP}$$
$$\mathrm{VALID}\ \ n\ \ (t_1, t_2)\ \equiv\ [\![\, t_1 \,]\!] \sim_n [\![\, t_2 \,]\!].$$

Lemma 6 **Soundness of inductive axiomatization up-to a given depth**

Let $n : \mathbb{N}$ *and* $\Gamma : \textsc{List}(\mathbb{C}_S \times \mathbb{C}_S)$ *be a context. For any closed terms* $t, s : \mathbb{C}_S$*, we have*

$$\Gamma \vdash t \sim^{\mathsf{i}} s \to \textsc{All}\ (\textsc{Valid}\, n)\ \Gamma \to \textsc{Valid}\ n\ (t, s).$$

Proof. Let $n : \mathbb{N}$, $\Gamma : \textsc{List}(\mathbb{C}_S \times \mathbb{C}_S)$, $t, s : \mathbb{C}_S$, $E : \Gamma \vdash t \sim s$ and $A : \textsc{All}\ (\textsc{Valid}\, n)\ \Gamma$. We proceed by *recursion* on n, thus we assume the following function:

$$\forall n.\ \forall \Gamma.\ \forall (t, s : \mathbb{C}_S).\ \Gamma \vdash t \sim^{\mathsf{i}} s \to \textsc{All}\ (\textsc{Valid}\, n)\ \Gamma \to \textsc{Valid}\ n\ (t, s). \qquad \text{(fix)}$$

By case-analysis on n, we consider two cases:

- **Case 1.** Assume that $n = \mathsf{zero}$.
  $[\![\, t \,]\!] \sim_0 [\![\, s \,]\!]$ holds by $\sim_0$-intro.
- **Case 2.** Assume that $n = \mathsf{suc}\, n'$ with $n' : \mathbb{N}$.
  By case-analysis on E:
  - **Case 2.1.** Assume that $M : (t, s) \in \Gamma$.
    By hypothesis $A : \textsc{All}\ (\textsc{Valid}(\mathsf{suc}\, n'))\ \Gamma$, it suffices to show that $(t, s) \in \Gamma$ which is given by assumption M.
  - **Case 2.2.** Assume that $t = \mathsf{var}\, x$, $s = \mathsf{var}\, y$ where $x, y : \textsc{Fin}\, 0$ and $e : x = y$.
    This case leads to a contradiction since the type of both x and y is empty.
  - **Case 2.3.** Assume that $t = \mathsf{rec}\, o \lhd os$ and $s = \mathsf{rec}\, o' \lhd os'$ where $o : S.\mathsf{Op}$, $o' : S.\mathsf{Op}$, $os : \textsc{Vec}\ (\mathbb{C}_S(1))\ (S.\,\mathsf{Ar}\, o)$, $os' : \textsc{Vec}\ (\mathbb{C}_S(1))\ (S.\,\mathsf{Ar}\, o')$. Furthermore, assume that $e : o = o'$ and $E' : \forall k.\ (t, s) :: \Gamma \vdash os(k)[* := t] \sim os'(e_*(k))[* := s]$.
    By induction on e, we can substitute e by refl and o' by o. Thus, the goal is:

    $$[\![\, \mathsf{rec}\, o \lhd os \,]\!] \sim_{\mathsf{suc}\, n'} [\![\, \mathsf{rec}\, o \lhd os' \,]\!].$$

    We generalize the term $(\mathsf{suc}\, n')$ in both the goal and the hypothesis A. Now, it remains to prove that:

    $$\forall (m : \mathbb{N}).\ \textsc{All}\ (\textsc{Valid}\, m)\ \Gamma \to [\![\, \mathsf{rec}\, o \lhd os \,]\!] \sim_m [\![\, \mathsf{rec}\, o \lhd os' \,]\!].$$

    Let $m : \mathbb{N}$ and $A : \textsc{All}\ (\textsc{Valid}\, m)\ \Gamma$. By induction on m:
    - ▸ **Base step.** Assume that $m = 0$.
      $[\![\, \mathsf{rec}\, o \lhd os \,]\!] \sim_0 [\![\, \mathsf{rec}\, o \lhd os' \,]\!]$ holds trivially by $\sim_0$-intro.
    - ▸ **Inductive step.** Assume that $m = \mathsf{suc}\, m'$ where $m' : \mathbb{N}$.
      The induction hypothesis is given:

      $$\textsc{All}\ (\textsc{Valid}\, m')\ \Gamma \to [\![\, \mathsf{rec}\, o \lhd os \,]\!] \sim_{m'} [\![\, \mathsf{rec}\, o \lhd os' \,]\!]. \qquad \text{(IH)}$$

      Clearly, we have $e' : o = o$ by reflexivity. Consequently, we use the constructor $\sim_{\mathsf{suc}}$-intro with e'. It remains to show that:

      $$\forall k.\ [\![\, os(k)[* := \mathsf{rec}\, o \lhd os] \,]\!] \sim_{m'} [\![\, os'(k)[* := \mathsf{rec}\, o \lhd os'] \,]\!].$$

      Let $k : \textsc{Fin}(S.\,\mathsf{Ar}\, o)$. We use the recursive function (fix). Here, m' is a subterm of $\mathsf{suc}\, n'$ since m was generalized from $\mathsf{suc}\, n'$. It remains to prove:

      $$\textsc{All}\ (\textsc{Valid}\, m')\ ((\mathsf{rec}\, o \lhd os, \mathsf{rec}\, o \lhd os') :: \Gamma).$$

We use the constructor $\mathsf{all}_{::}$, thus it remains to prove that:

- $\textsc{Valid}\ m'\ (\mathsf{rec}\,o \lhd os, \mathsf{rec}\,o \lhd os')$.
  We use the induction hypothesis (IH) along with the proof of $\textsc{All}(\textsc{Valid}\,m')\ \Gamma$ given below.
- $\textsc{All}\ (\textsc{Valid}\,m')\ \Gamma$:
  We have $A : \textsc{All}\ (\textsc{Valid}(\mathsf{suc}\,m'))\ \Gamma$, thus by functoriality of $\textsc{All}$, it suffices to show that:
  $$\forall t.\ \forall s.\ \textsc{Valid}\ (\mathsf{suc}\,m')\ (t,s) \rightarrow \textsc{Valid}\ m'\ (t,s).$$
  This is a consequence of Proposition 2.29 [Weakening of $\sim_n$]. □

Theorem 10 Soundness of inductive axiomatization

Let $t, s : \mathbb{C}_S$ be two closed cyclic terms. We have,
$$t \sim^{\mathsf{i}} s \rightarrow [\![\, t \,]\!] \sim [\![\, s \,]\!].$$

Proof. Let $t, s : \mathbb{C}_S$ and $E : [\,] \vdash t \sim^{\mathsf{i}} s$. We use the coinduction proof principle (Theorem 2.1) with the bisimulation up-to a given depth (Theorem 2.2). Thus, it suffices to prove that:
$$\forall (n : \mathbb{N}).\ [\![\, t \,]\!] \sim_n [\![\, s \,]\!].$$
This follows from E applied to Lemma 2.6 E. Finally, it remains to show that $\textsc{All}\ (\textsc{Valid}\,n)\ [\,]$. This holds immediately with constructor $\mathsf{all}_{[\,]}$. □

Completeness proof

The completeness proof is slightly more involved. Let $t_1, t_2 : \mathbb{C}_S$ be two closed terms. By Theorem 2.7, the set of subterms of both t_1 and t_2 are weakly finitely indexed. Thus there exist two maps $\mathsf{index}_1 : t_1{\leadsto} \rightarrow \textsc{Fin}(\#t_1{\leadsto})$ and $\mathsf{index}_2 : t_2{\leadsto} \rightarrow \textsc{Fin}(\#t_2{\leadsto})$. Furthermore, both index_1 and index_2 have a left-inverse, denoted respectively by $\mathsf{value}_1 : \textsc{Fin}(\#t_1{\leadsto}) \rightarrow t_1{\leadsto}$ and $\mathsf{value}_2 : \textsc{Fin}(\#t_2{\leadsto}) \rightarrow t_2{\leadsto}$. Now, we lift all the above functions to product of subterms. Thus, we define $T := (t_1, t_2) : \mathbb{C}_S \times \mathbb{C}_S$ and $\#T := \#t_1{\leadsto} \times \#t_2{\leadsto} : \mathbb{N}$. We lift the indexing on T as follows:

$$\begin{array}{l}\mathsf{Index} : T \rightarrow \textsc{Fin}\,\#T \\ \mathsf{Index}\ \ (t_1, t_2) \equiv (\mathsf{index}_1(t_1), \mathsf{index}_2(t_2)),\end{array}$$

and the value function:

$$\begin{array}{l}\mathsf{Value} : \textsc{Fin}\,\#T \rightarrow T \\ \mathsf{Value}\ \ (i_1, i_2)\ \equiv (\mathsf{value}_1(t_1), \mathsf{value}_2(t_2)).\end{array}$$

Note that, we have *silently* exploited the isomorphism $\textsc{Fin}(m \times n) \cong \textsc{Fin}\,m \times \textsc{Fin}\,n$ of Prop. 1.6 in the definition of both Index and Value.

It is straightforward to check that Index is a right-inverse of Value. Thus, we have
$$\forall (t : T).\ \mathsf{Value}(\mathsf{Index}\,t) \approx t. \qquad \text{(Value-Index)}$$

Lemma 7 Completeness of inductive axiomatization

Let l be a list of indices $\#T$, and Γ be a list of pair of closed cyclic terms. Moreover, let $s_1 : t_1{\leadsto}$ and $s_2 : t_2{\leadsto}$ be two subterms of t_1 and t_2 respectively. Then,
$$\forall (B : \lceil l \rceil).\ (\forall i.\ \mathsf{Value}(l[i]) \in \Gamma) \rightarrow [\![\, s_1 \,]\!] \sim [\![\, s_2 \,]\!] \rightarrow \Gamma \vdash s_1 \sim^{\mathsf{i}} s_2.$$

Proof. Let $l : \textsc{List}\,\#T$, $\Gamma : \textsc{List}\,T$, $s_1 : t_1\rightsquigarrow$, $s_2 : t_2\rightsquigarrow$, $B : [l]$. Moreover, let $I : \forall i.\ \mathsf{Value}(l[i]) \in \Gamma$ and $E : [\![s_1]\!] \sim [\![s_2]\!]$. We proceed by induction on B.

▶ **Base step.** Inductive type without a base constructor.

▶ **Inductive step.** The induction hypothesis is given by

$$\begin{aligned}&\forall(l' : \textsc{List}\,\#T).\ l' \succ l \to \forall\Gamma.\ \forall(s_1' : t_1\rightsquigarrow).\ \forall(s_2' : t_2\rightsquigarrow).\\ &\qquad(\forall i.\ \mathsf{Value}(l'[i]) \in \Gamma) \to [\![s_1']\!] \sim [\![s_2']\!] \to \Gamma \vdash s_1' \sim s_2'.\end{aligned} \tag{IH}$$

By case-analysis on $\mathsf{Index}(s_1, s_2) \in^? l$, we consider two cases:

- **Case 1.** Assume that $M : \mathsf{Index}(s_1, s_2) \in l$.
  By rule $\sim^{\mathsf{i}}_{\in}$, to prove $\Gamma \vdash s_1 \sim^{\mathsf{i}} s_2$, it suffices to show that $(s_1, s_2) \in \Gamma$. From the hypothesis M and the decidability of equality on $\textsc{Fin}\,\#T$, we can compute an index $i : \textsc{Fin}(\mathsf{length}\,l)$ such that $l[i] \approx \mathsf{Index}(s_1, s_2)$. From that and with I, we can deduce that $(s_1, s_2) \in \Gamma(i)$.

- **Case 2.** Assume that $N : \mathsf{Index}(s_1, s_2) \notin l$.
  By case-analysis on both s_1 and s_2, we have:

  - **Case 2.1.** Assume that $s_1 = \mathsf{var}\,x$ or $s_2 = \mathsf{var}\,y$ where $x, y : \textsc{Fin}\,0$.
    This case leads to a contradiction since the type of both x and y is empty.

  - **Case 2.2.** Assume that $s_1 = \mathsf{rec}\,o \lhd os$ and $s_2 = \mathsf{rec}\,o' \lhd os'$ where $o : S.\mathsf{Op}$, $o' : S.\mathsf{Op}$, $os : \textsc{Vec}\ (\mathbb{C}_S(1))\ (S.\mathsf{Ar}\,o)$, $os' : \textsc{Vec}\ (\mathbb{C}_S(1))\ (S.\mathsf{Ar}\,o')$.
    By rule $\sim^{\mathsf{i}}_{\mathsf{rec}}$, to show that $\Gamma \vdash \mathsf{rec}\,o \lhd os \sim^{\mathsf{i}} \mathsf{rec}\,o' \lhd os'$, it suffices to prove:

    - $e : o = o'$.
      By assumption, we have $E : [\![\mathsf{rec}\,o \lhd os]\!] \sim [\![\mathsf{rec}\,o' \lhd os']\!]$, thus by elimination of E, we have $o = o'$.

    - $\forall k.\ (s_1, s_2) :: \Gamma \vdash os(k)[* := s_1] \sim^{\mathsf{i}} os'(e_*(k))[* := s_2]$.
      Let $k : \textsc{Fin}(S.\mathsf{Ar}\,o)$. We use the induction hypothesis (IH) with l' defined to be $\mathsf{Index}(s_1, s_2) :: l$. It remains to show:

      - $\mathsf{Index}(s_1, s_2) :: l \succ l$.
        This follows from hypothesis $N : \mathsf{Index}(s_1, s_2) \notin l$.

      - $p_1 : t_1 \rightsquigarrow os(k)[* := s_1]$ and $p_2 : t_2 \rightsquigarrow os'(e_*(k))[* := s_2]$.
        By assumption, we have $t_1 \rightsquigarrow s_1$. By transitivity of the subterm relation, it suffices to show that $s_1 \rightsquigarrow os(k)[* := s_1]$ which is clearly true. The same reasoning is used to prove $t_2 \rightsquigarrow os'(e_*(k))[* := s_2]$.

      - $\forall i.\ \mathsf{Value}((\mathsf{Index}(s_1, s_2) :: l)(i)) \in (s_1, s_2) :: \Gamma$.
        Let $i : \textsc{Fin}(\mathsf{suc}(\mathsf{length}\,l))$. We consider two cases:

        - **Case 2.2.1.** Assume that $i = \mathsf{zero}$.
          We have to prove that $\mathsf{Value}(\mathsf{Index}(s_1, s_2)) \in (s_1, s_2) :: \Gamma$. By constructor here, it suffices to show that $\mathsf{Value}(\mathsf{Index}(s_1, s_2)) \approx (s_1, s_2)$ which follows from Equation (Value-Index).

- **Case 2.2.2.** Assume that $i = \mathsf{suc}\, i'$ where $i' : \textsc{Fin}(\mathsf{length}\, l)$.
  We have to prove that $l[i] \in (s_1, s_2) :: \Gamma$. By using constructor there, it suffices to show that $l[i] \in \Gamma$ which holds by assumption $I(i')$.

- $⟦ os(k)[* := s_1] ⟧ \sim ⟦ os'(e_*(k))[* := s_2] ⟧$.
  This follows from the elimination of the assumption E. □

Theorem 11 **Completeness of inductive axiomatization**

Let $t, s : \mathbb{C}_S$ be two closed cyclic terms. We have,

$$⟦ t ⟧ \sim ⟦ s ⟧ \to t \sim^{\mathsf{i}} s.$$

Proof. Let $t, s : \mathbb{C}_S$ and $E : ⟦ t ⟧ \sim ⟦ s ⟧$. To prove $[\,] \vdash t \sim^{\mathsf{i}} s$, we use Lemma 2.7 with E. Thus, it remains to prove:

- $\lceil [\,] \rceil$.
  This follows from Proposition A.15.
- $\forall i.\ [\,][i] \in [\,]$.
  Let $i : \textsc{Fin}\, 0$. The type of i is empty. Contradiction. □

Summary

Let $S : \textsc{Sig}$ be a arbitrary signature. So far, we considered two equivalence relations on closed cyclic terms. The first one is semantic:

$$t \sim^{\mathsf{s}} s \overset{\text{def}}{\iff} ⟦ t ⟧ \sim ⟦ s ⟧,$$

while the second one is syntactic:

$$t \sim^{\mathsf{i}} s \overset{\text{def}}{\iff} [\,] \vdash t \sim^{\mathsf{i}} s.$$

We proved that both relations, namely $\sim^{\mathsf{i}}$ and $\sim^{\mathsf{s}}$, are in fact equivalent. In addition, we showed that the type of closed cyclic terms $\mathbb{C}_S$ constitutes a sound and complete syntax for the type of regular tree $\textsc{Reg}_S$. All these results are summarized in the following theorem.

Theorem 12 **Cyclic term and regular tree isomorphism**

Let S be a signature. We have the following setoid isomorphisms:

$$(\mathbb{C}_S, \sim^{\mathsf{i}}) \cong (\mathbb{C}_S, \sim^{\mathsf{s}}) \cong (\textsc{Reg}_S, \sim).$$

Proof. Let $S : \textsc{Sig}$. By transitivity:

- $(\mathbb{C}_S, \sim^{\mathsf{i}}) \cong (\mathbb{C}_S, \sim^{\mathsf{s}})$.
  We have two maps $f, g : \mathbb{C}_S \to \mathbb{C}_S$ defined to be the identity function. Clearly, from Theorems 2.10 and 2.11, we have that $\sim^{\mathsf{i}} \leftrightarrow \sim^{\mathsf{s}}$ and thus f and g are setoid morphisms. It is straightforward to check that f is the inverse of g.

- $(\mathbb{C}_S, \sim^{\mathsf{s}}) \cong (\textsc{Reg}_S, \sim)$.
  The first map $f : \mathbb{C}_S \to \textsc{Reg}_S$ is given by $⟦_⟧$. Clearly, it is a setoid morphism. The map $g : \textsc{Reg}_S \to \mathbb{C}_S$ is defined to be $⟦_⟧^{-1}$. It is also a setoid morphism. Finally, it straightforward to check that f is the inverse of g. □

2.3. CONCLUSION

In the first part of this chapter, we have defined, formally, the type of regular trees over an arbitrary signature. In particular, we showed that the regularity property can be characterized equivalently through the set of paths in a tree or through the set of its subtrees. Such characterizations were explored, for instance, in the work of [Gin79, Nel83, Cou83]. Let us recall that regular trees are characterized through a finiteness property. However, as shown in the previous chapter, various *incomparable* definitions of finite can be considered in a constructive setting. Consequently, the choice of one particular characterization of finite types (or setoids) is important. Here, we have chosen a definition of finite that is weak enough to be able to characterize regular trees without assuming decidable equality on nodes of the trees, but still strong enough constructively to define meaningful computations and prove that regular trees are closed under tree destructors.

In the second part of this chapter, we studied a syntactic characterization of regular trees by means of cyclic terms. We used a representation of cycles through binders following the work of [GHUV06]. Then, we defined the semantics of cyclic terms as the infinite unfolding of their cycles. These induced infinite trees were then proven to be regular. Furthermore, we have also studied the converse problem, namely to *compute* a cyclic term representation of a regular tree.

Finally, we have proposed an inductive axiomatization of the equivalence of cyclic terms. In it is interesting to note that such axiomatization was studied by [BH97], dealing with the problem of defining the subtyping relation for iso-recursive types. In addition, these results were mechanized in the proof assistant AGDA by [DA10] to illustrate the usage of mixed inductive-coinductive style. However, such reasoning principle is not available in the COQ proof assistant and thus their approach was not directly applicable. Altogether, we proved that the type of cyclic terms is isomorphic to the type of regular trees, thus yielding a sound and complete syntactic characterization of regular trees.

CHAPTER THREE

TREE TRANSDUCERS

In this chapter, we study the problem of defining regular trees homomorphisms, *i.e.*, morphisms preserving the regularity property of trees. Our goal is to find a syntactic criterion ensuring, *by construction*, that corecursive definitions preserve this property. To this end, we use the formalism of tree transducers as a mean to represent the abstract syntax of (co)recursive functions. Then, tree transducers are reified into tree morphisms preserving regularity.

3.1. MOTIVATING EXAMPLE

A regular tree homomorphism between an input signature I and an output signature O is a morphism $\phi : \text{co}\mathbb{T}_I \to \text{co}\mathbb{T}_O$ preserving regularity. As an example, consider the following two mutually corecursive functions on streams of A's:

$$\begin{aligned} &\phi_1 : \textsc{Stream}\, A \to \textsc{Stream}\, A && \phi_2 : \textsc{Stream}\, A \to \textsc{Stream}\, A \\ &\phi_1 \;\; (x :: xs) \;\equiv\; f_1(x) :: \phi_2(xs) && \phi_2 \;\; (x :: xs) \;\equiv\; f_2(x) :: \phi_1(xs) \end{aligned}$$

where f_1, f_2 are two arbitrary self-maps on A. Showing that both ϕ_1 and ϕ_2 preserve the regularity property amounts to prove that, given a regular tree s, the type of its subtrees $\phi_i(s)\!\rightsquigarrow$ is weakly finitely indexed. One way to give such a proof is to use the result of Proposition 1.13 which shows that the finiteness property is transported along weak forms of split epimorphisms. Consequently, if we can prove that the type $\phi_i(s)\!\rightsquigarrow$ is a weak retract of $s\!\rightsquigarrow$, we will obtain a proof that ϕ_i preserves the regularity property. Nonetheless, such proof can be quite tedious and requires to construct a specific indexing function.

However, through a *syntactic* observation of the definitions of ϕ_i, it is easy to see that they ought to preserve regularity. Essentially, each function ϕ_i consists in applying the function f_i to the head of the input stream. Consequently, if the input stream σ is regular, *i.e.*, ultimately periodic, its image $\phi_i(\sigma)$ ought to be regular as well.

On the other hand, let's observe through another example a tree morphism failing to preserve regularity. For $i : \mathbb{N}$, we define the family of corecursive functions ψ_i as follows:

$$\begin{aligned} &\psi_i : \textsc{Stream}\, \mathbb{N} \to \textsc{Stream}\, \mathbb{N} \\ &\psi_i \;\; (x :: xs) \;\equiv\; f_i(x) :: \psi_{i+1}(xs). \end{aligned}$$

If we define, for all x, $f_i(x) := i$, then, for any input stream σ, we obtain a stream $\psi_0(\sigma) \sim \omega$ which is clearly not regular.

The difference between the two examples is that the family ψ_i is defined by means of an *infinite number of corecursive equations*. This result is not surprising. The regularity property may be characterized—equivalently—by the solutions of *finite* systems of guarded equations

as presented in [Cou83, Blo83, AAV01]. Here, in the definition of ψ_i, by discarding the input stream, we have actually defined an *infinite* system which cannot be reduced to a finite system.

3.2. TOP-DOWN TREE TRANSDUCERS

In this section, we study the formalism of top-down tree transducers [Rou68, Tha70, Rou70] as a tool to define regularity preserving tree morphisms. First, we recall the set-theoretic definitions of the formalism of tree transducers but restricted to the case of finite trees which is usually the case in the literature. Next, we give a type-theoretic translation of this formalism and extend its semantics to the denotational model of infinite trees.

Finally, we conclude this section with some extension to the formalism of tree transducers in order to handle more powerful tree transformations.

3.2.1. Definitions and Semantics

Top-down tree transducers is a formalism providing a formal model for syntax-directed semantics [FV98, CDG+07]. Tree transducers can be used to specify tree transformations in a formalism resembling attributed grammar with synthesized attributes. Here, we recall the (set-theoretic) definitions of top-down tree transducers and its semantics as found in the literature.

Notations. For all $k : \mathbb{N}$, we write $X_k = \{x_1, \ldots, x_k\}$ for a set of variables with k elements. Given a ranked alphabet Σ (signature) and a set X, the tree algebra, denoted $\mathbb{T}_\Sigma(X)$, is defined inductively, as follows:

$$\frac{x \in X}{x \in \mathbb{T}_\Sigma(X)}, \qquad \frac{\sigma \in \Sigma^{(k)} \qquad t_1, \ldots, t_k \in \mathbb{T}_\Sigma(X)}{\sigma(t_1, \ldots, t_k) \in \mathbb{T}_\Sigma(X)}.$$

When the type of variables X is the empty set, we write $\mathbb{T}_\Sigma$ instead of $\mathbb{T}_\Sigma(\varnothing)$. The set $\Sigma^{(k)}$ where Σ is a ranked alphabet and $k \in \mathbb{N}$ denotes the subset of function symbols in Σ of arity k. Given a tree $t \in \mathbb{T}_\Sigma(X_k)$, $k \geq 1$ and trees $s_1, \ldots, s_k \in \mathbb{T}_\Sigma$, we write $t[x_1 := s_1, \ldots, x_k := s_k]$ for the operation consisting in substituting each variable x_i in t by s_i. Given a family of trees $(s_i)_{i \in \{1,\ldots,k\}}$, we may abbreviate $t[x_1 := s_1, \ldots, x_k := s_k]$ as $t[x_i := s_i \mid i \in \{1, \ldots, k\}]$. When $k = 0$, the substitution operation is the identity. We denote by $\mathcal{C}_\Sigma$, the set of one-hole contexts over the ranked alphabet Σ, *i.e.*, trees $t \in \mathbb{T}_\Sigma(\square)$ where $\square$ appears exactly once in t. Given a context $\beta \in \mathcal{C}_\Sigma$ and a tree $t \in \mathbb{T}_\Sigma$, the operation replacing the hole $\square$ with t is denoted as $\beta[t]$.

Definition 1 **Top-down tree transducer [FV98]**

A top-down tree transducer (tdtt) is defined as a tuple $T = \langle Q, \Sigma, \Delta, q_0, R \rangle$ where:

- *Q is a finite unary ranked alphabet, called the states of the transducer,*
- *Σ, Δ denote an input and output ranked alphabet respectively,*
- *$q_0 \in Q$ is a designated element of Q, called the initial state,*
- *R is a finite set of rewrite rules of the following form:*

$$q(\sigma(x_1, \ldots, x_k)) \to \xi$$

where $q \in Q$, $\sigma \in \Sigma^{(k)}$, $x_1, \ldots, x_k \in X_k$ and $\xi \in \mathbb{T}_\Delta(QX_k)$.

The set of rewrite rules is said to be deterministic (resp. total) when for each left-hand side $q(\sigma(x_1, \ldots, x_k))$, there is at most (resp. at least) one applicable rule. When a tdtt is both deterministic and total, the right-hand side of a rewrite rule is uniquely determined by its left-hand

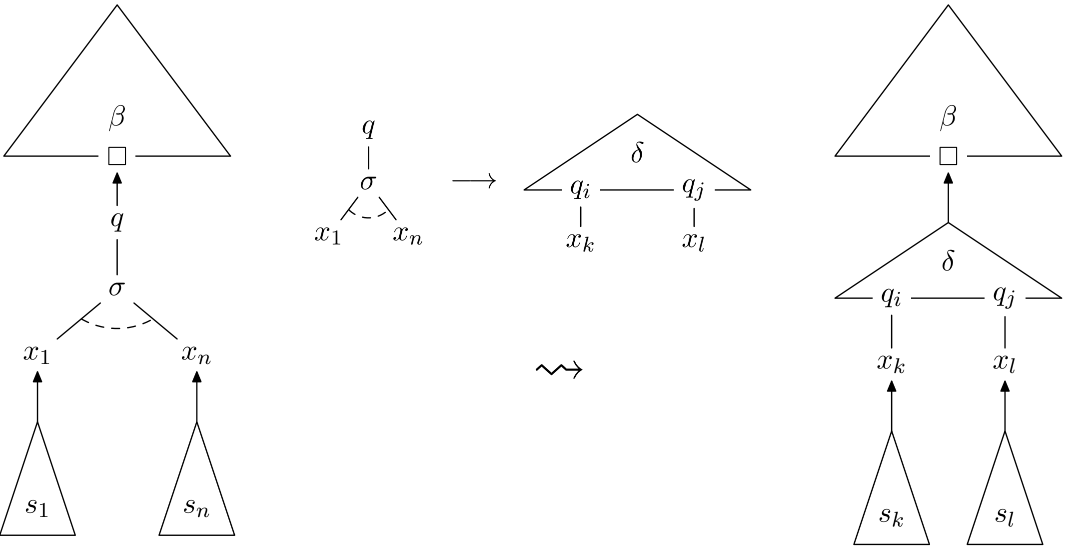


Figure 3.1. A derivation step induced by T.

side. In this case, given a state $q \in Q$ and a function symbol $\sigma \in \Sigma^{(k)}$, we write $rhs(q, \sigma)$ to denote the right-hand side of the rewrite rule $q(\sigma(x_1, \ldots, x_k)) \to \xi \in R$. From now on, we consider *only* total deterministic tdtt.

The semantics of a top-down tree transducer $T = \langle Q, \Sigma, \Delta, q_0, R\rangle$ is usually expressed as a *term rewriting system*.

Definition 2 Induced derivation relation

The derivation relation induced by T, is a binary relation, denoted $\Rightarrow_T$, over the set $\mathbb{T}_{Q\cup\Sigma\cup\Delta}$, and is defined as follows:

$$\frac{\beta \in \mathcal{C}_{Q\cup\Sigma\cup\Delta} \qquad q \in Q \qquad \sigma \in \Sigma^{(k)} \qquad s_1, \ldots, s_k \in \mathbb{T}_\Sigma \qquad \xi = rhs(q, \sigma)}{\beta[q(\sigma(s_1, \ldots, s_k))] \Rightarrow \beta[\xi[x_1 := s_1, \ldots, x_k := s_k]]}.$$

Figure 3.1 illustrates one step of derivation.

Definition 3 Language of a tdtt

The language of the top-down tree transducer T, denoted $\mathcal{L}(T)$, is defined as

$$\mathcal{L}(T) = \{\, (r, s) \in \mathbb{T}_\Sigma \times \mathbb{T}_\Delta \mid q_0(r) \Rightarrow_T^* s \,\}$$

where $\Rightarrow_T^*$ is the reflexive-transitive closure of the derivation relation induced by T.

When a tdtt T is both deterministic and total, it can be shown that the derivation relation induced by T is both confluent and normalizing (see [FV98] for details). Consequently, each state of the transducer T induces a tree morphism, which can be characterized inductively:

Definition 4 Induced tree morphism

Let $T = \langle Q, \Sigma, \Delta, q_0, R\rangle$ be a deterministic and total tdtt. For every $q \in Q$, the induced tree morphism, denoted τ_T^q, is defined as follows:

$$\begin{aligned} \tau_T^q \;&:\; \mathbb{T}_\Sigma \to \mathbb{T}_\Delta \\ \sigma(s_1, \ldots, s_k) &\mapsto rhs(q, \sigma)[(q'_i, x_i) := \tau_T^{q'_i}(s_i) \mid i \in \{\, 1, \ldots, k \,\}]. \end{aligned}$$

Remark. Let T be a total deterministic tdtt, the language of T, namely $\mathcal{L}(T)$, is precisely the graph of the induced morphism $\tau_T^{q_0}$, where q_0 is the initial state of the transducer T.

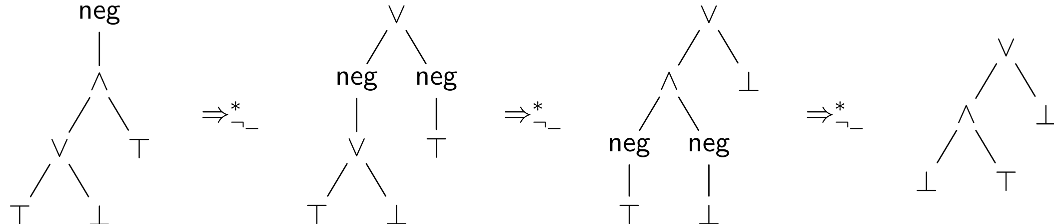
Example. As a first example, we consider a tree transducer which defines the negation operation on a fragment of the propositional calculus. Consider the ranked alphabet P, describing the syntax of the propositional calculus (without propositional variables):

$$P = \{ \bot^{(0)}, \top^{(0)}, \wedge^{(2)}, \vee^{(2)} \}.$$

We denote $\mathcal{P} = \mathbb{T}_P$ the term algebra induced by P. We define the negation operation as a single-state tree transducer $N = \langle \{\neg\}, P, P, \neg, R \rangle$ where the set of rules R is defined as follows:

$$\begin{aligned} \mathsf{neg}(\bot) &\to \top, \\ \mathsf{neg}(\top) &\to \bot, \\ \mathsf{neg}(\wedge(p_1, p_2) &\to \vee(\mathsf{neg}(p_1), \mathsf{neg}(p_2)), \\ \mathsf{neg}(\vee(p_1, p_2)) &\to \wedge(\mathsf{neg}(p_1), \mathsf{neg}(p_2)). \end{aligned}$$

Now, we define $\neg_ : \mathcal{P} \to \mathcal{P}$ as $\neg p := \tau_N^{\mathsf{neg}}(p)$, where τ is given by Definition 3.4. The following figure gives an example of an application of successive derivation steps induced by $\neg_$ on a tree.



Remark. Notice that the formalism of total deterministic tdtt shares syntactic similarities with a restricted form of pattern-matching in equations. Indeed, if we consider the problem of defining the function neg by means of a recursive function, we would write:

$$\begin{array}{lll} \mathsf{neg}' \; _ : \mathbb{T}_P \to \mathbb{T}_P & & \\ \mathsf{neg}' \; \bot & \equiv & \top \\ \mathsf{neg}' \; \top & \equiv & \bot \\ \mathsf{neg}' \; (\wedge(p_1, p_2)) & \equiv & \vee(\mathsf{neg}'\, p_1, \mathsf{neg}'\, p_2) \\ \mathsf{neg}' \; (\vee(p_1, p_2)) & \equiv & \wedge(\mathsf{neg}'\, p_1, \mathsf{neg}'\, p_2). \end{array}$$

Consequently, tdtt can be thought of as *an abstract syntax describing a subset of recursive functions defined by pattern-matching.*

3.2.2. Tree Transducers in Dependent Type Theory

In this section, we define the *type* of total deterministic tdtt and their semantics on finite trees. The definitions follow closely the ones presented in the previous section (based on the set-theoretic foundation). Here and in the subsequent sections, we make an extensive use of the free monad type induced by a signature (see Section 1.3.1). Given a signature O and a type of variables X, we denote by $O^*(X)$, the free monad induced by O and X. In addition, we write $O^+(X)$ to mean $O(O^*(X))$ and $O^n(X)$ to mean $\underbrace{O \cdots O}_{n \text{ times}}(X)$.

Definition 5 **Type of Top-Down Tree Transducers**

Let I, O be two signatures. The type of top-down tree transducer from I to O, denoted TDTT I O*, is defined as a dependent record as follows:*

$$
\begin{array}{l}
\textbf{record}\ \mathrm{TDTT}\ (I\ O : \textsc{Sig}) : \textsc{Type} \\
\quad \textbf{constructor}\ \langle _, _, _ \rangle \\
\qquad \left[\begin{array}{l}
\mathsf{state} : \textsc{Type} \\
\mathsf{init} : \mathsf{state} \\
\mathsf{rhs} : \mathsf{state} \to (i : I.\mathsf{Op}) \to O^*(\mathsf{state} \times \textsc{Fin}(I.\,\mathsf{Ar}\ i)).
\end{array} \right.
\end{array}
$$

When we introduced the type of tdtt, we followed an "unbundled approach": the type TDTT is indexed by the input and output signatures. The main benefit of this approach is that we can easily constrain the input/output signatures.

Another important point is that, in the definition TDTT, the state space is not required to be finite. However, when such an assumption is required, it will be mentioned explicitly.

The record field rhs is defined as a function (and not a relation) from state and function symbol to right-hand sides of rewrite rules. Therefore, the type TDTT actually characterizes *total deterministic* top-down tree transducers. Note that, the set of variables (X_k is the previous section) is canonical and is replaced by $\textsc{Fin}(O.\,\mathsf{Ar}\ i)$.

Example. The tree transducer N encoding the negation of a formula of propositional calculus is defined as follows. First, we define the signature P:

$$
\begin{array}{l}
P : \textsc{Sig} \\
P \ \equiv\ op \lhd ar \\
\quad \textbf{where}\ \ \textbf{inductive}\ op : \textsc{Type} \\
\qquad\qquad \bot \mid \top \mid \wedge \mid \vee : op \\
\qquad\quad ar : op \to \mathbb{N} \\
\qquad\quad ar\ \ \bot\ \equiv\ 0 \qquad ar\ \ \wedge\ \equiv\ 2 \\
\qquad\quad ar\ \ \top\ \equiv\ 0 \qquad ar\ \ \vee\ \equiv\ 2.
\end{array}
$$

Then, the definition of N is given by:

$$
\begin{array}{l}
N : \mathrm{TDTT}\ P\ P \\
N\ \equiv\ \langle \textsc{Unit}, \mathsf{tt}, \mathsf{rw\text{-}rules} \rangle \\
\quad \textbf{where}\ \ \mathsf{rw\text{-}rules} : \textsc{Unit} \to (o : op) \to P^*(\textsc{Unit} \times ar(o)) \\
\qquad\quad \mathsf{rw\text{-}rules}\ \ neg\ \ \bot\ \equiv\ \top \lhd [\,] \\
\qquad\quad \mathsf{rw\text{-}rules}\ \ neg\ \ \top\ \equiv\ \bot \lhd [\,] \\
\qquad\quad \mathsf{rw\text{-}rules}\ \ neg\ \ \wedge\ \equiv\ \vee \lhd \mathsf{var}(neg, \mathsf{zero}) :: \mathsf{var}(neg, \mathsf{suc\ zero}) :: [\,] \\
\qquad\quad \mathsf{rw\text{-}rules}\ \ neg\ \ \vee\ \equiv\ \wedge \lhd \mathsf{var}(neg, \mathsf{zero}) :: \mathsf{var}(neg, \mathsf{suc\ zero}) :: [\,].
\end{array}
$$

As illustrated in the example above, defining both the signature P and the tree transducer N is not so practical (syntactically speaking). Nevertheless, we could define a concrete syntax for both signatures and tree transducers which would considerably ease their definitions. However, it turns out that this syntax already exists in the language on which this mechanization is based: signatures are the abstract syntax of strictly positive inductive definitions (restricted to finite arities) and tdtt are the abstract syntax of a restricted form of recursive functions.

In the COQ proof assistant, the translation from the concrete syntax to the abstract syntax would have to be done through a plugin because at this time, we don't have access, in the language itself, to the abstract syntax of declarations (such as inductive types or functions). Though, in the AGDA proof assistant, thanks to the reflection module, such procedure could be defined directly through a meta-program. Furthermore, recent additions in AGDA allows (co)recursive functions to be defined through this reflection library.

As mentioned in the previous section, the induced tree morphisms of total deterministic tdtts admit an inductive characterization. Thus, it is straightforward to translate Definition 3.4 by means of a recursive function definition.

Definition 6 Induced tree morphism

Let I, O be two signatures and T : TDTT I O be a tree transducer. The induced tree morphism starting from a designated state is defined as follows:

$$\tau_T : T.\mathsf{state} \times \mathbb{T}_I \to \mathbb{T}_O$$
$$\tau_T\ \ (q, o \lhd os)\ \equiv\ T.\mathsf{rhs}\ q\ o \ggg= \lambda(q', k).\ \tau_T(q', os(k)).$$

Moreover, we define the semantics of the tdtt T as tree morphisms:

$$(\!|T|\!) : \mathbb{T}_I \to \mathbb{T}_O$$
$$(\!|T|\!)\ \ t\ \equiv\ \tau_T(T.\mathsf{init}, t).$$

Remark. Here, in the definition of τ_T, we assumed that $os(k)$ is a subterm of os.

3.2.3. Towards Tree Transducers on Infinite Trees

In this section, we discuss how to extend the formalism of top-down tree transducers in order to operate on infinite trees. As discussed previously, top-down tree transducers can be thought of as a way to describe the abstract syntax of recursive functions. When extended to infinite trees, this formalism can be used as the abstract syntax of a subset of corecursive functions.

Let us recall that, for tdtt on finite trees, the type of the right-hand sides of rewrite rules is given by the term algebra $\mathbb{T}_\Delta(Q \times X)$, where Δ is the output ranked alphabet, Q is the state space and X represents of a set variables. Notice that, in the case of finite trees, right-hand sides are allowed to be non-productive, *i.e.*, it is possible to write rules of the following form:

$$q(\sigma(x_1, \ldots, x_k)) \to q(x_i).$$

Here, q denotes a state of the tdtt and σ is a function symbol of the input signature. Such rules are not problematic because finite trees are inductively defined values. Consequently, the leaf function symbols ought to have an arity of 0. In this case, rewrite rules operating on nullary function symbols cannot contain state variables on the right-hand sides. Therefore, even with unproductive rewrite rules, the induced tree morphism yields a total function. Indeed, a stateless rewrite rule will always be reached *eventually*. Moreover, termination is ensured due to the fact that the argument of each recursive call—modeled as a state in the formalism of tdtt—is always a subtree of the input tree. This rule is enforced by the type of the right-hand sides.

However, when we consider infinite trees, this is no longer the case. Indeed, if such rules were allowed, we could obtain a non-productive cycle, that is, an infinite sequence of derivations where no output is ever produced. For instance, consider the following rewrite rule:

$$q(\sigma(x)) \to q(x). \qquad (*)$$

Then, when we apply successively the rule $(*)$ to the stream σ^ω:

$$q(\sigma\sigma\sigma\cdots) \rightsquigarrow q(\sigma\sigma\cdots) \rightsquigarrow q(\sigma\cdots) \rightsquigarrow q(\cdots) \rightsquigarrow \cdots \rightsquigarrow q(\cdots) \rightsquigarrow \cdots,$$

we obtain an infinite non-productive derivation. This is precisely for the same reasons that such rules are forbidden in corecursion definition in the proof assistants Coq or Agda. In order to forbid such problematic rules, we consider two solutions.

The first solution consists in characterizing the rules that are productive. Let $T : \mathsf{TDTT}\ I\ O$ be a top-down tree transducer as given by Definition 3.5. We define the subset of productive right-hand sides, denoted $\mathsf{Prod}_T^{\mathsf{rhs}} : \mathbb{T}_O(T.\mathsf{state} \times \alpha) \to \mathrm{Prop}$, as follows:

$$\mathsf{now}\ \frac{o : O.\mathsf{Op} \qquad os : \mathrm{Vec}\ (\mathbb{T}_O(T.\mathsf{state} \times \alpha))\ (O.\mathsf{Ar}\, o)}{o \triangleleft os \in \mathsf{Prod}_T^{\mathsf{rhs}}},$$

$$\mathsf{later}\ \frac{q : Q \qquad a : \alpha \qquad r : \eta_T(q, \mathsf{out}(\alpha)) \in \mathsf{Prod}_T^{\mathsf{rhs}}}{\mathsf{var}(q, a) \in \mathsf{Prod}_T^{\mathsf{rhs}}},$$

where α is an abstract type with an I-coalgebra structure $\mathsf{out} : \alpha \to I(\alpha)$ and the function η_T is defined as follows:

$$\eta_T : T.\mathsf{state} \times I(\alpha) \to O^*(T.\mathsf{state} \times \alpha)$$
$$\eta_T\ \ (q, (i, is))\ \equiv\ \mathsf{map}^{O^*(T.\mathsf{state} \times -)}\ (\lambda k.\ is(k))\ (T.\mathsf{rhs}(q, i)).$$

Then, by enforcing each rewrite rule to be productive, it is now possible to define a function that inductively searches the head function symbol eventually produced in a right-hand side. However, such an approach is not practical since the type of variables in the right-hand has to be abstracted. Indeed, it may be different for each rewrite rule. Finally, the type $\mathsf{Prop}_T^{\mathsf{rhs}}$ introduces a tight dependency between a proof of productivity and the induced tree morphism which can be difficult to deal with.

The second solution consists in enforcing—through typing—each rule to be productive. This approach is similar to the syntactic guard condition imposed on corecursive function definitions. However, it is different from type-based termination/productivity developed for example in [Abe07]. As an example, we consider a very simple form of top-down tree transducers which are characterized as morphism of signatures:

Definition 7 **Morphism of signatures**

Let $I, O : \mathrm{Sig}$ be two signatures. A morphism of signatures between I and O, denoted $I \Rightarrow O$, is given by a dependent pair of functions (l, r), where $l : I.\mathsf{Op} \to O.\mathsf{Op}$ is called a re-labeling map and $r : \forall (i : I.\mathsf{Op}).\ O.\mathsf{Ar}(l(i)) \to I.\mathsf{Ar}(i)$ is called a re-indexing map.

Morphisms of signatures can be thought of as a basic tdtt in the sense that they model rules of the form:

$$i(x_1, \ldots, x_k) \to o(y_1, \ldots, y_{k'})$$

where i (resp. o) is an input (resp. output) function symbol and $\{y_1, \ldots, y_{k'}\} \subseteq \{x_1, \ldots, x_k\}$. Reformulated in the formalism of tdtt (as introduced in the previous section), morphism of signatures are thus single-state tdtt with a set of rewrite rules given by a function:

$$\mathsf{rhs} : (i : I.\mathsf{Op}) \to O(I.\mathsf{Ar}\, i).$$

Indeed, through typing, we ensure that each rewrite rule is productive since an output function symbol has to be given for each input symbol. This contrasts with the previous definition of tdtt on finite trees where the codomain of rhs was given as the term algebra $\mathbb{T}_O(Q \times X)$ which, in particular, allows variables of type $Q \times X$ to appear unguarded by an output function symbol.

Then, it is straightforward to lift morphisms of signatures to tree morphisms:

Definition 8 Induced tree morphism

Let I, O be two signatures and $f : I \Rightarrow O$ be a morphism of signatures. The tree morphism induced by f, denoted $(\!| f |\!)$, is defined corecursively as follows:

$$(\!| f |\!) : \text{co}\mathbb{T}_I \to \text{co}\mathbb{T}_O$$
$$(\!| f |\!) \;\; (i \lhd is) \;\equiv\; l(i) \blacktriangleleft \lambda k.\; (\!| f |\!)(is(r(k))).$$

Here, the re-labeling function l is used to produce an output symbol from an input symbol thus ensuring that the tree morphism is productive. The re-indexing function r is used to select the argument of the corecursive call among the direct subtrees of the input tree.

In the previous chapter, we gave a coinductive-based characterization of regular trees. A tree $t : \text{co}\mathbb{T}_S$ over a signature S is said to be regular when the type of all its subtrees, namely $t{\leadsto}$, is weakly finitely indexed. In addition, we introduced the type of cyclic terms $\mathbb{C}_S$ and we showed that the type of closed cyclic terms is isomorphic to REG_S—the type of regular trees. Consequently, it is now possible to use the type of closed cyclic terms as an alternative characterization of regular trees. To this end, we redefine the predicate REGULAR as follows:

$$\text{REGULAR} : \forall \{S : \text{SIG}\}.\; \text{co}\mathbb{T}_S \to \text{TYPE}$$
$$\text{REGULAR}\;\; t \;\equiv\; \{\, c : \mathbb{C}_S \mid [\![\, c \,]\!] \sim t \,\}.$$

Furthermore, we overload this predicate to define regularity-preserving tree morphisms:

$$\text{REGULAR} : \forall \{I, O : \text{SIG}\}.\; (\text{co}\mathbb{T}_I \to \text{co}\mathbb{T}_O) \to \text{TYPE}$$
$$\text{REGULAR}\;\; f \;\equiv\; \{\, f_{\mathbb{C}} : \mathbb{C}_I \to \mathbb{C}_O \mid \forall (c : \mathbb{C}_I).\; [\![\, f_{\mathbb{C}}(c) \,]\!] \sim f[\![\, c \,]\!] \,\}.$$

To summarize, an infinite tree (resp. tree morphism) is said to be regular when there exists a semantically equivalent cyclic term (resp. cyclic term morphism).

In the following, we justify that, given a signature morphism f, its induced morphism $(\!| f |\!)$ preserves the regularity property. First, we detail the methodology used in the proof since the same principles will applied on every subsequent proofs establishing the regularity preservation.

Following the definition of REGULAR on tree morphisms, in order to show that $(\!| f |\!)$ preserves regularity, we have first to construct a cyclic tree morphism. To this end, we first generalize the induced morphism to operate on cyclic terms with free variables:

$$\langle\!| f |\!\rangle_{_} : \forall \{n : \mathbb{N}\}.\; \text{ENV}_O(n) \to \mathbb{C}_I(n) \to \mathbb{C}_O(n)$$
$$\langle\!| f |\!\rangle_n \;\; \Gamma \;\; (\mathsf{var}\; x) \;\equiv\; \Gamma(x)$$
$$\langle\!| f |\!\rangle_n \;\; \Gamma \;\; (\mathsf{rec}\; i \lhd is) \;\equiv\; \mathsf{rec}(l(i)) \lhd k \mapsto \langle\!| f |\!\rangle_{\mathsf{suc}\; n} \; (\mathsf{var\; zero} \,;\, \Gamma) \; (is(r(k))).$$

In order to improve readability, we assume that $is(r(k))$ is a subterm of $\mathsf{rec}\; i \lhd is$ in the definition of $\langle\!| f |\!\rangle$. In practice, we have to give an alternative definition in order to use structural recursion and then prove that the equations above hold propositionally.

Next, we show that $\langle\!| f |\!\rangle_n$ is semantically equivalent to $(\!| f |\!)$, *i.e.*, we have to prove the following:

$$\begin{aligned}&\forall (n : \mathbb{N}).\; \forall (t : \mathbb{C}_I(n)).\; \forall (\Gamma : \text{ENV}_O(n)).\; \forall (\rho_i : \text{CLOSURE}_I(n)).\; \forall (\rho_o : \text{CLOSURE}_O(n)).\\ &\quad (\forall (x : \text{FIN}\, n).\; [\![\, \Gamma(x) \,]\!]_{\rho_o} \sim (\!| f |\!)([\![\, t \,]\!]_{\rho_i})) \to [\![\, \langle\!| f |\!\rangle_n \; t \; \Gamma \,]\!]_{\rho_o} \sim (\!| f |\!)([\![\, t \,]\!]_{\rho_i}).\end{aligned} \qquad ((\!| f |\!) - \langle\!| f |\!\rangle)$$

Finally, we obtain the induced tree morphism on cyclic terms as follows:

$$\langle\!| f |\!\rangle : \mathbb{C}_I \to \mathbb{C}_O$$
$$\langle\!| f |\!\rangle \equiv \langle\!| f |\!\rangle_0(\varnothing)$$

where $\varnothing$ denotes the empty environment. The proof that it yields a morphism that is semantically equivalent to $(\!| f |\!)$ is given by the following theorem.

Theorem 1 Signature morphisms preserve regularity

Let I, O : Sig be two signatures and $f : I \Rightarrow O$ be a morphism of signature. Then, the induced morphism $(\!| f |\!)$ preserves regularity of trees.

Proof. Let I, O : Sig and $f : I \Rightarrow O$. To show that $(\!| f |\!)$ preserves regularity, we have to construct a semantically equivalent morphism on closed cyclic terms. We use the cyclic term morphism $\langle\!| f |\!\rangle : \mathbb{C}_I \to \mathbb{C}_O$ defined previously. It remains to show the equation $((\!| f |\!) - \langle\!| f |\!\rangle)$ holds. To this end, we use the coinduction proof principle with the bisimulation up-to a given depth. This yields the following generalized goal:

$$\begin{aligned}&\forall(n, p : \mathbb{N}).\ \forall(t : \mathbb{C}_I(p)).\ \forall(\Gamma : \textsc{Env}_O(p)).\ \forall(\rho_i : \textsc{Closure}_I(p)).\ \forall(\rho_o : \textsc{Closure}_O(p)).\\ &\quad(\forall(x : \textsc{Fin}\,p).\ [\![\Gamma(x)]\!]_{\rho_o} \sim_n (\!| f |\!)([\![\mathsf{var}\,x]\!]_{\rho_i})) \to [\![\langle\!| f |\!\rangle_p\ t\ \Gamma]\!]_{\rho_o} \sim_n (\!| f |\!)([\![t]\!]_{\rho_i}).\end{aligned} \qquad (\mathsf{fix}_n)$$

The proof is by recursion on n, thus we assume that the equation (fix_n) holds. Let $n, p : \mathbb{N}$, $t : \mathbb{C}_I(p)$, $\Gamma : \textsc{Env}_O(p)$, $\rho_i : \textsc{Closure}_I(p)$, $\rho_o : \textsc{Closure}_O(p)$ and $H : (\forall(x : \textsc{Fin}\,p).\ [\![\Gamma(x)]\!]_{\rho_o} \sim_n (\!| f |\!)([\![\mathsf{var}\,x]\!]_{\rho_i}))$. By case-analysis on n, we consider two cases:

- **Case 1.** Assume that $n = 0$.
  $[\![\langle\!| f |\!\rangle_p\ t\ \Gamma]\!]_{\rho_o} \sim_0 (\!| f |\!)([\![t]\!]_{\rho_i})$ holds by $\sim_0$-intro.
- **Case 2.** Assume that $n = \mathsf{suc}\,n'$ where $n' : \mathbb{N}$. Furthermore, by case-analysis on t:
  - **Case 2.1.** Assume that $t = \mathsf{var}\,x$ where $x : \textsc{Fin}\,p$.
    We have to show that $[\![\langle\!| f |\!\rangle_p\ (\mathsf{var}\,x)\ \Gamma]\!]_{\rho_o} \sim_{\mathsf{suc}\,n'} (\!| f |\!)([\![\mathsf{var}\,x]\!]_{\rho_i})$. This holds by H.
  - **Case 2.2.** Assume that $t = \mathsf{rec}\,i \lhd is$ where $i : I.\mathsf{Op}$ and $is : \textsc{Vec}\ (\mathbb{C}_I(\mathsf{suc}\,p))\ (I.\,\mathsf{Ar}\,i)$.
    We generalize the term $\mathsf{suc}\,n'$ in both the goal and the hypothesis H. Thus, we obtain:

    $$\begin{aligned}&\forall(m : \mathbb{N}).\ (\forall(x : \textsc{Fin}\,p).\\ &\quad[\![\Gamma(x)]\!]_{\rho_o} \sim_m (\!| f |\!)([\![\mathsf{var}\,x]\!]_{\rho_i})) \to [\![\langle\!| f |\!\rangle_p\ t\ \Gamma]\!]_{\rho_o} \sim_m (\!| f |\!)([\![\mathsf{rec}\,i \lhd is]\!]_{\rho_i}).\end{aligned}$$

    Let $m : \mathbb{N}$ and $H : \forall(x : \textsc{Fin}\,p).\ [\![\Gamma(x)]\!]_{\rho_o} \sim_m (\!| f |\!)([\![\mathsf{var}\,x]\!]_{\rho_i})$. By induction on m:
    - **Base step.** Assume that $m = 0$.
      $[\![\langle\!| f |\!\rangle_p\ t\ \Gamma]\!]_{\rho_o} \sim_0 (\!| f |\!)([\![t]\!]_{\rho_i})$ holds by $\sim_0$-intro.
    - **Inductive step.** Assume that $m = \mathsf{suc}\,m'$ where $m' : \mathbb{N}$.
      The induction hypothesis is given by

      $$\begin{aligned}&(\forall(x : \textsc{Fin}\,p).\ [\![\Gamma(x)]\!]_{\rho_o} \sim_{m'} (\!| f |\!)([\![\mathsf{var}\,x]\!]_{\rho_i}))\\ &\quad\to [\![\langle\!| f |\!\rangle_p\ (\mathsf{rec}\,i \lhd is)\ \Gamma]\!]_{\rho_o} \sim_{m'} (\!| f |\!)([\![\mathsf{rec}\,i \lhd is]\!]_{\rho_i}).\end{aligned} \qquad (\mathrm{IH})$$

      Clearly, we have $e : \mathsf{root}((\!| f |\!)([\![\mathsf{rec}\,i \lhd is]\!]_{\rho_i})) = \mathsf{root}((\!| f |\!)([\![\mathsf{rec}\,i \lhd is]\!]_{\rho_i}))$ by reflexivity. Consequently, we use the constructor $\sim_{\mathsf{suc}}$-intro with e. It remains to show that:

      $$\forall k.\ [\![\langle\!| f |\!\rangle_p\ (\mathsf{rec}\,i \lhd is)\ \Gamma]\!]_{\rho_o}\ k \sim_{m'} (\!| f |\!)([\![\mathsf{rec}\,i \lhd is]\!]_{\rho_i})\ k$$

      which is equivalent to the following goal:

      $$\forall k.\ [\![\langle\!| f |\!\rangle_{\mathsf{suc}\,p}\ (is(r(k)))\ \Gamma']\!]_{\rho_o'} \sim_{m'} (\!| f |\!)([\![is(r(k))]\!]_{\rho_i'})$$

where $\Gamma' := (\mathsf{var}\,\mathsf{zero}; \Gamma)$, $\rho_o' := (\langle\!|\, f \,|\!\rangle_p\ (\mathsf{rec}\, i \lhd is)\ \Gamma) :: \rho_o$ and $\rho_i' := (\mathsf{rec}\, i \lhd is) :: \rho_i$. Let $k : \textsc{Fin}(O.\,\mathsf{Ar}(l(i)))$. We use the recursive function (fix_n). Here, m' is a subterm of $\mathsf{suc}\, n'$ since m was generalized from $\mathsf{suc}\, n'$. It remains to prove that the invariant is preserved:

$$\forall (x : \textsc{Fin}(\mathsf{suc}\, p)).\ [\![\, \Gamma'(x) \,]\!]_{\rho_o'} \sim_{m'} (\!|\, f \,|\!)([\![\ \mathsf{var}\, x \,]\!]_{\rho_i'})$$

Let $x : \textsc{Fin}(\mathsf{suc}\, p)$. By case-analysis on x, we consider two cases:

- **Case 2.2.1.** Assume that $x = \mathsf{zero}$.
  We have $[\![\, \Gamma'(x) \,]\!]_{\rho_o'} \equiv [\![\, (\mathsf{var}\,\mathsf{zero}; \Gamma)(\mathsf{zero}) \,]\!]_{\rho_o'} \sim_{m'} [\![\, \langle\!|\, f \,|\!\rangle_p\ (\mathsf{rec}\, i \lhd is)\ \Gamma \,]\!]_{\rho_o}$. Likewise, for the right-hand side, we have $(\!|\, f \,|\!)([\![\ \mathsf{var}\,\mathsf{zero} \,]\!]_{\rho_i'}) \sim_{m'} (\!|\, f \,|\!)([\![\ \mathsf{rec}\, i \lhd is \,]\!]_{\rho_i})$. By (IH), in order to prove that $[\![\, \langle\!|\, f \,|\!\rangle_p\ (\mathsf{rec}\, i \lhd is)\ \Gamma \,]\!]_{\rho_o} \sim_{m'} (\!|\, f \,|\!)([\![\ \mathsf{rec}\, i \lhd is \,]\!]_{\rho_i})$, it suffices to show that $\forall (x : \textsc{Fin}\, p).\ [\![\, \Gamma(x) \,]\!]_{\rho_o} \sim_{m'} (\!|\, f \,|\!)([\![\ \mathsf{var}\, x \,]\!]_{\rho_i})$. This is given by applying H to Proposition 2.29 [Weakening of $\sim_n$].
- **Case 2.2.2.** Assume that $x = \mathsf{suc}\, x'$ where $x' : \textsc{Fin}\, p$.
  We have $[\![\, \Gamma'(\mathsf{suc}\, x') \,]\!]_{\rho_0'} \equiv [\![\, (\mathsf{var}\,\mathsf{zero}; \Gamma)(\mathsf{suc}\, x') \,]\!] \sim_{m'} [\![\, \Gamma(x') \,]\!]_{\rho_0}$. Likewise, for the right-hand side, we have $(\!|\, f \,|\!)([\![\ \mathsf{var}(\mathsf{suc}\, x') \,]\!]_{\rho_i'}) \sim_{m'} (\!|\, f \,|\!)([\![\ \mathsf{var}\, x' \,]\!]_{\rho_i})$. Finally, this is proved by applying H to Proposition 2.29 [Weakening of $\sim_n$]. □

3.3. GUARDED TOP-DOWN TREE TRANSDUCERS

In this section, we extend the definition of top-down tree transducers to operate on infinite trees. To this end, we first characterize the subset of tdtt with right-hand sides in guarded-form. This restriction on the right-hand sides yields a straightforward extension of the type TDTT defined previously. Then, we show that such a subset induces a morphism on infinite trees. Finally, we prove that the induced tree morphism preserve regularity.

Definition 9 **Guarded Top-Down Tree Transducer**

Let $I, O : \textsc{Sig}$ be two signatures. The type of guarded top-down tree transducers, denoted ${}^{G}\mathrm{TDTT}\ I\ O$, is defined as a dependent record:

record ${}^{G}\mathrm{TDTT}\ (I\ O : \textsc{Sig}) : \textsc{Type}$
  constructor $\langle _, _, _, \rangle$
    $\mathsf{state} : \textsc{Type}$
    $\mathsf{init} : \mathsf{state}$
    $\mathsf{rhs} : \forall (q : \mathsf{state}).\ \forall (i : I.\mathsf{Op}).\ O^{+}(\mathsf{state} \times \textsc{Fin}(I.\,\mathsf{Ar}\, i)).$

The main difference with the type TDTT is that the type of rhs ends with $O^{+}(\mathsf{state} \times S.\,\mathsf{Ar}\, i)$ instead of $O^{*}(\mathsf{state} \times S.\,\mathsf{Ar}\, i)$. Consequently, each rewrite rule is required to always produce an output function symbol: *productivity is ensured by construction.* The definition of the induced tree morphism is not as straightforward as the one defined for morphisms of signatures. Here, the problem is that the right-hand sides of rewrite rules are—finite—trees of arbitrary depth. We define the induced tree morphism by coiteration. To this end, given a guarded tree transducer T, we give to the type $O^{*}(\mathsf{state} \times \mathrm{co}\mathbb{T}_I)$, a structure of an O-coalgebra:

```
rhs-coalg : O*(T.state × coT_I) → O+(T.state × coT_I)
rhs-coalg  (var(q, t))  ≡  η q t
rhs-coalg  (o ◁ os)     ≡  (o, os).
```

The function η which substitutes variables in right-hand sides of rewrite rules with the corresponding subtree is defined as follows:

$$\eta : T.\mathsf{state} \to \mathrm{co}\mathbb{T}_I \to O^+(T.state \times \mathrm{co}\mathbb{T}_I)$$
$$\eta\ \ q\ \ t\ \ \equiv\ \ \mathsf{map}^{O^+(T.\mathsf{state}\times -)}\ (\mathsf{br}\,t)\ (T.\mathsf{rhs}\ q\ (\mathsf{root}\,t)).$$

Intuitively, we apply the rewrite rule from the state q and the input function symbol given by the root of t. Finally, we use the functoriality of the type $O^+(T.\mathsf{state} \times -)$ to replace the variable indexing a subtree of t by the actual subtree of t. This step is illustrated in Figure 3.1. Now, it is straightforward to define the induced tree morphism of a guarded tdtt:

Definition 10 **Induced tree morphism**

Let I, O be two signatures and $T : {}^{\mathrm{G}}\mathrm{TDTT}\ I\ O$ be a guarded tree transducer. The induced tree morphism starting from a designated state q, denoted $(\!|T|\!)_q$, is defined as follows:

$$(\!|T|\!)__ : T.\mathsf{state} \to \mathrm{co}\mathbb{T}_I \to \mathrm{co}\mathbb{T}_O$$
$$(\!|T|\!)_q\ \ t\ \ \equiv\ \ \mathsf{coiter}\ \ \mathsf{rhs\text{-}coalg}\ (\mathsf{var}(q,t)).$$

Moreover, we define the semantics of the guarded tdtt T as:

$$(\!|T|\!) : \mathrm{co}\mathbb{T}_I \to \mathrm{co}\mathbb{T}_O$$
$$(\!|T|\!)\ \ \equiv\ \ (\!|T|\!)_{T.\mathsf{init}}.$$

The following recursive equation justifies that a derivation step has been correctly applied:

Proposition 1 **Rewrite rule application**

Let $I, O : \mathrm{SIG}$ be two signatures and $T : {}^{\mathrm{G}}\mathrm{TDTT}\ I\ O$ be a guarded tree transducers. Then, for each state $q : T.\mathsf{state}$ and infinite tree $t : \mathrm{co}\mathbb{T}_I$, we have:

$$(\!|T|\!)_q(t) \sim \eta\ q\ t\ \ggg^+\ \mathsf{uncurry}\ \ (\!|T|\!).$$

Proof. The proof is straightforward after proving, by induction on t, the following generalized goal:

$$\forall (t : O^*(T.\mathsf{state} \times \mathrm{co}\mathbb{T}_I)).\ \mathsf{coiter}\ \mathsf{rhs\text{-}coalg}\ t \sim t \ggg \mathsf{uncurry}\ (\!|T|\!). \qquad \square$$

Finally, we show that the induced morphisms of guarded tree transducers preserve regularity.

Theorem 2 **$(\!|-|\!)$ preserves regularity**

Let $I, O : \mathrm{SIG}$ be two signatures and let $T : {}^{\mathrm{G}}\mathrm{TDTT}\ I\ O$ be a guarded tree transducer. If the state space $T.\mathsf{state}$ is finite, then the induced tree morphism $(\!|T|\!)$ preserves regularity.

Proof. Instead of giving a direct proof of this result, we exploit the fact that it is an instance of a more general result. Indeed, this follows from Proposition 3.3 and of Theorem 3.4. $\square$

3.4. GUARDED TREE TRANSDUCERS WITH ε-RULES

Guarded top-down tree transducers, as defined previously, impose that each rule consumes the root of the input tree. When the semantics of tree transducers is defined on *finite* tree, this restriction is mandatory to ensure that the successive applications of the rewrite rules eventually terminate. However, when we consider the application of such rewrite rules on infinite trees , it is possible to drop this constraint altogether and allow rewrite rules which do not alter the input tree [Rou70, Tha70]. For instance, a rewrite rule such as

$$q \to f(q)$$

where f is an arbitrary unary function symbol is accepted by the productivity checker of COQ. This leads to the following extension of the definition of guarded top-down tree transducers:

Definition 11 **Guarded tdtt with ε-rules**

Let I, O : SIG be two signatures. The type of guarded top-down tree transducers with ε-rules, denoted ${}^{\varepsilon}\mathrm{TDTT}\ I\ O$, is defined as follows:

$$\begin{array}{l}
\textbf{record}\ {}^{\varepsilon}\mathrm{TDTT}\ (I\ O : \mathrm{SIG}) : \mathrm{TYPE} \\
\quad \textbf{constructor}\ \langle _, _, _, \rangle \\
\quad\quad \left[\begin{array}{l}
\mathsf{state} : \mathrm{TYPE} \\
\mathsf{init} : \mathsf{state} \\
\mathsf{rhs} : \mathsf{state} \to \forall (i : I.\mathsf{Op}).\ O^{+}(\mathsf{state} \times \mathrm{MAYBE}(\mathrm{FIN}(I.\,\mathsf{Ar}\, i))).
\end{array}\right.
\end{array}$$

The main difference with the type ${}^{\mathrm{G}}\mathrm{TDTT}$ is in the type of variables of the right-hand sides. Before, variables ought to be picked from an indexing set of subtrees of the input tree whereas now, we encapsulate variables within MAYBE. Intuitively, when a variable is nothing, it means that the computation should be carried on without consuming the input tree. On the other hand, when the variable is $\mathsf{just}(k)$, it expresses the fact that the computation resumes on the subtree indexed by k, thus consuming the input context. In order to define the induced tree morphism of guarded tdtt with ε-rules, we follow the same procedure than for guarded tdtt. Let T denote a guarded tdtt with ε-rules. First, we give to the type of right-hand sides an O-coalgebra structure:

$$\begin{array}{l}
\mathsf{rhs\text{-}coalg} : O^{*}(T.\mathsf{state} \times \mathrm{co}\mathbb{T}_I) \to O^{+}(T.\mathsf{state} \times \mathrm{co}\mathbb{T}_I) \\
\mathsf{rhs\text{-}coalg}\ \ (\mathsf{var}(q,t)) \ \equiv\ \eta\ q\ t \\
\mathsf{rhs\text{-}coalg}\ \ (o \lhd os) \quad \equiv\ (o, os).
\end{array}$$

The function η which substitutes variables in right-hand sides of rewrite rules with the corresponding subtree is defined as follows:

$$\begin{array}{l}
\eta : T.\mathsf{state} \times \mathrm{co}\mathbb{T}_I \to O^{+}(T.\mathsf{state}\ \times \mathrm{co}\mathbb{T}_I) \\
\eta\ \ q\ \ t\ \equiv\ \mathsf{map}^{O^{+}(T.\mathsf{state} \times -)}\ h\ (\mathsf{rhs}\ q\ (\mathsf{root}\, t)) \\
\qquad \textbf{where}\ \ h : \mathrm{MAYBE}(\mathsf{root}\, t) \to \mathrm{co}\mathbb{T}_I \\
\qquad\qquad\quad h\ \ \mathsf{nothing}\ \equiv\ t \\
\qquad\qquad\quad h\ \ (\mathsf{just}\, k)\ \equiv\ t(k).
\end{array}$$

Now, we define the induced tree morphism of a guarded tdtt with ε-rules:

Definition 12 **Induced tree morphism**

Let I, O be two signatures and T : ${}^{\varepsilon}\mathrm{TDTT}\ I\ O$ be a guarded tree transducer with ε-rules. The induced tree morphism starting from a designated state is defined as follows:

$$\begin{array}{l}
(\!|\, T \,|\!)_{_} : T.\mathsf{state} \to \mathrm{co}\mathbb{T}_I \to \mathrm{co}\mathbb{T}_O \\
(\!|\, T \,|\!)_{q}\ \ t\ \equiv\ \mathsf{coiter}\ \mathsf{rhs\text{-}coalg}\ (\mathsf{var}(q,t)).
\end{array}$$

Moreover, we define the semantics of the guarded tdtt T as:

$$\begin{array}{l}
(\!|\, T \,|\!) : \mathrm{co}\mathbb{T}_I \to \mathrm{co}\mathbb{T}_O \\
(\!|\, T \,|\!)\ \equiv\ (\!|\, T \,|\!)_{T.\mathsf{init}}.
\end{array}$$

The following recursive equation justifies that a derivation step has been correctly applied:

Proposition 2 **Rewrite rule application**

Let I, O : Sig be two signatures and T : ${}^{\varepsilon}\mathrm{TDTT}\ I\ O$ be a guarded tree transducers. Then, for each state q : T.state and infinite tree t : $\mathrm{co}\mathbb{T}_I$, we have:

$$(\!| T |\!)_q(t) \sim \eta\ q\ t \ggg^{+} \mathsf{uncurry}\ (\!| T |\!).$$

Proof. The proof is straightforward after proving by induction on t the following generalized goal:

$$\forall (t : O^{*}(T.\mathsf{state} \times \mathrm{co}\mathbb{T}_I)).\ \mathsf{coiter}\ \mathsf{rhs\text{-}coalg}\ t \sim t \ggg \mathsf{uncurry}\ (\!| T |\!). \qquad \square$$

The formalism of guarded top-down tree transducer with ε-rules subsumes the formalism of guarded top-down tree transducers:

$$
\begin{aligned}
&{}^{\mathrm{G}\to\varepsilon}\mathsf{TDTT} : \forall \{I, O : \mathrm{Sig}\}.\ {}^{\mathrm{G}}\mathrm{TDTT}\ I\ O \to {}^{\varepsilon}\mathrm{TDTT}\ I\ O \\
&{}^{\mathrm{G}\to\varepsilon}\mathsf{TDTT}\ \ T \ \equiv\ \langle T.\mathsf{state}, T.\mathsf{init}, {}^{\varepsilon}\mathsf{rhs} \rangle \\
&\quad \textbf{where}\ \ {}^{\varepsilon}\mathsf{rhs} : T.\mathsf{state} \to (i : I) \to O^{+}(T.\mathsf{state} \times \mathrm{Maybe}(\mathrm{Fin}(S.\mathsf{Ar}\ i))) \\
&\quad\qquad\quad {}^{\varepsilon}\mathsf{rhs}\ \ q\ \ i\ \equiv\ \mathsf{map}^{O^{+}(T.\mathsf{state} \times -)}\ \mathsf{just}\ (T.\mathsf{rhs}\ q\ i).
\end{aligned}
$$

We still have to check that the translation is sound:

Proposition 3 **${}^{\mathrm{G}}\mathrm{TDTT}$ to ${}^{\varepsilon}\mathrm{TDTT}$ soundness**

Let T : ${}^{\mathrm{G}}\mathrm{TDTT}$ be a guarded top-down tree transducer. Then, there exists a guarded top-down tree transducer with ε-rules ${}^{\varepsilon}T$: ${}^{\varepsilon}\mathrm{TDTT}\ I\ O$ such that

$$\forall (t : \mathrm{co}\mathbb{T}_I).\ (\!| T |\!)(t) \sim (\!| {}^{\varepsilon}T |\!)(t).$$

Proof. The transducer ${}^{\varepsilon}T$ is given by ${}^{\mathrm{G}\to\varepsilon}\mathsf{TDTT}$. Then, the proof of the equation above is straightforward by coinduction. $\square$

In the following, we show that restricted to finite guarded tree transducer with ε-rules, *i.e.*, tree transducers with a finite state space, the induced tree morphism preserves regularity.

Theorem 3 **$(\!| - |\!)$ preserves regularity**

Let I, O : Sig be two signatures and let T : ${}^{\varepsilon}\mathrm{TDTT}\ I\ O$ be a guarded tree transducer with ε-rules. If the state space T.state is finite, then the induced tree morphism $(\!| T |\!)$ preserves regularity.

Proof. Instead of giving a direct proof of this result, we exploit the fact that it is an instance of a more general result. Indeed, this follows from Proposition 3.4 and of Theorem 3.4. $\square$

3.5. TREE TRANSDUCERS WITH FINITE LOOK-AHEAD

Previously, we introduced guarded top-down tree transducers as a mean to ensure productivity by construction. Then, guarded top-down tree transducers were extended further by allowing of rewrite rules which that do not alter the input tree. Here, we define yet another extension of the formalism of guarded top-down tree transducer with a notion of *finite* look-ahead. This sorts of extension were extensively studied in the literature about tree transducers. Some examples include [Eng77, SV95]. Generally, these extensions are defined over finite trees. As a result, the look-ahead function may be computed, for instance, through a bottom-up tree automaton [CDG+07]. However, since we work with infinite trees, we have to somehow restrict the depth of the look-ahead since it can be arbitrarily large.

Definition 13 **Guarded tdtt with finite look-ahead**

Let I, O : SIG be two signatures. The type of top-down tree transducers with finite look-ahead, denoted $^{\mathsf{L}}\mathrm{TDTT}\ I\ O$, is defined as a dependent record:

$$
\begin{array}{l}
\textbf{record}\ {}^{\mathsf{L}}\mathrm{TDTT}\ (I\ O : \mathrm{SIG}) : \mathrm{TYPE} \\
\quad \textbf{constructor}\ \langle_,_,_\rangle \\
\quad\quad \left[\begin{array}{l}
\mathsf{state} : \mathrm{TYPE} \\
\mathsf{init} : \mathsf{state} \\
\mathsf{rhs} : \forall(q : \mathsf{state}).\ \sum_{d:\mathbb{N}} (c : I^d(\mathrm{UNIT})) \to O^+(\mathsf{state} \times \{p \mid p \in \mathscr{P}_d(c)\}).
\end{array}\right.
\end{array}
$$

The formalism of tdtt with finite look-ahead is extended in two ways. On the one hand, each state q is annotated with a given depth d, expressing how deep the context needs to be in order to produce a value. Contexts are represented by iterating d times the functor induced by the input signature I. Note that the empty context is allowed. On the other hand, the type of variables indexing the subtrees of the input trees is generalized. Since the context can be arbitrarily deep, we need a way to index subtrees in the context in order to specify how much of the context is actually consumed. This indexing is achieved through the predicate $\mathscr{P}_d(c)$ where $c : I^d(\top)$ denotes a context of depth d. The predicate $\mathscr{P}_n(c)$ represents the set of paths within a context $c : I^n(X)$ and is defined inductively as follows:

$$
\mathscr{P}_\varepsilon\ \frac{c : I^d(X)}{\varepsilon \in \mathscr{P}_n(c)}, \qquad \mathscr{P}_{::}\ \frac{c : I^{\mathsf{suc}\,d}(X) \qquad e : n = I.\mathsf{Ar}(\mathsf{proj}_1\ c) \qquad p \in \mathscr{P}_n(\mathsf{proj}_2\ c\ (e_*(k)))}{(m,k) :: p \in \mathscr{P}_{\mathsf{suc}\,n}}.
$$

Note that, the type $\mathscr{P}_n(-)$ is similar to the type of paths in an unranked coalgebra as defined in Section 2.1.3. Furthermore, it shares the same properties. In particular, it is straightforward to prove by path induction that:

- $\forall p.\ p \in \mathscr{P}_n(c) \leftrightarrow p \in \mathscr{P}_n(\mathsf{map}^{I^n}\ f\ c)$, ($\mathscr{P}_n$-map)
- $\forall c.\ \forall p.\ \forall(P, Q : p \in \mathscr{P}_n(c)).\ P = Q$, ($\mathscr{P}_n$-mere-prop)
- $\forall(t : \mathrm{CO}\mathbb{T}_I).\ p \in \mathscr{P}_n([\blacktriangleright]^n\ t) \to p \in \mathscr{P}_{\mathrm{CO}\mathbb{T}}(t)$. ($\mathscr{P}_n$-unfold)

Given a top-down tree transducer with finite look-ahead T, we define the function depth representing the depth of the input context starting from a given state:

$$
\begin{array}{l}
\mathsf{depth} : \forall\{T : {}^{\mathsf{L}}\mathrm{TDTT}\ I\ O\}.\ T.\mathsf{state} \to \mathbb{N} \\
\mathsf{depth}\ \ \{T\} \ \equiv\ \mathsf{proj}_1 \circ T.\mathsf{rhs}.
\end{array}
$$

In addition, we define the following variant of rhs:

$$
\begin{array}{l}
\mathsf{rhs}' : \forall\{T : {}^{\mathsf{L}}\mathrm{TDTT}\ I\ O\}.\ \forall\{\alpha\}.\ (q : T.\mathsf{state}) \to (c : I^{\mathsf{depth}\,q}(\alpha)) \to O^+(T.\mathsf{state} \times cc \in \mathbb{P}_\alpha) \\
\mathsf{rhs}'\ \ \{T\}\ \ q\ \equiv\ \mathsf{proj}_2(T.\mathsf{rhs}\,q).
\end{array}
$$

Now, we show how to define the induced tree morphism of a guarded top-down tree transducer with finite look-ahead. First, let us recall that the type of infinite tree $\mathrm{CO}\mathbb{T}_S$ can be regarded as both a S-algebra and a S-coalgebra. Thus, we have the following two maps:

$$
\begin{array}{ll}
[\blacktriangleleft] : \forall\{S : \mathrm{SIG}\}.\ S(\mathrm{CO}\mathbb{T}_S) \to \mathrm{CO}\mathbb{T}_S & [\blacktriangleright] : \forall\{S : \mathrm{SIG}\}.\ \mathrm{CO}\mathbb{T}_S \to S(\mathrm{CO}\mathbb{T}_S) \\
{[\blacktriangleleft]}\ \ (o, os)\ \equiv\ o \blacktriangleleft os. & [\blacktriangleright]\ \ t\ \equiv\ (\mathsf{root}\,t, \mathsf{tabulate}(\mathsf{br}\,t)).
\end{array}
$$

Both $[\blacktriangleleft]$ and $[\blacktriangleright]$ can be iterated to yield a S^n-algebra and a S^n-coalgebra respectively:

$$
\begin{array}{ll}
[\blacktriangleleft]^- : \forall\{S : \mathrm{SIG}\}.\ \forall n.\ S^n(\mathrm{CO}\mathbb{T}_S) \to \mathrm{CO}\mathbb{T}_S & [\blacktriangleright]^- : \forall\{S : \mathrm{SIG}\}.\ \forall n.\ S^n(\mathrm{CO}\mathbb{T}_S) \to \mathrm{CO}\mathbb{T}_S \\
{[\blacktriangleleft]}^{\mathsf{zero}}\ \equiv\ \mathsf{id} & [\blacktriangleright]^{\mathsf{zero}}\ \equiv\ \mathsf{id} \\
{[\blacktriangleleft]}^{\mathsf{suc}\,n}\ \equiv\ [\blacktriangleleft] \circ \mathsf{map}^S\ [\blacktriangleleft]^n\,. & [\blacktriangleright]^{\mathsf{suc}\,n}\ \equiv\ \mathsf{map}^S\ [\blacktriangleright]^n \circ [\blacktriangleright]\,.
\end{array}
$$

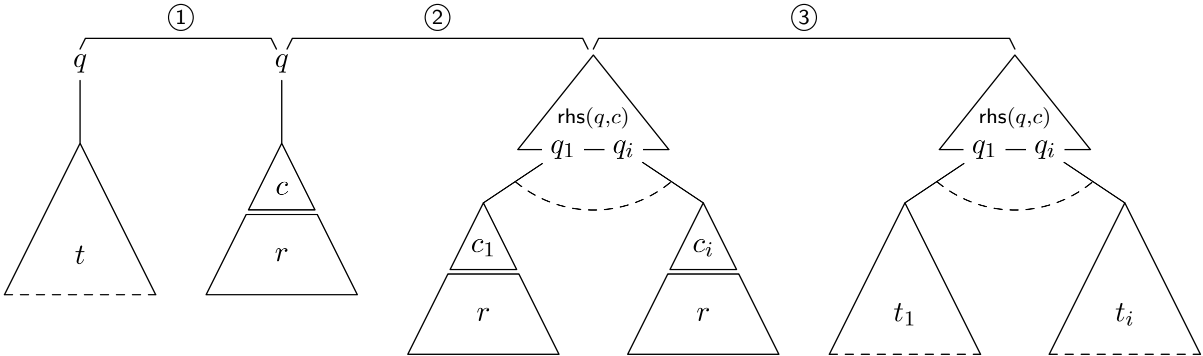


Figure 3.2. Derivation step for a tree transducer with finite look-ahead.

Moreover, it is straightforward to show by induction on $n : \mathbb{N}$ that we have:

$$\forall (S : \textsc{Sig}).\ \forall (n : \mathbb{N}).\ \forall (t : \text{co}\mathbb{T}_S).\ [\blacktriangleleft]^n ([\blacktriangleright]^n\, t) \sim t.$$

Now that we have defined the iterated version of S-(co)algebras on infinite trees, it is straightforward to generate a look-ahead up-to a given depth:

$$\begin{aligned}&\mathsf{look\text{-}ahead_} : \forall \{S : \textsc{Sig}\}.\ \forall (n : \mathbb{N}).\ \text{co}\mathbb{T}_S \to S^n(\textsc{Unit})\\ &\mathsf{look\text{-}ahead}_n \equiv \mathsf{map}^{S^n}\ (\mathsf{const\ tt}) \circ [\blacktriangleright]^n .\end{aligned}$$

The look-ahead function is defined by iterating the destructor of infinite trees. Then, we use the functoriality of the free monad to replace the subtree variables with element of the type Unit. This context is used in conjunction with the map rhs of the tree transducer. In addition, we define the function that folds back the remaining context:

$$\begin{aligned}&{\downarrow}[_] : \forall \{n : \mathbb{N}\}.\ \forall \{t : \text{co}\mathbb{T}_I\}.\ \forall \{p : \textsc{Path}\}.\ p \in \mathscr{P}_n(\mathsf{look\text{-}ahead}_n\ t) \to \text{co}\mathbb{T}_I\\ &{\downarrow}[P] \equiv \mathsf{from\text{-}just}\ \ (t|_{P_t})\ \ H\\ &\qquad \textbf{where}\ \ P_t : p \in \mathscr{P}_{\text{co}\mathbb{T}}(t)\\ &\qquad\qquad\quad\ H : \mathsf{is\text{-}just}(t|_{P_t})\end{aligned}$$

where P_t is a proof obtained as follows:

$$p \in \mathscr{P}_n(\mathsf{look\text{-}ahead}_n\ t) \xrightarrow{\mathscr{P}_n\text{-}\mathsf{map}} p \in \mathscr{P}_n([\blacktriangleright]^n\, t) \xrightarrow{\mathscr{P}_n\text{-}\mathsf{unfold}} p \in \mathscr{P}_{\text{co}\mathbb{T}}(t),$$

and H is a proof obtained from Proposition 2.5 applied to P_t. Clearly, we have also that:

$$\forall (n : \mathbb{N}).\ \forall (t : \text{co}\mathbb{T}_I).\ \forall (p : \textsc{Path}).\ \forall (P : p \in \mathscr{P}_n(\mathsf{look\text{-}ahead}_n\ t)).\ t \rightsquigarrow {\downarrow}[P]. \qquad (\mathsf{look\text{-}ahead}{\rightsquigarrow})$$

In order to define the induced tree morphism of guarded tdtt with finite look-ahead, we follow the same procedure than for simple guarded top-down tree transducers.

$$\begin{aligned}&\mathsf{rhs\text{-}coalg} : O^*(T.\mathsf{state} \times \text{co}\mathbb{T}_I) \to O^+(T.\mathsf{state} \times \text{co}\mathbb{T}_I)\\ &\mathsf{rhs\text{-}coalg}\ \ (\mathsf{var}(q,t)) \equiv \eta\ q\ t\\ &\mathsf{rhs\text{-}coalg}\ \ (o \lhd os) \quad \equiv (o, os).\end{aligned}$$

The function η which substitutes variables in right-hand sides is given by:

$$\begin{aligned}&\eta : T.\mathsf{state} \times \text{co}\mathbb{T}_I \to O^+(T.\mathsf{state} \times \text{co}\mathbb{T}_I)\\ &\eta\ \ q\ \ t\ \equiv\ \mathsf{map}^{O^+(T.\mathsf{state}\times -)}\ ({\downarrow}[_] \circ \mathsf{proj}_2)\ \big(\mathsf{rhs}'\ q\ \ (\mathsf{look\text{-}ahead}_{(\mathsf{depth}\ q)}\ t)\big)\ .\end{aligned}$$

Here, the function η can be decomposed into three main steps as illustrated in Figure 3.2:

① a look-ahead context of depth given by depth q is generated from the input tree,

② the applicable rewrite rule is used,
③ the look-ahead context that is not consumed is folded back into the input tree.

Now, we define the induced tree morphism of a guarded tdtt with finite look-ahead:

Definition 14 **Induced tree morphism**

Let I, O be two signatures and $T : {}^{\mathsf{L}}\mathrm{TDTT}\ I\ O$ be a guarded tree transducer with finite look-ahead. The induced tree morphism starting from a designated state is defined as follows:

$$\llparenthesis T \rrparenthesis_{_} : T.\mathsf{state} \to \mathrm{co}\mathbb{T}_I \to \mathrm{co}\mathbb{T}_O$$
$$\llparenthesis T \rrparenthesis_q\ t \equiv \mathsf{coiter}\ \mathsf{rhs\text{-}coalg}\ (\mathsf{var}(q, t)).$$

Moreover, we define the semantics of the guarded tdtt T with finite look-ahead as:

$$\llparenthesis T \rrparenthesis : \mathrm{co}\mathbb{T}_I \to \mathrm{co}\mathbb{T}_O$$
$$\llparenthesis T \rrparenthesis \equiv \llparenthesis T \rrparenthesis_{T.\mathsf{init}}.$$

The formalism of guarded top-down tree transducer with finite look-ahead subsumes the formalism of top-down tree transducers with ε-rules:

$$
\begin{aligned}
&{}^{\varepsilon\to\mathsf{L}}\mathsf{TDTT} : \forall\{I, O : \mathrm{SIG}\}.\ {}^{\varepsilon}\mathrm{TDTT}\ I\ O \to {}^{\mathsf{L}}\mathrm{TDTT}\ I\ O\\
&{}^{\varepsilon\to\mathsf{L}}\mathsf{TDTT}\ \ T \equiv \langle T.\mathsf{state}, T.\mathsf{init}, {}^{\mathsf{L}}\mathsf{rhs}\rangle\\
&\quad \textbf{where}\ {}^{\mathsf{L}}\mathsf{rhs} : \forall(q : T.\mathsf{state}).\ \textstyle\sum_{(d:\mathbb{N})}(c : I^d(\mathrm{UNIT})) \to O^+(T.\mathsf{state} \times \{p \mid p \in \mathscr{P}_d(c)\})\\
&\qquad\quad {}^{\mathsf{L}}\mathsf{rhs}\ \ q \equiv \big(1, \lambda(c : I(\mathrm{UNIT})).\ \mathsf{map}^{O^+(T.\mathsf{state}\times -)}\ (h(c))\ (T.\mathsf{rhs}\ q\ (\mathsf{proj}_1\ c))\big)\\
&\qquad\quad h : \forall(c : I(\mathrm{UNIT})).\ \mathrm{MAYBE}(\mathrm{FIN}(I.\,\mathsf{Ar}\,i)) \to \{p \mid p \in \mathscr{P}_d(c)\}\\
&\qquad\quad h\ \ c\ \ \mathsf{nothing} \equiv (\varepsilon, \mathscr{P}_\varepsilon)\\
&\qquad\quad h\ \ c\ \ (\mathsf{just}\,k) \equiv (k :: \varepsilon, \mathscr{P}_{::}\ k\ \mathscr{P}_\varepsilon).
\end{aligned}
$$

We still have to check that the translation is sound:

Proposition 4 ${}^{\varepsilon}\mathrm{TDTT}$ **to** ${}^{\mathsf{L}}\mathrm{TDTT}$ **soundness**

Let $T : {}^{\varepsilon}\mathrm{TDTT}$ be a guarded top-down tree transducer with ε-rules. Then, there exists a guarded top-down tree transducer with finite look-ahead ${}^{\mathsf{L}}T : {}^{\mathsf{L}}\mathrm{TDTT}\ I\ O$ such that

$$\forall(t : \mathrm{co}\mathbb{T}_I).\ \llparenthesis T \rrparenthesis(t) \sim \llparenthesis {}^{\mathsf{L}}T \rrparenthesis(t).$$

Proof. The transducer ${}^{\mathsf{L}}T$ is given by ${}^{\varepsilon\to\mathsf{L}}\mathsf{TDTT}$. Then, the proof of the equation above is straightforward by coinduction. □

Example. Top-down tree transducers with finite look-ahead can be thought of as an abstract syntax for corecursive function with deep pattern-matching, *i.e.*, by allowing terms to be pattern-matched several times consecutively. As an illustration, consider the following example which consists in swapping consecutive elements of a stream:

$$
\begin{aligned}
&\mathsf{swap} : \mathrm{STREAM}\ \ A \to \mathrm{STREAM}\ \ A\\
&\mathsf{swap}\ \ (x_1 :: x_2 :: xs) \equiv x_2 :: x_1 :: \mathsf{swap}\ xs.
\end{aligned}
$$

The corecursive function swap can be defined by the following tdtt with finite look-ahead:

$$
\begin{aligned}
&T_{\mathsf{swap}} : {}^{\mathsf{L}}\mathrm{TDTT}\ (\mathrm{STR}_A)\ (\mathrm{STR}_A)\\
&T_{\mathsf{swap}} \equiv \langle \mathrm{UNIT}, \mathsf{tt}, \lambda(_ : \mathrm{UNIT}).\ (2, f)\rangle\\
&\quad \textbf{where}\ f : (c : \mathrm{STR}_A^2(\top)) \to \mathrm{STR}_A^+(\mathrm{UNIT} \times \{p \mid p \in \mathscr{P}_2(c)\})\\
&\qquad\quad f\ \ (x_1 \lhd [x_2 \lhd [_]]) \equiv x_2 \lhd [x_1 \lhd [\mathsf{tt}, \mathscr{P}_{::}\ \mathsf{zero}\ (\mathscr{P}_{::}\ \mathsf{zero}\ \mathscr{P}_\varepsilon)]].
\end{aligned}
$$

Here, we show that restricted to finite guarded tree transducer with finite look-ahead, *i.e.*, tree transducers with a finite state space, the induced tree morphism preserves regularity.

Theorem 4 Induced tree morphism preserves regularity

Let I, O : Sig be two signatures and let T : $^{\mathsf{L}}$TDTT I O be a tree transducer with finite look-ahead. If the state space T.state is finite, then the induced tree morphism $⦇ T ⦈$ preserves regularity.

Proof. The proof that the induced morphism preserves regularity of trees is similar to Thm. 3.1. Let T : $^{\mathsf{L}}$TDTT I O be a top-down tree transducer with finite look-ahead. We assume that the type of state of T, namely T.state is weakly finitely indexed (WFI) (the underlying equality for states is Leibniz equality). First, we show how to compute, starting from a regular tree χ, a cyclic term that is semantically equivalent to χ. Let $\chi : \textsc{Reg}_I$ be a regular tree. By definition, we have that the type of successors of χ, namely $\chi{\rightsquigarrow}$, is WFI. Clearly, the product of two WFI types is again WFI, thus $T.\mathsf{state} \times \chi{\rightsquigarrow}$ is again WFI. We abbreviate the type of indices $\textsc{Fin}\,\#(T.\mathsf{state} \times \chi{\rightsquigarrow})$ as $\mathcal{I}$. Consequently, we have an indexing map index : $T.\mathsf{state} \times \chi{\rightsquigarrow} \to \mathcal{I}$ and a valuation map value : $\mathcal{I} \to T.\mathsf{state} \times \chi{\rightsquigarrow}$ such that index is a right-inverse of value. The function building a cyclic term starting from an arbitrary subtree of χ is defined as follows:

$$
\begin{array}{l}
⦇ T ⦈ : \forall\{l : \textsc{List}\,\mathcal{I}\}.\ \lceil l \rceil \to \textsc{Env}_O(\mathsf{length}\, l) \to T.\mathsf{state} \to \chi{\rightsquigarrow} \to \mathbb{C}_O(\mathsf{length}\, l) \\
⦇ T ⦈\ \{l\}\ B\ \Gamma\ q\ t \equiv \ \textbf{match}\ (q,t) \in^? l\ \textbf{with} \\
\qquad\qquad\qquad\qquad \mid \mathsf{yes}(i, _) \Rightarrow \Gamma(i) \\
\qquad\qquad\qquad\qquad \mid \mathsf{no}\, np \quad \Rightarrow \mathsf{rec}^+\ (\eta'\ q\ t)\ (⦇ T ⦈\ (np\lceil :: \rceil B)\ (\mathsf{var\,zero}; \Gamma)) \\
\qquad\qquad\qquad\qquad \textbf{end}
\end{array}
$$

The function η' is an extension of η such that it operates on subtrees of the root tree χ :

$$
\begin{array}{l}
\eta' : T.\mathsf{state} \times \chi{\rightsquigarrow} \to O^+(T.\mathsf{state} \times \chi{\rightsquigarrow}) \\
\eta'\ \ (q, (t, p)) \ \equiv\ \mathsf{map}^{O^+(T.\mathsf{state} \times -)}\ h\ (\mathsf{rhs}\ q\ (\mathsf{look\text{-}ahead}_{(\mathsf{depth}\, q)}\ t)) \\
\qquad \textbf{where}\ \ h : \{p \mid p \in \mathscr{P}_{(\mathsf{depth}\, q)}(\mathsf{look\text{-}ahead}_{(\mathsf{depth}\, q)}\ t)\} \to \chi{\rightsquigarrow} \\
\qquad\qquad\quad h\ \ (p, P) \ \equiv (\downarrow [P], p{+\!\!+}p') \\
\qquad\qquad\quad p' : t \rightsquigarrow\downarrow [P]
\end{array}
$$

where the path p' is obtained from ($\mathsf{look\text{-}ahead}{\rightsquigarrow}$). The function rec^+ is defined as follows:

$$
\begin{array}{l}
\mathsf{rec}^+ : O^+(X) \to (X \to \mathbb{C}_S(\mathsf{suc}\, n)) \to \mathbb{C}_S(n) \\
\mathsf{rec}^+\ \ (o, os)\ \ f \ \equiv\ \mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\lambda t.\ t \ggg\!= f)\ os
\end{array}
$$

while the function $_ \ggg\!= _$ is given by:

$$
\begin{array}{l}
_ \ggg\!= _ : \forall X.\ \forall(n : \mathbb{N}).\ S^*(X) \to (X \to \mathbb{C}_S(n)) \to \mathbb{C}_S(n) \\
t \ggg\!= f \ \equiv\ \mathsf{fold}_*\ f\ [\lhd]\ t \\
\qquad \textbf{where}\ \ [\lhd] : \forall\{n : \mathbb{N}\}.\ S(\mathbb{C}_S(n)) \to \mathbb{C}_S(n) \\
\qquad\qquad\quad [\lhd]\ \ (o, os) \ \equiv \mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ \mathsf{weaken}\ os.
\end{array}
$$

Then, we prove that the semantics mapping regular trees to cyclic terms coincides with the semantics defined previously on plain infinite trees. We begin by the following generalized lemma:

$$
\begin{array}{l}
\forall(n : \mathbb{N}).\ \forall(l : \textsc{List}\,\mathcal{I}).\ \forall(B : \lceil l \rceil).\ \forall(q : T.\mathsf{state}).\ \forall(t : \chi{\rightsquigarrow}). \\
\qquad \forall(\Gamma : \textsc{Env}_O(\mathsf{length}\, l)).\ \forall(\rho : \textsc{Closure}_O(n)). \\
\qquad\qquad (\forall(x : \textsc{Fin}(\mathsf{length}\, l)).\ l[i] \sim_n [\![\, \Gamma(x) \,]\!]_\rho) \to ⦇T⦈\ q\ t \sim_n [\![\, ⦇ T ⦈\ l\ B\ q\ t\ \Gamma \,]\!]_\rho.
\end{array}
\qquad (⦇ T ⦈)
$$

This proof has the same structure than Theorem 3.1. It only requires the following intermediate lemma:

$$
\begin{array}{l}
\forall(S : \textsc{Sig}).\ \forall X, Y.\ \forall(u : S^*(X)).\ \forall(v : S^*(Y)).\ \forall(m, n : \mathbb{N}). \\
\qquad \forall(f : X \to \mathrm{co}\mathbb{T}_S).\ \forall(g : Y \to \mathbb{C}_S(m)).\ \forall(\rho : \textsc{Closure}_S(m)).\ \forall(h : Y \to X). \\
\qquad\qquad u = \mathsf{map}^{S^*}\ h\ v \to (\forall(y : Y).\ f(h(y)) \sim_n [\![\, g(y) \,]\!]_\rho) \to (u \ggg\!= f) \sim_n [\![\, v \ggg\!= g \,]\!]_\rho.
\end{array}
$$

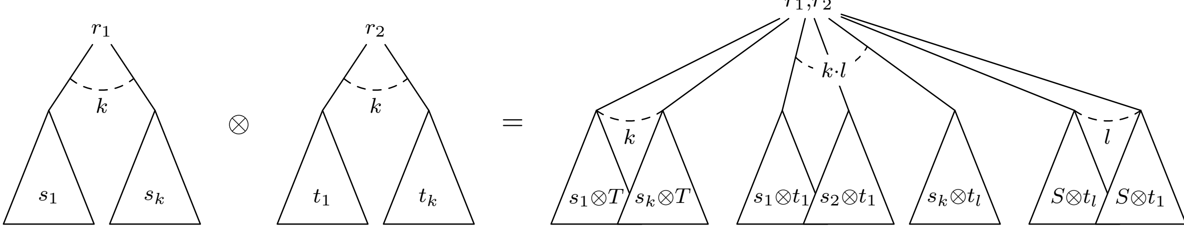


Figure 3.3. Lifting of product of trees to trees over product of signatures

The proof is straightforward by induction on v. Finally, from ($\llparenthesis T \rrparenthesis$) and Theorem 2.6, we can easily deduce that the semantics coincides on the domain of regular trees:

$$\forall q.\ \forall(t : \text{REG}_I).\ \llparenthesis T \rrparenthesis\ q\ t \sim [\![\llparenthesis T \rrparenthesis'\ q\ t]\!] \qquad \text{(sem-equiv)}$$

where $\llparenthesis T \rrparenthesis' := \llparenthesis T \rrparenthesis\ []\ B\ q\ t\ \varnothing$ with $B : \lceil [] \rceil$ obtained from the finiteness of the type of FIN as given by Proposition A.15, and $\varnothing : \text{ENV}_O(\mathsf{zero})$ represents the empty environment. Finally, we can show that $\llparenthesis T \rrparenthesis : \text{co}\mathbb{T}_I \to \text{co}\mathbb{T}_O$ preserves the regularity of trees. To this end, it suffices to give a cyclic term morphism:

$$\llparenthesis T \rrparenthesis^{\mathbb{C}} : \mathbb{C}_I \to \mathbb{C}_O$$
$$\llparenthesis T \rrparenthesis^{\mathbb{C}}\ c\ \equiv\ \llparenthesis T \rrparenthesis'\ T.\mathsf{init}\ [\![c]\!]$$

and prove that:

$$\forall(c : \mathbb{C}_I).\ \llparenthesis T \rrparenthesis([\![c]\!]) \sim [\![\llparenthesis T \rrparenthesis^{\mathbb{C}}\ c]\!]$$

which is given by (sem-equiv). □

3.6. BINARY TOP-DOWN TREE TRANSDUCERS

In this section, we study the problem of extending the formalism of tree transducers to operate over multiple trees at the same time. In order to ease the presentation, the results presented here deal only with the case of binary tdtt but they can be generalized straightforwardly to n-ary tdtt.

Instead of generalizing each variant (productive/guarded tdtt, tdtt with finite look-ahead) to the n-ary case, we introduce an operation lifting a product of trees to a tree over a product signatures. Then, each formalism introduced in the previous section can be used without modification. Intuitively, we define one possible "internalization" of the binary case to the unary case. To this end, we introduce a combinator on signatures:

Definition 15 Product of signatures

Let S_1, S_2 be two signatures. We define the product of signatures, denoted $S_1 \otimes S_2$, as follows:

$$_\otimes_ : \text{SIG} \to \text{SIG} \to \text{SIG}$$
$$S_1 \otimes S_2\ \equiv\ S_1.\mathsf{Op} \times S_2.\mathsf{Op} \lhd \mathsf{uncurry}(\lambda o_1.\ \lambda o_2.\ S_1.\mathsf{Ar}\ o_1 + S_1.\mathsf{Ar}\ o_1 \times S_2.\mathsf{Ar}\ o_2 + S_2.\mathsf{Ar}\ o_2).$$

Intuitively, this signature is used as a way to encode a traversal into a product of tree. The arity function is composed of three main parts. Here, the sum models choice. Thus, the arity function should be read as: either we walk down into the first tree while keeping the second tree intact or we walk down both trees, or we walk down into the second while leaving the first tree untouched.

Definition 16 Product lifting

Let S_1, S_2 : SIG be two signatures. The operation lifting a product on trees t_1 and t_2, over a tree on a product of signatures, denoted $t_1 \otimes t_2$, is defined by corecursion, as follows:

$$_\otimes_ : \text{co}\mathbb{T}_{I_1} \to \text{co}\mathbb{T}_{I_2} \to \text{co}\mathbb{T}_{I_1 \otimes I_2}$$

$$\begin{aligned} t_1 \ \otimes \ t_2 \ \equiv \ \mathsf{root}\, t_1, \mathsf{root}\, t_2 \blacktriangleleft \lambda k.\ \ & \textbf{match } k \textbf{ with} \\ & \mid \mathsf{inl}(\mathsf{inl}(i,j)) \Rightarrow t_1(i) \otimes t_2(j) \\ & \mid \mathsf{inr}(\mathsf{inl}(i)) \ \ \Rightarrow t_1(i) \otimes t_2 \\ & \mid \mathsf{inr}(\mathsf{inr}(j)) \ \ \Rightarrow \ \ t_1 \otimes t_2(j) \\ & \textbf{end} \end{aligned}$$

Note that in the definition, we have silently used the isomorphisms $\text{Fin}(n+m) \cong \text{Fin}\, n \uplus \text{Fin}\, m$ and $\text{Fin}(n * m) \cong \text{Fin}\, n \times \text{Fin}\, m$. The lifting operation is represented graphically in Figure 3.3. Intuitively, this operation represents the combination of a synchronous and asynchronous product of trees. In particular, it models the combination of choices that are available regarding what should be consumed by the rewrite rules.

Theorem 5 Product lifting preserves regularity

Let I_1, I_2 : SIG be two signatures. The function lifting the product of trees to trees on a product of signatures preserves regularity of trees.

Proof. The proof follows the same structure and arguments than the one showing that tdtts with finite look-ahead preserve regularity of trees (Thm. 3.4). First, we build a function computing the product lifting on cyclic terms. Assume that we have two regular trees $\chi_1 : \text{Reg}_{I_1}$, $\chi_2 : \text{Reg}_{I_2}$. By definition, both types $\chi_1 \rightsquigarrow$ and $\chi_2 \rightsquigarrow$ are weakly finitely indexed and so is their product $\chi_1 \rightsquigarrow \times \chi_2 \rightsquigarrow$. We abbreviate the type of indices $\text{Fin}\, \#(\chi_1 \rightsquigarrow \times \chi_2 \rightsquigarrow)$ as $\mathcal{I}$.

$$\mathsf{prod}_{\mathbb{C}} : \forall (l : \text{List}\, I).\ \lceil l \rceil \to \text{Env}_O(\mathsf{length}\, l) \to \chi_1 \rightsquigarrow \to \chi_2 \rightsquigarrow \to \mathbb{C}_O(\mathsf{length}\, l)$$

$$\begin{aligned} \mathsf{prod}_{\mathbb{C}}\ \ \{l\}\ \ B\ \ \Gamma\ \ t_1\ \ t_2 \ \equiv \ \ & \textbf{match } (t_1, t_2) \in^{\#}?\ l \textbf{ with} \\ & \mid \mathsf{yes}(i, _) \Rightarrow \Gamma(i) \\ & \mid \mathsf{no}\, np \quad \ \Rightarrow \mathsf{rec}\,(\mathsf{root}\, t_1, \mathsf{root}\, t_2) \lhd \mathsf{tabulate}\ \mathsf{prod}_{\mathbb{C}}\text{-}\mathsf{sub} \\ & \textbf{end} \end{aligned}$$

$$\begin{aligned} \textbf{where}\ \ & \mathsf{prod}_{\mathbb{C}}\text{-}\mathsf{sub} : \text{Fin}(I_1.\,\mathsf{Ar}(\mathsf{root}\, t_1) \times I_2.\,\mathsf{Ar}(\mathsf{root}\, t_2)) \to \mathbb{C}_O(\mathsf{length}(\mathsf{suc}\, l)) \\ & \mathsf{prod}_{\mathbb{C}}\text{-}\mathsf{sub}\ \ (\mathsf{inl}(\mathsf{inl}(i,j))) \ \equiv \ \mathsf{prod}_{\mathbb{C}}\ _\ (np[::]B)\ (\mathsf{var}\,\mathsf{zero}; \Gamma)\ (t_1(i))\ (t_2(j)) \\ & \mathsf{prod}_{\mathbb{C}}\text{-}\mathsf{sub}\ \ (\mathsf{inr}(\mathsf{inl}(i))) \quad \equiv \ \mathsf{prod}_{\mathbb{C}}\ _\ (np[::]B)\ (\mathsf{var}\,\mathsf{zero}; \Gamma)\ (t_1(i))\ t_2 \\ & \mathsf{prod}_{\mathbb{C}}\text{-}\mathsf{sub}\ \ (\mathsf{inr}(\mathsf{inr}(j))) \quad \equiv \ \mathsf{prod}_{\mathbb{C}}\ _\ (np[::]B)\ (\mathsf{var}\,\mathsf{zero}; \Gamma)\ t_1\ (t_2(j)). \end{aligned}$$

Then, we prove that the semantics mapping regular trees to cyclic terms coincides with the semantics defined previously on plain infinite trees. To this end, we begin by the following generalized lemma:

$$\begin{aligned} & \forall (n : \mathbb{N}).\ \forall (l : \text{List}\, \mathcal{I}).\ \forall (B : \lceil l \rceil).\ \forall (t_1 : \chi_1 \rightsquigarrow).\ \forall (t_2 : \chi_2 \rightsquigarrow). \\ & \quad \forall (\Gamma : \text{Env}_O(\mathsf{length}\, l)).\ \forall (\rho : \text{Closure}_O(n)). \\ & \quad\quad (\forall (x : \text{Fin}(\mathsf{length}\, l)).\ l[i] \sim_n [\![\Gamma(x)]\!]_\rho) \to t_1 \otimes t_2 \sim_n [\![\mathsf{prod}_{\mathbb{C}}\ l\ B\ t_1\ t_2\ \Gamma]\!]_\rho. \end{aligned}$$

The proof proceeds by induction on the depth n. Then, starting from χ_1 and χ_2, we instantiate the result above and with Theorem 2.2, we obtain:

$$\chi_1 \otimes \chi_2 \sim [\![\mathsf{prod}_{\mathbb{C}}\ [\,]\ B\ t_1\ t_2\ \varnothing]\!]$$

where $B : \lceil [\,] \rceil$ is a proof obtained from the finiteness of Fin and $\varnothing$ denotes the empty environment. Consequently, we have shown that there exists a cyclic term representing the product lifting of tree regular trees. $\square$

Example. The zipWith operation over two streams is defined corecursively as follows:

$$\mathsf{zipWith} : \textsc{Stream}\,A \to \textsc{Stream}\,B \to \textsc{Stream}\,C$$
$$\mathsf{zipWith}\ \ (a :: as)\ \ (b :: bs)\ \equiv\ (f\ \ a\ \ b) :: \mathsf{zipWith}\ as\ bs$$

where $f : A \to B \to C$ is an arbitrary function. Then, we can show that zipWith preserves regularity. To this end, we define a tree transducer from streams over the product of signatures to streams ${}^{\mathrm{G}}\mathrm{TDTT}\ (\textsc{Str}_A \otimes \textsc{Str}_B)\ \textsc{Str}_C$. Clearly, the following corecursive definition can be expressed as a guarded top-down tree transducer:

$$\mathsf{zipWith}_{\otimes} : \mathrm{co}\mathbb{T}_{\textsc{Str}_A \otimes \textsc{Str}_B} \to \mathrm{co}\mathbb{T}_{\textsc{Str}_C}$$
$$\mathsf{zipWith}_{\otimes}\ \ ((a, b) \blacktriangleleft abs)\ \equiv\ (f\ \ a\ \ b) \blacktriangleleft \lambda k.\ \mathsf{zipWith}_{\otimes}(abs(\mathsf{inl}(\mathsf{inl}(\mathsf{zero}, \mathsf{zero})))).$$

Finally, zipWith can be obtained from the induced tree morphism of $\mathsf{zipWith}_{\otimes}$ and by precomposition with the product lifting function:

$$\mathsf{zipWith}\ \ f\ \ s_1\ \ s_2\ \equiv\ \mathsf{zipWith}_{\otimes}(s_1 \otimes s_2).$$

Since both function preserves regularity, namely $\mathsf{zipWith}_{\otimes}$ and $(s_1 \otimes s_2)$, so does zipWith.

3.7. CONCLUSION

In this chapter, we have studied the problem of defining regularity preserving tree morphisms. To this end, we used the formalism of top-down tree transducer as a way to model the abstract syntax of a subset of corecursive functions. Indeed, our goal was to find *a syntactic criterion* to characterize the subset of corecursive functions that preserve regularity. In this context, we have introduced various sorts of tree transducers. The first one, called guarded top-down tree transducers, was used to model the abstract syntax of productive corecursive function consuming the root symbol of the input tree at each step of the recursion. Then, we generalized this recursive scheme by allowing ε-rules, that is, rewrite rules which do not necessarily consume the root of the input tree. In essence, this new sort of tree transducer allows some rewrite rules to be *purely productive*. In this last extension, we generalized the type of rewrite rules further with the introduction of guarded top-down tree transducers with finite look-ahead. This extension models a subset of corecursive functions with deep pattern-matching, *i.e.*, when patterns can be matched recursively. Finally, we introduced a product lifting operation on infinite trees in order to derive binary tree transducers from the previous ones.

CHAPTER FOUR

APPLICATIONS

In this chapter, we study two main applications of the theory of regular trees developed thus far. First, we consider the problem of defining an operation, namely parallel composition, on a process algebra. In particular, this process algebra contains a constructor allowing the definition of recursive processes. Semantically, recursive processes are identified with their infinite unfolding. In this context, we show how a termination problem, occurring in the definition of the parallel composition, can be recast into a productivity problem. Next, we define an interpretation of the coalgebraic μ-calculus [CKP11] over regular trees. In particular, we prove that satisfiability is decidable. Finally, we give a proof that bisimilarity is decidable on regular trees through a reduction to a model-checking problem.

4.1. SEQUENTIALIZATION OF PROCESSES

In this section, we show how a termination problem can be viewed as a productivity problem.

Problem

Consider the problem of defining the synchronous parallel product of processes for a *fragment* of process algebra such as BPA [BK88] or CSP [Hoa78]. Intuitively, this operation consists in a *sequentialization* of the parallel composition operator, in the sense that this operator is eliminated. The syntax PROC of processes is defined inductively as follows:

$$P, Q ::= \mathsf{STOP} \mid \mathsf{SKIP} \mid a \to P \mid P \,\square\, Q \mid \mu X.\, P \mid X$$

where SKIP (resp. STOP) denotes the successful (resp. unsuccessful) terminating process. The process $a \to P$ accepts the action a, where a ranges over a set of atomic actions, and then behaves as P. The non-deterministic choice between P and Q is denoted $P \,\square\, Q$. Finally, $\mu X.\, P$ denotes a recursively-defined process. Various semantic models for such process algebra exists. For instance, there are denotational models based on trace-semantics where processes are identified with the sequence of their observable communications. Operational semantics interpret processes over labeled transition systems. Regardless of the actual model considered $[\![-]\!] : \textsc{Proc} \to \mathcal{D}$, the recursive operator ought to be indistinguishable from its unfolding. Consequently, the following equation shall hold:

$$[\![\mu X.\, P]\!] =_{\mathcal{D}} [\![P[X := \mu X.\, P]]\!]. \qquad (\mu\text{-UNFOLD})$$

The problem with this equation is that it is more of a semantic notion rather than a syntactic one. Let us recall that the syntax of processes is given by an inductive definition. Consequently, this definition induces an *induction principle*. For the type PROC, this induction principle is given by:

$$\text{STOP}\ \frac{}{P(\mathsf{STOP})}, \qquad \text{SKIP}\ \frac{}{P(\mathsf{SKIP})}, \qquad \text{ACT}\ \frac{P(p)}{P(a \to P)} \qquad \text{CHOICE}\ \frac{P(p) \quad P(q)}{P(p \,\square\, q)}$$

$$\text{VAR}\ \frac{}{P(\mathsf{var}\ x)} \qquad \text{REC}\ \frac{P(p)}{P(\mu X.\, p)}.$$

Here, the rule REC does not take into account the equation (μ-UNFOLD). The main issue being that an inductive framework is based around the notion of well-foundedness. However, this notion may be violated by the unfolding operation. Indeed, an unfolded recursive process may lead to a strictly larger term—syntactically speaking. Now, consider the following axiomatization of the synchronous parallel composition operation:

$$\begin{array}{lcll@{\qquad}lcllr}
\mathsf{STOP} & \| & P & = \mathsf{STOP} & (a \to P) & \| & (a \to Q) & = a \to (P \| Q) & \\
P & \| & \mathsf{STOP} & = \mathsf{STOP} & (a \to P) & \| & (b \to Q) & = \mathsf{STOP} & a \neq b \\
P & \| & \mathsf{SKIP} & = P & (P \,\square\, Q) & \| & R & = (P\|R) \,\square\, (Q\|R) & \\
\mathsf{SKIP} & \| & P & = P & R & \| & (P \,\square\, Q) & = (P\|R) \,\square\, (Q\|R). &
\end{array}$$

Here, no axiom is given for the μ constructor because it can be derived. Indeed, parallel composition should be congruence and thus shall satisfy:

$$(\mu X.\, P) \,\|\, Q = (P[X := \mu X.\, P]) \,\|\, Q, \qquad P \,\|\, (\mu X.\, Q) = P \,\|\, (Q[X := \mu X.\, Q]).$$

This set of axioms can be read as a recursive function definition. If we try to define the parallel composition operation as a *function* on processes in a proof assistant such as COQ, this definition would be rejected by the termination checker. The problem here concerns the last two equations. Indeed, for both right-hand sides the function is called recursively with arguments that are not subterms of the input terms. However, it is not obvious how to justify that such computation does indeed terminate. Now, we recast this problem by considering the use of non-well-founded syntax, that is, by defining the syntax of process, denoted ∞PROC, coinductively:

$$P, Q ::= \mathsf{STOP} \mid \mathsf{SKIP} \mid a \to P \mid P \,\square\, Q.$$

The only notable difference with the previous syntax is the removal of the μ constructor. In a coinductive setting, it is possible to define infinite objects and thus recursive processes. Now, if we define the parallel composition operation corecursively on the non-well-founded syntax, namely ∞PROC, we obtain:

$$\begin{array}{lcll}
\multicolumn{4}{l}{_\|^\infty_ : \infty\mathrm{PROC} \to \infty\mathrm{PROC} \to \infty\mathrm{PROC}} \\
\mathsf{STOP} & \|^\infty & _ & \equiv \mathsf{STOP} \\
_ & \|^\infty & \mathsf{STOP} & \equiv \mathsf{STOP} \\
\mathsf{SKIP} & \|^\infty & P & \equiv P \\
P & \|^\infty & \mathsf{SKIP} & \equiv P \\
(a \to P) & \|^\infty & (b \to Q) & \equiv \textbf{if } a \stackrel{?}{=} b \textbf{ then } a \to P \,\|^\infty\, Q \textbf{ else } \mathsf{STOP} \textbf{ end} \\
(P \,\square\, Q) & \|^\infty & R & \equiv (P \,\|^\infty\, R) \,\square\, (Q \,\|^\infty\, R) \\
R & \|^\infty & (P \,\square\, Q) & \equiv (R \,\|^\infty\, P) \,\square\, (R \,\|^\infty\, Q).
\end{array}$$

Such definition is accepted by the productivity checker of COQ. Indeed, every corecursive call is guarded by a constructor. Clearly, we can define a function from PROC to ∞PROC by mapping recursive processes to their unfolding[1]. If we could define a map in the opposite direction, namely from ∞PROC to PROC, we could use the corecursive definition in order to *derive* the parallel

[1] We can extend ∞PROC with an additional value to deal with non-guarded processes such as $\mu X.X$.

composition operation on PROC. Therefore, in this context, a termination problem can be recast into a productivity problem.

Solution

As explained previously, our goal is to define, in a proof assistant such as COQ, a map representing the parallel composition operation on processes. Here, we show how the framework of regular trees introduced in the previous chapters provides a solution to this problem.

The syntax of processes PROC can be expressed as a cyclic term $\mathbb{C}_{\text{PROC}}$ where the signature PROC is given by:

$$\text{PROC} := \left\{ \mathsf{STOP}^{(0)} \ ; \ \mathsf{SKIP}^{(0)} \ ; \ \to_a^{(1)} \ ; \ \Box^{(2)} \right\}.$$

Moreover, we defined a semantic function $[\![-]\!] : \mathbb{C}_{\text{PROC}} \to \text{REG}_{\text{PROC}}$ mapping cyclic terms to regular trees. Consequently, we can define a map by corecursion following the definition of $_\|^{\infty}_$. Clearly, this function is productive and for each equation, each argument occurring in the corecursive call is a subterm of the input terms. As a result, this corecursive definition can be expressed as a 2-ary guarded top-down tree transducers T. Then, since guarded top-down tree transducers preserve regularity, we can conclude that the parallel composition operation $_\|^{\infty}_$ preserves regularity. Finally, by exploiting the inverse map $[\![-]\!]^{-1} : \text{REG}_{\text{PROC}} \to \mathbb{C}_{\text{PROC}}$, we can obtain a cyclic term. To summarize, we can derive the definition of $_\|_$ from $_\|^{\infty}_$ as follows:

$$P \parallel Q := [\![(\!| T |\!) \ [\![P]\!] \ [\![Q]\!]]\!]^{-1}.$$

In particular, this derived parallel composition operator respects the (μ-UNFOLD) equation, *i.e.*,

$$(\mu X.\, P) \parallel Q \approx (P[X := \mu X.\, P]) \parallel Q.$$

4.2. MODEL-CHECKING ON REGULAR TREES

In this section, we introduce the syntax for a coalgebraic μ-calculus [CKP11] and define its semantics interpreted over an arbitrary T-coalgebra. The treatment of the underlying modal logic is parametric in the sense that modal operators are given by an arbitrary signature. Instantiated with the suitable signature, one can obtain the propositional modal μ-calculus [Koz83]. Next, we show that formula satisfiability is decidable when the semantics is interpreted over finite T-coalgebras. Finally, this abstract framework is instantiated over the type of regular trees thus yielding a model-checking procedure.

4.2.1. Syntax

The full syntax of the propositional modal μ-calculus [Koz83] is given as follows:

$$\Phi ::= X \mid p \mid \top \mid \bot \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \langle a \rangle \Phi \mid [a]\Phi \mid \nu X.\, \Phi \mid \mu X.\, \Phi$$

where X ranges over variables, p denotes atomic propositions and a ranges over a set of actions. The operators *true* ($\top$), *false* ($\bot$), *negation* ($\neg_$), *and* ($_\wedge_$), *or* ($_\vee_$) and *propositional variables* (p) are those from the propositional logic. The operators *box* ($[_]$) and *diamond* ($\langle_\rangle$) are modal operators. Finally, the least fixpoint operator is denoted by μ while ν denotes the greatest fixpoint.

It is quite common to restrict the syntax of the μ-calculus to a minimal core by removing operators which are derivable by duality. However, some restrictions on the negation operator are imposed in order to ensure that the semantics is monotone thus justifying the existence of fixpoints. More concretely, the set of terms is restricted to the ones where the negation operator

appear *positively* in front of variables. Informally, positivity means that there shall be only an even number of nested applications of the negation operator in front of variables. For instance, in the term $\nu X.\, p \wedge \neg X$, the negation operator occurs negatively in front of X while it occurs positively in the term $\nu X.\, p \wedge \neg[a]\neg X$.

Instead of giving the syntax of the propositional μ-calculus, we follow the approach presented in [CKP11] which abstract away the set of modal operators into a signature $\Lambda : \textsc{Sig}$.

$$
\begin{aligned}
&\textbf{inductive}\ \textsc{Term}\ (n:\mathbb{N}) : \textsc{Type}\\
&\quad \mathsf{var} : \textsc{Fin}\, n \to \textsc{Term}\, n\\
&\quad \mu : \textsc{Term}(\mathsf{suc}\, n) \to \textsc{Term}\, n\\
&\quad \nu : \textsc{Term}(\mathsf{suc}\, n) \to \textsc{Term}\, n\\
&\quad _\wedge_ : \textsc{Term}\, n \to \textsc{Term}\, n \to \textsc{Term}\, n\\
&\quad _\vee_ : \textsc{Term}\, n \to \textsc{Term}\, n \to \textsc{Term}\, n\\
&\quad \circ : \Lambda(\textsc{Term}\, n) \to \textsc{Term}\, n\\
&\quad \bullet : \Lambda(\textsc{Term}\, n) \to \textsc{Term}\, n.
\end{aligned}
$$

The binders μ and ν are implemented by means of a nested datatype definition and follow the de Bruijn typing discipline. The last two constructors, namely $\circ$ and its dual $\bullet$, represent the modal operators as given by the signature Λ. Let us emphasize that we have removed the negation operator $\neg$ from the syntax. In addition, we have also removed $\top$ (resp. $\bot$) designating the universally true (resp. false) formula. However, all these operators can be derived. In the following, we give the definition of the negation operator which consists in swapping each operator by its dual:

$$
\begin{aligned}
&\neg_ : \forall\{n:\mathbb{N}\}.\ \textsc{Term}\, n \to \textsc{Term}\, n\\
&\neg(\mathsf{var}\, x) &&\equiv\ \mathsf{var}\, x\\
&\neg(\mu\Phi) &&\equiv\ \nu(\neg\Phi)\\
&\neg(\nu\Phi) &&\equiv\ \mu(\neg\Phi)\\
&\neg(\Phi_1 \wedge \Phi_2) &&\equiv\ (\neg\Phi_1) \vee (\neg\Phi_2)\\
&\neg(\Phi_1 \vee \Phi_2) &&\equiv\ (\neg\Phi_1) \wedge (\neg\Phi_2)\\
&\neg(\circ\, \Phi) &&\equiv\ \bullet(\Lambda\ (\neg_)\ \Phi)\\
&\neg(\bullet\, \Phi) &&\equiv\ \circ(\Lambda\ (\neg_)\ \Phi).
\end{aligned}
$$

Notice that, for the last two constructors, namely $\circ$ and $\bullet$, we use the functor induced by the signature Λ (see Section 1.3.1). As a result, we can lift the recursive function $\neg_$ to operate on terms of type $\Lambda(\textsc{Term}\, n)$ thus we have $\Lambda(\neg_) : \Lambda(\textsc{Term}\, n) \to \Lambda(\textsc{Term}\, n)$. Furthermore, it is interesting to remark that the recursive call $\Lambda\ (\neg_)\ \Phi$ is accepted by the COQ termination checker. The true (resp. false) formula is defined as follows:

$$
\begin{aligned}
&\top : \forall\{n:\mathbb{N}\}.\ \textsc{Term}\, n \qquad &&\bot : \forall\{n:\mathbb{N}\}.\ \textsc{Term}\, n\\
&\top\ \equiv\ \nu(\mathsf{var}\,\mathsf{zero}). &&\bot\ \equiv\ \mu(\mathsf{var}\,\mathsf{zero}).
\end{aligned}
$$

We introduce, given a predicate $P : A \to \textsc{Prop}$, universal quantification for vectors and signatures:

$$
\begin{aligned}
&\textsc{Vec}^{\forall} : \forall\{n\}.\ \textsc{Vec}\ A\ n \to \textsc{Prop} \qquad &&_^{\forall} : \forall(S:\textsc{Sig}).\ S(A) \to \textsc{Prop}\\
&\textsc{Vec}^{\forall}\ \ v\ \equiv\ \forall k.\ P(v(k)). &&S^{\forall}\ \ P\ \equiv\ \textsc{Vec}^{\forall}\, P \circ \mathsf{proj}_2.
\end{aligned}
$$

The binary case is given by:

$$
\begin{aligned}
&\textsc{Vec}^{\forall_2} : \forall\{m,n\}.\ \textsc{Vec}\ A\ m \to \textsc{Vec}\ A\ n \to \textsc{Prop} \qquad &&_^{\forall_2} : \forall(S:\textsc{Sig}).\ S(A) \to S(A) \to \textsc{Prop}\\
&\textsc{Vec}^{\forall_2}\ \ v\ \ v' \equiv \exists(e:m=n).\ \forall k.\ P\ (v(k))\ (v'(e_* k)). &&S^{\forall_2}\ \ s\ \ s' \equiv \textsc{Vec}^{\forall_2}\ (\mathsf{proj}_2\, s)\ (\mathsf{proj}_2\, s').
\end{aligned}
$$

4.2.2. Denotational Semantics

Coalgebraic μ-calculus formulae are interpreted over an arbitrary T-coalgebra. For the remaining of this section, given an endofunctor T, we fix an arbitrary T-coalgebra $(X, \gamma : X \to T(X))$. The domain of the denotational semantics is given by mapping from environments to subsets of states. We denote $\wp : \textsc{Setoid} \to \textsc{Setoid}$, the contravariant power set functor. Type-theoretically, $\wp$ is defined as a characteristic function:

$$\wp : \textsc{Setoid} \to \textsc{Setoid} \qquad _^{-1} : (X \longrightarrow Y) \to \wp(Y) \longrightarrow \wp(X)$$
$$\wp\ \ X \ \equiv\ X \longrightarrow \textsc{Prop}, \qquad f^{-1}\ \ A \ \equiv\ \{\, x : f(x) \in A \,\}.$$

Here, $\{\, x : P \,\} : \wp(_)$ is a notation for $\lambda x.\ P : \wp(_)$. Moreover, we may denote function application $P(x)$ by $x \in P$ when P designates a predicate. We recall that the type $\wp(X)$ is a complete lattice (see [DP02]). The equivalence (resp. order) relation on $\wp(X)$ is given by:

$$_{\approx}_ : \wp(X) \to \wp(X) \to \textsc{Prop} \qquad _{\sqsubseteq}_ : \wp(X) \to \wp(X) \to \textsc{Prop}$$
$$A \approx B \ \equiv\ \forall x.\ x \in A \leftrightarrow x \in B, \qquad A \sqsubseteq B \ \equiv\ \forall x.\ x \in A \to x \in B.$$

The join and meet operation are given respectively by:

$$_{\sqcup}_ : \wp(X) \to \wp(X) \to \wp(X) \qquad _{\sqcap}_ : \wp(X) \to \wp(X) \to \wp(X)$$
$$A \sqcup B \ \equiv\ \{\, x : x \in A \lor x \in B \,\}, \qquad A \sqcap B \ \equiv\ \{\, x : x \in A \land x \in B \,\}.$$

The greatest (resp. least) lower (resp. upper) bound operator is defined as:

$$\bigsqcup : \wp(\wp(X)) \to \wp(X) \qquad \bigsqcap : \wp(\wp(X)) \to \wp(X)$$
$$\bigsqcup I \ \equiv \{\, x : \exists A.\ A \in I \land x \in A \,\}, \qquad \bigsqcap I \ \equiv \{\, x : \forall A.\ A \in I \to x \in A \,\}.$$

If we assume the law of excluded-middle (LEM), the type $\wp(X)$ is also a complete boolean algebra and the negation operation is given by:

$$\complement : \wp(X) \to \wp(X) \qquad \mathsf{univ} : \wp(X) \qquad \varnothing : \wp(X)$$
$$\complement A \ \equiv\ \{\, x : x \notin A \,\}, \qquad \mathsf{univ} \ \equiv\ \{\, x : \top \,\}, \qquad \varnothing \ \equiv\ \{\, x : \bot \,\}.$$

Orders. A preorder on a type A is a binary relation $_{\sqsubseteq}_ : A \to A \to \textsc{Prop}$ that is reflexive and transitive. A partial order is a preorder that is also antisymmetric. Given a preorder $(A, \sqsubseteq)$ (resp. partial order), we can define the dual order, denoted $(A, \sqsubseteq^\partial)$ as follows:

$$\forall (a, a' : A).\ a \sqsubseteq^\partial a' \overset{\text{def}}{\iff} a' \sqsubseteq a.$$

A function $f : (A, \sqsubseteq_A) \to (B, \sqsubseteq_B)$ is said to be order-preserving when it is monotone, *i.e.*,

$$\textsc{monotone}(f) := \forall (a, a' : A).\ a \sqsubseteq_A a' \to f(a) \sqsubseteq_B f(b).$$

We denote by $A \xrightarrow{m} B$ the type of function from A to B that are order-preserving. Concretely,

$$A \xrightarrow{m} B := \{f : A \to B \mid \textsc{monotone}(f)\}.$$

A function is said to be anti-monotone when it is monotone on the dual order:

$$\textsc{anti-monotone}(f) := \forall (a, a' : A).\ a \sqsubseteq_A a' \to f(b) \sqsubseteq_B f(a).$$

Fixpoints. The fixpoint operators, namely lfp and gfp, are derived as follows:

$$\begin{aligned}
&\mathsf{lfp} : (\wp(X) \to \wp(X)) \to \wp(X) \\
&\mathsf{lfp}(f) \;\equiv\; \textstyle\bigsqcap \mathsf{pre}_f \\
&\qquad \textbf{where}\;\; \mathsf{pre}_f : \wp(\wp(X)) \\
&\qquad\qquad\quad \mathsf{pre}_f = \{\, X : f(X) \sqsubseteq X \,\},
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{gfp} : (\wp(X) \to \wp(X)) \to \wp(X) \\
&\mathsf{gfp}(f) \;\equiv\; \textstyle\bigsqcup \mathsf{post}_f \\
&\qquad \textbf{where}\;\; \mathsf{post}_f : \wp(\wp(X)) \\
&\qquad\qquad\quad \mathsf{post}_f = \{\, X : X \sqsubseteq f(X) \,\}.
\end{aligned}$$

When $f : \wp(X) \to \wp(X)$ is monotone, we have that

$$\begin{aligned}
\mathsf{lfp}(f) \approx f(\mathsf{lfp}(f)) \wedge \forall x.\, x \approx f(x) \to \mathsf{lfp}(f) \sqsubseteq x, \\
\mathsf{gfp}(f) \approx f(\mathsf{gfp}(f)) \wedge \forall x.\, x \approx f(x) \to x \sqsubseteq \mathsf{gfp}(f).
\end{aligned} \qquad \text{(Knaster-Tarski [Tar55])}$$

From the definitions and equations above, the following proof principles can be derived:

$$\text{IND}\;\frac{f(x) \sqsubseteq x}{\mathsf{lfp}(f) \sqsubseteq x}, \qquad\qquad \text{COIND}\;\frac{x \sqsubseteq f(x)}{x \sqsubseteq \mathsf{gfp}(f)}.$$

It is straightforward to prove that $_\sqcap_$ (resp. $_\sqcup_$) and lfp (resp. gfp) are monotone. Also, the composition of monotone functions is again monotone.

Semantics definition

In order to give the denotational semantics of the coalgebraic μ-calculus, we require that we are given a predicate lifting map $[\![\circ]\!] : \forall X.\, \Lambda(\wp(X)) \to \wp(T(X))$ such that for any $f : Y \to X$:

$$\begin{array}{ccc}
\Lambda(\wp(Y)) & \xrightarrow{\;[\![\circ]\!]_Y\;} & \wp(T(Y)) \\
\Big\uparrow{\scriptstyle \Lambda(f^{-1})} & & \Big\uparrow{\scriptstyle T(f)^{-1}} \\
\Lambda(\wp(X)) & \xrightarrow[\;[\![\circ]\!]_X\;]{} & \wp(T(X))
\end{array}$$

If we have a function symbol $\lambda^{(k)} \in \Lambda$ of arity k, the naturality condition expressed above corresponds to:

$$\forall (A_1, \ldots, A_k : \wp(X)).\, T(f)^{-1}([\![\circ]\!]^{\lambda}_X(A_1, \ldots, A_k)) = [\![\circ]\!]^{\lambda}_Y(f^{-1}(A_1), \ldots, f^{-1}(A_k)).$$

The other map, namely $[\![\bullet]\!]_X$ is derived from $[\![\circ]\!]_X$ by duality as follows:

$$\begin{aligned}
&[\![\bullet]\!] : \forall X.\, \Lambda(\wp(X)) \to \wp(T(X)) \\
&[\![\bullet]\!]_X \;\equiv\; \complement \circ [\![\circ]\!] \circ \Lambda(\complement).
\end{aligned}$$

The denotational semantics of the coalgebraic μ-calculus is thus computed as follows:

$$\begin{aligned}
&[\![_]\!]_{_} : \forall \{n : \mathbb{N}\}.\; \textsc{Term}\, n \to \textsc{Env}_X(n) \to \wp(X) \\
&[\![\;\mathsf{var}\, x\;]\!]_\rho \;\equiv\; \rho(x) \\
&[\![\;\mu\Phi\;]\!]_\rho \;\equiv\; \mathsf{lfp}(\lambda(A : \wp(X)).\, [\![\Phi]\!]_{(A :: \rho)}) \\
&[\![\;\nu\Phi\;]\!]_\rho \;\equiv\; \mathsf{gfp}(\lambda(A : \wp(X)).\, [\![\Phi]\!]_{(A :: \rho)}) \\
&[\![\;\Phi_1 \wedge \Phi_2\;]\!]_\rho \;\equiv\; [\![\Phi_1]\!]_\rho \sqcap [\![\Phi_2]\!]_\rho \\
&[\![\;\Phi_1 \vee \Phi_2\;]\!]_\rho \;\equiv\; [\![\Phi_1]\!]_\rho \sqcup [\![\Phi_2]\!]_\rho \\
&[\![\;\circ\, \Phi\;]\!]_\rho \;\equiv\; \gamma^{-1}([\![\circ]\!](\Lambda\; [\![_]\!]_\rho\; \Phi)) \\
&[\![\;\bullet\, \Phi\;]\!]_\rho \;\equiv\; \gamma^{-1}([\![\bullet]\!](\Lambda\; [\![_]\!]_\rho\; \Phi))
\end{aligned}$$

where the type of environments is defined as $\textsc{Env}_X(n) := \textsc{Vec}\ (\wp(X))\ n$. We define the following order relation on environments:

$$\begin{aligned}&\forall (n : \mathbb{N}).\ \forall (p, \rho' : \textsc{Env}_X(n)).\\ &\qquad \rho \sqsubseteq \rho' \overset{\text{def}}{\iff} \forall (x : \textsc{Fin}\, n).\ \rho(x) \sqsubseteq_{\wp(X)} \rho'(x).\end{aligned}$$

Moreover, we lift an order $(A, \sqsubseteq_A)$ to $\Lambda(A)$ as follows:

$$\forall (a, a' : \Lambda(A)).\ a \sqsubseteq a' \overset{\text{def}}{\iff} a\ (\Lambda^{\forall_2} \sqsubseteq_A)\ a'.$$

Definition 1 **Satisfiability**

Let n be a natural number, Φ be a formula with at most n variables and ρ be an environment. Moreover, let $s : X$ denote a state in the T-coalgebra. We say that the formula Φ is satisfied at state s for ρ, written $s \vDash_\rho \Phi$, when we have $s \in [\![\Phi]\!]_\rho$. In order to make explicit the underlying T-coalgebra over which formulae are interpreted, we may write $(X, \gamma), s \vDash_\rho \Phi$.

In order to justify the existence of fixpoints, we have to prove that the semantics is monotone.

Proposition 1 $[\![-]\!]$ **is monotone**

Let n be a natural number and let t be a term with at most n free variables. If the map $[\![\circ]\!]$ is monotone, then so is the semantic map:

$$\forall (\rho, \rho' : \textsc{Env}_X(n)).\ \rho \sqsubseteq \rho' \to [\![t]\!]_\rho \sqsubseteq [\![t]\!]_{\rho'}.$$

Proof. Let $\rho, \rho' : \textsc{Env}_X(n)$ be two environments and $H : \rho \sqsubseteq \rho'$. We have to show that $[\![t]\!]_\rho \sqsubseteq [\![t]\!]_{\rho'}$, *i.e.*, $\forall (x : X).\ x \in [\![t]\!]_\rho \to x \in [\![t]\!]_{\rho'}$. We proceed by induction on the term t. We consider only the case for var, μ, $\wedge$, and $\circ$ since the others follow by duality.

- **Base step.** Assume that $t = \mathsf{var}\, x$ where $x : \textsc{Fin}\, n$.

  Immediate by assumption H.

- **Inductive step.** Assume that $t = \mu\Phi$ where Φ.

  The induction hypothesis is given by:

  $$\textsc{monotone}[\![\Phi]\!] \tag{IH}$$

  We have to prove that $[\![\mu\Phi]\!]$ is monotone, *i.e.*, $[\![\mu]\!](\lambda A.\ [\![\Phi]\!]_{A :: \rho}) \sqsubseteq [\![\mu]\!](\lambda A.\ [\![\Phi]\!]_{A :: \rho'})$. Since lfp is order-preserving, it suffices to show that $\forall A, A'.\ A \sqsubseteq A' \to [\![\Phi]\!]_{A :: \rho} \sqsubseteq [\![\Phi]\!]_{A' :: \rho'}$. Assume that $H' : A \sqsubseteq A'$. We use the induction hypothesis (IH), thus it remains to show that $A :: \rho \sqsubseteq A' :: \rho'$. This follows directly from H and H'.

- **Inductive step.** Assume that $t = \Phi_1 \wedge \Phi_2$ where $\Phi_1, \Phi_2 : \textsc{Term}\, n$.

  The induction hypothesis is given by:

  $$\textsc{monotone}[\![\Phi_1]\!] \wedge \textsc{monotone}[\![\Phi_2]\!]. \tag{IH}$$

  The map $_\sqcap_ : \wp(X) \to \wp(X) \to \wp(X)$ is monotone on both of its arguments. Thus, we conclude with the induction hypothesis (IH).

▸ **Inductive step.** Assume that $t = \circ\,\Phi$ where $\Phi : \Lambda(\textsc{Term}\,n)$.
The induction hypothesis is given by:

$$\Lambda^{\forall}\ (\textsc{monotone} \circ [\![_]\!])\ \Phi. \tag{IH}$$

The pre-image function $_^{-1} : (X \longrightarrow Y) \to \wp(Y) \longrightarrow \wp(X)$ is also clearly monotone. By assumption, so is $[\![\circ]\!]$. Finally, we conclude with the induction hypothesis (IH).

▸ **Inductive step.** Assume that $t = \bullet\,\Phi$ where $\Phi : \Lambda(\textsc{Term}\,n)$.
It suffices to show that the derived map is monotone. By definition, we have $[\![\bullet]\!]_X \equiv \complement \circ ([\![\circ]\!]_X \circ \Lambda(\complement))$. The map $\complement$ is anti-monotone thus $\Lambda(\complement)$ is also anti-monotone. Moreover, the composition of a monotone map with an anti-monotone map yields an anti-monotone map. As a result, the map $[\![\circ]\!]_X \circ \Lambda(\complement)$ is anti-monotone. Finally, the composition of two anti-monotone map yields a monotone map. Consequently, $[\![\bullet]\!]_X$ is monotone. □

As a sanity check, we verify that the derived operators have the intended semantics:

Proposition 2 **Semantics of derived operators**

Let n be a natural number, t be a term with at most n variables and ρ be an environment with n elements. Then, we have:

(*i*) $[\![\top]\!]_\rho \approx \mathsf{univ}$,
(*ii*) $[\![\bot]\!]_\rho \approx \varnothing$,
(*iii*) *if we assume* LEM, $[\![\neg\Phi]\!]_\rho \approx \neg[\![\Phi]\!]_{\neg\rho}$ *where* $\neg\rho := \mathsf{map}^{\textsc{Vec}}\ (\neg_)\ \rho$.

Proof. Let $n : \mathbb{N}$, $t : \textsc{Term}\,n$ and $\rho : \textsc{Env}_X(n)$.

(*i*) We have $[\![\top]\!]_\rho \equiv \mathsf{gfp}\,\mathsf{id}$. By antisymmetry, we have that $\mathsf{gfp}\,\mathsf{id} \sqsubseteq \mathsf{univ}$. It remains to show that $\mathsf{univ} \sqsubseteq \mathsf{gfp}\,\mathsf{id}$. By COIND, it suffices to show that $\mathsf{univ} \sqsubseteq \mathsf{id}(\mathsf{univ})$ which holds by reflexivity.
(*ii*) dual of (*i*).
(*iii*) Straightforward by induction on Φ. □

Decidability

In this section, we show that restricted to finite T-coalgebras, *i.e.*, coalgebras such that the carrier is finite, satisfiability is decidable. For the remaining of this section, we fix a *finite* T-coalgebra $(X, \gamma : X \to T(X))$.

Decidability on $\wp(X)$. We say that a morphism $f : \wp(X) \to \wp(X)$ preserves decidability when:

$$\textsc{Decidable}^{\textsc{fun}}(f) := \forall (A : \wp(X)).\ \textsc{Decidable}(A) \to \textsc{Decidable}(f(A)).$$

Here, $\textsc{Decidable}(A)$ is equivalent to say that membership in A is decidable. It is straightforward to check that $_^{-1}$, $_\sqcup_$, $_\sqcap_$ and $\complement_$ preserve decidability. Consequently, it remains to show how to compute least and greatest fixpoints.

Computation of fixpoints. Let $(A, \sqsubseteq_A)$ be a partial order with a minimal element $\bot$. Moreover, assume that A is streamless and let $f : A \to A$ be a monotone self-map.

[**Existence.**] Define the ω-chain $c_n := f^n(\bot)$. Clearly, the chain $c : \mathbb{N} \to A$ is monotone:

$$\bot \sqsubseteq f(\bot) \sqsubseteq \cdots \sqsubseteq f^n(\bot) \sqsubseteq \cdots.$$

Moreover, since A is streamless, there exist two positions $i, j : \mathbb{N}$ such that $i < j$ and $c_i = c_j$. By a straightforward induction on $i < j$ and by exploiting the antisymmetry of $\sqsubseteq_A$, we can show that $c_i = c_{i+1}$, *i.e.*, $f^i(\bot) = f^{i+1}(\bot) = f(f^i(\bot))$.

[Least fixpoint.] Assume that x is a fixpoint of f, *i.e.*, $x = f(x)$. Clearly, we have $\bot \sqsubseteq x$ and since f is monotone, we can conclude that $f(\bot) \sqsubseteq f(x)$. By iterating i times, we have $f^i(\bot) \sqsubseteq f^i(x)$, thus $f^i(\bot) \sqsubseteq x$ since x is a fixpoint of f.

We have proved that on streamless types with a minimal element, any monotone self-map on A has a least fixpoint which is computed by iterating f, starting from the minimal element. The construction above can be generalized to setoids by requiring that the function f is a setoid morphism. Now, we explain how this result can be extended to the power set functor $\wp(-)$. Let X be a type. Moreover, assume that X is finite (at least weakly-finitely indexed) and let $f : \wp(X) \to \wp(X)$ be a map such that f is monotone and preserves decidability. Since X is finite so is 2^X. A decidable predicate $P : \wp(X)$ induces a map on 2^X defined as follows:

$$
\begin{array}{l}
_^? : (P : \wp(X)) \to \{\text{DECIDABLE}\, P\} \to 2^X \\
P^? \;\; a \;\equiv\; \textbf{match}\ a \in^? P\ \textbf{with} \\
\qquad\qquad \mid \mathsf{yes}\,_ \Rightarrow \mathsf{true} \\
\qquad\qquad \mid \mathsf{no}\,_ \;\Rightarrow \mathsf{false} \\
\qquad\quad \textbf{end}
\end{array}
$$

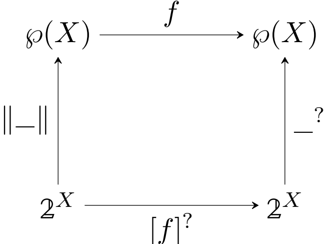
Similarly, every map 2^X induces a map on $\wp(X)$ defined as follows:

$$
\begin{array}{l}
\|_\| : 2^X \to \wp(X) \\
\|p\| \;\equiv\; \{\, x : p(x) = \mathsf{true} \,\}.
\end{array}
$$

Clearly, we have that both maps are inverse of one another:

- $\forall (p : 2^X).\ \forall x.\ \|p\|^?(x) = p(x)$,
- $\forall (P : \wp(X)).\ \text{DECIDABLE}(P) \to \|P^?\| \approx P$.

A map $f : \wp(X) \to \wp(X)$ preserving decidability induces a map $[f]^? : 2^X \to 2^X$ such that the following diagram commutes:



Then, by iterating this commutative square, we obtain the following diagram:

$$
\begin{array}{ccccccccccccc}
\wp(X) & \xrightarrow{\mathsf{const}\ \varnothing} & \wp(X) & \xrightarrow{f} & \wp(X) & \xrightarrow{f} & \wp(X) & \cdots\to & \wp(X) & \xrightarrow{f} & \wp(X) & \cdots\to \\
\uparrow{\scriptstyle \|_\|} & & \uparrow{\scriptstyle \|_\|} & & \uparrow{\scriptstyle \|_\|} & & \uparrow{\scriptstyle \|_\|} & & \uparrow{\scriptstyle \|_\|} & & \uparrow{\scriptstyle \|_\|} & \\
2^X & \xrightarrow[{[\mathsf{const}\ \varnothing]^?}]{} & 2^X & \xrightarrow[{[f]^?}]{} & 2^X & \xrightarrow[{[f]^?}]{} & 2^X & \cdots\to & 2^X & \xrightarrow[{[f]^?}]{} & 2^X & \cdots\to
\end{array}
$$

We can use the fixpoint computation defined previously, on the bottom on the diagram. Indeed, the type 2^X is finite, $[\mathsf{const}\,\varnothing]^? \approx \mathsf{const\,false}$ is a map to the minimal element of 2^X and $[f]^?$ is monotone. Thus, there exists a number i, such that $\mathsf{lfp}[f]^? \approx ([f]^?)^i(\mathsf{const\,false})$. Furthermore, since all squares commute, we have $\|\,\mathsf{lfp}[f]^?\| \approx \|([f]^?)^i(\mathsf{const\,false})\| \approx f^i(\varnothing) \approx \mathsf{lfp}\, f$. Finally, the map $\|_\|$ transports decidability, thus membership is decidable in $\mathsf{lfp}\, f$.

Theorem 1 **Decidability of satisfiability**

Let n be a natural number, Φ be a term with at most n free variables and ρ be an environment. Then, we have

$$\textsc{Vec}^{\forall}\ \textsc{Decidable}\ \rho \to \textsc{Decidable}(\llbracket \Phi \rrbracket_{\rho}).$$

Proof. Let $n : \mathbb{N}$, $\Phi : \textsc{Term}\, n$, $\rho : \textsc{Env}_X(n)$ and $H : \textsc{Vec}^{\forall}\ \textsc{Decidable}\ \rho$. We proceed by induction on Φ.

- **Base step.** Assume that $t = \mathsf{var}\, x$ where $x : \textsc{Fin}\, n$.
  We have $\textsc{Decidable}(\llbracket \mathsf{var}\, x \rrbracket_{\rho}) \equiv \textsc{Decidable}(\rho(x))$ which is a consequence of H.
- **Inductive step.** Assume that $t = \mu\Phi$ where $\Phi : \textsc{Term}(\mathsf{suc}\, n)$.
  The induction hypothesis is given by:

  $$\forall(\rho : \textsc{Env}_X(\mathsf{suc}\, n)).\ \textsc{Vec}^{\forall}\ \textsc{Decidable}\ \rho \to \textsc{Decidable}(\llbracket \Phi \rrbracket_{\rho}). \quad \text{(IH)}$$

  By $\textsc{Dec}$-lfp to prove $\textsc{Decidable}(\mathsf{lfp}(\lambda A.\ \llbracket \Phi \rrbracket_{(A :: \rho)}))$, it suffices to show that $(\lambda A.\ \llbracket \Phi \rrbracket_{(A :: \rho)})$ preserves decidability. Let $A : \wp(X)$ be such that membership is decidable. To prove $\textsc{Decidable}(\llbracket \Phi \rrbracket_{(A :: \rho)})$, we use the induction hypothesis (IH). Thus, it remains to show that $\textsc{Decidable}(\textsc{Vec}^{\forall}\ \textsc{Decidable}\ \rho)$. By rule $\textsc{Dec}$-$\textsc{Vec}^{\forall}$, it suffices to show that all elements of the ρ are decidable which is true by assumption.
- **Inductive step.** Assume that $t = \Phi_1 \sqcap \Phi_2$ where $\Phi_1, \Phi_2 : \textsc{Term}\, n$.
  The induction hypothesis is given by:

  $$\forall(\rho : \textsc{Env}_X(n)).\ \textsc{Vec}^{\forall}\ \textsc{Decidable}\ \rho \to \textsc{Decidable}(\llbracket \Phi_1 \rrbracket_{\rho}) \wedge \textsc{Decidable}(\llbracket \Phi_2 \rrbracket_{\rho}). \quad \text{(IH)}$$

  This case is trivial with rule $\textsc{Dec}$-$\sqcap$.
- **Inductive step.** Assume that $t = \circ\,\Phi$ where $\Phi : \Lambda(\textsc{Term}\, n)$.
  The induction hypothesis is given by:

  $$\forall(\rho : \textsc{Env}_X(n)).\ \textsc{Vec}^{\forall}\ \textsc{Decidable}\ \rho \to \Lambda^{\forall}\ \textsc{Decidable}\ (\Lambda\ \llbracket - \rrbracket_{\rho}\ \Phi). \quad \text{(IH)}$$

  We have to show $\textsc{Decidable}(\gamma^{-1}(\llbracket \circ \rrbracket(\Lambda\ \llbracket - \rrbracket_{\rho}\ \Phi)))$. By rule $\textsc{Dec}$-$^{-1}$ and assumption I, it suffices to prove that $\Lambda^{\forall}\ \textsc{Decidable}\ (\Lambda\ \llbracket - \rrbracket_{\rho}\ \Phi)$ which holds by (IH).
- **Inductive step.** Assume that $t = \bullet\,\Phi$ where $\Phi : \Lambda(\textsc{Term}\, n)$.
  It suffices to show that the derived map $\llbracket \bullet \rrbracket$ preserves decidability. Let $t : \Lambda(\wp(X))$ such that $\Lambda^{\forall}\ \textsc{Decidable}\ t$. By definition, we have that $\llbracket \bullet \rrbracket_X \equiv \complement \circ (\llbracket \circ \rrbracket_X \circ \Lambda(\complement))$. We have that $\complement$ preserves decidability ($\textsc{Dec}$-$\complement$). By assumption, $\llbracket \circ \rrbracket_X$ preserve decidability if $\Lambda^{\forall}\ \textsc{Decidable}\ (\Lambda\ \complement\ t)$. By rule $\textsc{Dec}$-Λ-map, it suffices to show that $\Lambda^{\forall}\ \textsc{Decidable}\ t$ which holds by assumption, and that $\complement\,_$ preserves decidability which is given by $\textsc{Dec}$-$\complement$. □

4.2.3. Decidability of Bisimilarity

In this section, we show that bisimilarity is decidable over the type of regular trees. For the remaining of this section, we fix a signature $S : \textsc{Sig}$. In addition, we assume that the set of operators, namely $S.\mathsf{Op}$, has a *decidable Leibniz equality*. Let us recall that bisimilarity is defined as coinductively as follows:

```
coinductive _∼_ : coT_S → coT_S → Prop
  | ∼-intro : ∀{t₁, t₂}. (e : root t₁ = root t₂) → (∀k. t₁(k) ∼ t₂(e_*(k))) → t₁ ∼ t₂.
```

$$\text{Dec-}\varnothing\ \frac{}{\textsc{Decidable}\,\varnothing} \qquad \text{Dec-univ}\ \frac{}{\textsc{Decidable}\,\mathsf{univ}} \qquad \text{Dec-}^{-1}\ \frac{f : X \to Y}{\textsc{Decidable}^{\textsc{fun}}(f^{-1})}$$

$$\text{Dec-}\sqcup\ \frac{\textsc{Decidable}\,P \qquad \textsc{Decidable}\,Q}{\textsc{Decidable}(P \sqcup Q)} \qquad \text{Dec-}\sqcap\ \frac{\textsc{Decidable}\,P \qquad \textsc{Decidable}\,Q}{\textsc{Decidable}(P \sqcap Q)}$$

$$\text{Dec-}\Lambda^{\forall}\ \frac{t : \Lambda(\wp(X)) \qquad \textsc{Vec}^{\forall}\ \textsc{Decidable}\ (\mathsf{proj}_2\, t)}{\Lambda^{\forall}\ \textsc{Decidable}\ t} \qquad \text{Dec-}\textsc{Vec}^{\forall}\ \frac{\forall x.\ \textsc{Decidable}(\rho(x))}{\textsc{Vec}^{\forall}\ \textsc{Decidable}\ \rho}$$

$$\text{Dec-}\Lambda\text{-map}\ \frac{t : \Lambda(\wp(X)) \qquad \Lambda^{\forall}\ \textsc{Decidable}\ t \qquad \textsc{Decidable}^{\textsc{fun}}\ f}{\Lambda^{\forall}\ \textsc{Decidable}\ (\Lambda\ f\ t)}$$

$$\text{Dec-}\complement\ \frac{\textsc{Decidable}\ P}{\textsc{Decidable}(\complement P)} \qquad \text{Dec-app}\ \frac{\textsc{Decidable}^{\textsc{fun}}\ f \qquad \textsc{Decidable}\ P}{\textsc{Decidable}(f(P))}$$

$$\text{Dec-lfp}\ \frac{A \text{ finite} \qquad f : \wp(A) \to \wp(A) \qquad f \text{ monotone} \qquad \textsc{Decidable}^{\textsc{fun}}\ f}{\textsc{Decidable}(\mathsf{lfp}\ f)}$$

$$\text{Dec-gfp}\ \frac{A \text{ finite} \qquad f : \wp(A) \to \wp(A) \qquad f \text{ monotone} \qquad \textsc{Decidable}^{\textsc{fun}}\ f}{\textsc{Decidable}(\mathsf{gfp}\ f)}$$

Figure 4.1. Decidability proof rules

In order to prove that the type $(\forall (t, s : \mathrm{co}\mathbb{T}_S).\ \textsc{Dec}(t \sim s))$ is inhabited, we show that the bisimilarity relation can be encoded as a μ-calculus formula. To this end, we consider the following signature which extends S with a nullary function symbol:

```
S⊥ : Sig
S⊥  ≡  Maybe S.Op ◁ ar
   where  ar : Maybe(S.Op) → ℕ
          ar  nothing   ≡  zero
          ar  (just o)  ≡  S. Ar o.
```

Now, we compute a synchronous product of infinite trees. The product is given by the following function defined corecursively:

```
_⊗_ : coT_S → coT_S → coT_S⊥
t ⊗ s  ≡  match root t =? root s with
          | yes e  ⇒  just(root t) ◀ λk. t(k) ⊗ s(e_*(k))
          | no _   ⇒  nothing ◁ ⊥-elim
          end
```

Intuitively, given two infinite trees, we compare their roots. If they are the same, we repeat this process on all of their subtrees. Note that, when the roots are equal, we are given a proof e witnessing this equality. Then, this proof is used to "cast" an index of t in an index of s. As soon as the roots differ, the computation halts. This indicates that no further synchronization is possible. Clearly, the definition of $_\otimes_$ can be expressed as a 2-ary top-down tree transducer. Consequently, the function $_\otimes_$ preserves the regularity of trees. Now, we can think of bisimilarity as a temporal logic property:

"two trees are bisimilar if and only they can always be synchronized through $\otimes$".

Now, we define the signature of temporal operators that we need to express this statement:

```
Λ : Sig
Λ ≡ Λ_op ◁ Λ_ar
    where inductive Λ_op : Type
                    pred : (S⊥.Op → 𝔹) → Λ_op
                       □ : Λ_op
          Λ_ar : op → ℕ
          Λ_ar □        ≡ 1
          Λ_ar (pred _) ≡ 0,

⟦ pred p ⟧ : ℘(S⊥(X))
⟦ pred p ⟧ ≡ { s : p(proj₁(s)) = true },

⟦ □ ⟧ : ℘(X) → ℘(S⊥(X))
⟦ □ ⟧ R ≡ { s : S⊥^∀ (λs′. s′ ∈ R) s }.
```

Intuitively, given a formula Φ, the formula $\square\,\Phi$ means that Φ should hold on all successor states. The other nullary operator pred is used to express predicates over states. The semantics of these operators can be expressed on an arbitrary $S_\bot$-coalgebra and is given above.

Clearly, we have that $[\![\,\square\,]\!]$ is monotone and preserves decidability. The same holds for $[\![\,\mathsf{pred} -]\!]$. Moreover, given a $S_\bot$-coalgebra (X, γ), a formula $\Phi : \textsc{Term}\, n$ and an environment $\rho : \textsc{Env}_X(n)$, it is straightforward to check that:

$$s \vDash_\rho \square\,\Phi \leftrightarrow S_\bot^\forall\ (\lambda s'.\ s' \vDash_\rho \Phi)\ (\gamma(s)),$$

and

$$s \vDash_\rho \mathsf{pred}\, p \leftrightarrow p(\mathsf{proj}_1(\gamma(s))) = \mathsf{true}.$$

Let us recall that for any signature S and regular tree $\chi : \textsc{Reg}_S$, the type of successors of $\chi\!\rightsquigarrow$ has a S-coalgebra structure inherited from $\textsc{co}\mathbb{T}_S$. Now, we interpret the formula of the coalgebraic μ-calculus on this successor-coalgebra, *i.e.*, on $t\!\rightsquigarrow$ for a given $t : \textsc{Reg}_{S_\bot}$. In addition, we define the formula OK as follows:

```
OK : Term zero
OK ≡ pred SYNC?
    where SYNC? : Maybe S.Op → 𝔹
          SYNC? nothing  ≡ false
          SYNC? (just _) ≡ true.
```

Lemma 1 Bisimilarity as a μ-formula

Let $t, s : \textsc{Reg}_S$ be two regular trees. We have

$$t \sim s \leftrightarrow (t \otimes s, \varepsilon) \vDash \nu X.\, \textsc{ok} \wedge \square\, X.$$

Proof. Let $t, s : \textsc{Reg}_S$.

($\Longrightarrow$) Assume that $E : t \sim s$.

We have to show that $(t\otimes s, \varepsilon) \vDash \nu X.\, \textsc{ok}\wedge\square\, X$. By rule COIND, it suffices to find a post fixpoint r of $\lambda X.\, \textsc{ok}\wedge\square\, X$ such that $(t\otimes s, \varepsilon) \in r$. We pick $r := \{\, u : \exists t.\ \exists s.\ \exists p.\ t \sim s \wedge u \approx (t\otimes s, p)\,\}$. Clearly, we have $(t \otimes s) \in r$. It remains to show that $(t \otimes s, \varepsilon) \vDash_{[r]} \textsc{ok} \wedge \square(\mathsf{var\,zero})$. To prove a conjunction, it suffices to show:

- $(t \otimes s, \varepsilon) \vDash_{[r]} \textsc{ok}$:
  this is equivalent to the following proposition $\textsc{sync?}(\mathsf{root}(t \otimes s)) = \mathsf{true}$. Since we have $t \sim s$ by assumption, t and s have equal roots. Consequently, we can prove that necessarily $\textsc{sync?}(\mathsf{root}(t \otimes s)) = \mathsf{true}$.

- $(t \otimes s, \varepsilon) \vDash_{[r]} \Box(\mathsf{var\,zero})$:
  By elimination of $E : t \sim s$, we obtain a proof that E_r : $\mathsf{root}\, t = \mathsf{root}\, s$. With this proof, we can show that the goal if the same as $\forall k.\ (t(k) \otimes s(k), k :: \varepsilon) \vDash_{[r]} \mathsf{var\,zero}$ which amounts to show that $\forall k.\ (t(k) \otimes s(k), k :: \varepsilon) \in r$ which is clearly true.

($\Longleftarrow$) Assume that $(t \otimes s, \varepsilon) \vDash \nu X.\, \text{OK} \wedge \Box X$.
By corecursion, we assume

$$\forall t.\ \forall s.\ \forall p.\ (t \otimes s, p) \vDash \nu X.\, \text{OK} \wedge \Box X \to t \sim s. \qquad \text{(cofix)}$$

Clearly, since the semantics of ν is a greatest fixpoint, we can unfold it one time. Thus we can deduce that $(t \otimes s, \varepsilon) \vDash \text{OK} \wedge \Box(\nu X.\, \text{OK} \wedge \Box X)$. By elimination of the conjunction, we can deduce that necessarily the root of t and s are synchronizable and thus equal. We call E_r : $\mathsf{root}\, t = \mathsf{root}\, s$ this proof of equality. Furthermore, we have that $H : \forall k.\ (t(k) \otimes s(e_* k), k :: \varepsilon) \vDash (\nu X.\, \text{OK} \wedge \Box X)$. Finally, we can use the constructor $\sim$-intro with the proof E_r. It remains to show that $\forall k.\ t(k) \sim t(e_* k)$. We conclude by applying H to (cofix). □

Theorem 2 Decidability of bisimilarity

Bisimilarity is decidable for regular trees.

Proof. Let t, s : Reg_S be two regular trees. By Lemma 4.1, we have that $t \sim s$ is equivalent to a μ-formula. Moreover, by Theorem 4.1 satisfiability for this formula is decidable since the carrier of the $S_\perp$-coalgebra is $(t \otimes s)\rightsquigarrow$ which is weakly finitely indexed. Indeed, the tree product $t \otimes s$ is regular. Consequently, in order to prove that $\text{Dec}(t \sim s)$, it suffices to show that it is equivalent to a proposition that is decidable which is indeed the case. □

4.3. CONCLUSION

In this chapter, we considered an application of tree transducers to highlight how a termination problem occurring in the definition of the parallel composition operation of a process algebra with recursive processes, can be recast into a productivity problem. In particular, this example illustrated how functions on cyclic terms can be derived from functions on infinite trees. Moreover, we considered decidability problems on regular trees. To this end, we developed an interpretation of a coalgebraic μ-calculus [CKP11] over regular trees that is parametric over the interpretation model and the syntax. Several formalizations or mechanizations of various modal logics in the context of proof assistants exists in the literature. For the Coq proof assistants, [TW06] presents a formalization of the branching temporal logic CTL*, [Spr98] contains a mechanization of a model-checker for the propositional modal μ-calculus. The last one is close to our work, though their mechanization in not parametric over the type of models not over the syntax of modal operators. However, it should be emphasized that our initial goal with the interpretation of μ-calculus on regular trees was not to obtain an efficient implementation of a model-checker but merely to be used as a tool to derive decidability results. In particular, we showed that bisimilarity can be interpreted as a μ-calculus formula. One possible extension of our work would be to generalize further our mechanization by allowing the semantic domain to be also parametric. For instance, the semantic interpretation could be defined by requiring a contravariant functor from types to complete boolean algebras instead of the power set functor. In this case, decidability of satisfiability would follow from the ability to define a complete boolean algebra such that all operations (join, greatest upper bounds, ...) on it are computable. With this extension, it should be possible to reuse the work on binary decision diagrams (BDDs) of [BJM13, VGPA00] or [GLSG15] in the context of finite trees, to obtain more efficient implementations.

CONCLUSION

In this thesis, we have presented a mechanization of the theory of regular trees in dependent type theory. Regular trees are infinite trees characterized by a finiteness property, namely such that the set of their distinct subtrees is finite. In this work, we have considered two main problems. Firstly, we considered the problem of characterizing regular trees as a type. To this end, we addressed regular trees semantically by means of coinductive types—a natural framework to reason about infinite objects. Then, we defined a syntax of regular trees and proved its soundness with respect to the semantic characterization. Secondly, we considered the problem of defining tree transformations preserving the regularity property. Throughout our work, one of our main concerns was to use sufficiently abstract data structures together with as weak as possible hypotheses while remaining in a constructive setting. For instance, we worked in a type theory without assuming LEM (Law of Excluded-Middle), proof-irrelevance, the extensionality principle on functions or UIP (Uniqueness of Identity Proofs). The remaining paragraphs highlight our related contributions.

The first contribution of this thesis is a study of the problem of *characterizing finite types in type theory*. In a constructive setting, several notions of finiteness exist. This problem was refined by considering setoids—types equipped with an equivalence relation—in order to enable reasoning on coinductive types modulo bisimilarity. We compared several notions of finite types which have been classified into three categories: enumerable, weakly finitely indexed, streamless. Weakly finitely indexed setoids were of particular interest since they give a characterization of finite setoids where the underlying equivalence relation is not required to be decidable.

The second contribution of this thesis is *a semantic and syntactic characterization of the type of regular trees* over an arbitrary signature S. First, we specified the type of regular trees as a restriction of infinite trees such that the set of its subtrees is weakly finitely indexed—up-to bisimilarity. In particular, we did not assume that the equality on the function symbols given by the signature is decidable. In addition, we proved that the type of regular trees has an S-coalgebra structure. Next, we gave a syntactic characterization of the type of regular trees by means of cyclic terms where cycles in the term are encoded through binding structures. We showed that cyclic terms have a semantic interpretation as regular trees by unfolding the cycles infinitely. Furthermore, we also considered the converse problem, that is, *computing* a cyclic term representation of a regular tree.

The third contribution of this thesis is a study of the problem of *defining regularity preserving tree morphisms*. To this end, we used the formalism of top-down tree transducers as a tool to model the abstract syntax of a subset of corecursive function definition. Essentially, tree transducers describe term rewriting systems. Then, tree transducers were reified into tree morphisms obtained by transforming an input tree according the specified rewrite rules. In addition, we proved that these induced tree morphisms preserve regularity. In this context, we studied three different variants of tree transducers with increasing syntactic power. The first variant is a straightforward

extension of classical top-down tree transducers on finite trees to infinite trees, called guarded top-down tree transducers. In this formalism, a restriction is imposed on the right-hand sides of rewrite rules to ensure productivity *by construction*. Then, we extended the formalism of guarded tree transducers with ε-rules, that is, rewrite rules which may not consume the input tree. These rewrite rules can thus be thought of as purely productive. The last variant we considered, named guarded top-down tree transducers with finite look-ahead, extends rewrite rules with the ability to observe a finite context before producing a value. Moreover, the formalism of ε-rules was generalized in the sense that rewrite rules may specify the part of the context that is actually consumed. Furthermore, we considered the problem of representing binary top-down tree transducer, *i.e.*, tree transformations operating on two trees simultaneously. To this end, we defined a lifting operation which consists in transforming a product of trees into a tree over a product of signatures. Then, binary tree transducers were obtained through unary tree transducers by a pre-processing phase applying this lifting operation. Finally, for every variant of tree transducers introduced, we proved that their induced tree morphisms preserve regularity.

The last contribution of this thesis is an *interpretation of a coalgebraic μ-calculus over the type of regular trees*. In particular, we proved the decidability of satisfiability. This decidability result was then used to show that bisimilarity on regular trees is decidable. In addition, we considered an application of tree transducers to highlight how a termination problem occurring in the definition of the parallel composition operation of a process algebra with recursive processes, can be recast into a productivity problem.

PERSPECTIVES

There are several ways to extend the work presented in this thesis. In the following, we mention some promising topics for further research.

Regular trees in Homotopy Type Theory. Homotopy Type Theory (HoTT), part of *Univalent Foundations* [Uni13], is an extension of Intensional Martin-Löf Type Theory [ML75]. The extension is given by the *Univalence Axiom*, which captures the idea of reasoning modulo isomorphism, an ubiquitous informal principle in mathematics. The types in HoTT correspond to ∞-groupoids, infinite-dimensional categorical structures. On the other hand, types as considered in traditional type theory correspond to 1-groupoids, that is, those ∞-groupoids that have a trivial higher structure. In that sense, the universe of discourse of type theory used in this thesis is a subuniverse of that of HoTT. In [ACS15], we show how to derive coinductive types (indexed M-types) over an arbitrary indexed signature. These coinductive types are characterized by a *universal property*. This construction relies on the function extensionality principle, a consequence of the univalence axiom. In particular, we prove that such coinductive types satisfy the coinduction proof principle [Rut00]:

$$s \,\mathcal{R}\, t \to s = t$$

where $_\mathcal{R}_$ denotes an arbitrary bisimulation relation and $_=_$ denotes the identity type. It could be interesting to generalize the work of this thesis to higher dimensional structures by passing to the formal framework of Univalent Foundations. The coinductive types derived in [ACS15] constitutes a first step towards this direction. Furthermore, it could be interesting to see whether the type of cyclic terms could be define by means of *Higher Inductive Types*[Uni13, LS13], a newly introduced device in HoTT. Higher inductive types are an extension of inductive types with path constructors. Through these path constructors, it should be possible to specify that a cyclic term ought to be identified with its unfolding.

Towards quotient of regular coinductive types. In Chapter 1, we explained that coinductive types, as implemented in COQ, correspond categorically only to *weakly* final coalgebras and are thus not characterized by a universal property. A main consequence is that the identity type is not the greatest bisimulation, *i.e.*, this inclusion $_\sim_ \subseteq _=_$ is not derivable. Another consequence is that bisimilarity is not a congruence and thus bisimilar terms cannot be substituted in an arbitrary context. If we restrict the coinductive types to the subset of decidable regular coinductive types, it should be possible to define a normalization map [Li15]:

$$\Downarrow: \mathrm{REG}_S \to \mathbb{C}_S$$

mapping regular trees to cyclic terms. A map with the same signature, namely $[\![_]\!]^{-1}$, was defined in Chapter 2 when we established the completeness of the syntactic representation of regular trees. However, what is missing is a minimization step. Such minimization problems over finite cyclic structures such as finite-state automata or regular expression have been studied in-depth in the literature and several (efficient) algorithms exist such as [Hop71, Brz62].

Regular coinductive types. Throughout all chapters, we developed a mechanized library of regular trees in COQ. What is still missing is a syntax within COQ to simplify its instantiation. For instance, we could extend the coinductive datatype definition with a new keyword **regular**. Consequently, one could define the type type of cyclic lists as follows:

regular coinductive LIST $(A : \mathrm{TYPE}) : \mathrm{TYPE}$
| $[\,] : \mathrm{LIST}\, A$
| $_::_ : A \to \mathrm{LIST}\, A \to \mathrm{LIST}\, A.$

From this definition a signature S could be generated and instantiated to the type of cyclic terms $\mathbb{C}_S$. Then, the **CoFixpoint** operator of COQ could be extended to deal with regular coinductive types. Intuitively, from a corecursive function definition from a finite LIST-coalgebra, we extract the corresponding top-down tree transducer T and then use its induced morphism $(\!| T |\!) : X \to \mathrm{LIST}(A)$ which preserves regularity. Such an approach was used in in [JKS12], where the authors present an extension of OCAML called COCAML aiming to add support for programming with regular trees. Their approach consists in extending recursive function definitions with solver annotations. Then, a set of equations is generated, following the definition of the recursive function, and is solved through the annotated solver. For instance, they consider the problem of producing inductive values by consuming regular trees as well as the problem of building regularity preserving functions. Our approach is slightly different in the sense that we define specific—syntactically-constrained—recursive schemes by means of tree transducers which are then solved up-front.

Tree transducers as regular trees. Tree transducers, as described in Chapter 3, are an abstract syntax for (co)recursive function definitions. When restricted to the subset for finite-state tree transducers over finite signature, tree transducers can be described by a finite system of (guarded) equations whose unique solutions are regular trees. Consequently, it should be possible to represent such tree transducers by means of cyclic terms. Then, by reusing the decidability theory developed in Chapter 4, we could show that tree transducer equivalence is decidable, thus recovering known results [Sei90, Sei94, Man14]. Another interesting application of this cyclic term representation of tree transducers is the problem of productivity. Indeed, all the tree transducers introduced in this thesis required the assumption that every rewrite rule ought to be *immediately* productive.

Regular trees over many-sorted signatures. As shown in [AGH+15], it possible to derive indexed W-types (IW-types) from their non-indexed counterparts, namely W-types [MLS84]. Here, W-trees correspond to the term algebra induced by a signature while IW-trees are the indexed term algebra induced by many-sorted signatures. Intuitively, this derivation is obtained starting from a type IW^{PRE} of pre-IW-trees where indices have been internalized through sigma-types. Then, a typing relation $\tau : W^{\text{PRE}} \to \textsc{Type}$ is defined over such pre-terms. Finally, the type of IW-trees is defined as the subset type of well-typed pre-terms:

$$IW := \sum_{t:IW^{\text{PRE}}} \tau(t).$$

As described in [AGH+15], such a construction can be reused to derive IM-types from M-types. It could be interesting to check whether such a construction could be employed to derive a notion of regular trees over many-sorted signatures, the question being whether typing constraints should be considered in the characterization of the regularity property. For instance, consider the type of $\mathbb{N}$-indexed streams over a base type A defined as follows:

coinductive $\textsc{Stream}^{\mathbb{N}}\ (A : \textsc{Type}) : \mathbb{N} \to \textsc{Type}$
$\quad|\ _{::}_ : \forall\{n : \mathbb{N}\}.\ A \to \textsc{Stream}^{\mathbb{N}}(\mathsf{suc}\, n) \to \textsc{Stream}^{\mathbb{N}}\, n.$

Intuitively, each element of the stream is indexed by its depth. Now, given an element $a : A$, we can define the constant stream a^{ω}:

$a^{\omega} : \textsc{Stream}^{\mathbb{N}}\ \mathsf{zero}$
$a^{\omega}\ \equiv\ a :: a^{\omega}.$

Clearly, if we forget the indices, we are left with a regular stream. However, if we consider the indexed, we have an infinity of substreams with a distinct type. As a step towards this direction, we could consider the work of [MW15] where they give a category-theoretic proof that the substitution operation over the infinitary lambda-calculus preserves regularity.

BIBLIOGRAPHY

[AAV01] Peter Aczel, Jiří Adámek, and Jiří Velebil. A coalgebraic view of infinite trees and iteration. *Electronic Notes in Theoretical Computer Science*, 44(1):1–26, 2001. 10.1016/S1571-0661(04)80900-9

[Abe07] Andreas Abel. *Type-based termination: a polymorphic lambda-calculus with sized higher-order types*. PhD thesis, Ludwig Maximilians University Munich, 2007.

[ACP+08] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL*, pages 3–15. ACM, 2008. 10.1145/1328438.1328443

[ACS15] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In *TLCA*, volume 38 of *LIPIcs*, pages 17–30. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2015. 10.4230/LIPIcs.TLCA.2015.17

[AGH+15] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015. 10.1017/S095679681500009X

[Ahr15] Benedikt Ahrens. Initiality for typed syntax and semantics. *Journal of Formalized Reasoning*, 8(2):1–155, 2015. 10.6092/issn.1972-5787/4712

[AM13] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. In *ICFP*, pages 197–208. ACM, 2013. 10.1145/2500365.2500597

[AR99] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. 10.1007/3-540-48168-0_32

[AS14] Benedikt Ahrens and Régis Spadotti. Terminal semantics for codata types in intensional Martin-Löf type theory. In *TYPES*, volume 39 of *LIPIcs*, pages 1–26. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2014. 10.4230/LIPIcs.TYPES.2014.1

[BC00] Gilles Barthe and Thierry Coquand. An introduction to dependent type theory. In *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 1–41. Springer, 2000. 10.1007/3-540-45699-6_1

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. 10.1007/978-3-662-07964-5

[BCP03] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003. 10.1017/S0956796802004501

[BH97] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *TLCA*, volume 1210 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 1997. 10.1007/3-540-62688-3_29

[BJM13] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. Implementing hash-consed structures in Coq. In *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 477–483. Springer, 2013. 10.1007/978-3-642-39634-2_36

[BK88] Jan A. Bergstra and Jan Willem Klop. Process theory based on bisimulation semantics. In *REX Workshop*, volume 354 of *Lecture Notes in Computer Science*, pages 50–122. Springer, 1988. 10.1007/BFb0013021

[Blo83] Stephen L. Bloom. All solutions of a system of recursion equations in infinite trees and other contraction theories. *Journal of Computer and System Sciences*, 27(2):225–255, 1983. 10.1016/0022-0000(83)90041-7

[BPT15] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Foundational extensible corecursion: a proof assistant perspective. In *ICFP*, pages 192–204. ACM, 2015. 10.1145/2784731.2784732

[Brz62] Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical theory of Automata*, 12(6):529–561, 1962.

[CDG+07] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2007. http://www.grappa.univ-lille3.fr/tata

[CDP14] Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without K. In *ICFP*, pages 257–268. ACM, 2014. 10.1145/2628136.2628139

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, 1988. 10.1016/0890-5401(88)90005-3

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types—A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. http://mitpress.mit.edu/books/certified-programming-dependent-types

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. 10.2307/2266170

[CKP11] Corina Cîrstea, Clemens Kupke, and Dirk Pattinson. EXPTIME tableaux for the coalgebraic μ-calculus. *Logical Methods in Computer Science*, 7(3), 2011. 10.2168/LMCS-7(3:3)2011

[Coh13] Cyril Cohen. Pragmatic quotient types in Coq. In *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2013. 10.1007/978-3-642-39634-2_17

[Coq93] Thierry Coquand. Infinite objects in type theory. In *TYPES*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 1993. 10.1007/3-540-58085-9_72

[Coq16] The Coq development team. *The Coq proof assistant reference manual*. INRIA, 2016. Version 8.5. http://coq.inria.fr

[Cou83] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983. 10.1016/0304-3975(83)90059-2

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. 10.1007/3-540-52335-9_47

[DA10] Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In *MPC*, volume 6120 of *Lecture Notes in Computer Science*, pages 100–118. Springer, 2010. 10.1007/978-3-642-13321-3_8

[Dan10] Nils Anders Danielsson. Beating the productivity checker using embedded languages. In *PAR*, volume 43 of *EPTCS*, pages 29–48, 2010. 10.4204/EPTCS.43.3

[dB72] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972. 10.1016/1385-7258(72)90034-0

[DN08] Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *IFL*, volume 5836 of *Lecture Notes in Computer Science*, pages 80–99. Springer, 2008. 10.1007/978-3-642-24452-0_5

[DP02] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[EHB13] Jörg Endrullis, Dimitri Hendriks, and Martin Bodin. Circular coinduction in Coq using bisimulation-up-to techniques. In *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 354–369. Springer, 2013. 10.1007/978-3-642-39634-2_26

[Eng77] Joost Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10:289–303, 1977. 10.1007/BF01683280

[FU15] Denis Firsov and Tarmo Uustalu. Dependently typed programming with finite sets. In *WGP*, pages 33–44. ACM, 2015. 10.1145/2808098.2808102

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-Directed Semantics. Formal Models Based on Tree Transducers.* Monographs in Theoretical Computer Science. Springer, 1998. 10.1007/978-3-642-72248-6

[GHUV06] Neil Ghani, Makoto Hamana, Tarmo Uustalu, and Varmo Vene. Representing cyclic structures as nested datatypes. In *Proceedings of 7th Trends in Functional Programming*, pages 173–188. Intellect, 2006. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.263

[Gim96] Eduardo Giménez. *A Calculus of Infinite Constructions and its applications to the verification of communicating systems.* PhD thesis, École Normale Supérieure de Lyon, 1996.

[Gin79] Susanna Ginali. Regular trees and the free iterative theory. *Journal of Computer and System Sciences*, 18(3):228–242, 1979. 10.1016/0022-0000(79)90032-1

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[GLSG15] Pierre Genevès, Nabil Layaïda, Alan Schmitt, and Nils Gesbert. Efficiently deciding μ-calculus with converse over finite trees. *ACM Transactions on Computational Logic*, 16(2):16, 2015.

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006. 10.1007/11780274_27

[GP15] Daniel Gustafsson and Nicolas Pouillard. Foldable containers and dependent types. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.646.3537, 2015

[Hed98] Michael Hedberg. A coherence theorem for martin-löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998. http://journals.cambridge.org/action/displayAbstract?aid=44199

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. 10.1145/359576.359585

[Hop71] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford University, 1971.
http://i.stanford.edu/TR/CS-TR-71-190.html

[JKS12] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. CoCaml: Programming with coinductive types. Technical report, Cornell University, 2012.
http://hdl.handle.net/1813/30798

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983. 10.1016/0304-3975(82)90125-6

[Li15] Nuo Li. *Quotient types in type theory.* PhD thesis, University of Nottingham, 2015.
http://eprints.nottingham.ac.uk/28941/

[LS13] Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *LICS*, pages 223–232. IEEE Computer Society, 2013. 10.1109/LICS.2013.28

[Man14] Sebastian Maneth. Equivalence problems for tree transducers: A brief survey. In *AFL*, volume 151 of *EPTCS*, pages 74–93, 2014. 10.4204/EPTCS.151.5

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. *Studies in Logic and the Foundations of Mathematics*, 80:73–118, 1975.
10.1016/S0049-237X(08)71945-1

[MLS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory.* Studies in proof theory. Bibliopolis, Napoli, 1984. http://opac.inria.fr/record=b1093069

[MW15] Stefan Milius and Thorsten Wißmann. Finitary corecursion for the infinitary lambda calculus. In *CALCO*, volume 35 of *LIPIcs*, pages 336–351. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2015. 10.4230/LIPIcs.CALCO.2015.336

[Nel83] Evelyn Nelson. Iterative algebras. *Theoretical Computer Science*, 25:67–94, 1983.
10.1016/0304-3975(83)90014-2

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[Par14] Erik Parmann. Investigating streamless sets. In *TYPES*, volume 39 of *LIPIcs*, pages 187–201. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2014.
10.4230/LIPIcs.TYPES.2014.187

[Pau86] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2(4):325–355, 1986.
10.1016/S0747-7171(86)80002-5

[Pau89] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris-Diderot—Paris VII, 1989. https://tel.archives-ouvertes.fr/tel-00431825

[PdAC+16] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. University of Pennsylvania, 2016. http://www.cis.upenn.edu/~bcpierce/sf/

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208. ACM, 1988. 10.1145/53990.54010

[Pic12] Célia Picard. *Représentation coinductive des graphes*. PhD thesis, Université Toulouse III Paul Sabatier, 2012. http://thesesups.ups-tlse.fr/1642/

[Pit01] Andrew M. Pitts. Nominal logic: A first order theory of names and binding. In *TACS*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer, 2001. 10.1007/3-540-45500-0_11

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. 10.1007/3-540-06859-7_148

[Rou68] William C. Rounds. *Trees, transducers and transformations*. PhD thesis, Stanford University, 1968.

[Rou70] William C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4(3):257–287, 1970. 10.1007/BF01695769

[Rut00] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000. 10.1016/S0304-3975(00)00056-6

[Saï97] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *POPL*, pages 292–301. ACM Press, 1997. 10.1145/263699.263742

[SC10] Arnaud Spiwack and Thierry Coquand. Constructively Finite? In *Contribuciones científicas en honor de Mirian Andrés Gómez*, pages 217–230. Universidad de La Rioja, 2010. https://hal.inria.fr/inria-00503917

[Sei90] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990. 10.1137/0219027

[Sei94] Helmut Seidl. Equivalence of finite-valued tree transducers is decidable. *Mathematical Systems Theory*, 27(4):285–346, 1994. 10.1007/BF01192143

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008. 10.1007/978-3-540-71067-7_23

[Soz09] Matthieu Sozeau. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009. 10.6092/issn.1972-5787/1574

[Soz10] Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2010. 10.1007/978-3-642-14052-5_29

[Spa15] Régis Spadotti. A mechanized theory of regular trees in dependent type theory. In *ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 405–420. Springer, 2015. 10.1007/978-3-319-22102-1_27

[Spr98] Christoph Sprenger. A verified model checker for the modal μ-calculus in Coq. In *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1998. 10.1007/BFb0054171

[ST14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In *ITP*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014. 10.1007/978-3-319-08970-6_32

[SV95] Giora Slutzki and Sándor Vágvölgyi. Deterministic top-down tree transducers with iterated lookahead. *Theoretical Computer Science*, 143(2):285–308, 1995. 10.1016/0304-3975(94)00111-U

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. http://projecteuclid.org/euclid.pjm/1103044538

[Tha70] James W. Thatcher. Generalized sequential machine maps. *Journal of Computer and System Sciences*, 4(4):339–367, 1970. 10.1016/S0022-0000(70)80017-4

[TW06] Ming-Hsien Tsai and Bow-Yaw Wang. Formalization of CTL* in calculus of inductive constructions. In *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2006. 10.1007/978-3-540-77505-8_25

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013. http://homotopytypetheory.org/book

[VGPA00] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN*, volume 1961 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2000. 10.1007/3-540-44464-5_13

APPENDIX A

DEFINITIONS AND NOTATIONS

IDENTITY TYPE

Definition 1 **Equality on functions**

Let $A, B : \textsc{Type}$ *and* $f, g : A \to B$*. Then,*

$$f \doteq g \overset{\text{def}}{\iff} \forall x.\ f(x) = g(x).$$

Definition 2 **Lifting of equalities to equalities on functions (congruence)**

Let A, B *be two types. Then,*

$$\mathsf{ap} : \forall (f : A \to B).\ \forall \{a, a' : A\}.\ a = a' \to f(a) = f(a')$$
$$\mathsf{ap}\ \ f\ \ \mathsf{refl}\ \equiv\ \mathsf{refl}.$$

Definition 3 **Substitution (transport)**

$$\mathsf{subst} : \forall \{A : Type\}.\ \forall (B : A \to \textsc{Type}).\ \forall \{a, a' : A\}.\ x = y \to B(x) \to B(y)$$
$$\mathsf{subst}\ \ B\ \ \mathsf{refl}\ \equiv\ \mathsf{id}.$$

Proposition 1 **Substitution property**

Let $A, B : \textsc{Type}$ *be two types,* $f : A \to B$ *and* $C : B \to Type$ *be a type family. For all* $x, y : A$*,* $p : x = y$*,* $c : C(f(x))$*, we have:*

$$\mathsf{subst}\ (C \circ f)\ p\ u = \mathsf{subst}\ C\ (\mathsf{ap}\ f\ p)\ u.$$

Proof. Immediate by induction on p. □

UNIQUENESS OF IDENTITY PROOFS

Definition 4 **Uniqueness of identity proofs (UIP)**

The uniqueness of identity proofs principle is given by:

$$\textsc{uip} : \textsc{Type} \to \textsc{Prop}$$
$$\textsc{uip}\ \ A\ \equiv\ \forall \{x, y : A\}.\ \forall (p, q : x = y).\ p = q.$$

Theorem 1 **Hedberg's theorem [Hed98]**

Let A *be a type. If Leibniz equality is decidable on* A *then* A *satisfies* UIP*. Formally,*

$$\forall (A : \textsc{Type}).\ (\forall (a, a' : A).\ (a = a') \uplus (a \neq a')) \to \textsc{uip}\ A.$$

SIGNATURE

EXTENSION FUNCTOR DEFINITION

Definition 5 Extension functor

The extension functor induced by a signature S is defined as follows:

$$\mathsf{ext} : \textsc{Sig} \to \textsc{Type} \to \textsc{Type}$$
$$\mathsf{ext}\ \ S\ \ X\ \ \equiv\ \sum_{o:S.\mathsf{Op}} \textsc{Vec}\ X\ (S.\,\mathsf{Ar}\ o)$$

We specify a coercion from signatures to functions through ext:

Coercion $\mathsf{ext} : \textsc{Sig} \rightarrowtail \textsc{Funclass}$

Definition 6 Action on functions

The action on functions of the extension functor of a signature S is given by:

$$\mathsf{map}^{\textsc{Sig}} : \{S : \textsc{Sig}\} \to (X \to Y) \to S(X) \to S(Y)$$
$$\mathsf{map}^{\textsc{Sig}}\ \ f\ \ (o, os)\ \ \equiv\ \ (o, \mathsf{map}^{\textsc{Vec}}\ \ f\ \ os)$$

COMBINATORS

Definition 7 Constant signature

Given a type X, the constant signature is defined as follows:

$$\mathsf{const} : \textsc{Type} \to \textsc{Sig}$$
$$\mathsf{const}\ \ X\ \ \equiv\ \ X \triangleleft \lambda_.\ \mathsf{zero}$$

Definition 8 Disjoint sum of signatures

Given two signatures S_1 and S_2, the disjoint sum of S_1 and S_2 is defined as follows:

$$_\uplus_ : \textsc{Sig} \to \textsc{Sig} \to \textsc{Sig}$$
$$S_1 \uplus S_2\ \ \equiv\ \ S_1.\mathsf{Op} \uplus S_2.\mathsf{Op} \triangleleft [S_1.\,\mathsf{Ar}, S_2.\,\mathsf{Ar}]$$

Definition 9 Product of signatures

Given two signatures S_1 and S_2, the product of S_1 and S_2 is defined as follows:

$$_\times_ : \textsc{Sig} \to \textsc{Sig} \to \textsc{Sig}$$
$$S_1 \times S_2\ \ \equiv\ \ S_1.\mathsf{Op} \times S_2.\mathsf{Op} \triangleleft \mathsf{uncurry}(\lambda o_1.\ \lambda o_2.\ S_1.\,\mathsf{Ar}\ o_1 + S_2.\,\mathsf{Ar}\ o_2)$$

FREE MONAD

DEFINITION

Definition 10 **Free monad definition**

The free monad of S is obtained from the term algebra induced by the signature $S \uplus \mathsf{const}\, X$.

$$_^*_ : \textsc{Sig} \to \textsc{Type} \to \textsc{Type}$$
$$S^*(X) \equiv \mathbb{T}_{S_X}$$
$$\textbf{where } S_X : \textsc{Sig}$$
$$S_X \equiv S \uplus \mathsf{const}(X)$$

Definition 11 **Guarded free monad**

The guarded free monad of S is defined as follows:

$$_^+_ : \textsc{Sig} \to \textsc{Type} \to \textsc{Type}$$
$$S^+(X) \equiv S(S^*(X))$$

CONSTRUCTORS

Definition 12 **Variable constructor**

The constructor introducing variables is defined as follows:

$$\mathsf{var} : \{S : \textsc{Sig}\} \to \{X : \textsc{Type}\} \to X \to S^*(X)$$
$$\mathsf{var}\ \ x \equiv \mathsf{inr}\, x \lhd [\,]$$

Definition 13 **Term constructor**

The constructor introducing terms is defined as follows:

$$_ \lhd _ : \{S : \textsc{Sig}\} \to \{X : \textsc{Type}\} \to S^+(X) \to S^*(X)$$
$$o \lhd v \equiv \mathsf{inl}\, o \lhd v$$

Definition 14 **Guarded term constructor**

The constructor introducing guarded terms is defined as follows:

$$_ \lhd^+ _ : \{S : \textsc{Sig}\} \to \{X : \textsc{Type}\} \to S(S^+(X)) \to S^+(X)$$
$$_ \lhd^+ _ \equiv \mathsf{map}^{S(-)}\ (_ \lhd _)$$

ITERATOR

Definition 15 **Iterator definition**

The iterator function on free monads is defined as follows:

$$\mathsf{fold}^* : \{S : \textsc{Sig}\} \to (X \to A) \to (S(A) \to A) \to S^*(X) \to A$$
$$\mathsf{fold}^*\ \ v\ \ f\ \ (\mathsf{var}\, x) \equiv v(x)$$
$$\mathsf{fold}^*\ \ v\ \ f\ \ (o \lhd os) \equiv f\ o\ (\mathsf{map}^{\textsc{Vec}}\ (\mathsf{fold}^*\ v\ f)\ os)$$

MAYBE

DEFINITIONS

Definition 16 **Inductive datatype definition**

The type MAYBE *is defined as follows:*

$$
\begin{array}{l}
\textbf{inductive}\ \textsc{Maybe}\ (A : \textsc{Type}) : \textsc{Type} \\
\quad \mid \ \ \mathsf{just} : A \to \textsc{Maybe}\, A \\
\quad \mid \ \mathsf{nothing} : \textsc{Maybe}\, A
\end{array}
$$

Definition 17 **Action on functions**

The action on functions of MAYBE *is given by:*

$$
\begin{array}{llll}
\mathsf{map}^{\textsc{Maybe}} : (A \to B) \to \textsc{Maybe}\, A \to \textsc{Maybe}\, A \\
\mathsf{map}^{\textsc{Maybe}}\ f\ \ \mathsf{nothing} & \equiv & \mathsf{nothing} \\
\mathsf{map}^{\textsc{Maybe}}\ f\ \ (\mathsf{just}\, a) & \equiv & \mathsf{just}(f(a))
\end{array}
$$

Definition 18 **Susbtitution**

The substitution operation on MAYBE *is given by:*

$$
\begin{array}{llll}
_ \gg\!= _ : \textsc{Maybe}\, A \to (A \to \textsc{Maybe}\, B) \to \textsc{Maybe}\, B \\
\mathsf{nothing} & \gg\!= & f & \equiv \mathsf{nothing} \\
\mathsf{just}\, a & \gg\!= & f & \equiv f(a)
\end{array}
$$

PREDICATE LIFTING

Definition 19 **Predicate lifting (exists)**

Given a predicate $P : A \to \textsc{Prop}$*, the operation lifting* P*, denoted* $\lfloor P \rfloor$ *is introduced as follows:*

$$
\begin{array}{l}
\textbf{inductive}\ \lfloor _ \rfloor\ \{A : \textsc{Type}\}\ (P : A \to \textsc{Prop}) : \textsc{Maybe}\, A \to \textsc{Prop} \\
\quad \mid \ \mathsf{exists} : \{a : A\} \to P(a) \to \mathsf{just}\, a \in \lfloor P \rfloor
\end{array}
$$

Definition 20 **Predicate lifting (forall)**

Given a predicate $P : A \to \textsc{Prop}$*, the operation lifting* P*, denoted* $\textsc{All} P$ *is introduced as follows:*

$$
\begin{array}{l}
\textbf{inductive}\ \textsc{All}\ \{A : \textsc{Type}\}\ (P : A \to \textsc{Prop}) : \textsc{Maybe}\, A \to \textsc{Prop} \\
\quad \mid \ \mathsf{nothing} : \mathsf{nothing} \in \textsc{All} P \\
\quad \mid \ \ \mathsf{just} : \{a : A\} \to P(a) \to \mathsf{just}\, a \in \textsc{All} P
\end{array}
$$

Definition 21 **Constructor predicate (just)**

The function is-just *characterizes terms in* MAYBE *that are produced with* just*:*

$$\text{is-just} : \textsc{Maybe}\, A \to \textsc{Prop}$$
$$\text{is-just} \;\equiv\; \lfloor \text{const}\, \top \rfloor$$

Definition 22 **Constructor predicate (nothing)**

The function is-nothing *characterizes terms of* MAYBE *that are produced with* nothing*.*

$$\text{is-nothing} : \textsc{Maybe}\, A \to \textsc{Prop}$$
$$\text{is-nothing} \;\equiv\; \neg_ \circ \text{is-just}$$

PROPERTIES

Proposition 2 **Extraction of a value from a proof of is-just**

Given a term $m : \textsc{Maybe}\, A$ *and a proof of* $p : \text{is-just}\, m$*, we can extract a value* $a : A$*, such that* $m = \text{just}\, a$*.*

Proof. The value is obtained from the following function:

$$\text{from-just} : \{m : \textsc{Maybe}\, A\} \to \text{is-just}\, m \to A$$
$$\text{from-just}\;\; (\text{just}\, a) \;\; _ \;\equiv a$$
$$\text{from-just}\;\; \text{nothing} \;\; ()$$

Then, by case-analysis on m, that we have $m = \text{just}(\text{from-just}\; m\; p)$. □

Proposition 3 $\lfloor _ \rfloor$ **specification**

Let $A : \textsc{Type}$*, and* P *a predicate on* A*. For all* $m : \textsc{Maybe}\, A$*, we have*

$$m \in \lfloor P \rfloor \iff \exists (a : A).\; m = \text{just}\, a \wedge P(a).$$

Proof. Immediate by case-analysis on m. □

Proposition 4 **is-just and map**

Let $A, B : \textsc{Type}$*,* $f : A \to B$ *and* $m : \textsc{Maybe}\, A$*. Then, we have:*

$$\text{is-just}(\text{map}^{\textsc{Maybe}}\; f\; m) \iff \text{is-just}\, m$$

Proof. Immediate by case-analysis on m. □

Proposition 5 $\lfloor _ \rfloor$ **and substitution**

Let $A, B : \textsc{Type}$*,* $P : A \to \textsc{Prop}$*,* $f : A \to \textsc{Maybe}\, B$ *and* $m : \textsc{Maybe}\, A$*. Then, we have:*

$$m \gg\!= f \in \lfloor P \rfloor \iff m \in \big\lfloor \lfloor P \rfloor \circ f \big\rfloor.$$

Proof. Immediate by case-analysis on m. □

CANONICAL FINITE SET

DEFINITION

Definition 23 **Inductive definition**

The type of canonical finite sets is defined as follows:

$$\textbf{inductive}\ \textsc{Fin} : \mathbb{N} \to \textsc{Type}$$
$$\mathsf{zero} : \{n : \mathbb{N}\} \to \textsc{Fin}(\mathsf{suc}\, n)$$
$$\mathsf{suc} : \{n : \mathbb{N}\} \to \textsc{Fin}\, n \to \textsc{Fin}(\mathsf{suc}\, n).$$

Proposition 6 **Decidable equality**

For all $n : \mathbb{N}$, equality is decidable on canonical finite sets $\textsc{Fin}\, n$:

$$\forall (n : \mathbb{N}).\ \textsc{EqDec}(\textsc{Fin}\, n).$$

Proof. By induction on term of type Fin. □

Definition 24 **Heterogeneous equality**

The heterogenous equality on canonical finite sets is defined inductively as follows:

$$\textbf{inductive}\ _ \simeq _ : \forall \{n\}.\ \forall \{n'\}.\ \textsc{Fin}\, n \to \textsc{Fin}\, n' \to \textsc{Prop}$$
$$\simeq\text{-}\mathsf{zero} : \forall \{n\}.\ \mathsf{zero}_n \simeq \mathsf{zero}_n$$
$$\simeq\text{-}\mathsf{suc} : \forall \{n, n'\}.\ \forall \{i : \textsc{Fin}\, n\}.\ \forall \{i' : \textsc{Fin}\, n'\}.\ i \simeq i' \to \mathsf{suc}\, i \simeq \mathsf{suc}\, i'.$$

INDEXED SUM

Definition 25 **Indexed sum**

Given a natural number n and a map $f : \textsc{Fin}\, n \to \mathbb{N}$, the indexed sum of f, denoted $\underset{k:\textsc{Fin}\, n}{\mathsf{sum}} f(k)$ or simply $\mathsf{sum}\, f$, is defined as follows:

$$\mathsf{sum} : \forall \{n : \mathbb{N}\}.\ (\textsc{Fin}\, n \to \mathbb{N}) \to \mathbb{N}$$
$$\mathsf{sum}\ \{\mathsf{zero}\}\ \ _ \ \equiv\ 0$$
$$\mathsf{sum}\ \{\mathsf{suc}\, n\}\ \ f\ \equiv\ f(\mathsf{zero}) + \mathsf{sum}(f \circ \mathsf{suc}).$$

Proposition 7 **Fin-indexed sum is finite**

Let $n : \mathbb{N}$ and $f : \textsc{Fin}\, n \to \mathbb{N}$ be a fin-indexed function. Then, we have

$$\underset{k:\textsc{Fin}\, n}{\mathsf{sum}} \left(\textsc{Fin}(f(k))\right) \cong \textsc{Fin}(\underset{k}{\mathsf{sum}}\, f(k)).$$

Proof. Let $n : \mathbb{N}$ and $f : \textsc{Fin}\, n \to \mathbb{N}$. The proof is straightforward by induction on n and by proving the following intermediate decomposition result:

$$\forall (m : \mathbb{N}).\ \forall (g : \textsc{Fin}(\mathsf{suc}\, m) \to \mathbb{N}).\ \left(\underset{k:\textsc{Fin}(\mathsf{suc}\, m)}{\mathsf{sum}} g(k)\right) \cong \left(\textsc{Fin}(g(\mathsf{zero})) \uplus \underset{k:\textsc{Fin}\, m}{\mathsf{sum}} g(\mathsf{suc}\, k)\right).$$ □

VECTOR

DEFINITION

Definition 26 **Inductive definition**

$$\textbf{inductive}\ \textsc{Vec}\ (A : \textsc{Type}) : \mathbb{N} \to \textsc{Type}$$
$$[\,] : \textsc{Vec}\ A\ \mathsf{zero}$$
$$_::_ : \{n : \mathbb{N}\} \to A \to \textsc{Vec}\ A\ n \to \textsc{Vec}\ A\ (\mathsf{suc}\, n).$$

VECTORS AND FINITE MAPS

Definition 27 **From finite maps to vectors**

$$\mathsf{tabulate} : \{A : \textsc{Type}\} \to \{n : \mathbb{N}\} \to (\textsc{Fin}\, n \to A) \to \textsc{Vec}\ A\ n$$
$$\mathsf{tabulate}\ \ \{A\}\ \ \{\mathsf{zero}\}\ \ _\ \equiv\ [\,]$$
$$\mathsf{tabulate}\ \ \{A\}\ \ \{\mathsf{suc}\, n\}\ \ f\ \equiv\ f(\mathsf{zero}) :: \mathsf{tabulate}(f \circ \mathsf{suc})$$

Definition 28 **From vectors to finite maps**

$$\mathsf{lookup} : \{A : \textsc{Type}\} \to \{n : \mathbb{N}\} \to \textsc{Vec}\ A\ n \to \textsc{Fin}(n) \to A$$
$$\mathsf{lookup}\ \ (x :: _\ )\ \ \mathsf{zero}\ \equiv\ x$$
$$\mathsf{lookup}\ \ (_ :: xs)\ \ (\mathsf{suc}\, i)\ \equiv\ \mathsf{lookup}\ xs\ i$$

We specify a coercion from vectors to functions through lookup:

Coercion lookup : $\textsc{Vec} \rightarrowtail \textsc{FunClass}$

Proposition 8 **Lookup-tabulate identity**

Let $n : \mathbb{N}$ and $f : \textsc{Fin}\, n \to A$. Then, we have:

$$\forall (i : \textsc{Fin}\, n).\ \mathsf{lookup}\ (\mathsf{tabulate}\ f)\ i = f(i).$$

Proof. Immediate by induction on n. □

Proposition 9 **Tabulate-lookup identity**

Let $n : \mathbb{N}$. Then, we have:

$$\forall (v : \textsc{Vec}\ A\ n).\ \mathsf{tabulate}(\mathsf{lookup}\, v) = v.$$

Proof. Immediate by induction on the vector. □

Proposition 10 **Extensional equality**

Let $n : \mathbb{N}$ and $A : \textsc{Type}$. For all $v, v' : \textsc{Vec}\ A\ n$, if $\forall (k : \textsc{Fin}\, n).\ v(k) = v'(k)$ then $v = v'$.

Proof. Immediate by vector induction. □

FUNCTORIALITY

Definition 29 **Action on functions**

$$\begin{array}{l} \mathsf{map}^{\mathrm{Vec}} : \forall n.\ (A \to B) \to \mathrm{Vec}\ A\ n \to \mathrm{Vec}\ A\ n \\ \mathsf{map}^{\mathrm{Vec}}\ \ f\ \ [\,] \qquad\qquad \equiv\ [\,] \\ \mathsf{map}^{\mathrm{Vec}}\ \ f\ \ (x :: xs)\ \equiv\ f(x) :: \mathsf{map}^{\mathrm{Vec}}\ f\ xs \end{array}$$

Proposition 11 **Map lookup simplification**

Let $A, B : \mathrm{Type}$, $f : A \to B$, $n : \mathbb{N}$, *and* $v : \mathrm{Vec}\ A\ n$*. Then, we have*

$$\forall (k : \mathrm{Fin}\, n).\ \left(\mathsf{map}^{\mathrm{Vec}}\ f\ v\right)(k) = f(v(k)).$$

Proof. Immediate by induction on k. □

LIST

DEFINITION

Definition 30 **Inductive definition**

The type of lists over A is introduced as follows:

$$\textbf{inductive}\ \textsc{List}\ (A : \textsc{Type}) : \textsc{Type}$$
$$\mid\ [\,] : \textsc{List}\ A$$
$$\mid\ _::_ : A \to \textsc{List}\ A \to \textsc{List}\ A.$$

LISTS AS FINITE MAPS

Definition 31 **Finite maps to lists**

Given a finite map $f : \textsc{Fin}\,n \to A$:

$$\mathsf{tabulate} : \{A : \textsc{Type}\} \to \{n : \mathbb{N}\} \to (\textsc{Fin}\,n \to A) \to \textsc{List}\ A$$
$$\mathsf{tabulate}\ \{A\}\ \{\mathsf{zero}\}\ _ \equiv [\,]$$
$$\mathsf{tabulate}\ \{A\}\ \{\mathsf{suc}\,n\}\ f \equiv f(\mathsf{zero}) :: \mathsf{tabulate}(f \circ \mathsf{suc})$$

Definition 32 **Lists to finite maps**

A list $l : A$ induces a finite map $\textsc{Fin}(\mathsf{length}\,l) \to A$ defined as follows:

$$\mathsf{lookup} : \{A : \textsc{Type}\} \to \forall\{n : \mathbb{N}\}.\ \forall(l : \textsc{List}\ \ A).\ \to \textsc{Fin}(\mathsf{length}\,l) \to A$$
$$\mathsf{lookup}\ (x :: _\,)\ \mathsf{zero} \equiv x$$
$$\mathsf{lookup}\ (_ :: xs)\ (\mathsf{suc}\,i) \equiv \mathsf{lookup}\ xs\ i$$

We specify a coercion from lists to functions through lookup:

Coercion lookup : $\textsc{List} \rightarrowtail \textsc{FunClass}$

PREDICATE LIFTING

Definition 33 **Exists predicate definition**

The predicate lifting exists

$$\textbf{inductive}\ \textsc{Any}\ (P : A \to \textsc{Prop}) : \textsc{List}\ A \to \textsc{Prop}$$
$$\mid\ \mathsf{here} : \forall\{x\}.\ \forall\{xs\}.\ P(x) \to \textsc{Any}\ \ P\ (x :: xs)$$
$$\mid\ \mathsf{there} : \forall\{x\}.\ \forall\{xs\}.\ \textsc{Any}\ \ P\ xs \to \textsc{Any}\ \ P\ (x :: xs).$$

Definition 34 **All predicate definition**

The predicate all

$$\textbf{inductive}\ \textsc{All}\ (P : A \to \textsc{Prop}) : \textsc{List}\ A \to \textsc{Prop}$$
$$\mid\ \mathsf{all}_{[\,]} : \textsc{All}\ P\ [\,]$$
$$\mid\ \mathsf{all}_{::} : \forall\{x\}.\ \forall\{xs\}.\ P(x) \to \textsc{All}\ P\ xs \to \textsc{All}\ P\ (x :: xs).$$

LIST MEMBERSHIP

Definition 35 Membership

Let $(A, \approx_A)$ be a setoid. The membership operation is defined as follows:

$$_\in_ : A \to \textsc{List}\, A \to \textsc{Prop}$$
$$x \in l \;\equiv\; \textsc{Any}\ (\lambda y.\ y \approx x)\ l$$

Definition 36 Proof-relevant membership

Let $(A, \approx_A)$ be a setoid. The proof-relevant membership operation is defined as follows:

$$_[\in]_ : A \to \textsc{List}\, A \to \textsc{Prop}$$
$$x\,[\in]\,l \;\equiv\; \{\, i : \textsc{Fin}(\mathsf{length}\, l) \mid l(i) \approx x \,\}\,.$$

PROPERTIES

Proposition 12 ALL lookup

Let $A : \textsc{Type}$, $P : A \to \textsc{Prop}$ and $l : \textsc{List}\, A$. We have:

$$\forall (a : A).\ a \in l \to \textsc{All}\ P\ l \to P(a).$$

Proof. Immediate by induction on the membership proof $a \in l$ and case-analysis on $\textsc{All}\ P\ l$. □

Proposition 13 ALL action on predicates

Let $A : \textsc{Type}$, $P, Q : A \to \textsc{Prop}$ and $l : \textsc{List}\, A$. We have:

$$(\forall x.\ P(x) \to Q(x)) \to \textsc{All}\ P\ l \to \textsc{All}\ Q\ l.$$

Proof. Immediate by induction on the $\textsc{All}\ P\ l$ predicate. □

DUPLICATE-FREE LIST

Definition 37 Inductive definition

inductive $\textsc{NoDup} : \textsc{List}\, A \to \textsc{Prop}$

| $\mathsf{nodup}_{[]} : \textsc{NoDup}\,[\,]$
| $\mathsf{nodup}_{::} : \forall\{x\}.\ \forall\{xs\}.\ x \notin xs \to \textsc{NoDup}\, xs \to \textsc{NoDup}(x :: xs).$

Proposition 14 Duplicate removal

Let $(A, \approx_A)$ be a setoid such that $\approx_A$ is decidable. Then, for any list $l : \textsc{List}\, A$, there exists a list l' such that:
(i) $\forall a.\ a \in l \leftrightarrow a \in l'$,
(ii) l' is duplicate-free.

BOUNDED-INDUCTION

Definition 38 **Inductive definition**

inductive $_\succ_ \ \{A : \textsc{Type}\} : \textsc{List}\ A \to \textsc{List}\ A \to \textsc{Prop}$
| $\succ$-intro $: \forall(x : A).\ \forall(l : \textsc{List}\ A).\ \to x \notin l \to x :: l \succ l.$

Definition 39 **Accessibility definition**

$\lceil_\rceil : \{A : \textsc{Type}\} \to \textsc{List}\ A \to \textsc{Prop}$
$\lceil l \rceil \ \equiv\ \textsc{Acc}\ (_\succ_)\ l$

this is an example Definition A.39.

Definition 40 **Elimination**

$_\lceil::\rceil_ : \{A : \textsc{Type}\}\ \{x : A\}\ \{l : \textsc{List}\ A\} \to x \notin l \to \lceil l \rceil \to \lceil x :: l \rceil$
$p\ \lceil::\rceil\ (\text{acc-intro}\ f)\ \equiv\ f(p)$

Proposition 15 **Well-founded**

Let $(A, \approx)$ be a setoid such that setoid equality is decidable. If A is listable then the relation $_\succ_$ is well-founded.

INDEX MANAGEMENT

In the remaining of this section, we assume that $(A, \approx_A)$ is a weakly finitely indexed setoid. By definition , there exist two maps index $: A \to \textsc{Fin}\ \#A$ and value $: \textsc{Fin}\ \#A \to A$.

Definition 41 **Lookup index**

$_[_] : \forall(l : \textsc{List}\ \#A).\ \textsc{Fin}(\text{length}\ l) \to A$
$l[i]\ \equiv\ \text{value}(l(i))$

Definition 42 **Membership test**

$_\in^{\#}?_ : \forall(a : A).\ \forall(l : \textsc{List}(\textsc{Fin}\ \#A)).\ \{i : \textsc{Fin}(\text{length}\ l) \mid l[i] \approx a\} \uplus \{\text{index}\ a \notin l\}$

APPENDIX B

CYCLIC TERMS PROPERTIES

In this appendix, we prove some technical results about cyclic terms. Ultimately, this consists in proving that the type $\mathbb{C}_S : \mathbb{N} \to \textsc{Type}$ is a monad. A categorical treatment of syntax with binders can be found in [Ahr15]. In the following, we summarize the various syntactic properties on cyclic terms. We fix an arbitrary signature S.

EQUALITY PROOF RULES

In this section, we show that vectors—viewed as finite maps—satisfy an extensionality principle.

Proposition 1 **Extensionality principle on vectors**

Let A be a type, n be a natural number and $v, v' : \textsc{Vec}\ A\ n$ be two vectors. Then, we have:

$$\textsf{vec-ext}\ \frac{\forall (i : \textsc{Fin}\, n).\ v(i) = v'(i)}{v = v'}.$$

Proof. Let $n : \mathbb{N}$, $v, v' : \textsc{Vec}\ A\ n$ be two vectors and $H : \forall i.\ v(i) = v'(i)$. By induction on v:

- **Base step.** Assume that $v = [\,]$.
  Here, the vector v' ought to be $[\,]$. Consequently, we have $v = v'$.
- **Inductive step.** Assume that $n = \textsf{suc}\, n'$ and $v = a :: u$ where $a : A$ and $u : \textsc{Vec}\ A\ n'$.
  The induction hypothesis is given by

$$\forall (v' : \textsc{Vec}\ A\ n').\ (\forall (i : \textsc{Fin}\, n').\ u(i) = v'(i)) \to u = v'. \qquad \text{(IH)}$$

  By case-analysis on v', there exist $a' : A$ and $u' : \textsc{Vec}\ A\ n'$ such that $v' = a :: u'$. Then,

$$\begin{aligned} v = a :: u &= a :: u' && \text{(by rewriting (IH) and proving its premise with } H \circ \textsf{suc}) \\ &= a' :: u' = v'. && \text{(by rewriting } H(\textsf{zero}) : a = a') \end{aligned}$$ □

The extensionality principle on vectors extends to the constructor $\textsf{rec}\,_ \lhd _$ of cyclic terms.

Proposition 2 **Extensionality principle on rec**

Let n be a natural number, $o : S.\textsf{Op}$ be a function symbol, $os, os' : \textsc{Vec}\ (\mathbb{C}_S(\textsf{suc}\, n))\ (S.\textsf{Ar}\, o)$ be two vectors of arguments of cyclic terms. Then, we have:

$$\textsf{rec-ext}\ \frac{\forall (i : \textsc{Fin}(S.\textsf{Ar}\, o)).\ v(i) = v'(i)}{\textsf{rec}\, o \lhd os = \textsf{rec}\, o \lhd os'}.$$

Proof. Consequence of Proposition B.1. □

The following proposition can be used on function defined by induction over cyclic terms.

Proposition 3 **Extensionality principle on** $\mathsf{rec}^{\mathsf{map}}\text{-}\mathsf{ext}$

Let m, m', n be natural numbers, $o : S.\mathsf{Op}$ be a function symbol, os, os', os'' be two vectors of arguments of cyclic terms. Moreover, let $f : \mathbb{C}_s(m) \to \mathbb{C}_S(\mathsf{suc}\, n)$ and $g : \mathbb{C}_S(m') \to \mathbb{C}_S(\mathsf{suc}\, n)$ be two functions on cyclic terms. Then, we have:

$$\mathsf{rec}^{\mathsf{map}}_{\mathsf{L}}\text{-}\mathsf{ext}\ \frac{\forall i.\ f(os(i)) = os'(i)}{\mathsf{rec}\, o \lhd \mathsf{map}^{\mathrm{VEC}}\ f\ os = \mathsf{rec}\, o \lhd os'} \qquad \mathsf{rec}^{\mathsf{map}}_{\mathsf{R}}\text{-}\mathsf{ext}\ \frac{\forall i.\ os(i) = f(os'(i))}{\mathsf{rec}\, o \lhd os = \mathsf{rec}\, o \lhd \mathsf{map}^{\mathrm{VEC}}\ f\ os'}$$

$$\mathsf{rec}^{\mathsf{map}}\text{-}\mathsf{ext}\ \frac{\forall i.\ f(os(i)) = g(os'(i))}{\mathsf{rec}\, o \lhd \mathsf{map}^{\mathrm{VEC}}\ f\ os = \mathsf{rec}\, o \lhd \mathsf{map}^{\mathrm{VEC}}\ f\ os'}.$$

Proof. Let $m, m', n : \mathbb{N}$, $o : S.\mathsf{Op}$, $os : \mathrm{VEC}\ (\mathbb{C}_S(m))\ (S.\mathsf{Ar}\, o)$, $os' : \mathrm{VEC}\ (\mathbb{C}_S(m'))\ (S.\mathsf{Ar}\, o)$. Moreover, let $f : \mathbb{C}_S(m) \to \mathbb{C}_S(\mathsf{suc}\, n)$ and $g : \mathbb{C}_S(m') \to \mathbb{C}_S(\mathsf{suc}\, n)$ be two functions on cyclic terms. We prove only the last rule, namely rec-ext. The others can be derive from it by taking either f or g to be the identity map. Assume that $H : \forall i.\ f(os(i)) = g(os'(i))$:

$$\begin{aligned}
& \mathsf{rec}\, o \lhd \mathsf{map}^{\mathrm{VEC}}\ f\ os = \mathsf{rec}\, o \lhd \mathsf{map}^{\mathrm{VEC}}\ g\ os' \\
\iff & \quad \{ \text{ by Proposition B.1 [vec-ext] } \} \\
& \forall i.\ (\mathsf{map}^{\mathrm{VEC}}\ f\ os)(i) = (\mathsf{map}^{\mathrm{VEC}}\ g\ os)(i) \\
\iff & \quad \{ \text{ by Proposition A.11 } [(\mathsf{map}^{\mathrm{VEC}}\ f\ v)(k) = f(v(k))] \} \\
& \forall i.\ f(os(i)) = g(os(i)) \\
\iff & \quad \{ \text{ by assumption } \} \\
& \top
\end{aligned}$$

□

SHIFT OPERATION

Here, we show the functoriality of an operation which consists in shifting by one all elements of a canonical finite set.

Definition 1 shift **operation**

The action of suc *on morphisms is given by:*

$$\begin{aligned}
& \mathsf{shift} : (\mathrm{FIN}\, m \to \mathrm{FIN}\, n) \to \mathrm{FIN}(\mathsf{suc}\, m) \to \mathrm{FIN}(\mathsf{suc}\, n) \\
& \mathsf{shift}\ \ f\ \ \mathsf{zero} \quad\ \ \equiv\ \mathsf{zero} \\
& \mathsf{shift}\ \ f\ \ (\mathsf{suc}\, m)\ \equiv\ \mathsf{suc}(f(m))
\end{aligned}$$

Now, we prove the functor laws:

Proposition 4 shift **functor laws**

Let $n : \mathbb{N}$. The successor operation on finite sets is functorial:

(i) $\mathsf{shift}\ \mathsf{id} \doteq \mathsf{id}_{\mathrm{FIN}(\mathsf{suc}\, n)}$,

(ii) $\forall (f : \mathrm{FIN}\, m \to \mathrm{FIN}\, n).\ \forall (g : \mathrm{FIN}\, n \to \mathrm{FIN}\, p).\ \mathsf{shift}(g \circ f) \doteq \mathsf{shift}(g) \circ \mathsf{shift}(f)$.

(iii) $\forall (f, g : \mathrm{FIN}\, m \to \mathrm{FIN}\, n).\ f \doteq g \to \mathsf{shift}(f) \doteq \mathsf{shift}(g)$.

Proof. All three cases are trivial by case-analysis over elements of the type FIN. □

RENAMING OPERATION

The renaming operation establishes the functoriality of the type $\mathbb{C}_S : \mathbb{N} \to \textsc{Type}$ of cyclic terms. Intuitively, this operation consists in applying a function—acting on finite sets—on each var constructor of a cyclic term.

Definition 2 — Renaming operation

The renaming operation on cyclic terms is defined recursively as follows:

$$\begin{aligned}
&\mathsf{rename} : (\textsc{Fin}\, m \to \textsc{Fin}\, n) \to \mathbb{C}_S(m) \to \mathbb{C}_S(n)\\
&\mathsf{rename}\ f\ (\mathsf{var}\, x) \equiv \mathsf{var}(f(x))\\
&\mathsf{rename}\ f\ (\mathsf{rec}\, o \lhd os) \equiv \mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{rename}(\mathsf{shift}\, f))\ os
\end{aligned}$$

Remark. Note that, even though the recursive call of rename occurs within the function used by $\mathsf{map}^{\textsc{Vec}}$, the Coq termination checker still infers that such a recursive call is valid.

Proposition 5 — Functoriality of cyclic terms

The renaming operation is functorial:

(*i*) $\forall (f, g : \textsc{Fin}\, m \to \textsc{Fin}\, n).\ f \doteq g \to \mathsf{rename}\, f \doteq \mathsf{rename}\, g$,

(*ii*) $\mathsf{rename}\ \mathsf{id} \doteq \mathsf{id}_{\mathbb{C}_S(n)}$,

(*iii*) $\forall (f : \textsc{Fin}\, m \to \textsc{Fin}\, n).\ \forall (g : \textsc{Fin}\, n \to \textsc{Fin}\, p).\ \mathsf{rename}(g \circ f) \doteq \mathsf{rename}(g) \circ \mathsf{rename}(f)$.

Proof. By induction on the cyclic term renamed.

(*i*) Let $m, n : \mathbb{N}$, $f, g : \textsc{Fin}\, m \to \textsc{Fin}\, n$, $E : f \doteq g$, $t : \mathbb{C}_S(m)$.

- **Base step.** Assume that $t = \mathsf{var}\, x$ where $x : \textsc{Fin}\, m$.

$$\begin{aligned}
&\mathsf{rename}\ f\ (\mathsf{var}\, x) = \mathsf{rename}\ g\ (\mathsf{var}\, x)\\
\iff\ &\quad \{ \text{ by reflexivity } \}\\
&\mathsf{var}(f(x)) = \mathsf{var}(g(x))\\
\iff\ &\quad \{ \ _{=}_ \text{ is a congruence } \}\\
&f(x) = g(x)\\
\iff\ &\quad \{ \text{ by assumption } E \ \}\\
&\top
\end{aligned}$$

- **Inductive step.** Assume that $t = \mathsf{rec}\, o \lhd os$ where $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, m))\ (S.\mathsf{Ar}\, o)$. The induction hypothesis is given by:

$$\forall f.\ \forall g.\ f \doteq g \to \forall i.\ \mathsf{rename}\ f\ (os(i)) = \mathsf{rename}\ g\ (os(i)). \tag{IH}$$

$$\begin{aligned}
&\mathsf{rename}\ f\ (\mathsf{rec}\, o \lhd os) = \mathsf{rename}\ g\ (\mathsf{rec}\, o \lhd os)\\
\iff\ &\quad \{ \text{ by reflexivity } \}\\
&\mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{rename}(\mathsf{shift}\, f))\ os = \mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{rename}(\mathsf{shift}\, g))\ os\\
\iff\ &\quad \{ \text{ by Proposition B.2 [rec-ext] and induction hypothesis (IH) } \}\\
&\mathsf{shift}\, f \doteq \mathsf{shift}\, g\\
\iff\ &\quad \{ \text{ by Proposition B.4 } (iii) \ \}\\
&\top
\end{aligned}$$

(ii) Let $n : \mathbb{N}$ and $t : \mathbb{C}_S(n)$.

- **Base step.** Assume that $t = \mathsf{var}\, x$ where $x : \textsc{Fin}\, n$.

$$\mathsf{rename\ id}\ (\mathsf{var}\, x) = \mathsf{var}\, x \Leftrightarrow \mathsf{var}(\mathsf{id}\, x) = \mathsf{var}\, x \quad \text{(by reflexivity)}$$
$$\Leftrightarrow \top \quad \text{(by reflexivity)}$$

- **Inductive step.** Assume that $t = \mathsf{rec}\, o \lhd os$ where $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\,\mathsf{Ar}\, o)$. The induction hypothesis is given by:

$$\forall i.\ \mathsf{rename\ id}\ (os(i)) = os(i). \tag{IH}$$

$$\mathsf{rename\ id}\ (\mathsf{rec}\, o \lhd os) = \mathsf{rec}\, o \lhd os$$
$$\Longleftrightarrow \quad \{\text{by reflexivity}\}$$
$$\mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{rename}(\mathsf{shift\ id}))\ os = \mathsf{rec}\, o \lhd os$$
$$\Longleftrightarrow \quad \{\text{ by Proposition B.2 [rec-ext] and induction hypothesis (IH) }\}$$
$$\mathsf{shift\ id} \doteq \mathsf{id}$$
$$\Longleftrightarrow \quad \{\text{ by Proposition B.4 } (i)\}$$
$$\top$$

(iii) Let $m, n, p : \mathbb{N}$, $f : \textsc{Fin}\, m \to \textsc{Fin}\, n$, $g : \textsc{Fin}\, n \to \textsc{Fin}\, p$ and $t : \mathbb{C}_S(m)$.

- **Base step.** Assume that $t = \mathsf{var}\, x$ where $x : \textsc{Fin}\, n$.

$$\mathsf{rename}\ (g \circ f)\ (\mathsf{var}\, x) = \mathsf{rename}\ g\ (\mathsf{rename}\ f\ (\mathsf{var}\, x))$$
$$\Longleftrightarrow \quad \{\text{reflexivity}\}$$
$$\mathsf{var}(g(f(x))) = \mathsf{var}(g(f(x)))$$
$$\Longleftrightarrow \quad \{\text{ by reflexivity }\}$$
$$\top$$

- **Inductive step.** Assume that $t = \mathsf{rec}\, o \lhd os$ where $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\,\mathsf{Ar}\, o)$. The induction hypothesis is given by:

$$\forall f.\ \forall g.\ \forall i.\ \mathsf{rename}\ (g \circ f)\ (os(i)) = \mathsf{rename}\ g\ (\mathsf{rename}\ f\ (os(i))). \tag{IH}$$

$$\mathsf{rename}\ (g \circ f)\ (\mathsf{rec}\, o \lhd os) = \mathsf{rename}\ g\ (\mathsf{rename}\ f\ (\mathsf{rec}\, o \lhd os))$$
$$\Longleftrightarrow \quad \{\text{ by reflexivity }\}$$
$$\mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{rename}(\mathsf{shift}(g \circ f)))\ os$$
$$= \mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{rename}(\mathsf{shift}\, g))\ (\mathsf{map}^{\textsc{Vec}}\ (\mathsf{rename}(\mathsf{shift}\, f))\ os)$$
$$\Longleftrightarrow \quad \{\text{ by Proposition B.3 [rec}^{\mathsf{map}}\text{-ext] and Proposition B.4 } (ii)\}$$
$$\forall i.\ \mathsf{rename}\ (\mathsf{shift}\, g \circ \mathsf{shift}\, f)\ (os(i))$$
$$= \mathsf{rename}\ (\mathsf{shift}\, g)\ (\mathsf{map}^{\textsc{Vec}}\ (\mathsf{rename}(\mathsf{shift}\, f))\ (os(i)))$$
$$\Longleftrightarrow \quad \{\text{ by Proposition A.11 } [(\mathsf{map}^{\textsc{Vec}}\ f\ v)(k) = f(v(k))]\}$$
$$\forall i.\ \mathsf{rename}\ (\mathsf{shift}\, g \circ \mathsf{shift}\, f)\ (os(i))$$
$$= \mathsf{rename}\ (\mathsf{shift}\, g)\ (\mathsf{rename}\ (\mathsf{shift}\, f)\ (os(i)))$$
$$\Longleftrightarrow \quad \{\text{ by induction hypothesis (IH) }\}$$
$$\top$$

□

The weakening operation consists in shifting every variable by one.

Definition 3 **Weakening**

The weakening operation on cyclic terms is defined as follows:

$$\mathsf{weaken} : \mathbb{C}_S(n) \to \mathbb{C}_S(\mathsf{suc}\, n)$$
$$\mathsf{weaken} \equiv \mathsf{rename\ suc}$$

LIFTING OPERATION

Here, we define the lifting operation that will be used in the definition of the substitution function:

Definition 4 **Lifting operation**

The lifting operation is defined as follows:

$$\mathsf{lift} : (\textsc{Fin}\, m \to \mathbb{C}_S(n)) \to (\textsc{Fin}(\mathsf{suc}\, m) \to \mathbb{C}_S(n))$$
$$\mathsf{lift}\ f\ \mathsf{zero} \equiv \mathsf{var\ zero}$$
$$\mathsf{lift}\ f\ (\mathsf{suc}\, m) \equiv \mathsf{weaken}(f(m))$$

Proposition 6 **Properties on lift**

The lift operation is a congruence and commutes—in a suitable sense—with renaming:

(i) $\forall (f, g : \textsc{Fin}\, m \to \mathbb{C}_S(n)).\ f \doteq g \to \mathsf{lift}(f) \doteq \mathsf{lift}(g)$

(ii) $\forall (f : \mathbb{C}_S(m) \to \mathbb{C}_S(n)).\ \forall (g : \textsc{Fin}\, n \to \textsc{Fin}\, p).\ \mathsf{lift}(\mathsf{rename}\, g \circ f) \doteq \mathsf{rename}(\mathsf{shift}\, g) \circ \mathsf{lift}(f).$

Proof. Trivial by case-analysis over elements of Fin. □

PARALLEL SUBSTITUTION OPERATION

The parallel substitution is an important operation on cyclic terms. In particular, it justifies that the type $\mathbb{C}_S : \mathbb{N} \to \textsc{Type}$ has the structure of a monad.

Definition 5 **Parallel substitution operation**

The parallel substitution function is defined as follows:

$$\mathsf{subst} : (\textsc{Fin}(m) \to \mathbb{C}_S(n)) \to \mathbb{C}_S(m) \to \mathbb{C}_S(n)$$
$$\mathsf{subst}\ f\ (\mathsf{var}\, x) \equiv f(x)$$
$$\mathsf{subst}\ f\ (\mathsf{rec}\, o \lhd os) \equiv \mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{subst}(\mathsf{lift}\, f))\ os$$

Moreover, we define the one variable substitution as follows:

$$_[* := _] : \{n : \mathbb{N}\} \to \mathbb{C}_S(\mathsf{suc}\, n) \to \mathbb{C}_S(n) \to \mathbb{C}_S(n)$$
$$t[* := u] \equiv \mathsf{subst}\ \sigma_u\ t$$
$$\textbf{where}\ \sigma_ : \mathbb{C}_S(n) \to \textsc{Fin}(\mathsf{suc}\, n) \to \mathbb{C}_S(n)$$
$$\sigma_u\ \mathsf{zero} \equiv u$$
$$\sigma_u\ (\mathsf{suc}\, x) \equiv \mathsf{var}\, x$$

In the following, we prove the monad laws.

Proposition 7 **Substitution operation is a congruence**

The substitution operation is a congruence:

$$\forall (f, g : \textsc{Fin}\, m \to \mathbb{C}_S(n)).\ f \doteq g \to \mathsf{subst}(f) \doteq \mathsf{subst}(g).$$

Proof. Straightforward by induction over the cyclic term. □

The left-identity law of the monad, namely $\mathsf{subst}\ f\ (\mathsf{var}\, x) = f\ x$ holds definitionally. We prove the right-identity law.

Proposition 8 Identity substitution

The constructor var *is the identity substitution:*

$$\mathsf{subst}\ \mathsf{var} \doteq \mathsf{id}.$$

Proof. Let $n : \mathbb{N}$ and $t : \mathbb{C}_S(n)$. We proceed by induction on t:

- **Base step.** Assume that $t = \mathsf{var}\, x$ where $x : \mathrm{FIN}\, m$.
  This case is trivial by reflexivity.
- **Inductive step.** Assume that $t = \mathsf{rec}\, o \lhd os$ where $os : \mathrm{VEC}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\, \mathsf{Ar}\, o)$.
  The induction hypothesis is given by:

$$\forall i.\ \mathsf{subst}\ \mathsf{var}\ (os(i)) = os(i). \tag{IH}$$

Then, we have:

$$\begin{aligned}
& \mathsf{subst}\ \mathsf{var}\ (\mathsf{rec}\, o \lhd os) = \mathsf{rec}\, o \lhd os \\
\iff\ & \quad \{ \text{ by reflexivity } \} \\
& \mathsf{rec}\, o \lhd \mathsf{map}^{\mathrm{VEC}}\ (\mathsf{subst}(\mathsf{lift}\, \mathsf{var}))\ os = \mathsf{rec}\, o \lhd os \\
\iff\ & \quad \{ \text{ by } \mathsf{rec}^{\mathsf{map}}_{\mathsf{L}}\text{-}\mathsf{ext} \} \\
& \forall i.\ \mathsf{subst}\ (\mathsf{lift}\, \mathsf{var})\ (os(i)) = os(i) \\
\iff\ & \quad \{ \mathsf{lift}(\mathsf{var}) \doteq \mathsf{var} \text{ by case-analysis on } \mathrm{FIN} \} \\
& \forall i.\ \mathsf{subst}\ \ \mathsf{var}(os(i)) = os(i) \\
\iff\ & \quad \{ \text{ by induction hypothesis (IH) } \} \\
& \top
\end{aligned}$$

□

In the following, we prove properties related to the renaming operation.

Proposition 9 Renaming commutes with substitution

Renaming commutes with substitution:

$$\begin{gathered}
\forall (t : \mathbb{C}_S(m)).\ \forall (f : \mathrm{FIN}\, m \to \mathbb{C}_S(n)).\ \forall (g : \mathrm{FIN}\, n \to \mathrm{FIN}\, p). \\
\mathsf{rename}\ g\ (\mathsf{subst}\ f\ t) = \mathsf{subst}\ (\mathsf{rename}\, g \circ f)\ t.
\end{gathered}$$

Proof. Let $m, n, p : \mathbb{N}$, $t : \mathbb{C}_S(m)$, $f : \mathrm{FIN}\, m \to \mathbb{C}_S(n)$ and $g : \mathrm{FIN}\, n \to \mathrm{FIN}\, p$. We proceed by induction on t:

- **Base step.** Assume that $t = \mathsf{var}\, x$ where $x : \mathrm{FIN}\, m$.
  This case is trivial by reflexivity.
- **Inductive step.** Assume that $t = \mathsf{rec}\, o \lhd os$ where $os : \mathrm{VEC}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\, \mathsf{Ar}\, o)$.
  The induction hypothesis is given by:

$$\begin{gathered}
\forall (n, p : \mathbb{N}).\ \forall (f : \mathrm{FIN}\, m \to \mathbb{C}_S(n)).\ \forall (g : \mathrm{FIN}\, n \to \mathrm{FIN}\, p).\ \forall k. \\
\mathsf{rename}\ g\ (\mathsf{subst}\ f\ (os(k))) = \mathsf{subst}\ (\mathsf{rename}\, g \circ f)\ (os(k)).
\end{gathered} \tag{IH}$$

Then, we have:

$$
\begin{aligned}
& \textsf{rename}\ g\ (\textsf{subst}\ f\ (\textsf{rec}\ o \lhd os)) = \textsf{subst}\ (\textsf{rename}\ g \circ f)\ (\textsf{rec}\ o \lhd os) \\
\Longleftrightarrow\ & \quad \{ \text{ by reflexivity } \} \\
& \textsf{rec}\ o \lhd \textsf{map}^{\text{VEC}}\ (\textsf{rename}(\textsf{shift}\ g))\ (\textsf{map}^{\text{VEC}}\ (\textsf{subst}(\textsf{lift}\ f))\ os) \\
& \qquad = \textsf{rec}\ o \lhd \textsf{map}^{\text{VEC}}\ (\textsf{subst}(\textsf{lift}(\textsf{rename}\ g \circ f)))\ os \\
\Longleftrightarrow\ & \quad \{ \text{ by Proposition B.3 } [\textsf{rec}^{\textsf{map}}\text{-ext}] \text{ and } (\textsf{map}^{\text{VEC}}\ f\ v)(k) = f(v(k)) \ \} \\
& \forall i.\ \textsf{rename}\ (\textsf{shift}\ g)\ (\textsf{subst}\ (\textsf{lift}\ f)\ (os(i))) = \textsf{subst}\ (\textsf{lift}(\textsf{rename}\ g \circ f))\ (os(i)) \\
\Longleftrightarrow\ & \quad \{ \text{ by Proposition B.6 } [\textsf{lift}(\textsf{rename}\ g \circ f) \doteq \textsf{rename}(\textsf{shift}\ g) \circ \textsf{lift}\ f] \ \} \\
& \forall i.\ \textsf{rename}\ (\textsf{shift}\ g)\ (\textsf{subst}\ (\textsf{lift}\ f)\ (os(i))) = \textsf{subst}\ (\textsf{rename}(\textsf{shift}\ g) \circ \textsf{lift}\ f)\ (os(i)) \\
\Longleftrightarrow\ & \quad \{ \text{ by induction hypothesis (IH) } \} \\
& \top
\end{aligned}
$$

□

Proposition 10 Substitution commutes with renaming

Substitution commutes with renaming:

$$
\forall (t : \mathbb{C}_S(m)).\ \forall (f : \text{FIN}\, m \to \mathbb{C}_S(n)).\ \forall (g : \text{FIN}\, n \to \text{FIN}\, p).
\\ \textsf{subst}\ (g \circ f)\ t = \textsf{subst}\ g\ (\textsf{rename}\ f\ t).
$$

Proof. The proof is by induction on t and similar to the one given in Proposition B.9. □

Now, we specialize the previous results for the one-variable substitution function:

Proposition 11 Unbound variable substitution

Substitution of an unbound variable has no effect:

$$
\forall (t : \mathbb{C}_S(\textsf{suc}\, n)).\ \forall (u : \mathbb{C}_S(n)).\ \textsf{weaken}\, t[* := u] = t.
$$

Proof. Direct consequence of Proposition B.10 $[\textsf{subst}(g \circ f) \doteq \textsf{subst}\ g \circ \textsf{rename}\ f]$. □

The following result about lift is needed to complete the proof of the third monad law.

Proposition 12 Lifting distributes over substitution

The lift *operation distributes over substitution:*

$$
\forall (x : \text{FIN}\, m).\ \forall (f : \text{FIN}\, m \to \mathbb{C}_S(n)).\ \forall (g : \text{FIN}\, n \to \mathbb{C}_S(p)).
\\ \textsf{lift}\ (\textsf{subst}\ g \circ f)\ x = \textsf{subst}\ (\textsf{lift}\ g)\ (\textsf{lift}\ f\ x).
$$

Proof. Let $x : \text{FIN}\, x$, $f : \text{FIN}\, m \to \mathbb{C}_S(n)$ and $g : \text{FIN}\, n \to \mathbb{C}_S(p)$. We proceed by case-analysis on x:

- **Case 1.** Assume that $x = \textsf{zero}$.
  This case is trivial by reflexivity.
- **Case 2.** Assume that $x = \textsf{suc}\, x'$ where $x' : \text{FIN}\, m'$.

$$
\begin{aligned}
& \mathsf{lift}\ (\mathsf{subst}\, g \circ f)\ (\mathsf{suc}\, x') \\
\equiv\ & \quad \{ \text{ by definition } \} \\
& \mathsf{weaken}(\mathsf{subst}\ g\ (f(x'))) \\
=\ & \quad \{ \text{ by Proposition B.9 } [\mathsf{rename}\, g \circ \mathsf{subst}\, f \doteq \mathsf{subst}(\mathsf{rename}\, g \circ f)] \ \} \\
& \mathsf{subst}\ (\mathsf{lift}\, g \circ \mathsf{suc})\ (f(x')) \\
=\ & \quad \{ \text{ by Proposition B.10 } [\mathsf{subst}(g \circ f) \doteq \mathsf{subst}\, g \circ \mathsf{rename}\, f] \ \} \\
& \mathsf{subst}\ (\mathsf{lift}\, g)\ (\mathsf{weaken}(f(x'))) \\
\equiv\ & \quad \{ \text{ by definition } \} \\
& \mathsf{subst}\ (\mathsf{lift}\, g)\ (\mathsf{lift}\ f\ (\mathsf{suc}\, x')).
\end{aligned}
$$

□

Proposition 13 Associativity of substitution

The substitution operation is associative:

$$
\begin{aligned}
& \forall (f : \textsc{Fin}\, m \to \mathbb{C}_S(n)).\ \forall (g : \textsc{Fin}\, n \to \mathbb{C}_S(p)).\ \forall (t : \mathbb{C}_S(m)). \\
& \qquad \mathsf{subst}\ g\ (\mathsf{subst}\ f\ t) = \mathsf{subst}\ (\mathsf{subst}\ g \circ f)\ t.
\end{aligned}
$$

Proof. Let $f : \textsc{Fin}\, m \to \mathbb{C}_S(n)$, $g : \textsc{Fin}\, n \to \mathbb{C}_S(p)$ and $t : \mathbb{C}_S(m)$. We proceed by induction on the cyclic term term t:

- **Base step.** Assume that $t = \mathsf{var}\, x$ where $x : \textsc{Fin}\, m$.
  This case is trivial by reflexivity.
- **Inductive step.** Assume that $t = \mathsf{rec}\, o \lhd os$ where $os : \textsc{Vec}\ (\mathbb{C}_S(\mathsf{suc}\, n))\ (S.\, \mathsf{Ar}\, o)$.
  The induction hypothesis is given by:

$$
\begin{aligned}
& \forall k.\ \forall (n, p : \mathbb{N}).\ \forall (f : \textsc{Fin}(\mathsf{suc}\, m) \to \mathbb{C}_S(n)).\ \forall (g : \textsc{Fin}\, n \to \mathbb{C}_S(p)). \\
& \qquad \mathsf{subst}\ g\ (\mathsf{subst}\ f\ (os(k))) = \mathsf{subst}\ (\mathsf{subst}\, g \circ f)\ (os(k)).
\end{aligned} \tag{IH}
$$

$$
\begin{aligned}
& \mathsf{subst}\ g\ (\mathsf{subst}\ f\ (\mathsf{rec}\, o \lhd os)) = \mathsf{subst}\ (\mathsf{subst}\, g \circ f)\ (\mathsf{rec}\, o \lhd os) \\
\iff\ & \quad \{ \text{ by reflexivity } \} \\
& \mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{subst}(\mathsf{lift}\, g))\ \big(\mathsf{map}^{\textsc{Vec}}\ (\mathsf{subst}(\mathsf{lift}\, f))\ os\big) \\
& \qquad = \mathsf{rec}\, o \lhd \mathsf{map}^{\textsc{Vec}}\ (\mathsf{subst}(\mathsf{lift}(\mathsf{subst}\, g \circ f)))\ os \\
\iff\ & \quad \{ \text{ by Prop. B.2 [rec-ext] and Prop. A.11 } [(\mathsf{map}^{\textsc{Vec}}\ f\ v)(k) = f(v(k))] \ \} \\
& \forall i.\ \mathsf{subst}\ (\mathsf{lift}\, g)\ (\mathsf{subst}\ (\mathsf{lift}\, f)\ (os(i))) \\
& \qquad = \mathsf{subst}\ (\mathsf{lift}(\mathsf{subst}\, g \circ f))\ (os(i)) \\
\iff\ & \quad \{ \text{ by rewriting the induction hypothesis (IH) } \} \\
& \mathsf{subst}\ (\mathsf{subst}(\mathsf{lift}\, g) \circ \mathsf{lift}\, f)\ (os(i)) = \mathsf{subst}\ (\mathsf{lift}(\mathsf{subst}\, g \circ f))\ (os(i)) \\
\iff\ & \quad \{ \text{ by Proposition B.7 } [\textsc{subst-cong}] \ \} \\
& \top
\end{aligned}
$$

□

Proposition 14 Substitution commutes with one-variable substitution

The substitution operation commutes with one-variable substitution:

$$
\begin{aligned}
& \forall (f : \textsc{Fin}\, m \to \mathbb{C}_S(n)).\ \forall (t : \mathbb{C}_S(\mathsf{suc}\, n)).\ \forall (u : \mathbb{C}_S(m)). \\
& \qquad \mathsf{subst}\ f\ (t[* := u]) = (\mathsf{subst}\ (\mathsf{lift}\, f)\ t)[* := \mathsf{subst}\ f\ u]
\end{aligned}
$$

Proof. Let $f : \textsc{Fin}\, m \to \mathbb{C}_S(n)$, $t : \mathbb{C}_S(\mathsf{suc}\, n)$ and $u : \mathbb{C}_S(m)$. Then, we have:

$$(\mathsf{subst}\ (\mathsf{lift}\, f)\ t)[* := \mathsf{subst}\ f\ u] = \mathsf{subst}\ f\ (t[* := u])$$

$\iff$ { by definition of $_[* := _]$ }

$$\mathsf{subst}\ \sigma_{(\mathsf{subst}\, f\, u)}\ (\mathsf{subst}\ (\mathsf{lift}\, f)\ t) = \mathsf{subst}\ f\ (\mathsf{subst}\ \sigma_u\ t)$$

$\iff$ { by Proposition B.13 [$\mathsf{subst}\, g \circ \mathsf{subst}\, f \mathrel{\hat{=}} \mathsf{subst}(\mathsf{subst}\, g \circ f)$] }

$$\mathsf{subst}\ (\mathsf{subst}(\sigma_{\mathsf{subst}\, f\, u}) \circ \mathsf{lift}\, f)\ t = \mathsf{subst}\ f\ (\mathsf{subst}\ \sigma_u\ t)$$

$\iff$ { by Proposition B.7 [SUBST-CONG] }

$$\forall (x : \textsc{Fin}\, m).\ \mathsf{subst}\ \sigma_{\mathsf{subst}\ f\ u}\ (\mathsf{lift}\ f\ x) = \mathsf{subst}\ f\ (\sigma_u(x))$$

$\iff$

By case-analysis on x:

- **Case 1.** Assume that $x = \mathsf{zero}$.

  This case is trivial by reflexivity.

- **Case 2.** Assume that $x = \mathsf{suc}\, x'$ where $x' : \textsc{Fin}\, m'$.

$$\mathsf{subst}\ \sigma_{(\mathsf{subst}\, f\, u)}\ (\mathsf{lift}\ f\ (\mathsf{suc}\, x'))$$

$\equiv$ { by definition }

$$\mathsf{subst}\ \sigma_{(\mathsf{subst}\, f\, u)}\ (\mathsf{weaken}(f(x')))$$

$=$ { by Proposition B.10 [$\mathsf{subst}(g \circ f) \mathrel{\hat{=}} \mathsf{subst}\, g \circ \mathsf{rename}\, f$] }

$$\mathsf{subst}\ (\sigma_{(\mathsf{subst}\, f\, u)} \circ \mathsf{suc})\ (f(x'))$$

$\equiv$ { by definition }

$$\mathsf{subst}\ \mathsf{var}\ (f(x'))$$

$=$ { by Proposition B.8 [$\mathsf{subst}\ \mathsf{var} \mathrel{\hat{=}} \mathsf{id}$] }

$$f(x')$$

$\equiv$ { by definition }

$$\mathsf{subst}\ f\ (\sigma_u\ (\mathsf{suc}\, x'))$$

□

APPENDIX C

TRANSITIVE CLOSURE COMPUTATION

Let $T : \mathcal{U}$ be a transition system. We call S the carrier of T. Moreover, we assume that the type S has a decidable Leibniz equality and that every list l of states is $\succ$-accessible, *i.e.*, we have a proof W : $\textsc{Well-founded}\ (_\succ_)$.

First, we define the following binary relation $\sqsubset$ on pairs of list of states:

$$(l_1, l_2) \sqsubset (l_1', l_2') \overset{\text{def}}{\iff} l_1 \succ l_1' \lor (l_1 = l_1' \land \mathsf{length}\, l_2 < \mathsf{length}\, l_2').$$

The relation $_\sqsubset_$ is the lexicographic order induced by $_<_$ on $\mathbb{N}$ and $_\succ_$ on lists (see Def. A.38). Since both relations are well-founded, so is their lexicographic product (see [Pau86]).

We compute the reflexive-transitive closure induced by the nexttransition function as follows:

$$
\begin{aligned}
&\mathsf{succs} : (M : \textsc{List}\, S) \to (P : \textsc{List}\, S) \to \textsc{Acc}\ (_\sqsubset_)\ (P, M) \to \textsc{List}\, S\\
&\mathsf{succs}\ \ M\ \ [\,]\qquad\qquad _ \qquad\qquad \equiv\ M\\
&\mathsf{succs}\ \ M\ \ (p :: P)\ \ (\mathsf{acc\text{-}intro}\ f)\ \equiv\ \textbf{match}\ p \in^{?} M\ \textbf{with}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid\ \mathsf{yes}\, m\ \ \Rightarrow \mathsf{succs}\ M\ P\ (f(H))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid\ \mathsf{no}\, nm \Rightarrow \mathsf{succs}\ (p :: M)\ (\mathsf{Next}(p) +\!\!+ P)\ (f(H'))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{end}\\
&\qquad \textbf{where}\ \ H\ \ : (M, P) \sqsubset (M, p :: P)\\
&\qquad\qquad\qquad H'\ : (p :: M, \mathsf{Next}(p) +\!\!+ P) \sqsubset (M, p :: P)
\end{aligned}
$$

The terms for both H and H' are omitted but the proofs are straightforward. Indeed, for H is suffices to show that $M = M$ and $\mathsf{length}(P) < 1 + \mathsf{length}\, P = \mathsf{length}(p :: P)$ and for H', the proof of $\lceil p :: M \rceil$ is derived from nm.

In the function succs, the list M represents a set of *marked* states, *i.e.*, states that have already been visited. The list P is a set of states to be *processed*, *i.e.*, states that have yet to be visited. The map Next used above is defined as the list of immediate successors of a state.

$$
\begin{aligned}
&\mathsf{Next} : S \to \textsc{List}\, S\\
&\mathsf{Next}\ \ s\ \ \equiv\ \mathsf{tabulate}(\mathsf{next}\, s)
\end{aligned}
$$

It is straightforward to check that the function Next satisfied the following specification:

$$\forall (s, s' : S).\ s' \in \mathsf{Next}(s) \iff \exists k.\ \mathsf{next}\ s\ k = s'. \qquad \text{(Next-spec)}$$

Finally, the reflexive-transitive of a state $s : S$ is computed as follows:

$$
\begin{aligned}
&[_]^* : S \to \textsc{List}\, S\\
&[s]^*\ \equiv\ \mathsf{succs}\ [\,]\ [\,s\,]\ m
\end{aligned}
$$

where m is a proof that $\textsc{Acc}\ (_\sqsubset_)\ m$ which is derived from W. It remains to show the soundness and completeness of $[_]^*$.

Lemma 1 **Soundness of succs**

Let M, P be two lists of states. Moreover, let A be a proof that the pair (M, P) is $\sqsubset$-accessible and let s be a state. If,

$$\forall (s' : S).\ (s' \in M) \uplus (s' \in P) \to s \rightsquigarrow s',$$

then,

$$\forall (s' : S).\ s' \in \mathsf{succs}\ M\ N\ A \to s \rightsquigarrow s'.$$

Proof. Let $M, P : \textsc{List}\, S$ be a list of states and s be a state. Moreover, let $A : \textsc{Acc}\ (_\sqsubset_)\ (M, P)$ be a proof that (M, P) is $\sqsubset$-accessible. We proceed by induction on the accessibility proof A:

- **Base step.** Inductive type without a base constructor.
- **Inductive step.** Assume that $A = \mathsf{acc\text{-}intro}\ f$ where $f : \forall L.\ L \sqsubset (M, P) \to \textsc{Acc}\ (_\sqsubset_)\ L$. Moreover, let $s' : S$ be such that $I : s' \in \mathsf{succs}\ M\ P\ A$ and let $H : \forall s'.\ (s' \in M) \uplus (s' \in P) \to s \rightsquigarrow s'$. The induction hypothesis is given by:

$$\begin{aligned}&\forall (M', P' : \textsc{List}\, S).\ \forall (H : (M', P') \sqsubset (M, P)).\\ &\qquad (\forall s'.\ (s' \in M') \uplus (s' \in P') \to s \rightsquigarrow s') \to \forall s'.\ s' \in \mathsf{succs}\ M'\ P'\ (f(H)) \to s \rightsquigarrow s'.\end{aligned} \tag{IH}$$

  By case-analysis on P, we consider two cases:

  - **Case 1.** Assume that $P = [\,]$.
    We prove that $s \rightsquigarrow s'$ with hypothesis H. Thus, it remains to show that either $s' \in M$ or $s' \in P$. The first one holds and is given by hypothesis $I : s' \in \mathsf{succs}\ M\ [\,]\ A = M$.
  - **Case 2.** Assume that $P = p :: P'$ where $p : S$ and $P' : \textsc{List}\, S$.
    By case-analysis on $p \in^?\ M$, we consider two cases:
    - **Case 2.1.** Assume that $e : p \in M$.
      We prove that $s \rightsquigarrow s'$ by instantiating the induction hypothesis (IH) with $M' := M$, $P' := P'$ and hypothesis I. Thus, it remains to show that:

      $$\forall s'.\ (s' \in M) \uplus (s' \in P') \to s \rightsquigarrow s'.$$

      Let $s' : S$ and $U : (s' \in M) \uplus (s' \in P')$. By case-analysis on U, we consider two cases:
      - **Case 2.1.1.** Assume that $e : s' \in M$.
        We conclude that $s \rightsquigarrow s'$ by applying e to hypothesis H.
      - **Case 2.1.2.** Assume that $e : s' \in P'$.
        Since $s' \in P'$, we derive a proof E' such that $s' \in p :: P' = P$. Thus, we conclude that $s \rightsquigarrow s'$ by applying E' to H.
    - **Case 2.2.** Assume that $e : p \notin M$.
      We prove that $s' \rightsquigarrow s$ by instantiating the induction hypothesis (IH) with $M' := p :: M$, $P := \mathsf{Next}(p) +\!\!+ P'$ and hypothesis I. Thus, it remains to show that:

      $$\forall s'.\ (s' \in p :: M) \uplus (s' \in \mathsf{Next}(p) +\!\!+ P') \to s \rightsquigarrow s'.$$

      Let $s' : S$ and $U : (s' \in p :: M) \uplus (s' \in \mathsf{Next}(p) +\!\!+ P')$. By case-analysis on U, we consider two cases:

- **Case 2.2.1.** Assume that $e : s' \in p :: M$.
  Here, we have yet two cases to consider.
  If $E : s' \approx_S p$, then we have a proof E' of $s' \approx_S p \in p :: P' = P$ and thus, we conclude that $s \rightsquigarrow s'$ with by applying E to hypothesis H.
  If $E : s' \in M$, we conclude that $s \rightsquigarrow s'$ by applying E to hypothesis H.
- **Case 2.2.2.** Assume that $e : s' \in \mathsf{Next}(p) +\!\!+ P'$.
  Here, we have yet two cases to consider.
  If $E : s' \in \mathsf{Next}(p)$, then there exist a k and a proof E' that $\mathsf{next}\ p\ k = s'$. Since we have $p \in p :: P'$, we can use this fact along with hypothesis H to derive that $s \rightsquigarrow p$. Finally, by transitivity of $_\rightsquigarrow_$, we have $s \rightsquigarrow p$ and $p \rightsquigarrow \mathsf{next}\ p\ k$ hence hence $s \rightsquigarrow s'$ (by rewriting E').
  If $E : s' \in P'$, we derive a proof E' that $s' \in p :: P'$ and thus, we conclude that $s \rightsquigarrow s'$ by applying E' to hypothesis H . □

In order to prove the completeness property, we first define two invariants and prove that some properties are preserved.

Invariant 1. The first invariant is given by:

$\textsc{Inv}_1 : (M\ \ P : \textsc{List}\,S) \to \textsc{Prop}$

$\textsc{Inv}_1\ \ M\ \ P\ \equiv\ \forall (s, s' : S).\ s \in M \to s' \notin M \to s' \in \mathsf{Next}(s) \to s' \in P.$

Proposition 1 — Invariant 1 preservation

Let M, P be two lists of states. We have,

(i) $\forall (p : S).\ p \in M \to \textsc{Inv}_1\ M\ (p :: P) \to \textsc{Inv}_1\ M\ P,$

(ii) $\forall (p : S).\ p \notin M \to \textsc{Inv}_1\ M\ (p :: P) \to \textsc{Inv}_1\ (p :: M)\ (\mathsf{Next}(p) +\!\!+ P)$

Proof. Let $M, P : \textsc{List}\,S$ be a list of states and let s, s' be two states.

(i) Let $I : \textsc{Inv}_1\ M\ (p :: P)$, $E : s \in M$, $N : s' \notin M$, $N' : s' \in \mathsf{Next}(s)$ and $H : p \in M$. We have to show that $s' \in P$. We instantiate hypothesis I with E, N and N', thus we have a proof $H' : s' \in p :: P$. Furthermore, we consider two cases by elimination on H':

- **Case 1.1.** Assume that $e : s' = p$.
  We can show that this case leads to a contradiction. Indeed, we have $N : s' = p \notin M$ and $H : p \in M$. Contradiction.
- **Case 1.2.** Assume that $e : s' \in P$.
  The proof is immediate with e.

(ii) Let $I : \textsc{Inv}_1\ M\ (p :: P)$, $E : s \in (p :: M)$, $N : s' \notin (p :: M)$, $N' : s' \in \mathsf{Next}(s)$ and $H : p \notin M$. We have to show that $s' \in \mathsf{Next}(p) +\!\!+ P$. We consider two cases by elimination of E:

- **Case 2.1.** Assume that $e : s = p$.
  To show that $s' \in \mathsf{Next}(p) +\!\!+ P$, it suffices to show that $s' \in \mathsf{Next}(p)$. This is given by hypothesis $N' : s' \in \mathsf{Next}(s) = \mathsf{Next}(p)$.
- **Case 2.2.** Assume that $e : s \in M$.
  Clearly, from N, we can derive a proof E' of $s' \notin M$. We instantiate hypothesis I with e, E' and N'. Thus, we obtain a proof $H' : s' \in (p :: P)$. By elimination of H':
  - **Case 2.2.1.** Assume that $e' : s' = p$.
    We can show that this case leads to a contradiction. Indeed, we have that $N : s' = p \notin (p :: M)$ but clearly, $p \in (p :: M)$. Contradiction.

- **Case 2.2.2.** Assume that $e' : s' \in P$.
  To show that $s' \in \mathsf{Next}(p) +\!\!+ P$, it suffices to show that $s' \in P$ which is given by hypothesis e'. □

Invariant 2. The second invariant is defined as follows:

$$\textsc{Inv}_2 : (M \ P : \textsc{List}\, S) \to \textsc{Prop}$$
$$\textsc{Inv}_2 \ \ M \ \ P \ \equiv \forall (s, s' : S).\ s \ {}^{M}\!\!\rightsquigarrow^{P} \ s'.$$

where the relation $_{}^{-}\!\!\rightsquigarrow^{-}_$ is defined inductively as follows:

$$\textbf{inductive } _{}^{-}\!\!\rightsquigarrow^{-}_ \ (M P : \textsc{List}\, S) : S \to S \to \textsc{Type}$$
$$\rightsquigarrow\text{-}\varepsilon : \forall s.\ s \in M \to s \ {}^{M}\!\!\rightsquigarrow^{P} \ s$$
$$\rightsquigarrow\text{-step} : \forall s, s', s''.\ s \in M \to s' \in \mathsf{next}(s) \to s' \ {}^{M}\!\!\rightsquigarrow^{P} \ s'' \to s \ {}^{M}\!\!\rightsquigarrow^{P} \ s''$$
$$\rightsquigarrow\text{-P} : \forall s, s'.\ s \notin M \to s \in P \to s \rightsquigarrow s' \to s \ {}^{M}\!\!\rightsquigarrow^{P} \ s'$$

Proposition 2 Path normalization

Let M, P be two lists of states. We have,

(i) $\forall (s, s' : S).\ s \in M \to \textsc{Inv}_1 \ M \ P \to s \rightsquigarrow s' \to s \ {}^{M}\!\!\rightsquigarrow^{P} \ s'$,

(ii) $\forall (s, s' : S).\ s \in P \to \textsc{Inv}_1 \ M \ P \to s \rightsquigarrow s' \to s \ {}^{M}\!\!\rightsquigarrow^{P} \ s'$.

Proof. Let $M, P : \textsc{List}\, S$ be two lists of states and let s, s' be two states. Morever, let $I : \textsc{Inv}_1 \ M \ P$ and let $H : s \rightsquigarrow s'$.

(i) Let $E : s \in M$. We proceed by induction on H:

- **Base step.** Assume that $e : s = s'$.
  We use constructor $\rightsquigarrow$-ε, thus it remains to prove that $s \in M$ which is given by E.
- **Inductive step.** Assume that there exists an index $k : \textsc{Fin}(\mathsf{rank}\, s)$ and a proof $e : s[k] \rightsquigarrow s'$.
  The induction hypothesis is given by:

$$s[k] \in M \to s[k] \ {}^{M}\!\!\rightsquigarrow^{P} \ s'. \tag{IH}$$

  By case-analysis on $s[k] \in^{?} M$, we consider two cases:

  - **Case 1.1.** Assume that $C : s[k] \in M$.
    We use constructor $\rightsquigarrow$-step, thus it remains to show that there exists a state s'' such that $s \in M$, $s'' \in \mathsf{Next}(s)$ and $s'' \ {}^{M}\!\!\rightsquigarrow^{P} \ s'$. We pick $s'' := s[k]$. The proof of $s \in M$ is given by E, the proof of $s'' \in \mathsf{Next}(s)$ is derived from (Next-spec), and finally, the proof of $s'' \ {}^{M}\!\!\rightsquigarrow^{P} \ s'$ is obtained from C applied to the induction hypothesis (IH).
  - **Case 1.2.** Assume that $C : s[k] \notin M$.
    We use constructor $\rightsquigarrow$-step, thus it remains to show that there exists a state s'' such that $s \in M$, $s'' \in \mathsf{Next}(s)$ and $s'' \ {}^{M}\!\!\rightsquigarrow^{P} \ s'$. We pick $s'' := s[k]$. The proof of $s \in M$ is given by E, the proof of $s'' \in \mathsf{Next}(s)$ is derived from (Next-spec). It still remains to show that $s'' \ {}^{M}\!\!\rightsquigarrow^{P} \ s'$. First, we prove that $C' : s[k] \in P$. This is obtained by applying E , C and $H' : s[k] \in \mathsf{Next}(s)$ (derived from (Next-spec)) to hypothesis I. Finally, to prove $s'' \ {}^{M}\!\!\rightsquigarrow^{P} \ s'$, we use constructor $\rightsquigarrow$-P, thus it remains to show that $s[k] \notin M$, $s[k] \in P$ and $s[k] \rightsquigarrow s'$. All of these are given by C, C' and e respectively.

(ii) Let $e : s \in P$. By case-analysis on $s \in^{?} M$, we consider two cases:

- **Case 2.1.** Assume that $C : s \in M$.
  We conclude that $s \ {}^{M}\!\!\rightsquigarrow^{P} \ s'$ with (i).

- **Case 2.2.** Assume that $C : s \notin M$.
  We use constructor $\rightsquigarrow$-P, thus it remains to show that $s \notin M$, $s \in P$ and $s \rightsquigarrow s'$. All of these are given by E, C and H respectively. □

Proposition 3 Invariant 2 preservation

Let M, P be two lists of states. We have,

(*i*) $\forall (p : S).\ p \in M \to \textsc{Inv}_1\ M\ (p :: P) \to \forall s, s'.\ s \ {}^{M}\!\!\rightsquigarrow^{p :: P}\ s' \to \textsc{Inv}_2\ M\ P\ s\ s'$,

(*ii*) $\forall (p : S).\ p \notin M \to \textsc{Inv}_1\ M\ (p :: P)$
$\to \forall s, s'.\ s \ {}^{M}\!\!\rightsquigarrow^{p :: P}\ s' \to \textsc{Inv}_2\ (p :: M)\ (\mathsf{Next}(p) +\!\!+ P)\ s\ s'$.

Proof. Let $M, P : \textsc{List}\, S$ be a two lists of states and let s, s' be two states.

(*i*) Let $I_1 : \textsc{Inv}_1\ M\ (p :: P)$, $I_2 : s \ {}^{M}\!\!\rightsquigarrow^{p :: P}\ s'$ and $H : p \in M$.
We have to show that $s \ {}^{M}\!\!\rightsquigarrow^{P}\ s'$. We proceed by induction on I_2:

- **Base step.** Assume that $e : s = s'$ and $e' : s \in M$.
  We use constructor $\rightsquigarrow$-ε, thus it suffices to show that $s \in M$ which is given by e'.
- **Base step.** Assume that $e : s \notin M$, $e' : s \in (p :: P)$ and $e'' : s \rightsquigarrow s'$.
  By case-analysis on e', we consider two cases:
  - **Case 1.1.** Assume that $C : s = p$.
    We have $e : s = p \notin M$ and $H : p \in M$. Contradiction.
  - **Case 1.2.** Assume that $C : s \in P$. We use constructor $\rightsquigarrow$-P, thus it suffices to show that $s \notin M$, $s \in P$ and $s \rightsquigarrow s'$. All of these are given by e, C, and e'' respectively.
- **Inductive step.** Assume that there exists $s'' : S$ such that $e : s \in M$ and $e' : s'' \in \mathsf{Next}(s)$.
  The induction hypothesis is given by:

$$s'' \ {}^{M}\!\!\rightsquigarrow^{P}\ s'. \tag{IH}$$

  We have to show that $s \ {}^{M}\!\!\rightsquigarrow^{P}\ s'$. We use constructor $\rightsquigarrow$-step, thus it suffices to show that there exists a state t such that $s \in M$, $t \in \mathsf{Next}(s)$, and $t \ {}^{M}\!\!\rightsquigarrow^{P}\ s'$. For t, we pick the state given by s''. All remaining goals are proven by e, e' and (IH) respectively.

(*ii*) Let $I_1 : \textsc{Inv}_1\ M\ (p :: P)$, $I_2 : s \ {}^{M}\!\!\rightsquigarrow^{p :: P}\ s'$ and $H : p \notin M$.
We have to show that $s \ {}^{p :: M}\!\!\rightsquigarrow^{\mathsf{Next}(p) +\!\!+ P}\ s'$. We proceed by induction on I_2:

- **Base step.** Assume that $e : s = s'$ and $e' : s \in M$.
  We use constructor $\rightsquigarrow$-ε, thus it suffices to show that $s \in (p :: M)$ which is given by e'.
- **Base step.** Assume that $e : s \notin M$, $e' : s \in (p :: P)$ and $e'' : s \rightsquigarrow s'$.
  By case-analysis on e', we consider two cases:
  - **Case 2.1.** Assume that $C : s = p$.
    To show that $s \ {}^{p :: M}\!\!\rightsquigarrow^{\mathsf{Next}(p) +\!\!+ P}\ s'$, we use Proposition C.2 (*i*). Thus, it remains to show that $s \in (p :: M)$, $\textsc{Inv}_1\ (p :: P)\ (\mathsf{Next}(p) +\!\!+ P)$ and $s \rightsquigarrow s'$. The first one is obtained from C, the second one is given by applying I_1 and H Proposition C.1 (*ii*), and the last one is given by e''.
  - **Case 2.2.** Assume that $C : s \in P$.
    To show that $s \ {}^{p :: M}\!\!\rightsquigarrow^{\mathsf{Next}(p) +\!\!+ P}\ s'$, we use Proposition C.2 (*ii*). Thus, it remains to show that $s \in \mathsf{Next}(p) +\!\!+ P$, $\textsc{Inv}_1\ (p :: P)\ (\mathsf{Next}(p) +\!\!+ P)$ and $s \rightsquigarrow s'$. The first one is obtained from C, the second one is given by applying I_1 and H to Proposition C.1 (*ii*). Finally, to prove $s \in \mathsf{Next}(p) +\!\!+ P$, it suffices to show that $s \in P$ which is given by C.

- **Inductive step.** Assume that there exists $s'' : S$ such that $e : s \in M$ and $e' : s'' \in \mathsf{Next}(s)$. The induction hypothesis is given by:

$$s'' \;{}^{p :: M}\!\!\rightsquigarrow^{\mathsf{Next}(p) +\!\!+ P}\; s'. \tag{IH}$$

We have to show that $s \;{}^{p :: M}\!\!\rightsquigarrow^{\mathsf{Next}(p) +\!\!+ P}\; s'$. We use constructor $\rightsquigarrow$-step, thus it suffices to show that there exists a state t such that $s \in (p :: M)$, $t \in \mathsf{Next}(s)$, and $t \;{}^{p :: M}\!\!\rightsquigarrow^{\mathsf{Next}(p) +\!\!+ P}\; s'$. For t, we pick the state given by s''. All remaining goals are proven by e, e' and (IH) respectively. □

Lemma 2 Completeness of succs

Let M, P be two lists of states. Moreover, let A be a proof that the pair (M, P) is $\sqsubset$-accessible and let s be a state. If,

$$\text{INV}_1\ M\ P$$

then,

$$\forall (s' : S).\ s \;{}^{M}\!\!\rightsquigarrow^{P}\; s' \to s' \in \mathsf{succs}\ M\ N\ A.$$

Proof. Let $M, P : \text{LIST}\ S$ be a list of states and let s be a state. Moreover, let $A : \text{ACC}\ (_\sqsubset_)\ (M, P)$ be a proof that (M, P) is $\sqsubset$-accessible. We proceed by induction on the accessibility proof A:

- **Base step.** Inductive type without a base constructor.
- **Inductive step.** Assume that $A = \text{acc-intro}\ f$ where $f : \forall L.\ L \sqsubset (M, P) \to \text{ACC}\ (_\sqsubset_)\ L$. Moreover, let $s' : S$ such that $I : s \;{}^{M}\!\!\rightsquigarrow^{P}\; s'$ and let $H : \text{INV}_1\ M\ P$. The induction hypothesis is given by:

$$\begin{aligned}&\forall (M', P' : \text{LIST}\ s).\ \forall (H : (M', N') \sqsubset (M, P)).\\ &\quad \text{INV}_1\ M'\ P' \to \forall s'.\ s \;{}^{M'}\!\!\rightsquigarrow^{P'}\; s' \to s' \in \mathsf{succs}\ M'\ P'\ (f(H)).\end{aligned} \tag{IH}$$

By case-analysis on P, we consider two cases:

- **Case 1.** Assume that $P = [\,]$.
  We have to show that $s' \in \mathsf{succs}\ M\ [\,]\ A = M$. We proceed by induction on $I : s \;{}^{M}\!\!\rightsquigarrow^{[\,]}\; s'$:
  - **Base step.** Assume that $e : s = s'$ and $e' : s \in M$.
    We conclude that $s' = s \in M$ with hypothesis e'.
  - **Base step.** Assume that $e : s \notin M$, $e' : s \in P$ and $e'' : s \rightsquigarrow s'$.
    Because P is the empty list, the proof e' leads to a contradiction.
  - **Inductive step.** Assume that there exists $s'' : S$ such that $e : s \in M$, $e' : s'' \in \mathsf{Next}(s)$ and $e'' : s'' \;{}^{M}\!\!\rightsquigarrow^{[\,]}\; s'$.
    The induction hypothesis is given by:

$$s' \in M. \tag{IH$'$}$$

    We conclude that $s' \in M$ with (IH$'$).

- **Case 2.** Assume that $P = p :: P'$ where $p : S$ and $P' : \text{LIST}\ P$.
  By case-analysis on $p \in^{?} M$, we consider two cases:
  - **Case 2.1.** Assume that $e : p \in M$.
    We prove that $s' \in \mathsf{succs}\ M\ P'\ _$ by instantiating the induction hypothesis (IH) with $M' := M$ and $P := P'$. Thus, it remains to show that both invariants are preserved. The proof of $\text{INV}_1\ M\ P'$ is given by applying e to Proposition C.1 (i). The proof of $s \;{}^{M}\!\!\rightsquigarrow^{P'}\; s'$ is given by applying e to Proposition C.3 (i).

- **Case 2.2.** Assume that $e : p \notin M$.
  We prove that $s' \in \mathsf{succs}\ (p :: M)\ (\mathsf{Next}(p) +\!\!+ P')\ _$ by instantiating the induction hypothesis (IH) with $M' := M$ and $P := P'$. Thus, it remains to show that both invariants are preserved. The proof of $\textsc{Inv}_1\ (p :: M)\ (\mathsf{Next}(p) +\!\!+ P')$ is given by applying e to Proposition C.1 (ii). The proof of $s\ {}^{p::M}\!\!\rightsquigarrow^{\mathsf{Next}(p)+\!\!+P'}\ s'$ is obtained by Proposition C.3 (ii) applied with e. □

Theorem 1 **Soundness of** $[_]^*$

Let s, s' be two states. If, $s' \in [s]^*$ then, $s \rightsquigarrow s'$.

Proof. Let s, s' be two states and $H : s' \in [s]^*$. We apply Lemma C.1, thus it remains to show that the invariant holds initially. Indeed, we have to prove that:

$$\forall (s' : S).\ (s \in [\,]) \uplus (s \in [s]) \to s \rightsquigarrow s'.$$

Let $s' : S$. In order to prove that $s \rightsquigarrow s'$, it suffices to show that $s \in [s]$ which holds trivially. □

Theorem 2 **Completeness of** $[_]^*$

Let s, s' be two states. If, $s \rightsquigarrow s'$ then, $s' \in [s]^*$.

Proof. Let s, s' be two states and $H : s \rightsquigarrow s'$. We apply Lemma C.2, thus it remains to show that the invariant holds initially. Indeed, we have to prove that:

$$(\forall (s, s' : S).\ s \in [\,] \to s' \notin [\,] \to s' \in \mathsf{next}(s) \to s' \in [s]) \wedge (s\ {}^{[\,]}\!\!\rightsquigarrow^{[s]}\ s').$$

- $\forall (s, s' : S).\ s \in [\,] \to s' \notin [\,] \to s' \in \mathsf{Next}(s) \to s' \in [s]$:
  Let s, s' be two states and $M : s \in [\,]$. Clearly, the type of M is empty. Contradiction.
- $s\ {}^{[\,]}\!\!\rightsquigarrow^{[s]}\ s'$:
  We use constructor $\rightsquigarrow$-P thus, it remains to prove $s \notin [\,]$, $s \in [s]$ and $s \rightsquigarrow s'$. The first two proofs are trivial and the last one is given by hypothesis H. □

APPENDIX D

EXTENDED ABSTRACT

Introduction

Les assistants de preuve sont des outils développés par les informaticiens dans le but de faciliter le raisonnement formel. En ce sens, ces outils fournissent un cadre formel permettant l'expression d'énoncés ainsi que de leurs propriétés. Ensuite, en utilisant les règles de preuve de la logique sous-jacente, les preuves des théorèmes sont données à l'ordinateur pour vérification. La théorie des types dépendants est un formalisme pouvant servir comme fondation alternative à la théorie des ensembles pour exprimer les mathématiques. Plus généralement, de telles théories des types offrent un cadre unifié permettant la définition de structures de données, des programmes manipulant ces structures et l'expression de leurs propriétés. Concrètement, cela signifie que le même langage est utilisé à la fois pour définir les programmes, pour énoncer leurs spécifications, et enfin exprimer la preuve de leur correction. La notion de constructivité peut être exploitée afin d'extraire un programme à partir du terme de preuve de sa spécification. Certaines théories intègrent de puissants principes de raisonnement tels que l'induction pour raisonner sur des objets finis, ou la co-induction pour raisonner sur les objets infinis.

Les graphes sont une structure de données omniprésente en informatique. Ils sont utilisés pour donner une sémantique aux logiques, pour modéliser des calculs, ou encore pour décrire des relations entre objets. Le problème consistant à représenter la structure de graphes en théorie des types dépendants peut s'avérer difficile. En effet, le principal obstacle est que, dans leur forme la plus générale, les structures de graphe peuvent être cycliques. En conséquence, le principe d'induction, qui est le principe de base du raisonnement en théorie des types dépendants, échoue à capturer une telle circularité. En effet, les types inductifs sont basés sur le principe de bonne-fondation. Néanmoins, il est bien connu que les approches basées sur les coalgèbres sont plus adaptées pour raisonner sur des structures non bien-fondées [Rut00], c'est-à-dire, des structures infinies ou présentant une forme de circularité. Dans ce contexte, les types co-inductifs offrent un cadre naturel pour définir et raisonner sur ce type d'objets. L'idée principale réside dans le fait que les structures cycliques peuvent être vues comme des arbres infinis en procédant à un dépliage infini de leurs cycles. Une telle approche a été poursuivie dans [Pic12] où a été considéré le problème de la représentation de graphes dans l'assistant de preuve Coq.

Cependant, ce type de raisonnement ne s'applique pas naturellement aux structures circulaires finies. En effet, une représentation co-inductive de ces structures circulaires finies entraîne une perte d'information : la propriété de finitude. En conséquence, les types co-inductifs ne nous permettent pas de pouvoir faire la distinction entre structures circulaires finies et structures infinies. Dès lors, il est intéressant de considérer la problématique de la caractérisation du sous-ensemble des termes co-inductifs possédant cette propriété de finitude, ainsi que le problème de sa préservation. D'autre part, sachant que l'on se place dans le cadre d'une théorie constructive, il existe des caractérisations de la propriété de finitude [SC10, FU15] qui ne sont pas toutes équivalentes. En effet, certaines caractérisations requièrent le fait que l'égalité soit décidable, c'est-à-dire, tel qu'il existe une procédure calculatoire justifiant cette égalité. De plus, il peut être intéressant de

considérer des variantes plus faibles de la propriété de finitude capturant des sous-ensembles plus larges de ces structures circulaires finies, tout en étant suffisamment fortes pour préserver la calculabilité. Les arbres réguliers [Cou83] sont un des exemples fondamentaux d'arbres infinis possédant une propriété de finitude. Intuitivement, les arbres réguliers sont un sous-ensemble des arbres infinis ayant la propriété que l'ensemble de leurs sous-arbres distincts est fini. Par exemple, ces arbres sont utilisés pour caractériser les solutions de systèmes finis d'équations.

Plus concrètement, les structures circulaires finies interviennent lorsque l'on considère le dépliage sémantique d'opérateurs d'itération présents dans la syntaxe de langages. On peut citer les langages fonctionnels contenant la construction let rec servant à exprimer la récursivité générale, les boucles while dans les langages impératifs, ou encore l'opérateur μ dans les algèbres de processus (resp. de certaines théories des types) dénotant les processus récursifs (resp. les types récursifs). Sémantiquement, de tels termes sont généralement *identifiés à leur dépliage*. En conséquence, le raisonnement sur ces termes doit prendre en compte cette identification. De plus, les opérations définies sur ces termes doivent être invariantes par dépliage. En revanche, le raisonnement inductif ne capture pas en général ce principe d'identification. Cependant, s'il l'on se place du point de vue des termes infinis, l'opération de dépliage devient *transparente*. Dans ce cas, il peut être intéressant de réutiliser le cadre des types co-inductifs qui par définition identifient les termes à leurs dépliages. Par exemple, ce cadre pourrait permettre de définir des opérations par co-itération ou permettre de raisonner sur celles-ci par co-induction.

L'objet de cette thèse est de proposer une théorie mécanisée des arbres réguliers dans la théorie des types dépendants. En particulier, nous attacherons une grande importance à donner une définition formelle des arbres réguliers prenant en compte les spécificités de la théorie des types dépendants dans laquelle nous travaillons, tout en restant entièrement constructif et définitionnel (sans axiome supplémentaire).

Plan et résumé des contributions

Dans cette section, nous fournissons un aperçu des chapitres composant cette thèse accompagné d'un résumé de chacune des contributions.

Chapitre I [Préliminaires]. Ce chapitre est une brève introduction à la théorie des types dépendants : le calcul des constructions (co)inductives. Cette théorie servira comme base pour les fondements mathématiques de tous les résultats présentés dans les chapitres ultérieurs. En particulier, nous introduisons les types inductifs et co-inductifs ainsi que les principes de raisonnement qui leur sont associés. De plus, nous donnons une présentation des types co-inductifs au travers d'une sémantique catégorique, c'est-à-dire, vus comme des coalgèbres (faiblement) finaux.
Ensuite, nous introduisons diverses définitions *non-équivalentes* des setoids finis. Ces définitions constituent une simple généralisation des définitions de types finis tels qu'elles sont trouvées dans la littérature. Cette généralisation vers les setoids est motivée par le fait que la logique sous-jacente dans laquelle nous travaillons ne possède pas de types quotients. De plus, la relation de bissimilarité sur les types co-inductifs n'est pas une congruence. Dès lors, il n'est pas possible de substituer des termes bissimilaires dans des contextes arbitraires.

La principale contribution de ce chapitre est une bibliothèque mécanisée des setoids finis. En particulier, nous considérons des formes faibles de setoids finis dans le sens où l'équivalence du setoid n'est pas nécessairement décidable.

La présentation des types co-inductifs s'appuie en partie sur ces travaux:

[AS14] Benedikt Ahrens and Régis Spadotti. Terminal semantics for codata types in intensional Martin-Löf type theory. In *TYPES*, volume 39 of *LIPIcs*, pages 1–26. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2014

[ACS15] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In *TLCA*, volume 38 of *LIPIcs*, pages 17–30. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2015

Chapitre II [Arbres réguliers]. Ce chapitre est divisé en deux parties. Dans un premier temps, nous formalisons le type des arbres réguliers comme une restriction des types co-inductifs sur une signature arbitraire. Cette formalisation des arbres réguliers utilise la hiérarchie des setoids finis introduite dans le chapitre 1. En particulier, nous ne prenons pas comme hypothèse le fait que le type des symboles de fonctions (décrit par une signature) possède une égalité décidable. Néanmoins, cette définition est suffisamment constructive pour prouver que les arbres réguliers sont clos pour la relation de sous-arbre.
Dans un second temps, nous définissons une syntaxe des arbres réguliers au moyen de termes cycliques. Cette représentation syntaxique est ensuite prouvée correcte et complète par rapport à la caractérisation co-inductives des arbres réguliers. De plus, nous donnons une axiomatisation correcte et complète de l'équivalence entre termes cycliques. Enfin, nous prouvons que la fonction associant un terme cyclique à un arbre régulier est un isomorphisme de setoids.

Les principales contributions de ce chapitre sont résumées dans la table suivante :

Type des arbres réguliers Reg_S	Proposition 2.14
Clôture pour la relation sous-arbre	Théorème 2.5
Type des termes cycliques $\mathbb{C}_S$	Proposition 2.15
Correction de la représentation syntaxique	Théorème 2.7
Complétude de la représentation syntaxique	Théorème 2.9
$[\![_]\!] : \mathbb{C}_S \to \text{Reg}_S$ est un isomorphisme de setoid	Théorème 2.12

Chapitre III [Transducteurs d'arbres]. Dans ce chapitre, nous étudions et définissons à l'aide de schémas (co)récursifs, différentes classes de morphismes d'arbres préservant la propriété de régularité. Dans ce but, nous utilisons le formalisme des transducteurs d'arbres comme outil pour modéliser la syntaxe abstraite des définitions de fonctions co-récursives comme en Agda ou en Coq. En particulier, nous étudions diverses sortes de transducteurs d'arbres d'expressivité croissante. Finalement, chaque terme de cette syntaxe abstraite est réifié en un morphisme entre arbres réguliers.

Les principales contributions de ce chapitre sont résumées dans la table ci-dessous :

Transducteur (d'arbres) descendant gardé	Proposition 3.9
Dérivation d'un morphisme d'arbre préservant la régularité	Théorème 3.2
Transducteur (d'arbres) descendant avec règles ε	Proposition 3.11
Dérivation d'un morphisme d'arbre préservant la régularité	Théorème 3.3

Transducteur (d'arbres) descendant à observation finie	Proposition 3.13
Dérivation d'un morphisme d'arbre préservant la régularité	Théorème 3.4
Transducteur (d'arbres) descendant binaire	Proposition 3.16
Dérivation d'un morphisme d'arbre préservant la régularité	Théorème 3.5

Chapitre IV [Applications]. Dans ce chapitre, nous présentons des exemples exploitant la théorie mécanisée des arbres réguliers qui a été développée dans les chapitres précédents. En prenant l'exemple du calcul de l'opération de composition parallèle d'une algèbre de processus telle que CSP, nous montrons comment un problème de terminaison peut être reformulé en un problème de productivité. La composition parallèle est définie comme un transducteur d'arbres descendant. Ensuite, nous considérons des problèmes de décidabilité sur le type des arbres réguliers. En effet, certaines propriétés indécidables sur les arbres infinis deviennent décidables sur ce fragment. Dans ce but, nous donnons une interprétation du μ-calcul coalgébrique sur le type des arbres réguliers. En particulier, nous montrons que la relation de bissimilarité est décidable sur les arbres réguliers via une traduction vers un problème de vérification de modèles.

Les principales contributions de ce chapitre sont détaillées dans la table suivante :

Définition de la composition parallèle définie par un transducteur d'arbres	Section 4.1
Interprétation du μ-calcul sur les arbres réguliers	Section 4.2
Décidabilité de la satisfiabilité des formules du μ-calcul (model-checking)	Théorème 4.1
Décidabilité de la bissimilarité	Théorème 4.2

Un sous-ensemble des résultats présentés dans les chapitres 3 à 4 a été publié dans

[Spa15] Régis Spadotti. A mechanized theory of regular trees in dependent type theory. In *ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 405–420. Springer, 2015

CHAPITRE UN

Préliminaires

Dans ce chapitre, nous introduisons la théorie des types utilisée dans cette thèse. Tout d'abord, nous donnons une introduction générale de la théorie des types dépendants qui nous servira comme cadre formel pour énoncer toutes les définitions et tous les résultats des chapitres qui suivent. Ensuite, nous étudions diverses notions de types finis et de setoids finis dans un cadre constructif. Les setoids finis constituent le principal outil théorique pour caractériser la propriété de régularité sur les arbres infinis. De plus, nous introduisons également la notion de signature afin d'abstraire les propriétés structurelles des arbres. Enfin, nous présentons une étude de la sémantique des types co-inductifs selon une approche coalgébrique.

I.1. Théorie des types dépendants

Dans cette section, nous introduisons la théorie des types qui nous servira à formaliser les différents résultats présentés dans cette thèse. Cette théorie des types se nomme le calcul des constructions (co-)inductives [CH88, CP88, Gim96] et est implémentée dans l'assistant de preuve Coq. Dans les sections suivantes, nous introduisons succinctement les différentes constructions syntaxiques présentes dans cette théorie.

I.1.1. Types de bases et opérateurs

Nous supposons que le lecteur est déjà familier à une théorie des types dépendants telle qu'elle peut être implémentée dans les assistants de preuves comme Agda ou Coq. C'est pourquoi, nous ne rappelons pas dans le détail les différentes règles de typage et de réductions. Néanmoins, nous introduisons les constructions syntaxiques ainsi que les notations que nous utiliserons car celles-ci diffèrent de celles utilisées en Coq.

Tout d'abord, nous supposons à la base de la hiérarchie, l'existence d'une sorte Prop utilisée pour représenter l'univers des propositions logiques. Ensuite, vient la sorte $\textsc{Type}_i$ indexée par un nombre naturel i, de sorte que Prop soit un sous-type de $\textsc{Type}_0$ et que, plus généralement, $\textsc{Type}_i$ soit un sous-type de $\textsc{Type}_{i+1}$. L'univers des propositions est supposé imprédicatif alors que les univers des types sont eux supposés prédicatifs. En Coq, les niveaux d'univers peuvent être inférés ainsi il n'est pas nécessaire de les mentionner explicitement. Nous suivons cette convention dans le reste de cette thèse. Il est important de noter que l'univers des propositions permet de représenter des termes dépourvus de contenu calculatoire. Dès lors, il existe une séparation stricte entre les sortes Prop et Type de sorte que les termes dans Prop ne peuvent pas être, en général, éliminés pour obtenir des termes dans Type.

Nous admettons l'existence de types de base comme le type $\mathbb{0} : \textsc{Type}$ (vide) qui n'est habité par aucun élement ; le type $\mathbb{1} : \textsc{Type}$ (unité) qui est habité par un unique élément tt. Ces types

de base ont un équivalent dans la sorte des propositions. Ainsi, nous notons $\bot$: PROP le type représentant la proposition toujours fausse et $\top$: PROP, la proposition qui est toujours vraie. De plus, nous supposons aussi l'existence d'opérateurs sur les types. Par exemple, l'opérateur $A \to B$ permet de construire le type des fonctions ou encore l'opérateur $A \times B$ (resp. $A \uplus B$) permet de construire le produit (resp. la somme disjointe) de types. Les équivalents dans la sorte PROP sont donnés par $A \wedge B$ et $A \vee B$ désignant respectivement la conjonction et la disjonction de propositions. L'ensemble de ces types de bases est résumé dans le tableau 1.1. Enfin, nous admettons l'existence de l'opérateur $\prod$ qui permet de construire des fonctions avec types dépendants ($\prod_{a:A} B(a)$) ainsi que l'opérateur $\sum$ qui permet de construire les paires dépendantes ($\sum_{a:A} B(a)$).

I.1.2. Types inductifs et co-inductifs

Les types inductifs offrent la possibilité d'étendre la théorie avec de nouveaux types. En particulier, ils permettent l'écriture de définitions récursives. Pour chaque définition inductive, l'assistant de preuve COQ génère un principe d'induction permettant de raisonner sur ces types. Par exemple, les nombres naturels sont définis inductivement de la façon suivante :

inductive $\mathbb{N}$: TYPE
| zero : $\mathbb{N}$
| suc : $\mathbb{N} \to \mathbb{N}$.

Le principe d'induction associé aux nombres naturels prend la forme suivante :

$$\mathbb{N}\text{-rect}\ \frac{P : \mathbb{N} \to \mathrm{TYPE} \qquad P(\mathsf{zero}) \qquad \forall (n:\mathbb{N}).\ P(n) \to P(\mathsf{suc}\, n)}{\forall (n:\mathbb{N}).\ P(n)}.$$

Les types inductifs peuvent aussi être indexés par d'autres types. Dans ce cas, on parle de familles inductives. Ainsi le type représentant les listes de taille fixe peut être introduit comme une famille inductive indexée par leur taille :

inductive VEC (A : TYPE) : $\mathbb{N} \to$ TYPE
| [] : VEC A zero
| _::_ : $\{n : \mathbb{N}\} \to A \to$ VEC A $n \to$ VEC A (suc n).

Les termes des types inductifs peuvent être éliminés en utilisant la construction syntaxique appelée filtrage par motif (pattern-matching) **match** ... **with** ... **end**. Il est aussi possible de procéder à cette élimination en utilisant des définitions de fonctions par équations. Dès lors, chaque équation constitue un cas à considérer dans la définition. Ensuite, chacune d'entre elles doit être vérifiée afin de s'assurer qu'aucun cas n'a été omis et que chaque terme a été correctement éliminé.

Les types enregistrements dépendants permettent aussi d'introduire des définitions inductives avec des fonctions de projections prédéfinies. Ainsi, le type des paires dépendantes peut être défini par un enregistrement dépendant de la façon suivante :

record Σ (A : TYPE) (B : $A \to$ TYPE) : TYPE
constructor _,_
[proj_1 : A
[proj_2 : $B(\mathsf{proj}_1)$.

Les deux fonctions de projections, ici proj_1 et proj_2 sont dérivées automatiquement et permettent d'extraire respectivement, le premier et le second élément d'une paire dépendante.

Dans le cadre d'une définition de fonction récursive, il est nécessaire de s'assurer que la fonction termine pour chaque entrée. Si l'on autorisait une forme de récursivité générale, il serait alors possible de montrer l'inconsistence de la théorie, vue comme un système logique. En effet, il serait très facile de construire une preuve de faux (type $\bot$) en définissant une fonction récursive bouclant indéfiniment. L'assistant de preuve Coq se base sur l'observation de schémas syntaxiques pour s'assurer de la terminaison des fonctions. En revanche, lorsque le schéma de définitions de la fonction ne s'inscrit pas dans ce cadre, il est nécessaire de fournir une preuve explicite de terminaison. Généralement, cela consiste à identifier une relation d'ordre bien-fondée (mesure) entre les éléments en entrée.

Les types co-inductifs sont duaux aux types inductifs. Intuitivement, ils permettent de produire des termes par application infinie des constructeurs. Par opposition, les termes des types inductifs sont construits par application finie des constructeurs. Les types co-inductifs sont introduits de la même manière que les types inductifs. Par exemple, le type des flots (Stream) se définit de la façon suivante :

coinductive Stream $(A : \text{Type}) : \text{Type}$
  | $_::_ : A \to \text{Stream}\ A \to \text{Stream}\ A.$

Le type des flots peut être utilisé, par exemple, pour modéliser le comportement infini de systèmes réactifs. Les types co-inductifs peuvent aussi être utilisés pour représenter des structures cycliques comme des automates ou des graphes.

I.1.3. Egalité, type identité et setoids

Nous introduisons deux types d'égalité afin de permettre un raisonnement équationnel. La première égalité est dite définitionnelle dans le sens où elle se situe au niveau de la méta-théorie. Nous notons $A \equiv B$ quand les termes A et B sont définitionnellement égaux. L'égalité définitionnelle prend en compte les différentes règles de réduction du calcul des constructions (co-)inductives (β, η, δ, ...).

Afin de pouvoir raisonner sur l'égalité entres termes (ou types) dans la théorie elle-même, il est nécessaire de procéder à une internalisation de l'égalité définitionnelle au moyen de l'égalité de Leibniz (type identité) :

inductive $_=_\ (A : \text{Type})\ (x : A) : A \to \text{Prop}$
  | refl $: x = x.$

Il n'y a qu'un unique constructeur refl (reflexivité) de sorte qu'une preuve de reflexivité n'est possible que si les termes sont définitionnellement égaux. Il est important de noter que le type identité est une congruence. En effet, il est possible de définir une fonction de substitution permettant la réécriture dans un contexte arbitraire. L'égalité de Leibniz est dite intensionnelle (par opposition à extensionnelle) car elle ne satisfait pas la règle suivante :

$$\textit{reflection rule}\ \frac{A = B}{A \equiv B}.$$

De plus, nous introduisons le type h-prop (mere propositions). Le type A est un h-prop s'il est habité et vérifie la propriété suivante :

$$\forall (p, q : A).\ p = q.$$

Parfois, l'égalité de Leibniz est trop fine et il peut être nécessaire de pouvoir identifier des termes à équivalence près (pour une notion adéquate d'équivalence). Par exemple, les fonctions

peuvent être considérées comme équivalentes lorsqu'elles sont extensionnellement égales. En théorie des ensembles, cette notion peut être réalisée au moyen d'ensembles quotients. Cependant, la théorie des types que nous utilisons ne permet pas la construction de types quotients. Néanmoins, une internalisation possible de ces types est obtenu par l'introduction des setoids [BCP03]. Un setoid est défini comme un type équipé d'une relation d'équivalence (réflexive, symmétrique, transitive) de sorte que les termes dans le setoid sont identifiés par cette relation. Les setoids seront particulièrement utiles pour raisonner sur les types co-inductifs à bissimilarité près. L'implémentation des setoids en Coq est basée sur les classes de types [SO08] introduisant une forme de polymorphisme ad hoc. En outre, cela autorise une forme de surcharge. Ainsi, une même notation peut être utilisée et spécialisée pour chaque type. Dans le contexte des setoids, nous utilisons la notation $_\approx_$ pour désigner la relation d'équivalence associée au setoid.

I.2. Types et setoids finis

Les types et setoids finis sont très utilisés tout au long de cette thèse. En particulier, ils nous seront utiles pour définir la propriété de régularité des arbres infinis. Dans ce qui suit, nous introduisons succinctement les différentes versions de types et setoids finis que nous avons étudiées. Les résultats présentés ci-après sont une généralisation immédiate aux setoids des résultats disponibles dans [FU15, SC10, GP15] pour les types finis.

I.2.1. Types finis canoniques

Les types finis canoniques sont indexés par leur cardinal (leur nombre d'éléments) et correspondent à la représentation en théorie des types de l'ensemble $\{m : m < n\}$. Ils peuvent aussi être introduits par une famille inductive (type Fin).

I.2.2. Setoids finis

Un type A (Finite A) est dit fini s'il existe une bijection avec un type fini canonique, pour un cardinal donné. Cette notion se généralise aux setoids lorsque la bijection respecte les relations d'équivalence des setoids. Nous montrons que les types et setoids finis sont clos pour les opérations de produit, somme et exponentiation.

I.2.3. Setoids énumérables

Une autre variante de types ou setoids finis est donnée par l'existence d'une fonction d'indexation des éléments. On parle alors de types (setoids) énumérables (FinitelyIndexed). Nous montrons que cette variante est équivalente à la caractérisation précédente (type/setoid fini). De plus, nous prouvons également que cette variante est équivalente à la définition plus classique de type fini, donnée par l'existence d'une liste sans élément dupliqué qui contient tous les habitants du type.

I.2.4. Setoids faiblement finiment indexés

Dans le contexte des setoids, nous avons étudié une version plus faible de la fonction d'indexation dans le sens où, celle-ci peut ne pas respecter la relation d'équivalence du setoid. Intuitivement, deux éléments équivalents dans le setoid peuvent être indexés par des éléments différents. On parle alors d'un setoid faiblement finiment indexé (WFI). Nous avons prouvé que les setoids énumérables sont faiblement finiment indéxés mais que la réciproque n'est pas vraie constructivement.

I.2.5. Setoids à fonction d'exploration

Les fonctions d'exploration (EXPLORE) constituent une autre manière de caractériser les types ou setoids finis. La principale différence avec les versions précédemment introduites est que ces fonctions sont caractérisées par des principes d'introduction et d'élimination. La notion de setoid à fonction d'exploration est équivalente à la notion de setoid faiblement finiment indexé.

I.2.6. Setoids sans suite injective

Enfin, la dernière variante étudiée définit un setoid comme fini dès lors qu'il n'existe pas de fonction injective depuis $\mathbb{N}$. On parle de setoid sans suite injective (STREAMLESS). Nous avons prouvé que cette notion est plus faible que la notion de setoid faiblement finiment indexé.

I.3. Signatures et signatures indexées

Dans cette section, nous introduisons les définitions de signatures ainsi que leur variante indéxée. Puis, nous définissons l'algèbre de termes (resp. co-termes) librement engendrée par une signature.

I.3.1. Signatures

Une signature (ou alphabet gradué) est définie comme la donnée d'un ensemble de symboles ou opérateurs et d'une fonction associant à chaque opérateur son arité. Le type des signatures est défini par un enregistrement dépendant $\mathsf{Op} \triangleleft \mathsf{Ar}$ où la première fonction de projection $\mathsf{Op} : \textsc{Type}$ associe le type des opérateurs et la seconde projection $\mathsf{Ar} : \mathsf{Op} \to \mathbb{N}$ donne la fonction d'arité.

Les signatures induisent un endofoncteur sur TYPE appelé foncteur d'extension.

$$\begin{aligned}&\mathsf{ext} : \textsc{Sig} \to \textsc{Type} \to \textsc{Type}\\&\mathsf{ext}\ \ S\ \ X\ \ \equiv\ \textstyle\sum_{o:S.\mathsf{Op}} \textsc{Vec}\ X\ (S.\mathsf{Ar}\ o).\end{aligned}$$

Le point fixe de ce foncteur peut être obtenu par une définition de type inductif. Cela permet, par exemple, de raisonner sur les types inductifs dans la théorie elle-même via leur signature.

I.3.2. Signatures indexées

Les signatures indexées (multi-sortées) constituent une généralisation des signatures en ajoutant des contraintes de typage (sortes) au niveau des arguments des opérateurs. Le type des signatures indexées est introduit par un enregistrement dépendant indexé par un type représentant les sortes S. La première fonction de projection $\mathsf{Op} : \textsc{Type}$ renvoie le type des opérateurs et la seconde projection $\mathsf{Typ} : \mathsf{Op} \to \textsc{List}\, S$ associe à chaque opérateur une liste de sortes pour chacun de ces arguments. Par ailleurs, l'arité de l'opérateur est donnée par la longueur de cette liste.

Comme pour les signatures, on peut construire le foncteur d'extension ainsi que son point fixe, correspondant dans ce cas aux familles inductives.

I.3.3. Algèbre de termes

Étant donné une signature S, le type de tous les termes construits à partir de l'ensemble des opérateurs de S est appelé l'algèbre de termes induite par S. Il est défini inductivement comme suit :

$$\begin{aligned}&\textbf{inductive}\ \mathbb{T}\ (S : \textsc{Sig}) : \textsc{Type}\\&\quad \mid\ _\triangleleft_ : (o : S.\mathsf{Op}) \to \textsc{Vec}\ \mathbb{T}_S\ (S.\mathsf{Ar}\ o) \to \mathbb{T}_S.\end{aligned}$$

L'ajout de variables de type X construit la monade libre induite par S:

inductive $\mathbb{T}_S$ $(X : \textsc{Type}) : \textsc{Type}$
| $_ \triangleleft _ : (o : S.\mathsf{Op}) \to \textsc{Vec}\ (\mathbb{T}_S(X))\ (S.\mathsf{Ar}\, o) \to \mathbb{T}_S(X)$
| $\mathsf{var} : X \to \mathbb{T}_S(X)$.

Notons qu'il est possible d'internaliser le type des variables dans la signature. La structure de monade du type $\mathbb{T}_S(X)$ est donnée par une opération subst de substitution possédant de bonnes propriétés algébriques ainsi que d'une opération (var) d'injection des variables dans $\mathbb{T}_S(X)$.

I.4. Coalgèbres et types co-inductifs

Dans cette section, nous étudions les relations entre les structures coalgébriques et les types co-inductifs.

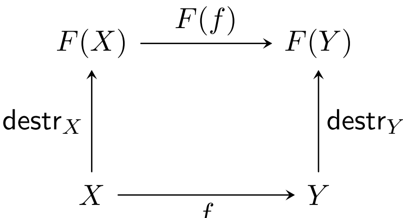
I.4.1. Sémantique des types co-inductifs

Nous rappellons qu'en théorie des catégories une coalgèbre pour un foncteur F est donnée par une paire $(X, \alpha : X \to F(X))$ où X est le support du coalgèbre et α est la fonction d'observation. Le type des co-termes induit par une signature S est défini co-inductivement comme suit :

coinductive $\textsc{co}\mathbb{T}$ $(S : \textsc{Sig}) : \textsc{Type}$
| $_ \blacktriangleleft _ : (o : S.\mathsf{Op}) \to (\textsc{Fin}(S.\mathsf{Ar}\ \ o) \to \textsc{co}\mathbb{T}_S) \to \textsc{co}\mathbb{T}_S$.

En particulier, le type $\textsc{co}\mathbb{T}_S$ possède une structure de S-coalgèbre.

Un morphisme de coalgèbres est donné par une fonction $f : X \to Y$ entre les supports des coalgèbres, de sorte que le diagramme suivant commute :



De plus, une coalgèbre X est dite terminale lorsque pour toute coalgèbre Y, il existe un unique morphisme $X \to Y$. On peut montrer que le type $\textsc{co}\mathbb{T}_S$ est une coalgèbre faiblement terminale. En effet, nous avons seulement l'existence du morphisme mais pas son unicité.

I.4.2. Égalité et types co-inductifs

Précédemment, nous avons mentionné que le type $\textsc{co}\mathbb{T}_S$ peut être vu comme une coalgèbre faiblement terminale. Cependant, il est intéressant de discuter pourquoi cette coalgèbre n'est pas terminale. Intuitivement, la raison principale est due au fait que l'égalité de Leibniz n'identifie pas les fonctions extensionnellement égales. En effet, pour être terminale, la coalgèbre $\textsc{co}\mathbb{T}_S$ doit satisfaire un principe d'extensionnalité appelé bissimilarité. La relation de bissimilarité est définie comme la plus grande relation de bissimulation. En particulier, lorsque la coalgèbre est terminale alors la relation de bissimilarité coïncide avec la relation d'égalité. On parle alors de principe de co-induction [Rut00].

CHAPITRE DEUX

Arbres réguliers

Dans ce chapitre, nous proposons une définition formelle du type des arbres réguliers [Cou83] sur une signature abstraite. Tout d'abord, nous donnons une caractérisation des arbres réguliers basée sur une restriction de la représentation co-inductive des arbres infinis. Intuitivement, cette restriction consiste à ne considérer que les arbres infinis tels que l'ensemble de leur sous-arbres est fini. Dans ce but, nous utilisons une grande partie des notions sur les setoids finis introduites dans le chapitre précédent. De plus, nous avons pour objectif de considérer une notion de sous-arbre qui soit aussi indépendante que possible de la représentation des arbres considérée. Bien que la notion de finitude choisie pour caractériser l'ensemble des sous-arbres soit faible[1], nous montrons qu'elle est néanmoins suffisante pour obtenir, de façon constructive, la propriété de fermeture suivante : « un sous-arbre d'un arbre régulier est encore régulier. »

Ensuite, nous introduisons une syntaxe ayant pour but de caractériser tous les arbres réguliers. Cette syntaxe est basée sur une représentation inductive des arbres infinis vus comme des termes cycliques [GHUV06]. Ici, les cycles dans les termes sont encodés au moyen de lieurs. Contrairement à la caractérisation basée sur les types co-inductifs, les cycles sont explicites. Le principal avantage d'une approche syntaxique est que les arbres ainsi obtenus sont réguliers *par construction*. Enfin, nous montrons que cette représentation syntaxique est correcte et complète par rapport à la représentation basée sur la caractérisation co-inductive des arbres réguliers.

II.1. Une caractérisation co-inductive

Dans cette section, nous avons pour objectif de donner une caractérisation co-inductive des arbres réguliers [Cou83]. Dans ce but, nous introduisons le type des arbres infinis sur une signature au moyen d'une définition co-inductive. Nous étudions précisément la notion d'isomorphisme d'arbres. Ensuite, nous donnons une définition abstraite de l'ensemble des sous-arbres d'un arbre infini sur des coalgèbres dites non graduées. Enfin, en considérant la restriction finie de cet ensemble, nous obtenons une caractérisation des arbres réguliers.

II.1.1. Arbres infinis

Le type des arbres infinis $\text{co}\mathbb{T}_S$ sur une signature est introduit au moyen une définition co-inductive :

$$\textbf{coinductive}\ \text{co}\mathbb{T}\,(S : \text{Sig}) : \text{Type}$$
$$\mid\ _ \blacktriangleleft _ : (o : S.\mathsf{Op}) \to (\text{Fin}(S.\,\mathsf{Ar}\,o) \to \text{co}\mathbb{T}_S) \to \text{co}\mathbb{T}_S.$$

[1] Par exemple, nous ne supposons pas que l'égalité sur les opérateurs de la signature soit décidable.

Le type $\mathrm{co}\mathbb{T}_S$ possède une structure de S-coalgèbre. En effet, nous pouvons observer la racine d'un arbre via la fonction root : $\mathrm{co}\mathbb{T}_S \to S$ ainsi que ses sous-arbres directs, donnés par la fonction br : $(t : \mathrm{co}\mathbb{T}_S) \to \mathrm{Fin}(S.\,\mathsf{Ar}(\mathsf{root}\, t)) \to \mathrm{co}\mathbb{T}_S$.

L'égalité de Leibniz est une relation trop forte pour identifier tous les arbres infinis isomorphes. En conséquence, nous introduisons la relation de bissimilarité définie comme la plus grande relation de bissimulation. Les relations de bissimulations sont caractérisées par une propriété de clotûre : si deux arbres sont en relation alors leurs racines sont égales et chacun de leurs sous-arbres sont eux-même en relation. La relation de bissimilarité $_\sim_$ peut être introduite au moyen d'une définition co-inductive :

```
coinductive _∼_ {S : SIG} : coT_S → coT_S → PROP
  | ∼-intro : ∀{t1, t2}. (e : root t1 = root t2) → (∀k. br t1 k ∼ br t2 (e*(k))) → t1 ∼ t2.
```

Ensuite, nous prouvons le *principe de co-induction* permettant d'établir que deux arbres infinis sont bissimilaires dès lors qu'il existe une relation de bissimulation les contenant. En s'appuyant sur ce résultat, nous montrons que la relation de bissimilarité à profondeur n :

$$t_1\, \mathcal{D}\, t_2 \overset{\text{def}}{\iff} \forall (n : \mathbb{N}).\ t_1 \sim_n t_2$$

est une relation de bissimulation. Cette propriété nous sera particulièrement utile pour établir des résultats de bissimilarité par induction.

II.1.2. Coalgèbres non graduées

Nous introduisons ici le type des coalgèbres non graduées. Le but étant ici de définir de façon abstraite la notion chemin et de successeur indépendamment du choix de la représentation des arbres infinis. Dès lors, les arbres infinis peuvent être considérés comme des systèmes de transitions. La signature dite non graduée est introduite comme suit :

$$\mathsf{U} := \mathbb{N} \triangleleft \mathsf{id}.$$

Le type des U-coalgèbres non graduées induit par U, noté $\mathcal{U}$, est défini par une paire dépendante :

$$\mathcal{U} := \sum_{X,\approx} X \longrightarrow \mathsf{U}(X).$$

Ici, nous considérons que les supports des colgèbres ne sont pas des types mais des setoids. La structure de coalgèbre est donnée par deux fonctions : rank : $\forall\{X : \mathcal{U}\}.\ X \to \mathbb{N}$ associant à chaque élément du colgèbre un rang, c'est-à-dire, le nombre de ses successeurs directs, et la fonction next : $\forall\{X : \mathcal{U}\}.\ \forall(s : X).\ \mathrm{Fin}(\mathsf{rank}\, s) \to X$ permettant leur observation.

II.1.3. Chemins et successeurs dans les coalgèbres non graduées

En s'appuyant sur les coalgèbres non graduées introduites précédemment, nous définissons la notion de chemin dans un système de transitions de même que le type des successeurs.

Concrètement, nous définissons le type des chemins en deux étapes. Dans un premier temps, nous introduisons le type des chemins dans un système de transitions comme la donnée d'une liste de nombres :

$$\mathrm{Path} := \mathrm{List}\,(\textstyle\sum_{n:\mathbb{N}} \mathrm{Fin}\ n).$$

Dans un souci de simplification, les nombres sont représentés ici par des paires où le premier élément désigne le rang n associé à l'état courant et le second élément désigne la transition effectivement choisie dans l'ensemble fini induit par n. Dans un second temps, nous introduisons le

type des chemins valides dans un système de transitions T, représenté par une coalgèbre non graduée, à partir d'un état s :

$$\mathbb{P}(s) := \{ p : \textsc{Path} \mid p \in \mathscr{P}_T(s) \} .$$

où $\mathscr{P}_T(s) : \textsc{Path} \to \textsc{Prop}$ est un prédicat caractérisant l'ensemble des chemins valides dans T partant de s. Ici, un chemin est dit valide si la liste des nombres qui composent le chemin respectent les fonctions d'observation (rank et next).

Le type des successeurs d'un état est obtenu en s'appuyant sur la notion de chemin. En effet, étant donné un chemin $p : \textsc{Path}$ et un état s, il est possible de calculer l'état successeur, noté $s|_p$, en itérant la fonction d'observation next le long de p. Le type représentant l'ensemble des successeurs de s, noté $s_{\rightsquigarrow}$, est défini comme suit :

$$s_{\rightsquigarrow} := \sum_{s':S} s \rightsquigarrow s'.$$

où $s \rightsquigarrow s'$ est la relation réflexive et transitive successeur.

Finalement, nous montrons qu'il existe un isomorphisme de setoids entre le setoid des chemins valides partant de s et le setoid représentant les états accessibles depuis s :

$$s_{\rightsquigarrow} \cong \mathbb{P}(s).$$

II.1.4. Type fini des successeurs

Dans cette section, nous définissons le type des états d'un système de transitions tels que l'ensemble de leurs états accessibles (successeurs) est fini. Cette notion est similaire à la propriété de régularité des arbres infinis.

Formellement, étant donné un système de transitions $T = (S, out_S)$, le type des états ayant un ensemble d'états accessibles finis est introduit comme suit :

$$\textsc{FSucc}_{(S,out_S)} := \sum_{s:S} \textsc{SuccFinite}(s)$$

où $\textsc{SuccFinite}(s)$ désigne le fait que le type des successeurs de s, $s_{\rightsquigarrow}$, est faiblement finiment indexé. Plus concrètement :

$$\textsc{SuccFinite}\, s := \mathrm{WFI}(s_{\rightsquigarrow}).$$

Enfin, nous montrons une propriété de clôture : le setoid FSucc peut être muni d'une structure de U-coalgèbre. Intuitivement, cette propriété énonce le fait que l'ensemble des états accessibles est clos par la relation successeur. La difficulté de cette preuve est due au fait que les setoids faiblement finiment indexés ne sont pas, en général, clos par inclusion.

II.1.5. Arbres réguliers

Dans cette section, nous considérons les arbres réguliers comme une instance des résultats généraux présentés précédemment.

Tous d'abord, étant donné une signature S, nous montrons que le setoid $(\textsc{co}\mathbb{T}_S, \sim)$ peut être muni d'une structure de U-coalgèbre. Dès lors, la notion de régularité coïncide avec la notion d'ensemble d'états accessibles fini. Plus concrètement :

$$\textsc{Regular}(t) := \textsc{SuccFinite}(t).$$

De même, le type des arbres réguliers sur la signature S, noté $\textsc{Reg}_S$, est obtenu par :

$$\textsc{Reg}_S := \textsc{FSucc}_{\textsc{co}\mathbb{T}_S}.$$

Enfin, les résultats génériques se transportent sur cette instance particulière et nous obtenons ainsi la propriété de clôture suivante : les sous-arbres d'un arbre régulier sont eux-mêmes réguliers.

II.2. Une syntaxe pour les arbres réguliers

Dans cette section, nous considérons une autre approche pour définir les arbres réguliers. L'objectif ici est d'introduire une syntaxe afin de caractériser les arbres réguliers *par construction*. L'avantage par rapport à l'approche présentée dans la section précédente, que nous pouvons qualifier de sémantique, réside dans le fait qu'il n'est pas nécessaire de fournir une preuve de finitude pour montrer qu'un arbre régulier. En effet, nous prouvons la correction et la complétude de cette représentation syntaxique des arbres réguliers par rapport à l'approche sémantique.

II.2.1. Termes cycliques

L'idée principale conduisant à la caractérisation syntaxique des arbres réguliers consiste à représenter les cycles de façon explicite. En effet, précédemment, les arbres infinis sont introduits au moyen d'équations co-récursives en exploitant le fait que les types co-inductifs permettent de définir des termes infinis. En revanche, dans ce cadre, les cycles deviennent implicites car sémantiquement identifiés à leur dépliage. Afin de conserver cette information, nous devons trouver un moyen de capturer les cycles dans la syntaxe. Dans ce but, nous utilisons l'approche développée par [GHUV06] qui consiste à encoder les cycles au moyen de lieurs. Syntaxiquement, l'arbre infini est représenté par son arbre couvrant et les lieurs sont utilisés pour marquer des points de retour implicites modélisant ainsi les cycles.

Concrètement, nous introduisons le type des termes cycliques $\mathbb{C}_S$ sur une signature S par le type inductif suivant :

inductive $\mathbb{C}\,(S : \textsc{Sig})\,(n : \mathbb{N}) : \textsc{Type}$
 | $\mathsf{var} : \textsc{Fin}\, n \to \mathbb{C}_S(n)$
 | $\mathsf{rec}_\lhd_ : (o : S.\mathsf{Op}) \to \textsc{Vec}\,(\mathbb{C}_S(\mathsf{suc}\ n))\,(\textsc{Fin}(S.\,\mathsf{Ar}\ o)) \to \mathbb{C}_S(n).$

Nous utilisons ici une représentation des lieurs dite anonyme car les variables sont implicitement renommées de façon canonique (de Bruijn). L'un des avantages de cette représentation est que l'α-équivalence, identifiant les termes α-convertibles, coïncide avec l'égalité de Leibniz. De plus, le type $\mathbb{C}_S$ est indexé par un nombre naturel n représentant le nombre de variables libres dans un terme. Ici, les variables sont introduites au moyen du constructeur var et sont choisies canoniquement dans l'ensemble fini induit par $\textsc{Fin}\, n$. L'autre constructeur, rec, est utilisé pour représenter le lieur. Il permet aussi d'injecter les opérateurs donnés par la signature. Il est important de noter que le nombre de variables libres est augmenté d'une unité sous le lieur. Ainsi, les règles de typages garantissent qu'une variable libre devient liée par application de rec.

II.2.2. Propriétés syntaxiques des termes cycliques

Dans cette section, nous rappelons les propriétés syntaxiques des termes cycliques. La finalité consiste à montrer que le type $\mathbb{C}_S : \mathbb{N} \to \textsc{Type}$ possède une structure de monade. Ainsi, nous définissons sur les termes cycliques une opération de renommage :

$$\mathsf{rename} : (\textsc{Fin}\ m \to \textsc{Fin}\ n) \to \mathbb{C}_S(m) \to \mathbb{C}_S(n),$$

ainsi qu'une opération de substitution :

$$\mathsf{subst} : (\textsc{Fin}\ m \to \mathbb{C}_S(n)) \to \mathbb{C}_S(m) \to \mathbb{C}_S(n).$$

En utilisant, l'opération de substitution générale, nous introduisons également l'opération consistant à substituer la variable liée dans le terme t par un terme u :

$$t[* := u] := \mathsf{subst}\, \sigma_u\, t$$

où σ_u désigne la fonction de substitution.

II.2.3. Sémantique des termes cycliques

Le type des termes cycliques $\mathrm{co}\mathbb{T}_S$ décrit une syntaxe des arbres réguliers. Afin de prouver la correction de cette caractérisation syntaxique, nous définissons une fonction sémantique associant à chaque terme cyclique sa représentation co-inductive en terme d'arbre infini :

$$\begin{array}{l}
[\![_]\!] : \mathbb{C}_S \to \mathrm{co}\mathbb{T}_S \\
[\![\ \mathsf{var}\,()\ ]\!] \\
[\![\ \mathsf{rec}\, o \lhd os\ ]\!] \ \equiv\ o \blacktriangleleft \lambda k.\ [\![\, os(k)[* := \mathsf{rec}\, o \lhd os]\,]\!].
\end{array}$$

Notons que la fonction sémantique n'est définie que sur les termes cycliques clos, c'est-à-dire, sans variable libre. Sémantiquement, un terme cyclique est identifié à son dépliage. Ici, le dépliage est défini en substituant la variable liée du terme cyclique par le terme lui-même.

Nous proposons une autre fonction sémantique utilisant la notion de fermeture, introduite par le type inductif suivant :

$$\begin{array}{l}
\textbf{inductive}\ \textsc{Closure}\,(S : \textsc{Sig}) : \mathbb{N} \to \textsc{Type} \\
\quad |\ \ [\,] : \textsc{Closure}_S(\mathsf{zero}) \\
\quad |\ \ _{::}_ : \forall (n : \mathbb{N}).\ \mathbb{C}_S(n) \to \textsc{Closure}_S(n) \to \textsc{Closure}_S(\mathsf{suc}\ n).
\end{array}$$

Intuitivement, le type des fermetures (Closure) permet de représenter des environnements garantissant, par construction, que toute variable libre peut être finalement associée à un terme clos. Dès lors, il est possible de définir une fonction (close-term) qui, étant un terme cyclique et une fermeture, produit un terme clos obtenu après substitution des variables libres par leur définition donnée par la fermeture :

$$\mathsf{close\text{-}term} : \{n : \mathbb{N}\} \to \mathbb{C}_S(n) \to \textsc{Closure}_S(n) \to \mathbb{C}_S.$$

Enfin, la seconde fonction sémantique associant un terme cyclique à son dépliage est définie comme suit[1] :

$$\begin{array}{l}
[\![_]\!]_ : \{n : \mathbb{N}\} \to \mathbb{C}_S(n) \to \textsc{Closure}_S(n) \to \mathrm{co}\mathbb{T}_S \\
[\![\ \mathsf{var\ zero}\ ]\!]_{(t :: \rho)} \ \equiv\ [\![\, t\,]\!]_\rho \\
[\![\ \mathsf{var}(\mathsf{suc}\ n)\ ]\!]_{(t :: \rho)} \ \equiv\ [\![\ \mathsf{var}(n)\,]\!]_\rho \\
[\![\ \mathsf{rec}\, o \lhd os\ ]\!]_\rho \ \equiv\ o \blacktriangleleft \lambda k.\ [\![\, os(k)\,]\!]_{(\mathsf{rec}\, o \lhd os\, ::\, \rho)}.
\end{array}$$

Dans le cas de la première variante, la fonction opère sur des termes clos. Ainsi, il est nécessaire de déplier le terme avant l'appel co-récursif afin de préserver cet invariant. En ce qui concerne la seconde variante, le dépliage ne s'observe que lorsque l'on atteint effectivement une variable libre. En conséquence, contrairement à la première variante, la seconde définition opère sur des termes avec des variables libres en mémorisant leur définition dans une fermeture.

Finalement, nous montrons que les deux fonctions sémantiques coïncident. Concrètement, nous prouvons que les arbres infinis obtenus par application des fonctions sémantiques sur un terme cyclique sont bissimilaires.

[1] Définition simplifiée s'affranchissant de la condition de garde syntaxique (voir chapitre 2).

II.2.4. Sous-termes des termes cycliques

L'objectif de cette section est d'introduire les différents résultats intermédiaires qui nous serons utiles pour établir la correction de la représentation syntaxique des arbres réguliers sous forme de termes cycliques.

Tout d'abord, nous montrons que le type des termes cycliques possède une structure de U-coalgèbre. Ainsi, nous pouvons réutiliser les définitions et propriétés de la section II.1. Dès lors, un terme cyclique peut être vu comme un système de transitions sur lequel les notions de chemins et de successeurs sont bien définies.

Ensuite, nous prouvons que l'ensemble des successeurs, c'est-à-dire, l'ensemble des sous-termes d'un terme cyclique est fini. Cette preuve s'appuie sur un lemme d'élimination consistant à décomposer et normaliser un chemin dans un terme cyclique obtenu par une substitution. Enfin, nous montrons que pour tout terme cyclique $t : \mathbb{C}_S(n)$, le setoid des sous-termes de $t{\leadsto}$ est faiblement finiment indexé.

II.2.5. Correction et complétude de la représentation syntaxique

Dans cette section, nous justifions le fait que la définition inductive des termes cycliques est une syntaxe pour les arbres réguliers. Dans ce but, nous devons prouver d'une part que l'image d'un terme cyclique par la fonction sémantique est un arbre régulier, et d'autre part, que tout arbre régulier peut être représenté, de manière équivalente, sous la forme d'un terme cyclique.

Formellement, la propriété de correction s'énonce comme suit :

$$\forall (t : \text{co}\mathbb{T}).\ \textsc{Regular}\ [\![\, t \,]\!].$$

La preuve s'obtient par induction sur le terme cyclique t en exploitant les résultats démontrés dans la section précédente (finitude de $t{\leadsto}$).

La preuve de complétude se décompose de deux parties. Tout d'abord, nous définissons une fonction, nommée $\mathsf{sem\text{-}inv} : \textsc{Reg}_S \to \mathbb{C}_S$, calculant à partir d'un arbre régulier, sa représentation en terme cyclique. Ensuite, nous prouvons la correction de cette fonction :

$$\forall (t : \textsc{Reg}_S).\ [\![\, \mathsf{sem\text{-}inv}\ t \,]\!] \sim t.$$

La combinaison de ces deux résultats (la correction et la complétude), nous permet de conclure que le type des termes cycliques est une syntaxe pour les arbres réguliers.

Enfin, nous considérons également le problème de donner une axiomatisation finie de l'équivalence entre termes cycliques. Cette relation d'équivalence, notée $\Gamma \vdash t \sim^{\mathsf{i}} s$, est définie inductivement comme suit :

$$\sim^{\mathsf{i}}_{\in}\ \frac{(t,s) \in \Gamma}{\Gamma \vdash t \sim^{\mathsf{i}} s}, \qquad \sim^{\mathsf{i}}_{\mathsf{var}}\ \frac{x = y}{\Gamma \vdash \mathsf{var}\,x \sim^{\mathsf{i}} \mathsf{var}\,y},$$

$$\sim^{\mathsf{i}}_{\mathsf{rec}}\ \frac{\forall k.\ (\mathsf{rec}\,o \lhd os, \mathsf{rec}\,o' \lhd os') :: \Gamma \vdash os(k)[* := \mathsf{rec}\,o \lhd os] \sim^{\mathsf{i}} os'(e_*(k))[* := \mathsf{rec}\,o' \lhd os']}{\Gamma \vdash \mathsf{rec}\,o \lhd os \sim^{\mathsf{i}} \mathsf{rec}\,o' \lhd os'}.$$

Nous prouvons que cette égalité syntaxique est équivalente à l'égalité sémantique $_ \sim^{\mathsf{s}} _$:

$$s \sim^{\mathsf{s}} t \overset{\text{def}}{\iff} [\![\, s \,]\!] \sim [\![\, t \,]\!].$$

Finalement, en combinant les résultats de ce chapitre, nous construisons les isomorphismes de setoids suivants :

$$(\mathbb{C}_S, \sim^{\mathsf{i}}) \cong (\mathbb{C}_S, \sim^{\mathsf{s}}) \cong (\textsc{Reg}_S, \sim).$$

CHAPITRE TROIS

Transducteurs d'arbres

Dans ce chapitre, nous étudions la problématique consistant à définir des morphismes d'arbres infinis préservant la régularité. Notre objectif est d'exhiber un critère syntaxique assurant, *par construction*, que les définitions co-récursives préservent cette propriété. Dans ce but, nous utilisons le formalisme des transducteurs d'arbres afin de représenter la syntaxe abstraite des fonctions (co-)récursives. Ensuite, chaque terme de cette syntaxe abstraite est réifié en un morphisme d'arbres réguliers.

III.1. Exemple d'illustration

Un morphisme d'arbres réguliers entre une signature d'entrée I et une signature de sortie O est un morphisme $\phi : \text{co}\mathbb{T}_I \to \text{co}\mathbb{T}_O$ tel que l'image de tout arbre régulier par ϕ est un arbre régulier. Considérons, par exemple, les deux fonctions mutuellement co-récursives suivantes :

$$\phi_1 : \textsc{Stream}\, A \to \textsc{Stream}\, A \qquad \phi_2 : \textsc{Stream}\, A \to \textsc{Stream}\, A$$
$$\phi_1 \ \ (x :: xs) \ \equiv \ f_1(x) :: \phi_2(xs) \qquad \phi_2 \ \ (x :: xs) \ \equiv \ f_2(x) :: \phi_1(xs)$$

où f_1 et f_2 sont deux fonctions arbitraires sur un type de base A. Afin de démontrer que les fonctions ϕ_1 et ϕ_2 préservent la propriété de régularité, il est nécessaire de prouver, étant donné un arbre régulier s en entrée, que le type des sous-arbres $\phi_i(s)\!\rightsquigarrow$ est fini. Néanmoins, produire une telle preuve peut se révéler être fastidieux. En effet, il convient de produire une fonction d'indexation vers un type fini de l'ensemble des sous-arbres et cela pour *tout* arbre régulier.

Cependant, si nous observons *syntaxiquement* les définitions des fonctions ϕ_i, il est aisé de remarquer qu'elles doivent nécessairement préserver la propriété de régularité. En effet, chaque fonction ϕ_i est définie de sorte que son action consiste à appliquer la fonction f_i sur le premier élément d'un flot. Par conséquent, si un flot σ en entrée est régulier (finalement périodique) alors son image par ϕ_i est nécessairement régulière.

À présent, considérons un exemple de fonction qui, cette fois, ne préserve pas la propriété de régularité. Pour tout $i : \mathbb{N}$, nous définissons une famille de fonctions co-récursives ψ_i comme suit :

$$\psi_i : \textsc{Stream}\ \mathbb{N} \to \textsc{Stream}\ \mathbb{N}$$
$$\psi_i \ \ (x :: xs) \ \equiv \ f_i(x) :: \psi_{i+1}(xs).$$

De plus, nous posons, pour toute valeur x, $f_i(x) := i$. Alors, nous pouvons conclure que pour tout flot en entré σ régulier ou non, le flot en sortie $\psi_0(\sigma)$ n'est pas régulier car ce dernier est équivalent au flot ω.

La différence entre les deux exemples précédents est que la famille de fonctions ψ_i est définie en utilisant une *infinité* d'équations co-récursives. Le fait que la famille ψ_i ne préserve pas la régularité n'est pas surprenant. En effet, la propriété de régularité peut être caractérisée par

l'existence de solutions pour des systèmes *finis* d'équations [Cou83, Blo83, AAV01]. Or, la famille ψ_i décrit en réalité en système d'équations infinis qui ne peut pas être réduit en un système fini d'équations équivalent.

III.2. Transducteurs d'arbres descendants

Dans cette section, nous présentons le formalisme des transducteurs descendants d'arbres. Par la suite, nous utiliserons ce formalisme comme outil pour définir des morphismes d'arbres préservant la propriété de régularité. Tout d'abord, nous rappelons les définitions et résultats des transducteurs d'arbres finis, en théorie des ensembles, tels qu'ils sont présentés dans la littérature. Ensuite, nous considérons le problème de définir, en théorie des types dépendants, des transducteurs qui opèrent cette fois sur des arbres infinis.

III.2.1. Définitions et sémantique

Les transducteurs d'arbres descendants [Rou68, Tha70, Rou70] sont un formalisme utilisé pour décrire des modèles sémantiques formels dirigés par la syntaxe. Ils permettent de spécifier des transformations d'arbres dans un formalisme qui ressemble à celui des grammaires attribuées avec attributs synthétisés.

Formellement, un transducteur (d'arbres finis) descendant est décrit par un quintuplet $T = \langle Q, \Sigma, \Delta, q_0, R\rangle$ où :

- Q est un *alphabet gradué unaire fini*, désignant les *états* du transducteur,
- Σ, Δ sont des alphabets gradués d'*entrée* et de *sortie* respectivement,
- $q_0 \in Q$ est un élément de Q, appelé *état initial*,
- R est un ensemble fini de *règles de réécriture* ayant la forme suivante :

$$q(\sigma(x_1, \ldots, x_k)) \to \xi$$

  où $q \in Q$, σ est un symbole Σ d'arité k, $x_1, \ldots, x_k \in X_k$ désignent des variables et ξ est un terme issu de l'algèbre de termes $\mathbb{T}_\Delta(QX_k)$.

L'ensemble R des règles de réécriture est dit *déterministe* (resp. complet) lorsque pour chaque partie gauche $q(\sigma(x_1, \ldots, x_k))$, il existe au plus (resp. au moins) une règle applicable. Quand cet ensemble est à la fois déterministe et complet, l'ensemble des parties droites des règles est déterminé de manière unique par sa partie gauche. Dans ce cas, la relation est fonctionnelle et nous notons la partie droite d'une règle $rhs(q, \sigma)$.

La sémantique des transducteurs d'arbres est généralement décrite par des systèmes de réécriture de termes. Ainsi la relation de dérivation induite par le transducteur T, notée $\Rightarrow_T \subseteq \mathbb{T}_{Q\cup\Sigma\cup\Delta} \times \mathbb{T}_{Q\cup\Sigma\cup\Delta}$, est définie par :

$$\frac{\beta \in \mathcal{C}_{Q\cup\Sigma\cup\Delta} \qquad q \in Q \qquad \sigma \in \Sigma^{(k)} \qquad s_1, \ldots, s_k \in \mathbb{T}_\Sigma \qquad \xi = rhs(q, \sigma)}{\beta[q(\sigma(s_1, \ldots, s_k))] \Rightarrow_T \beta[\xi[x_1 := s_1, \ldots, x_k := s_k]]}$$

où $\mathcal{C}$ désigne un contexte (terme avec une place) sur l'alphabet gradué $Q \cup \Sigma \cup \Delta$.

Le langage d'un transducteur d'arbres T, noté $\mathcal{L}(T)$, est défini par :

$$\mathcal{L}(T) = \{ (r, s) \in \mathbb{T}_\Sigma \times \mathbb{T}_\Delta \mid q_0(r) \Rightarrow_T^* s \}$$

où $\Rightarrow_T^*$ désigne la fermeture réflexive et transitive de $\Rightarrow_T$.

Lorsqu'un transducteur d'arbres T est déterministe et total, il est possible de montrer que la relation de dérivation induite par T est confluente et normalisante (voir [FV98]). De plus, chaque état du transducteur T induit un morphisme d'arbres, noté τ_T^q, admettant une caractérisation inductive :

$$\begin{aligned}(\tau_T^q)_{q\in Q} &: \mathbb{T}_\Sigma \to \mathbb{T}_\Delta \\ \sigma(s_1,\ldots,s_k) &\mapsto rhs(q,\sigma)\left[(q'_i,x_i) := \tau_T^{q'_i}(s_i)\,\middle|\, i\in\{1,\ldots,k\}\right].\end{aligned}$$

III.2.2. Transducteurs d'arbres en théorie des types dépendants

Dans cette section, nous définissons le *type* des transducteurs d'arbres *déterministes et totaux* opérant sur les arbres finis. Ces nouvelles définitions sont très proches de celles introduites (en théorie des ensembles) dans la section précédente.

Le type des transducteurs (d'arbres) descendants est défini au moyen d'un type enregistrement dépendant comme suit :

```
record TDTT (I O : Sig) : Type
  constructor ⟨_, _, _⟩
  [ state : Type
    init : state
    rhs : state → (i : I.Op) → O*(state × Fin(I.Ar i)).
```

La principale différence avec la définition des transducteurs de la section précédente concerne le type des états state. En effet, celui n'est pas nécessairement fini. De plus, les règles de réécriture sont représentées ici par une fonction (rhs) et non pas par une relation.

Finalement, nous pouvons définir les morphismes d'arbres induits par un transducteur d'arbres T au moyen d'une fonction récursive :

```
τ_T : T.state × 𝕋_I → 𝕋_O
τ_T  (q, o ◁ os)  ≡  (T.rhs q o)[(q', k) := τ_T(q', os(k))].
```

Ainsi, en partant de l'état initial du transducteur, nous obtenons un morphisme d'arbres, noté $(\!| T |\!)$, entre les algèbres de termes induites par les signatures I et O :

```
⦇T⦈ : 𝕋_I → 𝕋_O
⦇T⦈  t  ≡  τ_T(T.init, t).
```

III.2.3. Transducteurs descendants sur les arbres infinis

Dans cette section, nous présentons les problèmes rencontrés lorsque le formalisme des transducteurs descendants est étendu pour opérer sur les arbres infinis.

Tout d'abord, nous rappelons que pour les transducteurs sur les arbres finis, les parties droites des règles de réécriture sont des termes issus de l'algèbre de termes $\mathbb{T}_\Delta(Q\times X)$, où Δ désigne un alphabet gradué en sortie, Q un ensemble d'états et X un ensemble. En effet, les règles ayant la forme suivante :

$$q(\sigma(x_1,\ldots,x_k)) \to q(x_i)$$

sont syntaxiquement valides car $q(x_i) : \mathbb{T}_\Delta(Q\times X)$. Ainsi, il est tout à fait possible de définir des règles qui consomme un opérateur en entrée mais qui ne produisent pas de symbole en sortie. Ce type de règles n'est pas problématique pour les arbres finis car ce sont des structures inductives. En effet, les feuilles de l'arbre sont nécessairement d'arité zéro. Dans ce cas, les parties droites des

règles de réécriture ne peuvent pas contenir de symbole représentant un état. Ainsi, ces règles sont finalement productives.

En revanche, si nous considérons la possibilité que les arbres soient infinis, l'argument qui justifie la validité des règles de réécriture non productives n'est pas plus valable. Par exemple, considérons la règle de réécriture suivante :

$$q(\sigma(x)) \to q(x). \tag{$*$}$$

Si nous appliquons la règle $(*)$ successivement sur le flot σ^ω :

$$q(\sigma\sigma\sigma\cdots) \Rightarrow q(\sigma\sigma\cdots) \Rightarrow q(\sigma\cdots) \Rightarrow q(\cdots) \Rightarrow \cdots \Rightarrow q(\cdots) \Rightarrow \cdots,$$

nous obtenons une dérivation infinie non productive. C'est précisément pour ces raisons que de telles règles non productives sont interdites dans les définitions de fonctions co-récursives en COQ ou en AGDA.

Afin d'interdire ces règles de réécriture problématiques, nous proposons deux solutions. La première solution consiste à caractériser le sous-ensemble des règles productives. Dans ce cas, chaque règle de réécriture doit être accompagnée d'une preuve justifiant qu'elle sera *finalement* productive. Cette solution a l'avantage d'offrir une très grande liberté dans l'écriture de la règle. Néanmoins, produire le terme de preuve peut être relativement fastidieux. En effet, la productivité doit être garantie pour tout arbre en entrée. La seconde solution s'appuie sur un critère syntaxique, semblable à la condition de garde, afin de garantir la productivité des règles : tout état doit être gardé par un opérateur de l'alphabet gradué de sortie. L'avantage de cette solution est qu'il n'est plus nécessaire de fournir de preuve de productivité, les règles sont productives *par construction*. En revanche, des contraintes syntaxiques plus fortes s'appliquent pour l'écriture de ces règles.

III.3. Transducteurs descendants gardés

Dans cette section, nous modifions la définition du type TDTT des transducteurs descendants sur les arbres finis. Cette modification s'appuie sur la seconde solution présentée dans la section précédente consistant à caractériser les parties droites des règles de réécriture sous forme gardée :

```
record ᴳTDTT (I O : Sig) : Type
  constructor ⟨_, _, _,⟩
    [ state : Type
    [  init : state
    [   rhs : ∀(q : state). ∀(i : I.Op). O⁺(state × Fin(I. Ar i)).
```

La principale différence avec le type TDTT est que le type de retour de rhs est maintenant $O^+(\mathsf{state} \times \mathrm{FIN}(I.\,\mathsf{Ar}\,i))$ plutôt que $O^*(\mathsf{state} \times \mathrm{FIN}(I.\,\mathsf{Ar}\,i))$. Par conséquent, chaque règle de réécriture doit nécessairement produire un symbole en sortie : *la productivité est garantie par construction*. De la même façon que pour le type TDTT, nous définissons, cette fois, par co-itération, le morphisme sur les arbres infinis induit par un transducteur descendant gardé T :

$$(\!| T |\!) : \mathrm{co}\mathbb{T}_I \to \mathrm{co}\mathbb{T}_O.$$

Enfin, nous prouvons que ce morphisme préserve la propriété de régularité sur les arbres infinis. Ainsi, le type du morphisme induit peut être changé en :

$$(\!| T |\!) : \mathrm{REG}_I \to \mathrm{REG}_O.$$

III.4. Transducteurs descendants avec règles $\mathcal{E}$

Le formalisme des transducteurs descendants gardés, tel que définit précédemment, impose que chaque dérivation induite par toute règle de réécriture consomme la racine du sous-arbre en entrée. Lorsque nous avons présenté, au début de ce chapitre, les transducteurs descendants opérant sur les arbres finis, cette restriction était nécessaire pour garantir la terminaison de l'application successive des règles de réécriture. Cependant, une telle contrainte n'est plus indispensable dès lors que l'on considère les arbres infinis. En effet, dans ce contexte, seule la productivité compte. Ainsi, il est possible de lever totalement cette restriction et d'autoriser l'écriture de règles purement productives. Cela conduit à l'extension du formalisme des transducteurs gardés suivante :

```
record εTDTT (I O : Sig) : Type
  constructor ⟨_,_,_,⟩
    state : Type
    init : state
    rhs : state → (i : I.Op) → O⁺(state × Maybe(Fin(I. Ar i))).
```

Dans le cas des transducteurs gardés, le type des parties droites des règles étant donné par $O^{+}(\mathsf{state} \times \textsc{Fin}(I.\,\mathsf{Ar}\,i))$. Ici, le type $\textsc{Fin}(I.\,\mathsf{Ar}\,i)$ représente l'ensemble des variables. En particulier, il est utilisé afin d'indexer de manière canonique les sous-arbres de l'arbre en entrée. Maintenant, nous utilisons le foncteur MAYBE afin d'exprimer une modalité sur le choix de l'arbre sur lequel la dérivation doit se poursuivre. Ainsi, il est possible, de choisir une variable indexant un sous-arbre de l'arbre en entrée sur lequel la dérivation se poursuivra, ou alors de n'en choisir aucune. Dans ce cas, cela signifie que la dérivation se poursuit implicitement sur la totalité de l'arbre en entrée. Nous définissons par co-itération, le morphisme sur les arbres infinis induit par un transducteur gardé avec règles ε :

$$(\!|\, T \,|\!) : \mathrm{co}\mathbb{T}_I \to \mathrm{co}\mathbb{T}_O.$$

De plus, nous prouvons que ce morphisme préserve la propriété de régularité sur les arbres infinis. Ainsi, le type du morphisme induit peut être changé en :

$$(\!|\, T \,|\!) : \textsc{Reg}_I \to \textsc{Reg}_O.$$

Finalement, nous montrons que le formalisme des transducteurs gardés est inclus dans le formalisme des transducteurs gardés avec règles ε. Ainsi, il existe une fonction de traduction entre les deux, donnée par le profil suivant :

$$^{\mathrm{G}\to\varepsilon}\mathsf{TDTT} : \forall\{I, O : \textsc{Sig}\}.\ {}^{\mathrm{G}}\mathrm{TDTT}\ I\ O \to {}^{\varepsilon}\mathrm{TDTT}\ I\ O$$

et vérifiant pour tout transducteur $T : {}^{\mathrm{G}}\mathrm{TDTT}\ I\ O$:

$$\forall (t : \mathrm{co}\mathbb{T}_I).\ (\!|\, T \,|\!)(t) \sim (\!|\, {}^{\mathrm{G}\to\varepsilon}\mathsf{TDTT}\ T \,|\!)(t).$$

III.5. Transducteurs descendants à observation finie

Précédemment, nous avons introduit les transducteurs gardés sur les arbres infinis afin de garantir la productivité des règles par construction. Ensuite, ce formaliste a été étendu afin de permettre l'utilisation de règles de réécriture purement productives. Maintenant, nous définissons une autre extension du formalisme des transducteurs en ajoutant un contexte d'observation fini. Dans le cadre des arbres finis, ce contexte est généralement calculé au moyen d'automates d'arbres ascendants [CDG+07]. Cependant, nous travaillons ici sur des arbres infinis, par conséquent, nous devons imposer certaines restrictions. Par exemple, la profondeur du contexte d'observation doit être finie car celle-ci peut être arbitrairement large.

Le type des transducteurs d'arbres à observation finie est défini comme suit :

```
record ᴸTDTT (I O : Sig) : Type
  constructor ⟨_, _, _⟩
    ⎡ state : Type
    ⎢  init : state
    ⎣  rhs : ∀(q : state). ∑_{d:ℕ}(c : I^d(Unit)) → O⁺(state × {p | p ∈ 𝒫_d(c)}).
```

Ce nouveau formalisme est introduit selon deux axes. D'une part, chaque état q est annoté par un nombre d exprimant la profondeur du contexte d'observation nécessaire afin de produire une valeur. Les contextes sont représentés en itérant d fois le foncteur induit par l'alphabet d'entrée I. D'autre part, le type des variables indexant les sous-arbres est généralisé. Ici, ce type est donné par les chemins dans le contexte d'observation. En particulier, ces chemins servent à indiquer la partie du contexte d'observation qui est consommée par la règle de réécriture.

Nous définissons par co-itération, le morphisme sur les arbres infinis induit par un transducteur à observation finie :

$$(\!|\, T \,|\!) : \mathrm{co}\mathbb{T}_I \to \mathrm{co}\mathbb{T}_O.$$

Cette fonction s'obtient en 3 étapes (voir Figure 3.2) :

① le contexte d'observation de profondeur $\mathsf{depth}\, q$ est généré à partir de l'arbre en entrée,
② la règle de réécriture applicable est utilisée,
③ le contexte d'observation qui n'est pas consommé est remis dans l'arbre en entrée.

De plus, nous prouvons que ce morphisme préserve la propriété de régularité sur les arbres infinis. Ainsi, le type du morphisme induit peut être changé en :

$$(\!|\, T \,|\!) : \mathrm{Reg}_I \to \mathrm{Reg}_O.$$

Finalement, nous montrons que le formalisme des transducteurs avec règles ε est inclu dans le formalisme des transducteurs avec contexte d'observation fini. Ainsi, il existe une fonction de traduction entre les deux, donné par le profil suivant :

$$^{\varepsilon\to\mathrm{L}}\mathsf{TDTT} : \forall\{I, O : \mathrm{Sig}\}.\ {}^{\varepsilon}\mathrm{TDTT}\ I\ O \to {}^{\mathrm{L}}\mathrm{TDTT}\ I\ O$$

et vérifiant pour tout transducteur $T : {}^{\varepsilon}\mathrm{TDTT}\ I\ O$:

$$\forall(t : \mathrm{co}\mathbb{T}_I).\ (\!|\, T \,|\!)(t) \sim (\!|\, {}^{\varepsilon\to\mathrm{L}}\mathsf{TDTT}\ T \,|\!)(t).$$

III.6. Transducteurs descendants binaires

Dans cette section, nous expliquons comment les transducteurs descendants binaires sont obtenus à partir de leur variante unaire. Les transducteurs binaires opèrent sur deux arbres simultanément avec des signatures en entrée et en sortie potentiellement différentes. L'idée principale sur laquelle repose cette construction consiste à définir une opération de produit interne, notée $_\otimes_$, sur les signatures. Concrètement, cette opération est définie comme suit :

```
_⊗_ : Sig → Sig → Sig
S₁ ⊗ S₂  ≡  S₁.Op × S₂.Op ◁ uncurry(λo₁. λo₂. S₁.Ar o₁ + S₁.Ar o₁ × S₂.Ar o₂ + S₂.Ar o₂).
```

Ensuite, nous étendons ce produit de signatures aux arbres infinis. Ainsi, nous définissons une opération consistant à transformer un couple d'arbres infinis en un arbre infini sur un produit de signatures. Cette opération est donnée par la fonction co-récursive suivante :

$$
\begin{array}{l}
\otimes : \mathrm{CO}\mathbb{T}_{I_1} \to \mathrm{CO}\mathbb{T}_{I_2} \to \mathrm{CO}\mathbb{T}_{I_1 \otimes I_2} \\
t_1 \otimes t_2 \equiv \mathsf{root}\, t_1, \mathsf{root}\, t_2 \blacktriangleleft \lambda k. \quad \textbf{match } k \textbf{ with} \\
\qquad \mid \mathsf{inl}(\mathsf{inl}(i,j)) \Rightarrow t_1(i) \otimes t_2(j) \\
\qquad \mid \mathsf{inr}(\mathsf{inl}(i)) \Rightarrow t_1(i) \otimes t_2 \\
\qquad \mid \mathsf{inr}(\mathsf{inr}(j)) \Rightarrow t_1 \otimes t_2(j) \\
\textbf{end}
\end{array}
$$

En particulier, nous prouvons que ce produit d'arbres ($_\otimes_$) préserve la propriété de régularité. Par conséquent, nous pouvons utiliser cette opération pour dériver les transducteurs binaires à partir des transducteurs unaires. Cette construction se réalise en deux étapes.

Dans un premier temps, le transducteur binaire est décrit par sa variante unaire choisie parmi celles présentées précédemment. La signature en entrée de ce transducteur unaire est donnée par le produit des signatures en entrée du transducteur binaire.

Dans un second temps, le morphisme induit par le transducteur unaire est pré-composé avec l'opération de produit d'arbres. Ainsi, il est aisé de démontrer que le morphisme induit par le transducteur binaire préserve la propriété de régularité. En effet, les deux fonctions qui le définisse, à savoir le morphisme induit par le transducteur unaire et l'opération de produit, préservent toutes deux la régularité des arbres.

CHAPITRE QUATRE

Applications

Dans ce chapitre, nous étudions deux applications de la théorie des arbres réguliers développée jusqu'à présent. Dans un premier temps, nous considérons le problème consistant à définir l'opération de composition parallèle d'une algèbre de processus. En particulier, cette algèbre de processus contient un constructeur permettant de définir des processus récursifs. Sémantiquement, les processus récursifs sont identifiés à leur dépliage. Dans ce contexte, nous montrons comment un problème de terminaison peut être reformulé en problème de productivité. Dans un second temps, nous donnons une interprétation du μ-calcul coalgébrique [CKP11] sur les arbres réguliers. En particulier, nous prouvons la décidabilité de la satisfiabilité. Enfin, nous donnons une preuve de la décidabilité de la relation de bissimilarité sur les arbres réguliers en utilisant une réduction vers un problème de vérification de modèles.

IV.1. Séquentialisation de processus

Nous considérons ici le problème consistant à définir l'opération de composition parallèle synchrone pour un fragment d'une algèbre de processus telle que BPA [BK88] ou CSP [Hoa78]. La grammaire, notée Proc, des processus est donnée par :

$$P, Q ::= \mathsf{STOP} \mid \mathsf{SKIP} \mid a \to P \mid P \,\square\, Q \mid \mu X.\, P \mid X$$

où SKIP (resp. STOP) désigne un processus qui termine avec succès (resp. échec). Le processus $a \to P$ accepte l'action a puis se comporte comme P. Le choix non-déterministe entre deux processus P et Q s'écrit $P \,\square\, Q$. Enfin, la construction syntaxique $\mu X.P$ permet de représenter des processus récursifs où X désigne une variable de processus.

Généralement, les sémantiques associées aux algèbres de processus peuvent s'exprimer au moyen d'une fonction $[\![-]\!] : \textsc{Proc} \to \mathcal{D}$ où $\mathcal{D}$ désigne un domaine sémantique. Néanmoins, indépendamment du choix du domaine $\mathcal{D}$, l'opérateur d'itération μ doit être assimilé à de son dépliage. Par conséquent, l'équation suivante doit être valable sur le domaine sémantique :

$$[\![\mu X.\, P]\!] =_{\mathcal{D}} [\![P[X := \mu X.\, P]]\!] \qquad (\mu\text{-DÉPLIAGE})$$

L'opération de composition parallèle synchrone est axiomatisée comme suit :

$$\begin{array}{lllll} \mathsf{STOP} \parallel P = \mathsf{STOP} & & (a \to P) \parallel (a \to Q) = a \to (P \| Q) & \\ P \parallel \mathsf{STOP} = \mathsf{STOP} & & (a \to P) \parallel (b \to Q) = \mathsf{STOP} & a \neq b \\ P \parallel \mathsf{SKIP} = P & & (P \,\square\, Q) \parallel R = (P \| R) \,\square\, (Q \| R) & \\ \mathsf{SKIP} \parallel P = P & & R \parallel (P \,\square\, Q) = (P \| R) \,\square\, (Q \| R). & \end{array}$$

Ici, il n'y a pas d'axiome pour l'opérateur d'itération μ car les équations peuvent être dérivées de celle donnée par (μ-DÉPLIAGE). En effet, la composition parallèle doit être une congruence et ainsi doit satisfaire les équations suivantes :

$$(\mu X.\, P)\|Q = (P[X := \mu X.\, P])\|Q, \qquad P\|(\mu X.\, Q) = P\|(Q[X := \mu X.\, Q]).$$

L'ensemble des axiomes donné précédemment peut être vu comme une définition de fonction récursive. Néanmoins, si une telle définition était utilisée dans un assistant de preuve tel que COQ, celle-ci serait rejetée par le vérificateur de terminaison. Le problème vient de l'opérateur d'itération μ et de la règle sémantique qui lui est associée, à savoir le dépliage. En effet, le terme $P[X := \mu X.P]$ peut être syntaxiquement plus grand que $\mu X.P$.

Cependant, avec une syntaxe co-inductive, le problème de terminaison évoqué précédemment disparaît. En effet, dans un contexte co-inductif, l'opérateur d'itération μ n'est plus nécessaire car il est possible d'exprimer des termes infinis directement. Dès lors, l'axiomatisation précédente peut maintenant être vue comme une définition d'une fonction co-récursive :

$$\begin{array}{llll}
\multicolumn{4}{l}{_\|^\infty_ : \infty\textsc{Proc} \to \infty\textsc{Proc} \to \infty\textsc{Proc}} \\
\mathsf{STOP} & \|^\infty \ _ & \equiv & \mathsf{STOP} \\
_ & \|^\infty \ \mathsf{STOP} & \equiv & \mathsf{STOP} \\
\mathsf{SKIP} & \|^\infty \ P & \equiv & P \\
P & \|^\infty \ \mathsf{SKIP} & \equiv & P \\
(a \to P) & \|^\infty \ (b \to Q) & \equiv & \textbf{if } a \stackrel{?}{=} b \textbf{ then } a \to P \|^\infty Q \textbf{ else } \mathsf{STOP} \textbf{ end} \\
(P \,\square\, Q) & \|^\infty \ R & \equiv & (P \|^\infty R) \,\square\, (Q \|^\infty R) \\
R & \|^\infty \ (P \,\square\, Q) & \equiv & (R \|^\infty P) \,\square\, (R \|^\infty Q)
\end{array}$$

où ∞PROC désigne la syntaxe co-inductive. Cette définition est acceptée par COQ car elle est productive. En effet, chaque appel co-récursif est gardé par un constructeur. Ainsi, il est beaucoup plus aisé de définir l'opération de composition parallèle sur la syntaxe infinie.

Maintenant, nous allons utiliser la théorie sur les transducteurs d'arbres développée dans le chapitre précédent afin de dériver la définition de l'opération de composition parallèle sur la syntaxe finie à partir de celle sur la syntaxe infinie. L'idée est de remarquer que les processus peuvent être exprimés au moyen de termes cycliques sur la signature suivante :

$$\textsc{Proc} := \left\{ \mathsf{STOP}^{(0)} \ ; \mathsf{SKIP}^{(0)} \ ; \ \to_a^{(1)} \ ; \square^{(2)} \right\}$$

Ainsi, nous pouvons exploiter la fonction sémantique, $[\![-]\!] : \mathbb{C}_{\textsc{Proc}} \to \textsc{Reg}_{\textsc{Proc}}$, qui interprète les termes cycliques comme des arbres infinis réguliers. De plus, il est clair que la fonction $_\|^\infty_$ peut s'exprimer au moyen d'un transducteur binaire d'arbres. Sachant que les transducteurs préservent la propriété de régularité, l'arbre infini résultant de la composition parallèle est régulier. Enfin, en exploitant le morphisme inverse, $[\![-]\!]^{-1} : \textsc{Reg}_{\textsc{Proc}} \to \mathbb{C}_{\textsc{Proc}}$, on obtient une représentation de l'arbre régulier comme un terme cyclique. Pour résumer, on dérive la définition de la fonction $_\|_$ à partir de la définition de la fonction $_\|^\infty_$ comme suit :

$$P\|Q := [\![\, (\!| T |\!) \, [\![P]\!] \, [\![Q]\!] \,]\!]^{-1}$$

où T désigne le transducteur binaire représentant $_\|^\infty_$ et $(\!| T |\!)$ son morphisme induit. En particulier, cette définition respecte l'égalité sémantique suivante :

$$(\mu X.\, P)\|Q \approx (P[X := \mu X.\, P])\|Q.$$

IV.2. Vérification de modèles sur les arbres réguliers

Dans cette section, nous introduisons la syntaxe d'un μ-calcul coalgébrique [CKP11] et nous définissons sa sémantique interprétée sur une T-coalgèbre arbitraire. Le traitement de la logique modale sous-jacente est paramétrique, dans le sens où les opérateurs modaux sont donnés par une signature arbitraire. Ensuite, nous montrons que la satisfiabilité est décidable lorsque les formules du μ-calcul sont interprétées sur des T-coalgèbres finis et en particulier sur les arbres réguliers. Enfin en utilisant ce résultat, nous montrons que la relation de bissimilarité est décidable pour les arbres réguliers.

IV.2.1. Syntaxe

Étant donnée une signature Λ : Sig, la syntaxe du μ-calcul coalgébrique [CKP11] est définie inductivement comme suit :

$$
\begin{array}{l}
\textbf{inductive}\ \textsc{Term}\ (n:\mathbb{N}):\textsc{Type}\\
\quad \begin{array}{rl}
\mathsf{var} & : \textsc{Fin}\,n \to \textsc{Term}\,n\\
\mu & : \textsc{Term}(\mathsf{suc}\,n) \to \textsc{Term}\,n\\
\nu & : \textsc{Term}(\mathsf{suc}\,n) \to \textsc{Term}\,n\\
\wedge & : \textsc{Term}\,n \to \textsc{Term}\,n \to \textsc{Term}\,n\\
\vee & : \textsc{Term}\,n \to \textsc{Term}\,n \to \textsc{Term}\,n\\
\circ & : \Lambda(\textsc{Term}\,n) \to \textsc{Term}\,n\\
\bullet & : \Lambda(\textsc{Term}\,n) \to \textsc{Term}\,n.
\end{array}
\end{array}
$$

Les formules $\top$ and $\bot$ ainsi que l'opérateur de négation $\neg_$ ne sont pas dans la syntaxe car ils peuvent être dérivés, c'est-à-dire, définis comme fonctions sur la syntaxe. Les lieurs μ et ν ainsi que les variables utilisent une représentation anonyme canonique (indices de de Bruijn). Les constructeurs $\circ$ et $\bullet$ représentent les opérateurs modaux tels que données par la signature Λ.

IV.2.2. Sémantique dénotationnelle

Les formules du μ-calcul sont interprétés sur un T-coalgèbre arbitraire $(X, \gamma : X \longrightarrow T(X))$ où T est un endofoncteur. Le domaine sémantique est donné par les fonctions associant des environnements à des sous-ensembles d'états. Nous notons, $\wp$: Setoid $\to$ Setoid, le foncteur contravariant « ensemble des parties » :

$$
\begin{array}{ll}
\wp : \textsc{Setoid} \to \textsc{Setoid} & \qquad _^{-1} : (X \longrightarrow Y) \to \wp(Y) \longrightarrow \wp(X)\\
\wp\ \ X\ \equiv\ X \longrightarrow \textsc{Prop}, & \qquad f^{-1}\ \ A\ \equiv\ \{\, x : f(x) \in A \,\}.
\end{array}
$$

Nous rappelons que le setoid $\wp(X)$ est un treillis complet. De plus, si nous admettons la loi du tiers-exclus, c'est aussi une algèbre de boole complète.

Nous supposons l'existence d'une fonction $[\![\circ]\!] : \forall X.\ \Lambda(\wp(X)) \longrightarrow \wp(T(X))$, qui donne une sémantique à l'ensemble des opérateurs modaux de la signature Λ, telle que pour tout morphisme $f : Y \longrightarrow X$, le diagramme suivant commute :

$$
\begin{array}{ccc}
\Lambda(\wp(Y)) & \xrightarrow{[\![\circ]\!]_Y} & \wp(T(Y))\\
\uparrow{\scriptstyle \Lambda(f^{-1})} & & \uparrow{\scriptstyle T(f)^{-1}}\\
\Lambda(\wp(X)) & \xrightarrow[{[\![\circ]\!]_X}]{} & \wp(T(X))
\end{array}
$$

Étant donné un symbole de fonction $\lambda^{(k)} \in \Lambda$ d'arité k, la condition de naturalité exprimée précédemment correspond à l'équation suivante :

$$\forall (A_1, \ldots, A_k : \wp(X)).\ T(f)^{-1}(\llbracket \circ \rrbracket^{\lambda}_{X}(A_1, \ldots, A_k)) \approx \llbracket \circ \rrbracket^{\lambda}_{Y}(f^{-1}(A_1), \ldots, f^{-1}(A_k)).$$

La fonction qui donne une sémantique aux opérateurs modaux duaux est obtenue comme suit :

$$\begin{aligned} &\llbracket \bullet \rrbracket : \forall X.\ \Lambda(\wp(X)) \longrightarrow \wp(T(X)) \\ &\llbracket \bullet \rrbracket_X \ \equiv\ \complement \circ \llbracket \circ \rrbracket \circ \Lambda(\complement) \end{aligned}$$

où $\complement$ désigne l'opération de complémentation d'un ensemble.

Enfin, la sémantique dénotationnelle du μ-calcul coalgébrique est calculée de la façon suivante :

$$\begin{aligned} &\llbracket _ \rrbracket_{_} : \forall \{n : \mathbb{N}\}.\ \textsc{Term}\, n \rightarrow \textsc{Env}_X(n) \rightarrow \wp(X) \\ &\llbracket\ \mathsf{var}\, x\ \rrbracket_\rho \ \equiv\ \rho(x) \\ &\llbracket\ \mu\Phi\ \rrbracket_\rho \ \equiv\ \mathsf{lfp}(\lambda(A : \wp(X)).\ \llbracket \Phi \rrbracket_{(A :: \rho)}) \\ &\llbracket\ \nu\Phi\ \rrbracket_\rho \ \equiv\ \mathsf{gfp}(\lambda(A : \wp(X)).\ \llbracket \Phi \rrbracket_{(A :: \rho)}) \\ &\llbracket\ \Phi_1 \wedge \Phi_2\ \rrbracket_\rho \ \equiv\ \llbracket \Phi_1 \rrbracket_\rho \sqcap \llbracket \Phi_2 \rrbracket_\rho \\ &\llbracket\ \Phi_1 \vee \Phi_2\ \rrbracket_\rho \ \equiv\ \llbracket \Phi_1 \rrbracket_\rho \sqcup \llbracket \Phi_2 \rrbracket_\rho \\ &\llbracket\ \circ\, \Phi\ \rrbracket_\rho \ \equiv\ \gamma^{-1}(\llbracket \circ \rrbracket(\Lambda(\llbracket _ \rrbracket_\rho)\ \Phi)) \\ &\llbracket\ \bullet\, \Phi\ \rrbracket_\rho \ \equiv\ \gamma^{-1}(\llbracket \bullet \rrbracket(\Lambda(\llbracket _ \rrbracket_\rho)\ \Phi)) \end{aligned}$$

où le type des environnements est défini par $\textsc{Env}_X(n) := \textsc{Vec}\ (\wp(X))\ n$.

En supposant que la fonction $\llbracket \circ \rrbracket$ est monotone, on montre par induction sur la syntaxe, que la fonction sémantique $\llbracket - \rrbracket$ est monotone. Ce résultat justifie l'existence des points fixes. De plus, sur les T-coalgèbres finies, ces points fixes sont calculables. Dès lors, on montre que l'on peut, sur cette restriction (T-coalgèbre finie), décider de la satisfiabilité d'une formule du μ-calcul.

IV.2.3. Décidabilité de la relation de bissimilarité

Dans cette section, nous donnons une preuve de la décidabilité de la relation de bissimilarité sur les arbres réguliers. Cette preuve s'appuie sur tous les résultats précédents.

Soit S : $\textsc{Sig}$ une signature telle que l'égalité sur les opérateurs $S.\mathsf{Op}$ est décidable. Nous spécialisons le μ-calcul coalgébrique pour la signature suivante :

$$\Lambda := \left\{ \left(\mathsf{pred}^{(0)}\, p \right)_{(p : S_\perp.\mathsf{Op} \rightarrow \mathbb{B})}\ ;\ \Box^{(1)} \right\}$$

où $S_\perp$ désigne la signature obtenue à partir de S auquel a été adjoint un symbole d'arité zéro. Le constructeur $\mathsf{pred}\, p$ permet d'exprimer des prédicats sur les états alors que $\Box$ est l'opérateur de modalité « pour tout successeur. » La sémantique de chacun de ces opérateurs est donnée par :

$$\begin{aligned} &\llbracket \mathsf{pred}\, p \rrbracket : \wp(S_\perp(X)) && \llbracket \Box \rrbracket : \wp(X) \rightarrow \wp(S_\perp(X)) \\ &\llbracket \mathsf{pred}\, p \rrbracket \ \equiv\ \{\, s : p(\mathsf{proj}_1(s)) = \mathsf{true} \,\}, && \llbracket \Box \rrbracket\ R \ \equiv\ \{\, s : S^{\forall}_{\perp}\ (\lambda s'.\ s' \in R)\ s \,\}. \end{aligned}$$

Ici, le μ-calcul coalgébrique que l'on considère est très proche du μ-calcul propositionnel [Koz83].

Nous rappelons que, étant donné un arbre régulier χ : $\textsc{Reg}_S$ sur une signature arbitraire S, le type de ses successeurs $\chi{\leadsto}$, possède une structure de S-coalgèbre. Ainsi, nous pouvons interpréter les formules du μ-calcul sur cette coalgèbre. De plus, par définition de la propriété de régularité, le support de cette coalgèbre est fini. Cela implique, d'après le résultat de la section précédente, que nous pouvons décider de la satisfiabilité des formules. Par conséquent, pour prouver la décidabilité de la relation de bissimilarité, il suffit de montrer que cette relation peut être représentée, de manière équivalente, par une formule du μ-calcul.

Maintenant, nous détaillons la construction permettant de caractériser la relation de bissimilarité comme une formule du μ-calcul. Tout d'abord, nous précisons comment la signature $S_\perp$ est obtenue à partir de S :

$$
\begin{array}{l}
S_\perp : \textsc{Sig} \\
S_\perp \;\equiv\; \textsc{Maybe}\, S.\mathsf{Op} \lhd ar \\
\quad \textbf{where}\;\; ar : \textsc{Maybe}(S.\mathsf{Op}) \to \mathbb{N} \\
\qquad\qquad ar\;\; \mathsf{nothing} \;\equiv\; \mathsf{zero} \\
\qquad\qquad ar\;\; (\mathsf{just}\, o) \;\equiv\; S.\,\mathsf{Ar}\, o.
\end{array}
$$

L'idée générale consiste à définir un produit synchrone d'arbres. Intuitivement, le calcul de ce produit donne l'espace d'états sur lequel la formule du μ-calcul sera interprétée alors que la formule elle-même vérifiera si les arbres sont effectivement synchronisables. Le produit synchrone d'arbres est défini co-récursivement comme suit :

$$
\begin{array}{l}
\otimes : \mathrm{co}\mathbb{T}_S \to \mathrm{co}\mathbb{T}_S \to \mathrm{co}\mathbb{T}_{S_\perp} \\
t \otimes s \;\equiv\; \textbf{match}\; \mathsf{root}\, t \stackrel{?}{=} \mathsf{root}\, s \;\textbf{with} \\
\qquad\qquad \mid\; \mathsf{yes}\, e \Rightarrow \mathsf{just}(\mathsf{root}\, t) \blacktriangleleft \lambda k.\, t(k) \otimes s(e_*(k)) \\
\qquad\qquad \mid\; \mathsf{no}\, _ \Rightarrow \mathsf{nothing} \lhd \bot\text{-}\mathsf{elim} \\
\qquad\qquad \textbf{end}
\end{array}
$$

Intuitivement, étant donnés deux arbres, nous comparons leurs racines. Si elles sont égales, alors elles sont « synchronisées » et le calcul se poursuit co-récursivement sur les sous-arbres ; si elles diffèrent, le calcul du produit s'achève. Il est clair que la fonction $_\otimes_$ peut être exprimée par un transducteur binaire. Par conséquent, si les deux arbres en entrée sont réguliers alors leur produit synchrone le sera également. La formule du μ-calcul exprimant la propriété de synchronisation entre deux arbres est définie par :

$$\nu X.\, \textsc{ok} \wedge \Box X. \tag{$\otimes$-sync}$$

où $\textsc{ok}$ est une formule sur les états spécifiant si la synchronisation entre les racines a eu lieu. Elle est définie de la façon suivante :

$$
\begin{array}{l}
\textsc{ok} : \textsc{Term}\, \mathsf{zero} \\
\textsc{ok} \;\equiv\; \mathsf{pred}\, \textsc{sync?} \\
\quad \textbf{where}\;\; \textsc{sync?} : \textsc{Maybe}(S.\mathsf{Op}) \to \mathbb{B} \\
\qquad\qquad \textsc{sync?}\;\; \mathsf{nothing} \;\equiv\; \mathsf{false} \\
\qquad\qquad \textsc{sync?}\;\; (\mathsf{just}\, _) \;\equiv\; \mathsf{true}.
\end{array}
$$

Enfin, nous montrons que la relation de bissimilarité entre les deux états est caractérisée de manière équivalente par la formule de μ-calcul. Concrètement, nous prouvons l'énoncé suivant :

$$t \sim s \leftrightarrow (t \otimes s, \varepsilon) \vDash \nu X.\, \textsc{ok} \wedge \Box X.$$

où $(t \otimes s, \varepsilon) \vDash \nu X.\, \textsc{ok} \wedge \Box X$ est une notation pour $(t \otimes s, \varepsilon) \in [\![\, \nu X.\, \textsc{ok} \wedge \Box X \,]\!]_{[\,]}$.

Pour conclure, la relation de bissimilarité sur les arbres est décidable car celle-ci est équivalente à une formule du μ-calcul interprétée sur le produit synchrone d'arbres qui possède la structure d'une coalgèbre finie.

Conclusion

Dans cette thèse, nous avons présenté une mécanisation constructive d'une théorie des arbres réguliers en théorie des types dépendants. Les arbres réguliers sont des arbres infinis caractérisés par une propriété de finitude spécifiant que le sous-ensemble des sous-arbres distincts est fini. Dans ce travail, nous avons traité deux problématiques majeures. La première concerne la caractérisation des arbres réguliers comme un type. Ainsi, nous avons abordé ce sujet en considérant un point de vue sémantique visant à définir les arbres réguliers au moyen de types co-inductifs. Puis, nous avons proposé une syntaxe des arbres réguliers et prouvé la correction de la représentation syntaxique par rapport à la caractérisation sémantique. La seconde problématique consistait à introduire une notion de morphisme d'arbres réguliers. Dans ce travail, nous avons cherché des structures de données suffisamment abstraites avec des hypothèses aussi faibles que possible pour le cadre constructif considéré. Nous avons travaillé dans une théorie des types dans laquelle le principe du tiers-exclu (LEM) n'est pas admis, tout comme le principe d'indépendance des preuves (proof-irrelevance), le principe de l'extensionalité des fonctions ou encore le principe d'unicité des preuves d'égalité (UIP). Les paragraphes qui suivent résument les principales contributions.

La première contribution de cette thèse est une étude de la caractérisation des types finis en théorie des types. Plusieurs notions de types finis existent dans un cadre constructif. Cette problématique a été élargie en considérant des setoids finis, c'est-à-dire, des types munis d'une relation d'équivalence, dans le but de permettre le raisonnement sur les types co-inductifs modulo relation de bissimilarité. Nous avons comparé plusieurs notions de types finis et les avons classées en trois catégories: *setoid énumérable*, *setoid faiblement finiment indexable*, et *setoid sans suite injective* (streamless). Les setoids faiblement finiment indexables sont particulièrement intéressants puisqu'ils permettent de considérer des setoids finis dont la relation d'équivalence n'est pas nécessairement décidable.

La deuxième contribution de cette thèse consiste en une caractérisation *sémantique et syntaxique du type des arbres réguliers* sur une signature arbitraire S. Dans un premier temps, nous avons spécifié le type des arbres réguliers comme les arbres infinis dont l'ensemble de sous-arbres distincts (à bissimilarité près) est faiblement finiment indexable. En particulier, nous n'avons pas supposé que l'égalité sur les symboles de fonctions donnés par la signature est décidable. De plus, nous avons prouvé que le type des arbres réguliers possède une structure de S-coalgèbre. Puis, nous avons donné une caractérisation syntaxique du type des arbres réguliers au moyen de termes cycliques où les cycles ont été représentés au moyen de lieurs. Nous avons validé que les termes cycliques ont une interprétation sémantique en terme d'arbres réguliers en dépliant les cycles indéfiniment. En outre, nous avons aussi considéré le problème inverse, c'est-à-dire, le calcul d'une représentation par un terme cyclique d'un arbre régulier et établi l'isomorphisme entre les deux représentations.

La troisième contribution de cette thèse porte sur les morphismes d'arbres réguliers. Nous avons utilisé le formalisme des transducteurs d'arbres descendants pour modéliser la syntaxe

abstraite d'un sous-ensemble des fonctions co-récursives. Ensuite, les transducteurs d'arbres sont réifiés en morphismes d'arbres préservant la propriété de régularité. Dans ce contexte, nous avons étudié trois variantes de transducteurs d'arbres d'expressivité syntaxique croissante. La première variante est une extension immédiate des transducteurs d'arbres classiques sur les arbres finis vers les arbres infinis, appelée *transducteurs d'arbres descendants gardés*. Dans ce formalisme, nous imposons une restriction sur la partie droite des règles de réécriture afin d'assurer la productivité des règles *par construction*. Puis, nous avons étendu ce formalisme des *transducteurs d'arbres gardés avec l'ajout de règles* ε, c'est-à-dire, avec des règles de réécriture offrant la possibilité d'exprimer que la racine de l'arbre en entrée n'est pas consommée. Ces nouvelles règles de réécriture peuvent être considérées comme étant purement productives. La dernière variante considérée, nommée *transducteur d'arbres à observation finie*, étend les règles de réécriture avec la possibilité d'utiliser un contexte fini avant de produire une valeur. Par ailleurs, cette extension généralise le formalisme des règles ε dans le sens où l'on autorise la spécification dans les règles de réécriture de la partie du contexte qui est consommée. Enfin, nous avons aussi considéré le problème de la représentation des transducteurs binaires, c'est-à-dire, des transformations opérant sur deux arbres simultanément. Dans ce but, nous avons préalablement défini une *opération d'internalisation du produit d'arbres* consistant à transformer un couple d'arbres, en un arbre sur une signature produit. Ainsi, un transducteur binaire est obtenu en pré-composant un transducteur unaire avec ce produit interne. Enfin, nous avons prouvé que les morphismes d'arbre induits par chacune des variantes préservent la propriété de régularité.

La dernière contribution de cette thèse concerne l'application du cadre théorique élaboré d'une part, à *l'interprétation d'un* μ*-calcul coalgébrique sur le type des arbres réguliers*, et d'autre part, à *la définition d'un opérateur de composition parallèle sur une algèbre de processus récursifs*. Dans le premier cas, nous avons prouvé que le problème de satisfiabilité d'une formule du μ-calcul est décidable sur les arbres réguliers. Dans le second cas, nous avons montré comment un problème de terminaison peut être reformulé en un problème de productivité.

Régis Spadotti

A Mechanized Theory of Regular Trees in Dependent Type Theory

PhD Advisors: Jean-Paul Bodeveix
Mamoun Filali

PhD defended on *May 19, 2016* at IRIT—Université Paul Sabatier

Abstract

We propose two characterizations of regular trees. The first one is semantic and is based on coinductive types. The second one is syntactic and represents regular trees by means of cyclic terms. We prove that both of these characterizations are isomorphic. Then, we study the problem of defining tree morphisms preserving the regularity property. We show, by using the formalism of tree transducers, the existence of syntactic criterion ensuring that this property is preserved. Finally, we consider applications of the theory of regular trees such as the definition of the parallel composition operator of a process algebra or, the decidability problems on regular trees through a mechanization of a model-checker for a coalgebraic μ-calculus. All the results were mechanized and proved correct in the COQ proof assistant.

Keywords: proof assistant, regular tree, cyclic term, coinduction, tree transducer

Informatique—Sûreté du logiciel et calcul de haute performance

Institut de Recherche en Informatique de Toulouse—UMR 5505
Université Toulouse 3 Paul Sabatier
118 route de Narbonne, 31062 TOULOUSE cedex 4

Régis Spadotti

Une théorie mécanisée des arbres réguliers en théorie des types dépendants

Directeurs de thèse : Jean-Paul Bodeveix
Mamoun Filali

Thèse soutenue le *19 Mai 2016* à l'IRIT—Université Paul Sabatier

Résumé

Nous proposons deux caractérisations des arbres réguliers. La première est sémantique et s'appuie sur les types co-inductifs. La seconde est syntaxique et repose sur une représentation des arbres réguliers par des termes cycliques. Nous prouvons que ces deux caractérisations sont isomorphes. Ensuite, nous étudions le problème de la définition de morphisme d'arbres préservant la propriété de régularité. Nous montrons en utilisant le formalisme des transducteurs d'arbres, l'existence d'un critère syntaxique garantissant la préservation de cette propriété. Enfin, nous considérons des applications de la théorie des arbres réguliers comme la définition de l'opérateur de composition parallèle d'une algèbre de processus ou encore, les problèmes de décidabilité sur les arbres réguliers via une mécanisation d'un vérificateur de modèles pour un μ-calcul coalgébrique. Tous les résultats ont été mécanisés et prouvés corrects dans l'assistant de preuve COQ.

Mots-clés : assistant de preuve, arbre régulier, terme cyclique, co-induction, transducteur d'arbres

Informatique—Sûreté du logiciel et calcul de haute performance

Institut de Recherche en Informatique de Toulouse—UMR 5505
Université Toulouse 3 Paul Sabatier
118 route de Narbonne, 31062 TOULOUSE cedex 4