Université de Toulouse

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Université Toulouse III Paul Sabatier (UT3 Paul Sabatier)

Discipline ou spécialité :
Informatique - Sûreté du logiciel et calcul de haute performance

---

Présentée et soutenue par :
Celia Picard

le : 15 juin 2012

Titre :
Représentation coinductive des graphes

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
IRIT

Directeur(s) de Thèse :
Ralph Matthes

Rapporteurs :
Ulrich Berger, Swansea University
Yves Bertot, INRIA Sophia Antipolis

Membre(s) du jury :
Jean-Paul Bodeveix, Université de Toulouse
Marie-Laure Potet, ENSIMAG, Grenoble INP
Luigi Santocanale, Université de Provence
Tarmo Uustalu, Institute of Cybernetics, Tallinn, Estonie

Celia Picard

REPRÉSENTATION COINDUCTIVE DES GRAPHES

Directeur de thèse : Ralph Matthes, *CNRS*

Résumé

Contexte général
Ce travail s'inscrit à l'interface de l'Ingénierie Dirigée par les Modèles et de la théorie des types. Dans le contexte toulousain, fortement axé vers les systèmes embarqués et critiques, savoir certifier la représentation et la transformation des modèles est un enjeu majeur. La perspective visée ici est la transformation d'un modèle conforme à un métamodèle en un modèle conforme à un autre métamodèle en assurant certaines propriétés sur le modèle d'arrivée. Deux solutions sont possibles pour cela : vérifier les propriétés sur chaque modèle d'arrivée ou certifier que l'application de la transformation assure ces propriétés. Nous avons choisi cette seconde solution. Pour la certification, nous avons décidé dans un premier temps d'utiliser un prouveur interactif, le système Coq. La première étape consiste à représenter les métamodèles, la seconde à établir un langage permettant d'exprimer les propriétés à vérifier. Dans cette thèse, nous nous intéressons à la représentation et la manipulation de métamodèles sous forme de graphes.

La représentation des graphes
Nous avons décidé de représenter les graphes par des types coinductifs dont nous voulions explorer l'utilisation dans Coq. En effet, la représentation de la coinduction dans les prouveurs basés sur la théorie des types est en progrès continuel. De plus, l'utilisation des types coinductifs permet de rendre succincte et élégante notre représentation des graphes et d'obtenir la navigabilité par construction. Nous avons dû contourner la condition de garde dont le but est d'assurer la validité des opérations effectuées sur les objets coinductifs. Son implantation dans Coq (compromis entre expressivité et maniabilité, résultat d'une longue évolution) est restrictive, et interdit parfois des définitions sémantiquement correctes. Une formalisation canonique des graphes dépasse ainsi l'expressivité directe de Coq. Nous avons donc proposé une solution respectant ces limitations, puis nous nous sommes intéressés à la définition d'une relation plus permissive sur les graphes. Celle-ci permet d'obtenir la même notion d'équivalence qu'avec une représentation classique (ensemble de nœuds/ensemble d'arêtes) tout en gardant les avantages de la coinduction. En effet, notre définition des graphes crée un ordre implicite (horizontal et vertical) entre les nœuds. Notre nouvelle relation permet de nous en affranchir. Nous montrons qu'elle est équivalente à une relation basée sur des observations finies des graphes. Ces résultats ont fait l'objet de publications et sont certifiés par des développements Coq. Toutefois, ces derniers ont été transcrits en langage mathématique et la lecture de cette thèse ne requiert pas de connaissance de Coq.

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31062 TOULOUSE cedex 4

Celia Picard

COINDUCTIVE GRAPH REPRESENTATION

Thesis Advisor : Ralph Matthes, *C.N.R.S.*

Abstract

General context
This work stands at the interface between Model Driven Engineering (MDE) and type theory. In the industrial context of Toulouse, strongly oriented towards embedded and critical systems, certifying model representation and transformation is a major issue. The perspective aimed at here is the transformation of a model conforming to a metamodel into a model conforming to another metamodel, verifying some properties on the resulting model. There are two ways to complete this last step : check that the properties are verified on each resulting model, or certify that the application of the transformation ensures these properties regardless of the input model. We have chosen this second solution. For certification, we have decided, at least in the first instance, to use an interactive theorem prover, the Coq system. The first step to realize in order to complete this project consists in representing metamodels. The second one is about establishing a language to express the properties that the resulting model should verify. In this thesis, we are only interested in the representation and manipulation of metamodels as graphs.

Graph Representation
We have chosen to represent graphs using coinductive types. We wanted to explore their use in Coq. Indeed, the representation of coinduction in the theorem provers based on type theory progresses continually. Moreover, the use of coinductive types makes our graph representation succinct and elegant and ensures navigability by construction. We had to overcome the guardedness condition whose objective is to ensure validity of all operations made on coinductive objects. Its implementation in Coq (a compromise between expressivity and maniability, resulting from a long evolution) is restrictive and sometimes forbids definitions, even semantically correct ones. A canonical formalization of graphs thus surmounts Coq's direct expressivity. We have designed a solution respecting these limitations. We then defined a wider (parameterized) relation on graphs. This new relation is close to the notion of equivalence on the classical representation of graphs (set of nodes/set of edges), but keeps the advantages offered by coinduction. Indeed, our graph definition induces an implicit order (both horizontally and vertically) between the nodes. The new relation allows us to abstract from this order. We also show that this relation is equivalent to another one based on finite observations of the graphs. All these results have been published and are formally certified by a Coq development. This development has been translated to mathematical language. Therefore, reading this thesis should not require any particular knowledge of Coq.

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31062 TOULOUSE cedex 4

À Annette, Georges, Paul et Yvonne,
mes formidables grands-parents

Remerciements

CETTE partie est curieusement celle que j'ai le plus de mal à écrire. Le plus de plaisir peut être aussi. Je veux être à la hauteur de tous ceux qui m'ont aidée, encouragée et soutenue pendant ces quatre ans. N'oublier personne, trouver les mots justes.

ET pour commencer, je voudrais remercier tous les membres de mon jury. Ulrich Berger et Yves Bertot, tout d'abord, qui m'ont fait l'honneur d'être mes rapporteurs. Merci pour votre lecture attentive, vos remarques et vos commentaires, toujours pertinents et les nouvelles pistes que vous m'avez fait entrevoir. Merci également à Marie-Laure Potet, Luigi Santocanale et Tarmo Uustalu d'avoir accepté de faire partie de ce jury. Merci plus particulièrement à Marie-Laure pour ses nombreux conseils durant mes études et lors du choix de ma thèse. Merci également à Tarmo et aux autres membres du logic and semantics group de l'Institute of Cybernetics de Tallinn (en particulier Keiko Nakata et James Chapman) pour la collaboration entamée. Merci pour l'accueil que vous m'avez réservé lors de mon séjour à Tallinn et votre disponibilité, merci enfin pour l'intérêt que vous portez à mon travail.

SI j'ai pu réaliser cette thèse, c'est surtout grâce à mon directeur de recherche, Ralph Matthes. Merci de m'avoir acceptée tout d'abord, puis guidée, soutenue, aidée, encouragée. Merci également de m'avoir fait partager un peu de vos connaissances, d'avoir eu la patience de combler mes lacunes. J'apprécie aussi que vous m'ayez fait part de vos doutes parfois, de vos inquiétudes, de vos points de vue. Toutes nos discussions, m'ont été précieuses. Et je tiens à vous exprimer ici ma gratitude et mon profond respect.

TOUT naturellement, je veux remercier également les membres de l'équipe ACADIE qui m'ont accueillie si gentiment, si facilement, que ce soit à l'UPS ou à l'ENSEEIHT. Merci à tous ceux qui étaient là quand je suis arrivée, à ceux qui sont arrivés depuis. Merci de m'avoir permis de travailler dans les meilleures conditions. Merci également à David sans qui cette thèse n'aurait probablement jamais eu lieu.

GRAND merci surtout à Jean-Paul Bodeveix qui a posé les bases de cette thèse. Sans lui, je n'aurais probablement pas du tout suivi ce chemin. Merci aussi d'avoir accepté de faire partie de mon jury.

RIEN n'est plus important, au travail, qu'une bonne ambiance. Je tiens donc à remercier tout particulièrement mes deux indéfectibles compagnons de bureau. Merci à Christelle pour ses conseils, son amitié et toutes ces longues discussions. Merci surtout à Mathieu de m'avoir supportée au quotidien, sans jamais se plaindre. Merci pour ton humour, ta générosité, ta gentillesse. Et aussi bien sûr pour ta culture informatique inépuisable que tu as accepté de partager avec la profane que je suis sans jamais perdre patience.

ASSURÉMENT, cette thèse ne se serait pas déroulée dans d'aussi bonnes conditions sans l'aide de quelques personnes formidables qui œuvrent pour que le système fonctionne bien. Je veux donc remercier ici tout le personnel administratif de l'IRIT. En particulier, Véronique et Sabyne du département communication de l'IRIT, qui ont toujours des solutions pour tout. Merci aussi au "staff" de l'EDMITT (passé et présent : Agnès, Louis, Jean-Michel)

pour la confiance qu'ils m'ont accordée et leur enthousiasme pour mes projets. Et bien sûr, merci surtout à Martine, pour sa gentillesse, son aide et ses encouragements.

CEUX qui ont pris le temps de relire (ou au moins de regarder) ma thèse, même partiellement, méritent une place de choix dans ces remerciements. Un immense merci à Julien, qui même de l'autre côté du monde m'a donné son avis et des conseils avisés, à Bertrand pour ses opinions et les discussions lexicales toujours pertinentes et à Marie pour ses conseils techniques. Merci également à tous ceux qui m'ont aidée et conseillée dans la préparation de ma présentation.

EN général, je voudrai remercier tous les membres de l'IRIT que j'ai cotoyés et appréciés de m'avoir aidée, soutenue, accueillie pendant ces quatre ans.

AU cours de ces quatre années riches professionnellement et émotionnellement, j'ai pu traverser certaines épreuves grâce à Christine, Frédéric, Victor. Merci pour votre amitié et votre soutien indéféctibles. Merci de m'avoir aidée, conseillée, fait grandir, acceptée parmi vous. J'adore et j'admire la passion que vous mettez dans tout ce que vous faites, métier y compris. Je veux également remercier tous les autres avec qui j'ai partagé des déjeuners et plus, d'avoir rendu ces quatre ans agréables et conviviaux. Merci en particulier à Jean-Paul, Jérémy, Noélie et Nicolas.

VIA mon implication dans la vie du labo j'ai eu la chance de rencontrer de nombreux doctorants que je tiens à remercier. Avec certains, une vraie amitié s'est développée : merci donc à Anke, Bénédicte et Christophe pour tout ce qu'on a partagé et ce que j'espère on partagera encore (la thèse n'est pas la fin de tout !). Et plus généralement, merci à tous les doctorants qui comme moi veulent animer la vie du labo et s'impliquent. En particulier, une pensée spéciale pour les (ex-)doctorants de la commission des doctorants pour leur enthousiasme et leur dynamisme.

ON approche maintenant de la fin de ces remerciements. Il me reste à remercier mes proches, ceux sans qui je ne serais pas moi. Je commence par les amis. Merci à tous de m'avoir soutenue et aidée pendant ces quatre ans, de vous être intéressés à mon travail. Merci aux toulousains, qui ont été là quotidiennement pendant ces quatre ans : Manu, Noémie, Stéphanie, Thomas. Merci aussi à Arthur, Emilie, Mickaël et les autres. Merci surtout à Carine et Seb, pour ce que vous êtes et ce que vous faites. Merci à ceux qui se sont déplacés pour venir m'écouter et m'encourager ce 15 juin 2012.

UNIQUE, précieuse, voici comment je décrirai ma famille. Merci donc à tous ceux qui la composent d'avoir montré un intérêt pour mon travail, d'avoir eu confiance en moi, de m'avoir encouragée. Merci tout d'abord à mes cousines, avec qui j'ai pu partager beaucoup, de près ou de loin : Alexandra, Chantal, Vanina, Noémie (merci à toi pour le coup de main !). Merci à Clovis, mon frère, de ton soutien. Merci à Shu pour l'aide pour le pot et pour tout le reste. Merci bien sûr à mon papa, pour sa confiance, sa fierté et ses encouragements. Merci à ma maman, évidemment, pour son aide, ses conseils, ses avis. Je sais que tu es contente de pouvoir partager tout cela avec moi, moi j'en suis ravie et fière. Merci de m'avoir fait grandir dans l'amour de ce métier magnifique (et de l'informatique). Merci pour la confiance que tu as en moi. Merci enfin à ma belle-famille : Annick, Jean-Lou, Adline, Loïc. Merci de m'avoir adoptée, merci de m'avoir encouragée dans mon travail. De vous y être intéressé. Merci à Annick et Jean-Lou d'avoir fait le voyage pour être avec moi à la soutenance.

SURTOUT , surtout, merci à Yann. Merci d'être celui que tu es et d'avoir rendu ma vie si belle. Merci pour ton soutien et ton amour. Merci de me comprendre si bien, de partager mes coups de cœur et mes coups de gueule. Merci de m'encourager dans tout ce que je fais et de laisser libre cours à mon (trop grand) enthousiasme. Merci, tout simplement.

Table des matières

Introduction

DANS le contexte industriel toulousain (aéronautique, aérospatiale, etc.), les besoins en logiciels certifiés sont légion. En effet, il est généralement anodin qu'un logiciel d'utilisation courante tel qu'une suite bureautique, un jeu ou un navigateur web par exemple, ait des erreurs (s'il plante, on le relance simplement, avec pour principal inconvénient l'irritation de l'utilisateur, sauf en cas de failles de sécurité, mais on s'intéresse ici principalement à la fiabilité – même si celle-ci englobe en principe la sécurité). Pour les logiciels dits critiques, (pilote automatique d'un avion ou d'un métro, logiciel embarqué dans un satellite, etc.) les erreurs peuvent avoir des conséquences beaucoup plus dramatiques, allant de la perte de sommes considérables (des milliards d'euros pour les modules spatiaux) à des pertes humaines importantes (dans des accidents d'avions, par exemple).

Classiquement, pour repérer et corriger les erreurs dans un logiciel, on le teste. Cela permet d'essayer un certain nombre de configurations, en théorie un maximum, afin de valider que le logiciel se comporte bien comme on l'attend. Mais à partir du moment où le nombre possible de valeurs pour les paramètres d'entrée est infini (par exemple si un des paramètres est un entier naturel), le test ne peut plus être exhaustif. Il faut tester des cas "typiques" (y compris les cas limites), dans l'espoir de couvrir tous les cas, mais avec le risque d'en oublier et de ne pas tout tester.

Il est donc clair que cette technique peut s'avérer trop hasardeuse pour les systèmes critiques. C'est dans ce contexte que se sont développées les méthodes formelles. Elles permettent de donner une preuve *formelle* que le logiciel répond bien à un certain nombre de spécifications et donc de certifier qu'il est *correct*. Il y a de nombreux types différents de méthodes formelles. Elles peuvent consister en des raffinements successifs d'une spécification de départ, jusqu'à arriver à une implémentation, comme dans la méthode B [3]. Elles peuvent également consister en la représentation du programme par un automate et l'exploration exhaustive de cet automate afin de vérifier que toutes les situations sont valides, comme dans le model-checking [23] (cette technique est une forme de test du code : elle vient donc à postériori, contrairement à B, qui permet d'écrire le code par raffinements successifs). Le dernier type de méthode formelle que nous mentionnons ici est la preuve interactive de programme. C'est cette technique qui va nous intéresser dans cette thèse. Elle consiste à écrire les programmes dans l'assistant de preuve puis à prouver un certain nombre de propriétés sur ces programmes. Ces preuves se font principalement interactivement, même si certains mécanismes d'automatisation existent qui permettent d'aider à la résolution. Les systèmes correspondant à cette technique sont appelés assistants de preuve, parmi lesquels se trouvent par exemple Isabelle/HOL [64], Agda [5] et Coq [25, 34].

Genèse du projet

Une des préoccupations récurrentes pour les systèmes critiques concerne la transformation de modèles. En effet, de nombreuses opérations consistent à transformer un modèle, ins-

tance d'un métamodèle (modèle de modèles), en un autre modèle, instance d'un (autre) métamodèle. Ces opérations peuvent aussi bien se faire dans un cadre logiciel (la transformation d'un programme d'un langage à un autre ou la minimisation d'un automate, par exemple) que dans un cadre informel (raffinements successifs des exigences menant à la mise en production d'un produit, tel un satellite, par exemple). Mais quel que soit leur type, pour les systèmes critiques la préoccupation actuelle est leur certification. En effet, on souhaite pouvoir vérifier et assurer que le modèle transformé possède bien un certain nombre de propriétés afin de le valider (et de valider la transformation). Pour ce faire, deux méthodes existent. On peut vérifier les propriétés au cas par cas sur le résultat de chaque transformation (de façon ad-hoc donc) ou alors certifier la transformation de façon à garantir que son application produit un résultat répondant bien aux propriétés attendues.

Le contexte global dans lequel se situe cette thèse s'inscrit dans cette deuxième démarche. L'objectif final est de proposer un langage afin d'exprimer des propriétés sur les métamodèles, de les vérifier et de certifier ainsi les transformations. Les transformations peuvent être vues comme des fonctions d'un métamodèle vers un autre. Mais la première problématique de ce projet est la représentation des métamodèles.

C'est dans cette direction précise que tend cette thèse. Nous sommes partis de l'idée qu'on pouvait représenter les métamodèles par des graphes. Et nous nous sommes donc concentrés uniquement sur la représentation des graphes.

La représentation des graphes

Nous avions, en plus de la représentation des graphes, un défi supplémentaire : explorer la puissance et les limites de la coinduction dans Coq. En effet, dans les dernières années, la représentation de la coinduction dans les prouveurs (Agda, Coq) a fait de nombreux progrès (dans Coq, notamment grâce à Giménez [43]). Nous voulions profiter de ces avancées et étudier les possibilités offertes (mais aussi les restrictions imposées), et nous avons choisi de les étudier dans le système Coq. Tout d'abord parce que Coq permet de manipuler assez efficacement les types coinductifs. Mais également parce que ce système a maintenant une certaine maturité et il a montré qu'il supportait le passage à l'échelle, notamment avec le projet CompCert [53] dans lequel un compilateur C certifié a été développé.

De plus, la coinduction nous semblait spécialement adaptée pour représenter les graphes et nous avons basé nos travaux sur ce postulat. En effet, les types coinductifs permettent une représentation succincte et élégante des graphes, proche de la représentation des arbres. Ils facilitent également la navigabilité (transition d'un nœud à un autre, repérage des cycles) dans les graphes, de la même façon que les types inductifs la facilitent dans les arbres.

Mais la coinduction dans Coq est lourdement et malheureusement nécessairement limitée par la condition de garde. Dans nos développements, nous nous sommes rapidement heurtés à cette restriction et avons dû trouver des moyens pour la contourner : étant un compromis entre expressivité et maniabilité, et le résultat d'une longue évolution, il est délicat de vouloir la modifier, d'autant que ce n'était pas du tout l'objectif de ces travaux.

Contribution

Nous avions donc comme objectif de développer une représentation des graphes qui respecte ces limitations dues à la condition de garde. Nous proposons une solution coinductive

qui est acceptée par Coq. Nous la munissons d'un bon nombre de fonctionnalités supplémentaires (notions de bisimilarité, finitude, cycles, inclusions, etc.). Nous l'étendons ensuite afin de nous approcher au plus près de la liberté offerte par la représentation "classique" (sous forme d'ensemble de nœuds et ensemble d'arcs) tout en gardant les avantages offerts par la coinduction. En particulier, cette extension passe par la définition de nouvelles relations de bisimilarité sur les graphes.

Tous les résultats présentés ont été développés avec l'assistant de preuve Coq et sont accessibles avec une liste de correspondances entre les noms utilisés dans la thèse et les noms dans les scripts, ainsi que les fichiers correspondants et les lignes dans ces fichiers (voir Annexe B et [71]). Au total, cela représente un peu plus de 12500 lignes de code Coq, dont plus de 3000 lignes de définition et 7700 lignes de preuve. Les résultats obtenus sont entièrement certifiés. Seule une partie a nécessité l'ajout d'un axiome. Son utilisation est largement discutée et justifiée dans ce texte.

Nous nous sommes cependant attachés à transcrire tout le code présenté dans cette thèse en langage mathématique, si bien que la lecture de ce manuscrit ne demande pas de connaissance particulière de Coq.

Survol du manuscrit

Ce manuscrit est composé de trois grandes parties, chacune composée de deux chapitres.

Dans la première partie, nous dressons l'état des lieux des sujets abordés dans cette thèse. Le Chapitre 1 introduit les notions de "base" nécessaires à la compréhension de ce document : coinduction, types dépendants, et nous proposons également une introduction à Coq, puisque c'est l'outil que nous avons utilisé pour développer les résultats présentés ici, même si comme nous l'avons dit, la lecture de ce document ne requiert pas de connaissance de Coq. Dans le Chapitre 2, nous présentons les travaux connexes aux nôtres. Nous avons divisé ce chapitre en deux sections. Dans la première, nous présentons les travaux relatifs à la représentation des arbres et des graphes, objectif principal de cette thèse. Puis nous proposons une première implémentation des graphes et montrons comment nous sommes immédiatement bloqués par la condition de garde, principalement parce que nous essayons de mélanger types coinductifs et listes. La deuxième section de ce chapitre étudie donc différentes méthodes proposées pour contourner la condition de garde, particulièrement dans Coq, mais aussi dans Agda.

Dans la Partie II nous présentons l'outil qui va nous permettre de contourner la condition de garde dans la suite (au moins au niveau des définitions) : il s'agit de la version conteneur des listes [75] que nous appelons *ilist*. Cette partie présente une bibliothèque, indépendante du reste mais développée entièrement dans le but de représenter ensuite les graphes. Nous avons néanmoins tenté de la développer de la façon la plus générale possible afin qu'elle puisse servir dans d'autres contextes. Elle peut paraître parfois très technique, même si nous avons tenté d'alléger cet aspect en mettant une grande partie des preuves en annexe, afin de ne pas nuire à la fluidité de lecture. Elle représente cependant une partie importante du travail effectué.

Dans le Chapitre 3 (premier chapitre de la Partie II), nous définissons le type *ilist*, montrons qu'il est équivalent aux listes et le munissons de nombreux outils qui facilitent sa manipulation, en particulier des outils classiques des listes (relation d'équivalence canonique, quantification universelle, équivalent pour *map*, concaténation, etc.). Dans le Cha-

pitre 4, nous cherchons à définir une relation d'équivalence sur ces *ilist* qui capture la notion de permutations tout en supposant que les éléments de ces *ilist* sont comparables par une relation non nécessairement décidable. Nous commençons donc par étudier les représentations des permutations sur les listes dans Coq, puis nous proposons plusieurs solutions pour les *ilist*. Nous proposons plusieurs définitions inductives qui assurent différentes propriétés par construction et une relation utilisant des fonctions bijectives. Nous montrons finalement que ces relations sont équivalentes entre elles et équivalentes avec une des relations sur les listes.

Enfin, dans la Partie III, nous présentons notre représentation coinductive des graphes. Dans le Chapitre 5, nous utilisons *ilist* pour définir nos graphes et nous les munissons de nombreuses fonctionnalités telles qu'une relation de bisimilarité canonique, des notions d'inclusion, de cycle et de finitude. Nous donnons également quelques exemples d'utilisation (ces exemples restant pédagogiques dans un objectif d'illustration du propos, et non tirés de cas réels). Nous constatons ensuite dans le Chapitre 6 que la relation canonique de bisimilarité définie précédemment peut dans certains cas s'avérer trop restrictive parce qu'elle induit un ordre dans les nœuds, aussi bien horizontal (entre les fils d'un nœud) que vertical (dans le choix de la racine). Nous introduisons donc de nouvelles relations sur les graphes qui permettent d'assouplir ces ordres. Nous proposons d'abord une relation qui permet de supprimer l'ordre horizontal, grâce aux relations de permutations définies précédemment sur *ilist*. Nous recontrons de nouveau des problèmes avec la condition de garde, au niveau de la définition de propriétés cette fois-ci et nous proposons différentes méthodes pour résoudre ce problème (avec des définitions imprédicatives et des observations). Enfin, nous proposons une dernière relation qui permet de supprimer à la fois l'ordre horizontal et l'ordre vertical dans les nœuds.

Première partie

Etat des lieux

1 Introduction des concepts

DANS cette partie nous allons introduire les concepts de base nécessaires à la compréhension de cette thèse. Dans ce travail, nous avons étudié, développé et certifié la représentation des graphes à l'aide de types coinductifs dans l'assistant de preuve Coq. Nous allons donc dans un premier temps introduire le concept de coinduction et types coinductifs. Nous verrons ensuite ce qu'est la programmation avec les types dépendants, qui est grandement intervenue dans notre travail. Enfin, nous présenterons l'outil Coq en entrant dans les détails de la coinduction dans Coq et des défis qu'elle présente.

Mais pour commencer, nous allons rappeler quelques concepts de base qui interviendront à plusieurs reprises dans le travail présenté ainsi que les notations utilisées.

1.1 Concepts de base

1.1.1 Égalité de Leibniz

À de nombreuses reprises nous mentionnerons l'égalité de Leibniz. Cette égalité est une approximation de l'égalité standard en mathématiques. On dit que deux éléments sont égaux par l'égalité de Leibniz si on peut remplacer l'un par l'autre partout où ils apparaissent : c'est le principe de substitution. En général, ce principe n'est pas vrai pour les autres relations, à moins de le démontrer dans des cas précis (morphismes). Il est important de noter que l'égalité de Leibniz n'est pas extensionnelle.

L'égalité de Leibniz est une relation d'équivalence, c'est-à-dire qu'elle est réflexive, symétrique et transitive. Nous utiliserons beaucoup ces résultats dans la suite. Pour faire référence à l'égalité de Leibniz (par exemple comme relation de base), nous la noterons *eq*. Et si on a H un élément de type $x = y$, on notera *sym H* son symétrique, c'est-à-dire un élément du type $y = x$.

L'égalité de Leibniz est cependant bien souvent trop fine pour nos besoins (en particulier comme on va le voir dans le cas des types coinductifs ou lorsque l'on compare deux fonctions) et nous serons amenés à définir de nombreuses relations dans la suite.

1.1.2 Notations

Dans la suite, nous utilisons un certain nombre de notations pour représenter les différents éléments que l'on manipule. Nous les introduisons ici :

- T, U, V pour des types et t (resp. u et v) pour des éléments du type T (resp. U et V),
- n, m et k pour des entiers naturels ($\mathbb{N}$),
- l et q pour des listes et des éléments de *ilist*, defini au Chapitre 3,
- f (et parfois g, lorsque cela ne prête pas à confusion) pour des fonctions,
- g pour des éléments de *Graph* défini au Chapitre 5,

- R pour des relations (si R est une relation sur un type T, elle est de type $T \to T \to Prop$, où $Prop$ est l'ensemble des propositions de Coq dont nous parlerons plus en détail à la Section 1.4.1.1),
- R_{eq} si la relation R est (ou doit être si c'est une hypothèse) une relation d'équivalence,
- P pour des prédicats (si P est un prédicat sur un type T il est de type $T \to Prop$)
- i pour des éléments de $Fin\ n$, défini au Chapitre 3.

1.2 Coinduction et types coinductifs

Pour parler des types coinductifs, il faut tout d'abord revenir aux types inductifs.

1.2.1 Types inductifs

En général, ces types permettent de construire une infinité d'éléments finis (les énumérations sont une exception à cela puisqu'elles ne permettent de construire qu'un nombre fini d'éléments : les éléments de l'énumération). Les entiers naturels, les listes d'éléments d'un type T (si les éléments de T sont finis), les arbres finis sont des exemples de ces types.

Ces types de données peuvent être vus comme le plus petit point fixe d'un opérateur monotone sur un treillis complet d'ensembles [56] (l'existence de ce plus petit point fixe est garantie par le théorème de Tarski [80]).

Les types inductifs sont définis par des constructeurs, parmi lesquels doit se trouver un cas de base qui permet d'initier la récurrence (s'il n'y a pas de cas de base, le type est vide puisqu'on ne peut construire aucun élément, ce qui est cependant admissible). Certains constructeurs peuvent également être récursifs (c'est-à-dire faire appel à des éléments du type qui est en train d'être défini). Typiquement, on peut définir les listes à l'aide des deux constructeurs suivants :

Définition 1.1 (Type *list*)**.** $\dfrac{}{[] : list\ T}$ $\qquad$ $\dfrac{e : T \qquad l : list\ T}{e::l : list\ T}$

Remarque 1.1. *On note les définitions inductives sous forme de règles d'inférences dont les hypothèses et la conclusion sont séparées par un simple trait. Dans certains cas, la notation sous cette forme peut rendre la définition moins lisible. On l'abandonnera alors au profit d'une notation mathématique plus "classique".*

Ici, le constructeur [] est le constructeur de base et :: est un constructeur récursif (car il prend en paramètre un élément de *list*).

Les éléments d'un type inductif sont généralement créés par une séquence d'applications des constructeurs à partir du cas de base . Ainsi, par exemple 1::4::5::[] est un élément de *list* $\mathbb{N}$ résultant de trois applications successives du constructeur :: au constructeur [].

On peut aussi donner l'exemple des arbres binaires (non vides) d'entiers qui peuvent être définis à l'aide des deux constructeurs suivants :

Définition 1.2 (Type *Abin*)**.**

$$\frac{n : \mathbb{N}}{Leaf\ n : Abin} \qquad \frac{f_g : Abin \qquad r : \mathbb{N} \qquad f_d : Abin}{Node\ f_g\ r\ f_d : Abin}$$

Le constructeur *Leaf* permet de construire une "feuille" de l'arbre d'étiquette n et le constructeur *Node* permet de construire un nœud de l'arbre d'étiquette r, de fils gauche f_g et de fils droit f_d.

Dans certains cas, le cas de base peut être "caché" dans un cas inductif. Si on prend par exemple le cas des arbres n-aires (c'est-à-dire dont le dégré sortant d'un nœud est variable dans l'arbre), on pourrait le représenter avec les deux constructeurs suivants :

$$\frac{\mathrm{n} : \mathbb{N}}{\mathit{Leaf}\ n : \mathit{Anaire}} \qquad \frac{r : \mathbb{N} \qquad l : \mathit{list\ Anaire}}{\mathit{Node}\ r\ l : \mathit{Anaire}}$$

Mais quelle serait alors concrètement la différence entre *Leaf* 1 et *Node* 1 [] ? Comment choisir quel cas appliquer comme cas de base ? En fait, ici, le constructeur *Leaf* devient tout à fait superflu (il ajoute de même l'ambiguïté), et on peut donc simplement définir les arbres n-aires à l'aide du constructeur suivant :

Définition 1.3 (Type *Anaire*)**.** $\dfrac{r : \mathbb{N} \qquad l : \mathit{list\ Anaire}}{\mathit{Node}\ r\ l : \mathit{Anaire}}$

Ici, on a un mélange entre deux types inductifs (*Anaire* fait appel à *list* qui est lui même un type inductif) et son cas de base est en fait généré à partir du cas de base de *list* : un nœud est une feuille s'il n'a aucun fils, c'est-à-dire si $l = []$.

Pour raisonner par induction sur un type inductif, on doit avoir un principe d'induction. Pour les cas simples, comme *list* ou *Abin*, le principe est obtenu immédiatement. Ce principe d'induction est une généralisation du principe de récurrence sur les entiers, bien connu qui peut s'exprimer ainsi (pour une proposition P sur les entiers) :

$$P\ 0 \wedge (\forall n, P\ n \Rightarrow P\ (n+1)) \Rightarrow \forall n, P\ n$$

Ainsi, par exemple, le principe d'induction sur les listes peut être formalisé comme suit (pour un type de base T et une proposition sur les listes P) :

$$P\ [] \wedge (\forall t\ l, P\ l \Rightarrow P\ (t :: l)) \Rightarrow \forall l, P\ l$$

En revanche, sur les types plus compliqués, en particulier ceux qui ont d'autres types inductifs imbriqués (comme *Anaire* par exemple), le principe d'induction est moins trivial. En effet, si on suit mécaniquement la même logique que précédemment, le principe d'induction que l'obtiendrait serait (pour une proposition P sur *Anaire*) :

$$(\forall r\ l, P\ (\mathit{Node}\ r\ l)) \Rightarrow \forall t, P\ t$$

Mais ce principe n'apporte rien, puisqu'il n'a en fait rien d'inductif. En effet, dans *Anaire* la récursion est imbriquée dans un type inductif (les listes). Le principe d'induction doit donc lui aussi se servir de l'induction sur les listes. On peut le définir ainsi (pour une proposition P sur *Anaire*) :

$$(\forall r\ l, \widehat{P}\ l \Rightarrow P(\mathit{Node}\ r\ l)) \Rightarrow \forall t, P\ t$$

où $\widehat{P}$ est une proposition sur *list Anaire* qui correspond à la conjonction des $P\ t$ pour tous les éléments t de la liste (le "lifting" de P aux listes).

Pour tester l'égalité de deux éléments d'un même type inductif on peut tout simplement utiliser l'égalité de Leibniz. En effet, pour être égaux, il faut que les éléments aient été construits avec les mêmes constructeurs et que les paramètres soient égaux. Bien entendu,

cela sous-entend que les paramètres des constructeurs peuvent être comparés par l'égalité de Leibniz. Si ce n'est pas le cas, il faut redéfinir une relation sur le type qui prend en paramètre une relation sur les éléments afin de pouvoir les comparer. Ainsi, dans la suite, on aura besoin de comparer des listes d'éléments, dont le type n'est pas comparable par l'égalité de Leibniz. On définit donc une relation sur *list* qui prend cette contrainte en compte. On note R la relation sur T :

Définition 1.4 (*list_rel*)**.**

$$\frac{}{list\_rel\ R\ []\ []} \qquad \frac{R\ t_1\ t_2 \qquad list\_rel\ R\ q_1\ q_2}{list\_rel\ R\ (t_1{::}q_1)\ (t_2{::}q_2)}$$

1.2.2 Types coinductifs

Les types coinductifs sont le dual des types inductifs. Ainsi, les types coinductifs peuvent être vus comme le plus grand point fixe d'un opérateur monotone sur un treillis complet d'ensembles (là où les types inductifs sont vus comme le plus petit point fixe). L'existence de ce point fixe est ici encore assuré par le théorème de Tarski. Les types coinductifs permettent donc en général de créer une infinité d'éléments (potentiellement) infinis. Contrairement aux éléments de types inductifs, les éléments de types coinductifs ne peuvent pas être construits par application successive de constructeurs. En effet, pour cela on aurait besoin de "commencer" quelque part, c'est-à-dire comme pour les types inductifs, on aurait besoin de s'appuyer sur un cas de base. Mais dans un type avec des éléments infinis, on n'en a pas toujours. Nous avons donc besoin de la notion duale à la notion de constructeur. C'est la notion de "destructeur". En effet, au lieu de construire un élément à partir de constructeurs (dont le constructeur de base), on le décompose, afin de pouvoir raisonner dessus. Dans Coq, on suit la pratique des langages de programmation qui consiste à définir les types par des constructeurs, qui correspondent à l'inverse de couples de "destructeurs".

Pour illustrer, l'exemple classique d'un type coinductif est le type des listes infinies, généralement appelée *Stream*. Soit, T un type de base, *Stream* est définie par le constructeur suivant :

Définition 1.5 (Type *Stream*)**.** $$\frac{t : T \qquad s : Stream\ T}{t\ ::\ s : Stream\ T}$$

Remarque 1.2. *Les notions de list et de Stream étant très proches l'une de l'autre, on surcharge la notation* :: *pour les Stream. De plus, on pourra facilement inférer le type de* :: *(c'est-à-dire s'il s'agit de celui des list ou de celui des Stream) en fonction de ses paramètres.*

Remarque 1.3. *Similairement à ce que l'on a fait pour les définitions inductives, on note les définitions coinductives sous forme de règle d'inférence. La coinduction est dénotée par la double ligne qui sépare les hypothèses de la conclusion (contre une simple ligne dans le cas des définitions inductives), comme dans [62].*

Le type *Stream* nous permet uniquement de construire des éléments infinis. Par exemple, on peut définir la *Stream* de tous les entiers naturels. On commence par définir la *Stream* de tous les entiers supérieurs à un n donné en paramètre :

$$alln\ n := n :: alln(n+1)$$

La *Stream* de tous les entiers naturels est bien sûr définie par $alln\ 0$.

Dualement à la notion d'induction pour les types inductifs, on a la notion de coinduction pour les types coinductifs. C'est grâce à cette notion que l'on pourra effectuer des raisonnements sur des éléments de types coinductifs. Un raisonnement par coinduction correspond en fait à un raisonnement par plus grand point fixe. Contrairement à un raisonnement par induction, on n'a en général pas de cas de base sur lequel appuyer notre raisonnement. L'hypothèse de coinduction est donc simplement une copie de l'hypothèse de départ. Mais on ne peut l'utiliser qu'avec un "sous-terme" du terme initial.

Les éléments que l'on représente étant (potentiellement) infinis, l'égalité de Leibniz est trop stricte pour les comparer entre eux : exiger qu'on puisse remplacer l'un par l'autre est une restriction trop forte. En effet, par exemple, si on définit *alln'* de la façon suivante :

$$alln'\ n := n :: n+1 :: alln'(n+2)$$

Il est clair que *alln* et *alln'* contiennent les mêmes éléments. Mais comment le prouver ? Pour prouver que $alln\ n = alln'\ n$, on devrait montrer que $n :: (alln(n+1)) = n :: n+1 :: (alln'(n+2))$, c'est-à-dire que $n = n$ (c'est trivial) et $alln(n+1) = n+1 :: (alln'(n+2))$ c'est-à-dire encore que $n+1 :: alln(n+2) = n+1 :: alln'(n+2)$, c'est-à-dire finalement $alln(n+2) = alln'(n+2)$. On voit qu'on peut continuer comme cela indéfiniment sans arriver au résultat. On doit "simuler" les éléments un à un, mais il faut qu'on puisse s'arrêter, ce que l'égalité de Leibniz ne nous permet pas. On voudrait simplement pouvoir dire que les éléments sont simulablement égaux. On doit donc définir des relations de bisimilarité pour comparer les éléments des types coinductifs. Ces relations doivent être définies coinductivement également (de la même façon que lorsqu'on définit une relation sur un type inductif, on la définit inductivement, comme *list_rel* par exemple). Cela nous permettra de raisonner par coinduction.

Ainsi, par exemple on peut définir une relation sur *Stream* de la façon suivante :

Définition 1.6 (*EqSt*)**.**

$$\frac{t_1\ t_2 : T \quad l_1\ l_2 : Stream\ T \qquad t_1 = t_2 \qquad EqSt\ l_1\ l_2}{EqSt\ (t_1 :: l_1)\ (t_2 :: l_2)}$$

Grâce à cela, la preuve d'équivalence entre *alln n* et $alln'\ n$ se fait aisément. On prouve que $EqSt\ (alln\ n)\ (alln'\ n)$.

Démonstration. L'hypothèse de coinduction est $CH : \forall n, EqSt\ (alln\ n)\ (alln'\ n)$ On a :

$$\begin{aligned}
EqSt\ (alln\ n)\ (alln'\ n) &\Leftrightarrow EqSt\ (n :: (alln(n+1)))\ (n :: n+1 :: (alln'(n+2))) \\
&\Leftrightarrow n = n \wedge EqSt\ (alln(n+1))\ (n+1 :: (alln'(n+2))) \\
&\Leftrightarrow EqSt\ (n+1 :: alln(n+2))\ (n+1 :: (alln'(n+2))) \\
&\qquad \text{(On supprime la partie triviale)} \\
&\Leftrightarrow n+1 = n+1 \wedge EqSt\ (alln(n+2))\ (alln'(n+2)) \\
&\Leftrightarrow EqSt\ (alln(n+2))\ (alln'(n+2)) \\
&\qquad \text{(On supprime la partie triviale)}
\end{aligned}$$

Et $EqSt\ (alln(n+2))\ (alln'(n+2))$ est vrai d'après *CH* (on peut utiliser *CH* car l'expression que l'on veut prouver résulte du dépliage par rapport à *EqSt* de l'expression initiale). □

1.3 Programmation avec les types dépendants

Une grande partie de notre développement repose sur la programmation avec les types dépendants. C'est-à-dire sur la construction de structures et d'algorithmes reposants sur les types dépendants.

Dit simplement, un type dépendant est un type qui dépend d'une valeur. Ainsi par exemple, en logique propositionnelle, lorsqu'on énonce une proposition on crée en réalité un type qui est habité par les preuves qui rendent cette proposition vraie. Ces propositions sont paramétrées par les axiomes qui les composent. Ce premier exemple de type dépendant est très simple et presque transparent pour l'utilisateur. Néanmoins, on ne peut pas vraiment parler de programmation avec les types dépendants ici puisqu'on n'utilise pas ces types pour construire des structures et des algorithmes (dans l'univers des ensembles). On reste complètement dans l'univers des propositions.

Il existe cependant des types dépendants plus compliqués à manipuler. Par exemple, le type des tableaux à n cases est un type dépendant puisqu'il dépend de la valeur n. Pour illustrer, on définit un tableau d'éléments de T comme un type inductif dépendant de la valeur de n :

$$\frac{t : T}{(t) : tab\ T\ 0} \qquad \frac{n : \mathbb{N} \qquad t : T \qquad ta : tab\ T\ n}{(t, ta) : tab\ T\ (n+1)}$$

Les exemples peuvent se complexifier à loisir mais dans nos développements, les types dépendants que nous utiliserons, dans l'univers des ensembles (en opposition à celui des propositions), seront paramétrés par des entiers, comme *tab*.

1.4 Coq

Le système Coq est un assistant de preuve qui implémente la théorie des types intentionnelle. Il est basé sur le calcul de constructions inductives et coinductives (CIC) [27]. Il supporte assez efficacement les types coinductifs et les raisonnements qui leurs sont associés. C'est en partie pour cela que nous avons choisi de travailler avec ce système. Notons cependant que d'autres assistants de preuves permettent de manipuler les types coinductifs, entre autres on peut citer Agda et CIRC [22, 54, 76]. CIRC est un prouveur basé sur Maude [57] qui propose entre autre une automatisation des preuves coinductives circulaires. Cet outil est encore très récent puisque la toute première version est sortie en 2007 [55].

Tout le travail présenté a donc été développé et certifié avec Coq. Rappelons cependant que la lecture de ce manuscrit ne requiert à priori aucune connaissance de Coq puisque toutes les définitions, les lemmes et les démonstrations ont été traduits vers un langage mathématique plus universel. Il nous semble quand même primordial de présenter cet outil ici, dans la mesure où les preuves traduites suivent le schéma de la preuve initiale. Il nous paraît donc important d'expliquer le fonctionnement de quelques tactiques, afin d'éclairer certains raisonnements. D'autre part, une partie non négligeable du travail a consisté à contourner la condition de garde de Coq pour les types coinductifs. Nous expliquerons donc comment la coinduction est implémentée dans Coq et ce qu'est la condition de garde.

Cette introduction est cependant partielle, pour plus de détails, voir le très complet [14].

1.4.1 Présentation de Coq et concepts généraux

Sans faire une présentation exhaustive et technique de Coq, nous allons ici présenter quelques principes de fonctionnement et tactiques de preuve.

1.4.1.1 Présentation générale

Coq permet de définir des objets puis de prouver des propriétés sur ces objets. On peut définir des objets dans trois univers (trois sortes) : dans *Set*, c'est-à-dire l'univers des ensembles qui est principalement pour les types de données de base, dans *Prop*, c'est-à-dire l'univers des propositions qui a pour particularité d'être imprédicatif (*Set* est prédicatif par défaut depuis la version 8.0 de Coq) et dans *Type* pour les types de données plutôt d'une nature méta. *Set* et *Prop* sont inclus dans *Type*.

Pour des définitions inductives [67] (resp. coinductives) on utilise le mot clé `Inductive` (resp. `CoInductive`). Comme on l'a dit, ces définitions nécessitent des constructeurs. Ainsi, par exemple les listes définies précédemment sont décrites en Coq par le code suivant :

```
Inductive list (T : Type) : Type :=
    nil : list T | cons : T -> list T -> list T
```

Remarque 1.4. *On peut ensuite définir des notations infixées pour les opérateurs. Ainsi, on utilisera dorénavant* `t :: q` *pour* `Cons t q`*. On ne détaille pas la syntaxe de ces définitions ici.*

Pour des définitions récursives (resp. corécursives) on utilise le mot clé `Fixpoint` (resp. `CoFixpoint`). Ainsi, par exemple le foncteur *map* bien connu se définit ainsi :

```
Fixpoint map (T U: Type)(f: T -> U)(l: list T) : list U :=
  match l with nil => nil | t :: q => f t :: map f q end.
```

Remarque 1.5 (Arguments implicites)**.** *Dans l'appel récursif on peut écrire simplement* `map f q` *et non pas* `map T U f q` *parce que Coq peut inférer tout seul des arguments implicites (avec l'option* `Set Implicit Arguments`*). Ici T et U sont déduits simplement du type de* `f`*. Cela permet d'alléger notablement les notations. Nous utilisons beaucoup cette fonctionnalité dans la suite.*

Finalement, pour des définitions simples (non inductives, non récursives), on utilise le mot clé `Definition`. Ainsi par exemple la fonction qui teste si une liste est vide s'écrit :

```
Definition is_nil (T: Type)(l: list T) : Prop := l = nil.
```

Remarque 1.6. *Dans cette thèse, lorsqu'on parle de l'égalité de Leibniz on se réfère en fait exactement au '=' de Coq, qui est une égalité propositionnelle. Elle est définie par :*

```
Inductive eq (A: Type) (x: A) : A -> Prop := eq_refl : @eq A x x
```

et le principe d'induction associé est :

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop), P x ->
  forall y : A, x = y -> P y
```

Ce principe assure le principe de substitution par construction. L'égalité obtenue est donc une égalité de Leibniz, même si ce n'est pas la version imprédicative originale définie par $\forall P, (P\ x \Leftrightarrow P\ y)$ *pour exprimer l'égalité entre* x *et* y*.*

En plus de simplement définir des structures et des fonctions, on peut également démontrer des résultats sur ceux-ci. On peut ainsi énoncer des lemmes et des théorèmes entre autres avec les mots clés `Lemma` et `Theorem`, qui sont en fait simplement des synonymes (au sens de Coq) de `Definition`. La syntaxe générale est la suivante :

```
Lemma nom_du_lemme arguments : objectif.
```

En fait, on veut fournir un élément du type `objectif`. Par exemple, on peut montrer que pour toute fonction `f`, `map f nil` est vide :

```
Lemma map_nil (T U: Type): forall (f : T -> U), is_nil (map f nil).
```

Voyons donc maintenant comment se déroule une preuve en Coq. De façon générale, une preuve Coq a cette allure-ci :

```
Lemma nom_du_lemme arguments : objectif.
Proof
  (* commentaires *)
  séquence de tactiques séparées par des ';' ou des '.'
Qed.
```

La commande `Proof` est là par hygiène. Elle n'a aucun rôle. La commande `Qed` sert à enregistrer le résultat dans le système. A sa lecture le système effectue un certain nombre de vérifications : il vérifie qu'aucun but n'a été oublié, que toutes les variables existentielles ont été instanciées, que les conditions de garde, de stricte positivité sont respectées, etc. Il se peut donc qu'une preuve semble terminée, c'est-à-dire qu'il ne reste rien à prouver (Coq affiche alors le message `Proof completed`), mais qu'elle ne soit pas acceptée par le système. On ne peut considérer une preuve comme achevée qu'une fois la commande `Qed` acceptée.

Lorsqu'on fait lire la déclaration du lemme par le système, le système stocke alors en mémoire les hypothèses et garde comme objectif (goal) l'élément à prouver. S'il y a des quantificateurs, les quantificateurs restent avec l'objectif. Ainsi, par exemple, pour le lemme `map_nil`, le système enregistre `T` et `U` comme des hypothèses et on doit prouver que `forall (f : T -> U), is_nil (map f nil).` À un moment de la preuve, l'objectif initial peut se diviser en plusieurs sous-objectifs (lors d'une induction par exemple). On les prouve alors successivement. Concrètement, dans Coq les hypothèses et les objectifs sont représentés comme suit :

```
n subgoals
Hypothèse 1 : Type 1
...
Hypothèse m : Type m
_____________________________________(1/n)
Goal 1
...
_____________________________________(n/n)
Goal n
```

La première ligne indique le nombre de sous-objectifs à prouver. Viennent ensuite les hypothèses. Seules celles du sous-objectif "actif" (c'est-à-dire celui qu'on est en train de prouver) apparaissent. La ligne horizontale sépare les hypothèses de l'objectif. Ainsi, voici ce à quoi ressemble le contexte, une fois la ligne de déclaration de `map_nil` "digérée" par le système :

```
1 subgoal
T : Type
U : Type
______________________________________(1/1)
forall f : T -> U, is_nil (map f nil)
```

On peut également définir une notion de façon interactive. C'est-à-dire qu'au lieu de donner une définition du type :

```
Definition nom arguments : type := definition.
```

On ne donne pas explicitement la définition, on la construit interactivement :

```
Definition nom arguments : type.
Proof
  (* commentaires *)
  séquence de tactiques séparées par des ';' ou des '.'
Defined.
```

Dans ce cas, on doit terminer la preuve (de la définition) par `Defined` qui vérifie et enregistre la définition dans le système comme `Qed`. Mais là où `Qed` opacifie la preuve (lorsqu'on démontre un lemme, ce qui compte c'est le résultat, pas le terme de preuve), `Defined` la laisse visible ce qui permet de l'utiliser (dans une définition, le type de l'élément créé ne suffit pas : il faut également pouvoir obtenir sa valeur, c'est-à-dire le calcul qui y mène).

Il est important de souligner que les preuves en Coq se font en partant du but à atteindre. On le manipule, le transforme, le modifie jusqu'à obtenir quelque chose de connu (typiquement, une hypothèse). C'est-à-dire qu'on démontre, à l'envers, qu'on peut obtenir l'objectif à partir des hypothèses de la preuve. On raisonne en sens inverse de la démarche mathématique classique. Il faut bien garder cette idée à l'esprit, parce que lors de la transcription de nos preuves en langage mathématique, nous avons conservé cette façon de travailler.

Ce preuves s'effectuent à l'aide de tactiques. Nous en présentons quelques-unes ici.

1.4.1.2 Tactiques

Les tactiques que nous allons présenter ici sont celles qui reviennent le plus souvent dans nos preuves et qui nous paraissent pouvoir éclairer nos raisonnements.

`intros` et `revert` Ces tactiques permettent d'introduire des hypothèses dans le contexte (lorsqu'elles sont quantifiées par un $\forall$) ou au contraire de les supprimer du contexte en les réinjectant dans l'objectif. Ainsi, par exemple, voici les évolutions du contexte et de l'objectif à l'application de ces deux tactiques pour le lemme `map_nil` :

```
intros f.
```

```
1 subgoal
T : Type
U : Type
f : T -> U
______________________________________(1/1)
is_nil (map f nil)
```

```
revert f.
```
```
1 subgoal
T : Type
U : Type
_____________________________________(1/1)
forall f : T -> U, is_nil (map f nil)
```

Ce dernier exemple est donné à titre indicatif, on supposera dorénavant qu'on a introduit `f` dans le contexte.

`simpl` et `unfold` `simpl` permet de simplifier les formules affichées (c'est une transformation purement visuelle, les seules opérations appliquées sont les conversions par rapport à l'égalité définitionnelle qui est décidable). Parfois, pour "déplier" les définitions (c'est-à-dire remplacer le nom par l'expression qu'il représente), un simple `simpl` ne suffit pas. Il faut alors faire appel à la tactique `unfold` (la syntaxe est `unfold nom_de_la_def.`). Voici par exemple, le résultat de ces deux tactiques sur notre contexte précédent :

```
simpl.
```
```
1 subgoal
T : Type
U : Type
f : T -> U
_____________________________________(1/1)
is_nil nil
```

```
unfold is_nil.
```
```
1 subgoal
T : Type
U : Type
f : T -> U
_____________________________________(1/1)
nil = nil
```

`apply` Nous allons maintenant parler des tactiques qui agissent sur l'objectif, c'est-à-dire celles qui permettent de faire la preuve. Pour effectuer une preuve on va enchaîner des séquences d'applications de résultats et de réécriture d'hypothèses. Commençons donc par l'application de résultats (c'est-à-dire de lemmes, théorèmes, etc.). Un résultat est curryfié sous la forme `nom : param1 -> param2 -> ... -> paramn -> conclusion`. Les paramètres peuvent être des éléments d'un type (du type *list* par exemple) mais également des preuves. "Appliquer" un résultat dans Coq, revient à prouver les hypothèses qui sont des preuves (un sous-objectif par hypothèse est créé). Par exemple, dans Coq on a un lemme (qui fait partie de la librairie standard) qui dit que l'égalité de Leibniz est réflexive. Ce lemme s'appelle `eq_refl` et son énoncé est : `forall (A: Type) (x: A), x = x`. Dans ce lemme, `A` est un argument implicite. On peut donc utiliser ce lemme pour terminer la preuve de `map_nil` :

```
apply (eq_refl nil).
```
```
Proof completed.
```

On peut aussi simplement écrire `apply eq_refl`, parce que Coq infère le paramètre `x` tout seul. Pour terminer la preuve on doit enregistrer le lemme avec `Qed`. Coq nous répond alors `map_nil is defined`.

Remarque 1.7. *La tactique `reflexivity` est équivalente à `apply eq_refl`. De la même façon, la tactique `symmetry` applique la symétrie de l'égalité de Leibniz et `transitivity` la transitivité (il faut fournir à celle-ci l'élément "du milieu" au travers duquel s'applique la transitivité).*

`rewrite` Soit H une hypothèse de type $a = b$, la tactique `rewrite H` permet de remplacer toutes les occurrences de a par b dans l'objectif. Pour illustrer, démontrons le lemme :

```
Lemma map_eq (T U: Type)(f: T -> U)(l1 l2: list T) :
  l1 = l2 -> map f l1 = map f l2.
```

	1 subgoal T : Type U : Type f : T -> U l1 : list T l2 : list T _____(1/1) l1 = l2 -> map f l1 = map f l2
intros H.	1 subgoal T : Type U : Type f : T -> U l1 : list T l2 : list T H : l1 = l2 _____(1/1) map f l1 = map f l2
rewrite H.	1 subgoal T : Type U : Type f : T -> U l1 : list T l2 : list T H : l1 = l2 _____(1/1) map f l2 = map f l2
reflexivity.	Proof completed.

Remarque 1.8. *Rappelons que pour enregistrer la preuve, il faut la terminer par la commande* `Qed` *qui effectue en plus toutes les vérifications de correction. Dans les preuves que nous proposons dans cette introduction, nous n'explicitons jamais le* `Qed` *parce qu'il n'apporte pas d'information au lecteur, mais dans un script Coq il devrait toujours être présent.*

Dans certains cas, il peut être nécessaire de réintroduire certains éléments dans le contexte afin de garder un typage homogène, en particulier dans le cas des types dépendants.

Voyons un exemple. On définit le type des tableaux *tab* proposé dans la Section 1.3 :

```
Inductive tab (T: Set) : nat -> Set :=
  sing : T -> tab T 0
| comp : forall n, T -> tab T n -> tab T (S n).
```

On définit une fonction `fstn` dont le type est le suivant :

```
Definition fstn (T: Set)(n1 n2: nat)(t: tab T n1) :
  n2 <= n1 -> tab T n2.
```

Elle renvoie un tableau composé des premiers éléments du tableau jusqu'à la "case" n_2 (ainsi, si $n_2 = 0$, alors le tableau résultat est composé d'un élément). On ne détaille pas sa définition ici. On suppose également qu'on a démontré le lemme suivant (qui est trivial) :

```
Lemma eq_le (n1 n2 : nat) : n1 = n2 -> n2 <= n1.
```

On veut aussi convertir un élément de type `tab T n1` au type `tab T n2`, sous la condition que `n1 = n2`. Pour cela, il existe dans Coq une fonction spéciale de filtrage par motif (pour plus d'informations, voir [34, Chapitres 1.2.13 and 4.5.4]). On la définit ainsi pour `tab` :

```
Definition convTab (T: Set)(n1 n2: nat)(t: tab T n1)(h: n1 = n2):
  tab T n2 := match h in _=l return tab T l with eq_refl => t end.
```

Cela signifie qu'on identifie l'hypothèse `h` à un élément du type `_ = l` (on nomme la partie droite de l'égalité `l`). Et on renvoie un élément de type `tab T l` (ici `tab T n2`). On utilisera cette fonction de filtrage à plusieurs reprises dans la suite.

Enfin, on va supposer également qu'on a démontré le lemme suivant :

```
Lemma fstn_n (T: Set)(n: nat)(t: tab T n)(h: n <= n): fstn t h = t.
```

Remarque 1.9. *On ne détaille pas ici les démonstrations précédentes car elles dépassent le contexte de cette brève présentation. Néanmoins, la preuve du lemme* `fstn_n` *est détaillée à la Section 1.4.1.2.*

On veut démontrer que si `n1 = n2` alors le tableau qui contient les `n2` premiers éléments est le même que le tableau initial (à une conversion près). On l'exprime comme suit :

```
Lemma fstn_n1_n2 (T: Set)(n1 n2: nat)(t: tab T n1) (h: n1 = n2) :
  fstn t (eq_le h) = convTab t h.
```

Voyons sa démonstration. On veut pouvoir se débarasser du `convTab` puisqu'il ne fait que réécrire le type. Pour cela, on voudrait réécrire l'hypothèse `h` dans le type de `t`. Comme `t` est utilisé dans la conclusion, la modification de son type peut avoir des répercussions sur la conclusion. On doit donc le réintroduire dans le contexte, puis seulement faire la réécriture.

```
                  1 subgoal
                  T : Set
                  n1 : nat
                  n2 : nat
                  t : tab T n1
                  h : n1 = n2
                  ____________________________________(1/1)
                  fstn t (eq_le h) = convTab t h

 revert t.        1 subgoal
                  T : Set
                  n1 : nat
                  n2 : nat
                  h : n1 = n2
                  ____________________________________(1/1)
                  forall t : tab T n1, fstn t (eq_le h) = convTab t h
```

`rewrite h.`	`1 subgoal`
	`T : Set`
	`n1 : nat`
	`n2 : nat`
	`h : n1 = n2`
	`_____(1/1)`
	`forall t : tab T n2, fstn t (eq_le (eq_refl n2))`
	`  = convTab t (eq_refl n2)`

La tactique `rewrite h` a non seulement modifié le type de `t` mais aussi la conclusion, en remplaçant toutes les occurrences de `n1` par `n2` (ainsi, `h` devient `eq_refl n2`).

`simpl.`	`1 subgoal`
	`T : Set`
	`n1 : nat`
	`n2 : nat`
	`h : n1 = n2`
	`_____(1/1)`
	`forall t : tab T n2, fstn t (eq_le (eq_refl n2)) = t`

`intros t.`	`1 subgoal`
	`T : Set`
	`n1 : nat`
	`n2 : nat`
	`h : n1 = n2`
	`t : tab T n2`
	`_____(1/1)`
	`fstn t (eq_le (eq_refl n2)) = t`

`apply fstn_n.`	`Proof completed.`

On note également que même si deux types sont prouvablement égaux ($T = U$) ils restent distincts dans Coq. La construction de filtrage par motif présentée précédemment permet de convertir des éléments du type T vers le type U, mais un élément t de T n'est pas, pour Coq, de type U. De la même façon, pour un élément u de U, $t = u$ n'est pas correctement typé. Ici, Coq s'arrête à des considérations syntaxiques (deux éléments sont du même type si leurs types ont le "même nom", ils ne peuvent pas être simplement prouvablement égaux).

`induction` et `destruct` Comme expliqué à la Section 1.2.1, pour pouvoir raisonner par induction, on a besoin de principes d'induction sur les types inductifs. Dans Coq le principe d'induction "naturel" est généré automatiquement. Ainsi, pour `tab`, le principe généré est :

```
tab_ind : forall (T : Set) (P : forall n : nat, tab T n -> Prop),
 (forall t: T, P 0 (sing t)) -> (forall(n: nat)(t: T)(t': tab T n),
 P n t' -> P (S n) (comp t t'))-> forall(n: nat)(t: tab T n), P n t
```

Pour raisonner par induction on utilise la tactique `induction nom_de_variable`. Coq utilise alors le principe généré automatiquement. Par exemple, si `t` est de type `tab T n`, alors `induction t` utilise `tab_ind`. Comme on l'a dit, le principe d'induction naturel ne suffit pas toujours. Dans ce cas on peut en énoncer (et démontrer) un nouveau. Si ensuite on veut raisonner en utilisant celui-là, on écrit : `induction t using nouveau_principe`. On peut aussi donner explicitement un nom aux paramètres et hypothèses générés avec `induction t as [..|..]` où les `|` séparent les différents cas de l'induction.

Pour illustrer, prouvons le lemme `fstn_n` énoncé Section 1.4.1.2. Rappelons-en l'entête :

```
Lemma fstn_n (T: Set)(n: nat)(t: tab T n)(h: n <= n): fstn t h = t.
```

```
                    | 1 subgoal
                    | T : Set
                    | n : nat
                    | t : tab T n
                    | h : n <= n
                    | ____________________________________(1/1)
                    | fstn t h = t

 induction t as     | 2 subgoals
   [t |n t ta IH].  | T : Set
                    | t : T
                    | h : 0 <= 0
                    | ____________________________________(1/2)
                    | fstn (sing t) h = sing t
                    | ____________________________________(2/2)
                    | fstn (comp t ta) h = comp t ta
```

La tactique `induction` génère donc deux sous-objectifs à prouver. On prouve le premier :

```
 simpl.             | 2 subgoals
                    | T : Set
                    | t : T
                    | h : 0 <= 0
                    | ____________________________________(1/2)
                    | sing t = sing t
                    | ____________________________________(2/2)
                    | fstn (comp t ta) h = comp t ta

 reflexivity.       | 1 subgoal
                    | T : Set
                    | n : nat
                    | t : T
                    | ta : tab T n
                    | h : S n <= S n
                    | IH : forall h : n <= n, fstn ta h = ta
                    | ____________________________________(1/1)
                    | fstn (comp t ta) h = comp t ta
```

Le premier sous-objectif est donc prouvé. Démontrons maintenant le second :

```
 simpl.             | 1 subgoal
                    | T : Set
                    | n : nat
                    | t : T
                    | ta : tab T n
                    | h : S n <= S n
                    | IH : forall h : n <= n, fstn ta h = ta
                    | ____________________________________(1/1)
                    | comp t (fstn ta (le_S_n n n h)) = comp t ta
```

```
rewrite IH.
```

```
1 subgoal
T : Set
n : nat
t : T
ta : tab T n
h : S n <= S n
IH : forall h : n <= n, fstn ta h = ta
_____________________________________(1/1)
comp t ta = comp t ta
```

```
reflexivity.
```

```
Proof completed.
```

On ne donne pas ici d'exemple où il faut redéfinir le principe d'induction. On a déjà expliqué dans la théorie comment cela se passe et les choses ne sont pas plus compliquées dans Coq.

On peut aussi vouloir raisonner par analyse de cas (un peu comme une induction : on analyse les différents cas possibles de la définition, mais sans l'hypothèse d'induction). Dans ce cas, on utilise la tactique `destruct`. Ainsi, si `t` est un élément de type `tab T n`, `destruct t` sépare les différents cas possibles pour `t` selon sa définition et génère un sous-objectif par cas.

1.4.2 Preuves par l'absurde

Il arrive souvent qu'on veuille démontrer une négation, c'est-à-dire un résultat du type `not propriete`. Dans ce cas, la démonstration se fait la plupart du temps par l'absurde : on suppose la propriété vraie et on montre qu'on arrive à une contradiction. En réalité, dans Coq un résultat de type `not P` est un alias pour `P -> False`. `P` est donc vu comme un paramètre et on peut l'introduire dans le contexte. On cherche alors à montrer `False`. De la même façon, si on a dans le contexte un élément de type `H : not P'` et qu'on cherche à montrer `False`, alors on peut appliquer simplement `H`. L'objectif change et on doit alors montrer `P'`. Pour illustrer cela, montrons que si on a P alors on a $\neg\neg P$:

```
Lemma P_not_not_P (P: Prop) : P -> not (not P).
```

```
1 subgoal
P : Prop
_____________________________________(1/1)
P -> ~ ~ P
```

```
intros H1 H2.
```

```
1 subgoal
P : Prop
H1 : P
H2 : ~ P
_____________________________________(1/1)
False
```

```
apply H2.
```

```
1 subgoal
P : Prop
H1 : P
H2 : ~ P
_____________________________________(1/1)
P
```

```
apply H1.          | Proof completed.
```

Dans la suite, lorsqu'on utilisera ce type de raisonnement, on utilisera le terme de preuve par l'absurde.

1.4.3 Relations et morphismes paramétriques [78]

Dans Coq, on peut déclarer qu'une relation est une relation d'équivalence (avec `Add Parametric Relation`) ou qu'une fonction est un morphisme pour chacun de ses paramètres par rapport à des relations (avec `Add Parametric Morphism`). Il faut bien entendu démontrer que cela est vrai. Pour une relation d'équivalence, on doit démontrer qu'elle est réflexive, symétrique et transitive. Et pour montrer que $f : T \to U$ est un morphisme pour une relation R_T sur T et une relation R_U sur U, on doit montrer que $\forall t_1\ t_2, R_T\ t_1\ t_2 \Rightarrow R_U\ (f\ t_1)\ (f\ t_2)$.

Ces relations et fonctions sont alors connues du système comme étant des relations d'équivalence et des morphismes. Dans le cas des relations, cela permet d'utiliser les tactiques `reflexivity`, etc. comme s'il s'agissait de l'égalité de Leibniz. Pour les morphismes, cela permet de pouvoir faire de la réécriture, avec la tactique `rewrite`, comme s'il s'agissait également de l'égalité de Leibniz.

Nous nous sommes assez peu servis de ces fonctionnalités dans le reste aussi nous ne nous étendons pas plus à ce sujet. Pour plus d'informations, voir le chapitre du manuel de référence de Coq écrit par Sozeau [78].

1.4.4 La coinduction dans Coq

Une grande partie de notre travail repose sur les types coinductifs. Une des raisons pour lesquelles nous avons choisi Coq est justement leur support par le système. Ils ont été tout d'abord proposés par Coquand [26] puis implémentés par Giménez [43].

1.4.4.1 Présentation

Comme on l'a dit, dans Coq pour définir un type coinductif on utilise le mot-clé `CoInductive`. Pour le reste, les définitions sont exactement identiques à des définitions de types inductifs (d'un point de vue syntaxique), c'est-à-dire en particulier avec des constructeurs. Ainsi les listes toujours infinies (les *Stream* de la Section 1.2.2) se définissent ainsi :

```
CoInductive Stream (T: Type): Type :=
  Cons : T -> Stream T -> Stream T.
```

Pour définir une fonction corécursive, on utilise le mot-clé `CoFixpoint` (là où on utilisait `Fixpoint` pour les fonctions récursives). Par exemple, on peut définir l'équivalent du *map* sur les listes pour les *Stream* :

```
CoFixpoint map_st (T U: Type)(f: T -> U)(s: Stream T) : Stream U :=
  match s with
    Cons t q => Cons (f t) (map_st f q)
  end.
```

On peut également bien entendu définir des fonctions non corécursives sur des types coinductifs, comme par exemple les notions de tête et de queue d'une liste infinie :

```
Definition hd (T: Type)(s: Stream T) : T :=
  match s with Cons t _ => t end.

Definition tl (T: Type)(s: Stream T) : Stream T :=
  match s with Cons _ s => s end.
```

Comme premier exemple, on peut montrer que les notions de `hd` et `tl` sont cohérentes :

```
Lemma hd_tl (T: Type)(s: Stream T) : Cons (hd s) (tl s) = s.
```

	1 subgoal T : Type s : Stream T _____(1/1) Cons (hd s) (tl s) = s
destruct s.	1 subgoal T : Type t : T s : Stream T _____(1/1) Cons (hd (Cons t s)) (tl (Cons t s)) = Cons t s
simpl.	1 subgoal T : Type t : T s : Stream T _____(1/1) Cons t s = Cons t s
reflexivity.	Proof completed.

Pour définir des relations de bisimulation, on raisonne comme pour la définition d'objets coinductifs mais dans l'univers des propositions. Par exemple, deux *Stream* sont équivalentes par la bisimulation canonique si leurs têtes sont égales et leurs queues équivalentes :

```
CoInductive EqSt(T: Type): Stream T -> Stream T -> Prop :=
  eqst: forall s1 s2, hd s1 = hd s2 -> EqSt (tl s1) (tl s2) ->
  EqSt s1 s2.
```

Pour effectuer un raisonnement par coinduction, on utilise la tactique `cofix` qui "copie" dans le contexte l'objectif actuel (il faut donc faire attention à avoir le bon niveau d'abstraction). Bien entendu, pour que cela soit logiquement justifié, on ne peut entre autres l'utiliser qu'avec un sous-terme du terme courant (c'est-à-dire qu'on doit procéder par "destruction" du terme courant jusqu'à retrouver un élément de la forme précédente), l'ensemble des règles d'utilisation sont regroupées sous le nom de condition de garde dont nous parlerons plus en détail à la Section 1.4.4.2. Pour illustrer cela, montrons par exemple que l'application de `map_st` avec la fonction identité donne bien une *Stream* équivalente à l'originale :

```
Lemma map_st_id(T: Type)(s: Stream T):
  EqSt s (map_st (fun x : T => x) s).
```

```
1 subgoal
T : Type
s : Stream T
_____________________________________(1/1)
EqSt s (map_st (fun x : T => x) s)
```

Si on applique ici la tactique `cofix`, on obtient `EqSt s (map_st (fun x : T => x) s)` comme hypothèse de coinduction. On ne peut pas l'utiliser directement puisqu'il faut l'appeler avec un sous-terme de `s`, ce qui est impossible ici. L'hypothèse de coinduction manque d'abstraction. On doit réinjecter `s` dans le contexte avant d'appliquer la tactique `cofix`.

```
revert s.
```

```
1 subgoal
T : Type
_____________________________________(1/1)
forall s : Stream T,
  EqSt s (map_st (fun x : T => x) s)
```

```
cofix.
```

```
1 subgoal
T : Type
map_st_id : forall s : Stream T,
  EqSt s (map_st (fun x : T => x) s)
_____________________________________(1/1)
forall s: Stream T,
  EqSt s (map_st (fun x : T => x) s)
```

```
intros [t s].
```

```
1 subgoal
T : Type
map_st_id : forall s : Stream T,
  EqSt s (map_st (fun x : T => x) s)
t : T
s : Stream T
_____________________________________(1/1)
EqSt (Cons t s) (map_st (fun x : T => x) (Cons t s))
```

Remarque 1.10. *Notons l'utilisation d'une autre forme de la tactique* `intros` *qui remplace deux tactiques :* `intros s` *et* `destruct s as [t s].`

```
apply eqst.
```

```
2 subgoals
T : Type
map_st_id : forall s : Stream T,
  EqSt s (map_st (fun x : T => x) s)
t : T
s : Stream T
_____________________________________(1/2)
hd (Cons t s) =
hd (map_st (fun x : T => x) (Cons t s))
_____________________________________(2/2)
EqSt (tl (Cons t s))
     (tl (map_st (fun x : T => x) (Cons t s)))
```

`simpl.`	`2 subgoals`
	`T : Type`
	`map_st_id : forall s : Stream T,`
	`  EqSt s (map_st (fun x : T => x) s)`
	`t : T`
	`s : Stream T`
	`_____(1/2)`
	`t = t`
	`_____(2/2)`
	`EqSt (tl (Cons t s))`
	`  (tl (map_st (fun x : T => x) (Cons t s)))`

`reflexivity.`	`1 subgoal`
	`T : Type`
	`map_st_id : forall s : Stream T,`
	`  EqSt s (map_st (fun x : T => x) s)`
	`t : T`
	`s : Stream T`
	`_____(1/1)`
	`EqSt (tl (Cons t s))`
	`  (tl (map_st (fun x : T => x) (Cons t s)))`

`simpl.`	`1 subgoal`
	`T : Type`
	`map_st_id : forall s : Stream T,`
	`  EqSt s (map_st (fun x : T => x) s)`
	`t : T`
	`s : Stream T`
	`_____(1/1)`
	`EqSt s (map_st (fun x : T => x) s)`

`apply map_st_id.`	`Proof completed.`

Remarque 1.11. *Contrairement au lemme* `hd_tl` *on ne peut pas exprimer l'équivalence entre* `s` *et* `map_st (fun x : T => x) s` *avec l'égalité de Leibniz. En effet, dans* `hd` *et* `tl` *les éléments ne sont pas "touchés", ils sont tout simplement copiés. Alors que dans* `map_st` *ils sont modifiés (même si la fonction paramètre est l'identité). On ne se contente pas de copier les éléments.*

1.4.4.2 Condition de garde

Quand on travaille avec des types infinis, tout n'est pas aussi facile que présenté précédemment. Certaines opérations peuvent se révéler dangereuses. Prenons l'exemple du filtre. Sur les listes, c'est une opération classique. Elle prend en argument un prédicat et une liste et renvoie le liste des éléments de la liste paramètre pour lesquels le prédicat est vrai :

```
Fixpoint filter (T: Type)(f: T -> bool)(l: list T) : list T :=
  match l with
    nil => nil
  | t :: q => if (f t) then t :: filter f q else filter f q
  end.
```

Imaginons maintenant qu'on définisse cette même opération sur les listes infinies et qu'on veuille filtrer seulement les éléments pairs. Si on donne en entrée à ce filtre une liste infinie contenant seulement des éléments impairs, quel est le résultat ? Peut-on le représenter ? La réponse est non. Le résultat du filtre devrait être de type `Stream nat`, pourtant ici ce résultat ne contient aucun élément. Et on ne peut pas représenter de liste infinie vide...

Il est donc nécessaire d'avoir un "garde-fou" pour éviter ces cas là, qui sont sémantiquement incorrects : en Coq, il s'agit de la condition de garde. Le plus gros problème est bien sûr de passer d'une règle théorique à une règle implémentable en pratique. L'idée sous-jacente derrière cette notion de garde est la notion de productivité [35, 77]. Pour pouvoir s'assurer qu'une définition coinductive est correcte, on doit être sûr de pouvoir obtenir le prochain élément en un temps fini (ce qui n'est clairement pas le cas pour l'exemple du filtre des entiers pairs sur une *Stream* d'entiers impairs, puisqu'il n'y a pas de prochain élément). Cette règle reste cependant théorique. Son implémentation dans Coq est beaucoup plus syntaxique : pour qu'une définition coinductive ou corécursive soit "gardée", il faut que l'appel corécursif soit directement paramètre d'un constructeur (de type coinductif ou inductif)[42]. Ainsi, la définition de `map_st` est correcte puisque l'appel corécursif (à `map_st` est un argument du constructeur `Cons`). En revanche, si on voulait définir le filtre sur les *Stream*, on le définirait ainsi :

```
CoFixpoint filter_st (T: Type)(f: T -> bool)(l: Stream T): Stream T:=
  match l with
  | Cons t q => if (f t) then Cons t (filter_st f q)
                else filter_st f q
  end.
```

Dans un des cas ici, l'appel corécursif n'est pas argument d'un constructeur. Coq refuse cette définition avec le message `Unguarded recursive call in "filter_st T f q"`. C'est effectivement ce qu'on attendait. Cependant, le fait d'avoir traduit une notion sémantique en règle syntaxique induit une forte restriction des définitions possibles. Ainsi, prenons l'exemple, classique en Haskell, de la liste infinie de tous les entiers naturels. On peut la définir ainsi :

$$nats := 1 :: (map\ (\lambda x.x + 1)\ nats)$$

En Coq cela se traduirait par :

```
CoFixpoint nats := Cons 1 (map_st S nats).
```

Remarque 1.12. *En Coq, les entiers naturels sont définis inductivement par deux constructeurs, zéro et successeur :* `Inductive nat : Set := O : nat | S : nat -> nat.` *Ainsi, l'entier* 3 *est représenté par* `S (S (S O))` *dans Coq.*

Mais cette définition n'est pas acceptée par Coq parce que l'appel corécursif (à `nats`) ne se trouve pas **directement** sous un constructeur (il y a la fonction `map_st` entre le constructeur et l'appel corécursif). Pourtant, cette définition est productive (car `map_st` est productif) et il n'y a pas de raison sémantique de l'interdire (voir en particulier [16]). Il existe de nombreux autres exemples de fonctions sémantiquement correctes mais non gardées [13, 40, 37, 65].

On voit donc que la condition de garde peut s'avérer trop restrictive. Dans notre travail, nous nous sommes de nombreuses fois confrontés à ce problème. Cependant, elle est un compromis entre expressivité et maniabilité, résultat d'une longue évolution et il serait dangereux de vouloir la modifier. Nous travaillerons donc à la contourner.

Pour plus de détails à propos de la condition de garde dans Coq, voir [26, 16, 44].

Cette condition de garde a des répercussions sur le mélange entre types inductifs et types coinductifs, aussi bien au niveau de la définition qu'au niveau du raisonnement (on ne peut pas, par exemple, faire une preuve coinductive qui fait appel à une preuve inductive dans la partie de la preuve qui contient l'appel corécursif). Nous verrons dans la suite de nombreux exemples pour étayer ce point.

2 Etat de l'art

LE contexte introduit, nous pouvons maintenant rentrer dans le vif du sujet de cette thèse, qui est la représentation des graphes. Nous allons donc dans un premier temps nous pencher sur les représentations existantes et les comparer. Puis nous proposerons une première idée qui nous amènera dans un domaine plus technique : le contournement de la condition de garde, particulièrement dans Coq.

2.1 Représentation des graphes

L'idée de base que nous avions pour la représentation des graphes était de dualiser la représentation classique des arbres aux graphes. En effet, la représentation classique des arbres nous paraissait particulièrement pratique à manipuler et adaptée aux preuves et aux raisonnements. Nous allons donc dans un premier temps étudier les représentations classiques des arbres n-aires. Puis nous verrons quelles sont les représentations usuelles des graphes. Finalement, nous tenterons de dualiser la représentation des arbres de notre choix aux graphes.

2.1.1 Représentation des arbres n-aires

Dans un premier temps, nous allons voir la représentation "théorique" proposée par Courcelle puis nous ferons le lien avec la représentation plus pratique, classique des langages fonctionnels.

2.1.1.1 Représentation d'après Courcelle [28]

Dans [28], Courcelle présente une solution pour représenter les arbres. En simplifiant un peu, il dit qu'un arbre peut être vu comme une fonction partielle de $\mathbb{N}_+^*$ dans un alphabet F ($t : \mathbb{N}_+^* \to F$), avec $\mathbb{N}_+^*$ qui indique le chemin depuis la racine jusqu'au nœud (par exemple $n_1 n_2 \ldots$ indique qu'on passe par le fils n°n_1, puis par le fils n°n_2 de celui-ci, etc.). Pour illustrer ce principe, voyons comment serait défini l'arbre de la Figure 2.1 avec cette méthode. On aurait $F = 1, 2, 3, 4$ et si on appelle a l'arbre à représenter, il est défini par ($a(\epsilon)$ correspond à la racine) :

$$\begin{array}{ll} a(\epsilon) = 1 & a(1) = 3 \\ & a(2) = 2 \quad a(21) = 4 \end{array}$$

Courcelle propose ensuite de simplifier son écriture en se passant des chemins et en indiquant sous forme de liste, paramètre de chacun des nœuds, l'ensemble des nœuds qui sont accessibles depuis celui-ci. Pour l'exemple de la Figure 2.1, cela donnerait (l'alphabet ne change pas) : $1(3, 2(4))$. On peut même simplifier encore et n'écrire que $1(3, 24)$. Ces dernières notations ressemblent beaucoup à une notation en nœud (qui a un label)/liste de fils. C'est la représentation que nous allons voir maintenant.

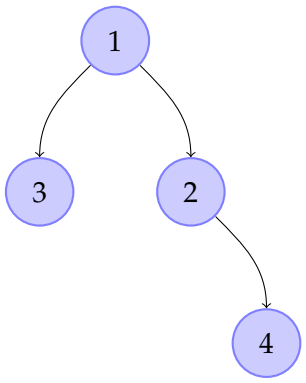


Figure 2.1 — Exemple d'arbre

2.1.1.2 Représentation fonctionnelle classique

Dans les langages fonctionnels, les arbres n-aires sont généralement représentés de façon inductive sous forme de nœud et de liste de fils (ce qui correspond tout à fait à la vision de Courcelle, comme nous venons de le voir). Ils sont un exemple bien connus de type de données pour la communauté autour du langage Haskell. Celle-ci les nomme rose trees (voir par exemple [18]). Typiquement, ils pourraient être modélisés par la définition suivante :

Définition 2.1 (*Tree*)**.**
$$\frac{t : T \qquad l : list\ (Tree\ T)}{mk\_Tree\ t\ l : Tree\ T}$$

Exemple 2.1. *Avec cette définition, on définirait l'arbre de la Figure 2.1 comme ceci (avec* $T = \mathbb{N}$*) :*

$$mk\_Tree\ 1\ [mk\_Tree\ 3\ [];mk\_Tree\ 2\ [mk\_Tree\ 4\ []]]$$

Remarque 2.1. *On voit qu'avec la définition proposée les feuilles sont représentées par un nœud qui a une étiquette et pas de fils (liste vide). Cependant, on ne peut pas représenter un arbre complètement vide (c'est-à-dire qui n'a aucun nœud). Avec cette représentation un arbre a toujours au moins une feuille. Si on voulait pouvoir représenter un arbre vide, on pourrait rajouter le constructeur suivant :*

$$\frac{}{mk\_Tree\_Vide : Tree\ T}$$

Cependant comme on l'a vu à la Section 1.2.1, ici on introduirait une forme d'ambiguïté dans la représentation. En effet, un même arbre pourrait avoir plusieurs représentations. Par exemple, si on prend l'arbre très simple de la Figure 2.2 issu de l'Exemple 2.1, on peut le représenter de plusieurs façons. On pourrait tout aussi bien écrire simplement : $mk\_Tree\ 1\ [mk\_Tree\ 2\ []]$ *que* $mk\_Tree\ 1\ [mk\_Tree\ 2\ [mk\_Tree\_Vide]]$ *ou* $mk\_Tree\ 1\ [mk\_Tree\ 2\ [mk\_Tree\_Vide];mk\_Tree\_Vide]$*, etc. Toutes ces définitions seraient correctes. Nous considèrerons donc ici seulement des arbres non*

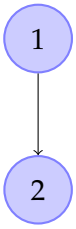


Figure 2.2 — Exemple simplifié d'arbre

vides (et c'est le standard dans la représentation des arbres n-aires).

Le gros avantage de cette définition, c'est qu'elle permet de naviguer très simplement dans l'arbre. Depuis un nœud, on a accès directement à ses fils. De plus, comme elle est inductive, c'est une structure sur laquelle on peut raisonner facilement (grâce en particulier à l'induction structurelle). Ce sont ces fonctionnalités qu'on voudrait pouvoir retrouver pour les graphes.

Etudions donc maintenant les différentes manières de représenter les graphes.

2.1.2 Représentation des graphes

Nous allons présenter trois méthodes différentes. La première est la méthode la plus classique pour représenter les graphes : on les présente sous forme d'ensemble de nœuds/ensemble d'arêtes. La deuxième est une méthode inductive proposée par Erwig dans [38] et permet de représenter des graphes dirigés. La troisième enfin, est proche de celle que nous avons présentée sur les arbres. Pour chaque méthode, nous dirons quels types de graphes elle permet de représenter et donnerons ses avantages et ses inconvénients.

Pour comparer et illustrer ces différentes méthodes, nous utiliserons les exemples des Figures 2.3 et 2.4.

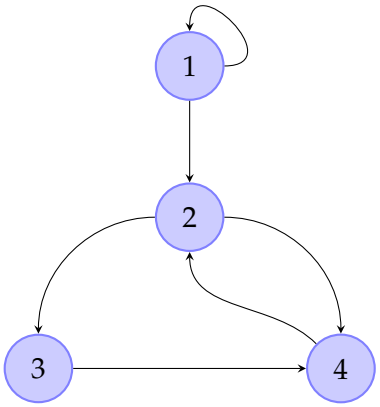


Figure 2.3 — Exemple de graphe enraciné et connexe

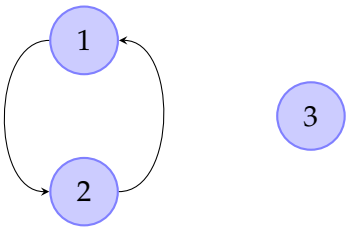


Figure 2.4 — Exemple de graphe non connexe

2.1.2.1 Représentation classique

La méthode qui prévaut en général pour représenter les graphes de façon inductive est la vision en ensemble de nœuds/ensemble d'arêtes. Les graphes sont représentés sous forme

de couple $G = (V, E)$, avec V l'ensemble des nœuds et E un sous-ensemble de $V \times V$, c'est-à-dire un ensemble de couples de nœuds.

Remarque 2.2. *Cette méthode est la plus largement employée pour représenter par exemple les automates à états finis, en y ajoutant en plus un symbole de transition.*

Exemple 2.2 (Représentation de l'exemple de la Figure 2.3)**.** *Avec cette méthode, on représenterait l'exemple de la Figure 2.3 de la façon suivante :*

$$V = \{1,2,3,4\} \quad E = \{(1,1),(1,2),(2,3),(2,4),(3,4),(4,2)\}$$

Exemple 2.3 (Représentation de l'exemple de la Figure 2.4)**.** *Nous pouvons aussi sans problème représenter l'exemple de la Figure 2.4 :*

$$V = \{1,2,3\} \quad E = \{(1,2),(2,1)\}$$

Il n'y a aucune arête qui arrive sur 3 *ou qui en part.*

Le gros avantage de cette représentation c'est que tous les nœuds sont au même niveau : il n'y a pas de racine. Il n'y a donc pas d'ordre : ni de hiérarchie verticale de type père-fils, ni de hiérarchie horizontale de type frère. En effet, comme un graphe peut être circulaire, cette hiérarchie n'a pas vraiment de sens (dans la Figure 2.3, 2 est-il le père de 4 ? Ou, est-ce l'inverse ?). De plus, cela permet également de représenter des graphes non connexes, comme nous l'avons vu dans l'Exemple 2.3.

Cette définition est donc très adaptée à la représentation de graphes finis quelconques. En revanche, elle est beaucoup moins efficace en ce qui concerne leur manipulation. En effet, pour connaître les nœuds accessibles depuis un nœud n (les "fils" d'un nœud), on doit parcourir l'intégralité de l'ensemble E pour trouver tous les couples dont le premier élément est n. Cela peut s'avérer fastidieux et extrêmement long pour des gros graphes et même impossible pour un graphe ayant une infinité d'arêtes.

Enfin, cette représentation est loin d'être intuitive. Lorsqu'on voit ces deux ensembles, on ne visualise pas immédiatement le graphe qu'ils représentent.

En conclusion, cette méthode est bonne mais non intuitive pour la représentation de graphes finis, et difficile à utiliser lorsqu'on veut raisonner sur ces graphes. En particulier, on ne retrouve pas du tout la navigabilité dans le graphe telle qu'on peut la connaître pour les arbres.

2.1.2.2 Représentation inductive d'Erwig [38]

Dans [38], Erwig propose une méthode inductive et incrémentale pour représenter les graphes orientés. Il propose d'ajouter les nœuds un à un avec la liste de leurs successeurs et de leurs prédécesseurs parmi les nœuds déjà ajoutés dans le graphe. Dans sa représentation, il identifie les nœuds par un identifiant unique d'un type *Node*. De plus, les nœuds et les arêtes peuvent être labellisés.

Il définit donc d'abord les types *Adj* (pour représenter un nœud adjacent à un autre) et *Context* (pour représenter une déclaration de nœud) de la façon suivante :

$$Adj\ U := list\ (U \times Node) \quad Context\ T\ U := Adj\ U \times Node \times T \times Adj\ U$$

Dans *Adj*, l'élément de type U représente le label de l'arête et l'élément de type *Node* l'identifiant du nœud adjacent. Dans *Context*, le premier *Adj U* représente la liste des prédecesseurs

du nœud (c'est-à-dire les nœuds dont une arête sortante pointe sur le nœud considéré) et le second la liste de ses successeurs (c'est-à-dire les nœuds qui sont pointés par une arête sortante du nœud considéré) ; l'élément de type *Node* représente l'identifiant du nœud ajouté et l'élément de type T son label. Il définit alors les graphes à l'aide d'un type de données algébrique à deux constructeurs :

$$\frac{}{Empty : Graph\ T\ U} \qquad \frac{c : Context\ T\ U \qquad g : Graph\ T\ U}{c\&g : Graph\ T\ U}$$

Exemple 2.4 (Représentation de l'exemple de la Figure 2.3)**.** *Avec cette méthode, on peut représenter l'exemple de la Figure 2.3 de la façon suivante (on prendra ici Node* $= \mathbb{N}$*,* $T = \mathbb{N}$*,* $U = \varnothing$ *– on notera* $()$ *un label vide) :*

$$\begin{array}{llllll} ([((),2);((),3)], & 4, & 4, & [((),2)]) & \& & \\ ([((),2)], & 3, & 3, & []) & \& & \\ ([((),1)], & 2, & 2, & []) & \& & \\ ([], & 1, & 1, & [((),1)]) & \& & Empty \end{array}$$

On pourrait bien entendu ajouter les nœuds dans un tout autre ordre sans que cela ne pose de problème. En revanche, une question reste ouverte. Que doit-on faire pour l'arête $1 \to 1$*. Doit on l'ajouter dans les successeurs de* 1 *(comme ce que nous avons fait ici), dans ses prédecesseurs ou dans les deux ?*

Exemple 2.5 (Représentation de l'exemple de la Figure 2.4)**.** *Nous pouvons aussi sans problème représenter l'exemple de la Figure 2.4 :*

$$\begin{array}{llllll} ([], & 3, & 3, & []) & \& & \\ ([((),1)], & 2, & 2, & [((),1)]) & \& & \\ ([], & 1, & 1, & []) & \& & Empty \end{array}$$

Remarque 2.3. *On peut très facilement représenter des graphes non orientés en ajoutant, pour chaque arête, le nœud cible comme prédécesseur et successeur du nœud courant.*

Cette solution a le très gros avantage d'être inductive et donc de permettre tous les raisonnements associés (filtrage par motif, récursion structurelle...). Erwig le montre bien dans [38]. De plus, il n'y a pas vraiment de hiérarchie entre les nœuds (on peut les voir comme des éléments d'une liste, comme dans la représentation précédente). Cependant, comme on l'a fait remarquer dans l'Exemple 2.4, il n'est pas évident de savoir comment représenter un nœud qui a une arête qui revient sur lui même.

Néanmoins, encore une fois cette représentation n'est pas du tout intuitive et surtout assez difficile à manipuler. Ici aussi, pour trouver les successeurs d'un nœud, on doit parcourir l'ensemble de tous les nœuds (ajoutés après lui) pour trouver ceux avec lesquels il a un rapport. La navigabilité dans le graphe est donc assez fastidieuse. En revanche, on peut noter qu'il revient au même de naviguer dans un sens ou dans l'autre (successeur/prédecesseur).

2.1.2.3 Représentation d'après Courcelle [28]

Dans [28], après avoir présenté les arbres, Courcelle propose d'étendre cette représentation (voir Section 2.1.1.1) aux arbres infinis (c'est-à-dire des graphes enracinés et connexes). Le problème est un peu plus compliqué que précédemment : on doit "déplier" l'arbre et donner les chemins sous forme d'expression régulière (si l'arbre est réellement infini (c'est-à-dire qu'il a des cycles par exemple), on ne peut évidemment pas énumérer tous les chemins possibles). Illustrons cela sur l'exemple de la Figure 2.3.

Exemple 2.6 (Représentation de l'exemple de la Figure 2.3)**.** *On prend* $F = 1, 2, 3, 4$ *et on appelle* g *le graphe à représenter. Pour illustrer le propos, nous allons définir les expressions pour les nœuds* 1, 2 *et* 3*. Nous ne donnons pas le nœud* 4 *parce qu'il serait très compliqué et n'apporterait rien à la compréhension.*

$$g(\epsilon) = g(2^*) = 1$$
$$g(2^*1(111 + 21)^*) = 2$$
$$g(2^*1(21)^*1(1(12)^*1)^*) = 3$$

Exemple 2.7 (Représentation de l'exemple de la Figure 2.4)**.** *Le graphe concerné n'étant ni enraciné, ni connexe, on ne peut pas le représenter (on peut représenter séparément chacune de ses composantes connexes, mais on ne peut pas les inclure dans une même définition).*

En dehors des restrictions imposées sur les arbres (enracinement et connexité), cette représentation impose une hiérarchie entre les nœuds. Comme on a pu voir, elle est également assez difficile à mettre en œuvre pour des graphes fortement cycliques.

Cependant, elle est inductive et permet la récursion structurelle. De plus, la navigabilité dans le graphe est assez aisée. Elle ressemble surtout beaucoup à la représentation que nous avons présentée sur les arbres (comme nous l'avons montré Section 2.1.1, la représentation de Courcelle et la représentation habituelle sont très proches) et vers laquelle nous voulons tendre.

2.1.3 Un premier essai

Dans les arbres, la navigabilité *finie* est représentée par un type inductif. Dans les graphes, cette navigabilité doit être *infinie* : en effet, même si le graphe est fini, c'est-à-dire a un nombre de nœuds (différents) finis, cette navigabilité infinie est nécessaire pour les cycles. Nous allons donc la représenter par un type coinductif.

Notre première idée était de les représenter exactement comme des arbres en remplaçant simplement le type inductif par un type coinductif :

$$\frac{t : T \qquad l : list\ (Graph\ T)}{mk\_Graph\ t\ l : Graph\ T}$$

A titre d'exemple, nous instancions cette définition pour quelques graphes simples :

Figure 2.5 — Exemple d'un graphe qui ne contient qu'une feuille

Exemple 2.8 (Exemple qui n'utilise pas la corécursion : juste une feuille)**.** *On représente le graphe de la Figure 2.5 de la façon suivante :*

$$Leaf := mk\_Graph\ 0\ []$$

Exemple 2.9 (Exemple d'un graphe fini)**.** *On représente le graphe de la Figure 2.6 de la façon suivante :*

$$Finite\_Graph := mk\_Graph\ 0\ [mk\_Graph\ 1\ [Finite\_Graph]]$$

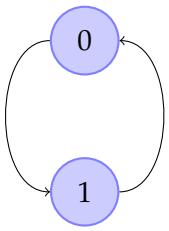


Figure 2.6 — Exemple d'un graphe fini

Remarque 2.4. *Ce graphe est fini mais il se déplie en un arbre infini (régulier) et permet donc une navigation infinie.*

Exemple 2.10 (Exemple d'un graphe infini)**.** *On représente la famille des graphes de la Figure 2.7 de la façon suivante :*

$$Infinite\_Graph_n := mk\_Graph\ n\ [Infinite\_Graph_{n+1}]$$

Le graphe de la Figure 2.7 correspond à $Infinite\_Graph_0$.

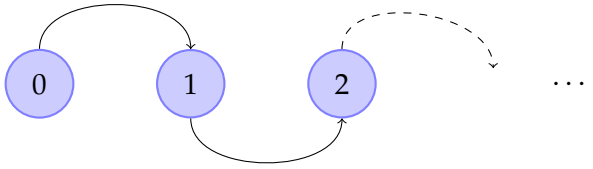


Figure 2.7 — Exemple d'un graphe infini

Cependant, ici, nous mélangeons induction (avec *list*) et coinduction. Comme nous l'avons expliqué Section 1.4.4.2, nous nous attendons à avoir des problèmes. Et en effet ils arrivent lorsqu'on essaye, par exemple, de définir un foncteur sur *Graph* qui applique une fonction à tous les nœuds du *Graph*. On voudrait la définir corécursivement ainsi :

$$\begin{aligned} &applyF2G : \forall T\ U,\ (T \to U) \to Graph\ T \to Graph\ U \\ &applyF2G\ f\ (mk\_Graph\ t\ l) := mk\_Graph\ (f\ t)\ (map\ (applyF2G\ f)\ l) \end{aligned}$$

Mais cela n'est pas accepté par Coq, car la condition de garde n'est pas respectée. En effet, l'appel corécursif à *applyF2G* est un argument de *map* alors que, comme nous l'avons expliqué, il devrait être directement l'argument d'un constructeur.

Nous devons donc trouver une solution pour contourner la condition de garde et pouvoir mélanger efficacement induction et coinduction.

2.2 Lutte contre la condition de garde

Comme nous l'avons dit, nous préférons contourner la condition de garde que tenter de la modifier, car il s'agirait alors de changer en profondeur le système Coq, ce qui est un sujet assez sensible. Nous ne sommes bien sûr pas les seuls à avoir dû affronter des problèmes de garde. Dans la théorie des types, le sujet a été étudié à de nombreuses reprises. Par exemple, dans [40, 41] Di Gianantonio et Miculan présentent une approche basée sur des constructions sémantiques dans le système Coq (mais avec un plongement profond de

leur développement, qui empêche l'animation de leurs structures par le noyau de Coq – ce plongement n'entre pas dans la convertibilité qui intervient automatiquement dans la vérification des types). Dans [2] et [10], la solution proposée est un changement au niveau du système de typage. Mais nous allons nous intéresser ici surtout à la façon dont il est exploré dans les prouveurs basés sur la théorie des types (ici, Coq et Agda). Nous allons donc voir ici quelles méthodes ont été employées. Nous verrons tout d'abord une proposition qui permet de résoudre les problèmes de définitions de fonctions sémantiquement correctes sur les *Stream*. Cela ne rentre pas directement dans le cadre de notre problème concret mais peut nous inspirer pour la suite. Nous verrons ensuite une proposition qui utilise la théorie des catégories, puis une autre qui propose une solution pour mélanger induction et coinduction. Concernant l'univers des propositions, nous regarderons du côté de l'imprédicativité et du style de Mendler [60, 59]. Finalement, nous verrons comment le mélange entre induction et coinduction est réalisé dans le prouveur Agda.

2.2.1 La solution de Bertot et Komendantskaya pour *Stream* [16]

Dans cet article, Bertot et Komendantskaya constatent qu'un certain nombre de fonctions sur les *Stream*, pourtant sémantiquement correctes, ne peuvent pas être définies à cause de la condition de garde. Ils prennent en particulier l'exemple de *nats* présenté à la Section 1.4.4.2

Dans ce travail, Bertot et Komendantskaya proposent une méthode générale pour réussir à définir quand même ces notions sémantiquement correctes mais non gardées. Leur idée est d'utiliser les fonctions. Ils montrent qu'il y a une bijection entre *Stream T* et les fonctions de type $\mathbb{N} \to T$, comme présenté sur la Figure 2.8.

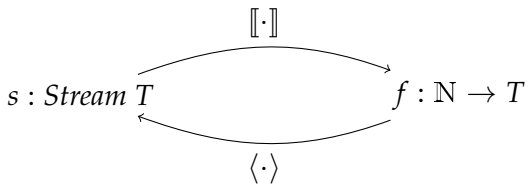


Figure 2.8 **—** Bijection entre les listes infinies et les fonctions

Ils énoncent alors leurs définitions non-gardées dans l'univers des fonctions et les transforment ensuite en *Stream* en contournant ainsi la condition de garde. Ils proposent également une méthode générale pour extraire la fonction équivalente.

Avec leur méthode, ils définissent la fonction f suivante :

$$\begin{array}{lcl} f & : & \mathbb{N} \to \mathbb{N} \\ f\ 0 & := & 1 \\ f\ (n+1) & := & (f\ n) + 1 \end{array}$$

et ils peuvent définir $nats := \langle f \rangle$. Bien sûr, cela ne pose plus de problème de garde puisque f est une fonction. Pour valider cette définition, ils montrent qu'elle respecte bien l'équation légitime de départ : $EqSt\ nats\ (1 :: (map\ (\lambda x.x+1)\ nats))$. On peut écrire cette proposition sans avoir à se soucier de problème de garde : en effet, le problème se pose à la définition, pas à l'énoncé de proposition.

Il est clair que le problème attaqué ici n'est pas le même que celui que nous avons exposé avec *Graph*. Cependant, l'idée de représenter des listes infinies par des fonctions pour contourner la condition de garde nous semble intéressante et efficace.

2.2.2 La solution de Niqui avec les catégories [66]

Dans cet article, Niqui s'attaque au même problème que celui résolu par Bertot et Komendantskaya, mais en le considérant sous un angle catégorique. L'idée est de formaliser directement dans Coq des schémas catégoriques de la théorie des coalgèbres [49], afin de contourner la nature trop syntaxique de la condition de garde. En particulier, il choisit d'implanter le schéma de λ-coitération de Bartels [9].

Il donne des outils généraux pour manipuler les coalgèbres, implantés dans Coq, puis les instancie dans le cadre des *Stream*, ce qui permet également de donner une validation de son travail.

A partir de foncteurs extensionnels (c'est-à-dire qui ne demandent que l'égalité extensionnelle à leurs fonctions paramètres), il définit des F-coalgèbres. Une F-coalgèbre est un couple composé d'un ensemble A et d'une structure de transition de A vers FA. Il définit ensuite la notion de relation de bisimulation sur ces coalgèbres, et en particulier de bisimulation maximale [45]. Il introduit ensuite la notion de coalgèbres faiblement finales. Quand une algèbre est faiblement finale, la relation de bisimulation maximale est la relation de bisimilarité [46]. Il finit enfin sa définition générale des coalgèbres dans Coq en implantant le schéma de λ-coitération. En particulier, il définit la fonction *coit* qui permet la commutativité modulo la bisimilarité.

Il instancie alors son cadre général dans le cas des *Stream*. Il définit donc les *Stream* comme étant des F-coalgèbres. La structure de transition est composée de deux transitions. Ces transitions correspondent aux deux destructeurs sur les *Stream* : la notion de tête et de queue d'une *Stream*. On voit donc qu'ici on a abandonné la représentation des *Stream* telle que proposée avec la coinduction de Coq, sous forme d'un constructeur, et qu'on est revenu à la notion originale à deux destructeurs. La relation de bisimulation maximale en revanche reste *EqSt* (mais redéfinie pour ce nouveau type, bien sûr).

Enfin, à titre d'exemple, il l'instancie entre autres pour *nats*, que nous avons présenté à la Section 2.2.1.

Contrairement à Bertot et Komendantskaya qui ne s'occupaient que des *Stream*, Niqui propose donc une solution générale pour résoudre les problèmes de garde liés à des définitions sur les types coinductifs, et l'instancie ensuite pour l'exemple des *Stream*. Cependant, cette solution nous a semblé compliquée à mettre en œuvre (contrairement à la solution précédente, qui nous paraît plus simple) et nous n'avons pas réussi à l'instancier pour autre chose que des *Stream*. Nous n'avons en particulier pas réussi à transposer cette théorie à notre problème concret.

2.2.3 La solution de Dams pour le mélange entre induction et coinduction [29]

Dans une discussion sur le *Coq-club*, Dams propose une solution pour mélanger induction et coinduction. Le problème posé est presque exactement le même que le nôtre (un type coinductif A qui a pour paramètre une liste d'éléments de A, et une définition corécursive incluant *map* refusée).

Dams, pour résoudre le problème, propose de travailler complètement coinductivement, avec des types définis mutuellement. C'est-à-dire qu'au lieu de travailler sur des listes finies il propose de travailler sur des listes potentiellement infinies (mais définies spécifiquement pour ce cas-ci, mutuellement avec le type de départ). Puis il restreint l'ensemble des possibilités admissibles aux seules listes finies mais en ajoutant une propriété additionnelle

inductive. On reste donc totalement dans le monde infini et on ne se restreint au monde fini que par des propriétés ad-hoc.

Cette solution fonctionne pour notre problème précis. En particulier, elle nous permet de définir un équivalent à la fonction *applyF2G*. On definit cette fonction complètement coinductivement (et simultanément sur le type des graphes et sur les listes infinies définies pour l'occasion) puis on la restreint en ajoutant une propriété.

Cependant, même si elle fonctionne, cette proposition est assez lourde à manipuler. En effet, on est obligé de toujours fournir la preuve de finitude pour les listes, ce qui rend donc les développements assez fastidieux.

En fait, l'idée sous-jacente est de dire qu'en définissant simultanément les deux types, on peut imbriquer moins profondément les types l'un dans l'autre et donc permettre de ne plus avoir de problème de garde. Ainsi, si on n'avait pas redéfini le type des listes infinies pour cette situation concrète, et de façon simultanée, on aurait toujours les mêmes problèmes. Or on ne peut définir simultanément que des types coinductifs (ou des types inductifs), mais pas un type coinductif et un type inductif. D'où la nécessité de passer par l'infini pour se restreindre ensuite. Mais cela veut également dire qu'on est obligé dans chaque cas de tout redéfinir simultanément. On ne peut pas ainsi avoir des propriétés générales sur les listes infinies qu'on peut réutiliser à chaque fois. Dans chaque cas concret il faudra les redéfinir et les redémontrer. Cela nous semble donc assez lourd à manipuler sur le long terme (même si dans certains cas précis cela peut être pratique). Il est difficile d'en extraire un résultat vraiment général.

2.2.4 L' imprédicativité

Une solution bien connue pour éviter les problèmes de garde est d'utiliser l'imprédicativité. Cependant, comme on l'a dit à la Section 1.4.1.1, dans Coq, l'univers des ensembles `Set` est prédicatif (plus de détails seront donnés sur ce point dans la Section 6.1.2.1). Cette solution ne peut donc être utilisée que dans l'univers des propositions, `Prop`. Dans le cas de *applyF2G* ce n'est donc pas une bonne option. Nous allons cependant l'étudier car il nous sera utile dans la suite.

Remarque 2.5. *L'étude que nous allons en faire ici reste cependant très légère et du domaine de l'intuition. En effet, nous n'avons utilisé l'imprédicativité que tardivement dans nos scripts et de façon assez marginale, plus à titre de test. De plus, c'est Ralph Matthes qui s'est chargé de cette partie là et elle ne rentre donc qu'indirectement dans le travail présenté ici.*

Nous allons tout d'abord présenter l'imprédicativité puis nous verrons qu'on peut dans Coq en utiliser une version plus légère, dite "à la Mendler", qui utilise quand même la coinduction telle que définie dans Coq.

2.2.4.1 Imprédicativité classique

Il s'agit ici simplement d'un changement de point de vue. On continue à considérer les définitions coinductivement (contrairement par exemple à la Section 2.2.1 où on abandonne partiellement la coinduction pour la remplacer par des fonctions) mais plus selon le mot clé `CoInductive` de Coq. On redéfinit en quelque sorte la coinduction. L'avantage c'est que comme tout est défini de façon ad-hoc et sans utiliser le mot clé `CoInductive`, on n'a plus de problème avec la condition de garde.

Le principe est de quantifier la propriété que l'on est en train de définir sur les propriétés en général (d'où l'appellation imprédicative). Il faut ensuite redéfinir "à la main" les trois notions de base dont on a besoin pour manipuler des définitions dans le style coinductif : le principe de coinduction, le principe de "dépliage" et la notion de constructeur.

Pour illustrer, redéfinissons la notion de bisimilarité sur les *Stream* dans un style imprédicatif. Elle est caractérisée comme ceci :

$$\forall s_1\ s_2, EqSt'\ s_1\ s_2 \quad \Leftrightarrow \quad \exists \mathcal{R}, \left(\forall s_1' s_2', \mathcal{R}\ s_1'\ s_2' \Rightarrow hd\ s_1' = hd\ s_2' \wedge \mathcal{R}\ (tl\ s_1')\ (tl\ s_2')\right) \wedge \mathcal{R}\ s_1\ s_2$$

On peut alors prouver les trois principes énoncés précédemment (on ne fait que donner leurs définitions ici, mais les preuves sont simples. On se sert en particulier du principe de coinduction pour prouver les deux autres puisque le principe de coinduction n'est qu'une autre lecture de la définition) :

[Le principe de coinduction] Supposons que

$$\forall s_1\ s_2,\ \mathcal{R}\ s_1\ s_2 \Rightarrow hd\ s_1 = hd\ s_2 \wedge \mathcal{R}\ (tl\ s_1)\ (tl\ s_2)$$

avec $\mathcal{R}$ une relation sur *Stream* T. Alors, $\forall s_1\ s_2,\ \mathcal{R}\ s_1\ s_2 \Rightarrow EqSt'\ s_1\ s_2$.

[Le principe de dépliage] $\forall s_1\ s_2, EqSt'\ s_1\ s_2 \Rightarrow hd\ s_1 = hd\ s_2 \wedge EqSt'\ (tl\ s_1)\ (tl\ s_2)$

[La notion de constructeur] $\forall s_1\ s_2, hd\ s_1 = hd\ s_2 \wedge EqSt'\ (tl\ s_1)\ (tl\ s_2) \Rightarrow EqSt'\ s_1\ s_2$

On peut montrer que $\forall s_1\ s_2, EqSt\ s_1\ s_2 \Leftrightarrow EqSt'\ s_1\ s_2$.

Remarque 2.6. *Cela est possible parce que EqSt est une relation "simple" (c'est-à-dire qu'elle est directe (elle n'appelle pas d'autres relations potentiellement inductives) et purement coinductive). Dans certains cas cette équivalence ne sera pas démontrable, voir pour exemple la Section 6.1.2.1.*

Pour plus de détails sur l'encodafe imprédicatif des plus grands points fixes, voir [39], où le problème de représentation est traité dans le système F.

2.2.4.2 Version dans le style de Mendler [81] et [61]

Avec le style de Mendler, on peut mélanger la souplesse de l'imprédicativité avec la maniabilité et la facilité d'utilisation (en particulier grâce à tous les outils fournis) de la coinduction telle que définie dans Coq. Cette fois-ci on utilise toujours la quantification universelle sur les propositions, mais en se servant du mot-clé `CoInductive` de Coq.

Pour illustrer nous allons donner l'exemple de la relation *EqSt* sur les *Stream*, encore une fois. Elle est définie comme suit, coinductivement :

$$\frac{\mathcal{R} \subseteq EqSt'' \qquad hd\ s_1 = hd\ s_2 \qquad \mathcal{R}\ (tl\ s_1)\ (tl\ s_2)}{EqSt''\ s_1\ s_2}$$

On peut montrer que $\forall s_1\ s_2, EqSt\ s_1\ s_2 \Leftrightarrow EqSt''\ s_1\ s_2$.

Remarque 2.7. *Ici encore, on peut montrer l'équivalence avec EqSt parce que EqSt est une relation "simple".*

Cependant, encore une fois, même si on se sert de la coinduction telle que définie dans Coq, on reste dans le monde imprédicatif et cette solution n'est donc utilisable que dans l'univers de propositions.

Pour plus de détails sur cette technique voir les articles de Uustalu et Vene [81] et de Nakata et Uustalu [61].

2.2.5 Dans Agda [30] et [31]

Dans [30] (ainsi que dans [31] avec Altenkirch), Danielsson décrit une méthode expérimentale proposée dans Agda pour contourner la condition de garde de Agda qui impose la même restriction concernant la productivité des définitions coinductives. Comme dans Coq, on ne peut pas définir *nats* (voir Section 2.2.1) directement dans Agda, la définition ne passant pas la condition de garde. Danielsson constate que la définition naturelle de *nats* est non gardée parce que l'appel corécursif se trouve sous *map* qui n'est pas un constructeur. L'idée est donc de dire que si certaines fonctions étaient en fait des constructeurs, de nombreuses définitions seraient gardées (ainsi si *map* était un constructeur, la définition naturelle de *nats* serait acceptée). Il propose donc de redéfinir chaque type coinductif en lui ajoutant autant de constructeurs que de fonctions "problématiques" (ici *map* par exemple). Il définit donc un langage dédié qui permet de définir un type avec les constructeurs "classiques" et des constructeurs spécifiques pour chaque fonction, comme *map*. Il définit ensuite des fonctions de conversion entre cette nouvelle définition du type et l'ancienne, qui ajoute une notion de comportement aux constructeurs additionnels (ainsi, *map* ne peut pas simplement être un constructeur classique, il faut pouvoir spécifier quelque part qu'il applique une fonction à tous les éléments de sa *Stream* paramètre).

De plus, Agda permettant de mélanger des constructeurs inductifs et coinductifs, cette solution permet également d'utiliser efficacement des types qui mélangent induction et coinduction. Cependant, cette proposition reste encore expérimentale dans Agda, qui est lui-même un outil assez expérimental.

Nous n'avons pas pu implanter cette solution dans Coq, même pour des exemples simples tels que *nats*, à priori principalement parce que le critère de garde dans Coq est plus syntaxique que celui d'Agda (dans Agda le système "déplie" un peu les définitions, ce qui facilite les choses, mais ne n'est pas le cas dans Coq). De plus, de par l'aspect expérimental de cette approche, nous avons préféré nous tourner vers quelque chose de plus pérenne, en attendant de voir quels résultats celle-ci pouvait donner.

Deuxième partie

Un outil pour la suite : un équivalent fonctionnel aux listes

3 Définition de *ilist* et propriétés de base

L'IDÉE ici est de réussir à se passer d'une structure inductive pour représenter des listes. Les listes sont généralement définies à l'aide de deux constructeurs comme présenté dans la Section 1.2.1.

Remarque 3.1 (Notations)**.** *Nous noterons @ la concaténation de deux listes.*

Par la suite, lorsque nous parlerons de listes, il sera toujours question de listes inductives définies comme précédemment.

Pour éviter d'utiliser une structure inductive, nous avons choisi de représenter les listes grâce à des fonctions (cette idée a aussi été proposée, entre autres, par Chlipala dans [21]). Cette solution est inspirée du travail de Bertot et Komendantskaya [16] qui utilisent des fonctions pour représenter des listes infinies. C'est en fait la version conteneur ("container") [75] des listes qui va être utilisée pour contourner la condition de garde, ce qui n'a jamais été fait à notre connaissance.

Une liste peut facilement être vue comme une fonction. En effet, à chaque position, elle associe une valeur du type de ses éléments (on l'appellera T). On peut donc la représenter par une fonction d'un ensemble à n éléments (n étant la longueur de la liste) dans T.

Par exemple, on peut transformer la liste $[10\ ;\ 2\ ;\ 5]$ en la fonction représentée graphiquement sur la Figure 3.1.

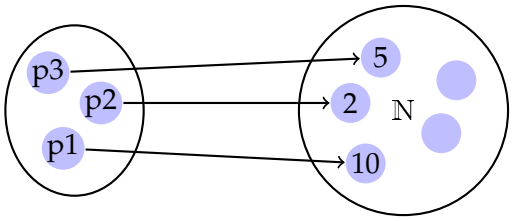


Figure 3.1 — Représentation fonctionnelle de la liste $[10\ ;\ 2\ ;\ 5]$

Une grande différence par rapport au travail de [16] est que pour eux l'ensemble de définition des fonctions était simplement $\mathbb{N}$. Pour nous, ce doit être un ensemble à n éléments. Comme on va le voir, cela va nettement compliquer les choses dans la suite (en particulier pour des raisons de conversions, si on veut ajouter ou enlever un élément).

3.1 Définition

La première chose à faire est donc de représenter l'ensemble fini à n éléments qui sera l'ensemble de définition de la fonction. Mais comme nous l'avons dit, la taille de cet ensemble correspond à la longueur de la liste et varie donc en fonction de celle-ci. Nous devons donc,

en réalité, représenter une famille d'ensembles, paramétrés par leur taille (le nombre de leurs éléments). Grâce à cela, nous pourrons définir l'équivalent fonctionnel des listes.

3.1.1 *Fin* - une famille de types pour des ensembles indexés finis

Nous allons donc représenter cette famille d'ensembles puis donner des outils pour la manipuler.

3.1.1.1 Définition

S'il est assez facile de représenter un ensemble à n éléments pour n fixé, ça l'est beaucoup moins pour n quelconque : il faut alors que n soit paramètre du type. Nous avons donc choisi la représentation utilisée également par Altenkirch dans [6] et par McBride et McKinna dans [58]. On l'appelle *Fin* et elle a pour type $\mathbb{N} \to Set$. Elle est définie par les deux constructeurs suivants :

Définition 3.1 (*Fin*, interprétée inductivement)**.**

$$\frac{n : \mathbb{N}}{first\ n : Fin\ (n+1)} \qquad \frac{n : \mathbb{N} \qquad i : Fin\ n}{succ\ i : Fin\ (n+1)}$$

Fin est ce que l'on appelle un type algébrique de données généralisé (GADT) parce que c'est un filtrage par le paramètre.

Remarque 3.2. *Le premier argument de succ, n, est déterminé par le type du second argument. Il est donc implicite et nous l'omettrons dans la suite. Nous ferons de même dans tous les cas similaires. C'est utile car cela allège beaucoup les notations et évite de donner des informations redondantes (c'est-à-dire qui peuvent être inférées à partir d'autres paramètres). Coq fait cette simplification automatiquement. Pour plus de détails sur les arguments implicites dans Coq, voir [34, Chapitre 2.7].*

Il est intéressant de voir que même si l'injectivité de Fin est difficile à démontrer (voir Lemme 3.10), Coq infère quand même tout seul le paramètre n de succ. En effet, Coq travaille syntaxiquement et ne prend pas en compte l'égalité de Leibniz qui rend l'injectivité difficile.

Pour valider cette définition, nous voulons montrer qu'elle remplit bien sa fonction : c'est-à-dire que *Fin n* permet bien de générer n éléments.

Lemme 3.1. $\forall n,\ card\ \{i \mid i : Fin\ n\} = n.$

Preuve (par induction).

[Cas 0] Aucun constructeur ne permet de créer un élément de type *Fin* 0.
Donc, $card\ \{\ i \mid i\ :\ Fin\ 0\ \}\ =\ 0$

[Cas n+1] L'hypothèse d'induction est : $IH : card\ \{i \mid i : Fin\ n\} = n$.
Le constructeur *succ* permet de créer autant d'éléments de types $Fin\ (n+1)$ qu'il y en a dans $Fin\ n$. Le constructeur *first* permet de créer un élément de plus de $Fin\ (n+1)$.
Donc, $card\ \{i \mid i : Fin\ (n+1)\} = card\ \{i \mid i : Fin\ n\} + 1 = n + 1$, en utilisant IH.

□

Remarque 3.3. *Nous utilisons ici card pour désigner informellement le cardinal d'un ensemble. Cette opération n'est pas disponible telle quelle dans Coq. Cependant, dans l'extension Ssreflect [79] de Coq, la notion de cardinal pour les types finis existe. Pour calculer ce cardinal, on doit alors fournir une liste qui correspond à une énumération de tous les éléments du type (chaque élément est une et une seule fois dans la liste). On peut également s'approcher de cette notion, sans utiliser Ssreflect, en fournissant une liste "énumération". On n'aura pas alors de fonction cardinal à proprement parler, mais on peut tout de même calculer le nombre d'éléments du type. Pour cela, on doit fournir la liste et prouver qu'il s'agit bien d'une énumération : c'est à dire que tous les éléments du type sont bien dans cette liste exactement une fois. Pour Fin, on peut par exemple utiliser la liste définie comme suit, qui prend en argument un entier naturel n et renvoie la liste de tous les éléments de Fin n.*

Définition 3.2 (*makeListFin*)**.**

$$\begin{array}{lcl} \mathit{makeListFin}\ n & : & \mathit{list}(\mathit{Fin}\ n) \\ \mathit{makeListFin}\ 0 & := & [] \\ \mathit{makeListFin}\ (n+1) & := & (\mathit{first}\ n)::(\mathit{map}\ \mathit{succ}\ (\mathit{makeListFin}\ n)) \end{array}$$

Remarque 3.4. *map est la fonction classique sur les listes, qui applique une fonction à tous les éléments d'une liste.*

On peut facilement démontrer les lemmes suivants :

Lemme 3.2. $\forall n, \mathit{length}\ (\mathit{makeListFin}\ n) = n.$

Lemme 3.3. $\forall n\ (i : \mathit{Fin}\ n), i \in \mathit{makeListFin}\ n$

Remarque 3.5. *On dénote par $\in$ l'appartenance d'un élément à une liste.*

Ces résultats nous permettent de nous convaincre déjà que makeListFin est bien une énumération de tous les éléments de Fin n. Pour le prouver formellement, nous aurions besoin d'outils présentés dans la suite. Cette preuve reste cependant simple et ne sera pas détaillée.

Nous pouvons déduire du fait qu'aucun élément n'a le type *Fin* 0 le lemme suivant :

Lemme 3.4. $\forall i : \mathit{Fin}\ 0, \mathit{false}$

3.1.1.2 Définitions sur *Fin*

Afin de faciliter la manipulation des éléments de *Fin*, nous allons définir quelques outils.

Premièrement, nous voulons définir une fonction qui nous permettra de transformer un élément i de *Fin* en entier naturel. L'idée est de "compter" le nombre de *succ* présents dans la définition de i. Ainsi, $\mathit{first}\ n$ sera associé à 0 (pour tout n) et $\mathit{succ}\ (\mathit{succ}\ (\mathit{first}\ n))$ à 2 (pour tout n). On appelle cette fonction *decode* et elle est définie comme suit :

Définition 3.3 (*decode*)**.**

$$\begin{array}{llcl} \mathit{decode} & n & : & \mathit{Fin}\ n \to \mathbb{N} \\ \mathit{decode} & (\mathit{first}\ n) & := & 0 \\ \mathit{decode} & (\mathit{succ}\ i') & := & (\mathit{decode}\ i') + 1 \end{array}$$

Une propriété intéressante que l'on peut démontrer à propos de *decode* est la suivante :

Lemme 3.5. $\forall n\ (i : Fin\ n),\ decode\ i < n$

La démonstration est une induction simple sur i, suivant le schéma de la définition.

Symétriquement à *decode*, on veut définir une fonction qui permet de transformer un entier naturel en élément de *Fin n*. En plus de l'entier m à transformer et de n, il faut également fournir une preuve de $m < n$ (pour se conformer à la propriété démontrée au Lemme 3.5). On appelle cette fonction *code* et elle est définie comme suit :

Définition 3.4 (*code*)**.**

$$\begin{array}{llllll} code & n & m & & : & m < n \to Fin\ n \\ code & (n+1) & 0 & h & := & first\ n \\ code & (n+1) & (m+1) & h & := & succ\ (code\ n\ m\ h') \end{array}$$

où h' est la conversion de $m+1 < n+1$ en $m < n$

Remarque 3.6. *On ne présente pas le cas où $n = 0$ parce qu'il est impossible (on aurait alors $m < 0$, ce qui est faux). On voit ici la nécessité d'avoir ajouté $m < n$ comme hypothèse.*

Remarque 3.7. *Lorsqu'on définit code, on doit lui fournir trois arguments : la taille totale n de l'ensemble, la valeur "numérique" m de l'élément de Fin à créer et une preuve de $m < n$. Maintenant, on peut déduire de $m < n$ les deux premiers paramètres. Comme on l'a fait pour succ, on pourra donc omettre ceux-ci dans la suite, et se contenter de donner à code uniquement son troisième paramètre, la preuve. Dans Coq, les paramètres implicites sont déterminés tout de suite après la définition.*

Dans la suite, on utilisera les équations de la Définition 3.4 mais en cachant les arguments implicites.

Une propriété importante à propos de *code* est l'indifférence de la preuve fournie (l'important n'est pas la preuve en elle même mais le résultat) :

Lemme 3.6 (Indifférence de la preuve dans *code*)**.** $\forall n\ m\ (h_1\ h_2 : m < n),\ code\ h_1 = code\ h_2$

La preuve est ici encore une induction simple qui suit directement la définition.

Remarque 3.8. *Le type $m < n$ n'admet qu'une seule preuve par rapport à l'égalité de Leibniz. Le résultat précédent n'est donc en fait qu'une instance spéciale de l'irrelevance de la preuve pour $m < n$. Cependant, nous n'avons pas besoin d'un résultat aussi général ici ; globalement, nous avons évité d'utiliser les résultats d'irrelevance de preuve dans nos développements. C'est pourquoi nous avons démontré ce genre de petits lemmes.*

Il est maintenant intéressant de montrer que les compositions de *code* et de *decode* donnent l'identité.

Lemme 3.7. $\forall n\ (i : Fin\ n)$, *code $h = i$ où h : decode $i < n$ est obtenu grâce au Lemme 3.5 instancié avec n et i*

Démonstration. Preuve détaillée en page 165. □

Lemme 3.8. $\forall n\ m\ (h : m < n),\ decode\ (code\ h) = m$

Démonstration. Preuve détaillée en page 166. □

Avec les Lemmes 3.7 et 3.8, nous avons en fait démontré qu'il y a une bijection entre *Fin n* et les segments d'entiers définis comme suit :

Définition 3.5 (*NatSeg*)**.** $NatSeg\ (n : \mathbb{N}) := \{\ m \mid m\ <\ n\ \}$

Enfin, nous voulons démontrer une propriété majeure pour la suite à propos de *decode*, son injectivité.

Lemme 3.9 (Injectivité de *decode*)**.** $\forall n\ (i_1\ i_2 : Fin\ n),\ decode\ i_1 = decode\ i_2 \Rightarrow i_1 = i_2$

Démonstration. On appellera h l'hypothèse $decode\ i_1 = decode\ i_2$.

$$
\begin{array}{lcll}
 & i_1 & = & i_2 \\
\Leftrightarrow & code\ h_1 & = & code\ h_2 \quad \text{(d'après le Lemme 3.7, } h_1 \text{ et } h_2 \text{ ont pour type} \\
 & & & \quad \text{respectivement } decode\ i_1 < n \text{ et } decode\ i_2 < n) \\
\Leftrightarrow & code\ h_1' & = & code\ h_2 \quad \text{(où } h_1' \text{ est le résultat de la réécriture de } h \text{ dans} \\
 & & & \quad \text{le type de } h_1 \text{ et a donc pour type } decode\ i_2 < n)
\end{array}
$$

Cette dernière équation est prouvée par le Lemme 3.6. □

Remarque 3.9 (Notations)**.** *Pour simplifier les notations dans la suite, nous noterons* $i =_{Fin} i'$ *pour* $decode\ i = decode\ i'$*, et ferons de même pour les autres opérateurs de comparaison* $<, \leq, >, \geq$ *que nous noterons respectivement* $<_{Fin}$, $\leq_{Fin}$, $>_{Fin}$ *et* $\geq_{Fin}$*.*

3.1.1.3 Injectivité de *Fin*

Une des propriétés majeures que l'on peut prouver sur *Fin* est son injectivité.

Lemme 3.10. $\forall n\ m,\ Fin\ n = Fin\ m \Rightarrow n = m$

Démonstration. La première chose que nous voulions faire était de montrer que tous les éléments de *Fin n* sont également dans *Fin m* et vice-versa, en ré-écrivant le type des éléments. Néanmoins, cela ne semble pas fonctionner en Coq (nous n'en avons, du moins, pas trouvé le moyen).

Toujours dans l'idée de faire la réécriture avec l'hypothèse, nous allons utiliser une étape intermédiaire. Nous allons convertir les éléments de *Fin n* en éléments de *Fin m*, et vice-versa.

En fait, nous allons montrer que s'il existe une bijection entre *Fin n* et *Fin m*, alors $n = m$. Nous nous servirons ensuite de cela pour montrer le résultat attendu.

Nous introduisons tout d'abord la notion de bijectivité avec la définition suivante :

Définition 3.6 (*bij*)**.**

$$\forall (f : T \to U)\ (g : U \to T), bij\ f\ g\ \Leftrightarrow (\forall t, g(f\ t) = t) \wedge (\forall u, f(g\ u) = u)$$

Remarque 3.10. *La surjectivité est un problème délicat d'un point de vue constructif. Et au lieu de seulement demander l'existence d'un inverse, nous préférons le fournir directement et qu'il fasse partie de notre définition de la bijection.*

De par la définition de *bij*, le lemme suivant sur la symétrie de *bij* se démontre trivialement :

Lemme 3.11 (*bij* symétrique). $\forall f\ g, bij\ f\ g \Leftrightarrow bij\ g\ f$

On peut également facilement démontrer la transitivité de la bijection pour la composition :

Lemme 3.12.
$$\forall (f_1 : T \to U)(g_1 : U \to T)(f_2 : U \to V)(g_2 : V \to U),\\ bij\ f_1\ g_1 \wedge bij\ f_2\ g_2 \Rightarrow bij\ (f_2 \circ f_1)(g_1 \circ g_2)$$

Démonstration. Preuve détaillée en page 166. □

Enfin, on peut également montrer que si *bij f g* alors *f* est injective (ce qui est attendu, bien entendu).

Lemme 3.13. $\forall (f : T \to U)\ (g : U \to T)\ t_1\ t_2, bij\ f\ g \wedge f\ t_1 = f\ t_2 \Rightarrow t_1 = t_2$

Démonstration. Preuve détaillée en page 167. □

Nous voulons donc montrer le lemme suivant :

Lemme 3.14. $\forall (f : Fin\ n \to Fin\ m)\ (g : Fin\ m \to Fin\ n), bij\ f\ g \Rightarrow n = m$

Voyons tout d'abord comment ce lemme nous permet de terminer la preuve.

D'après le Lemme 3.14, pour prouver que $n = m$, il nous suffit de prouver qu'il existe une bijection entre *Fin n* et *Fin m*. Ce que l'on doit faire ici, c'est donc convertir un élément *i* du type *Fin n* en un élément du type *Fin m*. En Coq, une fonction spéciale de filtrage par motif permet d'effectuer cette opération (nous l'avons déjà présentée Section 1.4.1.2). Nous allons ici l'utiliser dans le cas de *Fin* avec une hypothèse *Fin n = Fin m* (plus loin, nous l'utiliserons avec une hypothèse de type $n = m$). Elle a pour type $\forall n\ m\ (h : Fin\ n = Fin\ m), Fin\ n \to Fin\ m$. Nous l'appellerons *convFin*. On notera $convFin_h\ i$ l'élément *i* de type *Fin n* converti au type *Fin m* par l'égalité $h : Fin\ n = Fin\ m$.

Soit $H : Fin\ n = Fin\ m$. Il est très aisé de prouver que $\forall i, convFin_{sym\ H}(convFin_H\ i) = i$ et que $\forall i, convFin_H(convFin_{sym\ H}\ i) = i$. On a donc $H_1 : bij\ convFin_H\ convFin_{sym\ H}$, ce que l'on voulait. Pour terminer la preuve, il nous reste donc à démontrer le Lemme 3.14. □

Démonstration du Lemme 3.14. On a *f* et *g* telles que *bij f g*. On veut montrer que $n = m$. Raisonnons par induction sur n.

[Cas 0**]** Voyons les différentes possibilités pour m :

- **[Cas** 0**]** On veut alors démontrer que $0 = 0$, ce qui est vrai par réflexivité.
- **[Cas** $m + 1$**]** On veut démontrer que $0 = m + 1$. Manifestement, ceci est faux, on doit donc trouver une hypothèse fausse. La fonction *g* est de type $Fin\ (m + 1) \to Fin\ 0$. Donc $g(first\ m)$ est de type *Fin* 0, ce qui est faux puisque *Fin* 0 est vide.

[Cas $n + 1$**]** Voyons les différentes possibilités pour m :

- **[Cas** 0**]** On veut alors démontrer que $n + 1 = 0$. Comme précédemment on doit trouver une hypothèse fausse. La fonction *f* est de type $Fin\ (n + 1) \to Fin\ 0$. Donc $f(first\ n)$ est de type *Fin* 0, ce qui est faux puisque *Fin* 0 est vide.

[Cas $m+1$] L'hypothèse d'induction est

$$IH : \forall m\ (f' : Fin\ n \to Fin\ m)\ (g' : Fin\ m \to Fin\ n), bij\ f'\ g' \Rightarrow n = m$$

On veut montrer que $n+1 = m+1$. Il nous suffit donc en fait de prouver que $n = m$. D'après IH, il nous suffit de trouver $f' : Fin\ n \to Fin\ m$ et $g' : Fin\ m \to Fin\ n$ telles que $bij\ f'\ g'$. Pour cela, nous allons nous servir de f et g. On veut en quelque sorte transformer f et g en fonctions de type $Fin\ n \to Fin\ m$ et $Fin\ m \to Fin\ n$. Nous allons tout d'abord donner une idée du raisonnement avant de fournir la définition formelle de la fonction de conversion. Raisonnons dans le cas général. Soient n_1 et n_2 deux entiers. Supposons qu'on a deux fonctions $f_1 : Fin\ (n_1+1) \to Fin\ (n_2+1)$ et $f_2 : Fin\ (n_2+1) \to Fin\ (n_1+1)$ telle que $bij\ f_1\ f_2$. A partir de f_1 on veut obtenir $f_1' : Fin\ n_1 \to Fin\ n_2$. Soit donc i_1 de type $Fin\ n_1$. On veut obtenir un élément de $Fin\ n_2$. On obtient facilement avec $succ$ un élément de $Fin\ (n_1+1)$ et $f_1(succ\ i_1)$ est un élément de $Fin\ (n_2+1)$. La dernière étape est donc de réussir à obtenir un élément du type $Fin\ n_2$. On a deux possibilités pour i_2. Soit il est de la forme $succ\ i$ avec i de type $Fin\ n_2$ et on a ce qu'on voulait. Soit, il est de la forme $first\ n_2$. Si c'est le cas, alors d'après le Lemme 3.13 on sait que $f_1(first\ n_1)$ est de la forme $succ\ i$ (sinon on aurait $f_1(first\ n_1) = first\ n_2$, c'est-à-dire $f_1(first\ n_1) = f_1(succ\ i_1)$ c'est-à-dire d'après le Lemme 3.13 $first\ n_1 = succ\ i_1$, ce qui est faux). On associe donc ce i à i_1. Pour donner la définition formelle de cette fonction de conversion qu'on appellera *transfoFun*, nous aurons besoin d'une fonction qui permet de récupérer l'élément i' de Fin qui se trouve dans un élément i de la forme $succ\ i'$ (et qui est caractérisé par $0 < decode\ i$). On appelle cette fonction *getcons* et elle est définie comme suit :

Définition 3.7 (*getcons*)**.**

$$\begin{array}{lcl} getcons\ (n : \mathbb{N})(i : Fin\ (n+1)) : 0 < decode\ i \to Fin\ n & & \\ getcons\ n\ (succ\ i)\ h & = & i \\ getcons\ n\ (first\ n)\ h & = & non\ défini\ (contradiction\ avec\ h) \end{array}$$

On démontre immédiatement les propriétés suivantes à propos de *getcons* :

Propriété 3.1.

$$\forall n\ (i : Fin\ (n+1))\ (h : 0 < decode\ i),\ succ\ (getcons\ i\ h) = i \tag{3.1.1}$$
$$\forall n\ (i : Fin\ n)\ (h : 0 < decode\ (succ\ i)), getcons\ (succ\ i)\ h = i \tag{3.1.2}$$

On définit maintenant *transfoFun* :

Définition 3.8 (*transfoFun*)**.**

$$\begin{array}{l} transfoFun(f_1 : Fin(n_1+1) \to Fin(n_2+1))\ (f_2 : Fin(n_2+1) \to Fin(n_1+1)) : \\ \quad bij\ f_1\ f_2 \to Fin\ n_1 \to Fin\ n_2 \\ transfoFun\ f_1\ f_2\ H\ i := \begin{cases} getcons\ (f_1\ (first\ n_1))\ H_1 & si\ H' : decode(f_1(succ\ i)) = 0 \\ \qquad et\ avec\ H_1 : 0 < decode(f_1(first\ n_1))\ déduit\ de\ H' & \\ \qquad et\ du\ Lemme\ 3.13 & \\ getcons\ (f_1\ (succ\ i))\ H' & si\ H' : 0 < decode(f_1(succ\ i)) \end{cases} \end{array}$$

Remarque 3.11. *Ici on ne fournit f_2 que pour l'hypothèse bij, sinon, elle ne joue aucun rôle dans la définition.*

Avec cette définition, on peut transformer les deux fonctions. Reste à montrer que les fonctions transformées forment bien une bijection. On montre d'abord qu'une des compositions donne bien l'identité, l'autre se déduira immédiatement de cette première preuve. Pour simplifier encore la preuve du Lemme 3.16 on donne deux hypothèses de *bij* (une dans chaque sens) :

Lemme 3.15.

$$\forall f_1\ f_2\ (H_1 : bij\ f_1\ f_2)(H_2 : bij\ f_2\ f_1), \forall i, transfoFun\ H_2\ (transfoFun\ H_1\ i) = i$$

Démonstration. Preuve détaillée en page 167. □

Il nous reste donc à montrer qu'on a bien obtenu une bijection :

Lemme 3.16. $\forall f_1\ f_2\ (H : bij\ f_1\ f_2), bij\ (transfoFun\ H)\ (transfoFun\ H')$ *avec* $H' : bij\ f_2\ f_1$ *déduit de H et du Lemme 3.11.*

Démonstration. Preuve détaillée en page 168. □

Revenons à la preuve initiale. On a $f : Fin(n+1) \to Fin(m+1)$ et $g : Fin(m+1) \to Fin(n+1)$ telles que $H : bij\ f\ g$. Pour terminer notre preuve il nous suffit de trouver $f' : Fin\ n \to Fin\ m$ et $g' : Fin\ m \to Fin\ n$ telle que $bij\ f'\ g'$. On prend donc $f' := transfoFun\ H$ et $g' := transfoFun\ H'$ avec $H' : bij\ g\ f$ déduit de H et du Lemme 3.11. On prouve $bij\ f'\ g'$ avec le Lemme 3.16.

□

Remarque 3.12. *En utilisant la notion d'énumération évoquée Remarque 3.3, on peut obtenir une autre preuve de l'injectivité de Fin. Le développement est d'une complexité équivalente à celui proposé ici et n'est pas détaillé.*

Remarque 3.13 (Discussion à propos de la représentation de *Fin*)**.** *Comme on l'a démontré, Fin est équivalent au type des segments d'entiers présenté précédemment, NatSeg. Nous avons fait le choix de travailler avec Fin parce que c'est un type inductif, fortement structuré et ordonné. De plus, les éléments de NatSeg n sont composés d'un entier m et d'une preuve que* $m < n$*. Pour travailler confortablement avec ces éléments, il est bon d'utiliser l'irrelevance de la preuve de* $m < n$*. Même si cela est raisonnable, comme nous l'avons expliqué, nous n'avons pas souhaité travailler avec ce type de résultats généraux. Nous avons donc fait le choix de Fin, mais il aurait été possible de travailler de façon équivalente avec NatSeg. D'ailleurs, à chaque fois que l'on utilise decode (et cela arrive souvent), cela revient plus ou moins à utiliser NatSeg.*

3.1.2 Implémentation de *ilist*

Nous pouvons maintenant définir le type qui sera l'équivalent fonctionnel aux listes. La fonction en question a deux paramètres : le type des éléments de la liste et sa longueur (c'est-à-dire le paramètre de taille de *Fin*, sans quoi on ne pourrait pas définir l'ensemble de départ). On l'appelle *ilistn* et elle est définie de la façon suivante :

Définition 3.9 (*ilistn*)**.** $ilistn\ T\ n := Fin\ n \to T$

Les éléments de *ilistn* imitent les listes : à chaque élément d'un ensemble à n éléments, il associe un élément de type T. Cependant, il reste un problème : *ilistn* a besoin de deux paramètres. Or, une liste, elle, n'a besoin que de connaître le type des ses éléments. Sa taille

lui est inhérente. Nous créons donc un nouveau type qui contient à la fois la taille de la liste et la fonction (l'élément de type *ilistn*) associée. Nous appelons ce type *ilist* (pour *indexed list*).

Définition 3.10. $ilist\ T := \Sigma n : \mathbb{N}.\ ilistn\ T\ n$

On utilise ici le couple dépendant dénoté par $\Sigma x : A.B(x)$. Les éléments de ce type consistent en un élément a de type A et un élément b de type $B(a)$.

Remarque 3.14. *Dans la vue conteneur, n correspond à la forme ("shape") et Fin n est le type des positions (un élément de Fin n est une position).*

Nous appelons *lg* et *fct* les deux projections sur *ilist* qui correspondent respectivement à la longueur d'un élément de *ilist* et à sa fonction. Si on note $\langle \ldots, \ldots \rangle$ le constructeur pour des éléments de type $\Sigma x : A.B(x)$, alors un élément l de type $ilist\ T$ peut être "reconstruit" sous la forme $\langle lg\ l, fct\ l \rangle$.

3.2 Définitions et propriétés sur *ilist*

Nous voulons maintenant définir un certain nombre d'outils et de propriétés qui faciliteront la manipulation de *ilist*.

3.2.1 Une relation d'équivalence sur *ilist*

La première chose dont nous allons avoir besoin est une relation d'équivalence sur *ilist*. Cela est nécessaire pour plusieurs raisons. La première est que lorsqu'on compare deux éléments de *ilist*, de façon sous-jacente ce sont des fonctions que l'on compare réellement. Or ici déjà l'égalité de Leibniz ne suffit plus. En effet, comme on voudrait que *ilist* se comporte de la même façon que les listes, on voudrait que lorsque deux listes sont égales (par rapport à l'égalité de Leibniz) leurs équivalents en *ilist* le soient également. Par exemple, on voudrait que tous les éléments de *ilist* qui sont "vides" (c'est-à-dire pour lesquels $lg\ l = 0$) soient équivalents. Or, si on prend par exemple les éléments de *ilist* suivants : $\langle 0, \lambda i : Fin\ 0.3 \rangle$ et $\langle 0, \lambda i : Fin\ 0.0 \rangle$, ils ne sont clairement pas égaux aux yeux de l'égalité de Leibniz (comment prouver que $\lambda i : Fin\ 0.3 = \lambda i : Fin\ 0.0$?). On voudrait pourtant qu'ils soient équivalents et, plus généralement, nous voudrions pouvoir comparer les fonctions point à point. En plus de cela, il faut également que les longueurs des deux éléments soient égales. Il apparaît donc comme nécessaire de définir une relation d'équivalence sur *ilist* qu'on appellera *ilist_rel* dans la suite. Dans un premier temps, nous la définissons comme suit :

$$\forall l_1\ l_2 : ilist\ T, ilist\_rel\ l_1\ l_2 \Leftrightarrow lg\ l_1 = lg\ l_2 \wedge \forall i : Fin\ (lg\ l_1), fct\ l_1\ i = fct\ l_2\ i$$

Cependant, cette expression n'est pas bien typée dans Coq. En effet $fct\ l_2$ a pour type $Fin\ (lg\ l_2) \rightarrow T$ et i a pour type $Fin\ (lg\ l_1)$. Même si nous savons $lg\ l_1 = lg\ l_2$, les types $Fin\ (lg\ l_1)$ et $Fin\ (lg\ l_2)$ restent syntaxiquement différents. Et pourtant grâce au Lemme 3.10 on a bien $Fin\ (lg\ l_1) = Fin\ (lg\ l_2)$. Comme on l'a dit précédemment, cela signifie que si pour deux types T et U on a $T = U$, un élément t de type T n'a pas pour autant le type U. En revanche, l'égalité (sémantique) des types nous permet de convertir t en un élément de U en faisant simplement une réécriture de type dans la définition de t. Noud devons donc ici convertir i en un élément du type $Fin\ (lg\ l_2)$. Cette fois-ci, la fonction de conversion a pour

type $\forall n\ m\ (h : n = m), Fin\ n \to Fin\ m$ (grâce au Lemme 3.10, on peut se contenter d'une égalité sur les paramètres au lieu d'une égalité sur les types). Nous appellerons cette fonction de *conv*. On notera $conv_h\ i$ l'élément i de type $Fin\ n$ converti au type $Fin\ m$ par l'égalité $h : n = m$. La fonction *conv* est définie de telle sorte que la propriété suivante soit vraie :

Propriété 3.2. $\forall n\ m\ (h : n = m)\ (i : Fin\ n),\ i =_{Fin} (conv_h\ i)$

On peut donc maintenant redéfinir *ilist_rel* afin qu'elle soit bien typée :

$$\forall l_1\ l_2 : ilist\ T, ilist\_rel\ l_1\ l_2 \Leftrightarrow \exists h : lg\ l_1 = lg\ l_2, \forall i : Fin\ (lg\ l_1), fct\ l_1\ i = fct\ l_2\ (conv_h\ i)$$

Cependant, nous supposons ici que les éléments de T sont comparables par l'égalité de Leibniz ce qui n'est pas toujours vrai. En particulier, dans les utilisations que nous visons et qui mélangeront listes et types coinductifs, le type T sera lui-même coinductif. Or, l'égalité de Leibniz est généralement trop fine pour comparer deux éléments d'un type coinductif. Nous devons donc fournir à *ilist_rel* la relation à utiliser pour comparer les éléments de T (qui pourra être l'égalité de Leibniz le cas échéant), et l'utiliser. Nous pouvons donc finalement définir *ilist_rel* de la façon suivante :

Définition 3.11 (*ilist_rel*)**.**

$$\forall l_1\ l_2 : ilist\ T, ilist\_rel\ R\ l_1\ l_2 \Leftrightarrow \exists h : lg\ l_1 = lg\ l_2,\ \forall i : Fin\ (lg\ l_1), R\ (fct\ l_1\ i)\ (fct\ l_2\ (conv_h\ i))$$

Remarque 3.15. *Dans la suite, nous mettrons l'argument R en indice de ilist_rel, c'est-à-dire que nous écrirons* $ilist\_rel_R$ *pour ilist_rel R. Nous ferons de même dans tous les cas similaires.*

Comme premier résultat sur *ilist_rel*, on peut démontrer le lemme suivant, dont on a dit qu'on attendait qu'il soit vrai :

Lemme 3.17. $\forall R\ (ln_1\ ln_2 : ilistn\ T\ 0),\ ilist\_rel_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle$

Démonstration. Preuve détaillée en page 168. □

Montrons maintenant que *ilist_rel* préserve l'équivalence (c'est-à-dire que si R est une relation d'équivalence, alors $ilist\_rel_R$ en est une aussi). Pour cela, nous montrons séparément la préservation de la réflexivité, de la symétrie et de la transitivité, puis nous énonçons le lemme final.

Lemme 3.18 (*ilist_rel* préserve la réflexivité)**.** $R\ réflexive \Rightarrow \forall l, ilist\_rel_R\ l\ l$

Démonstration. On veut prouver que $ilist\_rel_R\ l\ l$. On sait par réflexivité de l'égalité de Leibniz que $H_1 : lg\ l = lg\ l$. On applique donc la Définition 3.11 et on doit prouver que : $\forall i, R\ (fct\ l\ i)\ (fct\ l\ (conv_{H_1}\ i))$, ce qui se simplifie en $R\ (fct\ l\ i)\ (fct\ l\ i)$, puisque $conv_{H_1}\ i = i$ par convertibilité. Ce qu'on prouve avec la réflexivité de R. □

Lemme 3.19 (*ilist_rel* préserve la symétrie)**.**

$$R\ symétrique \Rightarrow (\forall l_1\ l_2, ilist\_rel_R\ l_1\ l_2 \Rightarrow ilist\_rel_R\ l_2\ l_1)$$

Démonstration. Soit $H_1 : ilist\_rel_R\ l_1\ l_2$. On veut prouver que $ilist\_rel_R\ l_2\ l_1$. H_1 avec la Définition 3.11 nous donne deux nouvelles hypothèses :

$$H_2 : lg\ l_1 = lg\ l_2 \quad H_3 : \forall i, R\ (fct\ l_1\ i)\ (fct\ l_2\ (conv_{H_2}\ i))$$

On applique la Définition 3.11 avec le symétrique de H_2 ($sym\ H_2$) et on doit prouver que :

$$\forall i, R\ (fct\ l_2\ i)\ (fct\ l_1\ (conv_{(sym\ H_2)}\ i))$$

On montre aisément que $i = conv_{H_2}\ (conv_{(sym\ H_2)}\ i)$. On doit donc prouver que :

$$R\ (fct\ l_2\ (conv_{H_2}\ (conv_{(sym\ H_2)}\ i)))\ (fct\ l_1\ (conv_{(sym\ H_2)}\ i))$$

Ce qu'on prouve avec la symétrie de R et H_3. □

Lemme 3.20 (*ilist_rel* préserve la transitivité)**.**

$$R\ transitive \Rightarrow (\forall l_1\ l_2\ l_3, ilist\_rel_R\ l_1\ l_2 \wedge ilist\_rel_R\ l_2\ l_3 \Rightarrow ilist\_rel_R\ l_1\ l_3)$$

Démonstration. Soient H_1 : $ilist\_rel_R\ l_1\ l_2$ et H_2 : $ilist\_rel_R\ l_2\ l_3$. On veut prouver que $ilist\_rel_R\ l_1\ l_3$. H_1 et H_2 avec la Définition 3.11 nous donnent quatre nouvelles hypothèses :

$$H_3 : lg\ l_1 = lg\ l_2 \quad H_4 : \forall i, R\ (fct\ l_1\ i)\ (fct\ l_2\ (conv_{H_3}\ i))$$
$$H_5 : lg\ l_2 = lg\ l_3 \quad H_6 : \forall i, R\ (fct\ l_2\ i)\ (fct\ l_3\ (conv_{H_5}\ i))$$

On applique la Définition 3.11 avec le transitif de H_3 et H_5 ($trans\ H_3\ H_5$) et on doit prouver que :

$$\forall i, R\ (fct\ l_1\ i)\ (fct\ l_3\ (conv_{(trans\ H_3\ H_5)}\ i))$$

On montre aisément que $conv_{(trans\ H_3\ H_5)}\ i = conv_{H_5}\ (conv_{H_3}\ i)$. On doit donc prouver que :

$$R\ (fct\ l_1\ i)\ (fct\ l_3\ (conv_{H_5}\ (conv_{H_3}\ i)))$$

Ce qu'on prouve avec la transitivité de R, H_4 et H_6. □

On peut donc maintenant énoncer le lemme final :

Lemme 3.21 (*ilist_rel* préserve l'équivalence)**.** *R équivalence $\Rightarrow$ $ilist\_rel_R$ équivalence*

Démonstration. La preuve est simplement une application des trois lemmes précédents. □

On peut également démontrer que *ilist_rel* est monotone par rapport à sa relation de base (on notera par $R_1 \subseteq R_2$ le fait que R_1 et une sous relation de R_2, c'est-à-dire que $\forall t_1\ t_2, R_1\ t_1\ t_2 \Rightarrow R_2\ t_1\ t_2$) :

Lemme 3.22. $\forall R_1\ R_2\ l_1\ l_2, R_1 \subseteq R_2\ \wedge\ ilist\_rel_{R_1}\ l_1\ l_2 \Rightarrow ilist\_rel_{R_2}\ l_1\ l_2$

Démonstration. Preuve détaillée en page 168. □

Remarque 3.16 (*list_rel*)**.** *Comme on l'a expliqué précédemment, on a créé le type ilist parce que le mélange entre listes (inductives) et types coinductifs se passe mal en Coq. Cependant, même si on avait pu les utiliser, on aurait dû redéfinir une relation sur les listes, pour la seconde raison invoquée (la relation sur T). Nous l'avons déjà présentée précédemment, il s'agit de list_rel (Définition 1.4).*

3.2.2 Décidabilité de *ilist_rel*

On veut prouver que *ilist_rel* préserve la décidabilité. On définit la décidabilité d'une relation de la façon suivante :

Définition 3.12 (R décidable)**.** $\forall R, Dec\ R \Leftrightarrow \forall t_1\ t_2, R\ t_1\ t_2 \vee \neg(R\ t_1\ t_2)$

Remarque 3.17. *Ici, la disjonction $\vee$ a une forme fortement constructive qui vient avec une preuve du cas qui a été démontré (en Coq, il s'agit de* `sumbool` *qui appartient à l'univers* `Set` *des types calculables). On notera R_d les relations pour lesquelles on a de telles procédures de décision.*

On peut maintenant exprimer le fait que *ilist_rel* préserve la décidabilité :

Lemme 3.23. $Dec\ R \Rightarrow Dec\ (ilist\_rel_R)$

Idée de la preuve. L'idée ici est de montrer que comme, grâce à l'hypothèse H : $Dec\ R$, on peut décider pour chaque position si les éléments de deux *ilist* sont équivalents, on peut décider globalement si deux *ilist* sont équivalentes (par rapport à *ilist_rel*). Pour cela on va suivre la structure de *ilist_rel*. On va donc tout d'abord comparer les longueurs des deux *ilist*. Si elles sont différentes, alors les deux *ilist* ne sont pas en relation par *ilist_rel*. Si elles sont égales alors on raisonne par induction sur cette longueur. On comparera donc ensuite les "têtes" des deux *ilist* (en utilisant H) puis récursivement la "queue".

Démonstration. On appelle H l'hypothèse $Dec\ R$, c'est-à-dire qu'on a :

$$H : \forall t_1\ t_2, R\ t_1\ t_2 \vee \neg(R\ t_1\ t_2)$$

Et on veut montrer : $Dec\ (ilist\_rel_R)$, c'est-à-dire

$$\forall l_1\ l_2, ilist\_rel_R\ l_1\ l_2 \vee \neg(ilist\_rel_R\ l_1\ l_2)$$

Ou encore en explicitant l_1 et l_2 :

$$\forall n_1\ n_2\ ln_1\ ln_2, ilist\_rel_R\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle \vee \neg(ilist\_rel_R\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle)$$

Comparons tout d'abord n_1 et n_2 :

[Cas $H_0 : n_1 = n_2$**]** On va donc se passer ici de n_2 (en faisant une réécriture de type avec l'hypothèse H_0) et on veut montrer que :

$$ilist\_rel_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle \vee \neg(ilist\_rel_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle)$$

On va raisonner par récurrence sur n_1.

[Cas 0**]** Ici, nous prouvons directement que $ilist\_rel_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle$ en utilisant le Lemme 3.17.

[Cas $n_1 + 1$**]** L'hypothèse de récurrence est :

$$IH : ilist\_rel_R\ \langle n_1, ln_1'\rangle\ \langle n_1, ln_2'\rangle \vee \neg(ilist\_rel_R\ \langle n_1, ln_1'\rangle\ \langle n_1, ln_2'\rangle)$$

et nous voulons prouver que :

$$ilist\_rel_R\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle \vee \neg(ilist\_rel_R\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle)$$

Nous allons pour cela utiliser l'hypothèse H avec $t_1 = ln_1\ (first\ n_1)$ et $t_2 = ln_2\ (first\ n_1)$. Nous allons étudier les deux cas possibles :

[Cas $H_1 : R\ (ln_1\ (\mathit{first}\ n_1))\ (ln_2\ (\mathit{first}\ n_1))$**]** Nous savons donc que les deux premiers éléments de *ilist* sont égaux, mais qu'en est-il du reste ? Cela ne nous permet pas de décider si les deux *ilist* sont égales ou non. Pour cela, nous devons encore analyser les deux possibilités offertes par *IH* (en prenant comme *ilistn* paramètres, ln_1 et ln_2 privées de leur premier élément) :

[Cas $H_2 : \mathit{ilist\_rel}_R\ \langle n_1, ln_1 \circ \mathit{succ}\rangle\ \langle n_1, ln_2 \circ \mathit{succ}\rangle$**]** Nous voulons prouver ici que $\mathit{ilist\_rel}_R\ \langle n_1+1, ln_1\rangle\ \langle n_1+1, ln_2\rangle$. On a :

$$\begin{array}{ll} & \mathit{ilist\_rel}_R\ \langle n_1+1, ln_1\rangle\ \langle n_1+1, ln_2\rangle \\ \Leftrightarrow & \exists h : lg\ \langle n_1+1, ln_1\rangle = lg\ \langle n_1+1, ln_2\rangle,\ \forall i : Fin\ (lg\ \langle n_1+1, ln_1\rangle), \\ & \quad R\ (\mathit{fct}\ \langle n_1+1, ln_1\rangle\ i)\ (\mathit{fct}\ \langle n_1+1, ln_2\rangle\ (\mathit{conv}_h\ i)) \\ \Leftrightarrow & \exists h : n_1+1 = n_1+1,\ \forall i : Fin\ (n_1+1), R\ (ln_1\ i)\ (ln_2\ (\mathit{conv}_h\ i)) \end{array}$$

h s'obtient trivialement par réflexivité et on peut simplifier la dernière expression au vu de h. Il nous suffit donc de prouver que $\forall i : Fin\ (n_1 + 1), R\ (ln_1\ i)\ (ln_2\ i)$. Pour cela, étudions les deux possibilités pour i :

[Cas $\mathit{first}\ n_1$**]** On veut prouver que $R\ (ln_1\ (\mathit{first}\ n_1))\ (ln_2\ (\mathit{first}\ n_1))$, ce qui est l'hypothèse H_1

[Cas $\mathit{succ}\ i'$**]** On veut prouver que $R\ (ln_1\ (\mathit{succ}\ i'))\ (ln_2\ (\mathit{succ}\ i'))$. L'hypothèse H_2 nous donne, d'après la Définition 3.11, les deux nouvelles hypothèses suivantes (on les simplifie directement comme on a fait précédemment) :

$$H_3 : n_1 = n_1 \quad \text{et} \quad H_4 : \forall i : Fin\ n_1,\ R\ (ln_1\ (\mathit{succ}\ i))\ (ln_2\ (\mathit{succ}\ i))$$

On peut donc prouver ce que l'on voulait en utilisant directement H_4 avec i'.

[Cas $H_2 : \neg(\mathit{ilist\_rel}_R\ \langle n_1, ln_1 \circ \mathit{succ}\rangle\ \langle n_1, ln_2 \circ \mathit{succ}\rangle)$**]** Nous voulons donc prouver ici que : $\neg\ (\mathit{ilist\_rel}_R\ \langle n_1+1, ln_1\rangle\ \langle n_1+1, ln_2\rangle)$. Puisqu'il s'agit d'une négation, le principe que nous allons utiliser ici, et dans tous les cas similaires, est le raisonnement par l'absurde (au sens où nous l'avons expliqué Section 1.4.2). Supposons que nous avons l'hypothèse $H_3 : \mathit{ilist\_rel}_R\ \langle n_1+1, ln_1\rangle\ \langle n_1+1, ln_2\rangle$ et montrons que nous arrivons à une contradiction. On peut très facilement montrer que (la preuve n'est pas détaillée ici) :

$$\begin{array}{l} \forall n\ (ln_1\ ln_2 : \mathit{ilistn}\ (n+1)), \mathit{ilist\_rel}_R\ \langle n+1, ln_1\rangle\ \langle n+1, ln_2\rangle \\ \Rightarrow \mathit{ilist\_rel}_R\ \langle n, ln_1 \circ \mathit{succ}\rangle\ \langle n, ln_2 \circ \mathit{succ}\rangle \end{array}$$

En appliquant ce résultat à l'hypothèse H_3, on obtient que

$$\mathit{ilist\_rel}_R\ \langle n_1, ln_1 \circ \mathit{succ}\rangle\ \langle n_1, ln_2 \circ \mathit{succ}\rangle$$

ce qui est en contradiction avec l'hypothèse H_2.

[Cas $H_1 : \neg\ (R\ (ln_1\ (\mathit{first}\ n_1))\ (ln_2\ (\mathit{first}\ n_1)))$**]** Ici aussi on veut démontrer que $\neg\ (\mathit{ilist\_rel}_R\ \langle n_1+1, ln_1\rangle\ \langle n_1+1, ln_2\rangle)$. Nous allons donc raisonner par l'absurde. Supposons donc que $H_3 : \mathit{ilist\_rel}_R\ \langle n_1+1, ln_1\rangle\ \langle n_1+1, ln_2\rangle$. Grâce à la Définition 3.11, nous obtenons de H_3 les deux hypothèses suivantes (en simplifiant directement) :

$$H_4 : n_1+1 = n_1+1 \quad \text{et} \quad H_5 : \forall i : Fin\ (n_1+1),\ R\ (ln_1\ i)\ (ln_2\ i)$$

En utilisant H_5 avec $\mathit{first}\ n_1$, on obtient : $R\ (ln_1\ (\mathit{first}\ n_1))\ (ln_2\ (\mathit{first}\ n_1))$, ce qui est en contradiction avec l'hypothèse H_1.

[Cas $H_0 : n_1 \neq n_2$**]** Enfin, ici nous voulons prouver que $\neg\ (ilist\_rel_R\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle)$. Pour cela, nous allons encore utiliser un raisonnement par l'absurde. Nous supposons que H_2 : $ilist\_rel_R\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle$ et nous obtenons grâce à la Définition 3.11 les deux hypothèses suivantes :

$$H_3 : n_1 = n_2 \quad \text{et} \quad H_4 : \forall i : Fin\ n_1,\ R\ (ln_1\ i)\ (ln_2\ (conv_{H_3}\ i))$$

L'hypothèse H_3 est directement en contradiction avec H_0.

□

3.2.3 Bijection entre *ilist* et listes

Pour valider la définition de *ilist*, par rapport à nos besoins (nous voulions un équivalent fonctionnel aux listes), nous voulons montrer qu'il y a une bijection entre *ilist* et les listes.

Pour cela, nous allons créer deux fonctions : une qui permettra de transformer un élément de *ilist* en une liste (*ilist2list*), l'autre qui transformera une liste en un élément de *ilist* (*list2ilist*). Nous prouverons ensuite que les compositions de ces deux fonctions (*list2ilist* ∘ *ilist2list* et *ilist2list* ∘ *list2ilist*) sont point à point égales à l'identité.

Pour définir *ilist2list*, on procède en deux étapes. D'abord on crée une liste qui contient tous les éléments de $Fin(lg\ l)$ (dans l'ordre, c'est-à-dire dans l'ordre des *decode* associés). Par exemple, pour $lg\ l = 2$, cette liste sera : $[first\ 1; succ\ (first\ 0)]$. Puis on applique *fct l* à tous les éléments de cette liste. Nous allons utiliser la fonction *makeListFin* présentée à la Remarque 3.3. On peut démontrer simplement le résultat suivant :

Lemme 3.24. $\forall n\ m\ t\ (h : m < n), nth\ m\ (makeListFin\ n)\ t = code\ h$

où nth n l t désigne la fonction qui renvoie le $n^{ème}$ *élément de la liste l et t si cet élément n'existe pas.*

La fonction *ilist2list* s'écrit maintenant naturellement :

Définition 3.13 (*ilist2list*)**.** $ilist2list\ T\ l :\ list\ T\ :=\ map\ (fct\ l)\ (makeListFin\ (lg\ l))$

Pour valider cette définition, et avant de prouver une bijection avec les listes, on peut démontrer son adéquation avec *ilist_rel* :

Lemme 3.25. $\forall l_1\ l_2, ilist\_rel_{eq}\ l_1\ l_2 \Leftrightarrow ilist2list\ l_1 = ilist2list\ l_2$

Démonstration. Preuve détaillée en page 168. □

Pour définir *list2ilist* nous procédons également en deux étapes. Premièrement nous définissons une fonction *list2FinT* qui nous donne le "$i^{ème}$" élément d'une liste (où i désigne un élément de $Fin\ n$). Puis nous utilisons cette fonction comme la partie fonctionnelle de l'*ilist* que nous voulons créer, sa taille étant la même que la liste initiale.

La fonction *list2FinT* prend en paramètre une liste l et un élément i de $Fin\ (length\ l)$ et elle renvoie le $i^{ème}$ élément de l.

Définition 3.14 (*list2FinT*)**.**

$$\begin{array}{lllll} list2FinT & l & : & Fin\ (length\ l) \to T \\ list2FinT & (t::l) & i\ := & nth\ (decode\ i)\ (t::l)\ t \end{array}$$

Le cas où $l = []$ *n'est pas montré ici parce qu'il est impossible (on aurait* $i : Fin\ 0$ *ce qui est faux).*

Remarque 3.18. *On a besoin d'une valeur par défaut pour nth (elle est purement formelle puisqu'on ne l'utilisera jamais). On utilise la valeur à laquelle on a directement accès : t.*

La fonction *list2FinT* est donc telle que les deux propriétés suivantes sont correctes :

Propriété 3.3. $\forall t\ q,\ list2FinT\ (t::q)\ (first\ (length\ q)) = t$

Idée de la preuve. La preuve est triviale d'après la définition de *list2FinT*.

Propriété 3.4. $\forall t\ q\ i,\ list2FinT\ (t::q)\ (succ\ i) = list2FinT\ q\ i$

Idée de la preuve. La preuve est une induction simple sur q.

Les deux lemmes suivants sont également vérifiés.

Lemme 3.26.
$$\forall (l : list\ T)\ (f : T \to U)\ (i : Fin\ (length\ (map\ f\ l))),$$
$$list2FinT\ (map\ f\ l)\ i = f\ (list2FinT\ l\ (conv_h\ i))$$

où h est une preuve de $length\ (map\ f\ l) = length\ l$*, ce qui est un résultat bien connu.*

Idée de la preuve. La preuve est une simple analyse de cas sur l et sur i, qui utilise des résultats sur le calcul du $n^{\text{ème}}$ élément de *map f l*. Elle n'est pas détaillée ici.

Lemme 3.27.
$$\forall (l :\ ilist\ T)\ (i :\ Fin\ (length\ (makeListFin\ (lg\ l)))),$$
$$list2FinT\ (makeListFin\ (lg\ l))\ i = conv_h\ i$$

où h est une preuve de $length\ (makeListFin\ (lg\ l)) = lg\ l$*, ce qui est une instance du Lemme 3.2.*

Idée de la preuve. La preuve est ici aussi la combinaison d'analyses de cas sur l et sur i et de théorèmes simples sur les listes. Elle est simple et n'est pas détaillée ici.

Il est maintenant facile de définir *list2ilist* à partir de *list2FinT* :

Définition 3.15 (*list2ilist*)**.** $list2ilist\ T\ l :\ ilist\ T\ :=\ \langle length\ l,\ list2FinT\ l\rangle$

On peut exprimer la relation qui existe entre une liste et sa conversion en *ilist* de la façon suivante :

Lemme 3.28. $\forall l\ i\ t, nth\ (decode\ i)\ l\ t = fct\ (list2ilist\ l)\ i$

Démonstration. Preuve détaillée en page 169. □

On fait de même pour *ilist2list* :

Lemme 3.29. $\forall l\ n\ t\ (h : n < lg\ l), nth\ n\ (ilist2list\ l)\ t = fct\ l\ (code\ h)$

Démonstration. Preuve détaillée en page 170. □

Pour montrer qu'il y a une bijection entre *ilist* et *list*, on doit montrer que les compositions $ilist2list \circ list2ilist$ et $list2ilist \circ ilist2list$ sont extensionellement égales à l'identité, c'est-à-dire, point à point et par rapport à *ilist_rel* quand on compare des éléments de *ilist*.

Nous allons d'abord définir deux lemmes auxiliaires nécessaires pour la suite. Les preuves de ces deux lemmes sont triviales et ne sont pas détaillées ici.

Lemme 3.30. $\forall T\ l,\ lg\ (list2ilist\ (ilist2list\ l))\ =\ lg\ l$.

Lemme 3.31. $\forall T\ l,\ length\ (ilist2list\ (list2ilist\ l))\ =\ length\ l$.

On va maintenant prouver qu'il y a une bijection entre *ilist* et *list*.

Théorème 3.1 ($list2ilist \circ ilist2list = id$)**.** $\forall T\ l, ilist\_rel_{eq}\ l\ (list2ilist\ (ilist2list\ l))$.

Démonstration. En utilisant la Définition 3.11 on a

$$ilist\_rel_{eq}\ l\ (list2ilist\ (ilist2list\ l)) \Leftrightarrow \exists h : lg\ l = lg\ (list2ilist\ (ilist2list\ l))\ \forall i : Fin\ (lg\ l),\ fct\ l\ i = fct\ (list2ilist\ (ilist2list\ l))\ (conv_h\ i)$$

On obtient h grâce au Lemme 3.30. On doit donc seulement prouver que :

$$\forall i : Fin\ (lg\ l),\ fct\ l\ i = fct\ (list2ilist\ (ilist2list\ l))\ (conv_h\ i)$$

On a, pour tout i,

$$\begin{array}{lll} & fct\ (list2ilist\ (ilist2list\ l))\ (conv_h\ i) & \\ = & fct\ \langle length\ (ilist2list\ l),\ list2FinT\ (ilist2list\ l)\rangle\ (conv_h\ i) & \text{(d'après Définition 3.15)} \\ = & list2FinT\ (map\ (fct\ l)\ (makeListFin\ (lg\ l)))\ (conv_h\ i) & \text{(d'après Définition 3.13)} \\ = & fct\ l\ (list2FinT\ (makeListFin\ (lg\ l))\ (conv_{h'}\ (conv_h\ i)) & \text{(d'après Lemme 3.26)} \\ = & fct\ l\ (conv_{h''}\ (conv_{h'}\ (conv_h\ i))) & \text{(d'après Lemme 3.27)} \end{array}$$

avec h' preuve de $length\ (map\ (fct\ l)\ (makeListFin\ (lg\ l))) = length\ (makeListFin\ (lg\ l))$ et h'' preuve de $length\ (makeListFin\ (lg\ l)) = lg\ l$

Grâce à la Propriété 3.2, on obtient $(conv_{h''}\ (conv_{h'}\ (conv_h\ i))) =_{Fin} i$. De plus, $conv_{h''}\ (conv_{h'}\ (conv_h\ i))$ est de type $Fin\ (lg\ l)$, comme i. On peut donc appliquer le lemme d'injectivité de *decode* et on obtient : $conv_{h''}\ (conv_{h'}\ (conv_h\ i)) = i$.

D'où, finalement : $fct\ l\ i = fct\ (list2ilist\ (ilist2list\ l))\ (conv_h\ i)$

□

Remarque 3.19. *Évidemment, le Théorème 3.1 est aussi valable pour toute autre relation réflexive R puisque ilist_rel est monotone par rapport aux relations (Lemme 3.22).*

Théorème 3.2 ($ilist2list \circ list2ilist = id$)**.** $\forall T\ l, l = ilist2list\ (list2ilist\ l)$.

Démonstration (par induction sur l).

[Cas $[]$**]** Le résultat est obtenu en appliquant les définitions de *ilist2list* et de *list2ilist*.

[Cas $t::q$**]** L'hypothèse d'induction IH est $q = ilist2list\ (list2ilist\ q)$.
On introduit d'abord la propriété classique suivante sur *map* :

Propriété 3.5 (Fonctorialité de *map*)**.** $\forall l\ f\ g,\ map\ f\ (map\ g\ l) = map\ (f \circ g)\ l$

ilist2list (*list2ilist* (*t*::*q*)) se réduit de la façon suivante :

$$
\begin{array}{lll}
& \textit{ilist2list}\ (\textit{list2ilist}\ (t::q) & \\
= & \textit{ilist2list}\langle \textit{length}\ q + 1,\ \textit{list2FinT}\ (t::q)\rangle & \text{(d'après Définition 3.15)} \\
= & \textit{map}\ (\textit{list2FinT}\ (t::q))\ (\textit{makeListFin}\ (\textit{length}\ q + 1)) & \text{(d'après Définition 3.13)} \\
= & \textit{map}\ (\textit{list2FinT}\ (t::q)) & \text{(d'après Définition 3.2)} \\
& (\textit{first}\ (\textit{length}\ q)::\textit{map succ}\ (\textit{makeListFin}\ (\textit{length}\ q))) & \\
= & \textit{list2FinT}(t::q)\ (\textit{first}\ (\textit{length}\ q)):: & \text{(d'après Propriété 3.5)} \\
& \textit{map}\ ((\textit{list2FinT}\ (t::q)) \circ \textit{succ})\ (\textit{makeListFin}\ (\textit{length}\ q)) & \\
= & t::\textit{map}\ (\textit{list2FinT}\ q)\ (\textit{makeListFin}\ (\textit{length}\ q)) & \text{(d'après Propriétés 3.3 et 3.4} \\
& & \text{et l'extensionnalité de } \textit{map}) \\
= & t::\textit{ilist2list}\ \langle \textit{length}\ q,\ \textit{list2FinT}\ q\rangle & \text{(d'après Définition 3.13)} \\
= & t::\textit{ilist2list}\ (\textit{list2ilist}\ q) & \text{(d'après Définition 3.15)} \\
= & t::q & \text{(d'après } IH\text{)}
\end{array}
$$

□

Ceci prouve qu'il y a bien une bijection entre *list* et *ilist*, et cela valide notre définition de *ilist*.

3.2.4 Définition de fonctions sur *ilist* et quantification universelle

Comme on a une bijection entre *ilist* et *list*, on peut redéfinir facilement sur *ilist* n'importe quelle fonction f qui prend une liste en paramètre et/ou qui renvoie une liste. En particulier, cela signifie que toutes les fonctions usuelles (et les fonctions d'ordre supérieur) sur *list* ont leur équivalent sur *ilist*. Par exemple, la fonction classique *filter* sur *list*, peut se traduire sur *ilist* de la façon suivante (avec P prédicat sur les type des éléments) :

Définition 3.16. $\textit{ifilter}\ P\ l := \textit{list2ilist}\ (\textit{filter}\ P\ (\textit{ilist2list}\ l))$

Et général, toute fonction $f : \textit{list}\ T \to \textit{list}\ T$ peut se traduire dans *ilist* en une fonction f' de type $\textit{ilist}\ T \to \textit{ilist}\ T$. La fonction f' se défini comme suit :

$$f' := \textit{list2ilist} \circ f \circ \textit{ilist2list}$$

Cependant, ceci ne peut pas nous servir dans le cas où l'on a besoin de contourner une condition de garde (en général, dans ce cas là f est un appel corécursif). En effet, au lieu de "sortir" f pour qu'il ne soit plus "sous" une autre fonction, on le plonge plus profondément encore sous une fonction supplémentaire : *list2ilist*. Il faut donc redéfinir précisément chaque fonction "problématique".

3.2.4.1 *imap*

En particulier, dans la suite, nous aurons besoin d'un équivalent à la fonction *map*, et la conversion présentée précédemment ne peut pas nous satisfaire (à cause des problèmes de garde évoqués). Nous devons donc la redéfinir complètement. En réalité, cela est assez simple à réaliser puisque la partie de la *ilist* qui est affectée par le *map* est la partie fonctionnelle (*ilistn*). En y regardant de plus près, on voit que la fonction *imap* n'est guère plus qu'une composition de fonctions. On doit donc composer la partie fonctionnelle de la *ilist* avec la fonction à appliquer, et recréer la *ilist*. La taille d'une liste n'étant pas modifiée par l'application de la fonction *map*, il en va de même pour *imap*. On a donc :

Définition 3.17 (*imap*). $imap\ f\ l := \langle lg\ l,\ f \circ (fct\ l)\rangle$

Ici, la fonction f (on a dit précédemment qu'il pourrait s'agir d'un appel corécursif) est directement sous le constructeur $\langle \ldots, \ldots \rangle$. Cela satisfait donc la condition de garde (le cas échéant). On voit que l'utilisation des espaces des fonctions est considéré comme étant moins critique dans Coq que l'utilisation de types inductifs parce qu'ils sont plus primitifs. Ils font même partie du cadre logique. Ceci ne pourrait pas être fait sur des listes puisqu'elles sont définies inductivement et les fonctions qui les manipulent doivent l'être aussi. La récursion est le seul moyen de définir *map* sur les listes. Toutes ces fonctions d'ordre supérieur ajoutent une couche entre le constructeur et la fonction donnée en paramètre. Dans le cas où cette fonction est un appel corécursif, cela peut créer un conflit avec la condition de garde. Comme la fonction *imap* n'est pas définie récursivement, il n'y a pas de couche ajoutée et, comme on l'a dit, la condition de garde est satisfaite si besoin.

Le lemme suivant devient vrai par définition, avec la définition de *imap* donnée précédemment :

Lemme 3.32. $\forall (f : U \to T)\ (l : ilist\ U)(i : Fin\ (lg\ i)), fct\ (imap\ f\ l)\ i = f\ (fct\ l\ i)$

3.2.4.2 *iappend*

Une autre fonction qui nous sera utile par la suite est la fonction de concaténation. On peut la définir en utilisant les conversions de la façon suivante :

$$iappend\ (l_1\ l_2 : ilist\ T) := list2ilist\ ((ilist2list\ l_1)@ilist2list\ l_2)$$

Mais on peut aussi la définir directement. Pour cela, on doit savoir pour chaque élément de *Fin* s'il pointe vers un élément qui appartenait initialement à l_1 ou à l_2, le convertir au type $Fin\ (lg\ l_1)$ ou $Fin\ (lg\ l_2)$ et appliquer la fonction correspondante, comme le suggère la Figure 3.2.

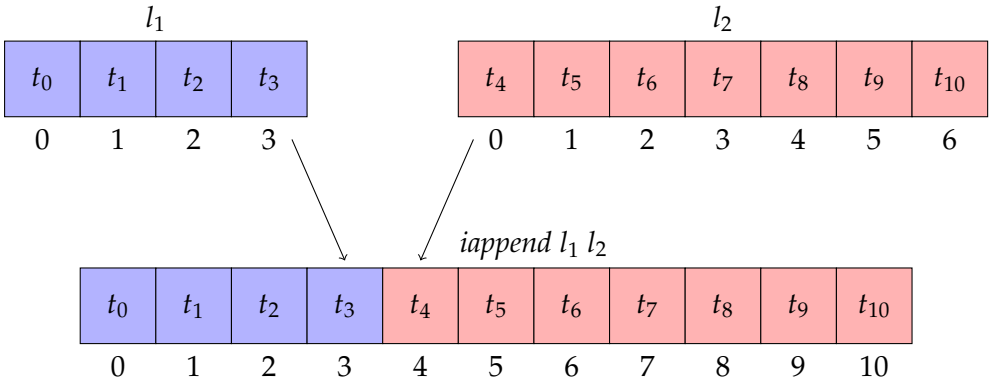


Figure 3.2 — Représentation de la concaténation de deux *ilist* et de la conversion des indices (représentés par des entiers pour simplifier)

Nous allons donc tout d'abord donner la fonction qui convertit un élément de type $Fin\ (n_1 + n_2)$ au type $Fin\ n_2$ (on appelle *rightFin* cette fonction) et l'objectif est que :

$$fct\ (iappend\ l_1\ l_2)\ i = l_2\ (rightFin\ i) \text{ si } lg\ l_1 \leq decode\ i$$

Définition 3.18. $rightFin\ (n_1\ n_2 : \mathbb{N})\ (i : Fin\ (n_1 + n_2))\ (h : n_1 \leq decode\ i) : Fin\ n_2 := code\ h'$

où h' est une preuve de $decode\ i - n_1 < n_2$ déduite du Lemme 3.5

On peut maintenant définir *iappend* en se servant de *rightFin*. On va commencer par définir sa partie fonctionnelle (*ilistn*) :

Définition 3.19.

$$\begin{array}{l} iappendn\ (l_1\ l_2 : ilist\ T) : ilistn\ T\ (lg\ l_1 + lg\ l_2) \\ iappendn\ l_1\ l_2\ i \quad := \quad \begin{cases} fct\ l_1\ (code\ h) & si\ h : decode\ i < lg\ l_1 \\ fct\ l_2\ (rightFin\ i\ h) & si\ h : lg\ l_1 \leq decode\ i \end{cases} \end{array}$$

Remarque 3.20. *La spécificité ici est que les cas entrent comme hypothèse h dans les branches de la définition.*

Et la définition de *iappend* est :

Définition 3.20.

$$\begin{array}{l} iappend : ilist\ T \to ilist\ T \to ilist\ T \\ iappend\ l_1\ l_2 := \langle lg\ l_1 + lg\ l_2, iappendn\ l_1\ l_2 \rangle \end{array}$$

Les propriétés suivantes sont alors naturelles sur *iappend* :

Propriété 3.6.

$$\forall l_1\ l_2, lg\ (iappend\ l_1\ l_2) = lg\ l_1 + lg\ l_2 \tag{3.6.1}$$

$$\forall l_1\ l_2\ i\ (h : decode\ i < lg\ l_1), fct\ (iappend\ l_1\ l_2)\ i = fct\ l_1\ (code\ h) \tag{3.6.2}$$

$$\forall l_1\ l_2\ i\ (h : lg\ l_1 \leq decode\ (conv_{h'}\ i)), fct(iappend\ l_1\ l_2)\ i = fct\ l_2\ (rightFin\ (conv_{h'}\ i)\ h)$$
$$avec\ h'\ de\ type\ lg\ (iappend\ l_1\ l_2) = lg\ l_1 + lg\ l_2\ déduit\ de\ la\ Propriété\ 3.6.1 \tag{3.6.3}$$

On peut également démontrer la compatibilité de notre définition avec la définition standard sur les listes (en utilisant la solution proposée au début de cette section) :

Lemme 3.33. $\forall l_1\ l_2, ilist\_rel_{eq}\ (iappend\ l_1\ l_2)\ (list2ilist\ ((ilist2list\ l_1)@ilist2list\ l_2))$

Idée de la preuve. On va utiliser ici la définition de *ilist_rel* et donc comparer point à point les deux *ilist*. Nous utiliserons les Propriétés 3.6.2 et 3.6.3 pour réduire la partie gauche, et des résultats bien connus sur *nth* pour la partie droite. Pour utiliser ces différents résultats, nous devrons comparer $lg\ l_1$ et *decode i*.

Démonstration. Preuve détaillée en page 170. □

Lemme 3.34. $\forall l_1\ l_2, ilist2list\ (iappend\ l_1\ l_2) = (ilist2list\ l_1)@(ilist2list\ l_2)$

Idée de la preuve. Le problème ici est que de façon sous-jacente, nous comparons des *ilist*. Et donc nous devons faire une comparaison point à point. Or les listes ne se manipulent pas de cette façon habituellement. On démontre et on utilise donc le résultat suivant, qui fait le lien entre égalité de deux listes et égalité de leur éléments :

Lemme 3.35. $\forall l_1\ l_2, length\ l_1 = length\ l_2 \wedge (\forall n\ d, nth\ n\ l_1\ d = nth\ n\ l_2\ d) \Rightarrow l_1 = l_2$

Démonstration. La preuve se fait par induction sur l_1. □

On en déduit immédiatement le corollaire suivant qui nous permettra de simplifier nos preuves (on analysera moins de cas) :

Corollaire 3.36.

$$\forall l_1\ l_2, length\ l_1 = length\ l_2 \wedge (\forall n\ d, n < length\ l_1 \rightarrow nth\ n\ l_1\ d = nth\ n\ l_2\ d) \Rightarrow l_1 = l_2$$

Le reste de la preuve du Lemme 3.34 se fait en comparant n (la position dans la liste des éléments comparés) et la longueur de l_1 (afin de pouvoir appliquer les Propriétés 3.6.2 et 3.6.3 et des lemmes bien connus sur *nth*).

Démonstration. Preuve détaillée en page 171. □

3.2.4.3 Quantification universelle

On aura également besoin dans la suite d'une propriété sur *ilist* qui exprime qu'un prédicat P est vrai pour tous les éléments d'une *ilist*. On l'appelle *iall* (c'est l'équivalent de la fonction Caml *for_all*) et elle est définie comme suit :

Définition 3.21 (*iall*)**.** $iall\ T\ P\ l : Prop := \forall i, P\ (fct\ l\ i)$

3.2.5 Manipulation de *ilist* à la manière des listes

Comme on l'a expliqué au début de cette section, l'objectif était d'obtenir une structure équivalente aux listes mais qui ne soit pas inductive. Nous avons montré dans la Section 3.2.3 que *ilist* remplissait bien cette fonction. Cependant, on est assez habitué à la manipulation des listes inductives et elle est souvent bien pratique. On aimerait donc, dans certains cas, pouvoir manipuler les *ilist* comme des listes. Nous allons pour cela présenter ici quelques outils qui le permettent. Il convient néanmoins de préciser que la structure des *ilist* n'est pas toujours très adaptée à une manipulation à la façon des listes. En effet, ces dernières se manipulent typiquement (et de façon native) sous forme de liste vide ou tête/queue, alors qu'obtenir le $n^{\text{ième}}$ élément est une opération assez coûteuse et non primitive. A l'opposé, il est primitif d'obtenir le $n^{\text{ième}}$ élément d'une *ilist* mais il est plus compliqué d'en obtenir seulement la "queue". Il n'est donc pas toujours judicieux d'utiliser ces opérations, lorsqu'on peut s'en passer.

Les toutes premières choses à définir pour pouvoir manipuler les *ilist* comme les listes sont les équivalents pour les deux constructeurs de *list*, *nil* ([]) et *cons* (::). On les nommera respectivement *inil* et *icons*.

Pour définir *inil*, on a besoin d'un élément du type $ilistn\ T\ 0$, c'est-à-dire une fonction de type $Fin\ 0 \rightarrow T$. Comme $Fin\ 0$ est vide, toutes les fonctions du type $ilistn\ T\ 0$ sont extensionnellement équivalentes (c'est-à-dire, point à point) et ces types sont habités pour tout T. On en appelle une $iniln_0$. On définit *inil* de la façon suivante :

Définition 3.22 (*inil*)**.** $inil\ T := \langle 0,\ iniln_0\ T \rangle$

Pour plus de clarté, nous allons d'abord définir la partie fonctionnelle de *icons* (*iconsn*). On doit ici "décaler" les indices de la *ilist* pour ajouter l'élément en tête, comme montré dans la Figure 3.3.

Définition 3.23 (*iconsn*)**.**

$$\begin{aligned} &iconsn : \forall T\ n, T \rightarrow ilistn\ T\ n \rightarrow ilistn\ T\ (n+1) \\ &iconsn\ T\ n\ t\ ln\ (first\ n) := t \\ &iconsn\ T\ n\ t\ ln\ (succ\ i') := ln\ i' \end{aligned}$$

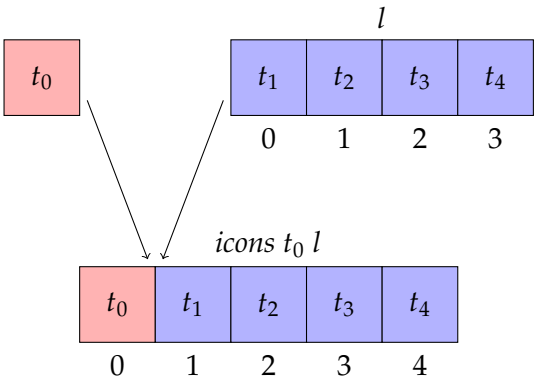


Figure 3.3 — Représentation de *icons*

Et la définition de *icons* suit naturellement :

Définition 3.24 (*icons*)**.** $icons\ T\ t\ l := \langle lg\ l + 1,\ iconsn\ t\ (fct\ l)) \rangle$

On peut montrer facilement les cinq résultats suivants, qui valident nos définitions.

Lemme 3.37. $ilist\_rel_{eq}\ (inil\ T)\ (list2ilist\ [])$

Lemme 3.38. $\forall t\ l, ilist\_rel_{eq}\ (icons\ t\ l)\ (list2ilist\ (t::(ilist2list\ l))$

Lemme 3.39. $ilist2list\ (inil\ T) = []$

Lemme 3.40. $ilist2list\ (icons\ t\ (list2ilist\ l)) = t::l$

Lemme 3.41. $ilist2list\ (icons\ t\ l) = t::(ilist2list\ l)$

Comme nous l'avons dit plus haut, les notions de base sur les listes sont les notions de tête et de queue (fonctions *head* et *tail*). Nous avons déjà utilisé implicitement leurs équivalents sur *ilist* dans la démonstration du Lemme 3.23. Nous allons les définir formellement ici. On les appellera *ihead* et *itail*.

Définition 3.25 (*ihead*)**.**

$$\begin{aligned} &ihead : \forall T, ilist\ T \to T \to T \\ &ihead\ T\ \langle 0,\ ln \rangle\ t := t \\ &ihead\ T\ \langle n+1,\ ln \rangle\ t := ln\ (first\ n) \end{aligned}$$

où t représente l'élément par défaut renvoyé par ihead dans le cas où la ilist donnée en paramètre est vide.

Définition 3.26 (*itail*)**.**

$$\begin{aligned} &itail : \forall T, ilist\ T \to ilist\ T \\ &itail\ T\ \langle 0,\ ln \rangle := inil\ T \\ &itail\ T\ \langle n+1,\ ln \rangle := \langle n,\ ln \circ succ \rangle \end{aligned}$$

Dans le cas où la *ilist* donnée en paramètre est vide, *itail* renvoie une *ilist* vide (*inil*).

Pour valider ces définitions, on peut facilement prouver que :

$$\forall T\ l\ t, lg\ l > 0 \to ilist\_rel_{eq}\ l\ (icons\ (ihead\ l\ t)\ (itail\ l))$$

Remarque 3.21 (Notations)**.** *Dans la suite, nous utiliserons une notation proche de celle des listes pour les éléments de ilist. Nous écrirons en particulier* $[\![]\!]$ *pour inil et* $[\![x; y; z; ...]\!]$ *pour des applications successives de icons sur inil.*

3.2.6 Parties gauche et droite d'une *ilist*

Nous aurons besoin dans la suite de pouvoir prendre la partie gauche ou la partie droite d'une *ilist* (par rapport à un indice donné), comme présenté dans la Figure 3.4.

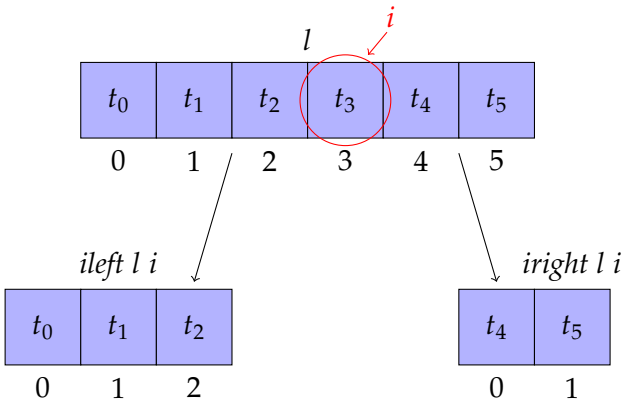


Figure 3.4 — Parties gauche et droite d'une *ilist* par rapport à l'indice i tel que $decode\ i = 3$

On les définit de la façon suivante (toujours en définissant d'abord leur partie fonctionnelle puis la fonction globale) :

Définition 3.27.

$$\begin{aligned} &ileftn\ (l : ilist\ T\ )\ (i : Fin\ (lg\ l)) : ilistn\ T\ (decode\ i) \\ &ileftn\ l\ i\ (i' : Fin\ (decode\ i)) := fct\ l\ (code\ h) \end{aligned}$$

avec $h : decode\ i' < lg\ l$ *déduit par transitivité du Lemme 3.5 appliqué à* i' *et à* i.

Définition 3.28.

$$\begin{aligned} &ileft(l : ilist\ T) : Fin\ (lg\ l) \to ilist\ T \\ &ileft\ l\ i := \langle decode\ i, ileftn\ l\ i\rangle \end{aligned}$$

Définition 3.29.

$$\begin{aligned} &irightn\ (l : ilist\ T\ )\ (i : Fin\ (lg\ l)) : ilistn\ T\ (lg\ l - (decode\ i + 1)) \\ &irightn\ l\ i\ (i' : Fin\ (lg\ l - (decode\ i + 1))) := fct\ l\ (code\ h) \end{aligned}$$

avec $h : decode\ i + 1 + decode\ i' < lg\ l$ *déduit du Lemme 3.5 appliqué à* i' *et de plusieurs opérations arithmétiques.*

Définition 3.30.

$$\begin{aligned} &iright(l : ilist\ T) : Fin\ (lg\ l) \to ilist\ T \\ &iright\ l\ i := \langle lg\ l - (decode\ i + 1), irightn\ l\ i\rangle \end{aligned}$$

Les propriétés suivantes sont immédiates sur les longueurs de *ileft* et *iright* :

Propriété 3.7.

$$\forall l\ i, lg\ (ileft\ l\ i) = decode\ i \tag{3.7.1}$$
$$\forall l\ i, lg\ (iright\ l\ i) = lg\ l - (decode\ i + 1) \tag{3.7.2}$$
$$\forall l\ i, lg\ (iright\ l\ i) + lg\ (ileft\ l\ i) + 1 = lg\ l \tag{3.7.3}$$

On peut également démontrer le lemme suivant qui valide nos définitions :

Lemme 3.42. $\forall l\ i, ilist2list\ (ileft\ l\ i)@(fct\ l\ i)::ilist2list(iright\ l\ i) = ilist2list\ l$

Idée de la preuve. Ici encore, comme de façon implicite on compare des *ilist*, on va utiliser le Corollaire 3.36 pour faire la preuve. Ensuite, on comparera les valeurs de n (la position des éléments comparés) et de *decode i* afin de savoir si l'élément considéré est "à gauche" ou "à droite" de i.

Démonstration. Preuve détaillée en page 172. □

On en déduit le corollaire suivant :

Corollaire 3.43. $\forall l\ i, ilist\_rel_{eq}\ (iappend\ (ileft\ l\ i)\ (icons\ (fct\ l\ i)\ (iright\ l\ i)))\ l$

Démonstration. Preuve détaillée en page 173. □

3.3 Multiplicités dans *ilist*

Dans la suite, il nous sera utile de pouvoir fixer finement la taille d'une *ilist*. De plus, la représentation que nous avons choisie s'y prête bien puisqu'on a accès directement à cette taille. Ce que nous voulons faire, c'est pouvoir imposer que la taille soit entre deux bornes données.

Pour mieux comprendre l'idée et montrer une utilité possible, examinons un exemple.

3.3.1 Exemple d'utilisation des multiplicités

L'exemple que nous allons présenter se base sur la représentation des métamodèles. Considérons l'exemple de la Figure 3.5.

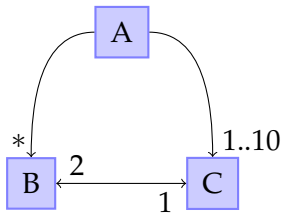


Figure 3.5 — Exemple d'un métamodèle avec multiplicités

Pour représenter cela avec les outils que nous avons actuellement à notre disposition, on définirait A, B et C simultanément comme suit :

$$\frac{l_b : ilist\ B \qquad l_c : ilist\ C}{mk\_A\ l_b\ l_c : A} \qquad \frac{c : C}{mk\_B\ c : B} \qquad \frac{b_1 : B \qquad b_2 : B}{mk\_C\ b_1\ b_2 : C}$$

On voit que même cet exemple extrêmement simple soulève plusieurs problèmes :

– Pour représenter la multiplicité $1 \ldots 10$ de la branche $A \to C$, on pourrait énumérer toutes les possibilités en augmentant le nombre de paramètres. Mais cela devient vite lourd pour des nombres un peu gros et impossible pour une borne indéterminée. On

ne voit alors pas d'autre possibilité que d'utiliser *ilist* (puisque les listes nous poseraient des problèmes de mélange d'induction et de coinduction). Cependant, on perd ici les bornes de la condition de multiplicité et donc on perd de l'information (au moins un élément, au maximum dix). Notre représentation est moins précise que le modèle initial (ce n'est donc pas un raffinement, ce qui peut être très problématique).
- La solution proposée n'est pas homogène. On a deux façons différentes de représenter une branche avec multiplicité :
  - on utilise une *ilist* pour représenter une multiplicité variable (par exemple $*$ ou $1 \dots 10$)
  - une séquence de $T \to T \to \dots \to T$ pour une multiplicité fixée (par exemple 1 ou 2)

On voit donc l'intérêt d'une extension de *ilist* qui permettrait de résoudre ces différents problèmes. C'est ce que nous allons présenter maintenant.

3.3.2 Implémentation des multiplicités

Premièrement, nous allons avoir besoin d'une propriété (appelons-la *PropMult*) qui dit qu'un nombre est entre les deux bornes de la condition de multiplicité. La borne inférieure (appelons-la *inf*) existe toujours. Elle peut valoir 0 mais elle a toujours une valeur. Elle est donc de type $\mathbb{N}$. Au contraire, la borne supérieure (on l'appelle *sup*) peut ne pas exister (multiplicité "$*$"). Elle doit donc être de type *option* $\mathbb{N}$.

Remarque 3.22 (*option*)**.** *Il s'agit du type bien connu dans les langages fonctionnels défini ainsi :*

$$\frac{}{None : option\ T} \qquad \frac{t : T}{Some\ t : option\ T}$$

On exprime la propriété *PropMult* comme suit :

Définition 3.31 (*PropMult*)**.**

$$\forall inf\ sup\ k, PropMult\ inf\ sup\ k \Leftrightarrow \begin{cases} k \geq inf \wedge k \leq s & si\ sup = Some\ s \\ k \geq inf & si\ sup = None \end{cases}$$

A l'aide de cette propriété, on peut raffiner notre définition de *ilist* afin de prendre en compte les multiplicités et de ne garder que celles dont la taille satisfait *PropMult*. On raffine d'abord *ilistn* :

Définition 3.32. $ilistnMult\ T\ inf\ sup\ n := \{ln : ilistn\ T\ n \mid PropMult\ inf\ sup\ n\}$

Remarque 3.23. *Les éléments de ilistnMult sont en fait des couples formés d'un élément de type ilistn et d'une preuve de PropMult inf sup n. Le type est donc vide si PropMult inf sup n n'est pas vérifié.*

On utilise maintenant *ilistnMult* pour définir *ilistMult* (les *ilist* avec multiplicité). C'est exactement la même chose que pour *ilist*, mais avec *ilistnMult* au lieu de *ilistn*.

Définition 3.33. $ilistMult\ T\ inf\ sup := \Sigma n : nat.\ ilistnMult\ T\ inf\ sup\ n$

Comme pour *ilist*, on définit les fonctions de projection *fctM* et *lgM*. On peut aussi définir une relation sur *ilistMult* de la même manière que sur *ilist*.

Définition 3.34 (*iM_rel*)**.**

$$\forall n\ (m : \mathit{option}\ \mathbb{N})\ (l_1\ l_2 : \mathit{ilistMult}\ T\ n\ m), \mathit{iM\_rel}\ R\ l_1\ l_2 \Leftrightarrow$$
$$\exists h : \mathit{lgM}\ l_1 = \mathit{lgM}\ l_2,\ \forall i : \mathit{Fin}\ (\mathit{lgM}\ l_1), R\ (\mathit{fctM}\ l_1\ i)\ (\mathit{fctM}\ l_2\ (\mathit{conv}_h\ i))$$

On montre comme pour *ilist_rel* que *iM_rel* préserve l'équivalence.

On peut également montrer qu'il y a une bijection entre *ilistMult T* 0 *None* et *list*. Comme pour *ilist*, on définit deux fonctions (*ilistMult2list* et *list2ilistMult*) et on prouve que leurs compositions sont extensionnellement égales à l'identité. Comme ce développement est exactement similaire à celui sur *ilist*, on ne le présente pas ici plus précisément.

Remarque 3.24. *Les multiplicités* 0 *et None s'expliquent par le fait qu'une liste peut être vide (donc inf* $= 0$*) ou qu'elle peut avoir un nombre fini mais non borné d'éléments (c'est-à-dire, multiplicité "*$*$*" et donc sup* $=$ *None)*

En combinant le résultat précédent et l'équivalence entre *ilist* et *list* (Théorèmes 3.1 et 3.2) on peut démontrer le lemme suivant (ici, $=$ correspond à l'égalité extensionnelle, c'est-à-dire point à point, des fonctions) :

Lemme 3.44 (Bijection entre *ilist* et *ilistMult*)**.**

$$\exists (f_1 : \mathit{ilistMult}\ T\ 0\ \mathit{None} \to \mathit{ilist}\ T)(f_2 : \mathit{ilist}\ T \to \mathit{ilistMult}\ T\ 0\ \mathit{None}), f_1 \circ f_2 = f_2 \circ f_1 = \mathit{id}$$

Démonstration. Preuve détaillée en page 173. □

Remarque 3.25. *Un résultat important à retenir c'est que toutes les définitions écrites avec ilist peuvent être écrites de façon équivalente avec ilistMult T* 0 *None.*

Si on revient à l'exemple de la Figure 3.5, on pourrait maintenant définir A, B et C de la façon suivante à l'aide de *ilistMult* :

$$\frac{l_b : \mathit{ilistMult}\ B\ 0\ \mathit{None} \qquad l_c : \mathit{ilistMult}\ C\ 1\ (\mathit{Some}\ 10)}{\mathit{mk\_A}\ l_b\ l_c : A}$$

$$\frac{c : \mathit{ilistMult}\ C\ 1\ (\mathit{Some}\ 1)}{\mathit{mk\_B}\ c : B} \qquad \frac{b : \mathit{ilistMult}\ B\ 2\ (\mathit{Some}\ 2)}{\mathit{mk\_C}\ b : C}$$

Ces définitions sont maintenant homogènes et complètes : aucune information n'est perdue.

4 Permutations

DANS la suite (voir Chapitre 6), nous aurons besoin d'une relation plus souple que *ilist_rel* sur *ilist*. En effet, nous voudrions nous rapprocher de la notion de multi-ensembles. Or, telle quelle, la représentation que nous avons, munie de la relation *ilist_rel*, est ordonnée. En effet, pour être équivalentes, deux *ilist* doivent avoir les mêmes éléments *à la même place*. On voudrait maintenant une relation qui permette d'exprimer que deux *ilist* sont équivalentes si elles ont simplement les mêmes éléments (en même quantité). C'est-à-dire si les multi-ensembles résultant des deux *ilist* sont identiques. Ou encore, si les deux *ilist* sont des permutations l'une de l'autre.

Remarque 4.1. *On s'éloigne ici de la vue conteneur, puisqu'il est bien connu qu'on ne peut pas les utiliser directement pour représenter les ensembles et les multi-ensembles. En revanche, ces derniers pourraient être représentés par des types quotients ("quotient types") [1], largement inexplorés.*

Dans un premier temps, nous allons voir de quelle façon les permutations sont traitées sur les listes dans Coq. Nous introduirons ensuite les différentes méthodes que nous proposons pour les *ilist*, et nous les étudierons en détail et les comparerons.

4.1 Permutations sur les listes

Le problème de définir une relation exprimant les permutations sur les listes est assez classique. Dans la librairie standard de Coq, on trouve deux définitions distinctes [32].

4.1.1 Permutations avec multi-ensembles

La première définition correspond à l'explication donnée précédemment : on extrait des multi-ensembles des listes et on les compare. Un multi-ensemble peut être vu comme une fonction qui, à chaque élément de son ensemble de définition, associe sa multiplicité dans l'ensemble. C'est de cette façon qu'ils sont définis dans Coq (voir [32]) :

Définition 4.1 (*multiset*, défini inductivement)**.**

$$\frac{f : T \to \mathbb{N}}{Bag\ f : multiset} \qquad \textit{où } T \textit{ est le type des éléments de l'ensemble.}$$

La relation de comparaison des multi-ensembles, *meq*, est définie grâce à la fonction *multiplicity*, qui renvoie la multiplicité d'un élément dans un multi-ensemble :

Définition 4.2 (*multiplicity* : $T \to multiset \to \mathbb{N}$)**.** $multiplicity\ t\ (Bag\ f) := f\ t$

Définition 4.3 (*meq*)**.** $\forall e_1\ e_2,\ meq\ e_1\ e_2 \Leftrightarrow \forall t : T, multiplicity\ t\ e1 = multiplicity\ t\ e2$

Deux multi-ensembles sont donc équivalents s'ils contiennent le même nombre de chaque élément.

Remarque 4.2. *Ce n'est pas directement visible ici, mais ces définitions sont paramétrées par une relation sur T qui doit être décidable.*

Enfin, nous aurons besoin dans la suite de trois outils supplémentaires. Nous ne détaillons pas leurs définitions ici. On appellera *EmptyBag* un *multiset* vide, *SingletonBag* un *multiset* à un seul élément et *munion* l'union entre deux *multiset* (pour chaque élément de T, les multiplicités sont ajoutées). Il est seulement important de noter que pour la définition de *SingletonBag* on a nécessairement besoin de la décidabilité sur la relation d'équivalence sur T (en effet, pour *SingletonBag t*, la fonction paramètre de *Bag* doit renvoyer 1 si l'argument de la fonction est équivalent à t et 0 sinon).

La dernière étape avant de pouvoir définir la notion de permutations avec les multi-ensembles est la transformation d'une liste en multi-ensemble. Ceci est fait grâce à la fonction *list_contents* :

Définition 4.4 (*list_contents*)**.**

$$\begin{array}{lcl} \mathit{list\_contents} & : & \mathit{list}\ T \to \mathit{multiset}\ T \\ \mathit{list\_contents}\ [] & := & \mathit{EmptyBag} \\ \mathit{list\_contents}\ (t :: l) & := & \mathit{munion}\ (\mathit{SingletonBag}\ t)\ (\mathit{list\_contents}\ l) \end{array}$$

Grâce à cette fonction, la notion de permutations utilisant les multi-ensembles est définie de la façon suivante dans [32] :

Définition 4.5 (*permutation*, permutations avec multi-ensembles)**.**

$$\forall l_1 l_2,\ \mathit{permutation}\ l_1\ l_2 \Leftrightarrow \mathit{meq}\ (\mathit{list\_contents}\ l_1)\ (\mathit{list\_contents}\ l_2)$$

La relation *permutation* est une relation d'équivalence.

4.1.2 Permutations inductives (voir [32])

La seconde définition est proche de la définition mathématique avec transposition. Globalement, il est dit qu'il existe quatre possibilités pour que deux listes soient des permutations l'une de l'autre :

- les deux listes sont vides
- si deux listes sont des permutations l'une de l'autre, alors les deux listes résultant de l'ajout du même élément en tête de chacune sont des permutations l'une de l'autre
- les deux listes résultant de l'ajout de deux éléments permutés en tête d'une liste sont des permutations l'une de l'autre
- il existe une troisième liste qui est une permutation de chacune des deux autres

Ceci est exprimé en Coq à partir de quatre constructeurs de type inductif de la façon suivante :

Définition 4.6 (*permutation'*, permutations inductives)**.**

$$\forall l_1\ l_2, \mathit{permutation'}\ l_1\ l_2 \quad \Leftrightarrow \quad \left\{ \begin{array}{ll} & l_1 = l_2 = [] \\ ou & \exists x\ l_1'\ l_2', l_1 = x :: l_1' \wedge l_2 = x :: l_2' \wedge \mathit{permutation'}\ l_1'\ l_2' \\ ou & \exists x\ y\ l, l_1 = x :: y :: l \wedge l_2 = y :: x :: l \\ ou & \mathit{existsl}, \mathit{permutation'}\ l_1\ l \wedge \mathit{permutation'}\ l\ l_2 \end{array} \right.$$

Cette relation est une relation d'équivalence. Dans [32], le résultat suivant est également démontré (dans le cas où la relation d'équivalence utilisée sur T dans *permutation* est l'égalité de Leibniz) :

Lemme 4.1. $\forall l_1\ l_2,\ permutation\ l_1\ l_2 \Leftrightarrow permutation'\ l_1\ l_2$

Cependant, comme on peut voir, dans les deux solutions qui sont proposées dans la librairie standard de Coq, on ne peut pas passer en paramètre une relation quelconque sur T (le type des éléments des listes) à la relation de permutations. Dans le premier cas, elle doit être décidable et dans le second c'est l'égalité de Leibniz qui est utilisée. Néanmoins, cette restriction est parfois trop forte et on aurait besoin d'une relation quelconque qui ne demande pas la décidabilité. C'est ce que nous allons voir maintenant.

4.1.3 Permutations inductives sans décidabilité

Dans [24], Contejean propose une définition des permutations sur les listes qui ne requiert pas la décidabilité de la relation sur T. Elle les définit ainsi :

Définition 4.7 (*permutation_indec*, permutations sans décidabilité)**.**

$$\forall R\ l_1\ l_2, permutation\_indec_R\ l_1\ l_2$$
$$\Leftrightarrow \quad \begin{cases} l_1 = l_2 = [] \\ ou \\ \exists a\ b\ l\ l'_1\ l'_2,\ l_1 = a :: l'_1 \wedge l_2 = l\ @\ (b :: l'_2) \wedge R\ a\ b \wedge permutation\_indec_R\ l'_1\ (l\ @\ l'_2) \end{cases}$$

Elle prouve que cette définition est en adéquation avec la définition mathématique et qu'elle conserve l'équivalence.

Comme on peut le voir, cette dernière définition est très simple et élégante, et présente l'avantage de ne pas nécessiter la décidabilité. Cependant, elle s'appuie fortement sur la structure des listes (avec les notions de tête, de queue et de concaténation).

Nous allons maintenant étudier les différentes méthodes que nous avons implémentées pour représenter les permutations directement sur les *ilist*.

4.2 Première méthode : comptage d'occurrences

Cette première proposition est assez proche de la solution sur les listes qui utilise les multi-ensembles. Cependant, l'idée ici n'est pas de créer un multi-ensemble mais simplement de compter le nombre d'occurrences de chaque élément de T dans chacune des *ilist*. Ce que l'on voudrait, c'est dire que deux *ilist* l_1 et l_2 sont des permutations l'une de l'autre si :

$$\forall t, card\ \{i : Fin\ (lg\ l_1) \mid R\ (fct\ l_1\ i)\ t\} = card\ \{i : Fin\ (lg\ l_2) \mid R\ (fct\ l_2\ i)\ t\}$$

Nous allons compter les occurrences "à la main". Nous appelons *nbocc* la fonction qui compte le nombre d'occurrences d'un élément dans une *ilist*. Pour implémenter *nbocc*, nous avons besoin de décider, pour chaque élément, s'il est équivalent ou non à celui que l'on cherche. Nous avons donc besoin de la décidabilité sur R, que l'on notera donc R_d. La procédure de décision est utilisée dans l'analyse de cas de la définition récursive suivante :

Définition 4.8 ($nbocc_{R_d}$)**.**

$$\begin{array}{lllll} nbocc_{R_d} & : & T \to ilist\ T \to nat & & \\ nbocc_{R_d}\ t\ \langle 0, & ln\rangle & := & 0 & \\ nbocc_{R_d}\ t\ \langle n{+}1, & ln\rangle & := & if\ (R_d\ t\ (ihead\langle n{+}1, ln\rangle)) & then\ nbocc_{R_d}\ t\ (itail\langle n{+}1, \quad ln\rangle) + 1 \\ & & & & else\ \ nbocc_{R_d}\ t\ (itail\langle n{+}1, \quad ln\rangle) \end{array}$$

En utilisant cette définition, nous pouvons maintenant décrire les permutations sur *ilist* :

Définition 4.9 (*iperm_occ*, permutations sur *ilist* avec décidabilité)**.**

$$\forall l_1\ l_2,\ iperm\_occ_{R_d}\ l_1\ l_2 \Leftrightarrow \forall t, nbocc_{R_d}\ t\ l_1 = nbocc_{R_d}\ t\ l_2$$

On peut facilement montrer que cette relation préserve l'équivalence. On peut aussi montrer que *ilist_rel* est plus fine que *iperm_occ* (dans le cas où R_d est une relation d'équivalence) :

Lemme 4.2 (*ilist_rel* plus fine que *iperm_occ*)**.** $\forall l_1\ l_2, ilist\_rel_{R_d}\ l_1\ l_2 \Rightarrow iperm\_occ_{R_d}\ l_1\ l_2$.

Idée de la preuve. On commence par prouver que les deux *ilist* ont la même longueur puis on raisonne par induction sur cette longueur. Dans le cas inductif, on analyse les différents cas possibles en utilisant la décidabilité de R_d.

Démonstration. Preuve détaillée en page 173. □

Cependant, comme nous l'avons dit, nous avons eu ici besoin de la décidabilité, et comme expliqué précédemment, ce critère peut dans certains cas être trop rigide. Nous devons donc réussir à nous en passer.

4.3 Deuxième méthode : définition inductive

Notre objectif ici est d'avoir une définition qui ne nécessite pas la décidabilité de la relation d'équivalence sur les éléments de T. Pour cela, au lieu de décrire ce qu'est une permutation, nous allons spécifier quand elle existe. Nous allons le faire constructivement, grâce à un processus de génération inductif. En considérant ce que l'on fait en sens inverse, le principe général est d'enlever récursivement des éléments équivalents deux à deux. Pour cela, nous avons besoin d'une fonction qui enlève un élément d'une *ilist*, l'idée étant de ne garder que les éléments "à gauche" et "à droite" de celui qu'on veut enlever.

4.3.1 Définitions et lemmes préliminaires

Avant de pouvoir définir la relation de permutations elle-même, nous allons avoir besoin de nombreuses définitions et de propriétés sur ces définitions.

4.3.1.1 *remEl*

On appelle *remEl* la fonction qui permet d'enlever un élément d'une *ilist*. Un exemple est donné dans la Figure 4.1. On voit que dans la partie gauche, la partie bleue, les indices ne changent pas (mais on passe d'une *ilist* à 6 éléments à une *ilist* à 5 éléments), dans la partie droite en revanche, la partie rouge, les indices sont diminués de 1.

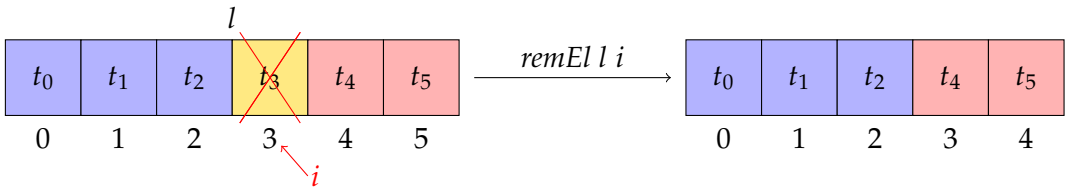


Figure 4.1 — Suppression de l'élément d'indice i (ici, $decode\ i = 3$) dans l

Pour des raisons de lisibilité, nous allons tout d'abord définir à part la partie fonctionnelle de *remEl* que nous appellerons *remEln*. Nous aurons besoin d'une fonction qui permette de convertir un élément de $Fin\ n$ en élément de $Fin\ (n+1)$ en conservant la valeur du *decode*. Logiquement parlant, il s'agit d'un affaiblissement. On appelle cette fonction *weakFin* et elle est définie comme suit :

Définition 4.10 (*weakFin*)**.**

$$\begin{array}{rcl} weakFin\ (n : nat) & : & Fin\ n \to Fin\ (n+1) \\ weakFin\ (first\ k) & := & first\ (k+1) \\ weakFin\ (succ\ i) & := & succ\ (weakFin\ i) \end{array}$$

Lemme 4.3. $\forall i, weakFin\ i =_{Fin} i$

La preuve est une simple induction sur i. La fonction *weakFin* augmente seulement l'indice de type de son argument. Nous pouvons maintenant définir *remEln* (qui "enlève" le $i^{\text{ème}}$ élément de ln) :

Définition 4.11 (*remEln*)**.**

$$\begin{array}{l} remEln\ (n : \mathbb{N}) : ilistn\ T\ (n+1) \to Fin\ (n+1)\ \to ilistn\ T\ n \\ remEln\ ln\ i\ i' \quad = \quad \begin{cases} ln\ (succ\ i') & si\ i \leq_{Fin} i' \\ ln\ (weakFin\ i') & si\ i' <_{Fin} i \end{cases} \end{array}$$

Et la définition de *remEl* se déduit naturellement :

Définition 4.12 (*remEl*)**.**

$$\begin{array}{rcl} remEl\ (l : ilist\ T) : Fin\ (lg\ l) & \to & ilist\ T \\ remEl\ \langle n+1, ln\rangle\ i & = & \langle n, remEln\ ln\ i\rangle \end{array}$$

Remarque 4.3. *Si l est vide, alors i (de type $Fin\ (lg\ l)$) n'existe pas.*

Des Définitions 4.11 et 4.12 on peut déduire immédiatement les trois caractérisations suivantes du comportement de *remEl* :

Propriété 4.1.

$$\forall l\ i, lg\ (remEl\ l\ i) + 1 = lg\ l \tag{4.1.1}$$

$$\forall l\ i\ i', i' <_{Fin} i \Rightarrow fct\ (remEl\ l\ i)\ i' = fct\ l\ (conv_h\ (weakFin\ i')) \tag{4.1.2}$$

avec h de type $lg\ (remEl\ l\ i) + 1 = lg\ l$ déduit de la Propriété 4.1.1

$$\forall l\ i\ i', i \leq_{Fin} i' \Rightarrow fct\ (remEl\ l\ i)\ i' = fct\ l\ (conv_h\ (succ\ i')) \tag{4.1.3}$$

avec h de type $lg\ (remEl\ l\ i) + 1 = lg\ l$ déduit de la Propriété 4.1.1

De ces propriétés, on peut déduire que tout élément de *remEl l i* est aussi un élément de *l* :

Lemme 4.4. $\forall l\ i\ i', \exists i'', fct\ (remEl\ l\ i)\ i' = fct\ l\ i''$

Démonstration. Pour pouvoir utiliser les Propriétés 4.1.2 et 4.1.3, comparons i et i'.

[Cas $H : i \leq_{Fin} i'$**]** D'après la Propriété 4.1.3, on a $fct\ (remEl\ l\ i)\ i' = fct\ l\ (conv_h\ (succ\ i'))$ avec $h : lg\ (remEl\ l\ i) + 1 = lg\ l$. On prend donc $i'' := conv_h\ (succ\ i')$.

[Cas $H : i' <_{Fin} i$**]** D'après la Propriété 4.1.2, on a $fct\ (remEl\ l\ i)\ i' = fct\ l\ (conv_h\ (weakFin\ i'))$ avec $h : lg\ (remEl\ l\ i) + 1 = lg\ l$. On prend donc $i'' := conv_h\ (weakFin\ i')$.

□

Afin de valider nos définitions, on peut montrer la compatibilité de *remEl* avec plusieurs notions introduites précédemment. Tout d'abord, avec *ilist_rel* :

Lemme 4.5.

$$\forall n\ ln_1\ ln_2\ i,\ ilist\_rel_R\ \langle n, ln_1\rangle\ \langle n, ln_2\rangle \Rightarrow ilist\_rel_R\ (remEl\ \langle n, ln_1\rangle\ i)\ (remEl\ \langle n, ln_2\rangle\ i)$$

Idée de la preuve. Comparer deux *ilist* avec *ilist_rel* revient à comparer leurs longueurs (ici cela se fait facilement) puis à les comparer point à point. Ici, notre objectif sera d'appliquer les Propriétés 4.1.2 et 4.1.3. Pour choisir quelle propriété appliquer, on compare la valeur de *decode i* avec celle de l'indice analysé.

Démonstration. Preuve détaillée en page 174. □

Puis avec *imap* :

Lemme 4.6.

$$\forall (f : T \to U)\ (l : ilist\ T)\ (i : Fin\ (lg\ l)), ilist\_rel_{eq}\ (remEl\ (imap\ f\ l)\ i)\ (imap\ f\ (remEl\ l\ i))$$

Idée de la preuve. L'idée est exactement la même que dans la démonstration précédente : comparer les longueurs des deux *ilist* puis comparer i avec la position de l'élément analysé pour pouvoir appliquer les Propriétés 4.1.2 et 4.1.3.

Démonstration. Preuve détaillée en page 175. □

Enfin, comme dernière validation de la définition de *remEl* on peut montrer qu'elle est en adéquation avec les définitions de *ileft* et de *iright* qu'on a données Section 3.2.6 :

Lemme 4.7. $\forall l\ i, ilist\_rel_{eq}\ (iappend\ (ileft\ l\ i)\ (iright\ l\ i))\ (remEl\ l\ i)$

Idée de la preuve. Ici encore, après avoir montré que les *ilist* étaient de même longueur, on va comparer i et la position de l'élément analysé pour pouvoir utiliser les Propriétés 3.6 et 4.1.

Démonstration. Preuve détaillée en page 177. □

On peut aussi en déduire le corollaire suivant qui nous sera utile par la suite :

Corollaire 4.8. $\forall l\ i, (ilist2list\ (ileft\ l\ i))@ilist2list\ (iright\ l\ i)\ = ilist2list\ (remEl\ l\ i)$

4.3.1.2 *addEl*

Symétriquement, on peut définir une fonction qui ajoute un élément dans une *ilist* à une place donnée. Un exemple est donné Figure 4.2. On voit que l'opération à effectuer sur les éléments (indices, etc.) est exactement symétrique à celle de *remEl*.

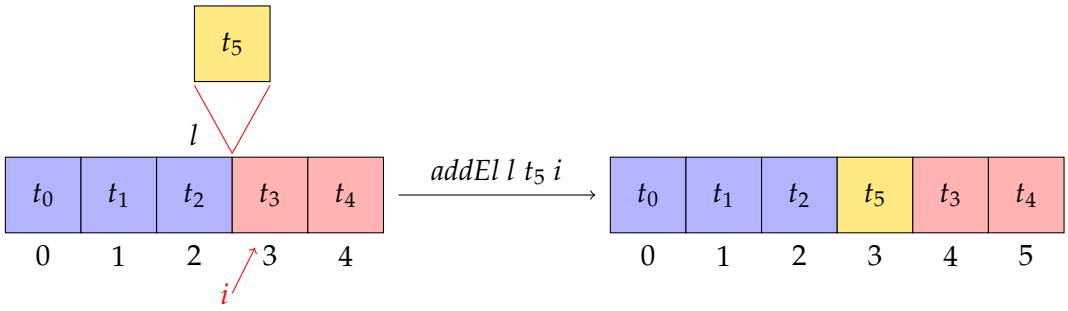


Figure 4.2 — Ajout d'un élément à l'indice i (ici, $decode\ i = 3$) dans l

On appelle cette fonction *addEl*. Pour la définir, on procède comme pour *remEl* et on définit d'abord la partie fonctionnelle que l'on appelle *addEln*.

Remarque 4.4. *La définition de addEl est seulement présentée ici pour la symétrie et pour contribuer à valider la définition de remEl. Elle ne sera plus utilisée par la suite, en particulier, elle n'entre aucunement dans la définition inductive des permutations.*

Dans la suite nous aurons également besoin de la fonction de conversion suivante, qui fait partie de la librairie standard de Coq et dont on donne seulement le type :

$$lt_0\ (n\ m : nat) : n < m \to 0 < m$$

Grâce à cela, on peut maintenant définir *addEln* (elle ajoute l'élément t dans ln à la place i) :

Définition 4.13 (*addEln*)**.**

$$addEln\ (n : \mathbb{N}) : ilistn\ T\ n \to T \to Fin\ (n+1)\ \to ilistn\ T\ (n+1)$$
$$addEln\ ln\ t\ i\ i' \quad = \quad \begin{cases} ln\ (getcons\ i'\ (lt_0\ hyp)) & si\ i <_{Fin} i'\ (hyp) \\ t & si\ i =_{Fin} i' \\ ln\ (code\ h) & si\ i >_{Fin} i'\ (hyp) \\ \qquad avec\ h\ preuve\ de\ decode\ i' < n, déduite\ de\ hyp\ en & \\ \qquad combinaison\ avec\ le\ Lemme\ 3.5 & \end{cases}$$

Remarque 4.5. *Nous avons introduit getcons à la Définition 3.7.*

Et la définition de *addEl* se déduit naturellement :

Définition 4.14 (*addEl*)**.**

$$addEl\ (l : ilist\ T) : T \to Fin\ (lg\ l + 1)\ \to ilist\ T$$
$$addEl\ \langle n, ln\rangle\ t\ i \quad = \quad \langle n+1, addEln\ ln\ t\ i\rangle$$

Comme pour *remEl*, des Définitions 4.13 et 4.14, nous pouvons déduire immédiatement les caractérisations suivantes du comportement de *addEl* :

Propriété 4.2.

$$\forall t\ i, lg\ (addEl\ l\ t\ i) = lg\ l + 1 \tag{4.2.1}$$
$$\forall l\ t\ i\ i'\ (h : i' <_{Fin} i), fct\ (addEl\ l\ t\ i)\ i' = fct\ l\ (code\ h') \tag{4.2.2}$$
où h' est une preuve de decode $i' < n$, déduite par transitivité de h et du Lemme 3.5
$$\forall l\ t\ i\ i', i =_{Fin} i' \rightarrow fct\ (addEl\ l\ t\ i)\ i' = t \tag{4.2.3}$$
$$\forall l\ t\ i\ i'\ (h : i <_{Fin} i'), fct\ (addEl\ l\ t\ i)\ i' = fct\ l\ (getcons\ (conv_{h_1}\ i')\ h_2)) \tag{4.2.4}$$
où h_1 est déduit de la Propriété 4.2.1 et h_2 est une preuve de $0 <$ decode $(conv_{h_1}\ i')$ déduite par transitivité de h, de la Propriété 3.2 et de l'utilisation de lt_0

Afin de valider les définitions de *remEl* et d'*addEl*, il est maintenant intéressant de prouver que ces deux fonctions sont bien des inverses l'une de l'autre : c'est-à-dire que la composition des deux donne bien l'identité.

Lemme 4.9. *$\forall l\ i, ilist\_rel_{eq}\ l\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i))$ avec h déduit de la Propriété 4.1.1.*

Idée de la preuve. Ici encore l'idée va être d'utiliser les caractérisations que nous avons données de *remEl* (Propriété 4.1) et d'*addEl* (Propriété 4.2). Pour cela, nous raisonnerons par comparaison entre *i* et la position de l'élément analysé.

Démonstration. Preuve détaillée en page 178. □

Lemme 4.10. *$\forall l\ i\ t, ilist\_rel_{eq}\ l\ (remEl\ (addEl\ l\ t\ i)\ (conv_h\ i))$ avec h déduit de la Propriété 4.2.1.*

Démonstration. La preuve est semblable à celle du Lemme 4.9. Nous ne la détaillons pas ici. □

4.3.1.3 *indexInRemEl* **et** *indexFromRemEl*

Nous aurons besoin de deux méthodes qui permettent de convertir des indices d'une *ilist* en indices de cette même *ilist* privée d'un élément (avec *remEl*) et vice-versa. On cherche i_c tel que $fct\ (remEl\ l\ i)\ i_c = fct\ l\ i'$ (avec $i' \neq i$) et symétriquement i_c tel que $fct\ (remEl\ l\ i)\ i' = fct\ l\ i_c$. On appelle ces fonctions respectivement *indexInRemEl* et *indexFromRemEl*. Ces fonctions calculent les changements d'indices tels qu'illustrés sur les Figures 4.1 et 4.2. Elles sont définies comme suit :

Définition 4.15 (*indexInRemEl*)**.**

$$\begin{array}{l} indexInRemEl\ (n : nat)\ (i\ i' : Fin\ (n+1)) : i \neq_{Fin} i' \rightarrow Fin\ n \\ indexInRemEl\ n\ i\ i'\ h \quad = \quad getcons\ i'\ (lt_0\ h') \quad si\ h' : i <_{Fin} i' \\ indexInRemEl\ n\ i\ i'\ h \quad = \quad code\ h'' \qquad\qquad si\ h' : i' <_{Fin} i \\ \text{où } h'' \text{ est une preuve de decode } i' < n \text{ déduite de } h' \text{ et du Lemme 3.5} \end{array}$$

Remarque 4.6. *Pour i et i' du type Fin $(n+1)$, $i \neq_{Fin} i' \Leftrightarrow i \neq i'$ mais il s'est avéré plus facile de travailler sur $\neq_{Fin}$.*

On peut également caractériser *indexInRemEl* par les propriétés suivantes (non démontrées ici, mais facilement démontrables) :

Propriété 4.3.

$$\forall i\ i'\ (h_1 : i \neq_{Fin} i'), i <_{Fin} i' \rightarrow decode\ (indexInRemEl\ i\ i'\ h_1) + 1 = decode\ i' \quad (4.3.1)$$

$$\forall i\ i'\ (h_1 : i \neq_{Fin} i'), i' <_{Fin} i \rightarrow indexInRemEl\ i\ i'\ h_1 =_{Fin} i' \quad (4.3.2)$$

$$\forall i\ i'\ (h_1 : i \neq_{Fin} weakFin\ i'),$$
$$weakFin\ i' <_{Fin} i \rightarrow indexInRemEl\ i\ (weakFin\ i')\ h_1 = i' \quad (4.3.3)$$

$$\forall i\ i'\ (h_1 : i \neq_{Fin} i'),$$
$$indexInRemEl\ i\ i'\ h_1 <_{Fin} i \rightarrow weakFin\ (indexInRemEl\ i\ i'\ h_1) = i' \quad (4.3.4)$$

$$\forall i\ i'\ (h_1 : i \neq_{Fin} (succ\ i')), i <_{Fin} succ\ i' \rightarrow indexInRemEl\ i\ (succ\ i')\ h_1 = i' \quad (4.3.5)$$

$$\forall i\ i'\ (h_1 : i \neq_{Fin} i'), i \leq_{Fin} indexInRemEl\ i\ i'\ h_1 \rightarrow succ\ (indexInRemEl\ i\ i'\ h_1) = i' \quad (4.3.6)$$

Remarque 4.7. *La condition dans la Propriété 4.3.6 est l'inverse de la condition dans la Propriété 4.3.4, c'est pour cela qu'elle est fournie sous cette forme. Cependant, elle ne sera vérifiée que pour $i =_{Fin} indexInRemEl\ i\ i'\ h_1$ (cela apparaît clairement grâce aux propriétés 4.3.1 et 4.3.2).*

On peut démontrer que le lemme suivant est vrai à propos de *indexInRemEl* :

Lemme 4.11.
$$\forall n\ (ln : ilistn\ T\ (n+1))\ (i\ i' : Fin\ (n+1))(h : i \neq_{Fin} i'),$$
$$ln\ i' = fct\ (remEl\ \langle n+1, ln\rangle\ i)\ (indexInRemEl\ i\ i'\ h)$$

Démonstration. Cela se démontre aisément en faisant une comparaison entre les valeurs de *decode i* et *decode i'* et en utilisant les Propriétés 4.1.2 et 4.1.3. □

Définition 4.16 (*indexFromRemEl*)**.**

$$indexFromRemEl\ (n : nat) : Fin\ (n+1) \rightarrow Fin\ n \rightarrow Fin\ (n+1)$$
$$indexFromRemEl\ n\ i\ i' \quad = \quad succ\ i' \qquad si\ i \leq_{Fin} i'$$
$$indexFromRemEl\ n\ i\ i' \quad = \quad weakFin\ i' \quad si\ i' <_{Fin} i$$

De même que pour *indexInRemEl*, on peut démontrer le lemme suivant pour *indexFromRemEl* :

Lemme 4.12.
$$\forall n\ (ln : ilistn\ T\ (n+1))\ (i : Fin\ (n+1))(i' : Fin\ n),$$
$$ln\ (indexFromRemEl\ i\ i') = fct\ (remEl\ \langle n+1, ln\rangle\ i)\ i'$$

Démonstration. De la même façon que pour *indexInRemEl*, cela se démontre en faisant une comparaison entre les valeurs de *decode i* et *decode i'* et en utilisant les Propriétés 4.1.2 et 4.1.3. □

On aura également besoin dans la suite du résultat suivant :

Lemme 4.13. $\forall n\ (i : Fin\ (n+1))\ (i' : Fin\ n), indexFromRemEl\ i\ i' \neq i$

Démonstration. La preuve est une simple analyse de cas sur les valeurs de *decode i'* et de *decode i* qui suit directement la définition de *indexFromRemEl*. □

Finalement, nous allons également donner deux lemmes qui permettent de lier ces deux définitions :

Lemme 4.14.

$$\forall n\ i\ i'\ (h : i \neq_{Fin} indexFromRemEl\ i\ i'), indexInRemEl\ i\ (indexFromRemEl\ i\ i')\ h = i'$$

Démonstration. Preuve détaillée en page 179. □

Remarque 4.8. *Le Lemme 4.13 nous fournit immédiatement une preuve pour l'hypothèse h.*

Lemme 4.15. $\forall n\ i\ i'\ (h : i \neq_{Fin} i'), indexFromRemEl\ i\ (indexInRemEl\ i\ i'\ h) = i'$

Démonstration. Preuve détaillée en page 180. □

4.3.1.4 Lemme d'interchangeabilité

Grâce à ces définitions, nous pouvons définir un des lemmes les plus importants sur *remEl* : l'interchangeabilité des indices retirés de la *ilist*. En effet, si on retire deux éléments d'une *ilist*, quel que soit l'ordre dans lequel on les retire, le résultat doit être le même. En revanche, l'étape intermédiaire est différente. C'est ce qui est illustré dans la Figure 4.3.

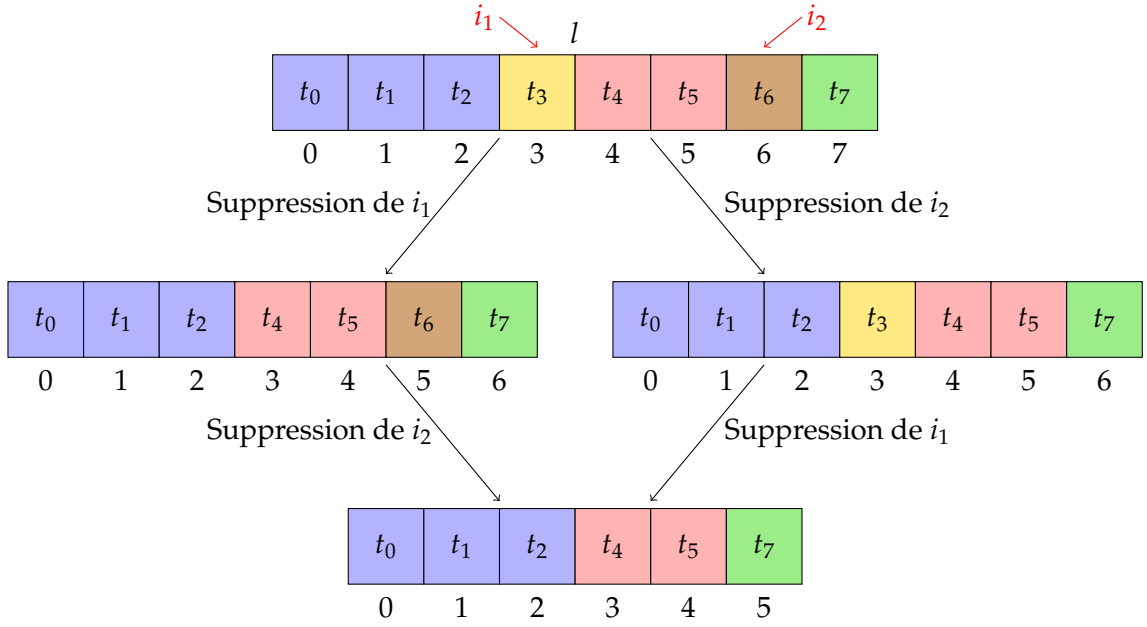


Figure 4.3 — Suppressions de deux éléments d'indice i_1 (ici, avec $decode\ i_1 = 3$) et i_2 (ici, avec $decode\ i_2 = 6$) dans l

Lemme 4.16.

$$\begin{array}{ll} & \forall n\ (ln : ilistn\ T\ (n+1))\ (i\ i' : Fin\ (n+1))\ (a : i \neq_{Fin} i')\ (a' : i' \neq_{Fin} i), \\ ilist\_rel_{eq} & (remEl\ (remEl\ \langle n+1, ln\rangle\ i)\ (indexInRemEl\ i\ i'\ a)) \\ & (remEl\ (remEl\ \langle n+1, ln\rangle\ i')\ (indexInRemEl\ i'\ i\ a')) \end{array}$$

Idée de la preuve. La preuve ici est assez longue et répétitive. Il s'agit principalement d'étudier les différents cas pour savoir à chaque fois laquelle des Propriétés 4.1.2 et 4.1.3 on peut appliquer. Cette preuve a pu être entièrement automatisée grâce au langage de tactiques de Coq, `Ltac`. Nous ne la détaillons donc pas ici.

4.3.2 Définitions de *iperm_ind*

Nous pouvons maintenant enfin définir la relation de permutations à l'aide de la fonction *remEl* définie précédemment. Cette relation peut être définie par n'importe laquelle des trois définitions suivantes. Nous prouverons ensuite qu'elles sont équivalentes.

Définition 4.17 (*iperm_ind*, vue inductivement)**.**

$$\forall l_1\ l_2,\ iperm\_ind_R\ l_1\ l_2 \Leftrightarrow \begin{cases} lg\ l_1 = lg\ l_2 = 0 \quad \text{ou} \\ \exists i_1\ i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge iperm\_ind_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2) \end{cases}$$

Définition 4.18 (*iperm_ind*′, vue inductivement)**.**

$$\forall l_1\ l_2,\ iperm\_ind'_R\ l_1\ l_2 \quad \Leftrightarrow \quad lg\ l_1 = lg\ l_2 \quad \wedge \quad \begin{array}{l} (\forall i_1 \exists i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \\ \wedge\ iperm\_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)) \end{array}$$

Définition 4.19 (*iperm_ind*″, vue inductivement)**.**

$$\forall l_1\ l_2,\ iperm\_ind''_R\ l_1\ l_2 \quad \Leftrightarrow \quad lg\ l_1 = lg\ l_2 \quad \wedge \quad \begin{array}{l} (\forall i_2 \exists i_1, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \\ \wedge\ iperm\_ind''_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)) \end{array}$$

Avant de prouver l'équivalence entre ces définitions, nous devons introduire quelques lemmes. Les deux premiers sont très simples et leur preuve est immédiate.

Lemme 4.17. $\forall l_1 l_2, iperm\_ind_R\ l_1\ l_2 \Rightarrow lg\ l_1 = lg\ l_2$

Idée de la preuve. C'est une simple induction qui n'est pas détaillée ici.

Remarque 4.9. *Pour les Définitions 4.18 et 4.19, ce lemme est évident.*

Lemme 4.18. $\forall ln_1\ ln_2, iperm\_ind'_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle$

Démonstration. Preuve très simple détaillée en page 180. □

Remarque 4.10. *Pour la Définition 4.17, ceci est évident. Pour la Définition 4.19, le lemme est également vrai. La démonstration est identique à celle pour la Définition 4.18 (en remplaçant le* $\forall i_1 \exists i_2$ *par* $\forall i_2 \exists i_1$*) et n'est pas détaillée ici.*

Nous aurons aussi besoin du lemme suivant dans la suite. Pour le moment, nous ne l'énonçons et ne le prouvons que pour la Définition 4.17 mais une fois le Théorème 4.1 démontré, il sera évidemment vrai pour n'importe laquelle des trois définitions. Ce lemme permet de faire la réécriture dans *iperm_ind* par rapport à *ilist_rel*.

Lemme 4.19. $\forall l_1\ l'_1\ l_2, ilist\_rel_{eq}\ l_1\ l'_1 \wedge iperm\_ind_R\ l_1\ l_2 \Rightarrow iperm\_ind_R\ l'_1\ l_2$

Démonstration. Preuve détaillée en page 181. □

On peut prouver de la même façon le lemme suivant :

Lemme 4.20. $\forall l_1\ l_2\ l'_2, ilist\_rel_{eq}\ l_2\ l'_2 \wedge iperm\_ind_R\ l_1\ l'_2 \Rightarrow iperm\_ind_R\ l_1\ l_2$

Grâce à tous ces nouveaux résultats, nous allons pouvoir effectuer la preuve de l'équivalence des Définitions 4.17, 4.18 et 4.19.

Théorème 4.1. $\forall l_1\ l_2, iperm\_ind_R\ l_1\ l_2 \Leftrightarrow iperm\_ind'_R\ l_1\ l_2 \Leftrightarrow iperm\_ind''_R\ l_1\ l_2$

Démonstration. Pour prouver ce théorème, nous pourrions essayer de prouver que $iperm\_ind \Rightarrow iperm\_ind' \Rightarrow iperm\_ind'' \Rightarrow iperm\_ind$. Cependant, du fait de la similitude entre les Définitions 4.18 et 4.19, on va prouver que $iperm\_ind \Leftrightarrow iperm\_ind'$. En effet, ensuite la preuve que $iperm\_ind \Leftrightarrow iperm\_ind''$ est identique (aux numéros des indices près). Et on aura donc ce que l'on cherche : $iperm\_ind \Leftrightarrow iperm\_ind' \Leftrightarrow iperm\_ind''$.

[Direction $iperm\_ind \Rightarrow iperm\_ind'$**]**

Idée de la preuve. On raisonne par induction sur $iperm\_ind_R\ l_1\ l_2$. Le cas de base se résout aisément. En revanche, le cas inductif est plus compliqué parce qu'on obtient des indices différents pour l_1 en utilisant les Définitions 4.17 et 4.18. On doit donc aller un cran plus loin des deux côtés afin de retirer les deux indices dans la conclusion et dans les hypothèses et on utilise ensuite le Lemme 4.16 pour terminer la preuve.

Cette preuve est la plus compliquée. Nous allons procéder par induction sur $lg\ l_1$ (qu'on notera n). Soient $H_1 : lg\ l_1 = n$ et $H_2 : iperm\_ind_R\ l_1\ l_2$.

[Cas 0**]** D'après le Lemme 4.18, il nous suffit de prouver que $lg\ l_1 = lg\ l_2 = 0$. On sait que $lg\ l_1 = 0$ d'après H_1 et on obtient $lg\ l_1 = lg\ l_2$ grâce au Lemme 4.17 et à H_2.

[Cas $n+1$**]** D'après la Définition 4.18, il nous suffit de prouver que :

1. $lg\ l_1 = lg\ l_2$: on prouve cela grâce au Lemme 4.17 et à H_2.
2. $\forall i_1 \exists i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge iperm\_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$: l'hypothèse d'induction est $IH : \forall l_1\ l_2, iperm\_ind_R\ l_1\ l_2 \wedge lg\ l_1 = n \Rightarrow iperm\_ind'_R\ l_1\ l_2$. On veut montrer que :

$$\begin{array}{lll} iperm\_ind_R\ l_1\ l_2 & \Rightarrow & \forall i_1 \exists i_2, \quad R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \\ & & \quad \wedge iperm\_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2) \end{array}$$

Nous allons passer par une étape intermédiaire et nous allons montrer que :

$$\begin{array}{l} iperm\_ind_R\ l_1\ l_2 \\ \Rightarrow \forall i_1 \exists i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge iperm\_ind_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2) \\ \Rightarrow \forall i_1 \exists i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge iperm\_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2) \end{array}$$

Supposons qu'on ait montré que

$$\begin{array}{lll} H_3 : \quad iperm\_ind_R\ l_1\ l_2 & \Rightarrow & \forall i_1 \exists i_2, \quad R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \\ & & \quad \wedge iperm\_ind_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2) \end{array}$$

On termine alors la preuve facilement. En effet, on veut montrer que :

$$\exists i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge iperm\_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$$

On prouve la première partie avec H_3 appliquée à H. Pour la deuxième partie, on utilise IH et encore H_3 appliquée à H.

Il nous suffit donc de prouver le lemme suivant pour terminer cette preuve :

Lemme 4.21.

$$\begin{array}{lll} \forall l_1\ l_2, \quad iperm\_ind_R\ l_1\ l_2 & \Rightarrow & \forall i_1 \exists i_2, \quad R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \\ & & \quad \wedge iperm\_ind_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2) \end{array}$$

Démonstration. Nous allons procéder par induction sur $H : iperm\_ind_R\ l_1\ l_2$.

[Cas de base] On a $H : lg\ l_1 = lg\ l_2 = 0$. Dans ce cas, $i_1 : Fin\ (lg\ l_1)$ n'existe pas (puisque $lg\ l_1 = 0$ et donc $Fin\ (lg\ l_1)$ est vide).

[Cas général] On a i_1 et i_2 tels que,

$$H_1 : R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \text{ et } H_2 :\ iperm\_ind_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$$

Et l'hypothèse d'induction IH est :

$$\begin{array}{ll} \forall i'_1\ \exists i'_2, & R\ (fct\ (remEl\ l_1\ i_1)\ i'_1)\ (fct\ (remEl\ l_2\ i_2)\ i'_2) \\ \quad \wedge & iperm\_ind_R\ (remEl\ (remEl\ l_1\ i_1)\ i'_1)\ (remEl\ (remEl\ l_2\ i_2)\ i'_2) \end{array}$$

On veut prouver

$$\forall i_1 \exists i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge\ iperm\_ind_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$$

(on sait déjà que $lg\ l_1 = lg\ l_2$ grâce au Lemme 4.17). Soit donc i'_1, on veut prouver que

$$\exists i'_2, R\ (fct\ l_1\ i'_1)\ (fct\ l_2\ i'_2) \wedge\ iperm\_ind_R\ (remEl\ l_1\ i'_1)\ (remEl\ l_2\ i'_2)$$

Comparons i_1 et i'_1 ; deux cas de figure se présentent :

[$H_3 : i_1 =_{Fin} i'_1$] Dans ce cas, on prend $i'_2 = i_2$ et H_1 et H_2 prouvent directement $R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge\ iperm\_ind_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$

[$H_3 : i_1 \neq_{Fin} i'_1$] Soient n, ln_1, ln_2 tels que $l_1 = \langle n+1, ln_1\rangle$ et $l_2 = \langle n+1, ln_2\rangle$ (on sait que $lg\ l_1 = lg\ l_2$ grâce au Lemme 4.17 et on peut prendre directement $n+1$ car sinon, i_1 n'existe pas). On veut donc prouver que :

$$\begin{array}{ll} \exists i'_2, & R\ (ln_1\ i'_1)\ (ln_2\ i'_2) \\ \quad \wedge & iperm\_ind_R\ (remEl\ \langle n+1, ln_1\rangle\ i'_1)\ (remEl\ \langle n+1, ln_2\rangle\ i'_2) \end{array}$$

Pour pouvoir trouver i'_2, nous allons utiliser l'hypothèse d'induction. Pour cela, nous devons trouver l'indice correspondant à i'_1 dans $remEl\ l_1\ i_1$. Il nous est donné par la fonction *indexInRemEl*. Nous l'appelons i'^{new}_1 et $i'^{new}_1 := indexInRemEl\ i_1\ i'_1\ H_3$. En instanciant IH avec i'^{new}_1, on obtient i'^{IH}_2 tel que

$$\begin{array}{lll} IH_1 : & R\ (fct\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ i'^{new}_1)\ (fct\ (remEl\ \langle n+1, ln_2\rangle\ i_2)\ i'^{IH}_2) \end{array}$$

et

$$\begin{array}{lll} IH_2 : & iperm\_ind_R & (remEl\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ i'^{new}_1) \\ & & (remEl\ (remEl\ \langle n+1, ln_2\rangle\ i_2)\ i'^{IH}_2) \end{array}$$

Pour obtenir i'_2, il ne nous reste donc plus maintenant qu'à retrouver l'indice correspondant à i'^{IH}_2 dans $\langle n+1, ln_2\rangle$. On utilise pour cela *indexFromRemEl* : $i'_2 := indexFromRemEl\ i_2\ i'^{IH}_2$. On veut donc maintenant prouver :

1. $R\ (ln_1\ i'_1)\ (ln_2\ i'_2)$: on a :

$$\begin{array}{rcll} ln_1\ i'_1 & = & fct\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ (indexInRemEl\ i_1\ i'_1\ H_3) & \text{(Lemme 4.11)} \\ & = & fct\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ i'^{new}_1 & \end{array}$$

et

$$\begin{array}{rcll} ln_2\ i'_2 & = & ln_2\ (indexFromRemEl\ i_2\ i'^{IH}_2) & \\ & = & fct\ (remEl\ \langle n+1, ln_2\rangle\ i_2)\ i'^{IH}_2 & \text{(Lemme 4.12)} \end{array}$$

Donc

$$\begin{array}{rl} & R\ (ln_1\ i'_1)\ (ln_2\ i'_2) \\ \Leftrightarrow & R\ (fct\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ i'^{new}_1)\ (fct\ (remEl\ \langle n+1, ln_2\rangle\ i_2)\ i'^{IH}_2) \end{array}$$

Cette dernière assertion est vraie d'après l'hypothèse IH_1.

2. $iperm\_ind_R\ (remEl\ \langle n+1, ln_1\rangle\ i'_1)\ (remEl\ \langle n+1, ln_2\rangle\ i'_2)$: l'idée ici est d'utiliser le Lemme 4.16 pour ensuite appliquer l'hypothèse d'induction. Pour cela, nous devons tout d'abord trouver les indices correspondants à i_1 dans $(remEl\ \langle n+1, ln_1\rangle\ i'_1)$ et à i_2 dans $(remEl\ \langle n+1, ln_2\rangle\ i'_2)$. On les appelle respectivement i_1^{new} et i_2^{new}. Ils sont définis comme suit :

$$\begin{array}{ll} i_1^{new} = indexInRemEl\ i'_1\ i_1\ H'_3 & (\text{avec } H'_3 : i'_1 \neq_{Fin} i_1 \text{ déduit de } H_3) \\ i_2^{new} = indexInRemEl\ i'_2\ i_2\ H_4 & (\text{avec } H_4 : i'_2 \neq_{Fin} i_2 \text{ déduit du Lemme 4.13}) \end{array}$$

D'après la Définition 4.17 utilisée avec i_1^{new} et i_2^{new} il nous suffit de prouver que :

(a) $R\ (fct\ (remEl\ \langle n+1, ln_1\rangle\ i'_1)\ i_1^{new})\ (fct\ (remEl\ \langle n+1, ln_2\rangle\ i'_2)\ i_2^{new}$: en appliquant deux fois le Lemme 4.11, on obtient que :

$$\begin{array}{ll} & R\ (fct\ (remEl\ \langle n+1, ln_1\rangle\ i'_1)\ i_1^{new})\ (fct\ (remEl\ \langle n+1, ln_2\rangle\ i'_2)\ i_2^{new} \\ \Leftrightarrow & R\ (ln_1\ i_1)\ (ln_2\ i_2) \end{array}$$

Cette dernière assertion est notre hypothèse H_1.

(b) $\begin{array}{ll} iperm\_ind_R & (remEl\ (remEl\ \langle n+1, ln_1\rangle\ i'_1)\ i_1^{new}) \\ & (remEl\ (remEl\ \langle n+1, ln_2\rangle\ i'_2)\ i_2^{new}) \end{array}$: on peut maintenant procéder à l'inversion des indices en utilisant le Lemme 4.16. On l'instancie deux fois et on obtient les deux hypothèses suivantes :

$$\begin{array}{lll} H_5 : & ilist\_rel_{eq} & (remEl\ (remEl\ \langle n+1, ln_1\rangle\ i'_1)\ i_1^{new}) \\ & & (remEl\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ (indexInRemEl\ i_1\ i'_1\ H_3)) \\ H_6 : & ilist\_rel_{eq} & (remEl\ (remEl\ \langle n+1, ln_1\rangle\ i'_2)\ i_2^{new}) \\ & & (remEl\ (remEl\ \langle n+1, ln_2\rangle\ i_2)\ (indexInRemEl\ i_2\ i'_2\ H'_4)) \\ & (\text{avec } H'_4 : i_2 \neq_{Fin} i'_2 \text{ déduit de } H_4) & \end{array}$$

D'après les Lemmes 4.19 et 4.20 utilisés avec H_5, H_6 il nous suffit maintenant de prouver que :

$$\begin{array}{ll} iperm\_ind_R & (remEl\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ (indexInRemEl\ i_1\ i'_1\ H_3)) \\ & (remEl\ (remEl\ \langle n+1, ln_2\rangle\ i_2)\ (indexInRemEl\ i_2\ i'_2\ H'_4)) \end{array}$$

C'est-à-dire, d'après la définition de $i_1'^{new}$:

$$\begin{array}{ll} iperm\_ind_R & (remEl\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ i_1'^{new}) \\ & (remEl\ (remEl\ \langle n+1, ln_2\rangle\ i_2)\ (indexInRemEl\ i_2\ i'_2\ H'_4)) \end{array}$$

Si on prouve que $indexInRemEl\ i_2\ i'_2\ H'_4 = i_2'^{IH}$, alors on obtiendra

$$\begin{array}{ll} iperm\_ind_R & (remEl\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ i_1'^{new}) \\ & (remEl\ (remEl\ \langle n+1, ln_2\rangle\ i_2)\ i_2'^{IH}) \end{array}$$

qui est notre hypothèse IH_2, ce qui terminerait donc la preuve. Il nous reste donc seulement à prouver que : $indexInRemEl\ i_2\ i'_2\ H'_4 = i_2'^{IH}$. Or :

$$\begin{array}{lll} & indexInRemEl\ i_2\ i'_2\ H'_4 & \\ = & indexInRemEl\ i_2\ (indexFromRemEl\ i_2\ i_2'^{IH})\ H'_4 & (\text{par définition}) \\ = & i_2'^{IH} & (\text{d'après Lemme 4.14}) \end{array}$$

□

[Direction $iperm\_ind' \Rightarrow iperm\_ind$**]** Ici, la preuve est simplement une induction sur $iperm\_ind'_R\ l_1\ l_2$. En effet, à chaque "niveau", la Définition 4.17 nous donne un couple d'éléments équivalents qu'on peut enlever, alors que la Définition 4.18 dit qu'à chaque "niveau" tous les éléments de l_1 ont un équivalent dans l_2 (et cela récursivement). On peut donc voir la Définition 4.17 comme un cas particulier de la Définition 4.18 puisque, si $lg\ l_1 > 0$, on peut choisir canoniquement i_1 comme étant le premier élément de son domaine.

[Cas $iperm\_ind \Leftrightarrow iperm\_ind''$**]** La preuve est ici tout à fait similaire à la preuve de $iperm\_ind \Leftrightarrow iperm\_ind'$. Nous ne la détaillons donc pas.

□

L'utilité d'avoir ces trois définitions équivalentes est qu'elles sont assez différentes structurellement (en tous cas la Définition 4.17 est très différente des Définitions 4.18 et 4.19, on présente ici la Définition 4.19 dans un souci de symétrie, mais en réalité on pourrait s'en passer). Donc, certaines propriétés peuvent être plus faciles à démontrer en utilisant une définition plutôt qu'une autre. C'est ce que nous allons voir maintenant.

4.3.3 Propriétés de *iperm_ind*

Dans cette section nous allons démontrer plusieurs propriétés à propos de *iperm_ind*. D'après le Théorème 4.1, ces propriétés seront donc aussi vraies pour $iperm\_ind'$ et $iperm\_ind''$. Comme nous l'avons dit précédemment, ces propriétés seront parfois même plus faciles à démontrer en utilisant ces autres définitions. Ce sont alors ces preuves que nous montrerons. Mais grâce au Théorème 4.1, ces propriétés resteront vraies sur *iperm_ind*, et c'est sur cette définition que nous les énoncerons. Nous allons montrer qu'elle préserve l'équivalence, qu'elle est moins fine que *ilist_rel*, qu'elle est décidable et enfin qu'elle est monotone vis à vis de sa relation de base.

4.3.3.1 *iperm_ind* préserve l'équivalence

On veut ici démontrer que *iperm_ind* préserve l'équivalence. Nous allons donc montrer successivement, qu'elle préserve la réflexivité, la symétrie et la transitivité. A chaque fois, nous discuterons du choix de la définition à utiliser pour faire la preuve.

Lemme 4.22 (*iperm_ind* préserve la réflexivité)**.** R *réflexive* $\Rightarrow \forall l,\ iperm\_ind_R\ l\ l$

Démonstration. Ici, quelle que soit la définition choisie, la démonstration est simple. Nous choisissons arbitrairement de la faire sur la Définition 4.17. Soient n et ln tels que $l = \langle n, ln\rangle$. On raisonne par induction sur n.

[Cas 0**]** On veut prouver que $iperm\_ind_R\ \langle 0, ln\rangle\ \langle 0, ln\rangle$: c'est immédiat avec la Définition 4.17.

[Cas $n+1$**]** L'hypothèse d'induction IH est : $\forall ln, iperm\_ind_R\ \langle n, ln\rangle\ \langle n, ln\rangle$. On veut prouver que $iperm\_ind_R\ \langle n+1, ln\rangle\ \langle n+1, ln\rangle$. D'après Définition 4.17 on a :

$$\begin{array}{l} iperm\_ind_R\ \langle n+1, ln\rangle\ \langle n+1, ln\rangle \\ \Leftrightarrow\quad \exists i_1\ i_2, R\ (ln\ i_1)\ (ln\ i_2) \wedge iperm\_ind_R\ (remEl\ ln\ i_1)\ (remEl\ ln\ i_2) \end{array}$$

On prend $i_1 = i_2 = first\ n$. On doit alors montrer que :

1. $R\ (ln\ (first\ n))\ (ln\ first\ n))$: vrai par hypothèse puisqu'on suppose R réflexive.

2. $iperm\_ind_R\ (remEl\ ln\ (first\ n))\ (remEl\ ln\ (first\ n))$. Or on a montré que $lg\ (remEl\ ln\ (first\ n)) = n$. On peut donc utiliser IH directement.

□

On note *flip R* la relation R "retournée". C'est-à-dire que $flip\ R = \lambda x_1 \lambda x_2 . R\ x_2\ x_1$. Pour prouver la symétrie de *iperm_ind*, on va d'abord prouver que $iperm\_ind_R$ est vraie si et seulement si $iperm\_ind_{flip\ R}$ est vraie aussi.

Lemme 4.23. $\forall l_1\ l_2, iperm\_ind_R\ l_1\ l_2 \Leftrightarrow iperm\_ind_{flip\ R}\ l_2\ l_1$

Démonstration. Ici, la Définition 4.17 est nettement plus appropriée que la Définition 4.18 ou la Définition 4.19 pour faire la preuve. En effet, la Définition 4.17 fait jouer un rôle similaire à l_1 et à l_2 (avec le $\exists i_1\ i_2$), alors que dans la Définition 4.18, l_1 et l_2 jouent des rôle différents : on "a" l'indice dans l_1 alors qu'on doit "trouver" l'indice dans l_2 (avec le $\forall i_1 \exists i_2$). La preuve ici serait beaucoup plus compliquée à faire directement. La situation est la même pour la Définition 4.19.

[Direction ⇒] On raisonne par induction sur $iperm\_ind_R\ l_1\ l_2$.

[Cas de base] On a $H : lg\ l_1 = lg\ l_2 = 0$. D'après la Définition 4.17, il nous suffit de prouver que $lg\ l_2 = lg\ l_1 = 0$, ce qu'on a directement grâce à H et à la symétrie de l'égalité de Leibniz.

[Cas inductif] On a i_1 et i_2 tels que $H_1 : R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)$ et que l'hypothèse d'induction soit $IH : iperm\_ind_{flip\ R}\ (remEl\ l_2\ i_2)\ (remEl\ l_1\ i_1)$. Pour prouver que $iperm\_ind_{flip\ R}\ l_2\ l_1$ on utilise simplement la Définition 4.17 avec H_1 et IH.

[Direction ⇐] La preuve ici est tout à fait identique à la précédente, nous ne la développons donc pas.

□

A l'aide du lemme précédent, on peut maintenant facilement prouver que *iperm_ind* préserve la symétrie.

Lemme 4.24 (*iperm_ind* préserve la symétrie)**.**

$$R\ symétrique \Rightarrow (\forall l_1\ l_2,\ iperm\_ind_R\ l_1\ l_2 \Rightarrow iperm\_ind_R\ l_2\ l_1)$$

Démonstration. Soit $H : iperm\_ind_R\ l_1\ l_2$. Du Lemme 4.23 appliqué à H on déduit $H_1 : iperm\_ind_{flip\ R}\ l_2\ l_1$. On va raisonner par induction sur l'hypothèse H_1.

[Cas de base] On a comme hypothèse : $H_2 : lg\ l_2 = lg\ l_1 = 0$. On veut prouver que $iperm\_ind_R\ l_2\ l_1$, ce qui est vrai d'après la Définition 4.17 et H_2.

[Cas inductif] On a donc i_1 et i_2 tels que :

$$H_2 : R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \quad \text{et} \quad IH : iperm\_ind_R\ (remEl\ l_2\ i_2)\ (remEl\ l_1\ i_1)$$

On applique le deuxième cas de la Définition 4.17. Il nous suffit de prouver que :

$$\exists i'_2\ i'_1, R\ (fct\ l_2\ i'_2)\ (fct\ l_1\ i'_1) \wedge iperm\_ind_R\ (remEl\ l_2\ i'_2)\ (remEl\ l_1\ i'_1)$$

On prend $i'_2 = i_2$ et $i'_1 = i_1$ et on utilise H_2, IH et la symétrie de R.

□

Inversement, de par sa structure hautement symétrique la Définition 4.17 se prête très mal à une preuve de transitivité, alors que la Définition 4.18 et la Définition 4.19 la rendent très facile. Nous allons donc faire cette preuve avec la Définition 4.18. Pour montrer la transitivité, nous allons déjà faire la preuve dans un cas précis, puis nous utiliserons ce résultat pour le cas général. On définit une relation de transitivité spéciale de la façon suivante :

Définition 4.20. $\forall t_1, transAt_R\ t_1 \Leftrightarrow (\forall t_2\ t_3, R\ t_1\ t_2 \wedge R\ t_2\ t_3 \Rightarrow R\ t_1\ t_3)$

En utilisant cette définition spécifique de la transitivité, on va prouver une version raffinée de la transitivité de *iperm_ind'* :

Lemme 4.25. $\forall l_1, (\forall i_1, transAt_R\ (fct\ l_1\ i_1)) \Rightarrow transAt_{iperm\_ind'_R\ l_1}$

Démonstration. Soit $H_1 : \forall i_1, transAt_R\ (fct\ l_1\ i_1)$. D'après la Définition 4.20, on suppose l_2 et l_3 tels que

$$H_2 : iperm\_ind'_R\ l_1\ l_2 \quad H_3 : iperm\_ind'_R\ l_2\ l_3$$

On veut prouver que $iperm\_ind'_R\ l_1\ l_3$. On raisonne par induction sur H_3, simultanément pour tout l_1. On a alors $H_4 : lg\ l_2 = lg\ l_3$ et l'hypothèse d'induction est :

$$IH : \forall i_2 \exists i_3, R\ (fct\ l_2\ i_2)\ (fct\ l_3\ i_3) \wedge iperm\_ind'_R\ (remEl\ l_2\ i_2)\ (remEl\ l_3\ i_3) \wedge \\ (\forall l_1, iperm\_ind'_R\ l_1\ (remEl\ l_2\ i_2) \wedge (\forall i_1, transAt_R(fct\ l_1\ i_1)) \Rightarrow iperm\_ind'_R\ l_1\ (remEl\ l_3\ i_3))$$

Grâce à la Définition 4.18, on obtient de H_2 :

$$H_5 : lg\ l_1 = lg\ l_2 \quad H_6 : \forall i_1 \exists i_2, R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge iperm\_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$$

D'après la Définition 4.18 il nous suffit de prouver que :

1. $lg\ l_1 = lg\ l_3$: on le prouve grâce à la transitivité de l'égalité de Leibniz et à H_5 et H_4.
2. $\forall i_1 \exists i_3, R\ (fct\ l_1\ i_1)\ (fct\ l_3\ i_3) \wedge iperm\_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_3\ i_3)$: H_6 appliqué à i_1 nous donne i_2 tel que :

   $$H_7 : R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \quad H_8 : iperm\_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$$

   Et *IH* appliquée à i_2 nous donne i_3 tel que :

   $$H_9 : \quad R\ (fct\ l_2\ i_2)\ (fct\ l_3\ i_3) \qquad H_{10} : \quad iperm\_ind'_R\ (remEl\ l_2\ i_2)\ (remEl\ l_3\ i_3)$$
   $$H_{11} : \quad \forall l_1, iperm\_ind'_R\ l_1\ (remEl\ l_2\ i_2) \wedge (\forall i_1, transAt_R(fct\ l_1\ i_1)) \\ \Rightarrow iperm\_ind'_R\ l_1\ (remEl\ l_3\ i_3)$$

   On instancie l'existencielle avec i_3 et il nous reste à montrer que :

   (a) $R\ (fct\ l_1\ i_1)\ (fct\ l_3\ i_3)$: on utilise H_1 avec H_7 et H_9.

   (b) $iperm\_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_3\ i_3)$: d'après H_{11} appliqué à H_8, il nous suffit de montrer que $\forall i, transAt_R(fct\ (remEl\ l_1\ i_1)\ i)$. Le Lemme 4.4 nous donne i' tel que $fct\ (remEl\ l_1\ i_1)\ i = fct\ l_1\ i'$. Il nous suffit donc de montrer que $transAt_R(fct\ l_1\ i')$, ce qui est vrai d'après H_1.

□

Le lemme général de préservation de la transitivité de *iperm_ind* se déduit alors du lemme précédent, dont il est un cas particulier.

Lemme 4.26 (*iperm_ind* préserve la transitivité)**.**

$$R \text{ transitive} \Rightarrow (\forall l_1\ l_2\ l_3,\ iperm\_ind_R\ l_1\ l_2 \wedge iperm\_ind_R\ l_2\ l_3 \Rightarrow iperm\_ind_R\ l_1\ l_3)$$

Démonstration. Comme on l'a dit, la Définition 4.18 se prête beaucoup mieux à la preuve de transitivité que la Définition 4.17. Ce que nous allons donc montrer en réalité est :

$$R \text{ transitive} \Rightarrow (\forall l_1\ l_2\ l_3,\ iperm\_ind'_R\ l_1\ l_2 \wedge iperm\_ind'_R\ l_2\ l_3 \Rightarrow iperm\_ind'_R\ l_1\ l_3)$$

les transformations se faisant grâce au Théorème 4.1. L'énoncé du lemme peut se transformer en :

$$R \text{ transitive} \Rightarrow (\forall l_1, transAt_{iperm\_ind'_R}\ l_1)$$

D'après le Lemme 4.25, il nous suffit de montrer que $\forall i_1, transAt_R(fct\ l_1\ i_1)$. C'est-à-dire que pour t_2 et t_3 fixé, on a $H_1 : R\ (fct\ l_1\ i_1)\ t_2$ et $H_2 : R\ t_2\ t_3$ et on doit montrer que $R\ (fct\ l_1\ i_1)\ t_3$. On montre cela simplement par transitivité de R avec H_1 et H_2. □

On peut également montrer une version hétérogène de la transitivité (dont on aura besoin plus loin), qui ne requiert pas d'hypothèse particulière sur ses relations de base :

Lemme 4.27. *Pour deux relations R_1 et R_2 fixées, soit R_3 la relation définie par*

$$\forall t_1\ t_3, R_3\ t_1\ t_3 \Leftrightarrow \exists t_2, R_1\ t_1\ t_2 \wedge R_2\ t_2\ t_3$$

On a alors

$$\forall l_1\ l_2\ l_3, iperm\_ind_{R_1}\ l_1\ l_2 \wedge iperm\_ind_{R_2}\ l_2\ l_3 \Rightarrow iperm\_ind_{R_3}\ l_1\ l_3$$

Démonstration. Preuve détaillée en page 181. □

On peut donc maintenant énoncer le lemme final :

Lemme 4.28 (*iperm_ind* préserve l'équivalence)**.** *R équivalence $\Rightarrow$ $iperm\_ind_R$ équivalence*

Démonstration. La preuve est simplement une utilisation des lemmes précédents (sauf du Lemme 4.27 qui est un aparté dans cette démonstration de l'équivalence). □

Remarque 4.11. *Comme on l'a dit, grâce au Théorème 4.1, ce résultat est également valable pour iperm_ind′ et iperm_ind″.*

4.3.3.2 Compatibilité avec d'autres notions

On va d'abord étudier la compatibilité de *iperm_ind* avec *ilist_rel* On veut montrer que, comme pour *iperm_occ*, *ilist_rel* est plus fine que *iperm_ind*.

Lemme 4.29 (Compatibilité de *iperm_ind* avec *ilist_rel*)**.**

$$\forall l_1\ l_2, ilist\_rel_R\ l_1\ l_2 \Rightarrow iperm\_ind_R\ l_1\ l_2$$

Démonstration. On peut faire cette preuve en utilisant n'importe laquelle des Définitions 4.17, 4.18 ou 4.19. Nous choisissons ici de la faire avec la Définition 4.18. On veut donc montrer que $iperm\_ind'_R\ l_1\ l_2$. Soit $H : ilist\_rel_R\ l_1\ l_2$ et soient n, n_2, ln_1, ln_2 tels que $l_1 = \langle n, ln_1\rangle$ et $l_2 = \langle n_2, ln_2\rangle$. De la définition de *ilist_rel*, on peut déduire de H que $lg\ l_1 = lg\ l_2$ c'est-à-dire $n = n_2$. On a donc en fait $l_2 = \langle n, ln_2\rangle$.

On raisonne par induction sur n.

[Cas 0**]** On veut prouver que $iperm\_ind'_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle$. Ceci est vrai d'après le Lemme 4.18.

[Cas $n+1$**]** L'hypothèse d'induction est :

$$IH : \forall ln_1\ ln_2, ilist\_rel_R\ \langle n, ln_1\rangle\ \langle n, ln_2\rangle \Rightarrow iperm\_ind'_R\ \langle n, ln_1\rangle\ \langle n, ln_2\rangle$$

On a :

$$\begin{array}{l} iperm\_ind'_R\ \langle n+1, ln_1\rangle\ \langle n+1, ln_2\rangle \\ \Leftrightarrow\quad n+1 = n+1\ \wedge\ \big(\forall i_1 \exists i_2, R\ (ln_1\ i_1)\ (ln_2\ i_2) \\ \qquad \wedge\ iperm\_ind'_R\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ (remEl\ \langle n+1, ln_2\rangle\ i_2)\big) \end{array}$$

On a trivialement $n+1 = n+1$. On fixe i_1 et comme l_1 et l_2 sont équivalentes par *ilist_rel*, on prend $i_2 := i_1$. On doit donc prouver que :

1. $R\ (ln_1\ i_1)\ (ln_2\ i_1)$. D'après la définition de *ilist_rel*, on peut déduire de H les deux hypothèses suivantes : $H_1 : n = n$ et $H_2 : \forall i, R\ (ln_1\ i)\ (ln_2\ (conv_{H_1}\ i))$. On a donc $R\ (ln_1\ i)\ (ln_2\ (conv_{H_1}\ i)$. Et d'après la Propriété 3.2 et le Lemme 3.9 $conv_{H_1}\ i = i$.
2. $iperm\_ind'_R\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ (remEl\ \langle n+1, ln_2\rangle\ i_1)$ On sait d'après la Propriété 4.1.1 que

   $$lg\ (remEl\ \langle n+1, ln_1\rangle\ i_1) = lg\ (remEl\ \langle n+1, ln_2\rangle\ i_1) = n$$

   On peut donc appliquer IH. On doit maintenant prouver que :

   $$ilist\_rel_R\ (remEl\ \langle n+1, ln_1\rangle\ i_1)\ (remEl\ \langle n+1, ln_2\rangle\ i_1)$$

   Ceci se prouve en utilisant le Lemme 4.5 avec l'hypothèse H.

□

On veut également montrer que *iperm_ind* est compatible avec *imap*. Pour cela, on veut montrer qu'on peut "sortir" *imap* des *ilist* paramètres pour le passer dans la relation :

Lemme 4.30 (Compatibilité de *iperm_ind* avec *imap*)**.**

$$\forall (f_1\ f_2 : T \to U)\ l_1\ l_2, iperm\_ind_R\ (imap\ f_1\ l_1)\ (imap\ f_2\ l_2) \Leftrightarrow iperm\_ind_{\lambda t_1.\lambda t_2.R\ (f_1\ t_1)\ (f_2\ t_2)}\ l_1\ l_2$$

Démonstration. On fait la preuve avec la Définition 4.17.

[Direction $\Rightarrow$**]** L'idée première serait de faire la preuve par induction sur

$$iperm\_ind_R\ (imap\ f_1\ l_1)\ (imap\ f_2\ l_2)$$

Cependant, ici on ne manipule pas "directement" l_1 et l_2 ce qui rend cette induction délicate (on les manipule à travers *imap*). Nous allons donc plutôt faire l'induction sur la longueur de l_1. Soit donc n tel que $H : lg\ l_1 = n$. On raisonne par induction sur n. Soit $H_1 : iperm\_ind_R\ (imap\ f_1\ l_1)\ (imap\ f_2\ l_2)$

[Cas 0**]** On veut montrer que $iperm\_ind_{\lambda t_1.\lambda t_2.R\ (f_1\ t_1)\ (f_2\ t_2)}\ l_1\ l_2$. D'après la Définition 4.17 il nous suffit de montrer que :

1. $lg\ l_1 = lg\ l_2$: ce qui est vrai d'après le Lemme 4.17 et H_1.
2. $lg\ l_1 = 0$: c'est H.

[Cas $n+1$] L'hypothèse d'induction est :

$$IH : \forall l_1\ l_2, lg\ l_1 = n \wedge iperm\_ind_R(imap\ f_1\ l_1)(imap\ f_2\ l_2) \Rightarrow iperm\_ind_R\ \lambda t_1.\lambda t_2.R\ (f_1\ t_1)\ (f_2\ t_2)\ l_1\ l_2$$

D'après la Définition 4.17, H nous donne i_1 et i_2 tels que :

$$H_1 : R\ (f_1\ (fct\ l_1\ i_1))\ (f_2\ (fct\ l_2\ i_2))$$
$$H_2 : iperm\_ind_R\ (remEl\ (imap\ f_1\ l_1)\ i_1)\ (remEl\ (imap\ f_2\ l_2)\ i_2)$$

D'après la Définition 4.17 appliquée avec i_1 et i_2 il nous suffit de prouver que :

1. $R\ (f_1\ (fct\ l_1\ i_1))\ (f_2\ (fct\ l_2\ i_2))$: c'est H_1.
2. $iperm\_ind_{\lambda t_1.\lambda t_2.R\ (f_1\ t_1)\ (f_2\ t_2)}\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$: d'après IH il nous suffit de prouver que :
   (a) $lg\ (remEl\ l_1\ i_1) = n$: c'est vrai d'après la Définition 4.12.
   (b) $iperm\_ind_R\ (imap\ f_1\ (remEl\ l_1\ i_1))\ (imap\ f_2\ (remEl\ l_2\ i_2))$: grâce au Lemme 4.6 on obtient

$$H_3 : ilist\_rel_{eq}\ (remEl\ (imap\ f_1\ l_1)\ i_1)\ (imap\ f_1\ (remEl\ l_1\ i_1))$$
$$H_4 : ilist\_rel_{eq}\ (remEl\ (imap\ f_2\ l_2)\ i_2)\ (imap\ f_2\ (remEl\ l_2\ i_2))$$

   Grâce aux Lemmes 4.19 et 4.20 on peut transformer notre but en :

$$iperm\_ind_R\ (remEl\ (imap\ f_1\ l_1)\ i_1)\ (remEl\ (imap\ f_2\ l_2)\ i_2)$$

   Ce qui est vrai d'après H_2.

[Direction $\Leftarrow$] Soit $H_1 : iperm\_ind_{\lambda t_1.\lambda t_2.R\ (f_1\ t_1)\ (f_2\ t_2)}\ l_1\ l_2$. On raisonne par induction sur H_1.

[Cas de base] On a $H_2 : lg\ l_1 = lg\ l_2 = 0$. D'après la Définition 4.17 il nous suffit de prouver que $lg(imap\ f_1\ l_1) = lg(imap\ f_2\ l_2) = 0$. Ou encore, $lg\ l_1 = lg\ l_2 = 0$, ce qui est vrai d'après H_2.

[Cas inductif] On a i_1 et i_2 tels que $H_2 : R\ (f_1\ (fct\ l_1\ i_1))\ (f_2\ (fct\ l_2\ i_2))$ et l'hypothèse d'induction est : $IH : iperm\_ind_R\ (imap\ f_1\ (remEl\ l_1\ i_1))\ (imap\ f_2\ (remEl\ l_2\ i_2))$. D'après la Définition 4.17, il nous suffit de montrer que :

1. $R\ (fct\ (imap\ f_1\ l_1)\ i_1)\ (fct\ (imap\ f_2\ l_2)\ i_2)$: d'après le Lemme 3.32, ceci revient à prouver $R\ (f_1\ (fct\ l_1\ i_1))\ (f_2\ (fct\ l_2\ i_2))$, ce qui est vrai d'après H_2.
2. $iperm\_ind_R\ (remEl\ (imap\ f_1\ l_1)\ i_1)\ (remEl\ (imap\ f_2\ l_2)\ i_2)$. Grâce au Lemme 4.6 on obtient $H_2 : ilist\_rel_{eq}\ (remEl\ (imap\ f_1\ l_1)\ i_1)\ (imap\ f_1\ (remEl\ l_1\ i_1))$, qu'on peut transformer, puisque eq est symétrique, en :

$$H_2 : ilist\_rel_{eq}\ (imap\ f_1\ (remEl\ l_1\ i_1))\ (remEl\ (imap\ f_1\ l_1)\ i_1)$$

   De la même façon, on obtient

$$H_3 : ilist\_rel_{eq}\ (imap\ f_2\ (remEl\ l_2\ i_2))\ (remEl\ (imap\ f_2\ l_2)\ i_2)$$

   Grâce aux Lemmes 4.19 et 4.20 on peut transformer notre but en :

$$iperm\_ind_R\ (imap\ f_1\ (remEl\ l_1\ i_1))\ (imap\ f_2\ (remEl\ l_2\ i_2))$$

   ce qui est vrai d'après IH.

□

4.3.3.3 Décidabilité de *iperm_ind*

Comme pour *ilist_rel*, on veut prouver que *iperm_ind* préserve la décidabilité.

Lemme 4.31. $Dec\ R \Rightarrow Dec\ (iperm\_ind_R)$

Afin de prouver ce résultat, nous aurons besoin d'utiliser les deux lemmes suivants. Le premier dit que si on enlève deux éléments équivalents de deux *ilist* qui sont des permutations l'une de l'autre, alors les *ilist* résultantes sont toujours des permutations l'une de l'autre. Nous supposerons que R est une relation d'équivalence.

Lemme 4.32.

$$\forall l_1\ l_2\ i_1\ i_2,\ iperm\_ind_R\ l_1\ l_2 \wedge R_{eq}\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \Rightarrow iperm\_ind_{R_{eq}}\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$$

Idée de la preuve. Il est curieux de voir à quel point ce résultat, pourtant tout à fait naturel et presque banal, est difficile à démontrer. La principale difficulté réside dans le fait qu'on a "choisi" dans la conclusion les indices à retirer mais l'hypothèse $iperm\_ind_{R_{eq}}\ l_1\ l_2$, quelle que soit la définition qu'on choisit d'utiliser, ne nous permet pas de choisir les indices à retirer. En utilisant la Définition 4.17 les 2 indices sont arbitraires, en utilisant la Définition 4.18 ou la Définition 4.19, on choisit un indice mais pas l'autre. On va donc de nouveau avoir besoin de recourir au Lemme 4.16 (sauf bien sûr dans le cas ou "par chance", les indices fournis et les indices choisis coincident). La manipulation des indices rend cette preuve assez lourde.

Démonstration. Preuve détaillée en page 182. □

Le second lemme est simplement une adaptation de l'hypothèse de décidabilité au cas des *ilist*.

Lemme 4.33. $Dec\ R \Rightarrow \forall n\ t\ ln,\ (\exists i, R\ t\ (ln\ i)) \vee \neg(\exists i, R\ t\ (ln\ i)))$

Démonstration. Preuve détaillée en page 184. □

En utilisant les deux résultats précédents, on va pouvoir démontrer le Lemme 4.31.

Idée de la preuve. La preuve suit ici à peu de choses près le même schéma que la preuve du Lemme 3.23. On compare tout d'abord les longueurs des deux *ilist*, puis, si elles sont égales, on raisonne par induction sur cette longueur (sinon, la preuve est immédiate). On doit ensuite comparer les différents cas de figure en utilisant le Lemme 4.33 (avec t le premier élément de l_1) et l'hypothèse d'induction.

Démonstration du Lemme 4.31. On appelle H l'hypothèse *Dec R*, c'est-à-dire qu'on a :

$$H : \forall t_1\ t_2, R\ t_1\ t_2 \vee \neg(R\ t_1\ t_2)$$

Et on veut montrer : $Dec\ (iperm\_ind_R)$, c'est-à-dire

$$\forall l_1\ l_2, iperm\_ind_R\ l_1\ l_2 \vee \neg(iperm\_ind_R\ l_1\ l_2)$$

Ou encore en explicitant l_1 et l_2 :

$$\forall n_1\ n_2\ ln_1\ ln_2, iperm\_ind_R\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle \vee \neg(iperm\_ind_R\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle)$$

Comparons tout d'abord n_1 et n_2 :

[Cas $n_1 = n_2$] On va donc se passer ici de n_2 et on veut montrer que :

$$iperm\_ind_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle \vee \neg(iperm\_ind_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle)$$

On va raisonner par induction sur n_1.

[Cas 0] Ici, nous prouvons directement que $iperm\_ind_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle$ en utilisant le Lemme 4.18.

[Cas $n_1 + 1$] L'hypothèse d'induction est :

$$IH : \forall ln_1\ ln_2, iperm\_ind_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle \vee \neg(iperm\_ind_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle)$$

et nous voulons prouver que :

$$iperm\_ind_R\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle \vee \neg(iperm\_ind_R\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle)$$

Nous allons pour cela utiliser le Lemme 4.33 avec $t := ln_1\ (first\ n_1)$. Nous allons étudier les deux cas possibles :

[Cas $H_1 : \exists i,\ R\ (ln_1\ (first\ n_1))\ (ln_2\ i)$] Nous savons donc qu'il existe un équivalent à $ln_1\ (first\ n_1)$ dans ln_2, mais qu'en est-il du reste ? Cela ne nous permet pas de décider si les deux *ilist* sont des permutations l'une de l'autre ou non. Pour cela, nous devons encore analyser les deux possibilités offertes par *IH* (en prenant comme *ilistn* paramètres, ln_1 et ln_2 privées respectivement de $first\ n_1$ et de i) :

[Cas $H_2 : iperm\_ind_R\ (remEl\langle n_1 + 1, ln_1\rangle\ (first\ n_1))\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i)$] Nous voulons prouver ici que $iperm\_ind_R\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle$. En utilisant la Définition 4.17, il suffit de prouver que $\exists i_1\ i_2, R\ (ln_1\ i_1)\ (ln_2\ i_2) \wedge iperm\_ind_R\ (remEl\ \langle n_1 + 1, ln_1\rangle\ i_1)\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i_2)$. On prend donc $i_1 = first\ n_1$ et $i_2 = i$ et on obtient ce qu'on cherche en utilisant H_1 et H_2.

[Cas $H_2 : \neg(iperm\_ind_R\ (remEl\ \langle n_1 + 1, ln_1\rangle\ (first\ n_1))\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i))$] Nous voulons prouver ici que : $\neg\ (iperm\_ind_R\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle)$. Nous allons raisonner par l'absurde. Supposons que nous avons l'hypothèse H_3 : $iperm\_ind_R\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle$ et montrons que nous arrivons à une contradiction.

Appliquons le Lemme 4.32 à H_3 et H_1. On obtient que :

$$iperm\_ind_R\ (remEl\ \langle n_1 + 1, ln_1\rangle\ (first\ n_1))\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i)$$

Ce qui est en contradiction avec H_2.

[Cas $H_1 : \neg\ (\exists i, R\ (ln_1\ (first\ n_1))\ (ln_2\ i))$] Ici aussi on veut démontrer que $\neg\ (iperm\_ind_R\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle)$. Nous allons donc encore raisonner par l'absurde. Supposons donc H_3 : $iperm\_ind_R\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle$. Le Lemme 4.21 appliqué à H_3 nous donne un i tel que : $R\ (ln_1\ (first\ n_1))\ (ln_2\ i)$, ce qui est en contradiction avec l'hypothèse H_1.

[Cas $H_1 : n_1 \neq n_2$] Ici encore nous allons prouver que $\neg\ (iperm\_ind_R\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle)$. Pour cela, nous allons aussi utiliser un raisonnement par l'absurde. Nous supposons que H_2 : $iperm\_ind_R\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle$. Nous obtenons grâce au Lemme 4.17 que $n_1 = n_2$, ce qui est directement en contradiction avec H_1.

□

4.3.3.4 Monotonie de *iperm_ind*

Enfin, nous voulons montrer, comme nous l'avons fait pour *ilist_rel* que *iperm_ind* est monotone par rapport à sa relation de base.

Lemme 4.34 (*iperm_ind*$_R$ monotone par rapport à R)**.**

$$\forall R_1\ R_2\ l_1\ l_2, R_1 \subseteq R_2\ \wedge\ iperm\_ind_{R_1}\ l_1\ l_2 \Rightarrow iperm\_ind_{R_2}\ l_1\ l_2$$

Démonstration. On va utiliser ici la Définition 4.17 mais quelle que soit la définition utilisée, la preuve est très simple. Soient H_1 et H_2 les hypothèses :

$$H_1 : R_1 \subseteq R_2 \qquad \text{et} \quad H_2 :\ iperm\_ind_{R_1}\ l_1\ l_2$$

On raisonne par induction sur H_2 :

[Cas de base] On a comme nouvelle hypothèse : $H_3 : lg\ l_1 = lg\ l_2 = 0$. On veut montrer que $iperm\_ind_{R_2}\ l_1\ l_2$, ce qui est vrai d'après la Définition 4.17 et H_3.

[Cas inductif] Nous avons comme nouvelles hypothèses :

$$i_1 : Fin\ (lg\ l_1) \quad i_2 : Fin\ (lg\ l_2) \quad H_3 : R_1\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)$$

L'hypothèse d'induction est : IH : $iperm\_ind_{R_2}\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$. D'après la Définition 4.17 il nous suffit de prouver :

1. $R_2\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)$: pour prouver ceci on utilise H_1 avec l'hypothèse H_3.
2. $iperm\_ind_{R_2}\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$: c'est l'hypothèse IH.

□

4.3.4 *iperm_ind* avec squelette

Avec les Définitions 4.17, 4.18 et 4.19, nous avons spécifié quand une permutation existait. Cependant, nous n'avons pas de moyen de manipuler explicitement cette permutation : on sait qu'elle existe mais on ne la connait pas. Dans cette section, nous allons donc enrichir notre définition afin de pouvoir non seulement savoir qu'une permutation existe mais aussi la manipuler.

Nous allons donc ajouter la notion de squelette qui témoigne de l'existence d'une permutation. Ce squelette correspond à la permutation des indices relativement à la Définition 4.17. Donc, chaque squelette consiste en un n-uplet de paires d'éléments de *Fin*. Dans chaque paire, le premier élément correspond à l'indice dans la première *ilist* et le second à l'indice dans la seconde *ilist*. Ces squelettes forment le type *skel_type* défini comme une famille de types indexée par les entiers naturels :

Définition 4.21 (*skel_type*, défini récursivement)**.**

$$skel\_type\ 0 := unit$$
$$skel\_type\ (n+1) := (Fin\ (n+1) \times Fin\ (n+1)) \times skel\_type\ n$$

Remarque 4.12. *Ici, unit est un type à un élément (on pourrait prendre unit* $:=$ *Fin* 1 *mais unit n'a rien à avoir avec les indices des ilist). On note tt l'élément unique de unit.*

Avant de pouvoir vraiment définir la nouvelle notion de permutations enrichie, nous aurons besoin d'un lemme et d'une définition supplémentaires. Le lemme, qui se rapporte à la longueur de *ilist* privées d'un élément, est tout naturel et se prouve immédiatement en utilisant la Propriété 4.1.1.

Lemme 4.35.

$$\forall (l_1\ l_2 : ilist\ T)\ (H : lg\ l_1 = lg\ l_2)\ (i_1 : Fin\ (lg\ l_1))\ (i_2 : Fin\ (lg\ l_2)),\\ lg\ (remEl\ l_1\ i_1) = lg\ (remEl\ l_2\ i_2)$$

Ce que nous voulons maintenant, c'est une définition qui nous permette d'obtenir un élément de type *skel_type* $(lg\ l_1)$ à partir de deux indices i_1 et i_2 et d'un élément de type *skel_type* $(lg\ (remEl\ l_1\ i_1))$. Grossièrement, nous voulons une fonction qui mette un nouveau couple "en tête" d'un élément de type *skel_type*. Cependant, pour cela, nous aurons besoin de convertir, si $n = m$, un élément du type *skel_type n* au type *skel_type m* (c'est l'équivalent de *conv* mais sur *skel_type*). On pourrait choisir de le définir de la même façon qu'on a défini *conv* sur *Fin* : c'est-à-dire qu'on fait simplement une conversion de type. Mais pour faciliter la manipulation de ces expressions, on va définir cette conversion plus finement en convertissant individuellement chacun des éléments qui composent le squelette. Nous appelons *convSkel* la fonction qui permet d'effectuer cette conversion. Elle est définie récursivement comme suit :

Définition 4.22 (*convSkel*)**.**

$$\begin{array}{llllll} convSkel\ (n\ m : \mathbb{N}) : n = m \to skel\_type\ n \to skel\_type\ m & & & & & \\ convSkel & 0 & 0 & h & s & := & s \\ convSkel & (n+1) & (m+1) & h & (i_1, i_2, s) & := & (conv_h\ i_1, conv_h\ i_2, convSkel_{h'}\ s) \\ & & & & & & \text{avec } h' \text{ de type } n = m \text{ déduit de } h \end{array}$$

Remarque 4.13. *Dans le cas où $n = m = 0$, on peut tout simplement renvoyer s parce que unit n'a qu'un seul élément. Aucune conversion n'est donc nécessaire.*

Grâce à *convSkel*, on peut définir *skel_type_aux* qui met un couple d'indices "en tête" d'un squelette.

Définition 4.23 (*skel_type_aux*)**.**

$$skel\_type\_aux\ (l_1\ l_2 : ilist\ T)\ (H : lg\ l_1 = lg\ l_2)\ (i_1 : Fin\ (lg\ l_1)) :\\ \quad Fin\ (lg\ l_2) \to skel\_type\ (lg\ (remEl\ l_1\ i_1)) \to skel\_type\ (lg\ l_1)\\ skel\_type\_aux\ \langle n+1, ln_1\rangle\ l_2\ H\ i_1\ i_2\ s := ((i_1, conv_H\ i_2), convSkel_{H'}\ s)$$

où H' est de type $lg\ (remEl\ \langle n+1, ln_1\rangle\ i_1) = n$ et est obtenue grâce à la Propriété 4.1.1.

On peut montrer aisément pour les deux définitions précédentes l'indifférence de la preuve fournie :

Lemme 4.36 (Indifférence de la preuve pour *convSkel*)**.**

$$\forall n_1\ n_2\ s\ (h_1\ h_2 : n_1 = n_2), convSkel_{h_1}\ s = convSkel_{h_2}\ s$$

Idée de la preuve. La preuve est une simple induction sur n_1.

Lemme 4.37 (Indifférence de la preuve pour *skel_type_aux*)**.**

$$\forall l_1\ l_2\ i_1\ i_2\ s\ (h_1\ h_2 : lg\ l_1 = lg\ l_2), skel\_type\_aux\ l_1\ l_2\ h_1\ i_1\ i_2\ s = skel\_type\_aux\ l_1\ l_2\ h_2\ i_1\ i_2\ s$$

Idée de la preuve. La preuve se fait par analyse de cas sur $lg\ l_1$.

Nous pouvons maintenant définir une notion de permutations (on l'appellera *iperm_ind_skel*) utilisant cette notion de squelettes. La Définition 4.18 ne conviendrait clairement pas, mais la Définition 4.17 oui. Afin de pouvoir utiliser *skel_type*, nous avons besoin d'un entier naturel qui sera l'indice des deux composants de la paire. C'est-à-dire que nous avons besoin de savoir que les deux *ilist* ont la même taille. Nous avons donc décidé d'ajouter ceci comme hypothèse de *iperm_ind_skel*. Cela semble raisonnable dans la mesure où nous avons prouvé avec le Lemme 4.17 que c'était une conséquence directe de la Définition 4.17. Nous appelons cette nouvelle hypothèse H_{lg}. Nous pouvons donc maintenant définir *iperm_ind_skel*, en suivant la Définition 4.17 et avec les données supplémentaires :

Définition 4.24 (*iperm_ind_skel*, vu inductivement)**.**

$$\forall l_1\ l_2\ H_{lg}\ s,\ iperm\_ind\_skel_R\ l_1\ l_2\ H_{lg}\ s \Leftrightarrow \begin{cases} lg\ l_1 = 0 \quad \text{ou} \\ \exists i_1\ i_2\ s', R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)\ \wedge s = skel\_type\_aux\ l_1\ l_2\ H_{lg}\ i_1\ i_2\ s' \wedge \\ iperm\_ind\_skel_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\ H'_{lg}\ s' \end{cases}$$

où H_{lg} est la variable mentionnée précédemment qui suppose que $lg\ l_1 = lg\ l_2$ et qui est utilisée pour construire H'_{lg} de type $lg\ (remEl\ l_1\ i_1) = lg\ (remEl\ l_2\ i_2)$ avec le Lemme 4.35, s est de type $skel\_type\ (lg\ l_1)$.

Comme pour *convSkel* et *skel_type_aux*, on peut montrer l'indifférence de la preuve fournie pour *iperm_ind_skel*.

Lemme 4.38 (Indifférence de la preuve pour *iperm_ind_skel*)**.**

$$\forall l_1\ l_2\ s\ (h_1\ h_2 : lg\ l_1 = lg\ l_2), iperm\_ind\_skel_R\ l_1\ l_2\ h_1\ s \Rightarrow iperm\_ind\_skel_R\ l_1\ l_2\ h_2\ s$$

Démonstration. Preuve détaillée en page 185. □

Nous voulons maintenant montrer qu'il y a équivalence entre *iperm_ind* et *iperm_ind_skel*. En effet, si *iperm_ind_skel* est plus explicite, elle ne fait que "montrer" des informations qui restent cachées dans *iperm_ind*. Mais elle n'ajoute rien. Il est donc naturel d'attendre de ces deux définitions qu'elles soient équivalentes.

Lemme 4.39 (*iperm_ind* $\Leftrightarrow$ *iperm_ind_skel*)**.**

$$\forall l_1\ l_2\ H_{lg}, iperm\_ind_R\ l_1\ l_2 \Leftrightarrow \exists s, iperm\_ind\_skel_R\ l_1\ l_2\ H_{lg}\ s$$

Démonstration. Etant donné que *iperm_ind_skel* est défini selon le même schéma que *iperm_ind*, la preuve se fera donc en utilisant la Définition 4.17.

[Direction $\Rightarrow$] On va raisonner ici par induction sur $iperm\_ind_R\ l_1\ l_2$.

[Cas de base] On a comme nouvelle hypothèse : $H_1 : lg\ l_1 = 0$ et on veut prouver que :

$$\exists s : skel\_type\ (lg\ l_1), iperm\_ind\_skel_R\ l_1\ l_2\ H_{lg}\ s$$

On prend $s := convSkel_{sym\ H_1}\ tt$, avec $sym\ H_1$ indiquant l'égalité symétrique de H_1 (c'est-à-dire une égalité de type $0 = lg\ l_1$). Selon la Définition 4.24 il nous suffit de prouver que $lg\ l_1 = 0$, ce qui est l'hypothèse H_1.

[Cas inductif] Nous avons comme nouvelles hypothèses :

$$i_1 : Fin\ (lg\ l_1) \qquad i_2 : Fin\ (lg\ l_2) \qquad H_1 : R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)$$
$$H_2 : iperm\_ind_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$$

L'hypothèse d'induction est :

$$IH : \forall H : lg\ (remEl\ l_1\ i_1) = lg\ (remEl\ l_2\ i_2)\ \exists s : skel\_type\ (lg\ (remEl\ l_1\ i_1)),\ iperm\_ind\_skel_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\ H\ s$$

Grâce à H_{lg} et au Lemme 4.35 on obtient une nouvelle hypothèse H'_{lg} de type $lg\ (remEl\ l_1\ i_1) = lg\ (remEl\ l_2\ i_2)$. On l'utilise donc dans IH qui nous donne de nouvelles hypothèses :

$$s : skel\_type\ (lg\ (remEl\ l_1\ i_1))\ \text{et}\ H_3 : iperm\_ind\_skel_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\ H'_{lg}\ s$$

Pour rappel on veut prouver que :

$$\exists s' : skel\_type\ (lg\ l_1), iperm\_ind\_skel_R\ l_1\ l_2\ H_{lg}\ s'$$

Ce qu'on veut, c'est construire une permutation à partir des informations dont on dispose. On va donc construire s' à l'aide de s (permutation jusqu'au rang précédent), i_1 et i_2 (indices du rang courant). On utilise pour cela *skel_type_aux* et on prend : $s' := skel\_type\_aux\ l_1\ l_2\ H_{lg}\ i_1\ i_2\ s$. Il nous reste donc à montrer que : $iperm\_ind\_skel_R\ l_1\ l_2\ H_{lg}\ s'$. Cela se fait directement avec la Définition 4.24, H_1, la définition de s' et H_3.

[Direction $\Leftarrow$] La preuve ici est immédiate. Dans *iperm_ind_skel* nous avons seulement ajouté des informations par rapport à *iperm_ind*. Il suffit donc "d'enlever" ces informations supplémentaires. Cela se prouve directement avec une simple induction sur l'hypothèse $\exists s, iperm\_ind\_skel_R\ l_1\ l_2\ H_{lg}\ s$.

□

Pour la suite, il est également intéressant de montrer que *iperm_ind_skel* est monotone par rapport à son argument relation :

Lemme 4.40 ($iperm\_ind\_skel_R$ monotone par rapport à R)**.**

$$\forall R_1\ R_2\ l_1\ l_2\ H_{lg}\ s, R_1 \subseteq R_2\ \wedge\ iperm\_ind\_skel_{R_1}\ l_1\ l_2\ H_{lg}\ s \Rightarrow iperm\_ind\_skel_{R_2}\ l_1\ l_2\ H_{lg}\ s$$

Démonstration. Preuve détaillée en page 185. □

Une des situations dans lesquelles nous aurons besoin d'expliciter cette information contenue dans la Définition 4.17 de façon implicite (et rendue visible par *iperm_ind_skel*), est lorsque nous travaillerons avec une famille de relations de base (et pas seulement une relation de base unique). En particulier, nous pouvons montrer un lemme sur les intersections dans l'argument relation de *iperm_ind_skel*.

Lemme 4.41 (Intersections dans l'argument relation de *iperm_ind_skel*)**.** *Pour l_1, l_2, H_{lg} fixés et un squelette s, $iperm\_ind\_skel_R\ l_1\ l_2\ H_{lg}\ s$ commute avec des intersections arbitraires d'un ensemble de relations R. En particulier, si pour tout n, $iperm\_ind\_skel_{R_n}\ l_1\ l_2\ H_{lg}\ s$ alors $iperm\_ind\_skel_{\cap_n R_n}\ l_1\ l_2\ H_{lg}\ s$. Formellement, on exprime cela ainsi (avec R une famille de relations, de type $R : I \to$ relation T et $R'\ t_1\ t_2 \Leftrightarrow \forall i, R\ i\ t_1\ t_2$) :*

$$\forall l_1\ l_2\ (H : lg\ l_1 = lg\ l_2)\ s\ I, (\forall i, iperm\_ind\_skel_{R\ i}\ l_1\ l_2\ H\ s) \Rightarrow\ iperm\_ind\_skel_{R'}\ l_1\ l_2\ H\ s$$

Idée de la preuve. Ici, la preuve est assez simple et linéaire. Il s'agit d'une induction sur $lg\ l_1$. Elle est cependant assez longue parce qu'il y a un peu de travail sur les hypothèses (pour les "ouvrir" et voir ce qu'elles contiennent) et de réécriture. Elle ne présente cependant pas de difficulté particulière.

Démonstration. Preuve détaillée en page 186. □

4.4 Troisième méthode : fonction bijective

Les solutions proposées précédemment sont constructives et ne requièrent pas la décidabilité de R. Cependant, comme les preuves le montrent, la manipulation des indices et de la fonction *remEl*, qui impose un changement de type à chaque fois qu'elle est appliquée, rendent l'utilisation de cette fonction un peu lourde (quoique tout à fait faisable). Enfin, elles sont surtout inductives et lorsqu'on voudra les mélanger avec des définitions coinductives nous aurons de nouveau des problèmes comme on avait avec les listes (même si cette fois ci on est dans l'univers des propositions et non plus celui des ensembles).

Nous allons donc montrer ici une troisième et dernière solution, déclarative, qui dit que deux *ilist* sont des permutations l'une de l'autre si et seulement s'il existe une fonction bijective des indices de la première sur les indices de la seconde (et où chaque paire d'indices en relation indique des éléments équivalents).

On définit *iperm_bij* en utilisant la définition de *bij* donnée dans la Section 3.1.1.3 :

Définition 4.25 (*iperm_bij*)**.**

$$\forall l_1\ l_2, iperm\_bij_R\ l_1\ l_2 \Leftrightarrow \exists f\ g, bij\ f\ g\ \wedge\ (\forall i, R\ (fct\ l_1\ i)\ (fct\ l_2\ (f\ i)))$$

Remarque 4.14. *On peut noter que iperm_bij a exactement la même structure logique que ilist_rel. En fait, on peut voir ilist_rel comme un cas particulier de iperm_bij.*

Comme pour *iperm_ind*, nous allons maintenant prouver quelques propriétés de *iperm_bij*. Nous allons tout d'abord montrer que *iperm_bij* préserve l'équivalence. Pour cela, nous allons procéder comme précédemment et le faire en trois étapes : réflexivité, symétrie, transitivité.

Lemme 4.42 (*iperm_bij* préserve la réflexivité)**.** *R réflexive $\Rightarrow$ $iperm\_bij_R$ réflexive*

Démonstration. Preuve détaillée en page 187. □

Lemme 4.43 (*iperm_bij* préserve la symétrie)**.** *R symétrique $\Rightarrow$ $iperm\_bij_R$ symétrique*

Démonstration. Preuve détaillée en page 187. □

Lemme 4.44 (*iperm_bij* préserve la transitivité)**.** *R transitive* $\Rightarrow$ *iperm_bij*$_R$ *transitive*

Démonstration. Preuve détaillée en page 188. □

On peut donc maintenant énoncer le lemme final :

Lemme 4.45 (*iperm_bij* préserve l'équivalence)**.** *R équivalence* $\Rightarrow$ *iperm_bij*$_R$ *équivalence*

Démonstration. La preuve est simplement une utilisation des trois lemmes précédents. □

On peut également prouver les deux lemmes équivalents aux Lemmes 4.17 et 4.18 sur *iperm_bij* :

Lemme 4.46. $\forall l_1 l_2, iperm\_bij_R\ l_1\ l_2 \Rightarrow lg\ l_1 = lg\ l_2$

Démonstration. Preuve détaillée en page 188. □

Lemme 4.47. $\forall ln_1\ ln_2, iperm\_bij_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle$

Démonstration. Preuve détaillée en page 189. □

Enfin, comme pour les précédentes relations, on va montrer que *iperm_bij* est monotone par rapport à son argument relation :

Lemme 4.48 (*iperm_bij*$_R$ monotone par rapport à R)**.**

$$\forall R_1\ R_2\ l_1\ l_2, R_1 \subseteq R_2\ \wedge\ iperm\_bij_{R_1}\ l_1\ l_2 \Rightarrow iperm\_bij_{R_2}\ l_1\ l_2$$

Démonstration. La preuve est immédiate puisque dans la définition de *iperm_bij*, R apparaît à une position strictement positive. □

4.5 Équivalence entre *iperm_ind* et *iperm_bij*

Les dernières définitions que nous avons présentées ont les mêmes hypothèses de base (c'est-à-dire en particulier qu'elles ne demandent pas la décidabilité sur R) et réalisent le même objectif. On veut donc montrer qu'elles sont équivalentes, afin de pouvoir les utiliser indifféremment, mais aussi afin de les valider. Nous voulons donc prouver que :

Théorème 4.2. $\forall l_1\ l_2, iperm\_ind_R\ l_1\ l_2 \Leftrightarrow iperm\_bij_R\ l_1\ l_2$

Idée de la preuve.

[Direction $\Rightarrow$] Ici, la difficulté va être de trouver deux fonctions à partir de *iperm_ind* et prouver qu'elles sont bijectives. Pour obtenir ces fonctions, on va utiliser la version avec squelette, *iperm_ind_skel* (on a le droit puisqu'on a prouvé qu'elle était équivalente à *iperm_ind* avec le Lemme 4.39) et transformer le squelette en une fonction des indices de la première *ilist* vers les indices de la seconde, et en la fonction inverse. On finit la preuve (la partie $R\ (fct\ l_1\ i)\ (fct\ l_2\ (f\ i))$) par une induction sur $lg\ l_1$

[Direction $\Leftarrow$] La preuve est une simple induction sur $lg\ l_1$. La principale difficulté est de recalculer tous les indices dans les fonctions pour les nouvelles *ilist* (où un élément a été enlevé), afin d'utiliser l'hypothèse d'induction.

Démonstration. Soient n_1, n_2, ln_1 et ln_2 tels que $l_1 = \langle n_1, ln_1 \rangle$ et $l_2 = \langle n_2, ln_2 \rangle$.

[Direction $\Rightarrow$] Soit $H_1 : iperm\_ind_R\ \langle n_1, ln_1 \rangle\ \langle n_2, ln_2 \rangle$. En utilisant le Lemme 4.17, on obtient l'hypothèse suivante : $H_2 : n_1 = n_2$. On peut donc se passer de n_2. On a donc :

$$l_1 = \langle n_1, ln_1 \rangle \quad \text{et} \quad l_2 = \langle n_1, ln_2 \rangle \quad \text{et} \quad H_1 : iperm\_ind_R\ \langle n_1, ln_1 \rangle\ \langle n_1, ln_2 \rangle$$

Et on veut prouver que : $iperm\_bij_R\ \langle n_1, ln_1 \rangle\ \langle n_1, ln_2 \rangle$. Pour cela, nous aurons besoin de fournir deux fonctions. Nous allons utiliser l'équivalence entre *iperm_ind* et *iperm_ind_skel* (Lemme 4.39) et transformer le *skel_type* que nous fournit *iperm_ind_skel* en une fonction de type $Fin\ (lg\ l_1) \to Fin\ (lg\ l_2)$ et une autre de type $Fin\ (lg\ l_2) \to Fin\ (lg\ l_1)$ (comme toutes deux ont en réalité le type $Fin\ n \to Fin\ n$ nous les présentons ainsi pour mieux les différencier). Nous définissons ici la fonction qui nous permet de transformer un élément de type *skel_type n* en fonction de type $Fin\ n \to Fin\ n$:

Définition 4.26.

$$\begin{array}{lll} skel\_type\_fun\ (n : nat) : skel\_type\ n \to Fin\ n \to Fin\ n & & \\ skel\_type\_fun\ (n+1)\ ((i_1, i_2), s)\ i = i_2 & & si\ i_1 =_{Fin} i \\ skel\_type\_fun\ (n+1)\ ((i_1, i_2), s)\ i = & & \\ \qquad indexFromRemEl\ i_2\ (skel\_type\_fun\ s\ (indexInRemEl\ i_1\ i\ h)) & & sinon \end{array}$$

on ne donne pas la définition pour $n = 0$ puisque dans ce cas i n'existe pas.

skel_type_fun nous donne directement la fonction du type $Fin\ (lg\ l_1) \to Fin\ (lg\ l_2)$. Pour celle de type $Fin\ (lg\ l_2) \to Fin\ (lg\ l_1)$, il nous suffit "d'inverser" les couples de l'élément de type *skel_type* et d'appliquer *skel_type_fun*. Nous définissons donc ici une fonction qui permet d'inverser les couples :

Définition 4.27.

$$\begin{array}{llll} skel\_type\_inv\ (n : nat) : skel\_type\ n \to skel\_type\ n & & & \\ skel\_type\_inv & 0 & s & = \ s \\ skel\_type\_inv & (n+1) & ((i_1, i_2), s) & = \ ((i_2, i_1), skel\_type\_inv\ s) \end{array}$$

On peut très facilement démontrer le lemme suivant (on ne donne pas la preuve ici) :

Lemme 4.49. $\forall n\ (s : skel\_type\ n), skel\_type\_inv\ (skel\_type\_inv\ s) = s$

Nous voulons prouver que $iperm\_bij_R\ \langle n_1, ln_1 \rangle\ \langle n_1, ln_2 \rangle$, c'est-à-dire que :

$$\exists f\ g, bij\ f\ g\ \wedge\ (\forall i, R\ (fct\ l_1\ i)\ (fct\ l_2\ (f\ i)))$$

Pour trouver f et g, nous allons utiliser les deux définitions précédentes. Appliquons donc le Lemme 4.39 à H_1 (H_{lg} s'obtient immédiatement par réflexivité). Nous obtenons de nouvelles hypothèses :

$$s : skel\_type\ n \quad \text{et} \quad H_2 : iperm\_ind\_skel_R\ \langle n_1, ln_1 \rangle\ \langle n_1, ln_2 \rangle\ H_{lg}\ s$$

On prend $f := skel\_type\_fun\ s$ et $g := skel\_type\_fun\ (skel\_type\_inv\ s)$. Nous devons montrer que :

1. $bij\ (skel\_type\_fun\ s)\ (skel\_type\_fun\ (skel\_type\_inv\ s))$, c'est-à-dire que :
   (a) $\forall i, skel\_type\_fun\ s\ (skel\_type\_fun\ (skel\_type\_inv\ s)\ i) = i$: pour cela, nous énonçons le résultat dans le cas général (pour tout s) et nous l'instancierons pour ce cas-ci :
   Lemme 4.50. $\forall n\ s\ i, skel\_type\_fun\ s\ (skel\_type\_fun\ (skel\_type\_inv\ s)\ i) = i$

La preuve ici se fait par induction sur n puis par analyse de cas pour utiliser les différents cas de la Définition 4.26. Elle est assez longue et répétitive et ne présente pas d'intérêt majeur, nous ne la développons pas ici.

On utilise donc directement le Lemme 4.50 pour prouver que $\forall i, skel\_type\_fun\ s\ (skel\_type\_fun\ (skel\_type\_inv\ s)\ i) = i$.

(b) $\forall i, skel\_type\_fun\ (skel\_type\_inv\ s)\ (skel\_type\_fun\ s\ i) = i$: on va réutiliser le Lemme 4.50. Pour cela, nous devons essayer d'obtenir quelque chose de la forme : $skel\_type\_fun\ s'\ (skel\_type\_fun\ (skel\_type\_inv\ s')\ i)$. On a :

$$\begin{array}{lll} & skel\_type\_fun\ (skel\_type\_inv\ s)\ (skel\_type\_fun\ s\ i) & \\ = & skel\_type\_fun\ (skel\_type\_inv\ s) & \\ & \qquad (skel\_type\_fun\ (skel\_type\_inv\ (skel\_type\_inv\ s))\ i) & \\ & \qquad\qquad \text{(d'après Lemme 4.49)} & \\ = & i & \text{(d'après Lemme 4.50)} \end{array}$$

2. $\forall i, R\ (ln_1\ i)\ (ln_2\ (skel\_type\_fun\ s\ i)))$: on va raisonner par induction sur n_1 :

[Cas 0**]** Dans ce cas i est de type $Fin\ 0$ qui est vide.

[Cas $n_1 + 1$**]** Pour H_2, on est donc dans le deuxième cas de la Définition 4.24 (on ne peut pas avoir $lg\ l_1 = 0$ puisque $lg\ l_1 = n_1 + 1$). On a comme nouvelles hypothèses :

$$\begin{array}{c} i_1, i_2 : Fin\ n_1 \quad s' : skel\_type\ (lg\ (remEl\ \langle n_1 + 1, ln_1\rangle\ i_1)) \quad H_3 : R\ (l_1\ i_1)\ (l_2\ i_2) \\ H_4 : s = skel\_type\_aux\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle\ H_{lg}\ i_1\ i_2\ s' \\ H_5 : iperm\_ind\_skel_R\ (remEl\ \langle n_1 + 1, ln_1\rangle\ i_1)\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i_2)\ H'_{lg}\ s' \end{array}$$

Et l'hypothèse d'induction est :

$$\begin{array}{rll} IH : & \forall ln_1\ ln_2\ s\ H_{lg}, & iperm\_ind\_skel_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle\ H_{lg}\ s \\ & \Rightarrow & \forall i, R\ (l_1\ i)\ (l_2\ (skel\_type\_fun\ s\ i)) \end{array}$$

En réécrivant H_4, on doit montrer que :

$$R\ (ln_1\ i)\ (ln_2\ (skel\_type\_fun\ (skel\_type\_aux\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle\ H_{lg}\ i_1\ i_2\ s')\ i))$$

C'est-à-dire que :

$$R\ (ln_1\ i)\ (ln_2\ (skel\_type\_fun\ ((i_1, i_2), convSkel_{H'}\ s')\ i))$$

avec H' obtenue grâce à la Propriété 4.1.1. Comparons déjà i et i_1 :

[Cas $H_6 : i_1 =_{Fin} i$**]** Alors on a
- $ln_1\ i = ln_1\ i_1$ (d'après Lemme 3.9)
- $ln_2\ (skel\_type\_fun\ ((i_1, i_2), convSkel_{H'}\ s')\ i) = ln_2\ i_2$ (d'après Définition 4.26)

On veut donc montrer que $R\ (ln_1\ i_1)\ (ln_2\ i_2)$ ce qui est vrai d'après l'hypothèse H_3.

[Cas $H_6 : i_1 \neq_{Fin} i$**]** D'après la Définition 4.26, on veut montrer que

$$\begin{array}{ll} R & (ln_1\ i) \\ & (ln_2\ (indexFromRemEl\ i_2\ (skel\_type\_fun\ (convSkel_{H'}\ s') \\ & \qquad\qquad (indexInRemEl\ i_1\ i\ H_6)))) \end{array}$$

Soit $i'_2 := \mathit{skel\_type\_fun}\ (\mathit{convSkel}_{H'}\ s')\ (\mathit{indexInRemEl}\ i_1\ i\ H_6))$. On veut donc montrer que $R\ (ln_1\ i)\ (ln_2\ (\mathit{indexFromRemEl}\ i_2\ i'_2))$. Grâce au Lemme 4.13, on peut facilement montrer l'hypothèse suivante : H_7 : $i_2 \neq_{Fin} i'_2$. Pour pouvoir utiliser *IH*, nous allons transformer notre but afin d'obtenir la forme voulue. En utilisant le Lemme 4.11, on transforme notre but en :

$$\begin{array}{ll} R & (\mathit{fct}\ (\mathit{remEl}\ \langle n_1{+}1, ln_1\rangle\ i_1)\ (\mathit{indexInRemEl}\ i_1\ i\ H_6)) \\ & (\mathit{fct}\ (\mathit{remEl}\ \langle n_1{+}1, ln_2\rangle\ i_2)\ (\mathit{indexInRemEl}\ i_2\ (\mathit{indexFromRemEl}\ i_2\ i'_2)\ H_7)) \end{array}$$

Ou encore :

$$\begin{array}{ll} R & (\mathit{remEln}\ ln_1\ i_1\ (\mathit{indexInRemEl}\ i_1\ i\ H_6)) \\ & (\mathit{remEln}\ ln_2\ i_2\ (\mathit{indexInRemEl}\ i_2\ (\mathit{indexFromRemEl}\ i_2\ i'_2)\ H_7)) \end{array}$$

Grâce au Lemme 4.14, on transforme cela en :

$$R\ (\mathit{remEln}\ ln_1\ i_1\ (\mathit{indexInRemEl}\ i_1\ i\ H_6))\ (\mathit{remEln}\ ln_2\ i_2\ i'_2)$$

On termine donc en appliquant simplement *IH* avec H_5.

[Direction $\Leftarrow$ (du Théorème 4.2)] Soit H_1 : $\mathit{iperm\_bij}_R\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle$. En utilisant le Lemme 4.46, on obtient l'hypothèse suivante : H_2 : $n_1 = n_2$. On peut donc se passer de n_2. On a donc :

$$l_1 = \langle n_1, ln_1\rangle \quad \text{et} \quad l_2 = \langle n_1, ln_2\rangle \quad \text{et} \quad H_1 : \mathit{iperm\_bij}_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle$$

Et H_1 nous donne f et g tels que :

$$H_2 : \forall i, g\ (f\ i) = i \quad \text{et} \quad H_3 : \forall i, f\ (g\ i) = i \quad \text{et} \quad H_4 : \forall i, R\ (ln_1\ i)\ (ln_2\ (f\ i))$$

On veut montrer que : $\mathit{iperm\_ind}_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle$. On raisonne par induction sur n_1 :

[Cas 0**]** On applique simplement la Définition 4.17.

[Cas $n_1 + 1$**]** L'hypothèse d'induction est :

$$\begin{array}{ll} \mathit{IH} : & \forall ln_1\ ln_2\ f'\ g', (\forall i, g'\ (f'\ i) = i)\ \wedge\ (\forall i, f'\ (g'\ i) = i)\ \wedge\ (\forall i, R\ (ln_1\ i)\ (ln_2\ (f'\ i))) \\ \Rightarrow & \mathit{iperm\_ind}_R\ \langle n_1, ln_1\rangle\ \langle n_1, ln_2\rangle \end{array}$$

Selon la Définition 4.17 utilisée avec *first* n_1 et f (*first* n_1) il nous suffit de prouver que :

1. $R\ (ln_1\ (\mathit{first}\ n_1))\ (ln_2\ (f\ (\mathit{first}\ n_1)))$: on applique H_4.
2. $\mathit{iperm\_ind}_R\ (\mathit{remEl}\ \langle n_1, ln_1\rangle\ (\mathit{first}\ n_1))\ (\mathit{remEl}\ \langle n_1, ln_2\rangle\ (f\ (\mathit{first}\ n_1)))$: on veut appliquer *IH*, mais il nous manque encore les fonctions f et g. On peut montrer facilement les deux assertions suivantes :

$$\begin{array}{ll} H_5 : & \forall i, f\ (\mathit{first}\ n_1) \neq_{Fin} f\ (\mathit{succ}\ i) \\ H_6 : & \forall i, \mathit{first}\ n_1 \neq_{Fin} g\ (\mathit{indexFromRemEl}\ (f\ (\mathit{first}\ n_1))\ i) \end{array}$$

On va prendre pour fonctions

$$\begin{array}{ll} f' := & \lambda i.\mathit{indexInRemEl}\ (f\ (\mathit{first}\ n_1))\ (f\ (\mathit{succ}\ i))\ (H_5\ i) \\ g' := & \lambda i.\mathit{indexInRemEl}\ (\mathit{first}\ n_1)\ (g\ (\mathit{indexFromRemEl}\ (f\ (\mathit{first}\ n_1))\ i))\ (H_6\ i) \end{array}$$

Comme $\mathit{remEl}\ \langle n_1, ln_1\rangle\ (\mathit{first}\ n_1)$ et $\mathit{remEl}\ \langle n_1, ln_2\rangle\ (f\ (\mathit{first}\ n_1))$ sont de type $\mathit{ilistn}\ T\ n_1$, on peut maintenant utiliser *IH*. On doit prouver que :

(a) $\forall i, g'\ (f'\ i) = i$ et $\forall i, f'\ (g'\ i) = i$
Ces deux preuves se font par analyse de cas sur les valeurs des éléments de *Fin* en utilisant les différents lemmes et propriétés que nous avons introduits pour *indexInRemEl* et *indexFromRemEl*.

(b) $\forall i, R\ (fct\ (remEl\ \langle n_1, ln_1\rangle\ (first\ n_1))\ i)\ (fct\ (remEl\ \langle n_1, ln_2\rangle\ (f\ (first\ n_1))\ (f'\ i))$
On a

$$fct\ (remEl\ \langle n_1, ln_1\rangle\ (first\ n_1))\ i = ln_1\ (succ\ i)$$

Donc pour prouver l'assertion ci-dessus, il nous suffit de prouver que

$$fct\ (remEl\ \langle n_1, ln_2\rangle\ (f\ (first\ n_1))\ (f'\ i) = ln_2\ (f\ (succ\ i))$$

et d'utiliser H_4. Cette dernière preuve se fait grâce au Lemme 4.11.

□

4.6 Équivalence avec Contejean

Dans la Section 4.1.3, nous avons présenté la méthode utilisée par Contejean pour représenter les permutations sur les listes. Afin de valider et de vérifier les méthodes que nous avons proposées ici, nous allons prouver l'équivalence entre *iperm_ind* et *permutation_indec*. Comme *permutation_indec* s'applique sur les listes et que *iperm_ind* s'applique sur *ilist*, nous allons utiliser les fonctions de conversion entre *list* et *ilist* (*ilist2list* et *list2ilist*). On veut montrer que :

Lemme 4.51. $\forall l_1\ l_2, permutation\_indec_R\ l_1\ l_2 \Rightarrow iperm\_ind_R\ (list2ilist\ l_1)\ (list2ilist\ l_2)$

Idée de la preuve. On raisonne par induction sur $permutation\_indec_R\ l_1\ l_2$. Le cas de base est trivial. Le cas inductif nous donne les éléments a et b à "enlever" de l_1 et l_2 en utilisant la Définition 4.17 (il nous faudra calculer les indices correspondants).

Démonstration. Soit $H_1 : permutation\_indec_R\ l_1\ l_2$. On raisonne par induction sur H_1.

[Cas de base] On veut montrer que $iperm\_ind_R\ (list2ilist\ [])\ (list2ilist\ [])$. On utilise directement la Définition 4.17.

[Cas inductif] On a maintenant a, b, l, l'_1 et l'_2 tels que :

$$H_2 : R\ a\ b \quad H_3 : permutation\_indec_R\ l'_1\ (l@l'_2)$$

L'hypothèse d'induction est :

$$IH : iperm\_ind_R\ (list2ilist\ l'_1)\ (list2ilist\ (l@l'_2)))$$

On veut montrer que :

$$iperm\_ind_R\ (list2ilist(a::l'_1))\ (list2ilist(l@b::l'_2)))$$

On veut ici utiliser la Définition 4.17 pour "enlever" les éléments a et b et utiliser IH. L'élément a étant le premier de la liste $a::l'_1$, il est en position $first\ (length\ l'_1)$. En ce qui concerne b, il est à la $(length\ l)^{\text{ème}}$ place. Nous allons donc utiliser *code*. Pour cela, nous prouvons que : $H_4 : length\ l < lg(list2ilist(l@b::l'_2))$ (la preuve est immédiate), et b est donc à la position $code\ H_4$. On applique donc la Définition 4.17 avec $i_1 := first\ (length\ l'_1)$ et $i_2 := code\ H_4$. On doit maintenant prouver que :

1. $R\ (fct\ (list2ilist(a::l))\ (first(length\ l'_1)))\ (fct(list2ilist\ (l@b::l'_2))\ (code\ H_4))$: on introduit les deux propriétés suivantes bien connues sur *nth* :
   Propriété 4.4. $\forall l\ l'\ d\ n, n \geq length\ l \Rightarrow nth\ n\ (l@l')\ d = nth\ (n - length\ l)\ l'\ d$
   Propriété 4.5. $\forall l\ l'\ d\ n, n < length\ l \Rightarrow nth\ n\ (l@l')\ d = nth\ n\ l\ d$
   On a :

   $$fct\ (list2ilist\ (a::l'_1))\ (first\ (length\ l'_1)) = a \quad \text{(d'après Définition 3.15 et Propriété 3.3)}$$

   et

   $$\begin{array}{lll} & fct\ (list2ilist\ (l@b::l'_2))\ (code\ H_4) & \\ = & nth\ (decode\ (code\ H_4))\ (l@b::l'_2)\ b & \text{(d'après Lemme 3.28)} \\ = & nth\ (length\ l)\ (l@b::l'_2)\ b & \text{(d'après Lemme 3.8)} \\ = & nth\ (length\ l - length\ l)\ (b::l'_2)\ b & \text{(d'après Propriété 4.4)} \\ = & b & \end{array}$$

   On doit donc en fait montrer que $R\ a\ b$ ce qui est vrai d'après l'hypothèse H_2.
2. $$\begin{array}{ll} iperm\_ind_R & (remEl\ (list2ilist\ (a::l'_1))\ (first\ (length\ l'_1))) \\ & (remEl\ (list2ilist\ (l@b::l'_2))\ (code\ H_4)) \end{array}$$
   Afin d'utiliser *IH*, on va montrer que :
   (a) $Heq_1 : ilist\_rel_{eq}\ (list2ilist\ l'_1)\ (remEl\ (list2ilist\ (a::l'_1))\ (first\ (length\ l'_1)))$
   Soit $H_5 : lg\ (list2ilist\ l'_1) = lg\ (remEl\ (list2ilist\ (a::l'_1))\ (first\ (length\ l'_1)))$ (la preuve est immédiate). On applique la Définition 3.11 et on doit maintenant prouver que :

   $$\forall i, fct\ (list2ilist\ l'_1)\ i = fct\ ((remEl\ (list2ilist\ (a::l'_1))\ (first\ (length\ l'_1))))\ (conv_{H_5}\ i)$$

   Ou encore en simplifiant d'après les définitions (un travail où les ordinateurs sont efficaces) :

   $$fct\ (list2ilist\ l'_1)\ i = nth\ (decode\ i)\ l'_1\ a$$

   On applique directement le Lemme 3.28.
   (b) $Heq_2 : ilist\_rel_{eq}\ (list2ilist(l@l'_2))\ (remEl\ (list2ilist\ (l@b::l'_2))\ (code\ H_4))$
   Soit $H_5 : lg\ (list2ilist\ (l@l'_2)) = lg\ (remEl\ (list2ilist\ (l@b::l'_2))\ (code\ H_4))$ (la preuve est immédiate). On applique la Définition 3.11 et on doit maintenant prouver que :

   $$\forall i, fct\ (list2ilist\ (l@l'_2))\ i = fct\ (remEl\ (list2ilist\ (l@b::l'_2))\ (code\ H_4))\ (conv_{H_5}\ i)$$

   On a :

   $$fct\ (list2ilist\ (l@l'_2))\ i \quad = \quad nth\ (decode\ i)\ (l@l'_2)\ b \quad \text{(d'après le Lemme 3.28)}$$

   Pour $fct\ (remEl\ (list2ilist\ (l@b::l'_2))\ (code\ H_4))\ (conv_{H_5}\ i)$ comparons les valeurs de $code\ H_4$ et de $conv_{H_5}\ i$:
   [Cas $H_6 : code\ H_4 \leq_{Fin} conv_{H_5}\ i$**]** On remarque que $H_6 \Leftrightarrow length\ l \leq_{Fin} i$. On a :

   $$\begin{array}{ll} & fct\ (remEl\ (list2ilist\ (l@b::l'_2))\ (code\ H_4))\ (conv_{H_5}\ i) & \\ = & fct\ (list2ilist\ (l@b::l'_2))\ (conv_{H_7}\ (succ\ (conv_{H_5}\ i))) & \\ & \text{(d'après la Propriété 4.1.3 et avec } H_7 \text{ de type} & \\ & lg\ (remEl\ (list2ilist\ (l@b::l'_2))\ (code\ H_4)) = lg\ (list2ilist\ (l@b::l'_2))) & \\ = & nth\ (decode\ (conv_{H_7}\ (succ\ (conv_{H_5}\ i))))\ (l@b::l'_2)\ b & \text{(d'après Lemme 3.28)} \\ = & nth\ (decode\ i + 1)\ (l@b::l'_2)\ b & \text{(d'après Propriété 3.2)} \\ = & nth\ (decode\ i + 1 - length\ l)\ (b::l'_2)\ b & \text{(d'après Propriété 4.4)} \\ = & nth\ (decode\ i - length\ l)\ l'_2\ b & \end{array}$$

De la même façon, on a :

$$nth\ (decode\ i)\ (l@l'_2)\ b = nth\ (decode\ i - length\ l)\ l'_2\ b \quad \text{(d'après Propriété 4.4)}$$

[Cas $conv_{H_5}\ i <_{Fin} code_{H_4}$**]** On remarque que $H_6 \Leftrightarrow i <_{Fin} length\ l$. On a :

$$\begin{array}{ll}
& fct\ (remEl\ (list2ilist\ (l@b::l'_2))\ (code\ H_4))\ (conv_{H_5}\ i) \\
= & fct\ (list2ilist\ (l@b::l'_2))\ (conv_{H_7}\ (weakFin\ (conv_{H_5}\ i))) \\
& \quad \text{(d'après la Propriété 4.1.2 et avec } H_7 \text{ de type} \\
& \quad lg\ (remEl\ (list2ilist\ (l@b::l'_2))\ (code\ H_4)) = lg\ (list2ilist\ (l@b::l'_2))) \\
= & nth\ (decode\ (conv_{H_7}\ (weakFin\ (conv_{H_5}\ i))))\ (l@b::l'_2)\ b \quad \text{(Lemme 3.28)} \\
= & nth\ (decode\ i)\ (l@b::l'_2)\ b \quad \text{(Propriété 3.2 et Lemme 4.3)} \\
= & nth\ (decode\ i)\ l\ b \quad \text{(Propriété 4.5)}
\end{array}$$

De la même façon, on a :

$$nth\ (decode\ i)\ (l@l'_2)\ b = nth\ (decode\ i)\ l\ b \quad \text{(d'après Propriété 4.5)}$$

En utilisant les Lemmes 4.19 et 4.20 avec Heq_1 et Heq_2 il nous reste à prouver que :

$$iperm\_ind_R\ (list2ilist\ l'_1)\ (list2ilist\ (l@l'_2))$$

ce qui est vrai d'après IH.

□

On veut également montrer l'autre sens de l'équivalence (mais avec la conversion inverse) :

Lemme 4.52. $\forall l_1\ l_2, iperm\_ind_R\ l_1\ l_2 \Rightarrow permutation\_indec_R\ (ilist2list\ l_1)\ (ilist2list\ l_2)$

Idée de la preuve. On raisonne par induction sur $iperm\_ind'_R\ l_1\ l_2$. La principale difficulté est d'obtenir des *ilist* de la forme $a::l$ et $l'_1@b::l'_2$. Pour le premier cas, c'est facile : on l'obtient directement avec la Définition 4.18 (la Définition 4.17 ne suffirait pas) en prenant pour i_1 le premier élément de l_1. Mais pour le second cas, on a besoin des notions de parties gauche et droite présentées Section 3.2.6 et des lemmes que nous y avons démontrés.

Démonstration. Soit H_1 : $iperm\_ind_R\ l_1\ l_2$. On veut ici utiliser la Définition 4.18. Grâce au Théorème 4.1, on transforme H_1 en H_1 : $iperm\_ind'_R\ l_1\ l_2$. On va raisonner par induction sur H_1. On a comme nouvelle hypothèse : H_2 : $lg\ l_1 = lg\ l_2$ et l'hypothèse d'induction est :

$$\begin{array}{lll}
IH : & \forall i_1 \exists i_2, & R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)\ \wedge\ iperm_ind'_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2) \\
& \wedge & permutation_indec_R\ (ilist2list\ (remEl\ l_1\ i_1))\ (ilist2list\ (remEl\ l_2\ i_2))
\end{array}$$

Soient n_1, n_2, ln_1, ln_2 tels que $l_1 = \langle n_1, ln_1 \rangle$ et $l_2 = \langle n_2, ln_2 \rangle$. Grâce à H_2 on a $n_1 = n_2$. On peut donc se passer de n_2 et on a maintenant :

$$\begin{array}{lll}
IH : & \forall i_1 \exists i_2, & R\ (ln_1\ i_1)\ (ln_2\ i_2)\ \wedge\ iperm_ind'_R\ (remEl\ \langle n_1, ln_1 \rangle\ i_1)\ (remEl\ \langle n_1, ln_2 \rangle\ i_2) \\
& \wedge & permutation_indec_R\ (ilist2list\ (remEl\ \langle n_1, ln_1 \rangle\ i_1))\ (ilist2list\ (remEl\ \langle n_1, ln_2 \rangle\ i_2))
\end{array}$$

et on veut montrer que : $permutation\_indec_R\ (ilist2list\ \langle n_1, ln_1 \rangle)\ (ilist2list\ \langle n_1, ln_2 \rangle)$. Nous allons chercher à appliquer l'un ou l'autre des cas de la définition. Pour cela, analysons les valeurs possibles pour n_1 :

[Cas 0**]** On applique directement la Définition 4.7.

[Cas $n_1 + 1$**]** Pour pouvoir utiliser la Définition 4.7, il faut qu'on obtienne des listes de la forme $a::l$ et $l_1@b::l_2$. Si on utilise *IH* avec *first* n_1, on obtient i_2 tel que :

$$\begin{array}{ll} H_3: & R\ (ln_1\ (\mathit{first}\ n_1))\ (ln_2\ i_2) \\ H_4: & \mathit{iperm\_ind}'_R\ (\mathit{remEl}\ \langle n_1+1, ln_1\rangle\ (\mathit{first}\ n_1))\ (\mathit{remEl}\ \langle n_1+1, ln_2\rangle\ i_2) \\ H_5: & \mathit{permutation\_indec}_R\ (\mathit{ilist2list}\ (\mathit{remEl}\ \langle n_1+1, ln_1\rangle\ (\mathit{first}\ n_1))) \\ & \qquad (\mathit{ilist2list}\ (\mathit{remEl}\ \langle n_1+1, ln_2\rangle\ i_2)) \end{array}$$

On a directement

$$\mathit{ilist2list}\ \langle n_1+1, ln_1\rangle = ln_1(\mathit{first}\ n_1)::\mathit{ilist2list}(\mathit{remEl}\ \langle n_1+1, ln_1\rangle\ (\mathit{first}\ n_1))$$

On veut donc maintenant montrer qu'il existe l_1, b et l_2 tels que :

$$\mathit{ilist2list}\ \langle n_1+1, ln_2\rangle = l_1@b::l_2$$

Pour cela, on va avoir besoin de la notion de parties gauche et droite d'une *ilist* (on pourrait tout aussi bien raisonner sur les listes, mais on a choisi ici de le faire sur *ilist*, notre développement a montré que la solution en raisonnant sur les listes était un peu plus compliquée que celle en raisonnant sur *ilist*). On va utiliser les définitions de *ileft* et *iright* données en Section 3.2.6 et on va "couper" $\langle n_1+1, ln_2\rangle$ en i_2. D'après le Lemme 3.42 on a :

$$\begin{array}{l} \mathit{ilist2list}\ \langle n_1+1, ln_2\rangle = \\ \quad (\mathit{ilist2list}\ (\mathit{ileft}\ \langle n_1+1, ln_2\rangle\ i_2))@(ln_2\ i_2)::\mathit{ilist2list}\ (\mathit{iright}\ \langle n_1+1, ln_2\rangle\ i_2) \end{array}$$

Ce qui nous donne la forme voulue. On peut donc appliquer la Définition 4.7 et on doit maintenant prouver que :

1. $R\ (ln_1\ (\mathit{first}\ n_1))\ (ln_2\ i_2)$: ce qui est vrai d'après H_3
2. $\begin{array}{ll} \mathit{permutation\_indec}_R & (\mathit{ilist2list}(\mathit{remEl}\ \langle n_1{+}1, ln_1\rangle\ (\mathit{first}\ n_1))) \\ & (\mathit{ilist2list}\ (\mathit{ileft}\ \langle n_1{+}1, ln_2\rangle\ i_2))@(\mathit{ilist2list}\ (\mathit{iright}\ \langle n_1{+}1, ln_2\rangle\ i_2) \end{array}$

   D'après le Corollaire 4.8, on a :

   $$\begin{array}{l} \mathit{ilist2list}\ (\mathit{ileft}\ \langle n_1+1, ln_2\rangle\ i_2))@(\mathit{ilist2list}\ (\mathit{iright}\ \langle n_1+1, ln_2\rangle\ i_2) \\ = \mathit{ilist2list}\ (\mathit{remEl}\ \langle n_1+1, ln_2\rangle\ i_2) \end{array}$$

   On doit donc prouver :

   $$\begin{array}{ll} \mathit{permutation\_indec}_R & (\mathit{ilist2list}(\mathit{remEl}\ \langle n_1+1, ln_1\rangle\ (\mathit{first}\ n_1))) \\ & (\mathit{ilist2list}\ (\mathit{remEl}\ \langle n_1+1, ln_2\rangle\ i_2)) \end{array}$$

   Ce qui est vrai d'après H_5.

□

On peut montrer les autres possibilités en utilisant les deux lemmes précédents et l'équivalence entre *ilist* et *list*. On les énonce ici sans expliciter les démonstrations :

Lemme 4.53. $\forall l_1\ l_2, \mathit{iperm\_ind}_R\ (\mathit{list2ilist}\ l_1)\ (\mathit{list2ilist}\ l_2) \Rightarrow \mathit{permutation\_indec}_R\ l_1\ l_2$

Lemme 4.54. $\forall l_1\ l_2, \mathit{permutation\_indec}_R\ (\mathit{ilist2list}\ l_1)\ (\mathit{ilist2list}\ l_2) \Rightarrow \mathit{iperm\_ind}_R\ l_1\ l_2$

Remarque 4.15. *A partir de permutation_indec, on peut donc définir une nouvelle relation de permutations sur les ilist :*

Définition 4.28 (*iperm_cont*)**.**

$$\forall l_1\ l_2, iperm\_cont_R\ l_1\ l_2 \Leftrightarrow permutation\_indec_R\ (ilist2list\ l_1)\ (ilist2list\ l_2)$$

Grâce aux Lemmes 4.52 et 4.54, on sait que iperm_cont est équivalente à iperm_ind et donc aussi à iperm_bij.

Cela nous permet donc de valider définitivement toutes ces définitions.

4.7 Comparaison des définitions de permutations

Dans ce chapitre, nous avons présenté sept notions différentes de permutations sur *ilist*. Comme nous l'avons déjà dit, la première, *iperm_occ*, mélange des aspects déclaratifs et algorithmiques. Ces derniers nécessitent la décidabilité sur la relation de base, ce qui, comme nous l'avons expliqué, rend son utilisation impossible pour nos besoins.

Nous avons démontré ensuite que les six autres représentations étaient extensionnellement équivalentes, bien qu'elles restent conceptuellement assez différentes.

Les trois premières, les Définitions 4.17, 4.18 et 4.19, sont déclaratives parce qu'elles ne disent pas ce qu'est une permutation et elles sont directes parce qu'elles utilisent uniquement les concepts des *ilist*. Nous considérons que la Définition 4.17 est la définition qui correspond le mieux à l'intuition que l'on a des permutations. C'est la définition que nous avons choisie comme étant notre définition de référence. Les Définitions 4.18 et 4.19 partagent la même intuition d'enlever recursivement des couples d'éléments, mais avec une uniformité supplémentaire qui donne la possibilité de choisir le couple. Cependant, cette uniformité implique également une redondance qui n'était pas présente dans la Définition 4.17. Comme nous l'avons expliqué précédemment, ces définitions se prêtent bien à la preuve de la transitivité mais la façon dont elles sont construites manque de symétrie. Or, la symétrie fait, pour nous, aussi partie de l'intuition derrière les permutations. C'est pour ces raisons que nous avons choisi la Définition 4.17 comme relation de référence.

La variante avec les squelettes, *iperm_ind_skel*, ne fait que rendre explicites des informations contenues dans la Définition 4.17 mais de façon cachée. Il devient nécessaire de montrer cette information lorsqu'on considère plusieurs relations de base comme dans le Lemme 4.41. Les squelettes représentent des fonctions bijectives sur les indices mais cette information reste implicite. En effet, chaque indice correspond à une situation intermédiaire : ils ne font pas référence directement à la *ilist* de départ, mais à celle-ci privée d'un certain nombre de ses éléments.

La Définition 4.25 en revanche, est basée sur une représentation explicite de fonctions bijectives sur les indices. La relation est vraie si les éléments pointés par les couples d'indices sont équivalents par rapport à la relation de base. Conceptuellement, cela sépare le concept en deux : il faut d'abord donner tous les couples d'indices et ensuite prouver que les éléments pointés sont équivalents. Bien que cette solution soit assez simple, nous la trouvons moins intuitive et élégante que la Définition 4.17. Elle a cependant l'énorme avantage, par rapport à la Définition 4.17, de ne pas être inductive et donc de ne pas être problématique à mélanger avec une définition coinductive.

Enfin, la Définition 4.28 s'appuie sur une représentation des permutations basée sur les listes. Or, comme nous l'avons expliqué, nous ne pouvons pas nous permettre d'utiliser les listes dans nos utilisations futures (parce que ces utilisations seront coinductives et qu'il est difficile de mélanger induction et coinduction dans Coq). Nous préférons donc utiliser une définition qui s'appuie directement sur *ilist* plutôt qu'une définition qui utilise une conversion des listes vers *ilist*. De plus, comme nous l'avons déjà dit, la manière "naturelle" de manipuler les listes est assez différente de la manière naturelle de manipuler les *ilist*. Or, évidemment, *iperm_cont* est basé sur les principes de tête, queue et concaténation de listes qui ne sont pas des concepts immédiats sur *ilist*. Cela rend donc *iperm_cont* encore plus difficile d'utilisation pour nous.

4.8 Transposition aux permutations sur les listes

Comme nous l'avons montré dans la Section 4.1, les possibilités offertes par la librairie standard de Coq en matière de représentation de permutations sur les listes sont assez limitées, en particulier si on veut pouvoir utiliser une relation de base arbitraire. Comme nous l'avons présenté, la seule méthode que nous avons trouvée (et qui ne fait pas partie de la librairie standard) est celle de Contejean [24]. Nous avons donc voulu transposer les représentations que nous avions sur les listes, surtout celles qui pour nous représentaient le mieux l'intuition de ce qu'est une permutation. Nous avons donc transposé sur les listes les Définitions 4.17 et 4.18 (4.19 étant très similaire à 4.18 nous l'avons omise). Pour définir l'équivalent de *remEl* sur les listes, nous avons utilisé les notions de partie gauche et partie droite d'une liste, les fonctions *firstn* et *skipn* de la librairie standard de Coq. Comme nous avons transposé ces notions, nous nous sommes également beaucoup servi de la fontion *nth*.

Puis nous avons montré que ces deux notions étaient équivalentes entre elles et avec la représentation proposée par Contejean. Les preuves suivent le même schéma que celles montrées précédemment (en particulier, nous avons défini des équivalents pour *indexInRemEl*, *indexFromRemEl* et les propriétés associées). Mais comme il n'y a pas autant de travail sur les indices à faire qu'avec *ilist* (en particulier pas de conversion de types, qui rendent les développements sur *ilist* parfois longs et fastidieux), les preuves sont beaucoup plus simples et élégantes. Elles ne sont cependant pas détaillées ici.

Tout ceci nous permet d'avoir aujourd'hui trois façons différentes (et équivalentes) de représenter les permutations sur les listes (et toutes les trois prennent en paramètre leur relation de base et ne lui demandent pas d'être décidable).

Le développement de cette petite bibliothèque est élégant et étonnamment court. On obtient de jolies notions, facilement manipulables.

Troisième partie

Une représentation coinductive des graphes

5 Des graphes ordonnés, enracinés, connexes

NOTRE objectif est de représenter des graphes. Comme nous l'avons dit à la Section 2.1.3, la représentation canonique avec les listes ne peut pas être utilisée. Nous abandonnons donc cette représentation et remplaçons les listes par les *ilist* que nous avons définies au Chapitre 3, dans ce but. Nous définirons quelques outils sur cette nouvelle représentation et analyserons ses points forts et ses points faibles.

5.1 Définition de *Graph*

On définit donc *Graph* comme proposé dans la Section 2.1.3 mais en utilisant *ilist* à la place de *list*.

Définition 5.1 (*Graph*)**.**
$$\frac{t : T \qquad l : \mathit{ilist}\ (\mathit{Graph}\ T)}{\mathit{mk\_Graph}\ t\ l : \mathit{Graph}\ T}$$

On peut instancier facilement les mêmes exemples que précédemment.

Exemple 5.1 (Exemple de la Figure 2.5)**.** *On représente le graphe de la Figure 2.5 comme suit :*

$$\mathit{Leaf} := \mathit{mk\_Graph}\ 0\ [\![]\!]$$

Exemple 5.2 (Exemple de la Figure 2.6)**.** *On représente le graphe de la Figure 2.6 comme suit :*

$$\mathit{Finite\_Graph} := \mathit{mk\_Graph}\ 0\ [\![\mathit{mk\_Graph}\ 1\ [\![\mathit{Finite\_Graph}]\!]]\!]$$

Exemple 5.3 (Exemple de la Figure 2.7)**.** *On représente la famille des graphes de la Figure 2.7 comme suit :*

$$\mathit{Infinite\_Graph}_n := \mathit{mk\_Graph}\ n\ [\![\mathit{Infinite\_Graph}_{n+1}]\!]$$

Le graphe de la Figure 2.7 correspond à $\mathit{Infinite\_Graph}_0$.

On veut maintenant définir, comme précédemment *applyF2G* :

Définition 5.2 (*applyF2G*, définie corécursivement)**.**

$$\begin{aligned}&\mathit{applyF2G} : \forall T\ U,\ (T \to U) \to \mathit{Graph}\ T \to \mathit{Graph}\ U\\ &\mathit{applyF2G}\ f\ (\mathit{mk\_Graph}\ t\ l) := \mathit{mk\_Graph}\ (f\ t)\ \big(\mathit{imap}\ (\mathit{applyF2G}\ f)\ l\big)\end{aligned}$$

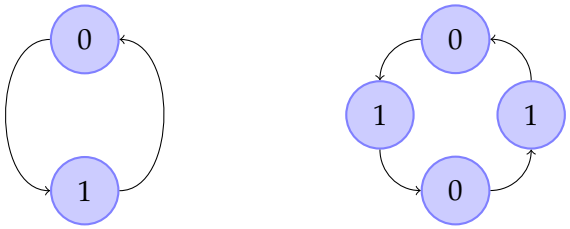
Cette fois, la définition est acceptée par Coq pour les raisons que nous avons déjà évoquées Section 3.2.4.1 (*imap* cache en fait un constructeur ce qui satisfait la condition de garde).

On définit également sur *Graph* les deux projections *label* and *sons* telles que $\mathit{label}\ (\mathit{mk\_Graph}\ t\ l) = t$ et $\mathit{sons}\ (\mathit{mk\_Graph}\ t\ l) = l$ et que le lemme suivant soit correct :

Lemme 5.1. $\forall g,\ g = \mathit{mk\_Graph}\ (\mathit{label}\ g)\ (\mathit{sons}\ g)$

Ici on a le droit d'utiliser l'égalité de Leibniz pour comparer des éléments de *Graph* puisqu'ils sont égaux par construction et pas seulement bisimulés. Cependant, cela n'est pas toujours le cas. C'est ce que nous allons voir maintenant.

5.2 Bisimilarité sur *Graph*



(a) Exemple de la Figure 2.6

(b) Exemple de la Figure 2.6 déplié une fois

Figure 5.1 — Exemple de graphes équivalents mais non égaux

Les éléments de *Graph* étant coinductifs, on ne peut pas toujours utiliser l'égalité de Leibniz pour les comparer : elle est trop fine et ne peut pas être établie par coinduction. On a besoin d'une relation de bisimilarité. Pour illustrer ceci, la Figure 5.1 montre une situation dans laquelle deux graphes ne sont pas égaux vis-à-vis de l'égalité de Leibniz alors que nous voulons qu'ils soient équivalents. En effet, ces deux graphes sont différents (graphiquement, cela est évident) mais l'arbre infini régulier résultant est le même dans les deux cas. Ils représentent donc le même élément mais ont des représentations (syntaxiques) différentes. Si on voulait différencier les nœuds 0 (resp. 1), il faudrait utiliser un type plus riche.

Pour comparer deux éléments de *Graph*, nous aurons besoin, comme pour *ilist*, de comparer leurs deux projections. Les étiquettes (projection *label*) sont comparées par une relation R sur T passée en paramètre. Les fils (projection *sons*), représentés par des *ilist*, sont comparés par la relation *ilist_rel* définie sur *ilist*. Comme le paramètre de type pour les *ilist* est *Graph T*, *ilist* prendra en paramètre la relation sur *Graph* qu'on est en train de définir. Cette relation doit donc être définie coinductivement. On appelle *Geq* cette relation et elle est définie comme suit :

Définition 5.3 (*Geq*, vue coinductivement)**.**

$$\frac{g_1\ g_2 : \mathit{Graph}\ T \qquad R\ (\mathit{label}\ g_1)\ (\mathit{label}\ g_2) \qquad \mathit{ilist\_rel}_{\mathit{Geq}_R}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)}{\mathit{Geq}_R\ g_1\ g_2}$$

A titre d'exemple et de première validation (puisqu'on a dit que c'est ce qu'on recherchait), on va montrer que les deux graphes de la Figure 5.1 sont effectivement équivalents par rapport à *Geq* (avec *eq* comme relation de base). Le graphe de la Figure 5.1(b) est défini par :

$$\mathit{Finite\_Graph}' := \mathit{mk\_Graph}\ 0\ [\![\mathit{mk\_Graph}\ 1\ [\![\mathit{mk\_Graph}\ 0\ [\![\mathit{mk\_Graph}\ 1\ [\![\mathit{Finite\_Graph}']\!]]\!]]\!]]\!]$$

Lemme 5.2. $\mathit{Geq}_{eq}\ \mathit{Finite\_Graph}\ \mathit{Finite\_Graph}'$

Démonstration. On raisonne par coinduction. L'hypothèse de coinduction est CH : $\mathit{Geq}_{eq}\ \mathit{Finite\_Graph}\ \mathit{Finite\_Graph}'$. On applique la Définition 5.3. On doit prouver que

1. $\mathit{label}\ \mathit{Finite\_Graph} = \mathit{label}\ \mathit{Finite\_Graph}'$: c'est évident

2. $ilist\_rel_{eq}$ $(sons\ Finite\_Graph)$ $(sons\ Finite\_Graph')$. On montre aisément que $lg(sons\ Finite\_Graph) = lg(sons\ Finite\_Graph') = 1$. On peut donc appliquer la Définition 3.11. On doit montrer que

$$Geq_{eq}\ (fct\ (sons\ Finite\_Graph)\ (first\ 0))\ (fct\ (sons\ Finite\_Graph')\ (first\ 0))$$

On applique alors de nouveau la Définition 5.3, puis on fait le même genre de preuves que ce qu'on vient de faire (quatre fois au total, en comptant la première), jusqu'à revenir à $Geq_{eq}\ Finite\_Graph\ Finite\_Graph'$ qu'on prouve avec CH.

□

Montrons maintenant que *Geq* préserve l'équivalence. Pour cela, encore une fois, on va montrer qu'elle préserve la réflexivité, la symétrie et la transitivité.

Lemme 5.3 (*Geq* préserve la reflexivité)**.** $R\ réflexive \Rightarrow \forall g, Geq_R\ g\ g$

Démonstration. On raisonne par coinduction. Soit CH l'hypothèse de coinduction : $\forall g, Geq_R\ g\ g$. Soient t et l tels que $g = mk\_Graph\ t\ l$. On veut prouver que

$$Geq_R\ (mk\_Graph\ t\ l)\ (mk\_Graph\ t\ l)$$

En appliquant la Définition 5.3 on doit prouver que :

1. $R\ t\ t$: on prouve cela en utilisant la réflexivité de R
2. $ilist\_rel\ Geq_R\ l\ l$: ici on voudrait utiliser la préservation de la réflexivité de *ilist_rel*. Cependant, Coq ne l'accepte pas (il n'ouvre pas assez la preuve pour savoir ce qu'elle contient, même si elle n'est pas inductive dans ce cas). On doit donc la refaire (et ce sera la même chose pour la symétrie et la transitivité). On sait que $H : lg\ l = lg\ l$ par refléxivité. On applique donc la Définition 3.11 et on doit prouver que :

$$\forall i, Geq_R\ (fct\ l\ i)\ (fct\ l\ (conv_H\ i))$$

C'est-à-dire que : $\forall i, Geq_R\ (fct\ l\ i)\ (fct\ l\ i)$. Ceci se prouve avec CH.

□

Lemme 5.4 (*Geq* préserve la symétrie)**.** $R\ symétrique \Rightarrow (\forall g_1\ g_2, Geq_R\ g_1\ g_2 \Rightarrow Geq_R\ g_2\ g_1)$

Démonstration. On raisonne par coinduction. Soit CH l'hypothèse de coinduction : $\forall g_1\ g_2, Geq_R\ g_1\ g_2 \Rightarrow Geq_R\ g_2\ g_1$. Et soit $H : Geq_R\ g_1\ g_2$. Soient t_1, l_1, t_2 et l_2 tels que $g_1 = mk\_Graph\ t_1\ l_1$ et $g_2 = mk\_Graph\ t_2\ l_2$. On veut prouver que

$$Geq_R\ (mk\_Graph\ t_2\ l_2)\ (mk\_Graph\ t_1\ l_1)$$

A partir de H et de la Définition 5.3, on obtient deux nouvelles hypothèses :

$$H_1 : R\ t_1\ t_2 \quad H_2 : ilist\_rel_{Geq_R}\ l_1\ l_2$$

En appliquant la Définition 5.3 on doit prouver que :

1. $R\ t_2\ t_1$: on prouve cela en utilisant la symétrie de R sur H_1.

2. *ilist_rel* $Geq_R\ l_2\ l_1$: ici non plus on ne peut pas utiliser la symétrie de *ilist_rel* directement. Grâce à la Définition 3.11 et H_2, on obtient deux nouvelles hypothèses :

$$H_3 : lg\ l_1 = lg\ l_2 \quad H_4 : \forall i, Geq_R\ (fct\ l_1\ i)\ (fct\ l_2\ (conv_{H_3}\ i))$$

On applique la Définition 3.11 avec le symétrique de H_3 ($sym\ H_3$). On doit prouver que :

$$\forall i, Geq_R\ (fct\ l_2\ i)\ (fct\ l_1\ (conv_{(sym\ H_3)}\ i))$$

On montre aisément que $i = conv_{H_3}\ (conv_{(sym\ H_3)}\ i)$. On doit donc prouver que :

$$Geq_R\ (fct\ l_2\ (conv_{H_3}\ (conv_{(sym\ H_3)}\ i)))\ (fct\ l_1\ (conv_{(sym\ H_3)}\ i))$$

Ce qu'on prouve avec CH et H_4 (instanciée avec $conv_{(sym\ H_3)}\ i$).

□

Lemme 5.5 (*Geq* préserve la transitivité)**.**

$$R\ transitive \Rightarrow (\forall g_1\ g_2\ g_3, Geq_R\ g_1\ g_2 \wedge Geq_R\ g_2\ g_3 \Rightarrow Geq_R\ g_1\ g_3)$$

Démonstration. On raisonne par coinduction. Soit CH l'hypothèse de coinduction : $\forall g_1\ g_2\ g_3, Geq_R\ g_1\ g_2 \wedge Geq_R\ g_2\ g_3 \Rightarrow Geq_R\ g_1\ g_3$. Et soient $H_1 : Geq_R\ g_1\ g_2$ et $H_2 : Geq_R\ g_2\ g_3$. Soient t_1, l_1, t_2, l_2, t_3 et l_3 tels que $g_1 = mk\_Graph\ t_1\ l_1$, $g_2 = mk\_Graph\ t_2\ l_2$ et $g_3 = mk\_Graph\ t_3\ l_3$. On veut prouver que

$$Geq_R\ (mk\_Graph\ t_1\ l_1)\ (mk\_Graph\ t_3\ l_3)$$

A partir de H_1 et H_2 et de la Définition 5.3, on obtient quatre nouvelles hypothèses :

$$H_3 : R\ t_1\ t_2 \quad H_4 : ilist\_rel_{Geq_R}\ l_1\ l_2 \quad H_5 : R\ t_2\ t_3 \quad H_6 : ilist\_rel_{Geq_R}\ l_2\ l_3$$

En appliquant la Définition 5.3 on doit prouver que :

1. $R\ t_1\ t_3$: on prouve cela en utilisant la transitivité de R avec H_3 et H_5.
2. *ilist_rel* $Geq_R\ l_1\ l_3$: on va ici encore devoir refaire la preuve de la transitivité de *ilist_rel*. H_4 et H_6 nous donnent quatre nouvelles hypothèses (grâce à la Définition 3.11) :

$$H_7 : lg\ l_1 = lg\ l_2 \quad H_8 : \forall i, Geq_R\ (fct\ l_1\ i)\ (fct\ l_2\ (conv_{H_7}\ i))$$
$$H_9 : lg\ l_2 = lg\ l_3 \quad H_{10} : \forall i, Geq_R\ (fct\ l_2\ i)\ (fct\ l_3\ (conv_{H_9}\ i))$$

On applique la Définition 3.11 avec $trans\ H_7\ H_9$ (transitif de H_7 et H_9), on doit prouver :

$$\forall i, Geq_R\ (fct\ l_1\ i)\ (fct\ l_3\ (conv_{(trans\ H_7\ H_9)}\ i))$$

On montre aisément que $conv_{(trans\ H_7\ H_9)}\ i = conv_{H_9}\ (conv_{H_7}\ i)$. On doit donc prouver :

$$Geq_R\ (fct\ l_1\ i)\ (fct\ l_3\ (conv_{H_9}\ (conv_{H_7}\ i)))$$

Ce qu'on prouve avec CH, H_8 et H_{10}. □

On peut donc maintenant énoncer le lemme final :

Lemme 5.6 (*Geq* préserve l'équivalence)**.** *R équivalence* $\Rightarrow$ Geq_R *équivalence*

Démonstration. La preuve est simplement une utilisation des trois lemmes précédents. □

5.3 Outils sur *Graph*

Nous allons maintenant présenter quelques outils et définitions sur *Graph*. Tout d'abord nous allons présenter la notion d'inclusion d'un élément de *Graph* dans un autre et la notion de cycle dans un *Graph* qui lui est liée. Ensuite, nous présenterons la notion de finitude d'un élément de *Graph* et montrerons quelques techniques et preuves de finitude et d'infinitude.

5.3.1 Inclusion d'un élément de *Graph* dans un autre

Nous allons présenter ici deux notions d'inclusion. Une notion d'inclusion stricte (un élément de *Graph* n'est pas automatiquement inclus dans lui même) et une notion d'inclusion non stricte.

5.3.1.1 Inclusion stricte

La première notion d'inclusion que nous allons présenter est une notion d'inclusion stricte. C'est-à-dire qu'un élément g_1 de *Graph* est inclus dans un autre élément g_2 si et seulement si on peut accéder à g_1 depuis un des fils de g_2. Pour définir cela, on divise les possibilités en deux cas :

- g_1 est directement équivalent à un des fils de g_2
- g_1 est inclus dans un des fils de g_2 (définition inductive)

On appelle *GinG* cette propriété et elle est définie formellement comme suit :

Définition 5.4 (*GinG*, vue inductivement)**.**

$$\frac{i : Fin(lg(sons\ g_2)) \qquad Geq_R\ g_1\ (fct\ (sons\ g_2)\ i)}{GinG_R\ g_1\ g_2}\ (dir)$$

$$\frac{i : Fin(lg(sons\ g_2)) \qquad GinG_R\ g_1\ (fct(sons\ g_2)\ i)}{GinG_R\ g_1\ g_2}\ (indir)$$

On peut montrer que *GinG* est un morphisme paramétrique pour chacun de ses deux paramètres (si sa relation de base est une relation d'équivalence). C'est-à-dire :

Lemme 5.7 (*GinG* morphisme pour son premier paramètre)**.**

$$\forall g_1\ g_1'\ g_2, Geq_{R_{eq}}\ g_1\ g_1' \wedge GinG_{R_{eq}}\ g_1\ g_2 \Rightarrow GinG_{R_{eq}}\ g_1'\ g_2$$

Démonstration. Preuve détaillée en page 189. □

Remarque 5.1. *R_{eq} indique que la relation de base R doit être une relation d'équivalence.*

Lemme 5.8 (*GinG* morphisme pour son second paramètre)**.**

$$\forall g_1\ g_2\ g_2', Geq_{R_{eq}}\ g_2\ g_2' \wedge GinG_{R_{eq}}\ g_1\ g_2 \Rightarrow GinG_{R_{eq}}\ g_1\ g_2'$$

Démonstration. La preuve est tout à fait similaire à la preuve précédente, nous ne la détaillons pas ici. □

Pour valider notre définition de *GinG*, on peut par exemple montrer que si un élément de *Graph* est inclus dans un autre, alors ses fils le sont aussi :

Lemme 5.9. $\forall g_1\ g_2, GinG_R\ g_1\ g_2 \Rightarrow \forall i_1, GinG_R\ (fct\ (sons\ g_1)\ i_1)\ g_2$

Démonstration. Preuve détaillée en page 189. □

On peut également montrer que *GinG* est transitive si R est une relation d'équivalence :

Lemme 5.10 (*GinG* transitive)**.**

$$R\ équivalence \Rightarrow (\forall g_1\ g_2\ g_3, GinG_R\ g_1\ g_2 \wedge GinG_R\ g_2\ g_3 \Rightarrow GinG_R\ g_1\ g_3)$$

Démonstration. Preuve détaillée en page 190. □

Remarque 5.2. *GinG n'est bien sûr ni réflexive ni symétrique dans le cas général.*

5.3.1.2 Inclusion non stricte

Nous allons donner ici une définition de l'inclusion non stricte d'un élément g_1 de *Graph* dans un élement g_2. Non stricte signifie qu'un élément est toujours inclus dans lui même.

Cette fois-ci, nous allons paramétrer notre définition, non plus par une relation sur T (comme précédemment), mais directement par une relation sur *Graph*. On pourra ensuite l'instancier pour toute relation, comme par exemple pour *Geq*.

Comme pour *GinG*, nous allons diviser la définition en deux cas :
- g_1 est directement équivalent g_2
- g_1 est inclus dans un des fils de g_2 (définition inductive)

On appelle $GinG^*$ cette propriété et elle est définie formellement comme suit (on note R_{Graph} la relation sur *Graph*) :

Définition 5.5 ($GinG^*$, vue inductivement)**.**

$$\frac{R_{Graph}\ g_1\ g_2}{GinG^*_{R_{Graph}}\ g_1\ g_2}\ (dirG)$$

$$\frac{i : Fin(lg(sons\ g_2)) \qquad GinG^*_{R_{Graph}}\ g_1\ (fct(sons\ g_2)\ i)}{GinG^*_{R_{Graph}}\ g_1\ g_2}\ (indirG)$$

De par sa définition, $GinG^*$ est clairement réflexive. En revanche, on ne peut pas en dire beaucoup plus sans connaître la relation R_{Graph}.

A titre d'exemple, on peut instancier R_{Graph} par *Geq*. On a alors :

Définition 5.6. $GinG'_R := GinG^*_{Geq_R}$

On peut montrer pour *GinG'* les mêmes résultats que pour *GinG* (Lemmes 5.7, 5.8, 5.9 et 5.10). Les preuves sont très similaires, nous ne les détaillons pas ici. En revanche, nous allons montrer que *GinG* est plus fine que *GinG'* (ce qui validera partiellement nos définitions).

Lemme 5.11. $\forall g_1\ g_2, GinG_R\ g_1\ g_2 \Rightarrow GinG'_R\ g_1\ g_2$

Démonstration. Preuve détaillée en page 190. □

5.3.2 Cycles dans un élément de *Graph*

Nous voulons maintenant définir la notion de cycles dans un graphe. L'objectif est de pouvoir établir si un élément de *Graph* contient un cycle ou non. Nous allons d'abord donner deux définitions différentes pour cette propriété, puis nous donnerons quelques exemples de preuves qu'un graphe possède ou non un cycle.

5.3.2.1 Définitions

On va d'abord définir le fait qu'un élément g de *Graph* est lui même situé sur un cycle. On exprime cela en disant que g est inclus (strictement) dans lui même (l'idée est de dire que si un nœud est situé dans un cycle, alors on peut l'atteindre depuis lui même). On va donc utiliser *GinG*.

Définition 5.7 (*isCycle*)**.** $\forall g, isCycle_R\ g \Leftrightarrow GinG_R\ g\ g$

Remarque 5.3. *On a nécessairement besoin de l'inclusion stricte ici, sinon la propriété serait toujours vraie.*

Avec cette propriété, on peut maintenant définir la propriété visée qui indique si un élément g de *Graph* contient un cycle. On va la définir de deux façons différentes. La première est inductive et directe (elle n'utilise pas les définitions précédentes autres que *isCycle*). On différencie encore une fois deux cas :
- soit g est lui même sur un cycle
- soit un des fils de g a un cycle (définition inductive)

On la définit de la façon suivante :

Définition 5.8 (*hasCycle*, vue inductivement)**.**

$$\frac{isCycle_R\ g}{hasCycle_R\ g}\ (hC\_dir)$$

$$\frac{i : Fin(lg(sons\ g)) \qquad hasCycle_R\ (fct(sons\ g)\ i)}{hasCycle_R\ g}\ (hC\_indir)$$

La deuxième définition est "indirecte" : on dit qu'un élément g de *Graph* contient un cycle s'il inclut (non strictement) un élément de *Graph* qui est situé sur un cycle. On va donc utiliser ici *GinG'*. On la définit ainsi :

Définition 5.9 (*hasCycle'*)**.** $\dfrac{g' : Graph\ T \qquad isCycle_R\ g' \qquad GinG'_R\ g'\ g}{hasCycle'_R\ g}$

Remarque 5.4. *On a besoin de l'inclusion non stricte ici, parce que g lui même peut être situé sur un cycle.*

On va d'abord montrer l'équivalence entre ces deux notions. Pour cela, on suppose que R est une relation d'équivalence et on la note R_{eq}.

Lemme 5.12 (Equivalence entre *hasCycle* et *hasCycle'*)**.** $\forall g, hasCycle_{R_{eq}}\ g \Leftrightarrow hasCycle'_{R_{eq}}\ g$

Démonstration. Preuve détaillée en page 191. □

5.3.2.2 Exemples

Nous allons donner ici quelques exemples de preuves qu'un graphe contient/ne contient pas de cycle. Pour faire les preuves, nous allons utiliser la Définition 5.8, mais on aurait pu de façon équivalente utiliser la Définition 5.9.

Exemple 5.4. *On peut montrer que le graphe de la Figure 2.5 (défini à l'Exemple 5.1) ne contient pas de cycle.*

Lemme 5.13. $\neg\ (hasCycle_{eq}\ Leaf)$

Démonstration. On va raisonner par l'absurde. Supposons donc que H : $hasCycle_{eq}\ Leaf$. On a deux cas pour H :

[Cas hC_dir] On a H_1 : $isCycle_{eq}\ Leaf$. Ici encore, il faut différencier deux cas pour H_1 (cas dir et indir). Cependant, ces deux cas nous donnent un i : $Fin\ (lg\ (sons\ Leaf))$. Or $lg\ (sons\ Leaf) = 0$, donc $Fin\ (lg\ (sons\ Leaf))$ est vide. Ce qui contredit l'hypothèse i.

[Cas hC_indir] Ici aussi on a i : $Fin\ (lg\ (sons\ Leaf))$, donc on termine la preuve comme précédemment.

□

Exemple 5.5. *On peut également montrer que le graphe de la Figure 2.6 (défini à l'Exemple 5.2) contient un cycle.*

Lemme 5.14. $hasCycle_{eq}\ Finite\_Graph$

Démonstration. Comme *Finite_Graph* est lui même situé sur un cycle, on applique la règle hC_dir de la Définition 5.8. On doit prouver que : $isCycle_{eq}\ Finite\_Graph$. En revanche, pour accéder au nœud *Finite_Graph* depuis lui même il faut passer par un autre nœud (le nœud 1). Donc on doit appliquer la règle indir de la Définition 5.4, avec *first* 0 (puisqu'il n'y a qu'un fils). On doit donc montrer que $GinG_{eq}\ Finite\_Graph\ (fct\ (sons\ Finite\_Graph)\ (first\ 0))$. Or *Finite_Graph* est un des fils de $fct\ (sons\ Finite\_Graph)\ (first\ 0)$ (on a fait le tour du cycle). On peut donc appliquer la règle dir de la Définition 5.4 avec la réflexivité de *Geq* pour terminer la preuve. □

On remarque que pour un élément de *Graph* fini, il est assez facile de prouver l'existence ou l'absence de cycles. En particulier sur les exemples que nous avons montrés, les preuves sont immédiates. Cependant, s'il y a beaucoup de nœuds la preuve peut être longue. En effet, dans la mesure où elle est constructive, il faudra exhiber le cycle pour prouver qu'il existe (c'est-à-dire emprunter le bon chemin à travers le graphe) ou parcourir tous les chemins pour prouver qu'il n'y en a pas. Cette dernière opération peut se révéler fastidieuse.

5.3.3 Notion de finitude

Ici, nous allons présenter ce que nous entendons par élément de *Graph* "fini" et nous donnerons une définition formelle. Puis nous montrerons quelques exemples et techniques de preuve de finitude et d'infinitude. Enfin, nous montrerons une piste pour une autre définition que nous sommes en train d'explorer.

5.3.3.1 Définition

L'idée ici est de dire qu'un graphe est fini s'il a un nombre fini de nœuds différents (c'est-à-dire non bisimilaires deux à deux). Pour exprimer cela, nous allons dire qu'un graphe est fini s'il existe une liste (finie) de tous ses nœuds.

Pour définir cette propriété, nous allons avoir besoin de définir la quantification universelle sur *Graph*. C'est une propriété qui dit qu'un prédicat $P : Graph\ T \to Prop$ sur *Graph* est vérifié par un élément de *Graph* et tous ses descendants (fils, petits-fils, arrière-petits-fils, etc.). On appelle cette quantification universelle *Gall*. Comme *Graph* est coinductif, *Gall* doit être définie coinductivement également.

Définition 5.10 (*Gall*, vue coinductivement)**.**
$$\frac{P\ g \qquad iall\ (Gall\ P)\ (sons\ g)}{Gall\ P\ g}$$

Remarque 5.5. *Nous avons introduit iall à la Définition 3.21.*

On va également avoir besoin d'une propriété qui dit qu'un élément de *Graph* est inclus dans une liste (il est bisimilaire à un élément de la liste). On note $\in$ l'appartenance (réelle) d'un élément à une liste. On appelle *element_of* cette propriété.

Définition 5.11. $element\_of_R\ l\ g \Leftrightarrow \exists y,\ y \in l\ \wedge\ Geq_R\ g\ y$

Grâce à ces deux propriétés, on peut maintenant définir *G_finite* qui dit qu'un élément de *Graph* est fini s'il existe une liste telle que tous ses nœuds (*Gall*) y soient inclus (*element_of*).

Définition 5.12 (*G_finite*)**.** $\forall g,\ G\_finite_R\ g \Leftrightarrow \exists l,\ Gall\ (element\_of_R\ l)\ g$

Pour valider la définition de *Gall*, on peut montrer que si un prédicat *P* est vrai pour tous les nœuds d'un graphe alors il est vrai pour tous les graphes inclus dans celui ci. On suppose que *P* est un morphisme pour *Geq* (il est compatible avec Geq_R).

Lemme 5.15. $\forall P\ g\ g', Gall\ P\ g \wedge GinG_R\ g'\ g \Rightarrow P\ g'$

Démonstration. Preuve détaillée en page 191. □

On peut également montrer que si deux graphes sont bisimilaires et si une propriété est vraie sur tous les nœuds de l'un, alors elle est vraie pour tous les nœuds de l'autre.

Lemme 5.16. *On suppose que P est un morphisme pour son argument.*

$$\forall P\ g_1\ g_2, Geq_R\ g_1\ g_2 \wedge Gall\ P\ g_1 \Rightarrow Gall\ P\ g_2$$

Démonstration. Preuve détaillée en page 192. □

On peut alors montrer le même type de propriété pour *G_finite*, la preuve n'étant qu'une utilisation du lemme précédent. Il faut seulement montrer auparavant que *element_of* est un morphisme pour $Geq_{R_{eq}}$ (on suppose que *R* est une relation d'équivalence).

Lemme 5.17. $\forall g_1\ g_2\ l, Geq_{R_{eq}}\ g_1\ g_2 \wedge element\_of_{R_{eq}}\ l\ g_1 \Rightarrow element\_of_{R_{eq}}\ l\ g_2$

Démonstration. Soient $H_1 : Geq_{R_{eq}}\ g_1\ g_2$ et $H_2 : element\_of_{R_{eq}}\ l\ g_1$. H_2 nous donne g tel que $H_3 : g \in l$ et $H_4 : Geq_{R_{eq}}\ g_1\ g$. On veut prouver que $\exists y,\ y \in l\ \wedge\ Geq_{R_{eq}}\ g_2\ y$. On prend $y := g$ et on utilise H_3. Il nous reste à prouver que $Geq_{R_{eq}}\ g_2\ g$ ce qu'on fait en utilisant la transitivité de *Geq* avec le symétrique de H_1 et H_4. □

On peut maintenant prouver que si deux éléments g_1 et g_2 de *Graph* sont bisimilaires et si g_1 est fini, alors g_2 est fini.

Lemme 5.18. $\forall g_1\ g_2, Geq_{R_{eq}}\ g_1\ g_2 \wedge G\_finite_{R_{eq}}\ g_1 \Rightarrow G\_finite_{R_{eq}}\ g_2$

Démonstration. Soient $H_1 : Geq_{R_{eq}}\ g_1\ g_2$ et $H_2 : G\_finite_{R_{eq}}\ g_1$. Grâce à la Définition 5.12, H_2 nous donne l tel que $H_3 : Gall\ (element\_of_{R_{eq}}\ l)\ g_1$. On applique la Définition 5.12 avec l à notre but et il nous reste à montrer que $Gall\ (element\_of_{R_{eq}}\ l)\ g_2$. Pour terminer, on applique le Lemme 5.16 avec le Lemme 5.17 ($element\_of_{R_{eq}}$ est un morphisme), H_1 et H_3. □

Enfin, pour valider la définition de *G_finite*, on veut montrer que si un graphe est fini, alors tous les graphes qui sont inclus dedans le sont aussi. Pour commencer, on va le faire dans un cas plus simple : on va montrer qu'un graphe est fini si et seulement si tous ses fils le sont aussi. Pour pouvoir faire cela, on va d'abord avoir besoin de montrer la monotonie de *element_of* par rapport à la liste, puis la monotonie de *Gall* par rapport à son prédicat et enfin nous devrons également prouver que si tous les éléments d'une liste (de *Graph*) sont finis, alors il existe une liste qui est valable pour tous les éléments (la concaténation de toutes les listes individuelles).

Remarque 5.6. *On note $l \subseteq l'$ le fait que la liste l soit inclue dans l', c'est-à-dire $\forall t, t \in l \Rightarrow t \in l'$.*

Commençons donc par la monotonie de *element_of* (par rapport à l).

Lemme 5.19. $\forall l\ l'\ g, l \subseteq l' \wedge element\_of_R\ l\ g \Rightarrow element\_of_R\ l'\ g$

Démonstration. Preuve détaillée en page 192. □

Prouvons maintenant la monotonie de *Gall* par rapport à son prédicat.

Lemme 5.20. $\forall P\ P'\ g, (\forall g, P\ g \Rightarrow P'\ g) \wedge Gall\ P\ g \Rightarrow Gall\ P'\ g$

Démonstration. Preuve détaillée en page 193. □

De ces deux lemmes, on déduit immédiatement le résultat suivant :

Lemme 5.21. $\forall l\ l'\ g, l \subseteq l' \wedge Gall\ (element\_of_R\ l)\ g \Rightarrow Gall\ (element\_of_R\ l')\ g$

Démonstration. Preuve détaillée en page 193. □

On peut maintenant "collecter" les sous-listes comme expliqué précédemment (on doit supposer que R est une relation d'équivalence) :

Lemme 5.22. $\forall l, iall\ (G\_finite_{R_{eq}})\ l \Rightarrow \exists l', iall\ (Gall\ (element\_of_{R_{eq}}\ l'))\ l$

Idée de la preuve. L'idée ici est de rassembler toutes les listes "individuelles" : $iall\ (G\_finite_{R_{eq}})\ l$ signifie que pour tout élément de l, il existe une liste qui contient tous ses nœuds. Ce qu'on veut obtenir, c'est une liste qui fonctionne pour tous les éléments de l. Le plus facile est donc de concaténer les listes de chaque élément. On doit donc raisonner par induction sur la longueur de l.

Démonstration. Preuve détaillée en page 193. □

A l'aide des lemmes précédents, on va pouvoir montrer qu'un graphe est fini si et seulement si tous ses fils le sont aussi (on suppose que R est une relation d'équivalence).

Lemme 5.23. $\forall g, G\_finite_{R_{eq}}\ g \Leftrightarrow iall\ G\_finite_{R_{eq}}\ (sons\ g)$

Démonstration.

[Direction $\Rightarrow$] Grâce à la Définition 5.12, l'hypothèse H_1 : $G\_finite_{R_{eq}}\ g$ nous donne l tel que H_2 : $Gall\ (element\_of_{R_{eq}}\ l)\ g$. On applique la Définition 5.12 avec l à notre but et on doit montrer que : $Gall\ (element\_of_{R_{eq}}\ l)\ (fct\ (sons\ g)\ i)$. L'hypothèse H_2 nous donne une nouvelle hypothèse d'intérêt pour nous, H_3 : $iall\ (Gall\ (element\_of_{R_{eq}}\ l))\ (sons\ g)$. Pour terminer la preuve, il nous suffit d'appliquer H_3 avec i.

[Direction $\Leftarrow$] Soit H_1 : $iall\ G\_finite_{R_{eq}}\ (sons\ g)$. Le Lemme 5.22 appliqué à H_1 nous donne l tel que H_2 : $iall\ (Gall\ (element\_of_{R_{eq}}\ l))\ (sons\ g)$. On applique la Définition 5.12 avec $g::l$ à notre but (l correspond aux listes de tous les fils de g, il ne manque donc que g) et on doit montrer que : $Gall\ (element\_of_{R_{eq}}\ (g::l))\ g$. On applique la Définition 5.10 et il nous reste à montrer que :

1. $\exists y,\ y \in (g::l) \wedge Geq_{R_{eq}}\ g\ y$: on prend $y := g$. Montrer que $g \in (g::l)$ est immédiat (résultat bien connu) et on prouve que $Geq_{R_{eq}}\ g\ g$ grâce à la réflexivité de Geq (Lemme 5.3).
2. $\forall i, Gall\ (element\_of_{R_{eq}}(g::l))\ (fct\ (sons\ g)\ i)$: on montre que H_3 : $l \subseteq g::l$ (la preuve est immédiate). On peut alors utiliser le Lemme 5.21 avec H_3 et H_2 appliqué à i.

□

On peut maintenant finalement montrer que si un graphe est fini, alors tous ses sous-graphes le sont aussi.

Lemme 5.24. $\forall g\ g', G\_finite_{R_{eq}}\ g \wedge GinG_{R_{eq}}\ g'\ g \Rightarrow G\_finite_{R_{eq}}\ g'$

Démonstration. Soient H_1 : $G\_finite_{R_{eq}}\ g$ et H_2 : $GinG_{R_{eq}}\ g'\ g$, on raisonne par induction sur H_2.

[Cas de base (dir)] On a i tel que H_3 : $Geq_{R_{eq}}\ g'\ (fct\ (sons\ g)\ i)$. On applique le Lemme 5.18 avec le symétrique de H_3 à notre but et il nous reste à montrer que : $G\_finite_{R_{eq}}(fct\ (sons\ g)\ i)$. On utilise alors le Lemme 5.23 avec H_1.

[Cas inductif (indir)] On a i tel que IH : $G\_finite_{R_{eq}}\ (fct\ (sons\ g)\ i) \Rightarrow G\_finite_{R_{eq}}\ g'$ (c'est l'hypothèse d'induction). Et on doit montrer que $G\_finite_{R_{eq}}\ g'$. On applique donc simplement IH. Et on termine la preuve en appliquant le Lemme 5.23 avec H_1.

□

5.3.3.2 Exemples de preuves de finitude

Pour donner un exemple d'utilisation de *G_finite*, on va prouver que *Leaf* et *Finite_Graph* sont en effet finis.

Lemme 5.25. $G\_finite_{eq}\ Leaf$

Démonstration. Dans ce graphe il n'y a qu'un nœud. On va donc appliquer la Définition 5.12 avec $lg := [Leaf]$. On doit prouver que $Gall\ (element\_of_{eq}[Leaf])\ Leaf$. On applique la Définition 5.10 et il nous reste à montrer que :

1. $\exists y, y \in [Leaf] \wedge Geq_{eq}\ Leaf\ y$: on prend évidemment $y := Leaf$. Il est ensuite immédiat de prouver que $Leaf \in [Leaf]$ et on prouve que $Geq_{eq}\ Leaf\ Leaf$ par réflexivité de Geq (Lemme 5.3).
2. $\forall i, Gall\ (element\_of_{eq}[Leaf])\ (iniln_0\ i)$: i est de type $Fin\ 0$ qui est vide.

□

Lemme 5.26. $G\_finite_{eq}\ Finite\_Graph$

Démonstration. Dans ce graphe il y a deux nœuds. On va donc appliquer la Définition 5.12 avec $lg := [Finite\_Graph; mk\_Graph\ 1\ [\![Finite\_Graph]\!]]$. On doit prouver que $Gall\ (element\_of_{eq}[Finite\_Graph; mk\_Graph\ 1\ [\![Finite\_Graph]\!]])\ Finite\_Graph$. On raisonne par coinduction (parce que *Finite_Graph* est défini coinductivement). L'hypothèse de coinduction est : $CH : Gall\ (element\_of_{eq}[Finite\_Graph; mk\_Graph\ 1\ [\![Finite\_Graph]\!]])\ Finite\_Graph$. On applique la Définition 5.10 à notre but et il nous reste à montrer que :

1. $\exists y, y \in [Finite\_Graph; mk\_Graph\ 1\ [\![Finite\_Graph]\!]] \wedge Geq_{eq}\ Finite\_Graph\ y$: on prend évidemment $y := Finite\_Graph$, et on finit cette partie de la preuve comme pour *Leaf*.
2. $\forall i, Gall\ (element\_of_{eq}[Finite\_Graph; mk\_Graph\ 1\ [\![Finite\_Graph]\!]])\ (fct\ (sons\ Finite\_Graph)\ i)$: cela se simplifie en :

   $Gall\ (element\_of_{eq}[Finite\_Graph; mk\_Graph\ 1\ [\![Finite\_Graph]\!]])\ (mk\_Graph\ 1\ [\![Finite\_Graph]\!])$

   Ici encore on applique la Définition 5.10 (pour le deuxième nœud cette fois-ci) et on doit montrer que (en simplifiant directement comme montré précédemment) :

   (a) $\exists y,\ y \in [Finite\_Graph; mk\_Graph\ 1\ [\![Finite\_Graph]\!]] \wedge Geq_{eq}\ (mk\_Graph\ 1\ [\![Finite\_Graph]\!])\ y$
   On prend bien sûr $y := mk\_Graph\ 1\ [\![Finite\_Graph]\!]$, et on finit cette partie de la preuve comme précédemment.

   (b) $Gall\ (element\_of_{eq}[Finite\_Graph; mk\_Graph\ 1\ [\![Finite\_Graph]\!]])\ Finite\_Graph$: on applique directement CH.

□

On voit donc qu'il est assez simple de montrer qu'un graphe est fini. Il suffit de fournir une liste contenant tous les nœuds du graphe puis parcourir le graphe (éventuellement coinductivement) pour montrer que tous les nœuds sont bien dans la liste. Au total la preuve peut être assez longue et rébarbative, mais elle reste assez aisée.

5.3.3.3 Preuves d'infinitude : résultats généraux et exemples

Il est souvent plus simple de prouver qu'une propriété existentielle est vraie plutôt que de prouver qu'elle est fausse. Ici aussi il est plus compliqué de prouver qu'un graphe est infini plutôt que de prouver (constructivement) qu'il est fini. Nous allons donner quelques techniques et exemples de preuves qu'un graphe est infini.

On peut par exemple montrer que si un graphe d'entiers naturels est fini, alors ses étiquettes sont bornées. Donc en prenant la contraposée, on pourra montrer que si les étiquettes d'un graphe sont non bornées alors le graphe est infini. On pourra utiliser ce résultat pour prouver que *Infinite_Graph* est infini. De la même façon, on peut montrer que si le nombre de fils des nœuds est non borné, alors le graphe est infini (et inversement, s'il est fini alors le nombre des fils de ses nœuds est borné).

On a abstrait ces résultats en résultat général qu'on instanciera ensuite pour les exemples que nous avons cités. Nous allons montrer que si le graphe est fini, alors pour toute fonction $f : Graph\ T \to \mathbb{N}$ (on supposera que f est un morphisme pour *Geq*), l'application de f aux nœuds du graphe admet une borne supérieure. On montrera également la contraposée.

Lemme 5.27. $\forall g\ f, G\_finite_R\ g \Rightarrow \exists m, Gall\ (\lambda x.f\ x \leq m)\ g$

Idée de la preuve. L'idée ici est de dire qu'on va prendre pour m le maximum de la liste *map f l* (avec l obtenue à partir de $G\_finite_R\ g$). Pour trouver ce maximum, on a créé une fonction *max_list_nat* qui renvoie le maximum d'une liste d'entiers naturels (comme la liste est finie, ce maximum existe toujours, on renvoie 0 lorsque la liste est vide). Elle est définie comme suit (on note max la fonction prédéfinie qui renvoie le maximum de deux entiers naturels et *fold* l'itérateur bien connu des listes) :

Définition 5.13. $max\_list\_nat\ l = fold\ \max\ l\ 0$

On peut aisément montrer (par induction sur l) que :

Lemme 5.28. $\forall l\ x, x \in l \Rightarrow x \leq max\_list\_nat\ l$

Comme on pourra le montrer, ce maximum remplit toutes les conditions nécessaires à notre preuve. Ensuite la preuve se fait simplement par coinduction.

Démonstration. Preuve détaillée en page 194. □

On énonce la contraposée (qui est le résultat qui nous intéresse ici, puisqu'on veut faire des preuves d'infinité) :

Lemme 5.29. $\forall g\ f, \neg(\exists m, Gall\ (\lambda x.f\ x \leq m)\ g) \Rightarrow \neg(G\_finite_R\ g)$

Démonstration. La preuve est immédiate et nous ne la détaillons pas ici. □

On peut maintenant l'instancier pour les étiquettes d'un graphe d'entiers ou pour les longueurs des *ilist* de fils :

Lemme 5.30. $\forall g\ f, \neg(\exists m, Gall\ (\lambda x.label\ x \leq m)\ g) \Rightarrow \neg(G\_finite_R\ g)$

Démonstration. La preuve ici est simplement une instanciation du Lemme 5.29. La seule chose qu'on doit faire c'est montrer que *label* est un morphisme pour *Geq*. La preuve est immédiate et on ne la détaille pas ici. □

Lemme 5.31. $\forall g\ f, \neg(\exists m, Gall\ (\lambda x.lg\ (sons\ x) \leq m)\ g) \Rightarrow \neg(G\_finite_R\ g)$

Démonstration. Ici encore il s'agit simplement d'une instanciation du Lemme 5.29. On doit prouver que $\lambda x.lg\ (sons\ x)$ est un morphisme pour *Geq*. La preuve est ici aussi immédiate. □

En utilisant cette méthode, on peut donc prouver que *Infinite_Graph* est infini :

Lemme 5.32. $\forall n, \neg(G\_finite_R\ Infinite\_Graph_n)$

Idée de la preuve. L'idée ici est d'utiliser la technique précédente en montrant que les étiquettes sont non bornées. Pour montrer cela, on va raisonner par l'absurde et supposer qu'on a une borne. Puis on va prouver que $\forall n\ m, n < m \Rightarrow GinG_{eq}\ Infinite\_Graph_m\ Infinite\_Graph_n$. Et ensuite, on montrera que l'existence de la borne supérieure est incompatible avec ce résultat.

Démonstration. On applique le Lemme 5.30. Il nous suffit alors de prouver que : $\neg\ (\exists m, Gall\ (\lambda x.label\ x \leq m)\ Infinite\_Graph_n)$. On raisonne par l'absurde. Supposons donc qu'on a m tel que $H_1 : Gall\ (\lambda x.label\ x \leq m)\ Infinite\_Graph_n$. Grâce à H_1 on obtient $H_2 : label\ Infinite\_Graph_n \leq m$, c'est-à-dire $H_2 : n \leq m$. Pour trouver une contradiction, nous voulons montrer que : $\forall n\ m, n < m \Rightarrow GinG_{eq}\ Infinite\_Graph_m\ Infinite\_Graph_n$. On va commencer par le montrer sur un cas plus simple ($m = n + 1$) :

Lemme 5.33. $\forall n, GinG_{eq}\ Infinite\_Graph_{n+1}\ Infinite\_Graph_n$

Démonstration. Preuve détaillée en page 194. □

On peut maintenant prouver le lemme dans le cas général :

Lemme 5.34. $\forall n\ m, n < m \Rightarrow GinG_{eq}\ Infinite\_Graph_m\ Infinite\_Graph_n$

Démonstration. Preuve détaillée en page 194. □

Revenons donc maintenant à la preuve du Lemme 5.32. Grâce au Lemme 5.34 on sait que $H_3 : GinG_{eq}\ Infinite\_Graph_{m+1}\ Infinite\_Graph_n$. On veut maintenant prouver que $H_4 : m + 1 \leq m$ (ce qui terminera la preuve puisque c'est faux). On montre cela avec le Lemme 5.15 appliqué à H_1 et H_3. □

Cependant, ces méthodes ne suffisent pas toujours. Le graphe de la Figure 5.2 par exemple est infini, mais on ne peut pas utiliser ces méthodes pour le prouver : en effet, ses labels sont bornés (par 1) et tous ses nœuds n'ont qu'un fils. Pour utiliser la méthode proposée,

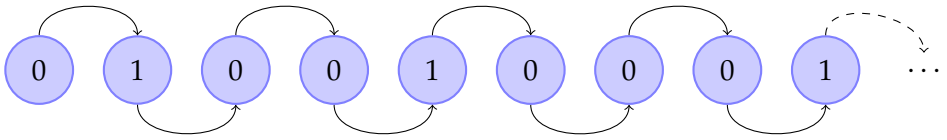


Figure 5.2 — Exemple d'un graphe infini mais avec labels et nombre de fils bornés

on doit trouver une fonction de type $Graph\ \mathbb{N} \to \mathbb{N}$ qui soit non bornée. On pourrait alors par exemple compter les 0 en tête du graphe (c'est-à-dire avant le premier 1). Si on pouvait faire cela, on aurait alors une fonction non bornée (globalement, on peut diviser le graphe en groupe de nœuds : n nœuds 0, puis un nœud 1, puis $n + 1$ nœuds 0, etc., donc cette fonction est non bornée pour les nœuds du graphe). Cependant, on se trouve avec de nouveau un problème de mélange entre induction et coinduction. En effet, compter les nœuds 0 est un problème inductif mais on veut le faire dans un graphe coinductif. Et la condition de garde nous empêche d'écrire cela. D'ailleurs, intuitivement, on voit bien qu'on ne peut pas définir une telle fonction : il peut y avoir un nombre infini de 0 (voir par exemple le graphe de la Figure 5.3). L'idée serait donc de rajouter une condition (du type, le graphe a au moins un nœud 1) qui assure la terminaison du calcul. Cependant, cette condition serait logique et non structurelle et le problème ne serait donc pas résolu immédiatement. Les travaux de Bertot

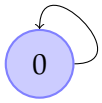


Figure 5.3 — Exemple d'un graphe avec une infinité de 0 en tête

et Komendantskaya [15] pourraient être une bonne piste pour avancer dans cette direction, mais nous n'avons pas été plus avant. En effet, nous avons quand même réussi à prouver que ce graphe est infini en utilisant d'autres techniques. La preuve est assez compliquée et ne présente pas d'intérêt ici (elle ne peut pas vraiment non plus servir de base à une technique plus générale). Nous ne la détaillons pas.

Il est cependant intéressant de noter que si au lieu de considérer le graphe de la Figure 5.2 on considère le graphe de la Figure 5.4, on peut utiliser la méthode présentée précédemment (la fonction considérée est la fonction qui renvoie la taille de la liste étiquette de chaque nœud, la méthode est ensuite exactement identique à celle utilisée pour le Lemme 5.32). Pourtant, le graphe de la Figure 5.4 peut être vu comme une compression du graphe de la Figure 5.2. Il contient les mêmes informations mais présentées sous une forme différente. Cette compression (qui permet de passer d'une représentation à une autre) est une transformation sur les graphes. Il serait intéressant de l'étudier plus avant pour obtenir une autre preuve que le graphe de la Figure 5.2 est infini, qui s'appuierait cette fois-ci sur la technique proposée plus haut.

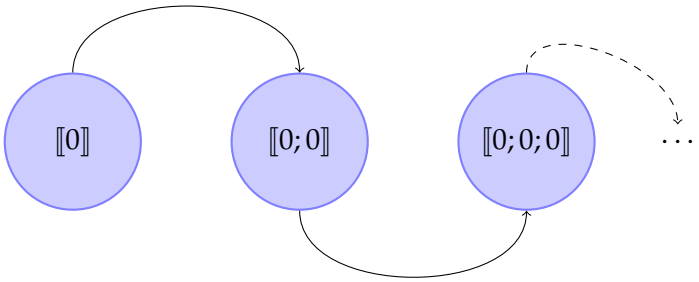


Figure 5.4 — Représentation comprimée du graphe de la Figure 5.2

5.3.3.4 Une autre piste

Nous sommes actuellement en train d'explorer une autre piste (conjointement avec l'équipe *Logic and Semantic Group* de l'*Institute of Cybernetics* de Tallinn, Estonie) pour établir un nouveau critère de finitude. L'idée est d'utiliser une autre représentation possible des graphes en arbre avec pointeurs de retour [7, 50]. On a alors deux types de nœuds : les nœuds "normaux" c'est-à-dire avec une étiquette et des fils et les nœuds "pointeurs" qui pointent simplement vers un nœud défini précédemment. Comme il n'y a pas de cycle à proprement parler, cette représentation peut être inductive, c'est-à-dire finie (si on considère bien sûr des graphes finis, c'est-à-dire avec un nombre fini de nœuds distincts). L'idée serait donc de dire qu'un graphe est fini s'il existe un arbre avec pointeur de retour "équivalent" (cette équivalence est bien entendu à définir).

Pour représenter les nœuds "pointeurs", nous avons utilisé l'idée de "remonter" dans

l'arbre à l'aide d'un parcours inverse. Pour cela il, faut pouvoir mémoriser le chemin déjà parcouru et pouvoir le traverser à l'envers. Nous utilisons pour cela la notion de chemins et de "zippers".

Cependant, ce travail est toujours en cours.

6 Vers une représentation plus souple

DANS la Section 5.2, nous avons expliqué la nécessité d'avoir une relation de bisimilarité sur *Graph*. Et nous en avons proposé une, *Geq*, qui répondait aux besoins présentés. Cependant, si on observe *Geq* on voit qu'elle donne implicitement un ordre vertical et horizontal aux nœuds. En effet, pour être équivalents, deux éléments de *Graph* doivent avoir été construits exactement de la même façon, dans le même ordre. Verticalement parce qu'on donne une racine au graphe et qu'elle doit être la même dans les deux graphes, horizontalement parce qu'il faut que les nœuds aient les mêmes fils, dans le même ordre. Dans les représentations classiques (typiquement, celle présentée Section 2.1.2.1), cet ordre n'existe pas puisque tous les nœuds sont représentés au même niveau, sans hiérarchie.

Notre objectif est d'obtenir cette relation plus libérale sur notre représentation constructive. Par exemple, nous voudrions que les graphes de la Figure 6.1 ou ceux de la Figure 6.2 soient équivalents. Comme nous venons de l'expliquer, selon *Geq* ils ne le sont pas puisque les nœuds ne sont pas dans le même ordre. Dans le cas de la Figure 6.1 c'est l'ordre horizontal (celui des fils) qui est différent, dans la Figure 6.2, c'est l'ordre vertical puisque le second graphe a été tourné de 180° par rapport au premier.

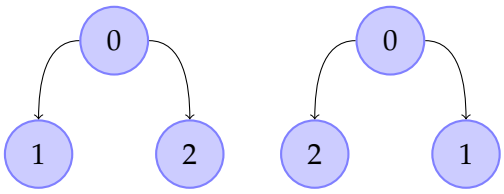


Figure 6.1 — Ordre différent dans les fils

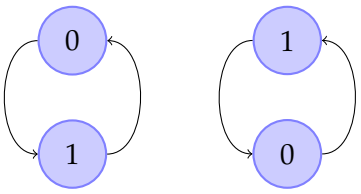


Figure 6.2 — Racines différentes

La nouvelle relation doit répondre à ces deux problématiques. La première correspond à une permutation dans l'ordre des fils, c'est ce que nous allons étudier Section 6.1. La seconde correspond à un changement du point de vue de l'observation du graphe, ce que nous étudierons Section 6.2.

Enfin, une autre restriction de la représentation des graphes par *Graph* est le fait que le graphe doive être enraciné (et donc connexe). Par exemple, on ne peut pas représenter à l'aide de *Graph* le graphe de la Figure 2.4. Nous verrons deux possibilités pour pallier ce manque dans la Section 6.3.

6.1 Une relation sur *Graph* qui inclut les permutations

Nous allons donc commencer par résoudre le problème de l'ordre horizontal (c'est-à-dire entre les fils d'un nœud). Comme on l'a dit, libéraliser l'ordre horizontal, c'est-à-dire autoriser deux nœuds qui ont les mêmes fils mais pas dans le même ordre à être équivalents, revient à autoriser les permutations dans les fils. On va donc tout naturellement utiliser les re-

lations que nous avons présentées au Chapitre 4. Comme nous sommes ici dans un contexte coinductif (avec *Graph*) obtenir la décidabilité de la relation sur *Graph* est impossible. Nous ne pourrons donc pas utiliser la relation *iperm_occ*. En revanche, nous avons montré que les autres relations de permutations sur *ilist* étaient équivalentes. Nous pouvons donc utiliser n'importe laquelle. Nous avons expliqué que la définition que nous retenions comme étant "notre" définition, était la Définition 4.17. C'est donc celle-ci que nous utiliserons ci-après, sauf indication contraire.

6.1.1 Définition

Nous allons définir une nouvelle relation sur *Graph*, sur le même modèle que *Geq* mais en utilisant cette fois-ci *iperm_ind* au lieu de *ilist_rel*.

Définition 6.1 (*GPerm*, vue coinductivement)**.**

$$\frac{g_1\ g_2 : Graph\ T \qquad R\ (label\ g_1)\ (label\ g_2) \qquad iperm\_ind_{GPerm_R}\ (sons\ g_1)\ (sons\ g_2)}{GPerm_R\ g_1\ g_2}$$

La principale différence structurelle par rapport à *Geq*, c'est que *iperm_ind*, contrairement à *ilist_rel*, est définie inductivement. Nous le savons, cela risque de nous poser des problèmes.

Dans l'optique de prouver que *GPerm* préserve l'équivalence, essayons tout d'abord de prouver qu'elle préserve la réflexivité.

Lemme 6.1 (*GPerm* préserve la réflexivité)**.** $R\ réflexive \Rightarrow \forall g, GPerm_R\ g\ g$

Démonstration. On fait la preuve par coinduction (c'est-à-dire avec la tactique `cofix` de Coq). L'hypothèse de coinduction est $CH : \forall g, GPerm_R\ g\ g$ et il nous suffit de prouver :

1. $R\ (label\ g)\ (label\ g)$: on le prouve grâce à la réflexivité de R.
2. $iperm\_ind_{GPerm_R}\ (sons\ g)\ (sons\ g)$: on pourrait vouloir utiliser le résultat qui dit que *iperm_ind* préserve la réflexivité (Lemme 4.22). Donc pour prouver que $iperm\_ind_{GPerm_R}\ (sons\ g)\ (sons\ g)$, on aurait seulement besoin de prouver $\forall g, GPerm_R\ g\ g$ ce qui est CH. Cependant, cette preuve n'est pas correcte pour Coq parce que l'appel à CH n'est pas gardé (il devrait être utilisé directement sous un constructeur, non pas par un lemme). On va donc refaire la preuve de la réflexivité de *iperm_ind* pour l'argument $sons\ g$. Cependant, comme *iperm_ind* est inductive, cette partie de la preuve est faite par induction, à l'intérieur de la coinduction initiale. L'appel corécursif est dans une construction récursive. Donc, la preuve résultante ne passe pas non plus la vérification de la condition de garde de Coq.

ABANDON.

Cet échec était attendu puisque, qu'on utilise le Lemme 4.22 ou qu'on refasse sa preuve dans la preuve précédente, on avait une preuve inductive incluse dans une preuve coinductive. Il est donc naturel qu'on ne puisse pas la faire directement. On répète seulement ici au niveau des preuves les problèmes que l'on aurait eus au niveau des définitions si on avait utilisé les listes au lieu des *ilist* dans la définition de *Graph*.

Pourtant il nous semble naturel (et correct) de vouloir prouver la réflexivité de *GPerm*. Nous allons donc essayer de montrer cela en utilisant d'autres moyens. Nous allons d'abord montrer une solution dans laquelle nous abandonnons l'utilisation de la coinduction telle

que définie dans Coq pour une définition imprédicative. Ensuite, nous allons essayer d'utiliser une notion inductive équivalente à *GPerm*, afin de ne plus avoir de problèmes de mélange entre induction et coinduction. Enfin, nous montrerons qu'en utilisant *iperm_bij* au lieu de *iperm_ind* les choses se passent mieux (mais nous tenions à utiliser *iperm_ind* puisque c'est la définition de notre choix).

6.1.2 Utilisation de l'imprédicativité

Le problème que nous avons rencontré précédemment (pour la preuve de la réflexivité de *GPerm*) est principalement dû à des limitations de la coinduction telle que définie dans Coq. Comme nous l'avons expliqué, ces preuves étaient pour nous "moralement" correctes. Pour appuyer cette conviction, nous avons décidé de redéfinir les principes de coinduction et d'abandonner donc la tactique `cofix` de Coq, trop bridée. Pour cela, nous avons redéfini *GPerm* imprédicativement. Nous allons tout d'abord présenter une version imprédicative "brute", puis nous présenterons une version raffinée dans le style de Mendler.

6.1.2.1 Version imprédicative

L'idée ici est donc toujours de voir *GPerm* coinductivement mais plus par rapport à la commande `CoInductive` de Coq. Nous nous inspirons d'une des approches présentées dans [39] où les travaux sont traités dans le système F. Nous les transposons simplement dans l'univers des propositions de Coq (c'est-à-dire un système avec des types dépendants).

Remarque 6.1. *Cette partie sur l'imprédicativité (Section 6.1.2) a été entièrement développée par Ralph Matthes. Elle est nécessaire pour la suite, c'est pourquoi nous la présentons. Cependant, aucune connaissance sur l'imprédicativité n'est nécessaire. C'est pour cette raison que nous n'approfondissons pas et présentons ces résultats très superficiellement.*

Remarque 6.2. *Rappelons que dans Coq, `Set` n'est plus imprédicatif (l'imprédicativité est optionnelle mais son utilisation est peu recommandée pour des bibliothèques de visée générale parce qu'elle risque d'entrer en conflit avec d'autres notions) mais `Prop`, l'univers des propositions, l'est encore. Nous n'aurions donc pas pu utiliser une version imprédicative pour contourner les problèmes de garde que nous avions à la définition de Graph, sans nous servir de cette option qui est incompatible avec d'autres constructions, notamment les types quotientés [33]. Ici, au contraire, on considère que l'utilisation de l'imprédicativité est raisonnable. Notons cependant qu'historiquement, l'imprédicativité dans `Set` a pu être exploitée avantageusement pour définir des types coinductifs avant leur implantation dans Coq. Ainsi, dans [52] et [68], Paulin-Mohring utilise les idées de Wraith [82] pour définir et raisonner sur des listes infinies imprédicativement dans Coq.*

On appellera *GPerm_imp* cette version de *GPerm*. On définit *GPerm_imp* de la façon suivante :

Définition 6.2 (*GPerm_imp*)**.**

$$\forall g_1\ g_2, GPerm\_imp_R\ g_1\ g_2 \quad \Leftrightarrow \quad \exists \mathcal{R}, \quad \begin{array}{l}(\forall g_1' g_2', \mathcal{R}\ g_1'\ g_2' \Rightarrow R\ (label\ g_1')\ (label\ g_2') \\ \wedge iperm\_ind_{\mathcal{R}}\ (sons\ g_1')\ (sons\ g_2')) \wedge \mathcal{R}\ g_1\ g_2\end{array}$$

On doit maintenant prouver les trois principes suivants, qui nous permettront d'utiliser *GPerm_imp* essentiellement comme s'il avait été défini dans Coq avec la commande `CoInductive`.

Lemme 6.2 (Principe de coinduction pour *GPerm_imp*)**.** *Supposons que*

$$\forall g_1\ g_2,\ \mathcal{R}\ g_1\ g_2 \Rightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm\_ind_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2)$$

avec $\mathcal{R}$ une relation sur Graph T. Alors, $\forall g_1\ g_2,\ \mathcal{R}\ g_1\ g_2 \Rightarrow GPerm\_imp_R\ g_1\ g_2$.

Démonstration. Soient

$$H_1 : \forall g_1\ g_2,\ \mathcal{R}\ g_1\ g_2 \Rightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm\_ind_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2) \quad H_2 : \mathcal{R}\ g_1\ g_2$$

On veut prouver que

$$\exists \mathcal{R}', \left(\forall g_1' g_2', \mathcal{R}'\ g_1'\ g_2' \Rightarrow R\ (label\ g_1')\ (label\ g_2') \wedge iperm\_ind_{\mathcal{R}'}\ (sons\ g_1')\ (sons\ g_2')\right) \wedge \mathcal{R}'\ g_1\ g_2$$

On prend $\mathcal{R}' := \mathcal{R}$. On doit donc prouver que :

1. $\forall g_1' g_2', \mathcal{R}\ g_1'\ g_2' \Rightarrow R\ (label\ g_1')\ (label\ g_2') \wedge iperm\_ind_{\mathcal{R}}\ (sons\ g_1')\ (sons\ g_2')$: c'est H_1.
2. $\mathcal{R}\ g_1\ g_2$: c'est H_2.

□

Cette preuve est une trivialité, c'est la force de l'imprédicativité de pouvoir forcer un tel principe par construction.

Lemme 6.3 (Principe de "dépliage" pour *GPerm_imp*)**.**

$$\forall g_1 g_2, GPerm\_imp_R\ g_1\ g_2 \Rightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2)$$

Démonstration. Soit $H : GPerm\_imp_R\ g_1\ g_2$. D'après la Définition 6.2, H nous donne $\mathcal{R}$ tel que :

$$H_1 : \forall g_1\ g_2, \mathcal{R}\ g_1\ g_2 \Rightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm\_ind_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2) \quad H_2 : \mathcal{R}\ g_1\ g_2$$

En appliquant H_1 à H_2, on obtient les deux nouvelles hypothèses $H_3 : R\ (label\ g_1)\ (label\ g_2)$ et $H_4 : iperm\_ind_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2)$. On doit prouver que :

1. $R\ (label\ g_1)\ (label\ g_2)$: c'est H_3.
2. $iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2)$: d'après le lemme de monotonie de *iperm_ind* (Lemme 4.34) utilisé avec H_4, il nous suffit de prouver que $\mathcal{R} \subseteq GPerm\_imp_R$, c'est-à-dire que : $\forall g_1' g_2', \mathcal{R}\ g_1'\ g_2' \Rightarrow GPerm\_imp_R\ g_1'\ g_2'$. On utilise le Lemme 6.2 avec H_1. □

Lemme 6.4 (Constructeur pour *GPerm_imp*)**.**

$$\forall g_1 g_2, R\ (label\ g_1)\ (label\ g_2) \wedge iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2) \Rightarrow GPerm\_imp_R\ g_1\ g_2$$

Démonstration. Soient $H_1 : R\ (label\ g_1)\ (label\ g_2)$ et $H_2 : iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2)$. On note $\mathcal{R}$ la relation sur *Graph* suivante :

$$\mathcal{R}\ g_1\ g_2 \Leftrightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2)$$

D'après le Lemme 6.2 utilisé avec H_1 et H_2, il nous suffit de prouver que :

$$\forall g_1' g_2', \mathcal{R}\ g_1'\ g_2' \Rightarrow R\ (label\ g_1')\ (label\ g_2') \wedge iperm\_ind_{\mathcal{R}}\ (sons\ g_1')\ (sons\ g_2')$$

L'hypothèse $\mathcal{R}\ g_1'\ g_2'$ nous donne

$$H_4 : R\ (label\ g_1')\ (label\ g_2') \quad H_5 : iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1')\ (sons\ g_2')$$

On doit prouver que :

1. $R\ (label\ g'_1)\ (label\ g'_2)$: c'est H_4.
2. $iperm\_ind_{\mathcal{R}}\ (sons\ g'_1)\ (sons\ g'_2)$: d'après le lemme de monotonie de *iperm_ind* (Lemme 4.34) utilisé avec H_5, il nous suffit de prouver que $GPerm\_imp_R \subseteq \mathcal{R}$, c'est-à-dire que

   $$\forall g\ g', GPerm\_imp_R\ g\ g' \Rightarrow R\ (label\ g)\ (label\ g') \wedge iperm\_ind_{GPerm\_imp_R}\ (sons\ g)\ (sons\ g')$$

   on utilise simplement le Lemme 6.3.

□

Il est maintenant facile de montrer que *GPerm* est plus fine que *GPerm_imp*.

Lemme 6.5. $GPerm_R \subseteq GPerm\_imp_R$

Démonstration. D'après le Lemme 6.2 utilisé avec $\mathcal{R} := GPerm_R$, il nous suffit de montrer que :

$$\forall g'_1\ g'_2, GPerm_R\ g'_1\ g'_2 \Rightarrow R\ (label\ g'_1)\ (label\ g'_2) \wedge iperm\_ind_{GPerm_R}\ (sons\ g'_1)\ (sons\ g'_2)$$

D'après la Définition 6.1, l'hypothèse $GPerm_R\ g'_1\ g'_2$ nous donne $H_2 : R\ (label\ g'_1)\ (label\ g'_2)$ et $H_3 : iperm\_ind_{GPerm_R}\ (sons\ g'_1)\ (sons\ g'_2)$, exactement ce que l'on voulait montrer. □

En revanche, le sens inverse est "moralement" vrai, mais non démontrable dans Coq, puisque de nouveau on a un mélange entre induction et coinduction.

On peut aisément prouver que *GPerm_imp* préserve l'équivalence :

Lemme 6.6 (*GPerm_imp* préserve la réflexivité)**.** R *réflexive* $\Rightarrow \forall g, GPerm\_imp_R\ g\ g$

Démonstration. D'après le Lemme 6.2 utilisé avec l'égalité de Leibniz, il nous suffit de prouver que :

1. $\forall g_1\ g_2, g_1 = g_2 \Rightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm\_ind_{eq}\ (sons\ g_1)\ (sons\ g_2)$: c'est-à-dire, en réécrivant l'hypothèse : $g_1 = g_2$:
   (a) $R\ (label\ g_1)\ (label\ g_1)$: on le prouve par réflexivité de R.
   (b) $iperm\_ind_{eq}\ (sons\ g_1)\ (sons\ g_1)$: on le prouve grâce au Lemme 4.22 et à la réflexivité de l'égalité de Leibniz.
2. $g = g$: on le prouve par réflexivité de l'égalité de Leibniz.

□

Lemme 6.7 (*GPerm_imp* préserve la symétrie)**.**

$$R\ symétrique \Rightarrow (\forall g_1\ g_2, GPerm\_imp_R\ g_1\ g_2 \Rightarrow GPerm\_imp_R\ g_2\ g_1)$$

Démonstration. Soit $H_1 : GPerm\_imp_R\ g_1\ g_2$ et $\mathcal{R}$ la relation $\forall g\ g', \mathcal{R}\ g\ g' \Leftrightarrow GPerm\_imp_R\ g'\ g$. D'après le Lemme 6.2 appliqué avec $\mathcal{R}$, il nous suffit de montrer que :

1. $\forall g'_1\ g'_2, GPerm\_imp_R\ g'_2\ g'_1 \Rightarrow R\ (label\ g'_1)\ (label\ g'_2) \wedge iperm\_ind_{\mathcal{R}}\ (sons\ g'_1)\ (sons\ g'_2)$: d'après le Lemme 6.3 et l'hypothèse $GPerm\_imp_R\ g'_2\ g'_1$, on a $H_2 : R\ (label\ g'_2)\ (label\ g'_1)$ et $H_3 : iperm\_ind_{GPerm\_imp_R}\ (sons\ g'_2)\ (sons\ g'_1)$. On veut prouver que :
   (a) $R\ (label\ g'_1)\ (label\ g'_2)$: on le prouve grâce à la symétrie de R et à H_2.

(b) $iperm\_ind_{\mathcal{R}}\ (sons\ g_1')\ (sons\ g_2')$: on le prouve grâce au Lemme 4.23 avec H_3.

2. $\mathcal{R}\ g_2\ g_1$: d'après la définition de $\mathcal{R}$, on doit montrer que $GPerm\_imp_R\ g_1\ g_2$, c'est H_1.

□

Lemme 6.8 (*GPerm_imp* préserve la transitivité)**.**

$$R\ transitive \Rightarrow (\forall g_1\ g_2\ g_3, GPerm\_imp_R\ g_1\ g_2 \wedge GPerm\_imp_R\ g_2\ g_3 \Rightarrow GPerm\_imp_R\ g_1\ g_3)$$

Démonstration. Soient H_1 : $GPerm\_imp_R\ g_1\ g_2$, H_2 : $GPerm\_imp_R\ g_2\ g_3$ et $\mathcal{R}$ la relation $\forall g_1\ g_3, \mathcal{R}\ g_1\ g_3 \Leftrightarrow \exists g_2, GPerm\_imp_R\ g_1\ g_2 \wedge GPerm\_imp_R\ g_2\ g_3$. D'après le Lemme 6.2 utilisé avec $\mathcal{R}$ il nous suffit de prouver que :

1. $\forall g_1'\ g_3', \mathcal{R}\ g_1'\ g_3' \Rightarrow R\ (label\ g_1')\ (label\ g_3') \wedge iperm\_ind_{\mathcal{R}}\ (sons\ g_1')\ (sons\ g_3')$: l'hypothèse $\mathcal{R}\ g_1'\ g_3'$ nous donne g_2' tel que H_3 : $GPerm\_imp_R\ g_1'\ g_2'$ et H_4 : $GPerm\_imp_R\ g_2'\ g_3'$. Le Lemme 6.3 utilisé deux fois (une avec H_3, une avec H_4) nous donne :

$$\begin{array}{ll} H_5 : R\ (label\ g_1)\ (label\ g_2) & H_6 : iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2) \\ H_7 : R\ (label\ g_2)\ (label\ g_3) & H_8 : iperm\_ind_{GPerm\_imp_R}\ (sons\ g_2)\ (sons\ g_3) \end{array}$$

   On veut prouver que :

   (a) $R\ (label\ g_1')\ (label\ g_3')$: on le montre par transitivité de R avec H_5 et H_7.

   (b) $iperm\_ind_{\mathcal{R}}\ (sons\ g_1')\ (sons\ g_3')$: on utilise le Lemme 4.27 avec H_6 et H_8.

2. $\mathcal{R}\ g_1\ g_3$: c'est-à-dire : $\exists g, GPerm\_imp_R\ g_1\ g \wedge GPerm\_imp_R\ g\ g_3$. On prend $g := g_2$ et on termine la preuve avec H_1 et H_2.

□

Lemme 6.9 (*GPerm_imp* préserve l'équivalence)**.** *R équivalence* $\Rightarrow$ $GPerm\_imp_R$ *équivalence*

Démonstration. La preuve est simplement une utilisation des trois lemmes précédents. □

6.1.2.2 Version à la Mendler

On peut également définir *GPerm* dans une version coinductive à la Mendler, directement exprimable avec la commande `CoInductive` de Coq, mais tout de même imprédicative. On appelle cette version *GPerm_mend*. On s'inspire ici du travail réalisé par Nakata et Uustalu dans [61].

Définition 6.3 (*GPerm_mend*, vue coinductivement)**.**

$$\frac{\mathcal{R} \subseteq GPerm\_mend_R \qquad R\ (label\ g_1)\ (label\ g_2) \qquad iperm\_ind_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2)}{GPerm\_mend_R\ g_1\ g_2}$$

On peut prouver que *GPerm_imp* et *GPerm_mend* sont équivalentes.

Lemme 6.10. $\forall g_1\ g_2, GPerm\_imp_R\ g_1\ g_2 \Leftrightarrow GPerm\_mend_R\ g_1\ g_2$

Démonstration. Preuve détaillée en page 195. □

6.1.3 Vers une représentation inductive équivalente

Une autre solution pour éviter les problèmes liés à la condition de garde est d'essayer de trouver une représentation inductive équivalente à notre représentation coinductive. Il faut donc dans un premier temps trouver cette structure, puis prouver son équivalence avec la précédente. Nous avons d'abord pensé à comparer les chemins à travers le graphe. Mais comme nous le verrons, dans le cas des permutations cela ne suffit pas. Nous serons donc obligés de faire appel à une autre solution plus complète.

6.1.3.1 Première idée : chemins dans le graphe

Nous avons d'abord voulu comparer les chemins à travers le graphe. L'idée était de dire que deux graphes sont équivalents si pour tout chemin, son exploration à travers le graphe mène au même nœud dans les deux graphes. L'idée ensuite est de montrer que la relation obtenue préserve l'équivalence et qu'elle est équivalente à *GPerm*.

Avant de nous attaquer au problème un peu compliqué des permutations, nous avons testé cette représentation sur un cas plus simple : celui de *Geq*.

Prise en main avec *Geq* Nous avons décidé de représenter les chemins par des listes d'entiers. Chaque entier représente le numéro du fils à choisir pour suivre le chemin. Ainsi, par exemple, si on empruntait le chemin $[2;1;2;1;1]$ dans le graphe de la Figure 2.3, on parcourrait les nœuds dans l'ordre (on commence bien sûr par la racine) $1,1,2,4,2,3$. Et si on voulait parcourir le chemin $[1,3]$, on ne pourrait pas puisqu'on passerait par le nœud 1 puis par le nœud 2 mais ensuite on voudrait passer dans son troisième fils qui n'existe pas.

Nous allons définir une fonction qui parcourt un graphe selon un chemin et renvoie la valeur (l'étiquette) du nœud d'arrivée. Comme nous l'avons vu, cette valeur peut exister (si le chemin est en effet un chemin existant) ou non (si le chemin n'est pas valide). Nous allons donc renvoyer un résultat de type *option T*. Pour les exemples précédents, nous voulons donc obtenir *Some* 3 pour le premier et *None* pour le second. Pour un chemin vide ([]), on renvoie la valeur de la racine (donc dans le cas du graphe de la Figure 2.3, *Some* 1). Nous appelons *rpath* cette fonction et elle est définie inductivement comme suit :

Définition 6.4 (*rpath*)**.**

$$
\begin{array}{llll}
\mathit{rpath} & & : & \mathit{list}\ \mathbb{N} \to \mathit{Graph}\ T \to T \\
\mathit{rpath} & [] \quad (\mathit{mk\_Graph}\ t\ l) & := & \mathit{Some}\ t \\
\mathit{rpath} & n{::}l \quad (\mathit{mk\_Graph}\ t\ \langle n', ln \rangle) & := & \begin{cases} \mathit{rpath}\ l\ (ln\ (\mathit{code}\ h)) & \mathit{si}\ h : n < n' \\ \mathit{None} & \mathit{si}\ h : n' \leq n \end{cases}
\end{array}
$$

On veut maintenant définir une nouvelle relation sur *Graph* qui dit que deux éléments de *Graph* sont équivalents si pour tout chemin, le parcours dans les deux graphes donne le même résultat. Les nœuds étant d'un type T quelconque, muni d'une relation R, pour pouvoir comparer les résultats du parcours, il faut donc qu'on puisse comparer des éléments de type *option T*. Pour cela on définit le transformeur de relation suivant qui permet de transformer une relation sur T en relation sur *option T*.

Définition 6.5 (*RelOp*)**.**

$$\begin{array}{llllll} RelOp & & & & : & relation\ T \to relation\ (option\ T) \\ RelOp & R & None & None & := & True \\ RelOp & R & None & Some\ t & := & False \\ RelOp & R & Some\ t & None & := & False \\ RelOp & R & Some\ t_1 & Some\ t_2 & := & R\ t_1\ t_2 \end{array}$$

La démonstration que *RelOp* préserve l'équivalence est immédiate.

On peut donc maintenant définir la relation sur *Graph* voulue :

Définition 6.6 (*GeqPath*)**.** $\forall g_1\ g_2, GeqPath_R\ g_1\ g_2 \Leftrightarrow \forall l, RelOp_R\ (rpath\ l\ g_1)\ (rpath\ l\ g_2)$

Ce que l'on voulait obtenir était une relation équivalente à *Geq*. Nous allons donc le prouver ici. Mais tout d'abord, nous allons montrer que les longueurs des *ilist* de fils sont respectées par *GeqPath*.

Lemme 6.11. $\forall g_1\ g_2, GeqPath_R\ g_1\ g_2 \Rightarrow lg\ (sons\ g_1) = lg\ (sons\ g_2)$

Démonstration. Preuve détaillée en page 195. □

Lemme 6.12. $\forall g_1\ g_2, Geq_R\ g_1\ g_2 \Leftrightarrow GeqPath_R\ g_1\ g_2$

Démonstration. Soient t_1, n_1, ln_1, t_2, n_2 et ln_2 tels que $g_1 = mk\_Graph\ t_1\ \langle n_1, ln_1\rangle$ et $g_2 = mk\_Graph\ t_2\ \langle n_2, ln_2\rangle$.

[Direction $\Rightarrow$] Soit $H : Geq_R\ (mk\_Graph\ t_1\ \langle n_1, ln_1\rangle)\ (mk\_Graph\ t_2\ \langle n_2, ln_2\rangle)$. On veut montrer que : $\forall l, RelOp_R\ (rpath\ l\ g_1)\ (rpath\ l\ g_2)$. On raisonne par induction sur l. Grâce aux Définitions 5.3 et 3.11, H nous donne :

$$H_1 : R\ t_1\ t_2 \quad H_2 : n_1 = n_2 \quad H_3 : \forall i, Geq_R\ (ln_1\ i)\ (ln_2\ (conv_{H_2}\ i))$$

[Cas de base ($l = []$)] Ici on veut montrer que

$$RelOp_R\ (rpath\ []\ (mk\_Graph\ t_1\ \langle n_1, ln_1\rangle))\ (rpath\ []\ (mk\_Graph\ t_2\ \langle n_2, ln_2\rangle))$$

Cela se simplifie, grâce aux Définitions 6.4 et 6.5, en $R\ t_1\ t_2$, qui est vrai d'après H_1.

[Cas inductif ($l = n::q$)] Ici on veut montrer que

$$RelOp_R\ (rpath\ n::q\ (mk\_Graph\ t_1\ \langle n_1, ln_1\rangle))\ (rpath\ n::q\ (mk\_Graph\ t_2\ \langle n_2, ln_2\rangle))$$

Pour pouvoir appliquer la Définition 6.4 (afin d'appliquer ensuite la Définition 6.5), on doit comparer n avec n_1 et n_2. Comme $n_1 = n_2$ (d'après H_2), il nous suffit de le faire pour n_1. Analysons les différents cas possibles :

[Cas $H_4 : n_1 \leq n$] En simplifiant on doit montrer que $RelOp_R\ None\ None$, ce qui est vrai d'après la Définition 6.5.

[Cas $H_4 : n < n_1$] On a donc $H_5 : n < n_2$. Et l'hypothèse d'induction est

$$IH : \forall g_1\ g_2, Geq_R\ g_1\ g_2 \Rightarrow RelOp_R\ (rpath\ q\ g_1)\ (rpath\ q\ g_2)$$

En simplifiant on doit montrer que

$$RelOp_R\ (rpath\ q\ (ln_1\ (code\ H_4)))\ (rpath\ q\ (ln_2\ (code\ H_5)))$$

D'après *IH* il nous suffit de montrer : $Geq_R\ (ln_1\ (code\ H_4))\ (ln_2\ (code\ H_5))$. On montre aisément que $code\ H_5 = conv_{H_2}\ (code\ H_4)$ et on finit la preuve avec H_3.

[Direction ⇐] On raisonne par coinduction. L'hypothèse de coinduction est

$$CH : \forall g_1\ g_2, GeqPath_R\ g_1\ g_2 \Rightarrow Geq_R\ g_1\ g_2$$

Soit $H_1 : \forall l, RelOp_R\ (rpath\ l\ (mk\_Graph\ t_1\ \langle n_1, ln_1\rangle))\ (rpath\ l\ (mk\_Graph\ t_2\ \langle n_2, ln_2\rangle))$. D'après la Définition 5.3 il nous suffit de montrer que :

1. $R\ t_1\ t_2$: ici il nous suffit d'appliquer H_1 avec $[]$.
2. $ilist\_rel_{Geq_R}\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle$: on obtient $H_2 : n_1 = n_2$ grâce au Lemme 6.11 appliqué à H_1. D'après la Définition 3.11, il nous suffit de prouver que :

   $$\forall i, Geq_R\ (ln_1\ i)\ (ln_2\ (conv_{H_2}\ i))$$

   Ou encore, d'après CH : $GeqPath_R\ (ln_1\ i)\ (ln_2\ (conv_{H_2}\ i))$. Pour l fixé, il nous suffit donc de prouver que : $RelOp_R\ (rpath\ l\ (ln_1\ i))\ (rpath\ l\ (ln_2\ (conv_{H_2}\ i)))$.
   H_1 appliqué à $decode\ i{::}l$ nous donne

   $$\begin{array}{rll} H_3 : & RelOp_R & (rpath\ (decode\ i{::}l)\ (mk\_Graph\ t_1\ \langle n_1, ln_1\rangle)) \\ & & (rpath\ (decode\ i{::}l)\ (mk\_Graph\ t_2\ \langle n_2, ln_2\rangle)) \end{array}$$

   On sait d'après le Lemme 3.5 que $H_4 : decode\ i < n_1$ et donc d'après H_2 que $H_5 : decode\ i < n_2$. On peut donc utiliser la Définition 6.4 dans H_3 qui nous donne :

   $$H_3 : RelOp_R\ (rpath\ l\ (ln_1\ (code\ H_4)))\ (rpath\ l\ (ln_2\ (code\ H_5)))$$

   Il nous suffit finalement de montrer que $code\ H_4 = i$ et que $code\ H_5 = conv_{H_2}\ i$, ce qui se fait aisément.

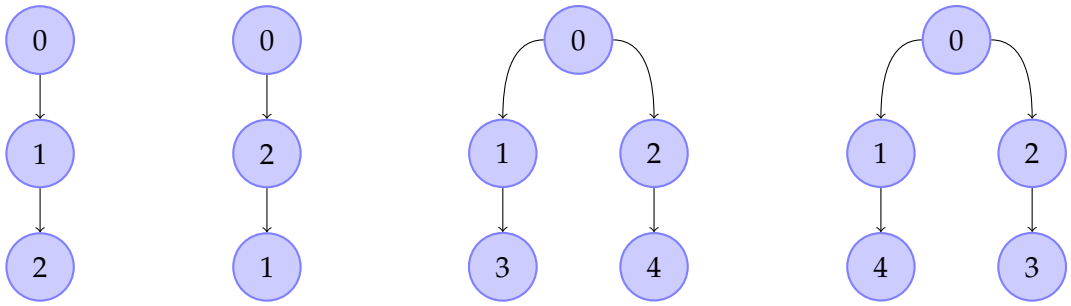
□

Nous avons donc bien obtenu une nouvelle relation sur *Graph*, avec des moyens inductifs cette fois-ci et équivalente à *Geq*. Voyons maintenant si nous pouvons faire la même chose pour *GPerm*.

Application aux permutations Ici, on ne peut pas utiliser exactement la même méthode que précédemment puisque les chemins dans les deux graphes ne sont pas les mêmes. Une première idée serait de réutiliser le même principe de base que pour *GeqPath*. On dirait alors que deux graphes sont équivalents si et seulement si pour tout chemin du premier, il existe un chemin du deuxième qui donne le même résultat, et inversement :

$$\begin{array}{rcl} \forall g_1\ g_2, GPermPath_R\ g_1\ g_2 & \Leftrightarrow & \forall l_1 \exists l_2, RelOp_R\ (rpath\ l_1\ g_1)\ (rpath\ l_2\ g_2) \\ & \wedge & \forall l_2 \exists l_1, RelOp_R\ (rpath\ l_1\ g_1)\ (rpath\ l_2\ g_2) \end{array}$$

Cependant, ici, on ne vérifie pas que les chemins traversent les mêmes nœuds. Dans le cas de *GeqPath*, cela ne posait pas de problèmes puisqu'on prenait le même chemin dans les deux graphes. Donc le fait que les nœuds traversés sont les mêmes était assuré par la définition. En effet, comme $GeqPath_R\ g_1\ g_2 \Leftrightarrow \forall l, RelOp_R\ (rpath\ l\ g_1)\ (rpath\ l\ g_2)$, on sait que si pour l on a $RelOp_R\ (rpath\ l\ g_1)\ (rpath\ l\ g_2)$, c'est également vrai pour tous les préfixes de l (c'est-à-dire, pour tous les nœuds traversés par le chemin). Mais pour *GPermPath*, la problématique est assez différente. Avec la définition donnée précédemment, on pourrait montrer que les paires de graphes de la Figure 6.3 sont équivalentes. En ajoutant une contrainte sur les longueurs des listes (en imposant qu'elles soient égales), on résoudrait le problème de



(a) Ordre vertical différent (b) Ordre horizontal différent

Figure 6.3 — Exemples de graphes équivalents par *GPermPath*

la Figure 6.3(a) mais pas celui de la Figure 6.3(b). Il nous faut donc vérifier que les nœuds traversés en parcourant un chemin sont bien les mêmes.

Cependant, encore une fois, cela ne serait pas suffisant : en effet, par exemple dans le cas de la Figure 6.4, les graphes représentés seraient équivalents (pour chaque chemin du premier, il en existe un de longueur égale dans le second qui traverse les mêmes nœuds), alors qu'on ne le souhaite pas. Ici, le problème vient du fait que le nombre de fils de la racine n'est pas le même dans les deux cas.

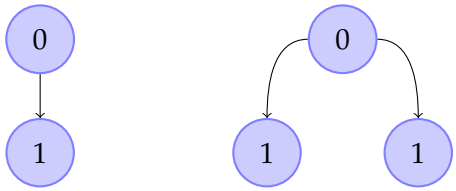


Figure 6.4 — Contre-exemple pour la solution avec nœuds traversés équivalents

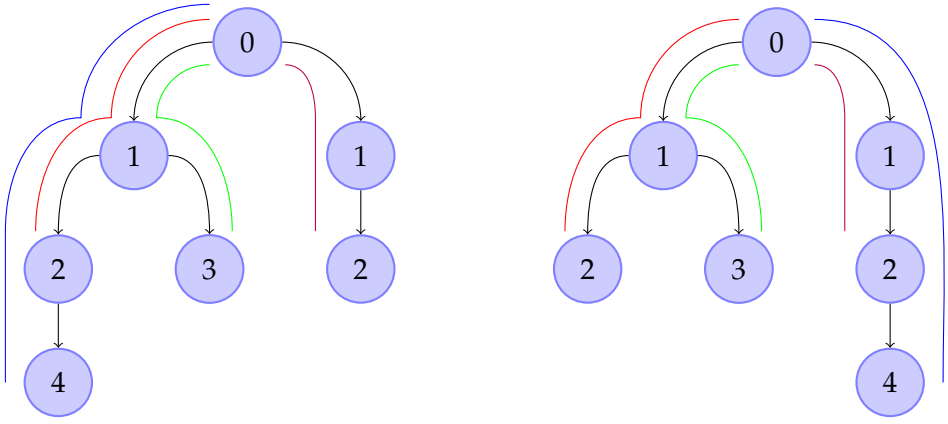


Figure 6.5 — Contre-exemple pour la solution avec même nombre de frères pour le nœud d'arrivée. Les chemins équivalents dans les deux graphes sont indiqués par les mêmes couleurs – on ne représente que les chemins de longueur supérieure ou égale à deux

On pourrait alors vérifier en plus pour les nœuds "d'arrivée" que les étiquettes des *ilist* auxquelles ils appartiennent sont des permutations l'une de l'autre. Mais cela ne suffirait toujours pas. On pourrait alors prouver que les graphes de la Figure 6.5 sont équivalents (alors que par *GPerm* ils ne le sont pas). Ici le problème est que si on vérifie bien que les

étiquettes des nœuds traversés sont équivalentes et que les *ilist* d'arrivées le sont aussi, on ne vérifie pas que les *ilist* traversées sont équivalentes.

On pourrait donc vérifier à chaque étape que les *ilist* traversées sont des permutations les unes des autres. Mais ici encore, on peut trouver des exemples de graphes qui seraient équivalents par cette relation et qui ne le seraient pas par *GPerm* (voir l'exemple de la Figure 6.6).

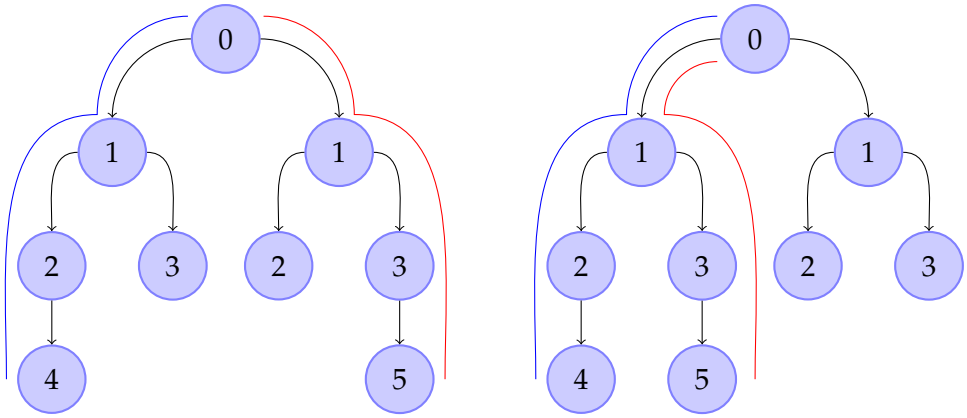


Figure 6.6 — Contre-exemple pour la solution avec même nombre de frères pour tous les nœuds traversés. On n'a représenté que les chemins de longueur 3 qui sont les seuls problématiques

Ici encore, le problème vient du fait qu'avec la notion de chemin on perd trop d'informations. Comme on l'a vu, en ajouter ne suffit pas. Il faut qu'on garde toute l'information des nœuds traversés, c'est-à-dire principalement l'arborescence. Nous devons donc abandonner l'idée de raisonner sur les chemins pour nous tourner vers les arbres.

6.1.3.2 Seconde idée : les arbres

L'intérêt d'utiliser les arbres c'est qu'ils gardent exactement la même structure arborescente que les graphes. Avec les arbres, nous voulons représenter des observations des graphes jusqu'à une certaine profondeur n. Par exemple, l'arbre de la Figure 6.7 représente l'observation du graphe de la Figure 2.3 jusqu'à la profondeur 3.

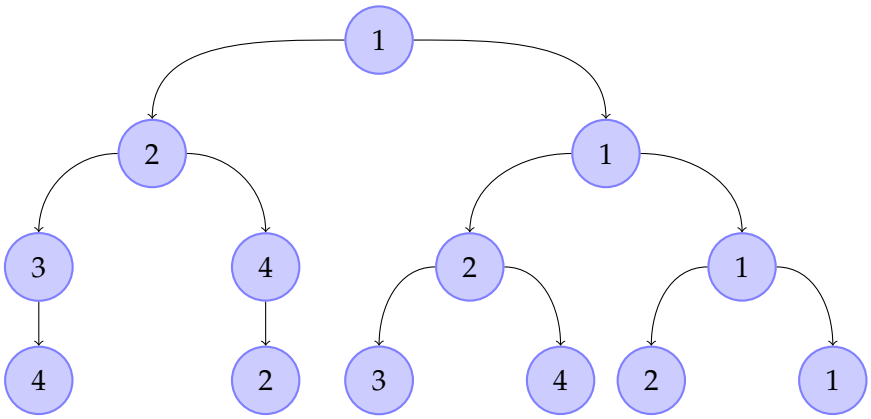


Figure 6.7 — Observation du graphe de la Figure 2.3 jusqu'à la profondeur 3

Observer jusqu'à la profondeur n signifie qu'on transforme le graphe en un arbre de profondeur (maximum) n, en coupant toutes les structures plus profondes.

Nous allons donc tout d'abord définir les arbres et la fonction "d'observation". Nous pourrons ensuite définir une relation sur *Graph* qui utilise les arbres (et qui n'est donc pas coinductive). Et on montrera que cette relation est équivalente à *GPerm*.

Définitions Pour représenter les arbres, nous allons utiliser une définition similaire à la Définition 2.1 mais en remplaçant les listes par des *ilist* (pour les fils), afin d'être le plus proche possible de la définition des graphes.

Définition 6.7 (*iTree*, vu inductivement)**.**

$$\frac{t : T \qquad l : \mathit{ilist}\ (\mathit{iTree}\ T)}{\mathit{mk\_iTree}\ t\ l : \mathit{iTree}\ T}$$

Comme on l'a fait pour *Graph*, on définit les deux projections *labeliT* et *sonsiT* telles que $\mathit{labeliT}\ (\mathit{mk\_iTree}\ t\ l) = t$, $\mathit{sonsiT}\ (\mathit{mk\_iTree}\ t\ l) = l$ et que le lemme suivant soit correct :

Lemme 6.13. $\forall t,\ t = \mathit{mk\_iTree}\ (\mathit{labeliT}\ t)\ (\mathit{sonsiT}\ t)$

Comme on l'a dit, on a besoin d'une fonction qui permet de transformer un graphe en arbre de profondeur n, c'est-à-dire d'observer un graphe jusqu'à la profondeur n :

Définition 6.8 (*G2iT*, définie récursivement)**.**

$$\begin{array}{llll} \mathit{G2iT} & & : & \mathbb{N} \to \mathit{Graph}\ T \to \mathit{iTree}\ T \\ \mathit{G2iT}\ \ 0 \qquad (\mathit{mk\_Graph}\ t\ l) & & := & \mathit{mk\_iTree}\ t\ [\![]\!] \\ \mathit{G2iT}\ \ (n+1) \quad (\mathit{mk\_Graph}\ t\ l) & & := & \mathit{mk\_iTree}\ t\ (\mathit{imap}\ (\mathit{G2iT}\ n)\ l) \end{array}$$

Comme on l'a fait pour les chemins, on peut définir, en passant par les observations une relation sur *Graph* équivalente à *Geq* (en utilisant *G2iT*). Nous ne la développons pas ici. Nous allons directement nous intéresser à la relation équivalente à *GPerm*, en commençant par définir le dual de *GPerm* pour *iTree*.

Définition 6.9 (*TPerm*, vue inductivement)**.**

$$\frac{t_1\ t_2 : \mathit{iTree}\ T \qquad R\ (\mathit{labeliT}\ t_1)\ (\mathit{labeliT}\ t_2) \qquad \mathit{iperm\_ind}_{\mathit{TPerm}_R}\ (\mathit{sonsiT}\ t_1)\ (\mathit{sonsiT}\ t_2)}{\mathit{TPerm}_R\ t_1\ t_2}$$

On va montrer que *TPerm* préserve l'équivalence (pour rappel, on n'a pas pu le faire directement pour *GPerm*). Comme précédemment, on va montrer indépendamment qu'elle préserve la réflexivité, la symétrie et la transitivité.

Lemme 6.14. *R réflexive* $\Rightarrow \forall t, \mathit{TPerm}_R\ t\ t$

Démonstration. On raisonne par induction sur t. Soient x, n et ln tels que l'hypothèse d'induction soit : $IH : \forall i, \mathit{TPerm}_R\ (ln\ i)\ (ln\ i)$ et que notre objectif soit $\mathit{TPerm}_R\ (\mathit{mk\_iTree}\ x\ \langle n, ln\rangle)\ (\mathit{mk\_iTree}\ x\ \langle n, ln\rangle)$. D'après la Définition 6.9, il nous suffit de prouver que :

1. $R\ x\ x$: on prouve cela par réflexivité de R.

2. $iperm\_ind_{TPerm_R}\ \langle n, ln\rangle\ \langle n, ln\rangle$: ici on voudrait utiliser la réflexivité de *iperm_ind* (Lemme 4.22) mais on aurait besoin comme hypothèse de savoir que *TPerm* est réflexive, ce que l'on essaye de montrer justement. On va donc raisonner par induction sur n afin d'utiliser IH.

   [Cas 0**]** Ici, on veut prouver que $iperm\_ind_{TPerm_R}\ \langle 0, ln\rangle\ \langle 0, ln\rangle$, ce qui est vrai d'après la Définition 4.17.

   [Cas $n+1$**]** L'hypothèse d'induction est

$$IH' : \forall ln, (\forall i, TPerm_R\ (ln\ i)\ (ln\ i) \Rightarrow iperm\_ind_{TPerm_R}\ \langle n, ln\rangle\ \langle n, ln\rangle)$$

   D'après la Définition 4.17, il nous suffit de prouver que (on prend $i_1 = i_2 = first\ n$) :

   (a) $TPerm_R\ (ln\ (first\ n))\ (ln\ (first\ n))$: on prouve cela directement avec IH.

   (b) $iperm\_ind_{TPerm_R}\ \langle n, ln \circ succ\rangle\ \langle n, ln \circ succ\rangle$: d'après IH' il nous suffit de prouver que $\forall i, TPerm_R\ (ln(succ\ i))\ (ln(succ\ i))$, ce qui est vrai d'après IH.

□

Lemme 6.15. *R symétrique* $\Rightarrow (\forall t_1\ t_2, TPerm_R\ t_1\ t_2 \Rightarrow TPerm_R\ t_2\ t_1)$

Démonstration. On raisonne par induction sur l'hypothèse $TPerm_R\ t_1\ t_2$. On a

$$H_1 : R\ (labeliT\ t_1)\ (labeliT\ t_2) \quad IH : iperm\_ind_{flip\ TPerm_R}\ (sonsiT\ t_1)\ (sonsiT\ t_2)$$

D'après la Définition 6.9 il nous suffit de prouver que :

1. $R\ (labeliT\ t_2)\ (labeliT\ t_1)$: il nous suffit d'utiliser la symétrie de R et H_1.
2. $iperm\_ind_{TPerm_R}\ (sonsiT\ t_2)\ (sonsiT\ t_1)$: d'après le Lemme 4.23, il nous suffit de montrer que $iperm\_ind_{flip\ TPerm_R}\ (sonsiT\ t_1)\ (sonsiT\ t_2)$, ce qui est vrai d'après IH.

□

Lemme 6.16. *R transitive* $\Rightarrow (\forall t_1\ t_2\ t_3, TPerm_R\ t_1\ t_2 \wedge TPerm_R\ t_2\ t_3 \Rightarrow TPerm_R\ t_1\ t_3)$

Démonstration. L'astuce ici va être d'utiliser le lemme intermédiaire démontré pour prouver la transitivité de *iperm_ind* (Lemme 4.25). Encore une fois, on ne pourra pas utiliser la transitivité de *iperm_ind* directement, en revanche, on peut appliquer ce lemme moins général. On raisonne par induction sur t_1. Cela nous donne t et l et l'hypothèse d'induction est :

$$IH : \forall i\ t_2\ t_3, TPerm_R\ (fct\ l\ i)\ t_2 \wedge TPerm_R\ t_2\ t_3 \Rightarrow TPerm_R\ (fct\ l\ i)\ t_3$$

Soient $H_1 : TPerm_R\ (mk\_iTree\ t\ l)\ t_2$ et $H_2 : TPerm_R\ t_2\ t_3$. D'après la Définition 6.9 H_1 et H_2 nous donnent :

$$\begin{array}{ll} H_3 : R\ t\ (labeliT\ l_2) & H_4 : iperm\_ind_{TPerm_R}\ l\ (sonsiT\ t_2) \\ H_5 : R\ (labeliT\ l_2)\ (labeliT\ l_3) & H_6 : iperm\_ind_{TPerm_R}\ (sonsiT\ t_2)\ (sonsiT\ t_3) \end{array}$$

Et toujours d'après la Définition 6.9, il nous suffit de montrer que :

1. $R\ t\ (labeliT\ t_3)$: on prouve cela par transitivité de R avec H_3 et H_5.
2. $iperm\_ind_{TPerm_R}\ l\ (sonsiT\ t_3)$: on utilise le Lemme 4.25 avec IH, H_4 et H_6.

□

Lemme 6.17. *R équivalence* $\Rightarrow$ $TPerm_R$ *équivalence*

Démonstration. La preuve est une combinaison des trois lemmes précédents. □

A l'aide de *TPerm* et de *G2iT*, on définit une notation qui exprime que les observations de deux graphes à la profondeur n sont équivalentes :

Définition 6.10. $\forall n\ g_1\ g_2,\ g_1 \equiv_{R,n} g_2 \Leftrightarrow TPerm_R\ (G2iT\ n\ g_1)\ (G2iT\ n\ g_2)$

Le lemme suivant se démontre immédiatement à l'aide du Lemme 6.17.

Lemme 6.18. *R équivalence $\Rightarrow \forall n, \equiv_{R,n}$ équivalence*

On peut également décrire $\equiv_{R,n}$ récursivement comme une fonction de n. Cela revient à dire que les propriétés suivantes sont vraies :

Propriété 6.1.

$$\forall g_1\ g_2, g_1 \equiv_{R,0} g_2 \quad \Leftrightarrow \quad R\ (label\ g_1)\ (label\ g_2) \tag{6.1.1}$$

$$\begin{aligned} \forall g_1\ g_2, g_1 \equiv_{R,n+1} g_2 \quad &\Leftrightarrow \quad R\ (label\ g_1)\ (label\ g_2) \\ &\wedge \quad iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2) \end{aligned} \tag{6.1.2}$$

Démonstration. Preuve détaillée en page 197. □

On peut également montrer que $\equiv_{R,n}$ préserve la décidabilité :

Lemme 6.19. *Dec $R \Rightarrow$ Dec $\equiv_{R,n}$*

Démonstration. Preuve détaillée en page 198. □

Enfin, on montre aussi que si deux observations sont équivalentes, alors toutes les observations moins profondes le sont aussi. Pour prouver ce lemme général, nous allons nous servir du résultat suivant qui exprime la même chose mais pour deux profondeurs consécutives (n et $n+1$) :

Lemme 6.20. $\forall g_1\ g_2\ n, g_1 \equiv_{R,n+1} g_2 \Rightarrow g_1 \equiv_{R,n} g_2$

Démonstration. Preuve détaillée en page 198. □

Lemme 6.21. $\forall g_1\ g_2\ n\ m, m \leq n\ \wedge\ g_1 \equiv_{R,n} g_2 \Rightarrow g_1 \equiv_{R,m} g_2$

Démonstration. On montre formellement le raisonnement par induction sur $H_1 : m \leq n$. Le schéma d'induction pour $\leq$ est le suivant :

$$\forall n\ P, P\ n \wedge (\forall m, n \leq m \wedge P\ m \Rightarrow P\ (m+1)) \Rightarrow (\forall m, n \leq m \Rightarrow P\ m)$$

[Cas de base] On a $H_2 : g_1 \equiv_{R,n} g_2$, ce qu'on voulait démontrer.

[Cas inductif] On a $H_1 : m \leq n$ et $H_2 : g_1 \equiv_{R,n+1} g_2$. L'hypothèse d'induction est IH : $g_1 \equiv_{R,n} g_2 \Rightarrow g_1 \equiv_{R,m} g_2$. On veut montrer que $g_1 \equiv_{R,m} g_2$. D'après IH, il nous suffit de montrer que : $g_1 \equiv_{R,n} g_2$. On démontre cela grâce au Lemme 6.20 et à H_2.

□

On peut maintenant, grâce à $\equiv_{R,n}$ définir une relation sur *Graph*. L'idée est de dire que deux éléments de *Graph* sont équivalents si et seulement si toutes leurs observations le sont (c'est-à-dire, les observations pour toutes les profondeurs). On appelle *GTPerm* cette relation.

Définition 6.11 (*GTPerm*)**.** $\forall g_1\ g_2, GTPerm_R\ g_1\ g_2 \Leftrightarrow \forall n, g_1 \equiv_{R,n} g_2$

On peut montrer que *GTPerm* préserve l'équivalence.

Lemme 6.22. *R équivalence* $\Rightarrow$ *GTPerm équivalence*

Démonstration. On se sert simplement du Lemme 6.18. □

Notre objectif maintenant est de montrer que *GPerm_imp* et *GTPerm* sont équivalents. Ainsi, on aura donné une description de *GPerm* utilisant des moyens inductifs. On aura aussi démontré "moralement" que *GPerm* préserve l'équivalence (grâce au Lemme 6.22).

Equivalence entre *GPerm_imp* et *GTPerm* Notre objectif est donc de montrer le théorème suivant, qui est le résultat principal de cette partie :

Théorème 6.1. $\forall g_1\ g_2, GPerm\_imp_R\ g_1\ g_2 \Leftrightarrow GTPerm_R\ g_1\ g_2$

Démonstration. **[Direction $\Rightarrow$]** On veut montrer que $GPerm\_imp_R\ g_1\ g_2 \Rightarrow \forall n, g_1 \equiv_{R,n} g_2$. On raisonne par induction sur n. D'après le Lemme 6.3, l'hypothèse $H : GPerm\_imp_R\ g_1\ g_2$ nous donne :

$$H_1 : R\ (label\ g_1)\ (label\ g_2) \quad H_2 : iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2)$$

[Cas 0**]** On veut prouver que $g_1 \equiv_{R,0} g_2$. On utilise la Propriété 6.1.1 avec H_1.

[Cas $n+1$**]** L'hypothèse d'induction est $IH : \forall g_1\ g_2, GPerm\_imp_R\ g_1\ g_2 \Rightarrow g_1 \equiv_{R,n} g_2$. On peut le lire également comme $IH : GPerm\_imp_R \subseteq \equiv_{R,n}$. On veut prouver que $g_1 \equiv_{R,n+1} g_2$. D'après la Propriété 6.1.2, on doit prouver que :

1. $R\ (label\ g_1)\ (label\ g_2)$: c'est H_1.
2. $iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$: d'après le Lemme 4.34 utilisé avec IH, il nous suffit de prouver que $iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2)$, et c'est H_2.

Remarque 6.3. *Grâce au Lemme 6.5, on a donc :* $\forall g_1\ g_2, GPerm_R\ g_1\ g_2 \Rightarrow GTPerm_R\ g_1\ g_2$*, la réciproque n'étant que "moralement" vraie.*

[Direction $\Leftarrow$] Soit $H_1 : GTPerm_R\ g_1\ g_2$. On fait la preuve coinductivement, c'est-à-dire qu'on applique le Lemme 6.2. Evidemment, on utilise ici $\mathcal{R} := GTPerm_R$. Il nous suffit donc de prouver que :

1. $\forall g_1'\ g_2', GTPerm\ g_1'\ g_2' \Rightarrow R\ (label\ g_1')\ (label\ g_2') \wedge iperm\_ind_{GTPerm_R}\ (sons\ g_1')\ (sons\ g_2')$: soit $H_2 : GTPerm\ g_1'\ g_2'$, c'est-à-dire $H_2 : \forall n, g_1' \equiv_{R,n} g_2'$. On veut prouver que :
   (a) $R\ (label\ g_1')\ (label\ g_2')$: on applique la Propriété 6.1.1 avec H_2 appliqué à 0.
   (b) $iperm\_ind_{GTPerm_R}\ (sons\ g_1')\ (sons\ g_2')$: on montre en réalité le résultat en général et on l'instancie ici, c'est le Lemme 6.23 et on l'utilise avec H_2. On énonce et on démontre ce résultat ci-dessous pour des raisons de lisibilité parce que la preuve en est longue. C'est ce résultat qui est le plus compliqué à démontrer.
2. $GTPerm\ g_1\ g_2$: c'est H_1.

□

Lemme 6.23. $\forall g_1\ g_2, GTPerm\ g_1\ g_2 \Rightarrow iperm\_ind_{GTPerm_R}\ (sons\ g_1)\ (sons\ g_2)$

Idée de la preuve. Le problème principal qui se pose ici est un problème de continuité. En effet, ce résultat ressemble assez (en partie) à la Propriété 6.1.2, mais pour tout n :

$$(\forall n, g_1 \equiv_{R,n} g_2) \Rightarrow iperm\_ind_{\cap_n \equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$$

L'idée est donc de pouvoir fixer une permutation pour *sons* g_1 et *sons* g_2 qui soit valable à tous les niveaux d'extraction avec *G2iT*. Même si cela paraît banal, ce n'est en fait pas trivial. En effet, certaines permutations peuvent n'être valables que jusqu'à une certaine profondeur. Par exemple, dans la Figure 6.8, seule la permutation symbolisée par les flèches bleues pleines est valide à la profondeur 2. Celle symbolisée par les flèches rouges pointillées n'est valide que jusqu'à la profondeur 1 mais pas au-delà. Il faut donc qu'on trouve un

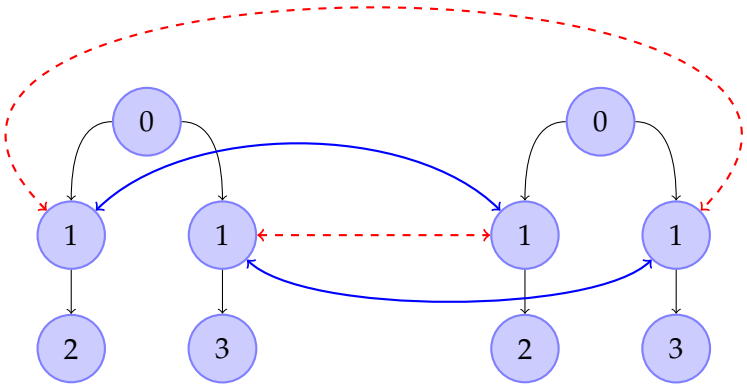


Figure 6.8 — Graphes avec plusieurs possibilités de permutations au premier niveau

moyen d'obtenir une "bonne" permutation dès le début (et montrer aussi qu'elle existe). C'est le défi principal de cette preuve, et comme nous allons le voir, on sera obligé pour démontrer cela d'avoir recours à un axiome non-constructif (mais consistant quand même avec le CIC) : le principe des tiroirs infini. On voit également que comme on va devoir manipuler des permutations "concrètes", on aura besoin de *iperm_ind_skel*.

En revanche, on verra que dans l'autre sens il n'y a pas de problème : une permutation qui est valide à la profondeur n est aussi valide à toutes les profondeurs m inférieures à n. C'est partiellement ce qu'exprime le Lemme 6.21. On n'a pas la notion de "même permutation", mais ce résultat nous suffira.

Démonstration. Comme nous l'avons dit, pour prouver ce résultat nous allons avoir recours à une formulation spécifique du principe des tiroirs infini. Informellement, ce principe dit que si une infinité d'objets est mise dans un nombre fini de tiroirs, alors au moins un tiroir contient une infinité d'objets. Même s'il est évident, ce principe n'a pas de justification constructive. En effet, aucune observation finie du processus infini qui consiste à mettre des objets dans des tiroirs ne permet de déterminer quel tiroir est utilisé une infinité de fois.

Dans notre problème concret, les objets en nombre infini sont les niveaux d'observation possibles du graphe et les tiroirs sont les permutations possibles pour *sons* g_1 et *sons* g_2. Comme *sons* g_1 et *sons* g_2 sont finis, le nombre de permutations possibles l'est aussi puisque

nous ne considérons que les permutations au premier niveau. Cependant, prendre les permutations possibles comme les "tiroirs" nous impose de connaître ces permutations et donc d'être capables de les manipuler. On doit donc utiliser *iperm_ind_skel* (Définition 4.24).

On n'aura besoin du principe des tiroirs infini que pour les types finis de la forme *skel_type m*. Comme on l'a dit, on va devoir ajouter ce principe comme un axiome au dessus de la théorie des types intuitionniste CIC sous-jacente à Coq. On verra plus bas une justification possible de ce principe en montrant qu'il est une conséquence de lois de la logique classique. Initialement, nous avions utilisé une version assez générale du principe des tiroirs, formulée comme suit :

$$\forall m\ \forall f : \mathbb{N} \to \mathit{skel\_type}\ m \exists s_0 : \mathit{skel\_type}\ m, (\forall n \exists n', n' \geq n \wedge f\ n' = s_0)$$

Cependant, cette version était trop générale pour qu'on puisse la justifier avec des lois de la logique classique, puisqu'elle demandait l'utilisation d'un principe de choix. Nous avons donc finalement utilisé une version plus spécifique du principe des tiroirs infini :

Axiome 1 (Principe des tiroirs infini)**.**

$$\begin{array}{l}\forall m\ \forall P : \mathbb{N} \to \mathit{skel\_type}\ m \to \mathit{Prop}, \\ \qquad (\forall n \exists s : \mathit{skel\_type}\ m, P\ n\ s) \quad \Rightarrow \quad \exists s_0 : \mathit{skel\_type}\ m, (\forall n \exists n', n' \geq n \wedge P\ n'\ s_0)\end{array}$$

Soit $H_1 : \mathit{GTPerm}\ g_1\ g_2$, ou encore $H_1 : \forall n, g_1 \equiv_{R,n} g_2$. On veut montrer que

$$\mathit{iperm\_ind}_{\mathit{GTPerm}_R}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)$$

On veut utiliser le principe des tiroirs infini avec *iperm_ind_skel*. On doit donc prouver que $H_{lg} : \mathit{lg}(\mathit{sons}\ g_1) = \mathit{lg}(\mathit{sons}\ g_2)$. D'après la Propriété 6.1.2 et H_1 appliqués à 1, on sait que $\mathit{iperm\_ind}_{\equiv_{R,0}}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)$ et donc d'après le Lemme 4.17, $\mathit{lg}(\mathit{sons}\ g_1) = \mathit{lg}(\mathit{sons}\ g_2)$.

Pour utiliser l'Axiome 1 avec $P := \mathit{iperm\_ind\_skel}_{\equiv_{R,n}}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)\ H_{lg}$, on doit encore prouver que :

$$H_2 : \forall n \exists s : \mathit{skel\_type}(\mathit{lg}(\mathit{sons}\ g_1)), \mathit{iperm\_ind\_skel}_{\equiv_{R,n}}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)\ H_{lg}\ s$$

D'après le Lemme 4.39, il nous suffit de prouver que $\mathit{iperm\_ind}_{\equiv_{R,n}}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)$ ce que l'on montre grâce à la Propriété 6.1.2 utilisée avec H_1 appliquée à $n+1$.

On peut donc maintenant utiliser le principe des tiroirs infini avec H_2. Cela nous donne s_0 tel que $H_3 : \forall n \exists m, n \leq m \wedge \mathit{iperm\_ind\_skel}_{\equiv_{R,m}}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)\ H_{lg}\ s_0$. Ici, s_0 est déjà la "bonne" permutation, valable pour tous les niveaux.

On veut toujours montrer que $\mathit{iperm\_ind}_{\mathit{GTPerm}_R}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)$. D'après le Lemme 4.39, il nous suffit de montrer que :

$$\mathit{iperm\_ind\_skel}_{\mathit{GTPerm}_R}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)\ H_{lg}\ s_0$$

Et d'après le Lemme 4.41, il nous suffit de montrer que :

$$\forall i, \mathit{iperm\_ind\_skel}_{\equiv_{R,i}}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)\ H_{lg}\ s_0$$

Grâce à H_3 on obtient m tel que

$$H_4 : i \leq m \quad H_5 : \mathit{iperm\_ind\_skel}_{\equiv_{R,m}}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)\ H_{lg}\ s_0$$

D'après le Lemme 4.40 utilisé avec H_5, il nous suffit de montrer que $\equiv_{R,m} \subseteq \equiv_{R,i}$, ce qui est vrai d'après le Lemme 6.21 et H_4. □

Remarque 6.4. *Nous avons ici présenté le principe des tiroirs infini comme un axiome de notre développement. En réalité, dans le code Coq, les lemmes sont énoncés en prenant comme hypothèse une instance de ce principe, ce qui revient au même.*

Pour montrer que l'utilisation du principe des tiroirs infini est raisonnable, nous allons voir qu'on peut le déduire du principe classique de l'élimination de la double négation. Ceci implique en particulier que l'axiome est prouvable dans Coq auquel on a ajouté le principe du tiers exclu.

Justification du principe des tiroirs infini avec élimination de la double négation On veut ici montrer que si on suppose vrai le principe de l'élimination de la double négation (qui est un principe de la logique classique), alors on peut démontrer le principe des tiroirs sous la forme dans laquelle il est présenté dans l'Axiome 1.

On exprime le principe de l'élimination de la double négation de la façon suivante :

Définition 6.12 (Élimination de la double négation)**.** $DNE := \forall P, \neg\neg P \Rightarrow P$

On peut montrer que supposer ce principe vrai, rend le lemme suivant vrai également :

Lemme 6.24. $DNE \Rightarrow (\forall P, \neg(\forall t, \neg(Pt)) \Rightarrow \exists t, P\ t)$

Démonstration. Preuve détaillée en page 199. □

Avant de prouver que le principe de l'élimination de la double négation permet de démontrer le principe des tiroirs infini tel qu'exprimé dans l'Axiome 1, nous allons montrer qu'il permet de le démontrer dans une forme simplifiée, puis nous nous servirons de ce résultat intermédiaire pour prouver qu'on peut déduire la forme de l'Axiome 1 de cette forme plus simple. Ici les "tiroirs" sont les éléments de *Fin m* (qui sont en nombre fini, puisqu'il n'y en a que m). Nous allons exprimer le principe des tiroirs infini sous une forme générale, que nous pourrons instancier dans le cas plus simple de *Fin* ou celui de *skel_type* :

Définition 6.13 (Version plus générale du principe des tiroirs infini)**.**

$$IPPGen\ T := \forall P : \mathbb{N} \to T \to Prop, (\forall n \exists s : T, P\ n\ s) \Rightarrow \exists s_0 : T, (\forall n \exists n', n' \geq n \wedge P\ n'\ s_0)$$

Maintenant la version avec *Fin* peut simplement se définir comme suit :

Définition 6.14. $IPPFin := \forall n, IPPGen(Fin\ n)$

Et on peut également démontrer que l'Axiome 1 peut s'écrire sous la forme $\forall n, IPPGen(skel\_type\ n)$.

On peut donc montrer que

Lemme 6.25. $DNE \Rightarrow IPPFin$

Démonstration. Soient $H_1 : DNE$ et $H_2 : \forall n \exists s, P\ n\ s$. On veut montrer que :

$$\exists i : Fin\ m, (\forall n \exists n', n' \geq n \wedge P\ n'\ i)$$

D'après le Lemme 6.24, il nous suffit de montrer que :

$$\neg(\forall i, \neg(\forall n \exists n', n' \geq n \wedge P\ n'\ i))$$

Raisonnons par l'absurde. Supposons donc que $H_3 : \forall i, \neg(\forall n \exists n', n' \geq n \wedge P\ n'\ i)$ et montrons qu'on arrive à une contradiction.

Montrons tout d'abord une variante de H_3 où la négation a été "poussée" à l'intérieur de la formule :

$$H_4 : \forall i, (\exists k \forall n, P\ n\ i \Rightarrow \neg n \geq k)$$

D'après H_3 appliqué à i on a $H_5 : \neg(\forall n \exists n', n' \geq n \wedge P\ n'\ i)$. D'après le Lemme 6.24, il nous suffit de prouver que :

$$\neg(\forall k, \neg(\forall n, P\ n\ i \Rightarrow \neg n \geq k))$$

Supposons que $H_6 : \forall k, \neg(\forall n, P\ n\ i \Rightarrow \neg n \geq k)$ et montrons qu'on arrive à une contradiction. On va montrer que $\forall k \exists n', n' \geq k \wedge P\ n'\ i$, ce qui est en contradiction avec H_5. Soit k fixé, d'après le Lemme 6.24, il nous suffit de montrer que $\neg(\forall n', \neg(n' \geq k \wedge P\ n'\ i))$. Pour cela on va encore raisonner par l'absurde. On va donc supposer que $H_7 : \forall n', \neg(n' \geq k \wedge P\ n'\ i)$ et montrer qu'on arrive à une contradiction. D'après H_6 appliqué à k on a $H_8 : \neg(\forall n, P\ n\ i \Rightarrow \neg n \geq k)$. Montrons que $\forall n, P\ n\ i \Rightarrow \neg n \geq k$, ce qui est en contradiction avec H_8. Soit $H_9 : P\ n\ i$. On veut montrer que $\neg n \geq k$. Supposons donc encore que $H_{10} : n \geq k$ et montrons qu'on arrive à une contradiction. D'après H_7 appliqué à n, on a $\neg(n \geq k \wedge P\ n\ i)$, ce qui est en contradiction avec H_9 et H_{10}.

Récapitulons, nous avons donc montré que :

$$H_4 : \forall i, (\exists k \forall n, P\ n\ i \Rightarrow \neg n \geq k)$$

Toutes les autres hypothèses sont devenues inutiles. Nous n'avons plus besoin que de H_1, H_2 et H_4.

On peut montrer sur *Fin* la propriété suivante de choix fonctionnel :

Lemme 6.26.

$$\forall m, (\forall R : Fin\ m \to \mathbb{N} \to Prop, (\forall x \exists y, R\ x\ y) \Rightarrow \exists f : Fin\ m \to \mathbb{N}, (\forall x, R\ x\ (f\ x)))$$

Démonstration. Preuve détaillée en page 199. □

On aura aussi besoin d'obtenir un élément supérieur ou égal au maximum d'une fonction de type $Fin\ n \to \mathbb{N}$. On appelle *MaxFin* la fonction qui renvoie ce maximum. On peut, par exemple, la définir comme suit :

Définition 6.15. $MaxFin\ n\ (f : Fin\ n \to \mathbb{N}) := max\_list\_nat\ (map\ f\ (makeListFin\ n))$

Elle est telle que :

Propriété 6.2. $\forall n\ (f : Fin\ n \to \mathbb{N}), (\forall i, MaxFin\ f \geq f\ i)$

Démonstration. Preuve détaillée en page 200. □

Revenons donc à la preuve initiale. On cherche une contradiction avec nos hypothèses. Le Lemme 6.26 appliqué à H_4 nous donne f, qui nous permet de déterminer k en fonction de i, tel que $H_5 : \forall x\ n, P\ n\ x \Rightarrow \neg(n \geq f\ x)$. On va montrer que $H_6 : \forall n, \neg(n \geq MaxFin\ f)$. Voyons tout d'abord comment cela nous permet de terminer la preuve. Si on applique H_6 à $MaxFin\ f + 1$, on a alors $\neg\ (MaxFin\ f + 1 \geq MaxFin\ f)$, ce qui est trivialement faux. Il nous reste donc seulement à prouver que :

$$\forall n, \neg(n \geq MaxFin\ f)$$

Soit n fixé. Supposons que $H_7 : n \geq MaxFin\ f$ et montrons qu'on arrive à une contradiction. H_2 nous donne i tel que $H_8 : P\ n\ i$. Grâce à H_5 appliqué à H_8 on obtient $H_9 : \neg(n \geq f\ i)$. On va montrer que $n \geq f\ i$, ce qui est en contradiction avec H_9. On prouve cela par transitivité de $\geq$ avec $MaxFin\ f$. On doit montrer que

1. $f\ i \leq MaxFin\ f$: ce qui est vrai d'après la Propriété 6.2.
2. $MaxFin\ f \leq n$: ce qui est vrai d'après H_7.

□

Pour démontrer que le principe de l'élimination de la double négation permet de prouver le principe des tiroirs infini tel qu'énoncé dans l'Axiome 1, nous allons montrer que les éléments de *skel_type* sont en bijection avec les éléments de *Fin*.

Lemme 6.27. $\forall n \exists m\ (f : skel\_type\ n \to Fin\ m)\ (g : Fin\ m \to skel\_type\ n), bij\ f\ g$

Démonstration. On raisonne par induction sur n.

[Cas 0**]** On veut montrer que $\exists m\ (f : skel\_type\ 0 \to Fin\ m)\ (g : Fin\ m \to skel\_type\ 0), bij\ f\ g$. Par définition, *skel_type* 0 a un seul élément, tt, on prend donc $m := 1$, $f := \lambda c.first\ 0$ et $g := \lambda i.tt$. On doit montrer que $bij\ f\ g$, c'est-à-dire que :

1. $\forall t, tt = t$: c'est vrai car *skel_type* 0 a un seul élément, tt.
2. $\forall u, first\ 0 = u$: c'est vrai car *Fin* 1 a un seul élément, *first* 0.

[Cas $n+1$**]** L'hypothèse d'induction nous donne m, $f_0 : skel\_type\ n \to Fin\ m$ et $g_0 : Fin\ m \to skel\_type\ n$ tels que :

$$IH : bij\ f_0\ g_0$$

Et on veut montrer que

$$\exists m\ (f : skel\_type\ (n+1) \to Fin\ m)\ (g : Fin\ m \to skel\_type\ (n+1)), bij\ f\ g$$

Dans $skel\_type\ (n+1)$, il y a $(n+1) * (n+1)$ fois plus d'éléments que dans $skel\_type\ n$. En effet, les éléments de $skel\_type\ (n+1)$ sont constitués d'un élément de type $skel\_type\ n$ et d'un couple d'éléments de $Fin\ (n+1)$. Comme $Fin\ (n+1)$ a $n+1$ éléments, il y a $(n+1) * (n+1)$ possibilités pour le couple d'éléments de $Fin\ (n+1)$ pour chaque élément de $skel\_type\ n$. Donc, s'il y a m éléments dans $skel\_type\ n$, il y en a $(n+1) * (n+1) * m$ dans $skel\_type\ (n+1)$. On instancie donc l'existencielle avec $(n+1) * (n+1) * m$ et on doit montrer que :

$$\begin{array}{ll} \exists & (f : skel\_type\ (n+1) \to Fin\ ((n+1) * (n+1) * m)) \\ & (g : Fin\ ((n+1) * (n+1) * m) \to skel\_type\ (n+1)), \quad bij\ f\ g \end{array}$$

Pour donner les fonctions de conversion, on a besoin d'une conversion entre $Fin\ n * Fin\ m$ et $Fin(n * m)$. On appelle *FnFmFnm* la fonction de type $Fin\ n * Fin\ m \to Fin(n * m)$ et *FnmFnFm* celle de type $Fin(n * m) \to Fin\ n * Fin\ m$. Pour *FnFmFnm*, on fait un codage vers la base m. L'élément de type $Fin\ n$ devient l'élément de "poids fort" et l'élément de type $Fin\ m$, celui de "poids faible". On définit *FnFmFnm* telle que :

$$\forall (i_1 : Fin\ n)\ (i_2 : Fin\ m), decode\ (FnFmFnm(i_1, i_2)) = decode\ i_1 * m + decode\ i_2$$

Et inversement, on fait la conversion dans l'autre sens pour *FnmFnFm*. Dans le couple renvoyé par la fonction, on peut considérer que le premier élément correspond au quotient de la division euclidienne de *decode i* par m et que le second élément correspond à son reste. On définit donc *FnmFnFm* telle que :

$$\forall (i : Fin\ (n * m)), decode\ i = decode(fst(FnmFnFm\ i)) * m + decode(snd(FnmFnFm\ i))$$

Où *fst* et *snd* sont les projections classiques sur un couple, telles que : $fst(x,y) = x$ et $snd(x,y) = y$.
FnFmFnm et *FnmFnFm* sont aussi telles que :

Propriété 6.3.

$$H_1 : decode\ i < m \Rightarrow FnmFnFm\ (i : Fin\ ((n+1) * m)) = (first\ n, code\ H) \quad (6.3.1)$$

$$H_1 : m \leq decode\ i \Rightarrow FnmFnFm\ (i : Fin\ ((n+1) * m)) = (succ(fst(FnmFnFm(code\ H_2))),\ snd(FnmFnFm(code\ H_2))) \text{ avec } H_2 : decode\ i - m < m * n \quad (6.3.2)$$

$$H_1 : decode\ i_1 = 0 \Rightarrow FnFmFnm\ (i_1 : Fin\ (n+1), i_2 : Fin\ m) = code\ H_2 \text{ avec } H_2 : decode\ i_2 < (n+1) * m \quad (6.3.3)$$

$$H_1 : 0 < decode\ i_1 \Rightarrow FnFmFnm\ (i_1 : Fin\ (n+1), i_2 : Fin\ m) = code\ H_2 \text{ avec } H_2 : decode\ (FnFmFnm(getcons\ H_1, i_2)) + m < (n+1) * m \quad (6.3.4)$$

$$bij\ FnFmFnm\ FnmFnFm \quad (6.3.5)$$

On prend alors :

$$f := \lambda(i_1, i_2, s).FnFmFnm(i_1, FnFmFnm\ (i_2, f_0\ s))$$
$$g := \lambda c.(i_1, fst(FnmFnFm\ i_2), g_0(snd(FnmFnFm\ i_2))) \quad \text{avec } (i_1, i_2) := FnmFnFm\ c$$

On doit encore montrer que $bij\ f\ g$. Ici la preuve se résume simplement à appliquer plusieurs fois la Propriété 6.3.5. Elle est simple mais assez lourde en notation. Elle n'est donc pas développée ici.

□

On veut également prouver que *IPPGen* préserve la bijectivité :

Lemme 6.28. $\forall (f : T \to U)\ (g : U \to T), bij\ f\ g \wedge IPPGen\ T \Rightarrow IPPGen\ U$

Démonstration. Preuve détaillée en page 200. □

On peut maintenant montrer ce que l'on voulait :

Lemme 6.29. $DNE \Rightarrow \forall n, IPPGen(skel\_type\ n)$

Démonstration. Soit $H_1 : DNE$. On veut montrer que $IPPGen(skel\_type\ n)$. Le Lemme 6.27 nous donne m, $f : skel\_type\ n \to Fin\ m$ et $g : Fin\ m \to skel\_type\ n$ tels que $H_2 : bij\ f\ g$. D'après le Lemme 3.11 on peut transformer H_2 en $bij\ g\ f$. D'après le Lemme 6.28 appliqué avec H_2 il nous suffit de montrer que $IPPGen\ (Fin\ m)$. Ce qu'on prouve avec le Lemme 6.25 et H_1. □

Justification du principe des tiroirs infini avec tiers exclu On peut également prouver que si on suppose le principe du tiers exclu vrai, alors l'Axiome 1 est démontrable. Il est bien connu qu'on peut déduire le principe de l'élimination de la double négation du principe du tiers exclu, et donc l'Axiome 1 aussi.

On exprime le principe du tiers exclu de la façon suivante :

Définition 6.16 (Tiers exclu). $TiersEx := \forall P, P \vee \neg P$

Remarque 6.5. *Pour $\vee$, il s'agit ici exceptionnellement de la disjonction "normale" de la logique, et non de la version constructive telle qu'utilisée pour Dec.*

Montrons donc qu'on peut déduire le principe de l'élimination de la double négation du principe du tiers exclu.

Lemme 6.30. *TiersEx $\Rightarrow$ DNE*

Démonstration. Preuve détaillée en page 200. □

La preuve qu'on peut déduire l'Axiome 1 du principe du tiers exclu est donc maintenant triviale :

Lemme 6.31. *TiersEx $\Rightarrow$ Axiome 1*

Démonstration. Il suffit de combiner les lemmes 6.29 et 6.30. □

6.1.4 Avec *iperm_bij*

Pour éviter les problèmes liés à la condition de garde on peut aussi utiliser *iperm_bij* pour définir la relation sur *Graph*. Comme *iperm_bij* n'est pas inductive, cela simplifie les choses. Cependant, comme on l'a déjà dit, on considère que *iperm_ind* reflète mieux l'intuition des permutations et c'est pour cela qu'on a développé tout le travail présenté précédemment.

On appelle *GPerm_bij* la relation sur *Graph* utilisant *iperm_bij*. Elle est définie comme suit :

Définition 6.17 (*GPerm_bij*, vue coinductivement)**.**

$$\frac{g_1\ g_2 : Graph\ T \qquad R\ (label\ g_1)\ (label\ g_2) \qquad iperm\_bij_{GPerm\_bij_R}\ (sons\ g_1)\ (sons\ g_2)}{GPerm\_bij_R\ g_1\ g_2}$$

On peut facilement démontrer que *GPerm_bij* préserve l'équivalence.

Lemme 6.32 (*GPerm_bij* préserve la réflexivité)**.** *R réflexive $\Rightarrow \forall g, GPerm\_bij_R\ g\ g$*

Démonstration. On raisonne par coinduction. L'hypothèse de coinduction est *CH* : $\forall g, GPerm\_bij_R\ g\ g$. D'après la Définition 6.17, il nous suffit de montrer que :

1. $R\ (label\ g)\ (label\ g)$: c'est vrai par réflexivité de R.
2. $iperm\_bij_{GPerm\_bij_R}\ (sons\ g)\ (sons\ g)$: la première idée serait ici d'utiliser le Lemme 4.42. Il nous resterait alors à montrer que $\forall g, GPerm\_bij_R\ g\ g$, ce qui est *CH*. Cependant, ici, l'appel à l'hypothèse de coinduction ne serait pas gardé. En effet, *CH* serait "argument" d'un lemme et non d'un constructeur directement. On doit donc développer la preuve de réflexivité de *iperm_bij*. D'après la Définition 4.25, on doit montrer que :

   $$\exists f\ g, bij\ f\ g\ \wedge\ (\forall i, GPerm\_bij_R\ (fct\ (sons\ g)\ i)\ (fct\ (sons\ g)\ (f\ i)))$$

   On prend $f = g = \lambda i.i$. On montre aisément (comme dans la preuve du Lemme 4.42) que $bij\ f\ g$. Il nous reste donc à montrer que $\forall i, GPerm\_bij_R\ (fct\ (sons\ g)\ i)\ (fct\ (sons\ g)\ i))$ ce qui est vrai d'après *CH*.

□

Lemme 6.33 (*GPerm_bij* préserve la symétrie)**.**

$$R \text{ symétrique} \Rightarrow (\forall g_1\ g_2, GPerm\_bij_R\ g_1\ g_2 \Rightarrow GPerm\_bij_R\ g_2\ g_1)$$

Démonstration. On raisonne également par coinduction. L'hypothèse de coinduction est $CH : \forall g_1\ g_2, GPerm\_bij_R\ g_1\ g_2 \Rightarrow GPerm\_bij_R\ g_2\ g_1$. D'après les Définitions 6.17 et 4.25, l'hypothèse $GPerm\_bij_R\ g_1\ g_2$ nous donne $H_1 : R\ (label\ g_1)\ (label\ g_2)$ et f et g telles que :

$$H_2 : bij\ f\ g \quad H_3 : \forall i, GPerm\_bij_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ (f\ i)))$$

D'après la Définition 6.17 on doit montrer que :

1. $R\ (label\ g_2)\ (label\ g_1)$: c'est vrai par symétrie de R et H_1.
2. $iperm\_bij_{GPerm\_bij_R}\ (sons\ g_2)\ (sons\ g_1)$: pour les mêmes raisons que précédemment, on ne peut pas utiliser le Lemme 4.43. D'après le Lemme 3.11 on sait que $H_2' : bij\ g\ f$. D'après la Définition 4.25 appliquée à H_2' il nous suffit de démontrer que :

   $$\forall i, GPerm\_bij_R\ (fct\ (sons\ g_2)\ i)\ (fct\ (sons\ g_1)\ (g\ i)))$$

   D'après CH il nous suffit de montrer que :

   $$GPerm\_bij_R\ (fct\ (sons\ g_1)\ (g\ i)))\ (fct\ (sons\ g_2)\ i)$$

   D'après la Définition 3.6, H_2 nous donne $H_4 : \forall i, f(g\ i) = i$. Donc il nous suffit de démontrer que :

   $$GPerm\_bij_R\ (fct\ (sons\ g_1)\ (g\ i)))\ (fct\ (sons\ g_2)\ (f(g\ i)))$$

   Ce qui est vrai d'après H_3. □

Lemme 6.34 (*GPerm_bij* préserve la transitivité)**.**

$$R \text{ transitive} \Rightarrow (\forall g_1\ g_2\ g_3, GPerm\_bij_R\ g_1\ g_2 \wedge GPerm\_bij_R\ g_2\ g_3 \Rightarrow GPerm\_bij_R\ g_1\ g_3)$$

Démonstration. On raisonne également par coinduction. L'hypothèse de coinduction est $CH : \forall g_1\ g_2\ g_3, GPerm\_bij_R\ g_1\ g_2 \wedge GPerm\_bij_R\ g_2\ g_3 \Rightarrow GPerm\_bij_R\ g_1\ g_3$. D'après les Définitions 6.17 et 4.25, les hypothèses $GPerm\_bij_R\ g_1\ g_2$ et $GPerm\_bij_R\ g_2\ g_3$ nous donnent $H_1 : R\ (label\ g_1)\ (label\ g_2)$ et $H_2 : R\ (label\ g_2)\ (label\ g_3)$ et f_1, f_1', f_2 et f_2' telles que :

$$H_3 : bij\ f_1\ f_1' \quad H_4 : \forall i, GPerm\_bij_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ (f_1\ i)))$$
$$H_5 : bij\ f_2\ f_2' \quad H_6 : \forall i, GPerm\_bij_R\ (fct\ (sons\ g_2)\ i)\ (fct\ (sons\ g_3)\ (f_2\ i)))$$

D'après la Définition 6.17 on doit montrer que :

1. $R\ (label\ g_1)\ (label\ g_3)$: c'est vrai par transitivité de R avec H_1 et H_2.
2. $iperm\_bij_{GPerm\_bij_R}\ (sons\ g_1)\ (sons\ g_3)$: on ne peut toujours pas utiliser le Lemme 4.44, on va donc refaire la preuve ici. D'après la Définition 4.25 et le Lemme 3.12, il nous suffit de montrer que :

   $$\forall i, GPerm\_bij_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_3)\ (f_2(f_1\ i)))$$

   D'après CH, il nous suffit de prouver que :
   (a) $GPerm\_bij_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ (f_1\ i))$: on utilise H_4.
   (b) $GPerm\_bij_R\ (fct\ (sons\ g_2)\ (f_1\ i))\ (fct\ (sons\ g_3)\ (f_2(f_1\ i)))$: on utilise H_6. □

Lemme 6.35 (*GPerm_bij* préserve l'équivalence)**.** *R équivalence* $\Rightarrow$ *GPerm_bij*$_R$ *équivalence*

Démonstration. La preuve est une combinaison des trois lemmes précédents. □

On voudrait maintenant prouver que cette nouvelle définition est en fait équivalente aux précédentes que nous avons présentées (*GPerm*, *GPerm_imp*, *GPerm_mend*). On sait déjà que *iperm_ind* $\Leftrightarrow$ *iperm_bij* et que *GPerm* $\Rightarrow$ *GPerm_imp* $\Leftrightarrow$ *GPerm_mend*. On voudrait maintenant montrer que *GPerm* $\Leftrightarrow$ *GPerm_bij*. En fait, comme on l'a vu, *GPerm* est compliquée à manipuler et très bridée par la condition de garde. On pourra montrer, comme précédemment, que *GPerm* $\Rightarrow$ *GPerm_bij* mais pas le sens inverse.

On va donc se contenter de prouver que *GPerm_imp* $\Leftrightarrow$ *GPerm_bij*. Comme on va le voir, le sens *GPerm_bij* $\Rightarrow$ *GPerm_imp* se résout assez facilement, directement. En revanche le sens inverse est plus compliqué. Pour le démontrer on va faire appel à une version équivalente de la relation *GPerm_bij* mais définie dans le style de Mendler.

Cette relation est définie comme suit :

Définition 6.18 (*GPerm_bij_mend*, vue coinductivement)**.**

$$\frac{\mathcal{R} \subseteq \mathit{GPerm\_bij\_mend}_R \qquad R\ (\mathit{label}\ g_1)\ (\mathit{label}\ g_2) \qquad \mathit{iperm\_bij}_{\mathcal{R}}\ (\mathit{sons}\ g_1)\ (\mathit{sons}\ g_2)}{\mathit{GPerm\_bij\_mend}_R\ g_1\ g_2}$$

Montrons tout d'abord que *GPerm_bij_mend* est effectivement équivalente à *GPerm_bij* (on peut faire cette preuve parce que *iperm_bij* est définie non-inductivement, contrairement à *iperm_ind*).

Lemme 6.36. $\forall g_1\ g_2, \mathit{GPerm\_bij}_R\ g_1\ g_2 \Leftrightarrow \mathit{GPerm\_bij\_mend}_R\ g_1\ g_2$

Démonstration. Preuve détaillée en page 201. □

Maintenant, on prouve que les deux versions dans le style de Mendler (*GPerm_mend* et *GPerm_bij_mend*) sont équivalentes. Avec ce résultat, on aura toutes les briques pour montrer que *GPerm_bij* est équivalente à *GPerm_imp*.

Lemme 6.37. $\forall g_1\ g_2, \mathit{GPerm\_mend}_R\ g_1\ g_2 \Leftrightarrow \mathit{GPerm\_bij\_mend}_R\ g_1\ g_2$

Démonstration. Preuve détaillée en page 201. □

A l'aide de ces résultats, on peut démontrer que *GPerm_bij* est équivalente à *GPerm_imp*.

Lemme 6.38. $\forall g_1\ g_2, \mathit{GPerm\_bij}_R\ g_1\ g_2 \Leftrightarrow \mathit{GPerm\_imp}_R\ g_1\ g_2$

Démonstration. On a

$$\begin{array}{lcll} \mathit{GPerm\_bij}_R\ g_1\ g_2 & \Leftrightarrow & \mathit{GPerm\_bij\_mend}_R\ g_1\ g_2 & \text{(d'après Lemme 6.36)} \\ & \Leftrightarrow & \mathit{GPerm\_mend}_R\ g_1\ g_2 & \text{(d'après Lemme 6.37)} \\ & \Leftrightarrow & \mathit{GPerm\_imp}_R\ g_1\ g_2 & \text{(d'après Lemme 6.10)} \end{array}$$

Remarque 6.6. *La preuve que* $\mathit{GPerm\_bij}_R\ g_1\ g_2 \Rightarrow \mathit{GPerm\_imp}_R\ g_1\ g_2$ *peut aussi se faire directement (c'est-à-dire, directement par coinduction) mais pour l'autre sens il y a un problème de garde et on est obligé de passer par la version dans le style de Mendler.*

□

Enfin, comme annoncé on peut montrer que $GPerm \Rightarrow GPerm\_bij$. Cela reste cependant plutôt anecdotique puisque comme on l'a vu, il est difficile de travailler avec $GPerm$.

Lemme 6.39. $\forall g_1\ g_2, GPerm_R\ g_1\ g_2 \Rightarrow GPerm\_bij_R\ g_1\ g_2$

Démonstration. Ici encore on pourrait faire la preuve directement par coinduction, mais comme tous les outils sont en place, on va se servir des lemmes précédents. On a :

$$\begin{array}{lll} GPerm_R\ g_1\ g_2 & \Rightarrow & GPerm\_imp_R\ g_1\ g_2 \quad \text{(d'après Lemme 6.5)} \\ & \Rightarrow & GPerm\_bij_R\ g_1\ g_2 \quad \text{(d'après Lemme 6.38)} \end{array}$$

□

Pour résumer, la Figure 6.9 donne une vue d'ensemble des relations que l'on a entre les différentes relations sur *Graph* qui incluent les permutations. On a donc obtenu 5 relations équivalentes qui peuvent toutes remplir la même fonction et une autre, la relation initiale *GPerm*, qui est trop rigide la plupart du temps, lorsqu'on veut prouver des résultats généraux. Dans le cas d'exemples cependant, on peut très bien travailler avec *GPerm* (puisque typiquement on n'a pas d'induction imbriquée dans les exemples).

Il est intéressant de noter que s'il est clair que la préservation de l'équivalence est très simple à prouver sur *GPerm_bij*, la preuve d'équivalence avec *GTPerm* n'aurait pas été simplifiée parce que les problèmes théoriques sous-jacents restent les mêmes.

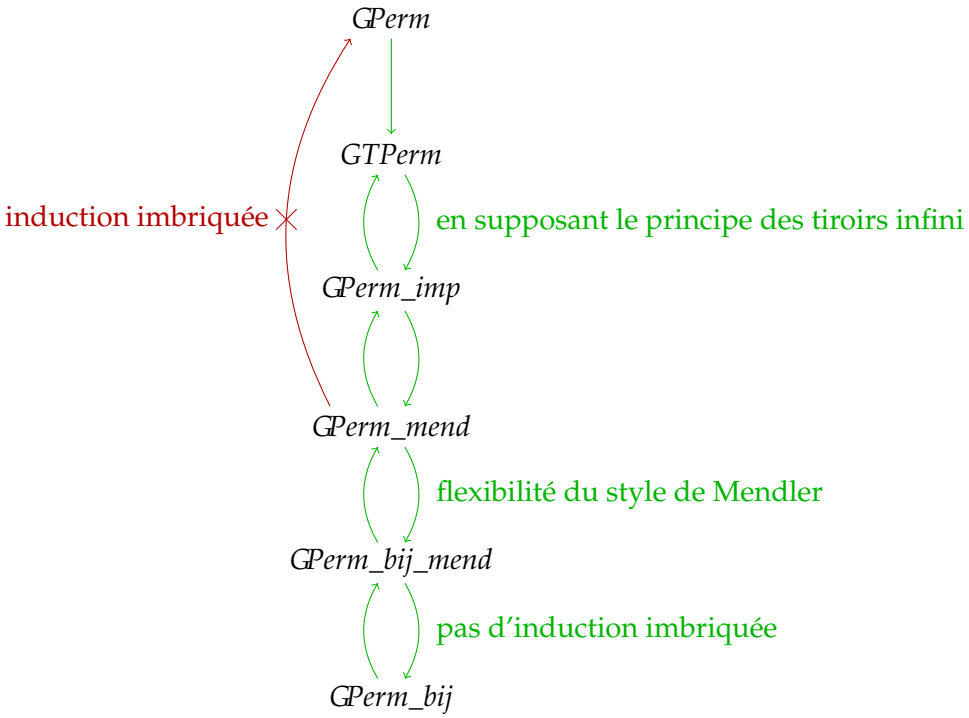


Figure 6.9 — Relations entre les relations sur *Graph* incluant les permutations

6.2 Une relation sans ordre dans les nœuds

La Section 6.1 résout le problème des permutations dans les fils d'un nœud, comme présenté dans la Figure 6.1. Nous allons maintenant résoudre le problème présenté dans la Figure 6.2,

celui du changement de racine.

6.2.1 L'idée

Le type *Graph* nous permet de représenter des graphes enracinés. C'est-à-dire que tous les nœuds sont atteignables depuis la racine. Dit autrement, cela signifie qu'il y a un chemin de la racine vers chaque nœud du graphe. Si donc on peut changer de racine dans la représentation du graphe, cela veut dire qu'il y a un chemin de la première racine vers la deuxième, et un chemin de la deuxième vers la première, cést-à-dire que les graphes sont inclus l'un dans l'autre. On peut donc "retomber" sur le premier graphe en parcourant le second. Par exemple, si on déplie les graphes de la Figure 6.2 une fois, comme montré dans la Figure 6.10, on retrouve en effet l'autre graphe (partie encadrée dans la figure).

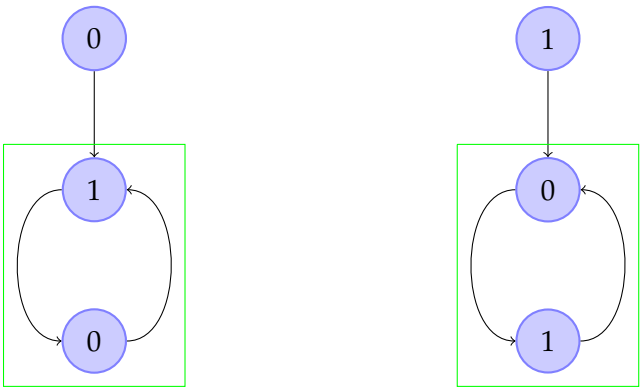


Figure 6.10 — Graphes de la Figure 6.2 dépliés une fois

Il est cependant important de noter que ce changement de racine ne peut intervenir qu'au plus haut niveau. En effet, on ne voudrait en aucun cas que les graphes de la Figure 6.11 soient équivalents. Dans le premier cas on peut atteindre le nœud 0 depuis le nœud 2 et dans le deuxième, c'est le nœud 1 qu'on atteint depuis le nœud 2. Ce sont donc deux graphes complètement distincts. Notre relation doit donc se préserver de cet écueil.

Figure 6.11 — Graphes dont le cycle intérieur à été tourné

Pour résumer, on va dire que deux graphes sont équivalents s'ils sont inclus non strictement l'un dans l'autre, en prenant en compte les permutations.

Nous allons utiliser la relation d'inclusion $GinG^*$. Cette fois-ci, nous allons utiliser pour la relation de base la relation définie précédemment prenant en compte les permutations : *GPerm_imp*. On appelle *GinGP* l'instanciation de $GinG^*$ avec *GPerm_imp*.

Définition 6.19 (*GinGP*). $GinGP_R := GinG^*_{GPerm\_imp_R}$

On peut maintenant montrer l'équivalent du Lemme 5.9 pour *GinGP*.

Lemme 6.40. $\forall g_1\ g_2, GinGP_R\ g_1\ g_2 \Rightarrow \forall i_1, GinGP_R\ (fct\ (sons\ g_1)\ i_1)\ g_2$

Démonstration. Preuve détaillée en page 202. □

On peut également montrer que *GinGP* préserve la transitivité.

Lemme 6.41. R *transitive* $\Rightarrow (\forall g_1\ g_2\ g_3, GinGP_R\ g_1\ g_2 \wedge GinGP_R\ g_2\ g_3 \Rightarrow GinGP_R\ g_1\ g_3)$

Démonstration. Preuve détaillée en page 203. □

6.2.2 *GeqPerm*

Grâce aux définitions précédentes, nous pouvons donner la définition de la relation finale sur *Graph*. Nous l'appelons *GeqPerm*. Cette relation vérifie que son premier paramètre est bien inclus (au sens de *GinGP*) dans le second, et inversement.

Définition 6.20 (*GeqPerm*)**.** $\forall g_1\ g_2,\ GeqPerm_R\ g_1\ g_2 \Leftrightarrow GinGP_R\ g_1\ g_2 \wedge GinGP_R\ g_2\ g_1$

On voit qu'il était nécessaire ici d'avoir une inclusion non stricte pour le cas où il n'y aurait pas de changement de racine. On veut maintenant montrer que *GeqPerm* préserve l'équivalence. Pour cela, on montre séparément la préservation de la réflexivité, de la symétrie et de la transitivité. Les preuves sont ici très simples : en effet, *GinGP* préserve la réflexivité, de par sa définition *GeqPerm* est symétrique et enfin la preuve de la transitivité est aisée grâce au Lemme 6.41.

Lemme 6.42 (*GeqPerm* préserve la réflexivité)**.** R *réflexive* $\Rightarrow \forall g, GeqPerm_R\ g\ g$

Démonstration. D'après la Définition 6.20, on doit montrer deux fois que $GinGP_R\ g\ g$. On ne fait la preuve qu'une fois ici. D'après la règle dirG de la Définition 5.5 il nous suffit de prouver que $GPerm\_imp_R\ g\ g$, ce qui est vrai d'après le Lemme 6.6. □

Lemme 6.43 (*GeqPerm* préserve la symétrie)**.**

$$R \text{ symétrique} \Rightarrow (\forall g_1\ g_2, GeqPerm_R\ g_1\ g_2 \Rightarrow GeqPerm_R\ g_2\ g_1)$$

Démonstration. L'hypothèse $GeqPerm_R\ g_1\ g_2$ nous donne H_1 : $GinGP_R\ g_1\ g_2$ et H_2 : $GinGP_R\ g_2\ g_1$. D'après la Définition 6.20, on doit montrer que $GinGP_R\ g_2\ g_1 \wedge GinGP_R\ g_1\ g_2$, ce qui est vrai d'après H_1 et H_2. □

Lemme 6.44 (*GeqPerm* préserve la transitivité)**.**

$$R \text{ transitive} \Rightarrow (\forall g_1\ g_2\ g_3, GeqPerm_R\ g_1\ g_2 \wedge GeqPerm_R\ g_2\ g_3 \Rightarrow GeqPerm_R\ g_1\ g_3)$$

Démonstration. L'hypothèse $GeqPerm_R\ g_1\ g_2$ nous donne H_1 : $GinGP_R\ g_1\ g_2$ et H_2 : $GinGP_R\ g_2\ g_1$. Et l'hypothèse $GeqPerm_R\ g_2\ g_3$ nous donne H_3 : $GinGP_R\ g_2\ g_3$ et H_4 : $GinGP_R\ g_3\ g_2$. D'après la Définition 6.20, on doit montrer que :

1. $GinGP_R\ g_1\ g_3$: on utilise le Lemme 6.41 avec H_1 et H_3.
2. $GinGP_R\ g_3\ g_1$: on utilise le Lemme 6.41 avec H_4 et H_2.

□

Lemme 6.45 (*GeqPerm* préserve l'équivalence)**.** R *équivalence* $\Rightarrow GeqPerm_R$ *équivalence*

Démonstration. La preuve est une combinaison des trois lemmes précédents. □

Remarque 6.7. *Il est intéressant de noter que les cycles internes des graphes de la Figure 6.11 sont équivalents par GeqPerm (et les racines sont identiques). Pourtant, les graphes "totaux" ne le sont pas. Avec GeqPerm on peut donc avoir des sous-graphes équivalents inclus dans des préfixes équivalents sans que les graphes "totaux"soient équivalents. En particulier, cela veut dire qu'on ne peut pas "réécrire" GeqPerm.*

Dans certains cas, cette propriété peut être gênante. On peut alors tout simplement utiliser GPerm sans le changement de racine. En fonction de l'utilisation visée, GeqPerm peut se révéler utile ou nocive.

Il est important de voir que nous l'avons développée comme un outil possible, mais pas comme la relation "basique" sur les graphes.

Cependant, comme nous l'avons dit, cette relation correspond exactement à celle que nous aurions obtenue avec une représentation des graphes sous forme d'ensemble de nœuds/ensemble d'arcs, et cela nous semble suffisant pour justifier son utilité.

6.2.3 Exemples

Nous allons montrer que les exemples des Figures 6.1 et 6.2 (voir page 125) sont effectivement équivalents par *GeqPerm* et que les graphes de la Figure 6.11 ne le sont pas.

6.2.3.1 Graphes de la Figure 6.1

On définit les deux graphes de la Figure 6.1 de la façon suivante :

$$g_{012} := mk\_Graph\ 0\ [\![mk\_Graph\ 1\ [\![]\!]; mk\_Graph\ 2\ [\![]\!]]\!]$$
$$g_{021} := mk\_Graph\ 0\ [\![mk\_Graph\ 2\ [\![]\!]; mk\_Graph\ 1\ [\![]\!]]\!]$$

On veut montrer que :

Lemme 6.46. $GeqPerm_{eq}\ g_{012}\ g_{021}$

Démonstration. Comme on doit montrer deux fois la même chose, on va commencer par montrer que :

$$H : GPerm\_imp_{eq}\ g_{012}\ g_{021}$$

D'après le Lemme 6.4, il nous suffit de prouver que :

1. $label\ g_{012} = label\ g_{021}$: c'est-à-dire qu'on veut montrer que $0 = 0$, ce qui est vrai par réflexivité.
2. $iperm\_ind_{GPerm\_imp_{eq}}\ (sons\ g_{012})\ (sons\ g_{021})$: en utilisant la Définition 4.17 on va enlever les couples de nœuds identiques deux à deux. L'élément qui correspond à $fct\ (sons\ g_{012})\ (first\ 1)$ dans $sons\ g_{021}$ est $succ\ (first\ 0)$. Utilisons donc la Définition 4.17 pour enlever ces deux éléments (on prouve que les nœuds en question sont équivalents par réflexivité de $GPerm\_imp_{eq}$). Il nous suffit maintenant de prouver que : $iperm\_ind_{GPerm\_imp_{eq}}\ (remEl\ (sons\ g_{012})\ (first\ 1))\ (remEl\ (sons\ g_{021})\ (succ(first\ 0)))$ On enlève de nouveau les deux éléments restants. On prouve qu'ils sont équivalents avec le Lemme 6.6 et on termine la preuve avec la Définition 4.17.

Revenons à la preuve initiale. D'après la Définition 6.20, on doit prouver que :

1. $GinGP_{eq}\ g_{012}\ g_{021}$: ici, les racines sont les mêmes, donc on termine la preuve avec la règle dirG de la Définition 5.5 appliquée à H.

2. $GinGP_{eq}\ g_{021}\ g_{012}$: les racines sont aussi les mêmes, donc d'après la règle dirG de la Définition 5.5 on doit montrer que $GPerm\_imp_{eq}\ g_{021}\ g_{012}$. On termine la preuve avec le Lemme 6.7 appliqué à H.

□

6.2.3.2 Graphes de la Figure 6.2

Il y a plusieurs solutions équivalentes pour définir les deux graphes de la Figure 6.2. Une solution est de partir du constat de la Figure 6.10 et de les définir simultanément (et coinductivement) :

$$g_{01} := mk\_Graph\ 0\ [\![g_{10}]\!] \quad g_{10} := mk\_Graph\ 1\ [\![g_{01}]\!]$$

On veut montrer que :

Lemme 6.47. $GeqPerm_{eq}\ g_{01}\ g_{10}$

Démonstration. On doit montrer que :

1. $GinGP_{eq}\ g_{01}\ g_{10}$: ici, les deux graphes n'ont pas la même racine : g_{01} est strictement inclus dans g_{10}. D'après la règle indirG de la Définition 5.5 il nous suffit de montrer que (g_{10} n'a qu'un fils) :

   $$GinGP_{eq}\ g_{01}\ (fct\ (sons\ g_{10})\ (first\ 0))$$

   C'est-à-dire, d'après la définition de g_{10}, on veut montrer que $GinGP_{eq}\ g_{01}\ g_{01}$. Ce qui est vrai par réflexivité de $GinG^*$.
2. $GinGP_{eq}\ g_{10}\ g_{01}$: la preuve est exactement symétrique à la précédente et nous ne la développons pas ici.

□

6.2.3.3 Graphes de la Figure 6.11

On utilise les définitions de g_{01} et g_{10} pour définir les graphes de la Figure 6.11 :

$$g_{201} := mk\_Graph\ 2\ [\![g_{01}]\!] \quad g_{210} := mk\_Graph\ 2\ [\![g_{10}]\!]$$

On veut cette fois-ci montrer que :

Lemme 6.48. $\neg(GeqPerm_{eq}\ g_{201}\ g_{210})$

Démonstration. Raisonnons par l'absurde. Supposons que $H_1 : GeqPerm_{eq}\ g_{201}\ g_{210}$ et montrons qu'on arrive à une contradiction. H_1 nous donne $H_2 : GinGP_{eq}\ g_{201}\ g_{210}$. Selon la Définition 5.5, on a deux possibilités pour H_2 :

[$H_2 : GPerm\_imp_{eq}\ g_{201}\ g_{210}$**]** Selon le Lemme 6.3 on aurait alors

$$iperm\_ind_{GPerm\_imp_{eq}}\ (sons\ g_{201})\ (sons\ g_{210})$$

Et selon la Définition 4.17 on a à nouveau deux possibilités :

[$lg(sons\ g_{201}) = lg(sons\ g_{210}) = 0$**]** Ce qui est faux d'après les définitions de g_{201} et g_{210}.

[$GPerm\_imp_{eq}\ (fct\ (sons\ g_{201})\ (first\ 0))\ (fct\ (sons\ g_{210})\ (first\ 0))$**]** C'est-à-dire $GPerm\_imp_{eq}\ g_{01}\ g_{10}$ D'après le Lemme 6.3 on aurait alors $label\ g_{01} = label\ g_{10}$, c'est-à-dire $0 = 1$ ce qui est trivialement faux.

[H_2 : $GinGP_{eq}\ g_{201}\ (fct\ (sons\ g_{210})\ (first\ 0))$] C'est-à-dire que H_2 : $GinGP_{eq}\ g_{201}\ g_{10}$. On va raisonner par récurrence structurelle pour montrer la contradiction : on va développer l'hypothèse H_2 jusqu'à retomber dessus (ou à trouver une contradiction). On pourra alors dire qu'on a notre contradiction (par point fixe).

Comme précédemment, on a deux possibilités pour H_2 :

[H_3 : $GPerm\_imp_{eq}\ g_{201}\ g_{10}$] D'après le Lemme 6.3 on aurait $label\ g_{201} = label\ g_{10}$, c'est-à-dire $2 = 1$ ce qui est trivialement faux.

[H_3 : $GinGP_{eq}\ g_{201}\ (fct\ (sons\ g_{10})\ (first\ 0))$] On a à nouveau deux possibilités pour H_3 :

[$GPerm\_imp_{eq}\ g_{201}\ (fct\ (sons\ g_{10})\ (first\ 0))$] C'est-à-dire $GPerm\_imp_{eq}\ g_{201}\ g_{01}$. D'après le Lemme 6.3 on aurait $label\ g_{201} = label\ g_{01}$ c'est-à-dire $2 = 0$, ce qui est trivialement faux.

[$GinGP_{eq}\ g_{201}\ (fct\ (sons\ g_{01})\ (first\ 0))$] C'est-à-dire $GinGP_{eq}\ g_{201}\ g_{10}$, ce qui était notre hypothèse de point fixe et qui termine notre preuve.

□

6.3 Des Graphes non connexes, non enracinés

Les graphes que nous avons représentés jusqu'à maintenant sont connexes et enracinés. C'est-à-dire que comme on l'a expliqué, on peut atteindre tous les nœuds depuis la racine.

Par exemple, on ne pourrait pas représenter le graphe de la Figure 6.12 avec *Graph*.

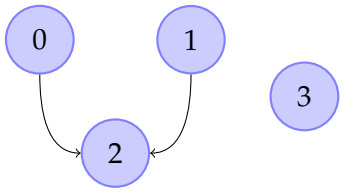


Figure 6.12 — Exemple de graphe non connexe et non enraciné

Il serait donc intéressant d'alléger un peu la rigidité de notre représentation afin de pouvoir représenter ce type de graphes. Nous allons proposer deux solutions possibles. Les résultats présentés ici sont des pistes, encore à explorer largement.

6.3.1 Nœuds fictifs

Dans cette première solution, on propose d'introduire des nœuds fictifs dans notre représentation afin de créer un "lien" (fictif) entre les différentes parties non connexes du graphe. Par exemple, pour le graphe de la Figure 6.12, on pourrait ajouter un nœud fictif à la racine qui aurait pour fils les trois parties non connexes du graphe, comme présenté dans la Figure 6.13.

Pour représenter ces nœuds fictifs, on peut utiliser le type *option*. Un nœud avec pour étiquette *Some t* sera considéré comme un nœud "réel" avec pour étiquette t et un nœud avec pour étiquette *None* sera considéré comme un nœud fictif. Forts de cette constatation, on peut tout simplement réutiliser notre représentation des graphes initiale, *Graph*, en l'instanciant avec *option T*.

Définition 6.21 (*AllGraph*). $AllGraph\ T := Graph\ (option\ T)$

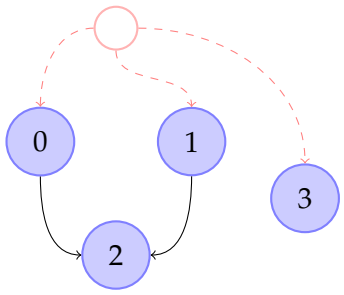


Figure 6.13 — Exemple de graphe non connexe et non enraciné, représenté avec un nœud fictif

L'avantage ici c'est que toutes les définitions que nous avons données sur *Graph* restent utilisables ici. On peut également redéfinir une relation sur *AllGraph* en combinant *Geq* et *RelOp*.

Définition 6.22 (*AGeq*)**.** $AGeq_R := Geq_{RelOp\ R}$

On montre immédiatement, en combinant le fait que *RelOp* et *Geq* préservent l'équivalence (resp. la réflexivité , la symétrie, la transitivité), que *AGeq* préserve l'équivalence (resp. la réflexivité, la symétrie, la transitivité).

On peut également définir une fonction qui permet de transformer un élément de *Graph* en élément de *AllGraph*. On transforme tous les nœuds en nœuds "réels".

Définition 6.23 (*G2AG*, vu coinductivement)**.**

$$\begin{array}{lcl} G2AG & : & Graph\ T \to AllGraph\ T \\ G2AG\ g & := & mk\_Graph\ (Some(label\ g))\ (imap\ G2AG\ (sons\ g)) \end{array}$$

Remarque 6.8. *Cette définition est acceptée par la condition de garde pour les mêmes raisons que applyF2G.*

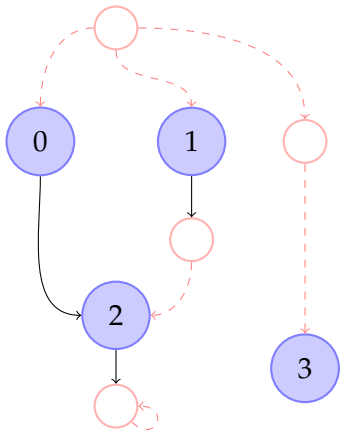


Figure 6.14 — Autre représentation pour le graphe de la Figure 6.13

Cependant, *AllGraph* pose quelques problèmes. On peut en effet ajouter autant de nœuds fictifs que l'on veut sans que cela change la représentation. Ainsi, un même graphe peut être représenté de plusieurs façons. Par exemple, la Figure 6.14 propose une alternative à la représentation de la Figure 6.13, pour le même graphe.

Bien sûr, déjà avec *Graph*, la représentation d'un graphe pouvait ne pas être unique (comme on le voit par exemple dans la Figure 5.1), mais ici le problème est plus compliqué. On peut modifier le nombre de fils d'un nœud (comme pour le nœud 2 par exemple), ajouter des nœuds au milieu d'une chaîne, introduire des cycles de nœuds fictifs, etc. Bref, il est probablement assez compliqué de trouver une "bonne" relation d'équivalence sur *AllGraph*, qui soit suffisamment permissive.

Cependant, en réalité, on a l'intuition que toutes les parties d'un graphe peuvent être réunies au plus haut niveau (c'est-à-dire directement sous la racine), comme proposé dans la Figure 6.13. Il semble que tous les autres nœuds fictifs sont inutiles.

Une solution pour définir une relation sur *AllGraph* serait peut être de revenir à cette représentation "normalisée" et de comparer le résultat. La tâche ne semble toutefois pas aisée.

6.3.2 Forêts de Graphes

Néanmoins, si on considère que tous les nœuds, hormis la racine n'ont pas de raison d'être fictifs, on peut tout simplement les en empêcher. Dans ce cas, un graphe serait une racine fictive avec des fils tous "réels". Mais ici, la racine ne sert plus à rien, la seule chose dont nous avons en réalité besoin étant la liste des fils. Ainsi, l'autre solution est de représenter ces graphes non connexes par des "forêts" de graphes. On définit ces forêts ainsi :

Définition 6.24 (*ForestGraph*)**.** $ForestGraph\ T := list\ (Graph\ T)$

On peut bien sûr définir une relation sur *ForestGraph* en combinant une relation sur *list* avec une relation sur *Graph*. Afin d'être le plus souple possible, on peut par exemple autoriser les permutations dans la forêt ainsi que dans chacun des graphes.

Définition 6.25 (*FGeq*)**.** $FGeq_R := permut_1\ (GeqPerm_R)$

On montre immédiatement, en combinant le fait que $permut_1$ et *GeqPerm* préservent l'équivalence (resp. la réflexivité , la symétrie, la transitivité), que *FGeq* préserve l'équivalence (resp. la réflexivité , la symétrie, la transitivité).

La principale différence, en terme d'expressivité, entre *AllGraph* et *ForestGraph* est que *ForestGraph* ne permet pas de représenter des graphes avec un nombre infini de parties. En effet, comme la forêt dans *ForestGraph* est représentée par une liste (finie), il ne peut y avoir qu'un nombre fini de parties non connexes dans le graphe. Alors que dans *AllGraph*, comme on peut enchaîner les nœuds fictifs, il peut y en avoir un nombre infini (tout en gardant bien sûr un nombre fini de fils pour chaque nœud). La Figure 6.15 présente un exemple d'arbre avec un nombre infini de parties non connexes.

Une solution pour étendre l'expressivité de *ForestGraph* à des graphes avec un nombre infini de parties non connexes serait de remplacer la liste par une liste potentiellement infinie (coliste). Cependant, nous nous intéressons surtout à des graphes qui ont un nombre fini de nœuds (et donc un nombre de parties non connexes fini également). Nous considérons donc cette solution satisfaisante.

On peut aisément transformer un élément l de *ForestGraph* en élément de *AllGraph* : $mk\_Graph\ None\ (list2ilist\ (map\ G2AG\ l))$. L'autre conversion est bien entendue plus compliquée, pour les raisons expliquées plus haut. Il serait cependant intéressant de montrer que la forme normalisée de *AllGraph* est équivalente à *ForestGraph* (mais pour cela il faudrait bien entendu qu'on soit capable de normaliser un élément de *AllGraph*).

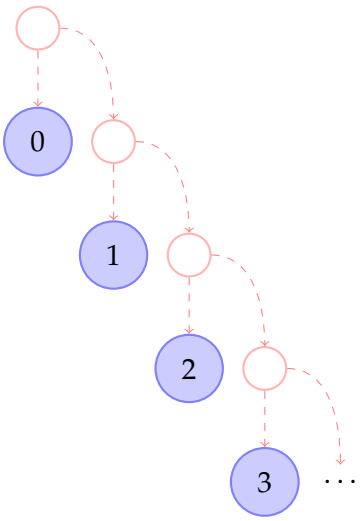


Figure 6.15 — Graphe avec un nombre infini de parties non connexes

Cela reste pour le moment une perspective de notre travail.

Conclusion et perspectives

Bilan

Dans cette thèse nous avons proposé une représentation complète des graphes à l'aide des types coinductifs, dans le but ensuite de l'utiliser pour représenter des métamodèles.

Pour les besoins de cette représentation, nous avons été amenés à développer une bibliothèque complète permettant de définir un équivalent fonctionnel aux listes, les *ilist* (la version conteneur des listes). Cela a été présenté dans la Partie II. Nous avons montré que les *ilist* étaient effectivement équivalentes aux listes et nous avons muni cette représentation de nombreux outils. En particulier, nous avons développé des outils permettant de manipuler les *ilist* à la manière des listes.

Nous avons également muni cette définition de relations d'équivalence, et en particulier de relations qui permettent de représenter les permutations (en réalité, nous avons spécifié leur existence, ou les avons représentées avec des certificats ou à l'aide de permutations sur les indices). Nous avons également montré que les différentes approches des permutations sur *ilist* (qui ne requièrent pas la décidabilité sur la relation de base) étaient équivalentes entre elles et avec une autre approche proposée par Contejean sur les listes. Il nous a également semblé opportun de transposer nos représentations des permutations aux listes. Il nous a en effet paru que la littérature à ce sujet dans Coq était assez peu fournie.

Au total, nous avons présenté plus de 60 définitions dans cette partie, justifiées par plus de 110 résultats (lemmes, théorèmes, propriétés, etc.). Dans le code Coq cela correspond à un total de 8350 lignes réparties en 10 fichiers, dont plus de 2000 lignes pour les définitions et plus de 5100 lignes pour les preuves.

La représentation coinductive des graphes que nous proposons dans la Partie III se sert de ces *ilist*. L'idée était de dualiser la représentation inductive classique des arbres, mais pour des problèmes de garde, nous avons dû remplacer les listes inductives normalement utilisées dans la représentation des arbres par un équivalent non inductif. C'est pour cela que nous avons eu besoin des *ilist*. La définition que nous proposons permet de représenter des graphes connexes et enracinés. Elle est elle également munie de nombreux outils. Ainsi nous avons défini plusieurs notions d'inclusion (d'un graphe dans un autre), la notion de cycle et celle de finitude.

Nous avons également défini la bisimilarité canonique sur les graphes. Mais nous avons constaté que cette notion était trop stricte pour nos besoins, et en particulier plus stricte que les notions classiques d'équivalence sur les graphes (quand ceux-ci sont représentés comme des ensembles de nœuds/ensemble d'arcs). En effet, la relation canonique induit implicitement un ordre dans les nœuds : un ordre horizontal puisque les fils doivent être

dans le même ordre, mais aussi un ordre vertical puisque la racine doit être la même (dans le cas d'un cycle, la "racine" peut ne pas être unique).

Nous avons donc tout d'abord cherché à régler le problème de l'ordre horizontal. Nous avons simplement redéfini la relation canonique, en utilisant cette fois-ci non plus la relation canonique sur *ilist* mais une des relations (inductive) de permutations sur *ilist*. Nous avons de nouveau été confrontés à des problèmes de mélange entre induction et coinduction, au niveau de la définition de propriétés cette fois. Nous avons proposé plusieurs solutions pour résoudre ces problèmes. Nous avons tout d'abord proposé des versions imprédicatives de cette nouvelle relation. Puis nous avons proposé une solution basée sur des observations finies. Enfin, nous avons proposé une solution utilisant une autre des relations de permutations sur *ilist*, non inductive cette fois. Et nous avons montré que ces différentes propositions étaient équivalentes entre elles (tout cela est résumé dans la Figure 6.9). Notons cependant que pour démontrer que la version basée sur les observations est équivalente à la version imprédicative, nous avons eu besoin de nous servir d'un axiome : le principe des tiroirs infini. Nous avons cependant bien insisté sur ce point et avons justifié son utilisation par des principes de la logique classique. Nous avons ensuite proposé une relation qui permet de résoudre le problème de l'ordre horizontal dans les nœuds.

Enfin, nous avons proposé plusieurs pistes afin de représenter des graphes non connexes et non enracinés. En effet, en gardant en tête l'objectif final qui est de pouvoir utiliser cette représentation des graphes afin de représenter des métamodèles, il nous semblait important de pouvoir proposer cette fonctionnalité. C'est également dans cette idée-là que nous avons proposé d'étendre la représentation des *ilist* à une représentation des multiplicités dans la Section 3.3. Ces derniers travaux ne sont cependant pas encore tout-à-fait mûrs.

Au total, pour la représentation des graphes nous avons présenté près de 40 définitions, justifiées par près de 90 résultats (et un axiome). Dans le code Coq cela correspond à un total de près de 4000 lignes réparties en 6 fichiers, dont plus de 870 lignes pour les définitions et plus de 2400 lignes pour les preuves.

Les représentations que nous avons proposées (pour *ilist* et pour les graphes) nous semblent aujourd'hui complètes et prêtes à être utilisées dans d'autres applications, en particulier dans le cadre du projet dans lequel s'inscrit cette thèse. Le développement complet de ces librairies représente un total de plus de 12500 lignes de code Coq, réparties en 17 fichiers, dont plus de 3000 lignes pour les définitions et plus de 7700 lignes pour les preuves. Dans cette thèse, nous avons présenté au total 100 définitions, justifiées par plus de 200 résultats.

La majorité des résultats présentés ici ont été publiés. Dans [69], présenté à l'atelier *Graph Computation Models* en octobre 2010 (GCM'10), nous avons exposé les résultats principaux des Chapitres 3 et 5 de cette thèse. Une version étendue [70] (24 pages) a ensuite été publiée dans *Electronic Communications of the EASST*. Nous avons présenté les résultats principaux des Chapitres 4 et 6 de cette thèse à *Coalgebraic Methods in Computer Science* en avril 2012 (CMCS'12) dans [72] (20 pages). Nous travaillons actuellement à la version finale qui sera publiée dans *Lecture Notes in Computer Science* (LNCS).

Perspectives

De nombreuses pistes s'ouvrent à nous pour compléter ce travail.

Étendre la représentation

Même si la représentation des graphes que nous avons proposée est assez mûre, il est encore possible de la compléter et l'étendre un peu, en particulier sur des points que nous avons déjà mentionnés.

Ainsi, comme nous l'avons dit, nous travaillons sur un nouveau critère de finitude qui utilise les arbres avec pointeur de retour [7, 50] (en ne considérant que les cycles et pas le partage). Ce travail est encore à l'état d'ébauche. Nous le réalisons en collaboration avec l'équipe *Logic and Semantic Group* de l'*Institute of Cybernetics* de Tallinn, Estonie. Ce travail est réalisé en parallèle sur Coq et Agda, ce qui permet également de comparer les prouveurs.

Il serait de plus intéressant d'approfondir la notion de forêt de graphes ainsi que celle de multiplicités. En effet, ces deux notions semblent assez prometteuse, notamment dans l'optique de la représentation des métamodèles.

Automates d'états finis

Une première utilisation directe de nos graphes serait de les utiliser pour représenter les automates d'états finis et de raisonner dessus. Une fois nos graphes instanciés pour la représentation des automates, on voudrait leur appliquer des transformations et certifier ces transformations. En effet, la transformation certifiée est, comme on l'a déjà dit, l'objectif principal du projet dans lequel s'inscrit cette thèse. On pourrait ainsi avoir un premier exemple de transformations certifiées utilisant nos graphes. Les transformations que nous visons seraient par exemple la minimisation ou la déterminisation des automates. Pour certifier la minimisation, on pourrait démontrer que l'application de la transformation à un automate d'entrée donne bien un automate qui reconnaît le même langage (un mot est reconnu par l'automate d'arrivée si et seulement si il est reconnu par l'automate d'entrée), que l'automate est bien minimal, etc.

Au printemps 2011, nous avons encadré un étudiant sur le sujet. Son objectif était d'instancier les graphes pour les automates et d'implanter et de vérifier l'algorithme de minimisation de Hopcroft [48] en se basant sur [73]. Malheureusement, le travail n'a pas totalement abouti, le temps imparti n'ayant pas été suffisant pour prendre en main Coq et réaliser le travail. De plus, l'algorithme tel que nous avons tenté de l'implémenter est fortement impératif, ce qui se marie très mal avec Coq et notre représentation des graphes.

Nous voulons donc maintenant reprendre et terminer ce travail. En particulier, on pourrait s'inspirer de [4].

Représentation des métamodèles

En plus des travaux dont nous avons parlé précédemment sur la notion de forêt de graphes et les multiplicités, qui sont encore à améliorer, d'autres problèmes se posent pour la représentation des métamodèles, en particulier la représentation de l'héritage. Nous avons testé une solution qui utilise le polymorphisme dans l'idée de [63, 20, 47] en utilisant des types coinductifs à la place des enregistrements. Cette solution est encore à approfondir largement

mais elle semble très prometteuse. Nous sommes en train de représenter le métamodèle de Ecore [36], très complet, en particulier parce que les paramètres des classes introduisent des cycles.

Dans une autre direction, on pourrait également s'inspirer des travaux sur la théorie des types pour les métamodèles de Poernomo [74]. Le travail de Boulmé dans FOCAL [19] qui a été réalisé en Coq pourrait également nous aider à résoudre ce problème d'héritage.

Généraliser les techniques

Il serait certainement intéressant de pouvoir généraliser les techniques mises en œuvre pour contourner la condition de garde. En particulier, il serait intéressant de pouvoir mélanger des types coinductifs avec n'importe quel type inductif, et pas seulement des listes. Cependant, nous n'avons pas du tout travaillé dans cette direction. Il semblerait que la théorie des catégories pourrait être d'une grande aide ici. On pourrait en particulier s'inspirer du travail de Niqui [66] présenté à la Section 2.2.2.

On pourrait également vouloir appliquer la compétence que nous avons acquise en matière de mélange entre induction et coinduction à d'autres exemples que les graphes. Ainsi, le travail théorique de Berger [12] (avec une implantation dans Haskell mais sans preuve de correction) sur la calculabilité des nombres réels utilise un mélange de types inductifs et coinductifs. Il pourrait donc être intéressant de pouvoir l'implémenter dans Coq et nous sommes convaincus que notre expertise en la matière nous simplifiera grandement la tâche.

Conteneurs

On pourrait également étendre les liens avec les conteneurs dans plusieurs directions.

Tout d'abord, la notion catégorique de conteneur vient naturellement avec une notion de morphisme. Les transformations sur *ilist* pourraient donc être vues comme des morphismes induits par des morphismes de conteneurs. Il serait certainement intéressant de regarder un peu plus de ce côté-là, en essayant de tirer parti des résultats généraux connus sur les conteneurs pour notre cas particulier.

On pourrait également, comme mentionné dans l'introduction du Chapitre 4, étudier la notion de types quotients [1]. En effet, il est bien connu qu'on ne peut pas utiliser directement les conteneurs pour représenter des ensembles ni des multi-ensembles. En revanche, il semblerait que ces derniers pourraient être représentés par des types quotients, qui sont encore largement inexplorés. Cela nous permettrait peut-être de généraliser un peu notre représentation des listes et de profiter également des résultats connus sur les conteneurs.

Enfin, il est bien connu que le plus grand point fixe d'un conteneur est lui-même un conteneur. Il pourrait alors être intéressant de voir si les graphes eux-mêmes ne pourraient pas être représentés comme des conteneurs. Cela nous permettrait probablement de nouveau de profiter des résultats sur les conteneurs pour notre représentation des graphes.

Quatrième partie

Annexes

A Preuves

Dans ce chapitre nous allons retranscrire toutes les preuves mentionnées dans les parties précédentes mais non détaillées. Ces preuves sont données dans l'ordre où elles ont été introduites. Le plan des sections ci-dessous suit le plan général de la thèse, de façon simplifiée.

A.1 Un outil pour la suite : un équivalent fonctionnel aux listes - Preuves

A.1.1 Définition de *ilist* et propriétés des base - Preuves

A.1.1.1 Preuve du Lemme 3.7

Rappel de l'énoncé. $\forall n\ (i : Fin\ n)$, *code h* $= i$ *où h : decode i* $< n$ *est obtenu grâce au Lemme 3.5 instancié avec n et i*

Démonstration par induction sur i. L'induction sur i élimine automatiquement le cas $n = 0$.
[Cas $i = first\ n$ **– induction sur** i**]**

On a $h : \underbrace{decode(first\ n)}_{0} < n + 1$, d'après la Définition 3.3.
D'où,

$$\begin{aligned} code\ h &= first\ n \quad \text{(d'après la Définition 3.4)} \\ &= i \end{aligned}$$

[Cas $i = succ\ i'$ **– induction sur** i**]**
L'hypothèse d'induction est : $IH : \forall i' : Fin\ n, code\ h' = i'$, où h' a pour type $decode\ i' < n$ et est déduit du Lemme 3.5.
On a $h : \underbrace{decode(succ\ i')}_{decode\ i' + 1} < n + 1$, d'après la Définition 3.3.
D'où,

$$\begin{aligned} code\ h &= succ\ (code\ h_1) && \text{(d'après la Définition 3.4, où } h_1 \text{ a pour type } decode\ i' < n \text{ (déduit de h))} \\ &= succ\ (code\ h_2) && \text{(d'après le Lemme 3.6, où } h_2 \text{ est obtenu grâce au Lemme 3.5)} \\ &= succ\ i' && \text{(d'après } IH\text{)} \\ &= i \end{aligned}$$

□

A.1.1.2 Preuve du Lemme 3.8

Rappel de l'énoncé. $\forall n\ m\ (h : m < n),\ decode\ (code\ h) = m$

Démonstration par induction sur n.

[Cas 0 **– induction sur n]**
L'hypothèse h aurait pour type $m < 0$ qui est vide.

[Cas $n+1$ **– induction sur n]**
L'hypothèse d'induction IH est $\forall m\ (h : m < n),\ decode\ (code\ h) = m$. On raisonne maintenant par analyse de cas sur m :

[Cas 0**]** On a $h : 0 < n+1$.

$$\begin{array}{rcll} decode\ (code\ h) & = & decode\ (first\ n) & \text{(d'après Définition 3.4)} \\ & = & 0 & \text{(d'après Définition 3.3)} \end{array}$$

[Cas $m+1$**]** On a $h : m+1 < n+1$.

$$\begin{array}{rcll} decode\ (code\ h) & = & decode\ (succ\ (code\ h')) & \text{(d'après Définition 3.4, avec } h' : m < n \text{ déduit de } h\text{)} \\ & = & decode\ (code\ h') + 1 & \text{(d'après Définition 3.3)} \\ & = & m+1 & \text{(d'après } IH\text{)} \end{array}$$

□

A.1.1.3 Preuve du Lemme 3.12

Rappel de l'énoncé.

$$\begin{array}{c} \forall (f_1 : T \to U)(g_1 : U \to T)(f_2 : U \to V)(g_2 : V \to U), \\ bij\ f_1\ g_1 \wedge bij\ f_2\ g_2 \Rightarrow bij\ (f_2 \circ f_1)(g_1 \circ g_2) \end{array}$$

Démonstration. D'après la Définition 3.6, les hypothèses $bij\ f_1\ g_1$ et $bij\ f_2\ g_2$ nous donnent

$$\begin{array}{ll} H_1 : \forall t, g_1(f_1\ t) = t & H_2 : \forall u, f_1(g_1\ u) = u \\ H_3 : \forall u, g_2(f_2\ u) = u & H_4 : \forall v, f_2(g_2\ v) = v \end{array}$$

D'après la Définition 3.6, on doit montrer que :

1. $\forall t, g_1(g_2(f_2(f_1\ t))) = t$: on a :

$$\begin{array}{rcll} g_1(g_2(f_2(f_1\ t))) & = & g_1(f_1\ t) & \text{(d'après } H_3\text{)} \\ & = & t & \text{(d'après } H_1\text{)} \end{array}$$

2. $\forall v, f_2(f_1(g_1(g_2\ v))) = v$: on a :

$$\begin{array}{rcll} f_2(f_1(g_1(g_2\ v)) & = & f_2(g_2\ v) & \text{(d'après } H_2\text{)} \\ & = & v & \text{(d'après } H_4\text{)} \end{array}$$

□

A.1.1.4 Preuve du Lemme 3.13

Rappel de l'énoncé.

$$\forall (f : T \to U)\ (g : U \to T)\ t_1\ t_2, bij\ f\ g \wedge f\ t_1 = f\ t_2 \Rightarrow t_1 = t_2$$

Démonstration. L'hypothèse $bij\ f\ g$ nous donne $H_1 : \forall t, g(f\ t) = t$. Soit $H_2 : f\ t_1 = f\ t_2$. On veut montrer que $t_1 = t_2$. D'après H_1 cela revient à montrer que $g(f\ t_1) = g(f\ t_2)$. C'est-à-dire d'après $H_2 : g(f\ t_1) = g(f\ t_1)$, ce qui est vrai par réflexivité. □

A.1.1.5 Preuve du Lemme 3.15

Rappel de l'énoncé.

$$\forall f_1\ f_2\ (H_1 : bij\ f_1\ f_2)(H_2 : bij\ f_2\ f_1), \forall i, transfoFun\ H_2\ (transfoFun\ H_1\ i) = i$$

Démonstration. H_1 nous donne $H_3 : \forall i, f_2(f_1\ i) = i$. Raisonnons par analyse de cas sur $decode(f_1(succ\ i))$:

[Cas $H_4 : decode(f_1(succ\ i)) = 0$**]** D'après la Définition 3.8 et H_4, on a $transfoFun\ H_1\ i = getcons\ (f_1(first\ n_1))\ H_4'$, avec $H_4' : 0 < decode(f_1(first\ n_1))$. On doit maintenant analyser $decode(f_2(succ(getcons\ (f_1(first\ n_1))\ H_4')))$. On a :

$$\begin{array}{ll} f_2(succ(getcons\ (f_1(first\ n_1))\ H_4')) & = f_2(f_1(first\ n_1)) \quad \text{(d'après Propriété 3.1.1)} \\ & = first\ n_1 \quad \text{(d'après } H_3\text{)} \end{array}$$

Donc $decode(f_2(succ(getcons\ (f_1(first\ n_1))\ H_4'))) = 0$. Donc d'après la Définition 3.8, $transfoFun\ H_2\ (getcons\ (f_1(first\ n_1))\ H_4') = getcons\ (f_2\ (first\ n_2))\ H_5$, avec $H_5 : decode\ (f_2\ (first\ n_2)) > 0$. On veut montrer que $getcons\ (f_2\ (first\ n_2))\ H_5 = i$. Ce qui revient à montrer que : $succ(getcons\ (f_2\ (first\ n_2))\ H_5) = succ\ i$. Ou encore, en utilisant la Propriété 3.1.1 à gauche et H_3 à droite : $f_2(first\ n_2) = f_2(f_1(succ\ i))$. Il nous suffit donc de montrer que $first\ n_2 = f_1(succ\ i)$. D'après le Lemme 3.9 il nous suffit de montrer que : $first\ n_2 =_{Fin} f_1(succ\ i)$, ce qui est vrai d'après H_4.

[Cas $H_4 : 0 < decode(f_1(succ\ i))$**]** D'après la Définition 3.8 et H_4, on a

$$transfoFun\ H_1\ i = getcons\ (f_1(succ\ i))\ H_4$$

On prouve aisément que $H_5 : 0 < decode(f_2(succ(getcons\ (f_1(succ\ i))\ H_4)))$. On a donc :

$$\begin{array}{l} transfoFun\ H_2\ (getcons\ (f_1(succ\ i))\ H_4) \\ =\ \ getcons\ (f_2(succ(getcons\ (f_1(succ\ i))\ H_4)))\ H_5 \end{array}$$

On veut montrer que : $getcons\ (f_2(succ(getcons\ (f_1(succ\ i))\ H_4)))\ H_5 = i$. On a :

$$\begin{array}{lll} & getcons\ (f_2(succ(getcons\ (f_1(succ\ i))\ H_4)))\ H_5 & \\ = & getcons\ (f_2(f_1(succ\ i)))\ H_5' & \text{(d'après Propriété 3.1.1 et où } H_5' : 0 < decode(f_2(f_1(succ\ i)))\text{)} \\ = & getcons\ (succ\ i)\ H_5'' & \text{(d'après } H_3 \text{ et où } H_5'' : 0 < decode(succ\ i)\text{)} \\ = & i & \text{(d'après Propriété 3.1.2)} \end{array}$$

□

A.1.1.6 Preuve du Lemme 3.16

Rappel de l'énoncé. $\forall f_1\ f_2\ (H : bij\ f_1\ f_2), bij\ (transfoFun\ H)\ (transfoFun\ H')$ *avec* $H' : bij\ f_2\ f_1$ *déduit de H et du Lemme 3.11.*

Démonstration. D'après la Définition 3.6, on doit montrer que :

1. $\forall i, transfoFun\ H'\ (transfoFun\ H\ i) = i$: on utilise simplement le Lemme 3.15.
2. $\forall i, transfoFun\ H\ (transfoFun\ H'\ i) = i$: on utilise encore le Lemme 3.15.

On utilise bien ainsi la symétrie du problème. □

A.1.1.7 Preuve du Lemme 3.17

Rappel de l'énoncé. $\forall R\ (ln_1\ ln_2 : ilistn\ T\ 0),\ ilist\_rel_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle$

Démonstration. On a

$$\begin{array}{rcl} ilist\_rel_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle & \Leftrightarrow & \exists h : lg\ \langle 0, ln_1\rangle = lg\ \langle 0, ln_2\rangle, \\ & & \forall i : Fin\ (lg\ \langle 0, ln_1\rangle), R\ (fct\ \langle 0, ln_1\rangle\ i)\ (fct\ \langle 0, ln_2\rangle\ (conv_h\ i)) \\ & \Leftrightarrow & \exists h : 0 = 0,\ \forall i : Fin\ 0, R\ (ln_1\ i)\ (ln_2\ (conv_h\ i)) \end{array}$$

Obtenir h est trivial et $\forall i : Fin\ 0, R\ (ln_1\ i)\ (ln_2\ (conv_h\ i))$ est toujours vrai puisqu'il n'existe aucun élément de type $Fin\ 0$. □

A.1.1.8 Preuve du Lemme 3.22

Rappel de l'énoncé. $\forall R_1\ R_2\ l_1\ l_2, R_1 \subseteq R_2\ \wedge\ ilist\_rel_{R_1}\ l_1\ l_2 \Rightarrow ilist\_rel_{R_2}\ l_1\ l_2$

Démonstration. La preuve est assez triviale. Soient H_1 et H_2 les hypothèses :

$$H_1 : R_1 \subseteq R_2 \quad et \quad H_2 : ilist\_rel_{R_1}\ l_1\ l_2$$

Grâce à H_2 (et à la Définition 3.11) on obtient deux nouvelles hypothèses :

$$H_3 : lg\ l_1 = lg\ l_2 \quad et \quad H_4 : \forall i, R_1\ (fct\ l_1\ i)\ (fct\ l_2\ (conv_{H_3}\ i))$$

On applique la Définition 3.11 (avec H_3) à notre but et on doit maintenant prouver que :

$$\forall i, R_2\ (fct\ l_1\ i)\ (fct\ l_2\ (conv_{H_3}\ i))$$

Pour cela on utilise H_1 et H_4. □

A.1.1.9 Preuve du Lemme 3.25

Rappel de l'énoncé. $\forall l_1\ l_2, ilist\_rel_{eq}\ l_1\ l_2 \Leftrightarrow ilist2list\ l_1 = ilist2list\ l_2$

Démonstration. Soient n_1, n_2, ln_1 et ln_2 tels que $l_1 = \langle n_1, ln_1\rangle$ et $l_2 = \langle n_2, ln_2\rangle$.

[Direction $\Rightarrow$] L'hypothèse $ilist\_rel_{eq}\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle$ nous donne :

$$H_1 : n_1 = n_2 \quad H_2 : \forall i, ln_1\ i = ln_2\ (conv_{H_1}\ i)$$

On réécrit H_1 partout et on peut donc se passer de n_2. De plus on montre simplement que H_2 peut maintenant s'écrire : $H_2 : \forall i, ln_1\ i = ln_2\ i$.

On veut prouver que $ilist2list\langle n_1, ln_1\rangle = ilist2list\langle n_1, ln_2\rangle$. C'est-à-dire que

$$map\ ln_1\ (makeListFin\ n_1) = map\ ln_2\ (makeListFin\ n_1)$$

On finit la preuve avec l'extensionnalité de *map* et H_2.

[Direction $\Leftarrow$] L'hypothèse $ilist2list\langle n_1, ln_1\rangle = ilist2list\langle n_2, ln_2\rangle$ se simplifie en H_1 : $map\ ln_1\ (makeListFin\ n_1) = map\ ln_2\ (makeListFin\ n_2)$. On en déduit facilement que $H_2 : n_1 = n_2$. On réécrit donc H_2 partout et on peut se passer de n_2. On veut prouver que $ilist\_rel_{eq}\ \langle n_1, ln_1\rangle\langle n_1, ln_2\rangle$. On prouve par réflexivité que

$$lg\ \langle n_1, ln_1\rangle = lg\langle n_1, ln_2\rangle$$

et il nous reste à prouver que $\forall i, ln_1\ i = ln_2\ i$.

On utilise le corrolaire suivant de l'extensionnalité de *map* :

$$\forall (f_1\ f_2 : T \to U)\ l, map\ f_1\ l = map\ f_2\ l \Rightarrow \forall t, t \in l \Rightarrow f_1\ t = f_2\ t$$

On finit la preuve avec H_1 et le Lemme 3.3.

□

A.1.1.10 Preuve du Lemme 3.28

Rappel de l'énoncé. $\forall l\ i\ t, nth\ (decode\ i)\ l\ t = fct\ (list2ilist\ l)\ i$

Démonstration. On raisonne par analyse de cas sur l :

[Cas $l = []$] Alors i a pour type $Fin\ (lg\ (list2ilist\ []))$ c'est-à-dire $Fin\ 0$, qui est vide.

[Cas $l = t'::q$] On veut montrer que : $nth\ (decode\ i)\ (t'::q)\ t = fct\ (list2ilist\ (t'::q))\ i$

$$\begin{array}{rlll} \text{On a :} & fct\ (list2ilist\ (t'::q))\ i & = & list2FinT\ (t'::q)\ i \quad \text{(d'après Définition 3.15)} \\ & & = & nth\ (decode\ i)\ (t'::q)\ t' \quad \text{(d'après Définition 3.14)} \end{array}$$

On doit donc montrer que : $nth\ (decode\ i)\ (t'::q)\ t = nth\ (decode\ i)\ (t'::q)\ t'$.

Or, le résultat suivant est bien connu sur *nth* :

Lemme A.1. $\forall l\ n\ d\ d', n < length\ l \Rightarrow nth\ n\ l\ d = nth\ n\ l\ d'$

On l'utilise donc et on prouve que : $decode\ i < length\ (t'::q)$ directement avec le Lemme 3.5.

□

A.1.1.11 Preuve du Lemme 3.29

Rappel de l'énoncé. $\forall l\ n\ t\ (h : n < lg\ l), nth\ n\ (ilist2list\ l)\ t = fct\ l\ (code\ h)$

Démonstration. On a :

$nth\ n\ (ilist2list\ l)\ t$		
$=$	$nth\ n\ (map\ (fct\ l)\ (makeListFin\ (lg\ l)))\ (fct\ l\ (code\ h))$	(d'après Lemme A.1, on donne $fct\ l\ (code\ h)$ comme élément par défaut à *nth*)
$=$	$fct\ l\ (nth\ n\ (makeListFin\ (lg\ l))\ (code\ h))$	(résultat bien connu sur *nth*)
$=$	$fct\ l\ (code\ h)$	(d'après Lemme 3.24)

□

A.1.1.12 Preuve du Lemme 3.33

Rappel de l'énoncé. $\forall l_1\ l_2, ilist\_rel_{eq}\ (iappend\ l_1\ l_2)\ (list2ilist\ ((ilist2list\ l_1)@ilist2list\ l_2))$

Démonstration. On prouve aisément

$$H_1 : lg\ (iappend\ l_1\ l_2) = lg\ (list2ilist\ ((ilist2list\ l_1)@ilist2list\ l_2))$$

On peut donc appliquer la définition de *ilist_rel* et on doit prouver que :

$$\forall i, fct\ (iappend\ l_1\ l_2)\ i = fct\ (list2ilist\ ((ilist2list\ l_1)@ilist2list\ l_2))\ (conv_{H_1}\ i)$$

On applique le Lemme 3.28 et on simplifie. On doit maintenant prouver que :

$$fct\ (iappend\ l_1\ l_2)\ i = nth\ (decode\ i)\ ((ilist2list\ l_1)@ilist2list\ l_2)\ (fct\ (iappend\ l_1\ l_2)\ i)$$

On peut ici remplacer $conv_{H_1}\ i$ par i grâce à la Propriété 3.2. Pour terminer, on va vouloir appliquer les Propriétés 3.6.2 et 3.6.3 et aussi les deux résultats suivants bien connus sur *nth* :

Lemme A.2. $\forall l\ l'\ d\ n, n < length\ l \Rightarrow nth\ n\ (l@l')\ d = nth\ n\ l\ d$

Lemme A.3. $\forall l\ l'\ d\ n, length\ l \leq n \Rightarrow nth\ n\ (l@l')\ d = nth\ (n - length\ l)\ l'\ d$

Pour utiliser ces différents résultats, nous devons maintenant comparer les valeurs de $lg\ l_1$ et de $decode\ i$:

[$H_2 : lg\ l_1 \leq decode\ i$] On prouve $H_2' : lg\ l_1 \leq decode\ (conv_{H_3} i)$ à l'aide de la Propriété 3.2 avec H_3 déduite de la Propriété 3.6.1. On a :

$fct\ (iappend\ l_1\ l_2)\ i$	$=$ $fct\ l_2\ (rightFin\ (conv_{H_3}\ i)\ H_2')$	(d'après Propriété 3.6.3)
	$=$ $fct\ l_2\ (code\ H_4)$	(d'après Définition 3.18 avec $H_4 : decode\ (conv_{H_3} i) - lg\ l_1 < lg\ l_2$)

et (en simplifiant) :

$nth\ (decode\ i)\ ((ilist2list\ l_1)@ilist2list\ l_2)\ (fct\ (iappend\ l_1\ l_2)\ i)$	
$= nth\ (decode\ i - lg\ l_1)\ (ilist2list\ l_2)\ (fct\ (iappend\ l_1\ l_2)\ i)$	(d'après Lemme A.3)
$= nth\ (decode\ (conv_{H_3}\ i) - lg\ l_1)\ (ilist2list\ l_2)\ (fct\ (iappend\ l_1\ l_2)\ i)$	(d'après Propriété 3.2)
$= fct\ l_2\ (code\ H_4)$	(d'après Lemme 3.29)

[$H_2 : decode\ i < lg\ l_1$**]** On a : $fct\ (iappend\ l_1\ l_2)\ i = fct\ l_1\ (code\ H_2)$ (d'après Propriété 3.6.2) et (en simplifiant) :

$$\begin{array}{ll} & nth\ (decode\ i)\ ((ilist2list\ l_1)@ilist2list\ l_2)\ (fct\ (iappend\ l_1\ l_2)\ i) \\ = & nth\ (decode\ i)\ (ilist2list\ l_1)\ (fct\ (iappend\ l_1\ l_2)\ i) \quad \text{(d'après Lemme A.2)} \\ = & fct\ l_1\ (code\ H_2) \quad \text{(d'après Lemme 3.29)} \end{array}$$

□

A.1.1.13 Preuve du Lemme 3.34

Rappel de l'énoncé. $\forall l_1\ l_2, ilist2list\ (iappend\ l_1\ l_2) = (ilist2list\ l_1)@(ilist2list\ l_2)$

Démonstration. On prouve aisément que

$$length\ (ilist2list\ (iappend\ l_1\ l_2)) = length\ (ilist2list\ l_1@ilist2list\ l_2)$$

On peut donc appliquer le Corollaire 3.36. On doit prouver que :

$$nth\ n\ (ilist2list\ (iappend\ l_1\ l_2))\ d = nth\ n\ (ilist2list\ l_1@ilist2list\ l_2)\ d$$

On sait que (en simplifiant) $H_1 : n < lg\ (iappend\ l_1\ l_2)$. D'après le Lemme 3.29, on a

$$nth\ n\ (ilist2list\ (iappend\ l_1\ l_2))\ d = fct\ (iappend\ l_1\ l_2)\ (code\ H_1)$$

Pour continuer la preuve, on va avoir besoin de savoir si l'élément que l'on cherche est dans la première ou la seconde *ilist*. Pour cela, on va comparer les valeurs de $lg\ l_1$ et de n.

[$H_2 : lg\ l_1 \leq n$**]** On déduit de H_1 et de H_2 que $H_3 : n - lg\ l_1 < lg\ l_2$. On a :

$$\begin{array}{rcll} nth\ n\ (ilist2list\ l_1@ilist2list\ l_2)\ d = & nth\ (n - lg\ l_1)\ (ilist2list\ l_2)\ d & \text{(d'après Lemme A.3)} \\ = & fct\ l_2\ (code\ H_3) & \text{(d'après Lemme 3.29)} \end{array}$$

et on déduit également de H_2 que $H_4 : lg\ l_1 \leq decode\ (conv_{H_5}\ (code\ H_1))$ avec $H_5 : lg\ (iappend\ l_1\ l_2) = lg\ l_1 + lg\ l_2$ déduit de la Propriété 3.6.1. D'après la Propriété 3.6.3, on a :

$$fct\ (iappend\ l_1\ l_2)\ (code\ H_1) = fct\ l_2\ (rightFin\ (conv_{H_5}\ (code\ H_1))\ H_4)$$

Il nous suffit donc maintenant de prouver que :

$$code\ H_3 = rightFin\ (conv_{H_5}\ (code\ H_1))\ H_4$$

ce qui est immédiat en utilisant le Lemme 3.8, la Propriété 3.2 et le Lemme 3.9.

[$H_2 : n < lg\ l_1$**]** De H_2 on déduit directement $H_3 : decode\ (code\ H_1) < lg\ l_1$. On a :

$$\begin{array}{rcll} nth\ n\ (ilist2list\ l_1@ilist2list\ l_2)\ d & = & nth\ n\ (ilist2list\ l_1)\ d & \text{(d'après Lemme A.2)} \\ & = & fct\ l_1\ (code\ H_2) & \text{(d'après Lemme 3.29)} \end{array}$$

et d'après la Propriété 3.6.2, on a : $fct\ (iappend\ l_1\ l_2)\ (code\ H_1) = fct\ l_1\ (code\ H_3)$. Pour finir la preuve, il nous suffit donc de prouver que : $code\ H_2 = code\ H_3$ ce qui est immédiat en utilisant les Lemmes 3.8 et 3.9.

□

A.1.1.14 Preuve du Lemme 3.42

Rappel de l'énoncé. $\forall l\ i, ilist2list\ (ileft\ l\ i)@(fct\ l\ i)::ilist2list(iright\ l\ i) = ilist2list\ l$

Démonstration. On va utiliser le Corollaire 3.36 pour faire la preuve. On montre aisément que : $length\ (ilist2list\ (ileft\ l\ i)@(fct\ l\ i)::ilist2list(iright\ l\ i)) = length\ (ilist2list\ l)$. On a maintenant $H_1 : n < length\ (ilist2list\ (ileft\ l\ i)@(fct\ l\ i)::ilist2list(iright\ l\ i))$ et on doit montrer que :

$$nth\ n\ (ilist2list\ (ileft\ l\ i)@(fct\ l\ i)::ilist2list(iright\ l\ i))\ d = nth\ n\ (ilist2list\ l)\ d$$

On transforme aisément H_1 en $n < lg\ l$. D'après le Lemme 3.29, on a :

$$nth\ n\ (ilist2list\ l)\ d = fct\ l\ (code\ H_1)$$

Pour continuer la preuve, on a besoin de savoir si l'élément correspondant à n est à "gauche" ou à "droite" de i. Pour cela, on compare les valeurs de n et de $decode\ i$:

[Cas $H_2 : decode\ i \leq n$**]** On a :

$$\begin{array}{ll} & nth\ n\ (ilist2list\ (ileft\ l\ i)@(fct\ l\ i)::ilist2list(iright\ l\ i))\ d \\ = & nth\ (n - decode\ i)\ (fct\ l\ i::ilist2list\ (iright\ l\ i))\ d \quad \text{(d'après Lemme A.3, en simplifiant)} \end{array}$$

On doit maintenant savoir si $n - decode\ i = 0$ ou non (pour trouver le résultat de nth) :

[Cas $H_3 : n - decode\ i = 0$**]** On a alors :

$$nth\ (n - decode\ i)\ (fct\ l\ i::ilist2list\ (iright\ l\ i))\ d = fct\ l\ i$$

Il nous suffit donc de montrer que $code\ H_1 = i$ qu'on simplifie en $n = decode\ i$ ce qui est immédiat avec H_3 et H_2.

[Cas $H_3 : n - decode\ i = m + 1$**]** On a alors :

$$nth\ (n - decode\ i)\ (fct\ l\ i::ilist2list\ (iright\ l\ i))\ d = nth\ m\ (ilist2list\ (iright\ l\ i))\ d$$

On montre que $H_4 : m < lg\ (iright\ l\ i)$ et on a :

$$\begin{array}{lcll} nth\ m\ (ilist2list\ (iright\ l\ i))\ d & = & fct\ (iright\ l\ i)\ (code\ H_4) & \text{(d'après Lemme 3.29)} \\ & = & fct\ l\ (code\ H_5) & \text{(d'après Définition 3.30,} \\ & & & \text{avec } H_5 : decode\ i + 1 + decode\ (code\ H_4) < lg\ l) \end{array}$$

Il nous suffit alors de montrer que $code\ H_1 = code\ H_5$. On a :

$$\begin{array}{lcl} decode\ (code\ H_5) & = & decode\ i + 1 + decode\ (code\ H_4) = decode\ i + 1 + m \\ & = & decode\ i + n - decode\ i = n = decode\ (code\ H_1) \end{array}$$

Donc, d'après le Lemme 3.9, $code\ H_5 = code\ H_1$.

[Cas $H_2 : n < decode\ i$**]** On déduit de H_2 que $H_3 : n < lg\ (ileft\ l\ i)$. On a :

$$\begin{array}{lll} & nth\ n\ (ilist2list\ (ileft\ l\ i)@(fct\ l\ i)::ilist2list(iright\ l\ i))\ d & \\ = & nth\ n\ (ilist2list\ (ileft\ l\ i))\ d & \text{(d'après Lemme A.2)} \\ = & fct\ (ileft\ l\ i)\ (code\ H_3) & \text{(d'après Lemme 3.29)} \\ = & fct\ l\ (code\ H_4) & \text{(d'après Définition 3.28,} \\ & & \text{avec } H_4 : decode\ (code\ H_3) < lg\ l) \end{array}$$

Il nous suffit donc de prouver que $code\ H_4 = code\ H_1$. On a :

$$decode\ (code\ H_4) = decode\ (code\ H_3) = n = decode\ (code\ H_1)$$

Donc, d'après le Lemme 3.9, $code\ H_4 = code\ H_1$.

□

A.1.1.15 Preuve du Corollaire 3.43

Rappel de l'énoncé. $\forall l\ i, ilist\_rel_{eq}\ (iappend\ (ileft\ l\ i)\ (icons\ (fct\ l\ i)\ (iright\ l\ i)))\ l$

Démonstration. D'après le Lemme 3.25, pour montrer

$$ilist\_rel_{eq}\ (iappend\ (ileft\ l\ i)\ (icons\ (fct\ l\ i)\ (iright\ l\ i)))\ l$$

il nous suffit de montrer que

$$ilist2list\ (iappend\ (ileft\ l\ i)\ (icons\ (fct\ l\ i)\ (iright\ l\ i))) = ilist2list\ l$$

Et

$$\begin{array}{lll} & ilist2list\ (iappend\ (ileft\ l\ i)\ (icons\ (fct\ l\ i)\ (iright\ l\ i))) & \\ = & ilist2list\ (ileft\ l\ i)@(ilist2list\ (icons\ (fct\ l\ i)\ (iright\ l\ i))) & \text{(d'après Lemme 3.34)} \\ = & ilist2list\ (ileft\ l\ i)@(fct\ l\ i)::(ilist2list\ (iright\ l\ i)) & \text{(d'après Lemme 3.41)} \\ = & ilist2list\ l & \text{(d'après Lemme 3.42)} \end{array}$$

□

A.1.1.16 Preuve du Lemme 3.44

Rappel de l'énoncé.

$$\exists (f_1 : ilistMult\ T\ 0\ None \to ilist\ T)(f_2 : ilist\ T \to ilistMult\ T\ 0\ None), f_1 \circ f_2 = f_2 \circ f_1 = id$$

Démonstration. On sait que $list2ilist \circ ilistMult2list$ est de type : $ilistMult\ T\ 0\ None \to ilist\ T$. Soit $f_1 := list2ilist \circ ilistMult2list$ et de la même façon soit $f_2 := list2ilistMult \circ ilist2list$.

Montrons que $f_1 \circ f_2 = id$.

$$\begin{array}{rcl} f_1 \circ f_2 & = & (list2ilist \circ ilistMult2list) \circ (list2ilistMult \circ ilist2list) \\ & = & list2ilist \circ (ilistMult2list \circ list2ilistMult) \circ ilist2list \\ & = & list2ilist \circ id \circ ilist2list \\ & = & list2ilist \circ ilist2list \\ & = & id \end{array}$$

La preuve que $f_2 \circ f_1 = id$ se fait de la même façon. On a donc $f_1 \circ f_2 = f_2 \circ f_1 = id$. □

A.1.2 Permutations - Preuves

A.1.2.1 Preuve du Lemme 4.2

Rappel de l'énoncé. $\forall l_1\, l_2, ilist\_rel_{R_d}\ l_1\ l_2 \Rightarrow iperm\_occ_{R_d}\ l_1\ l_2$.

Démonstration. Rappelons qu'on suppose que R est une relation d'équivalence (et R_d dénote le fait que R est décidable). On sait que

$$ilist\_rel_{R_d}\ l_1\ l_2 \Leftrightarrow \exists h : lg\ l_1 = lg\ l_2, \forall i : Fin\ (lg\ l_1), R_d\ (fct\ l_1\ i)\ (fct\ l_2\ (conv_h\ i))$$

Et on notera également n_1, n_2, ln_1 et ln_2 les éléments tels que : $l_1 = \langle n_1, ln_1 \rangle$ et $l_2 = \langle n_2, ln_2 \rangle$. On obtient donc deux nouvelles hypothèses :

$$H_1 : n_1 = n_2 \quad H_2 : \forall i, R_d\ (ln_1\ i)\ (ln_2\ (conv_{H_1}\ i))$$

ln_2 est de type *ilistn* n_2 T, mais grâce à H_1, on peut réécrire son type en *ilistn* n_1 T (comme ln_1) et en utilisant l'injectivité de *decode*, on obtient finalement une nouvelle forme pour $H_2 : \forall i, R\ (ln_1\ i)\ (ln_2\ i)$.

On procède maintenant par induction sur n_1 pour prouver que $iperm\_occ_{R_d}\ l_1\ l_2$, c'est-à-dire, $\forall t, nbocc_{R_d}\ t\ \langle n_1, ln_1 \rangle = nbocc_{R_d}\ t\ \langle n_1, ln_2 \rangle$.

[Cas 0] $nbocc_{R_d}\ t\ \langle 0, ln_1 \rangle = nbocc_{R_d}\ t\ \langle 0, ln_2 \rangle \Leftrightarrow 0 = 0$ (les deux *ilist* sont vides).

[Cas $n_1 + 1$**]** L'hypothèse d'induction IH est :

$$\forall ln_1'\ ln_2' : ilistn\ T\ n_1, (\forall i, R\ (ln_1'\ i)\ (ln_2'\ i) \Rightarrow \forall t, nbocc_{R_d}\ t\ \langle n_1, ln_1' \rangle = nbocc_{R_d}\ t\ \langle n_1, ln_2' \rangle)$$

On a, pour tout t

$$\begin{array}{l} nbocc_{R_d}\ t\ \langle n_1 + 1, ln_1' \rangle = nbocc_{R_d}\ t\ \langle n_1 + 1, ln_2' \rangle \\ \quad \Leftrightarrow \quad if\ (R_d\ t\ (ln_1\ (first\ n_1)))\ then\ nbocc_{R_d}\ t\ \langle n_1, ln_1 \circ succ \rangle + 1 \\ \qquad else\ nbocc_{R_d}\ t\ \langle n_1, ln_1 \circ succ \rangle = \\ \qquad if\ (R_d\ t\ (ln_2\ (first\ n_1)))\ then\ nbocc_{R_d}\ t\ \langle n_1, ln_2 \circ succ \rangle + 1 \\ \qquad else\ nbocc_{R_d}\ t\ \langle n_1, ln_2 \circ succ \rangle \end{array}$$

On a donc ici quatre cas à analyser :

[Cas $R_d\ t\ (ln_1\ (first\ n_1)) \wedge R_d\ t\ (ln_2\ (first\ n_1))$**]**
On veut prouver : $nbocc_{R_d}\ t\ \langle n_1, ln_1 \circ succ \rangle + 1 = nbocc_{R_d}\ t\ \langle n_1, ln_2 \circ succ \rangle + 1$.
C'est-à-dire $nbocc_{R_d}\ t\ \langle n_1, ln_1 \circ succ \rangle = nbocc_{R_d}\ t\ \langle n_1, ln_2 \circ succ \rangle$.
On utilise donc IH, et il nous reste à prouver : $\forall i, R\ (ln_1\ (succ\ i))\ (ln_2\ (succ\ i))$.
Or ceci est vrai d'après H_2.

[Cas $R_d\ t\ (ln_1\ (first\ n_1)) \wedge \neg(R_d\ t\ (ln_2\ (first\ n_1)))$**]**
On veut prouver : $nbocc_{R_d}\ t\ \langle n_1, ln_1 \circ succ \rangle + 1 = nbocc_{R_d}\ t\ \langle n_1, ln_2 \circ succ \rangle$.
En utilisant H_2 on obtient $R_d\ (ln_1\ (first\ n_1))\ (ln_2\ (first\ n_1))$. Par transitivité sur R_d avec l'hypothèse que $R_d\ t\ (ln_1\ (first\ n_1))$ on obtient $R_d\ t\ (ln_2\ (first\ n_1))$. Ce qui est une contradiction avec notre hypothèse de départ $\neg(R_d\ t\ (ln_2\ (first\ n_1)))$.

[Cas $\neg(R_d\ t\ (ln_1\ (first\ n_1))) \wedge R_d\ t\ (ln_2\ (first\ n_1))$**]**
On veut prouver : $nbocc_{R_d}\ t\ \langle n_1, ln_1 \circ succ \rangle = nbocc_{R_d}\ t\ \langle n_1, ln_2 \circ succ \rangle + 1$ Le raisonnement ici est symétrique à celui du cas précédent, nous ne l'écrivons donc pas ici de nouveau.

[Cas $\neg(R_d\ t\ (ln_1\ (first\ n_1))) \wedge \neg(R_d\ t\ (ln_2\ (first\ n_1)))$**]**
On veut prouver : $nbocc_{R_d}\ t\ \langle n_1, ln_1 \circ succ \rangle = nbocc_{R_d}\ t\ \langle n_1, ln_2 \circ succ \rangle$.
On utilise IH, et il nous reste à prouver : $\forall i, R\ (ln_1\ (succ\ i))\ (ln_2\ (succ\ i))$ ce qui est vrai d'après H_2.

□

A.1.2.2 Preuve du Lemme 4.5

Rappel de l'énoncé.

$$\forall n\ ln_1\ ln_2\ i,\ ilist\_rel_R\ \langle n, ln_1 \rangle\ \langle n, ln_2 \rangle \Rightarrow ilist\_rel_R\ (remEl\ \langle n, ln_1 \rangle\ i)\ (remEl\ \langle n, ln_2 \rangle\ i)$$

Démonstration. De $ilist\_rel_R\ \langle n, ln_1\rangle\ \langle n, ln_2\rangle$ on déduit les hypothèses :

$$H_1 : n = n \quad \text{et} \quad H_2 : \forall i,\ R\ (ln_1\ i)\ (ln_2\ (conv_{H_1}\ i))$$

On simplifie aisément H_2 en : $H : \forall i,\ R\ (ln_1\ i)\ (ln_2\ i)$

Comme on a par définition $lg\ \langle n, ln_1\rangle = lg\ \langle n, ln_2\rangle$, on peut facilement montrer :

$$H_3 : lg\ (remEl\ \langle n, ln_1\rangle\ i) = lg\ (remEl\ \langle n, ln_2\rangle\ i)$$

On peut donc appliquer la définition de *ilist_rel*. On doit montrer que :

$$\forall i',\ R\ (fct\ (remEl\ \langle n, ln_1\rangle\ i)\ i')\ (fct\ (remEl\ \langle n, ln_2\rangle\ i)\ (conv_{H_3}\ i'))$$

Pour appliquer les Propriétés 4.1.2 et 4.1.3 on va comparer *decode i* et *decode i'*.

[Cas $i' <_{Fin} i$**]** D'après la Propriété 3.2 on a aussi $conv_{H_3}\ i' <_{Fin} i$. On utilise la Propriété 4.1.2 et on a :

$$\begin{array}{l} R\ (fct\ (remEl\ \langle n, ln_1\rangle\ i)\ i')\ (fct\ (remEl\ \langle n, ln_2\rangle\ i)\ (conv_{H_3}\ i')) \\ \Leftrightarrow R\ (ln_1\ (conv_{h_1}\ (weakFin\ i')))\ (ln_2\ (conv_{h_2}\ (weakFin\ (conv_{H_3}\ i')))) \end{array}$$

Avec h_1 et h_2 de type $lg\ (remEl\ \langle n, ln_1\rangle\ i) + 1 = n$ (resp. $lg\ (remEl\ \langle n, ln_2\rangle\ i) + 1 = n$) obtenus grâce à la Propriété 4.1.1.

On peut facilement montrer que : $conv_{h_2}\ (weakFin\ (conv_{H_3}\ i')) = conv_{h_1}\ (weakFin\ i')$ (en utilisant la Propriété 3.2 et les Lemmes 4.3 et 3.9). On obtient alors le but :

$$R\ (ln_1\ (conv_{h_1}\ (weakFin\ i')))\ (ln_2\ (conv_{h_1}\ (weakFin\ i')))$$

Ce qui est vrai d'après H.

[Cas $i \leq_{Fin} i'$**]** D'après la Propriété 3.2 on a aussi $i \leq_{Fin} conv_{H_3}\ i'$. On utilise la Propriété 4.1.3 et on a :

$$\begin{array}{l} R\ (fct\ (remEl\ \langle n, ln_1\rangle\ i)\ i')\ (fct\ (remEl\ \langle n, ln_2\rangle\ i)\ (conv_{H_3}\ i')) \\ \Leftrightarrow R\ (ln_1\ (conv_{h_1}\ (succ\ i')))\ (ln_2\ (conv_{h_2}\ (succ\ (conv_{H_3}\ i')))) \end{array}$$

Avec h_1 et h_2 de type $lg\ (remEl\ \langle n, ln_1\rangle\ i) + 1 = n$ (resp. $lg\ (remEl\ \langle n, ln_2\rangle\ i) + 1 = n$) obtenus grâce à la Propriété 4.1.1.

On peut facilement montrer que : $conv_{h_2}\ (succ\ (conv_{H_3}\ i')) = conv_{h_1}\ (succ\ i')$ (en utilisant la Propriété 3.2 et le Lemme 3.9). On obtient alors le but :

$$R\ (ln_1\ (conv_{h_1}\ (succ\ i')))\ (ln_2\ (conv_{h_1}\ (succ\ i')))$$

Ce qui est vrai d'après H.

□

A.1.2.3 Preuve du Lemme 4.6

Rappel de l'énoncé.

$$\forall (f : T \to U)\ (l : ilist\ T)\ (i : Fin\ (lg\ l)), ilist\_rel_{eq}\ (remEl\ (imap\ f\ l)\ i)\ (imap\ f\ (remEl\ l\ i))$$

Démonstration. On a

$$\begin{array}{l} ilist\_rel_{eq}\ (remEl\ (imap\ f\ l)\ i)\ (imap\ f\ (remEl\ l\ i)) \\ \Leftrightarrow \quad \exists h : lg\ (remEl\ (imap\ f\ l)\ i) = lg\ (imap\ f\ (remEl\ l\ i)), \\ \qquad \forall i', fct\ (remEl\ (imap\ f\ l)\ i)\ i' = fct\ (imap\ f\ (remEl\ l\ i))\ (conv_h\ i') \end{array}$$

On démontre facilement, en utilisant la Propriété 4.1.1 que

$$\exists h : lg\ (remEl\ (imap\ f\ l)\ i) = lg\ (imap\ f\ (remEl\ l\ i))$$

On doit donc maintenant démontrer que, quelque soit i' :

$$fct\ (remEl\ (imap\ f\ l)\ i)\ i' = fct\ (imap\ f\ (remEl\ l\ i))\ (conv_h\ i')$$

On a

$$\begin{array}{l} fct\ (remEl\ (imap\ f\ l)\ i)\ i' = fct\ (imap\ f\ (remEl\ l\ i))\ (conv_h\ i') \\ \Leftrightarrow \quad fct\ (remEl\ (imap\ f\ l)\ i)\ i' = f\ (fct\ (remEl\ l\ i)\ (conv_h\ i')) \quad \text{(en utilisant le Lemme 3.32)} \end{array}$$

Pour pouvoir utiliser les Propriétés 4.1.2 et 4.1.3, on compare les valeurs de *decode* i et *decode* i' :

[Cas $i \leq_{Fin} i'$**]** On a :

$$\begin{array}{lll} & fct\ (remEl\ (imap\ f\ l)\ i)\ i' & \\ = & fct\ (imap\ f\ l)\ (conv_{h'}\ (succ\ i')) & \text{(d'après Propriété 4.1.3 et où } h' \text{ a pour type} \\ & & lg\ (remEl\ (imap\ f\ l)\ i) + 1 = lg\ (imap\ f\ l) \\ & & \text{et est déduit de la Propriété 4.1.1)} \\ = & f\ (fct\ l\ (conv_{h'}\ (succ\ i'))) & \text{(d'après Lemme 3.32)} \end{array}$$

et :

$$\begin{array}{lll} & f\ (fct\ (remEl\ l\ i)\ (conv_h\ i')) & \\ = & f\ (fct\ l\ (conv_{h''}\ (succ\ (conv_h\ i')))) & \text{(d'après Propriété 4.1.3 et où } h'' \text{ a pour type} \\ & & lg\ (remEl\ l\ i) + 1 = lg\ l \text{ et est déduit de la} \\ & & \text{Propriété 4.1.1, et où on prouve facilement} \\ & & i \leq_{Fin} conv_h\ i' \text{ avec la Propriété 3.2)} \end{array}$$

Donc, pour prouver que $fct\ (remEl\ (imap\ f\ l)\ i)\ i' = f\ (fct\ (remEl\ l\ i)\ (conv_h\ i'))$, il suffit de prouver que :

$$\begin{array}{lll} & conv_{h'}\ (succ\ i') = conv_{h''}\ (succ\ (conv_h\ i')) & \\ \Leftrightarrow & conv_{h'}\ (succ\ i') =_{Fin} conv_{h''}\ (succ\ (conv_h\ i')) & \text{(d'après Lemme 3.9)} \\ \Leftrightarrow & succ\ i' =_{Fin} succ\ (conv_h\ i') & \text{(d'après Propriété 3.2)} \\ \Leftrightarrow & i' =_{Fin} conv_h\ i' & \\ \Leftrightarrow & i' =_{Fin} i' & \text{(d'après Propriété 3.2)} \end{array}$$

[Cas $i' <_{Fin} i$**]** La preuve ici est tout à fait équivalente à la précédente (mais on utilise la Propriété 4.1.2 au lieu de la Propriété 4.1.3). On ne la détaille donc pas.

□

A.1.2.4 Preuve du Lemme 4.7

Rappel de l'énoncé. $\forall l\ i, ilist\_rel_{eq}\ (iappend\ (ileft\ l\ i)\ (iright\ l\ i))\ (remEl\ l\ i)$

Démonstration. On montre facilement que

$$H_1 : lg\ (iappend\ (ileft\ l\ i)\ (iright\ l\ i)) = lg\ (remEl\ l\ i)$$

On peut donc appliquer la Définition 3.11 et on doit montrer que :

$$\forall i', fct\ (iappend\ (ileft\ l\ i)\ (iright\ l\ i))\ i' = fct\ (remEl\ l\ i)\ (conv_{H_1}\ i')$$

Pour pouvoir utiliser les propriétés sur *iappend* et *remEl*, nous allons comparer les valeurs de i et de i' :

[$H_2 : i \leq_{Fin} i'$**]** D'après la Propriété 3.7.1, on sait que $decode\ i = lg\ (ileft\ l\ i)$. On définit donc

$$H'_2 : lg\ (ileft\ l\ i) \leq decode\ (conv_H\ i')$$
avec $H : lg\ (iappend\ (ileft\ l\ i)\ (iright\ l\ i)) = lg\ (ileft\ l\ i) + lg\ (iright\ l\ i)$
déduit de la Propriété 3.6.1

On a :

$$\begin{array}{lll} & fct\ (iappend\ (ileft\ l\ i)\ (iright\ l\ i))\ i' & \\ = & fct\ (iright\ l\ i)\ (rightFin\ (conv_H\ i')\ H'_2) & \text{(d'après Propriété 3.6.3)} \\ = & fct\ l\ (code\ H_3) & \text{(d'après Définition 3.30, et où} \\ & & H_3 : decode\ i + 1 + decode(rightFin\ (conv_H\ i')\ H'_2) < lg\ l) \end{array}$$

et

$$fct\ (remEl\ l\ i)\ (conv_{H_1}\ i') = fct\ l\ (conv_{H_4}(succ(conv_{H_1}\ i'))) \quad \text{(d'après Propriété 4.1.3 avec } H_4 : lg(remEl\ l\ i) + 1 = lg\ l)$$

Donc pour montrer que : $fct\ (iappend\ (ileft\ l\ i)\ (iright\ l\ i))\ i' = fct\ (remEl\ l\ i)\ (conv_{H_1}\ i')$, il nous suffit de montrer que

$$code\ H_3 = conv_{H_4}\ (succ\ (conv_{H_1}\ i'))$$

ou encore

$$code\ H_3 =_{Fin} conv_{H_4}\ (succ\ (conv_{H_1}\ i'))$$

On a

$$decode\ (conv_{H_4}\ (succ\ (conv_{H_1}\ i'))) = decode\ i' + 1$$

et

$$\begin{array}{rcl} decode\ (code\ H_3) & = & decode\ i + 1 + decode\ (rightFin\ (conv_H\ i')\ H'_2) \\ & = & decode\ i + 1 + decode\ i' - lg\ (ileft\ l\ i) \\ & = & decode\ i + 1 + decode\ i' - decode\ i \\ & = & decode\ i' + 1 \end{array}$$

[$H_2 : i' <_{Fin} i$**]** D'après la Propriété 3.7.1, on sait que $decode\ i = lg\ (ileft\ l\ i)$. On définit donc $H'_2 : decode\ i' < lg\ (ileft\ l\ i)$. On a :

$$\begin{array}{lll} & fct\ (iappend\ (ileft\ l\ i)\ (iright\ l\ i))\ i' & \\ = & fct\ (ileft\ l\ i)\ (code\ H'_2) & \text{(d'après Propriété 3.6.2)} \\ = & fct\ l\ (code\ H_3) & \text{(d'après Définition 3.28, et où} \\ & & H_3 : decode\ (code\ H'_2) < lg\ l) \end{array}$$

et

$$fct\ (remEl\ l\ i)\ (conv_{H_1}\ i') = fct\ l\ (conv_{H_4}(weakFin(conv_{H_1}\ i'))) \quad \text{(d'après Propriété 4.1.2 avec } H_4 : lg\ (remEl\ l\ i) + 1 = lg\ l\text{)}$$

Donc pour montrer que : $fct\ (iappend\ (ileft\ l\ i)\ (iright\ l\ i))\ i' = fct\ (remEl\ l\ i)\ (conv_{H_1}\ i')$, il nous suffit de montrer que

$$code\ H_3 = conv_{H_4}\ (weakFin\ (conv_{H_1}\ i'))$$

ou encore

$$code\ H_3 =_{Fin} conv_{H_4}\ (weakFin\ (conv_{H_1}\ i'))$$

On a

$$decode\ (conv_{H_4}\ (weakFin\ (conv_{H_1}\ i'))) = decode\ i'$$

et

$$decode\ (code\ H_3) = decode\ (code\ H_2') = decode\ i'$$

□

A.1.2.5 Preuve du Lemme 4.9

Rappel de l'énoncé. *$\forall l\ i, ilist\_rel_{eq}\ l\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i))$ avec h déduit de la Propriété 4.1.1.*

Démonstration. D'après la Définition 3.11 on a que

$$\begin{array}{ll} & \forall l\ i, ilist\_rel_{eq}\ l\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i)) \\ \Leftrightarrow & \forall l\ i\ \exists h_1 : lg\ l = lg\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i)), \\ & \forall i', fct\ l\ i' = fct\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i))\ (conv_{h_1}\ i') \end{array}$$

Des Propriétés 4.2.1 et 4.1.1, on déduit l'hypothèse

$$h_1 : lg\ l = lg\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i))$$

On doit donc prouver que

$$\forall i', fct\ l\ i' = fct\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i))\ (conv_{h_1}\ i')$$

Pour pouvoir utiliser les caractérisations du comportement de *remEl* et *addEl*, nous allons procéder par étude de cas sur $decode\ (conv_h\ i)$ et $decode\ (conv_{h_1}\ i')$ (ou de façon équivalente d'après la Propriété 3.2, sur $decode\ i$ et $decode\ i'$).

[Cas $i' <_{Fin} i$] On a :

$$\begin{array}{ll} & fct\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i))\ (conv_{h_1}\ i') \\ = & fct\ (remEl\ l\ i)\ (code\ h_2) \quad \text{(d'après Propriété 4.2.2, avec } h_2 \text{ preuve de } decode\ (conv_{h_1}\ i') < lg\ (remEl\ l\ i) \text{ et on a } code\ h_2 <_{Fin} i \text{ car } code\ h_2 =_{Fin} i'\text{)} \\ = & fct\ l\ (conv_{h_3}\ (weakFin\ (code\ h_2))) \quad \text{(d'après Propriété 4.1.2, avec } h_3 \text{ de type } lg\ (remEl\ l\ i) + 1 = lg\ l \text{ déduit de la Propriété 4.1.1)} \end{array}$$

Or,

$$\begin{array}{lll} conv_{h_3}\ (weakFin\ (code\ h_2)) & =_{Fin} & weakFin\ (code\ h_2) \quad \text{(d'après Propriété 3.2)} \\ & =_{Fin} & code\ h_2 \quad \text{(d'après Lemme 4.3)} \\ & =_{Fin} & i' \quad \text{(d'après Lemme 3.8)} \end{array}$$

Donc, $fct\ l\ (conv_{h_3}\ (weakFin\ (code\ h_2))) = fct\ l\ i'$

[Cas $i' =_{Fin} i$**]** On a :

$$\begin{array}{ll} & fct\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i))\ (conv_{h_1}\ i') \\ = & fct\ l\ i \quad \text{(d'après Propriété 4.2.3)} \\ = & fct\ l\ i' \quad \text{(d'après Lemme 3.9)} \end{array}$$

[Cas $i <_{Fin} i'$**]** On a :

$$\begin{array}{ll} & fct\ (addEl\ (remEl\ l\ i)\ (fct\ l\ i)\ (conv_h\ i))\ (conv_{h_1}\ i') \\ = & fct\ (remEl\ l\ i)\ (getcons\ (conv_{h_2}\ i')\ h_3) \quad \big(\text{d'après Propriété 4.2.4, avec } h_2 : lg\ l = lg\ (remEl\ l\ i) + 1 \text{ et } h_3 : 0 < decode\ (conv_{h_2}\ i')\big) \\ = & fct\ l\ (conv_{h_4}\ (succ\ (getcons\ (conv_{h_2}\ i')\ h_3))) \quad \text{(d'après Propriété 4.1.3, car } i \leq_{Fin} getcons\ (conv_{h_2}\ i')\ h_3 \text{ puisque } succ(getcons\ (conv_{h_2}\ i')\ h_3) =_{Fin} i' \text{ et avec } h_4 \text{ déduit de Propriété 4.1.1)} \\ = & fct\ l\ i' \quad \text{(d'après Propriété 3.1.1)} \end{array}$$

□

A.1.2.6 Preuve du Lemme 4.14

Rappel de l'énoncé.

$$\forall n\ i\ i'\ (h : i \neq_{Fin} indexFromRemEl\ i\ i'), indexInRemEl\ i\ (indexFromRemEl\ i\ i')\ h = i'$$

Démonstration. On compare les valeurs de i et i'. Nous avons deux cas.

[Cas $H_1 : i \leq_{Fin} i'$**]** On a :

$$\begin{array}{lll} & decode\ (indexInRemEl\ i\ (indexFromRemEl\ i\ i')\ h) + 1 & \text{(le +1 est pour utiliser la Propriété 4.3.1)} \\ = & decode(indexInRemEl\ i\ (succ\ i')\ h) + 1 & \text{(d'après Définition 4.16 et avec } h : i \neq_{Fin} succ\ i') \\ = & decode\ (succ\ i') & \text{(d'après Propriété 4.3.1 ; il est aisé de prouver } i <_{Fin} succ\ i' \text{ à partir de } H_1) \\ = & decode\ i' + 1 & \text{(d'après Définition 3.3)} \end{array}$$

Donc, en simplifiant et en utilisant le Lemme 3.9, on obtient :

$$indexInRemEl\ i\ (indexFromRemEl\ i\ i')\ h = i'$$

[Cas $H_1 : i' <_{Fin} i$**]** On a :

$$\begin{array}{lll} & \mathit{indexInRemEl}\ i\ (\mathit{indexFromRemEl}\ i\ i')\ h & \\ = & \mathit{indexInRemEl}\ i\ (\mathit{weakFin}\ i')\ h & \text{(d'après Définition 4.16 et avec } h : i \neq_{Fin} \mathit{weakFin}\ i') \\ =_{Fin} & \mathit{weakFin}\ i' & \text{(d'après Propriété 4.3.2 ; avec le Lemme 4.3 et } H_1\text{, la preuve de } \mathit{weakFin}\ i' <_{Fin} i \text{ est immédiate)} \\ =_{Fin} & i' & \text{(d'après Lemme 4.3)} \end{array}$$

Donc $\mathit{indexInRemEl}\ i\ (\mathit{indexFromRemEl}\ i\ i')\ h = i'$

□

A.1.2.7 Preuve du Lemme 4.15

Rappel de l'énoncé. $\forall n\ i\ i'\ (h : i \neq_{Fin} i'), \mathit{indexFromRemEl}\ i\ (\mathit{indexInRemEl}\ i\ i'\ h) = i'$

Démonstration. On compare les valeurs de i et i'. Sachant que $i \neq_{Fin} i'$, nous avons deux cas.
[Cas $H_1 : i <_{Fin} i'$**]** Afin d'utiliser la Définition 4.16, montrons $H_1' : i \leq_{Fin} \mathit{indexInRemEl}\ i\ i'\ h$.

$$\begin{array}{ll} & i <_{Fin} i' \\ \Rightarrow & \mathit{decode}\ i < \mathit{decode}\ (\mathit{indexInRemEl}\ i\ i'\ h) + 1 \quad \text{(d'après Propriété 4.3.1)} \\ \Leftrightarrow & i \leq_{Fin} \mathit{indexInRemEl}\ i\ i'\ h \end{array}$$

On peut maintenant utiliser la définition de *indexFromRemEl* dans l'hypothèse H_1' :

$$\mathit{indexFromRemEl}\ i\ (\mathit{indexInRemEl}\ i\ i'\ h) = \mathit{succ}\ (\mathit{indexInRemEl}\ i\ i'\ h)$$

On a donc

$$\begin{array}{lll} & \mathit{decode}\ (\mathit{indexFromRemEl}\ i\ (\mathit{indexInRemEl}\ i\ i'\ h)) & \\ = & \mathit{decode}\ (\mathit{indexInRemEl}\ i\ i'\ h) + 1 & \text{(par définition de } \mathit{decode}\text{)} \\ = & \mathit{decode}\ i' & \text{(d'après Propriété 4.3.1)} \end{array}$$

En utilisant une fois de plus le Lemme 3.9, on obtient le résultat attendu.
[Cas $H_1 : i' <_{Fin} i$**]** Afin d'utiliser la Définition 4.16, montrons $H_1' : \mathit{indexInRemEl}\ i\ i'\ h <_{Fin} i$.

$$i' <_{Fin} i \Rightarrow \mathit{indexInRemEl}\ i\ i'\ h <_{Fin} i \quad \text{(d'après Propriété 4.3.2)}$$

On peut maintenant utiliser la définition de *indexFromRemEl* avec l'hypothèse H_1' :

$$\begin{array}{lll} & \mathit{indexFromRemEl}\ i\ (\mathit{indexInRemEl}\ i\ i'\ h) & \\ =_{Fin} & \mathit{weakFin}(\mathit{indexInRemEl}\ i\ i'\ h) & \text{(par définition de } \mathit{indexFromRemEl}\text{, avec } H_1'\text{)} \\ =_{Fin} & \mathit{indexInRemEl}\ i\ i'\ h & \text{(d'après Lemme 4.3)} \\ =_{Fin} & i' & \text{(d'après Propriété 4.3.2)} \end{array}$$

□

A.1.2.8 Preuve du Lemme 4.18

Rappel de l'énoncé. $\forall ln_1\ ln_2, \mathit{iperm\_ind}'_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle$

Démonstration. On doit prouver que

1. $lg\ l_1 = lg\ l_2$, ce qui est évident puisque $lg\ l_1 = lg\ l_2 = 0$.
2. $(\forall i_1 \exists i_2, R\ (\mathit{fct}\ l_1\ i_1)\ (\mathit{fct}\ l_2\ i_2) \wedge\ \mathit{iperm\_ind}'_R\ (\mathit{remEl}\ l_1\ i_1)\ (\mathit{remEl}\ l_2\ i_2))$. Ceci est trivial également puisque i_1 est de type $\mathit{Fin}\ (lg\ l_1)$ c'est-à-dire $\mathit{Fin}\ 0$, qui est vide.

□

A.1.2.9 Preuve du Lemme 4.19

Rappel de l'énoncé. $\forall l_1\ l_1'\ l_2, ilist\_rel_{eq}\ l_1\ l_1' \to iperm\_ind_R\ l_1\ l_2 \to iperm\_ind_R\ l_1'\ l_2$

Démonstration. Comme on l'a dit, on ne va faire la preuve que pour Définition 4.17. On nomme H_1 l'hypothèse $ilist\_rel_{eq}\ l_1\ l_1'$ et $H_2 : iperm\_ind_R\ l_1\ l_2$. On raisonne par induction sur H_2. De H_1, on peut tirer les deux hypothèses supplémentaires suivantes :

$$H_3 : lg\ l_1 = lg\ l_1' \quad \text{et} \quad H_4 : \forall i, fct\ l_1\ i = fct\ l_1'\ (conv_{H_3}\ i)$$

[Case de base] On a $H_5 : lg\ l_1 = lg\ l_2 = 0$. Selon la Définition 4.17, il nous suffit de prouver que : $lg\ l_1' = lg\ l_2 = 0$. On obtient cela par transitivité entre le symétrique de H_3 et H_5.

[Case inductif] On a i_1 et i_2 tels que :

$$H_5 : R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \quad \text{et} \quad H_6 : iperm\_ind_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$$

L'hypothèse d'induction est :

$$IH : \forall l_1',\ ilist\_rel_{eq}\ (remEl\ l_1\ i_1)\ l_1' \Rightarrow iperm\_ind_R\ l_1'\ (remEl\ l_2\ i_2)$$

Selon la Définition 4.17 appliquée avec $conv_{H_3}\ i_1$ et i_2, il nous suffit de prouver que :

1. $R\ (fct\ l_1'\ (conv_{H_3}\ i_1))\ (fct\ l_2\ i_2)$
   On réécrit avec H_4 et on obtient : $R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)$, ce qui est vrai d'après H_5.
2. $iperm\_ind_R\ (remEl\ l_1'\ (conv_{H_3}\ i_1))\ (remEl\ l_2\ i_2)$
   On applique IH. On doit maintenant prouver que :

   $$ilist\_rel_{eq}\ (remEl\ l_1\ i_1)\ (remEl\ l_1'\ (conv_{H_3}\ i_1))$$

   On prouve cela en appliquant le Lemme 4.5.

. □

A.1.2.10 Preuve du Lemme 4.27

Rappel de l'énoncé. *Pour deux relations R_1 et R_2 fixées, soit R_3 la relation définie par*

$$\forall t_1\ t_3, R_3\ t_1\ t_3 \Leftrightarrow \exists t_2, R_1\ t_1\ t_2 \wedge R_2\ t_2\ t_3$$

On a alors

$$\forall l_1\ l_2\ l_3, iperm\_ind_{R_1}\ l_1\ l_2 \wedge iperm\_ind_{R_2}\ l_2\ l_3 \Rightarrow iperm\_ind_{R_3}\ l_1\ l_3$$

Démonstration. Comme dans la preuve du Lemme 4.26, on va raisonner avec $iperm\_ind'$. On raisonne par induction sur $iperm\_ind'_{R_1}\ l_1\ l_2$. L'induction nous donne $H_1 : lg\ l_1 = lg\ l_2$ et l'hypothèse d'induction est

$$\begin{aligned} IH : \quad & \forall i_1 \exists i_2, \quad R_1\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \wedge iperm\_ind'_{R_1}\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2) \\ & \wedge \quad (\forall l_3, iperm\_ind'_{R_2}\ (remEl\ l_2\ i_2)\ l_3 \Rightarrow iperm\_ind'_{R_3}\ (remEl\ l_1\ i_1)\ l_3) \end{aligned}$$

L'hypothèse $iperm\_ind'_{R_2}\ l_2\ l_3$ nous donne :

$$\begin{aligned} & H_2 : lg\ l_2 = lg\ l_3 \\ & H_3 : \forall i_2 \exists i_3, R_2\ (fct\ l_2\ i_2)\ (fct\ l_3\ i_3) \wedge iperm\_ind'_{R_2}\ (remEl\ l_2\ i_2)\ (remEl\ l_3\ i_3) \end{aligned}$$

D'après la Définition 4.18, on doit montrer que :

1. $lg\ l_1 = lg\ l_3$: on le montre par transitivité de H_1 et H_2.
2. $\forall i_1 \exists i_3, R_3\ (fct\ l_1\ i_1)\ (fct\ l_3\ i_3) \wedge iperm\_ind'_{R_3}\ (remEl\ l_1\ i_1)\ (remEl\ l_3\ i_3)$: IH appliqué à i_1 nous donne i_2 tel que :

$$H_4 : R_1\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)$$
$$H_5 : \forall l_3, iperm\_ind'_{R_2}\ (remEl\ l_2\ i_2)\ l_3 \Rightarrow iperm\_ind'_{R_3}\ (remEl\ l_1\ i_1)\ l_3$$

et H_3 appliqué à i_2 nous donne i_3 tel que :

$$H_6 : R_2\ (fct\ l_2\ i_2)\ (fct\ l_3\ i_3) \quad H_7 : iperm\_ind'_{R_2}\ (remEl\ l_2\ i_2)\ (remEl\ l_3\ i_3)$$

On instancie l'existencielle avec i_3 et on doit montrer que :

(a) $R_3\ (fct\ l_1\ i_1)\ (fct\ l_3\ i_3)$: c'est-à-dire $\exists t_2, R_1\ (fct\ l_1\ i_1)\ t_2 \wedge R_2\ t_2\ (fct\ l_3\ i_3)$. On instancie t_2 par $fct\ l_2\ i_2$ et on termine la preuve avec H_4 et H_6.

(b) $iperm\_ind'_{R_3}\ (remEl\ l_1\ i_1)\ (remEl\ l_3\ i_3)$: on le prouve grâce à H_5 appliqué à H_7.

□

A.1.2.11 Preuve du Lemme 4.32

Rappel de l'énoncé.

$\forall l_1\ l_2\ i_1\ i_2,\ iperm\_ind_{R_{eq}}\ l_1\ l_2 \wedge R_{eq}\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \Rightarrow iperm\_ind_{R_{eq}}\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)$

Démonstration. Soient H et H_0 les deux hypothèses :

$$H : iperm\_ind_{R_{eq}}\ l_1\ l_2 \quad \text{et} \quad H_0 : R_{eq}\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)$$

On raisonne par induction sur H (en utilisant la Définition 4.17).

[Cas de base] On a l'hypothèse suivante : $H_1 : lg\ l_1 = lg\ l_2 = 0$. Et i_1 est de type $Fin\ (lg\ l_1)$ c'est-à-dire d'après H_1, $Fin\ 0$ qui est vide.

[Cas inductif] Nous avons, grâce à l'induction, les hypothèses suivantes :

$$i'_1 : Fin\ (lg\ l_1) \qquad i'_2 : Fin\ (lg\ l_2) \qquad H_1 : R_{eq}\ (fct\ l_1\ i'_1)\ (fct\ l_2\ i'_2)$$
$$H_2 : iperm\_ind_{R_{eq}}\ (remEl\ l_1\ i'_1)\ (remEl\ l_2\ i'_2)$$

L'hypothèse d'induction est la suivante :

$$IH : \forall i_1\ i_2, R_{eq}\ (fct\ (remEl\ l_1\ i'_1)\ i_1)\ (fct\ (remEl\ l_2\ i'_2)\ i_2) \Rightarrow iperm\_ind_{R_{eq}}\ (remEl\ (remEl\ l_1\ i'_1)\ i_1)\ (remEl\ (remEl\ l_2\ i'_2)\ i_2)$$

Soient n_1, n_2, ln_1, ln_2 tels que $l_1 = \langle n_1, ln_1 \rangle$ et $l_2 = \langle n_2, ln_2 \rangle$.
On veut dans un premier temps prouver que $n_1 = n_2$ afin de se passer de n_2 et de pouvoir faire une analyse de cas sur n_1. On a :

$$\begin{array}{lll} & lg\ (remEl\ \langle n_1, ln_1 \rangle\ i'_1) = lg\ (remEl\ \langle n_2, ln_2 \rangle\ i'_2) & \text{(d'après Lemme 4.17)} \\ \Leftrightarrow & lg\ (remEl\ \langle n_1, ln_1 \rangle\ i'_1) + 1 = lg\ (remEl\ \langle n_2, ln_2 \rangle\ i'_2) + 1 & \\ \Leftrightarrow & lg\ \langle n_1, ln_1 \rangle = lg\ \langle n_2, ln_2 \rangle & \text{(d'après Propriété 4.1.1)} \\ \Leftrightarrow & n_1 = n_2 & \end{array}$$

On peut donc se passer de n_2. Les hypothèses s'écrivent maintenant :

$$\begin{array}{rl} & i'_1\ i'_2 : Fin\ n_1 \qquad\qquad H_1 : R_{eq}\ (fct\ \langle n_1, ln_1\rangle\ i'_1)\ (fct\ \langle n_1, ln_2\rangle\ i'_2) \\ & H_2 : iperm\_ind_{R_{eq}}\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ (remEl\ \langle n_1, ln_2\rangle\ i'_2) \\ IH : & \forall i_1\ i_2, R_{eq}\ (fct\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ i_1)\ (fct\ (remEl\ \langle n_1, ln_2\rangle\ i'_2)\ i_2) \\ & \Rightarrow iperm\_ind_{R_{eq}}\ (remEl\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ i_1)\ (remEl\ (remEl\ \langle n_1, ln_2\rangle\ i'_2)\ i_2) \end{array}$$

et on veut donc prouver que :

$$iperm\_ind_{R_{eq}}\ (remEl\ \langle n_1, ln_1\rangle\ i_1)\ (remEl\ \langle n_1, ln_2\rangle\ i_2)$$

Nous allons maintenant raisonner par analyse sur n_1 :

[Cas 0**]** Dans ce cas, i'_1 est de type $Fin\ 0$ qui est vide.

[Cas $n_1 + 1$**]** Ici, nous allons comparer les valeurs de i_1 et i'_1 ainsi que celles de i_2 et i'_2. Nous avons 4 cas différents :

[Cas $H_3 : i_1 =_{Fin} i'_1$ **et** $H_4 : i_2 =_{Fin} i'_2$**]** Dans ce cas, tout se passe bien : les indices que l'on veut retirer et ceux que l'on a effectivement retirés dans les hypothèses sont les mêmes. En réécrivant H_3 et H_4, on doit maintenant prouver que $iperm\_ind_{R_{eq}}\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ (remEl\ \langle n_1, ln_2\rangle\ i'_2)$, ce qui est l'hypothèse H_2.

[Cas $H_3 : i_1 =_{Fin} i'_1$ **et** $H_4 : i_2 \neq_{Fin} i'_2$**]** Ici, les indices coïncident pour la première *ilist* mais pas pour la seconde. En réécrivant H_3, on doit maintenant prouver que $iperm\_ind_{R_{eq}}\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ (remEl\ \langle n_1, ln_2\rangle\ i_2)$. Ce que nous voulons faire ici, c'est essayer d'obtenir les mêmes *ilist* en hypothèses et dans notre but en utilisant le Lemme 4.16. En ouvrant H_2 avec la Définition 4.19, on obtient la nouvelle hypothèse suivante pour H_2 :

$$\begin{array}{rl} H_2 : & \forall ii_2\ \exists ii_1, R_{eq}\ (fct\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ ii_1)\ (fct\ (remEl\ \langle n_1, ln_2\rangle\ i'_2)\ ii_2) \wedge \\ & iperm\_ind_{R_{eq}}\ (remEl\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ ii_1)\ (remEl\ (remEl\ \langle n_1, ln_2\rangle\ i'_2)\ ii_2) \end{array}$$

On l'utilise avec l'indice correspondant à i_2 dans $remEl\ \langle n_1, ln_2\rangle\ i'_2$: $indexInRemEl\ i'_2\ i_2\ (sym\ H_4)$. Cet élément ayant pour type $Fin\ n$, il faut en plus le convertir au bon type en utilisant la Propriété 4.1.1, on l'appellera ii_2. Cela nous donne les nouvelles hypothèses suivantes :

$$\begin{array}{rl} i_1 : & Fin\ (lg\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)) \\ H_5 : & R_{eq}\ (fct\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ i_1)\ (fct\ (remEl\ \langle n_1, ln_2\rangle\ i'_2)\ ii_2) \\ H_6 : & iperm\_ind_{R_{eq}}\ (remEl\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ i_1)\ (remEl\ (remEl\ \langle n_1, ln_2\rangle\ i'_2)\ ii_2) \end{array}$$

D'après la Définition 4.17, il nous suffit de prouver que :

$$\begin{array}{l} \exists i''_1\ i''_2, R_{eq}\ (fct\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ i''_1)\ (fct\ (remEl\ \langle n_1, ln_2\rangle\ i_2)\ i''_2) \\ \quad \wedge iperm\_ind_{R_{eq}}\ (remEl\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ i''_1)\ (remEl\ (remEl\ \langle n_1, ln_2\rangle\ i_2)\ i''_2) \end{array}$$

Pour utiliser les hypothèses, nous allons donc prendre $i''_1 = i_1$ et l'indice correspondant à i'_2 dans $remEl\ \langle n_1, ln_2\rangle\ i_2$ (pour cela, on utilise aussi *indexInRemEl* et on appelle ce nouvel indice ii'_2). Nous avons donc maintenant deux choses à prouver :

1. $R_{eq}\ (fct\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ i_1)\ (fct\ (remEl\ \langle n_1, ln_2\rangle\ i_2)\ ii'_2)$: pour prouver cela on utilise H_5, le Lemme 4.11 à propos de *indexInRemEl*, H_1 et H_0.

2. $iperm\_ind_{R_{eq}}\ (remEl\ (remEl\ \langle n_1, ln_1\rangle\ i'_1)\ i_1)\ (remEl\ (remEl\ \langle n_1, ln_2\rangle\ i_2)\ ii'_2)$: pour cette preuve, on utilise les Lemmes 4.20 et 4.16 et l'hypothèse H_6.

[Cas $H_3 : i_1 \neq_{Fin} i'_1$ **et** $H_4 : i_2 =_{Fin} i'_2$**]** La preuve ici est tout à fait semblable à celle du cas précédent, et nous ne la détaillons donc pas ici.

[Cas $H_3 : i_1 \neq_{Fin} i'_1$ **et** $H_4 : i_2 \neq_{Fin} i'_2$**]** Ici, nous sommes dans le cas "le plus général" où les indices sont tous différents. Nous allons donc prendre l'indice correspondant à i'_1 (resp. i'_2) dans $remEl\ \langle n_1, ln_1\rangle\ i_1$ (resp. $remEl\ \langle n_1, ln_2\rangle\ i_2$) en utilisant *indexInRemEl* et nous l'appellerons ii'_1 (resp. ii'_2). Nous devons donc ici aussi prouver deux choses :

1. $R_{eq}\ (fct\ (remEl\ \langle n_1, ln_1\rangle\ i_1)\ ii'_1)\ (fct\ (remEl\ \langle n_1, ln_2\rangle\ i_2)\ ii'_2)$: cela se prouve directement avec le Lemme 4.11 et H_1.
2. $iperm\_ind_{R_{eq}}\ (remEl\ (remEl\ \langle n_1, ln_1\rangle\ i_1)\ ii'_1)\ (remEl\ (remEl\ \langle n_1, ln_2\rangle\ i_2)\ ii'_2)$: ici, on utilise les Lemmes 4.19, 4.20, 4.16 et 4.11, l'hypothèse d'induction et H_0.

□

A.1.2.12 Preuve du Lemme 4.33

Rappel de l'énoncé. $Dec\ R \Rightarrow \forall n\ t\ ln,\ (\exists i, R\ t\ (ln\ i)) \vee \neg(\exists i, R\ t\ (ln\ i)))$

Démonstration. On raisonne par induction sur n.

[Cas 0**]** Étant donné que $Fin\ 0$ est vide, la preuve de $\neg(\exists i, R\ t\ (ln\ i))$ est triviale.

[Cas $n+1$**]** L'hypothèse d'induction est

$$IH : \forall ln : ilistn\ T\ n,\ (\exists i, R\ t\ (ln\ i)) \vee \neg(\exists i, R\ t\ (ln\ i)))$$

Nous voulons ici décider s'il existe ou non un élément de ln qui est équivalent à t. Pour cela, nous allons d'abord comparer t et $ln\ (first\ n)$ (grâce à l'hypothèse $Dec\ R$) puis nous utiliserons, si besoin, l'hypothèse d'induction avec $ln \circ succ$.

[Cas $H_1 : R\ t\ (ln\ (first\ n))$**]** Ici, on a déjà trouvé un élément de ln qui est équivalent à t : c'est $ln\ (first\ n)$. On peut donc directement prouver que $\exists i, R\ t\ (ln\ i)$.

[Cas $H_1 : \neg\ (R\ t\ (ln\ (first\ n)))$**]** Nous avons ici éliminé le premier élément de ln comme possiblement équivalent à t. Pour décider s'il en existe un, nous devons regarder dans le reste de ln, en utilisant IH. Nous avons encore deux cas :

[Cas $H_2 : \exists i, R\ t\ (ln\ (succ\ i))$**]** Soit donc i tel que $R\ t\ (ln\ (succ\ i))$ (l'hypothèse H_2 nous assure qu'il existe). On a donc un élément équivalent à t, c'est $ln\ (succ\ i)$. On peut donc directement montrer que $\exists i, R\ t\ (ln\ i)$.

[Cas $H_2 : \neg(\exists i, R\ t\ (ln\ (succ\ i)))$**]** Ici, on veut montrer que $\neg(\exists i, R\ t\ (ln\ i)))$. Pour cela, on va raisonner par l'absurde. Supposons que $\exists i, R\ t\ (ln\ i)$ et montrons que nous arrivons à une contradiction. Soit donc i tel que $H_3 : R\ t\ (ln\ i)$. On a deux possibilités pour i :

[Cas $i = first\ n$**]** On a donc, d'après H_3, $R\ t\ (ln\ (first\ n))$, ce qui est en contradiction avec H_1.

[Cas $i = succ\ i'$**]** On a donc, d'après H_3, $R\ t\ (ln\ (succ\ i'))$, ce qui est en contradiction avec H_2.

□

A.1.2.13 Preuve du Lemme 4.38

Rappel de l'énoncé.

$$\forall l_1\ l_2\ s\ (h_1\ h_2 : lg\ l_1 = lg\ l_2), iperm\_ind\_skel_R\ l_1\ l_2\ h_1\ s \Rightarrow iperm\_ind\_skel_R\ l_1\ l_2\ h_2\ s$$

Démonstration. Soit $H : iperm\_ind\_skel_R\ l_1\ l_2\ h_1\ s$. On raisonne par induction sur H.

[Cas de base] On a comme nouvelle hypothèse : $H_3 : lg\ l_1 = 0$. On applique simplement le premier cas de la Définition 4.24 avec H_3.

[Cas inductif] On a comme nouvelles hypothèses (d'intérêt pour nous) :

$$i_1 : Fin\ (lg\ l_1) \quad i_2 : Fin\ (lg\ l_2) \quad s' : skel\_type\ (remEl\ l_1\ i_1)$$
$$H_3 : R\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \quad H_4 : s = skel\_type\_aux\ l_1\ l_2\ h_1\ i_1\ i_2\ s'$$

Et l'hypothèse d'induction est : $IH : \forall H, iperm\_ind\_skel_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\ H\ s'$.

On applique le deuxième cas de la Définition 4.24 avec i_1, i_2, s' et H_3. On doit maintenant prouver que :

1. $s = skel\_type\_aux\ l_1\ l_2\ h_2\ i_1\ i_2\ s'$: on prouve cela avec H_4 et le Lemme 4.37.
2. $iperm\_ind\_skel_R\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\ H\ s'$, avec H déduit du Lemme 4.35 à partir de h_2 : on peut directement utiliser IH.

□

A.1.2.14 Preuve du Lemme 4.40

Rappel de l'énoncé.

$$\forall R_1\ R_2\ l_1\ l_2\ H_{lg}\ s, R_1 \subseteq R_2\ \wedge\ iperm\_ind\_skel_{R_1}\ l_1\ l_2\ H_{lg}\ s \Rightarrow iperm\_ind\_skel_{R_2}\ l_1\ l_2\ H_{lg}\ s$$

Démonstration. Soient H_1 et H_2 les hypothèses :

$$H_1 : R_1 \subseteq R_2 \qquad \text{et} \quad H_2 :\ iperm\_ind\_skel_{R_1}\ l_1\ l_2\ H_{lg}\ s$$

On raisonne par induction sur H_2 :

[Cas de base] On a comme nouvelle hypothèse : $H_3 : lg\ l_1 = 0$. On le prouve grâce à la Définition 4.24 et H_3.

[Cas inductif] Nous avons comme nouvelles hypothèses :

$$i_1 : Fin\ (lg\ l_1) \qquad i_2 : Fin\ (lg\ l_2) \qquad s' : skel\_type\ (lg\ (remEl\ l_1\ i_1))$$
$$H_3 : R_1\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2) \quad H_4 : s = skel\_type\_aux\ l_1\ l_2\ H_{lg}\ i_1\ i_2\ s'$$
$$H_5 : iperm\_ind\_skel_{R_1}\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\ H'_{lg}\ s'$$

avec H'_{lg} issu de H_{lg} et du Lemme 4.35. L'hypothèse d'induction est :

$$IH : iperm\_ind\_skel_{R_2}\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\ H'_{lg}\ s'$$

Selon la Définition 4.24 il nous suffit de prouver que :

1. $R_2\ (fct\ l_1\ i_1)\ (fct\ l_2\ i_2)$: pour prouver ceci on utilise H_1 avec l'hypothèse H_3.
2. $s = skel\_type\_aux\ l_1\ l_2\ H_{lg}\ i_1\ i_2\ s'$: c'est l'hypothèse H_4.
3. $iperm\_ind\_skel_{R_1}\ (remEl\ l_1\ i_1)\ (remEl\ l_2\ i_2)\ H'_{lg}\ s'$: c'est l'hypothèse H_5.

□

A.1.2.15 Preuve du Lemme 4.41

Rappel de l'énoncé. *Pour l_1, l_2, H_{lg} fixés et un squelette s, $iperm\_ind\_skel_R\ l_1\ l_2\ H_{lg}\ s$ commute avec des intersections arbitraires d'un ensemble de relations R. En particulier, si pour tout n, $iperm\_ind\_skel_{R_n}\ l_1\ l_2\ H_{lg}\ s$ alors $iperm\_ind\_skel_{\cap_n R_n}\ l_1\ l_2\ H_{lg}\ s$. Formellement, on exprime cela ainsi (avec R une famille de relations, de type $R : I \to$ relation T et $R'\ t_1\ t_2 \Leftrightarrow \forall i, R\ i\ t_1\ t_2$:*

$$\forall l_1\ l_2\ (H : lg\ l_1 = lg\ l_2)\ s\ I, (\forall i, iperm\_ind\_skel_{R\ i}\ l_1\ l_2\ H\ s) \Rightarrow\ iperm\_ind\_skel_{R'}\ l_1\ l_2\ H\ s$$

Démonstration. Soient n_1, ln_1, n_2, ln_2 tels que $l_1 = \langle n_1, ln_1\rangle$ et $l_2 = \langle n_2, ln_2\rangle$. On peut réécrire H et se passer de n_2. On va raisonner par induction sur n_1

[Cas 0**]** On peut appliquer directement la Définition 4.24.

[Cas $n_1 + 1$**]** Soit H l'hypothèse $\forall i, iperm\_ind\_skel_{R\ i}\ l_1\ l_2\ H\ s$. L'hypothèse d'induction est :

$$\begin{array}{ll} IH : & \forall ln_1\ ln_2\ s, (\forall i, iperm\_ind\_skel_{R\ i}\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle\ H_1\ s) \\ \Rightarrow & iperm\_ind\_skel_{R'}\ \langle n_1, ln_1\rangle\ \langle n_2, ln_2\rangle\ H_1\ s \end{array} \text{ avec } H_1 : n_1 = n_1$$

On sait aussi que s est de la forme (i_1, i_2, s'). Selon la Définition 4.24 utilisée avec i_1, i_2, et $convSkel_{H_2}\ s'$ (avec H_2 : $remEl\ \langle n_1 + 1, ln_1\rangle\ i_1 = n_1$ déduit de la Propriété 4.1.1), on doit maintenant montrer que :

1. $\forall i, R\ i\ (ln_1\ i_1)\ (ln_2\ i_2)$
   On applique H à i et on obtient H_4 : $iperm\_ind\_skel_{R\ i}\ l_1\ l_2\ H\ s$. Pour H_4, il y a deux cas : les deux cas de la Définition 4.24. Mais le premier est impossible (on aurait alors $n_1 + 1 = 0$). Le deuxième cas nous donne de nouvelles hypothèses. On a i'_1, i'_2 et s'' tels que (on ne donne que les hypothèses qui nous intéressent) :

   $$H_5 : R\ (ln_1\ i'_1)\ (ln_2\ i'_2) \quad H_6 : (i_1, i_2, s') = (i'_1, i'_2, convSkel_{H'_2}\ s'') \text{ avec } H'_2 \text{ le pendant de } H_2 \text{ pour } i'_1$$

   On en déduit donc que $i_1 = i'_1$ et $i_2 = i'_2$. On veut donc en réalité montrer que : $R\ i\ (ln_1\ i'_1)\ (ln_2\ i'_2)$. Ce qui est vrai d'après H_5.
2. $(i_1, i_2, s') = skel\_type\_aux\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle\ H_3\ i_1\ i_2\ (convSkel_{H_2}\ s')$ (avec $H_3 : n_1 + 1 = n_1 + 1$)
   Par définition,

   $$skel\_type\_aux\ \langle n_1 + 1, ln_1\rangle\ \langle n_1 + 1, ln_2\rangle\ H_3\ i_1\ i_2\ (convSkel_{H_2}\ s') = (i_1, i_2, convSkel_{sym\ \ H_2}\ (convSkel_{H_2}\ s'))$$

   On doit donc seulement montrer que $s' = convSkel_{sym\ \ H_2}\ (convSkel_{H_2}\ s'))$. Ceci se fait aisément (en énonçant le lemme dans le cas général). On ne développe pas ici.
3. $iperm\_ind\_skel_{R'}\ (remEl\ \langle n_1 + 1, ln_1\rangle\ i_1)\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i_2)\ H_3\ (convSkel_{H_2}\ s')$ (avec H_3 : $lg\ (remEl\ \langle n_1 + 1, ln_1\rangle\ i_1) = lg\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i_2)$ déduit du Lemme 4.35
   Comme on sait que $n_1 = lg(remEl\ \langle n_1 + 1, ln_1\rangle\ i_1)$, on peut appliquer IH. On doit maintenant prouver que :

   $$\forall i, iperm\_ind\_skel_{R\ i}\ (remEl\ \langle n_1 + 1, ln_1\rangle\ i_1)\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i_2)\ H_1\ (convSkel_{H_2}\ s')$$

   On applique H à i et on obtient H_4 : $iperm\_ind\_skel_{R\ i}\ l_1\ l_2\ H\ s$. Pour H_4, il y a deux cas : les deux cas de la Définition 4.24. Mais le premier est impossible (on aurait

alors $n_1 + 1 = 0$). Le deuxième cas nous donne de nouvelles hypothèses. On a i'_1, i'_2 et s'' tels que (on ne donne que les hypothèses qui nous intéressent, et on simplifie un peu) :

$$H_5 : (i_1, i_2, s') = (i'_1, i'_2, convSkel_{H'_2}\ s'')$$
avec $H'_2 : remEl\ \langle n_1 + 1, ln_1\rangle\ i'_1 = n_1$ déduit de la Propriété 4.1.1
$$H_6 : iperm\_ind\_skel_{R\ i}\ \ (remEl\ \langle n_1 + 1, ln_1\rangle\ i'_1)\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i'_2)\ H_3\ s''$$

On en déduit que $i_1 = i'_1$ et que $i_2 = i'_2$. On utilise le Lemme 4.38 et on doit maintenant montrer que :

$$iperm\_ind\_skel_{R\ i}\ \ (remEl\ \langle n_1 + 1, ln_1\rangle\ i'_1)\ (remEl\ \langle n_1 + 1, ln_2\rangle\ i'_2)\ H_3\ (convSkel_{H_2}\ s')$$

Il nous suffit donc de montrer que $convSkel_{H_2}\ s' = s''$ et on pourra terminer la preuve avec H_6. En utilisant H_5, cela revient à montrer que $convSkel_{H_2}(convSkel_{H'_2}\ s'') = s''$ et on finit la preuve de la même façon que dans 2.

□

A.1.2.16 Preuve du Lemme 4.42

Rappel de l'énoncé. *R réflexive* $\Rightarrow \forall l,\ iperm\_bij_R\ l\ l$

Démonstration. On veut donc montrer que :

$$\exists f\ g, bij\ f\ g\ \wedge\ (\forall i, R\ (fct\ l\ i)\ (fct\ l\ (f\ i)))$$

On va donc naturellement prendre $f = g = \lambda i.i$. En ouvrant directement *bij* on doit donc maintenant montrer trois choses :

1. $\forall i, f\ (g\ i) = i$: on a $f\ (g\ i) = f\ i = i$
2. $\forall i, g\ (f\ i) = i$: même chose
3. $\forall i, R\ (fct\ l\ i)\ (fct\ l\ (f\ i))$
   On a $R\ (fct\ l\ i)\ (fct\ l\ (f\ i)) \Leftrightarrow R\ (fct\ l\ i)\ (fct\ l\ i)$ ce qui est vrai par hypothèse.

□

A.1.2.17 Preuve du Lemme 4.43

Rappel de l'énoncé. *R symétrique* $\Rightarrow (\forall l_1\ l_2,\ iperm\_bij_R\ l_1\ l_2 \rightarrow iperm\_bij_R\ l_2\ l_1)$

Démonstration. Soient H_1 : *R* symétrique et $H_2 : iperm\_bij_R\ l_1\ l_2$. Grâce à H_1 on obtient f et g tels que :

$$H_3 : bij\ f\ g \quad \text{et} \quad H_4 : \forall i, R\ (fct\ l_1\ i)\ (fct\ l_2\ (f\ i))$$

Et on veut prouver que $\exists f'\ g', bij\ f'\ g'\ \wedge\ (\forall i, R\ (fct\ l_2\ i)\ (fct\ l_1\ (f'\ i)))$. On prend $f' := g$ et $g' := f$, et on doit donc prouver deux choses :

1. *bij g f* : on utilise directement le Lemme 3.11 avec l'hypothèse H_3.

2. $\forall i, R\ (fct\ l_2\ i)\ (fct\ l_1\ (g\ i)))$ Soit i. On instancie l'hypothèse H_4 avec $g\ i$ et cela nous donne l'hypothèse suivante : $H_5 : R\ (fct\ l_1\ (g\ i))\ (fct\ l_2\ (f\ (g\ i)))$. L'hypothèse H_3 nous donne les deux nouvelles hypothèses suivantes :

$$H_3^1 : \forall i, g\ (f\ i) = i \quad \text{et} \quad H_3^2 : \forall i, f\ (g\ i) = i$$

On utilise H_3^2 dans H_5 et cela nous donne : $H_5 : R\ (fct\ l_1\ (g\ i))\ (fct\ l_2\ i))$. En utilisant la symétrie de R (H_1) avec H_5, on obtient ce qu'on voulait prouver :

$$R\ (fct\ l_2\ i))\ (fct\ l_1\ (g\ i))$$

□

A.1.2.18 Preuve du Lemme 4.44

Rappel de l'énoncé.

$$R\ transitive \Rightarrow (\forall l_1\ l_2\ l_3,\ iperm\_bij_R\ l_1\ l_2 \wedge iperm\_bij_R\ l_2\ l_3 \rightarrow iperm\_bij_R\ l_1\ l_3)$$

Démonstration. Soient :

$$H_1 : R \text{ transitive} \quad H_2 : iperm\_bij_R\ l_1\ l_2 \quad H_3 : iperm\_bij_R\ l_2\ l_3$$

Grâce à H_2 et H_3 on obtient f_1, g_1, f_2, et g_2 tels que :

$$H_4 : bij\ f_1\ g_1 \quad H_6 : \forall i, R\ (fct\ l_1\ i)\ (fct\ l_2\ (f_1\ i)))$$
$$H_5 : bij\ f_2\ g_2 \quad H_7 : \forall i, R\ (fct\ l_2\ i)\ (fct\ l_3\ (f_2\ i)))$$

On veut prouver que

$$\exists f\ g, bij\ f\ g\ \wedge\ (\forall i, R\ (fct\ l_1\ i)\ (fct\ l_3\ (f\ i)))$$

On prend $f := f_2 \circ f_1$ et $g := g_1 \circ g_2$. On doit maintenant prouver que :

1. $bij\ f\ g$: on applique le Lemme 3.12 avec H_4 et H_5.
2. $\forall i, R\ (fct\ l_1\ i)\ (fct\ l_3\ (f\ i)))$ On utilise l'hypothèse de transitivité de R (H_1) avec $fct\ l_2\ (f_1\ i)$ et on doit prouver que :
   (a) $R\ (fct\ l_1\ i)\ (fct\ l_2\ (f_1\ i))$: on utilise directement H_6.
   (b) $R\ (fct\ l_2\ (f_1\ i))\ (fct\ l_3\ (f_2\ (f_1\ i)))$: on utilise directement H_7.

□

A.1.2.19 Preuve du Lemme 4.46

Rappel de l'énoncé. $\forall l_1 l_2, iperm\_bij_R\ l_1\ l_2 \Rightarrow lg\ l_1 = lg\ l_2$

Démonstration. Soit $H_1 : iperm\_bij_R\ l_1\ l_2$. Grâce à H_1, on obtient $f : Fin\ (lg\ l_1) \rightarrow Fin\ (lg\ l_2)$ et $g : Fin\ (lg\ l_2) \rightarrow Fin\ (lg\ l_1)$ telles que

$$H_2 : bij\ f\ g \quad \text{et} \quad H_3 : \forall i, R\ (fct\ l_1\ i)\ (fct\ l_2\ (f\ i)))$$

On utilise le Lemme 3.14 avec H_2 et on obtient $lg\ l_1 = lg\ l_2$. □

A.1.2.20 Preuve du Lemme 4.47

Rappel de l'énoncé. $\forall ln_1\ ln_2, iperm\_bij_R\ \langle 0, ln_1\rangle\ \langle 0, ln_2\rangle$

Démonstration. On prend $f = g = \lambda i.i$. On doit montrer :

1. $bij\ f\ g$: on l'a vu dans la preuve du Lemme 4.42.
2. $\forall i : Fin\ 0, R\ (fct\ l_1\ i)\ (fct\ l_2\ (f\ i))$: $Fin\ 0$ étant vide, cela est toujours vrai.

□

A.2 Une représentation coinductive des graphes

A.2.1 Des graphes ordonnés, enracinés, connexes

A.2.1.1 Preuve du Lemme 5.7

Rappel de l'énoncé. $\forall g_1\ g_1'\ g_2, Geq_{R_{eq}}\ g_1\ g_1' \wedge GinG_{R_{eq}}\ g_1\ g_2 \Rightarrow GinG_{R_{eq}}\ g_1'\ g_2$

Démonstration. Soient $H_1 : Geq_{R_{eq}}\ g_1\ g_1'$ et $H_2 : GinG_{R_{eq}}\ g_1\ g_2$. On raisonne par induction sur H_2.

[Cas de base (direct)] On a i tel que $H_3 : Geq_{R_{eq}}\ g_1\ (fct\ (sons\ g_2)\ i)$. Comme R_{eq} est une relation d'équivalence, on peut applique le Lemme 5.4 sur H_1 ce qui nous donne H_1' : $Geq_{R_{eq}}\ g_1'\ g_1$. On peut maintenant appliquer le Lemme 5.5 à H_1' et H_3 ce qui nous donne $H_4 : Geq_{R_{eq}}\ g_1'\ (fct\ (sons\ g_2)\ i)$. On prouve notre but en appliquant la règle dir de la Définition 5.4 avec H_4.

[Cas inductif (indirect)] On a i tel que $H_3 : GinG_{R_{eq}}\ g_1\ (fct\ (sons\ g_2)\ i)$ et l'hypothèse d'induction est $IH : \forall g_1', Geq_{R_{eq}}\ g_1\ g_1' \Rightarrow GinG_{R_{eq}}\ g_1'\ (fct\ (sons\ g_2)\ i)$. Pour prouver notre but, on applique la règle indir de la Définition 5.4 avec IH appliqué à H_1.

□

A.2.1.2 Preuve du Lemme 5.9

Rappel de l'énoncé. $\forall g_1\ g_2, GinG_R\ g_1\ g_2 \Rightarrow \forall i_1, GinG_R\ (fct\ (sons\ g_1)\ i_1)\ g_2$

Démonstration. On raisonne par induction sur $H_1 : GinG_R\ g_1\ g_2$.

[Cas de base (direct)] On a i_2 tel que $H_2 : Geq_R\ g_1\ (fct\ (sons\ g_2)\ i_2)$. On applique une première fois la règle indir de la Définition 5.4 (les fils de g_1 ne sont pas directement dans g_2) avec i_2. On doit prouver que :

$$GinG_R\ (fct\ (sons\ g_1)\ i_1)\ (fct\ (sons\ g_2)\ i_2)$$

H_2 nous donne une nouvelle hypothèse (d'intérêt dans ce cas) : H_3 : $ilist\_rel_{Geq_R}\ (sons\ g_1)\ (sons\ (fct\ (sons\ g_2)\ i_2))$. Elle nous donne elle même, en utilisant la Définition 3.11 deux nouvelles hypothèses :

$$H_4 : lg(sons\ g_1) = lg(sons\ (fct\ (sons\ g_2)\ i_2))$$
$$H_5 : \forall i, Geq_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ (fct\ (sons\ g_2)\ i_2))\ (conv_{H_4}\ i))$$

On peut maintenant appliquer la règle dir de la Définition 5.4 avec $conv_{H_4}\ i_1$. On doit prouver que :

$$Geq_R\ (fct\ (sons\ g_1)\ i_1)\ (fct\ (sons\ (fct\ (sons\ g_2)\ i_2))\ (conv_{H_4}\ i_1)))$$

Ce qu'on prouve avec H_5.

[Cas inductif (indirect)] On a i_2 tel que $H_2 : GinG_R\ g_1\ (fct\ (sons\ g_2)\ i_2)$. Et l'hypothèse d'induction est : $IH : GinG_R\ (fct\ (sons\ g_1)\ i_1)\ (fct\ (sons\ g_2)\ i_2)$. On applique une première fois la règle indir de la Définition 5.4 (les fils de g_1 ne sont pas directement dans g_2) avec i_2. On doit prouver que :

$$GinG_R\ (fct\ (sons\ g_1)\ i_1)\ (fct\ (sons\ g_2)\ i_2)$$

Ce qui est exactement IH.

□

A.2.1.3 Preuve du Lemme 5.10

Rappel de l'énoncé. $\forall g_1\ g_2\ g_3, GinG_{R_{eq}}\ g_1\ g_2 \wedge GinG_{R_{eq}}\ g_2\ g_3 \Rightarrow GinG_{R_{eq}}\ g_1\ g_3$

Démonstration. Soient $H_1 : GinG_{R_{eq}}\ g_1\ g_2$ et $H_2 : GinG_{R_{eq}}\ g_2\ g_3$. On raisonne par induction sur H_1.

[Cas de base (direct)] On a i tel que $H_3 : Geq_{R_{eq}}\ g_1\ (fct\ (sons\ g_2)\ i)$. On veut prouver que : $GinG_{R_{eq}}\ g_1\ g_3$. On applique le Lemme 5.7 avec le symétrique de H_3 et il nous reste à montrer que : $GinG_{R_{eq}}\ (fct\ (sons\ g_2)\ i)\ g_3$. Ce qu'on prouve avec le Lemme 5.9 et H_2.

[Cas inductif (indirect)] On a i tel que $H_3 : GinG_{R_{eq}}\ g_1\ (fct\ (sons\ g_2)\ i)$. Et l'hypothèse d'induction est : $IH : GinG_{R_{eq}}\ (fct\ (sons\ g_2)\ i)\ g_3 \Rightarrow GinG_{R_{eq}}\ g_1\ g_3$. On applique IH et il nous reste à montrer que : $GinG_{R_{eq}}\ (fct\ (sons\ g_2)\ i)\ g_3$. Ce qu'on fait avec le Lemme 5.9 et H_2.

□

A.2.1.4 Preuve du Lemme 5.11

Rappel de l'énoncé. $\forall g_1\ g_2, GinG_R\ g_1\ g_2 \Rightarrow GinG'_R\ g_1\ g_2$

Démonstration. On raisonne par induction sur $H : GinG_R\ g_1\ g_2$.

[Cas de base (direct)] On a i tel que $H_1 : Geq_R\ g_1\ (fct\ (sons\ g_2)\ i)$. On veut prouver que : $GinG'_R\ g_1\ g_2$. On applique la règle indirG de la Définition 5.5 avec i. On doit montrer que : $GinG'_R\ g_1\ (fct\ (sons\ g_2)\ i)$. Ce qu'on prouve avec la règle dirG de la Définition 5.5 et H_1.

[Cas inductif (indirect)] On a i tel que $H_3 : GinG_R\ g_1\ (fct\ (sons\ g_2)\ i)$. Et l'hypothèse d'induction est : $IH : GinG'_R\ g_1\ (fct\ (sons\ g_2)\ i)$. On applique la règle indirG de la Définition 5.5 avec i. On doit montrer que : $GinG'_R\ g_1\ (fct\ (sons\ g_2)\ i)$. Ce qu'on prouve avec IH.

□

A.2.1.5 Preuve du Lemme 5.12

Rappel de l'énoncé. $\forall g, hasCycle_{R_{eq}}\ g \Leftrightarrow hasCycle'_{R_{eq}}\ g$

Démonstration.

[Direction $\Rightarrow$] On raisonne par induction sur $H : hasCycle_{R_{eq}}\ g$.

[Cas de base (hC_dir)] On a $H_1 : isCycle_{R_{eq}}\ g$. On applique la Définition 5.9 avec H_1. Il nous reste à prouver que : $GinG'_{R_{eq}}\ g\ g$, ce qu'on fait avec la règle dirG de la Définition 5.5 et de la réflexivité de *Geq* (Lemme 5.3).

[Cas inductif (hC_indir)] On a i tel que $H_1 : hasCycle_{R_{eq}}\ (fct\ (sons\ g)\ i)$ et l'hypothèse d'induction est $IH : hasCycle'_{R_{eq}}\ (fct\ (sons\ g)\ i)$. On veut montrer que $hasCycle'_{R_{eq}}\ g$. Pour cela, on va utiliser le lemme suivant qui dit que si un des fils d'un nœud contient un cycle selon *hasCycle'*, alors le nœud lui même en contient un :

Lemme A.4. $\forall g\ i, hasCycle'_{R_{eq}}(fct\ (sons\ g)\ i) \Rightarrow hasCycle'_{R_{eq}}\ g$

Démonstration. Soit $H : hasCycle'_{R_{eq}}(fct\ (sons\ g)\ i)$. H nous donne g' tel que

$$H_1 : isCycle_{R_{eq}}\ g' \quad H_2 : GinG'_{R_{eq}}\ g'\ (fct\ (sons\ g)\ i)$$

On veut prouver que $hasCycle'_{R_{eq}}\ g$. On applique la Définition 5.9 avec H_1 et on doit prouver que : $GinG'_{R_{eq}}\ g'\ g$. On utilise la transitivité de *GinG'* avec H_2 et il nous reste à montrer que : $GinG'_{R_{eq}}\ (fct\ (sons\ g)\ i)\ g$. On utilise alors l'équivalent du Lemme 5.9 pour *GinG'* et on doit alors prouver que : $GinG'_{R_{eq}}\ g\ g$, ce qu'on fait avec la règle dirG de la Définition 5.5 et la réflexivité de *Geq*. □

On finit alors la preuve ici (de la direction $\Rightarrow$) en utilisant simplement le Lemme A.4 avec *IH*.

[Direction $\Leftarrow$] $H : hasCycle'_{R_{eq}}\ g$ nous donne g' tel que : $H_1 : isCycle_{R_{eq}}\ g'$ et $H_2 : GinG'_{R_{eq}}\ g'\ g$. On raisonne par induction sur H_2.

[Cas de base (dirG)] On a $H_3 : Geq_{R_{eq}}\ g'\ g$. On applique la règle hC_dir de la Définition 5.8. On doit montrer que $isCycle_{R_{eq}}\ g$. En utilisant les Lemmes 5.7 et 5.8, on peut montrer que *isCycle* est lui aussi un morphisme. On peut donc réécrire l'hypothèse H_3. Il nous reste à montrer que $isCycle_{R_{eq}}\ g'$, ce qu'on fait avec H_1.

[Cas inductif (indirG)] On a i tel que $IH : hasCycle_{R_{eq}}(fct\ (sons\ g)\ i)$ (c'est l'hypothèse d'induction). On doit prouver que $hasCycle_{R_{eq}}\ g$. On applique simplement la règle hC_indir de la Définition 5.8 avec i et IH.

□

A.2.1.6 Preuve du Lemme 5.15

Rappel de l'énoncé. $\forall P\ g\ g', Gall\ P\ g \wedge GinG_R\ g'\ g \Rightarrow P\ g'$

Démonstration. On suppose que P est un morphisme pour *Geq*. Soient $H_1 : Gall\ P\ g$ et $H_2 : GinG_R\ g'\ g$. Grâce à H_1 et à la Définition 5.10 on obtient $H_3 : iall\ (Gall\ P)\ (sons\ g)$. On raisonne par induction sur H_2.

[Cas de base (dir)] On a i tel que $H_4 : Geq_R\ g'\ (fct\ (sons\ g)\ i)$. Comme P est un morphisme pour Geq, grâce à H_4 il nous suffit de prouver que $P(fct\ (sons\ g)\ i)$. H_3 appliqué à i nous donne le résultat attendu.

[Cas inductif (indir)] On a i tel que $H_4 : GinG_R\ g'\ (fct\ (sons\ g)\ i)$ et l'hypothèse d'induction est $IH : iall\ (Gall\ P)\ (sons\ (fct\ (sons\ g)\ i)) \Rightarrow P\ g'$. On applique IH et il nous suffit de montrer que : $iall\ (Gall\ P)\ (sons\ (fct\ (sons\ g)\ i))$. On obtient cela grâce à H_3 appliqué à i, en dépliant $Gall$ une fois de plus.

□

A.2.1.7 Preuve du Lemme 5.16

Rappel de l'énoncé. *On suppose que P est un morphisme pour son argument.*

$$\forall P\ g_1\ g_2, Geq_R\ g_1\ g_2 \wedge Gall\ P\ g_1 \Rightarrow Gall\ P\ g_2$$

Démonstration. On raisonne par coinduction. L'hypothèse de coinduction est

$$CH : \forall g_1\ g_2, Geq_R\ g_1\ g_2 \wedge Gall\ P\ g_1 \Rightarrow Gall\ P\ g_2$$

Soient $H_1 : Geq_R\ g_1\ g_2$ et $H_2 : Gall\ P\ g_1$. H_2 nous donne deux nouvelles hypothèses (grâce à la Définition 5.10), $H_3 : P\ g_1$ et $H_4 : iall\ (Gall\ P)\ (sons\ g_1)$. On applique la Définition 5.10 à notre but et on doit prouver que :

1. $P\ g_2$: on prouve cela en utilisant le fait que P est un morphisme avec H_1 et on finit la preuve avec H_3.
2. $iall\ (Gall\ P)\ (sons\ g_2)$: grâce à la Définition 5.3, H_1 nous donne une nouvelle hypothèse (d'intérêt pour nous) $H_5 : ilist\_rel_{Geq_R}\ (sons\ g_1)\ (sons\ g_2)$. H_5 nous donne à son tour deux nouvelles hypothèses :

   $$H_6 : lg(sons\ g_1) = lg(sons\ g_2) \quad H_7 : \forall i, Geq_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ (conv_{H_6}\ i))$$

   On doit prouver que $\forall i, Gall\ P\ (fct\ (sons\ g_2)\ i)$. On prouve que $i = conv_{H_6}\ (conv_{sym\ H_6}\ i)$ (la preuve est immédiate). Il nous reste donc à prouver que

   $$Gall\ P\ (fct\ (sons\ g_2)\ (conv_{H_6}\ (conv_{sym\ H_6}\ i)))$$

   Pour cela, on applique CH avec H_7 et H_4.

□

A.2.1.8 Preuve du Lemme 5.19

Rappel de l'énoncé. $\forall l\ l'\ g, l \subseteq l' \wedge element\_of_R\ l\ g \Rightarrow element\_of_R\ l'\ g$

Démonstration. Soient $H_1 : l \subseteq l'$ et $H_2 : element\_of_R\ l\ g$. H_2 nous donne g' tel que $H_3 : g' \in l$ et $H_4 : Geq_R\ g\ g'$. On veut prouver que $\exists y,\ y \in l' \ \wedge\ Geq_R\ g\ y$. On prend $y := g'$. On doit prouver que

1. $g' \in l'$: on applique la définition de $\subseteq$ (donnée Remarque 5.6) avec H_3.
2. $Geq_R\ g\ g'$: c'est H_4.

□

A.2.1.9 Preuve du Lemme 5.20

Rappel de l'énoncé. $\forall P\ P'\ g, (\forall g, P\ g \Rightarrow P'\ g) \wedge Gall\ P\ g \Rightarrow Gall\ P'\ g$

Démonstration. On raisonne par coinduction. Soient $H_1 : \forall g, P\ g \Rightarrow P'\ g'$ et $H_2 : Gall\ P\ g$. L'hypothèse de coinduction est $CH : \forall g, Gall\ P\ g \Rightarrow Gall\ P'\ g$. Grâce à la Définition 5.10 H_2 nous donne deux nouvelles hypothèses : $H_3 : P\ g$ et $H_4 : iall\ (Gall\ P)\ (sons\ g)$. On applique la Définition 5.10 à notre but et il nous reste à prouver que :

1. $P'\ g$: on applique H_1 et H_3.
2. $iall\ (GallP')\ (sons\ g)$: c'est-à-dire $\forall i, Gall\ P'\ (fct\ (sons\ g)\ i)$. On applique CH avec H_4 appliqué à i.

□

A.2.1.10 Preuve du Lemme 5.21

Rappel de l'énoncé. $\forall l\ l'\ g, l \subseteq l' \wedge Gall\ (element\_of_R\ l)\ g \Rightarrow Gall\ (element\_of_R\ l')\ g$

Démonstration. Soit $H_1 : l \subseteq l'$. On veut montrer que :

$$Gall\ (element\_of_R\ l)\ g \Rightarrow Gall\ (element\_of_R\ l')\ g$$

D'après le Lemme 5.20, il nous suffit de prouver que

$$\forall g', element\_of_R\ l\ g' \Rightarrow element\_of_R\ l'\ g'$$

On termine la preuve avec le Lemme 5.19 et H_1. □

A.2.1.11 Preuve du Lemme 5.22

Rappel de l'énoncé. $\forall l, iall\ (G\_finite_{R_{eq}})\ l \Rightarrow \exists l', iall\ (Gall\ (element\_of_{R_{eq}}\ l'))\ l$

Démonstration. Soient $H_1 : iall\ (G\_finite_{R_{eq}})\ l$ et n et ln tels que $l = \langle n, ln\rangle$. On raisonne par induction sur n. Et on veut montrer que : $\exists l', iall\ (Gall\ (element\_of_{R_{eq}}\ l'))\ \langle n, ln\rangle$

[Cas 0**]** Comme ln est vide (elle est de type $ilistn\ (Graph\ T)\ 0$), on prend $lg := []$. On doit montrer que $\forall i : Fin\ (lg\ \langle 0, ln\rangle), Gall\ (element\_of_{R_{eq}}\ [])\ (fct\ \langle 0, ln\rangle\ i)$. Or on a $Fin\ (lg\ \langle 0, ln\rangle) = Fin\ 0$ et il est donc vide. Ce qui termine cette partie de la preuve.

[Cas $n+1$**]** On montre que $H_2 : iall\ (G\_finite_{R_{eq}})\ \langle n, \lambda i.ln(succ\ i)\rangle$ à l'aide de H_1. En appliquant IH à H_2, on obtient lc tel que $H_3 : iall\ (Gall\ (element\_of_{R_{eq}}\ lc))\ \langle n, \lambda i.ln(succ\ i)\rangle$. C'est-à-dire qu'on a obtenu (avec lc) la concaténation des listes de tous les éléments de la "queue" de l. Il nous manque à mettre la liste pour la tête. H_1 appliqué à $first\ n$ nous donne lc' tel que $H_4 : Gall\ (element\_of_{R_{eq}}\ lc')\ (ln\ (first\ n))$.

On prend donc $l' := lc'@lc$. On doit prouver que

$$\forall i, Gall\ (element\_of_{R_{eq}}(lc'@lc))\ (fct\ \langle n+1, ln\rangle\ i)$$

On va maintenant raisonner par analyse de cas sur i :

[Cas $first\ n$**]** Il est immédiat de montrer que $H_5 : lc' \subseteq lc'@lc$. On termine donc la preuve de ce cas avec le Lemme 5.21 avec H_5 et H_4.

[Cas $succ\ i'$**]** On montre cette fois-ci que $H_5 : lc \subseteq lc'@lc$. On utilise le Lemme 5.21 avec H_5. Il nous reste à prouver que : $Gall\ (element\_of_{R_{eq}}\ lc)\ (ln\ (succ\ i'))$. On termine la preuve en appliquant H_3 à i'.

□

A.2.1.12 Preuve du Lemme 5.27

Rappel de l'énoncé. $\forall g\ f, G\_finite_R\ g \Rightarrow \exists m, Gall\ (\lambda x.f\ x \leq m)\ g$

Démonstration. $H_1 : G\_finite_R\ g$, nous donne l telle que $H_2 : Gall\ (element\_of_R\ l)\ g$. On veut prouver que $\exists m, Gall\ (\lambda x.f\ x \leq m)\ g$

Comme on l'a dit, on veut prendre le maximum de $map\ f\ l$. Pour cela, on a créé une fonction *max_list_nat* qui renvoie le maximum d'une liste d'entiers naturels (voir Définition 5.13). On prend donc $m := max\_list\_nat\ (map\ f\ l)$. On doit prouver que : $Gall\ (\lambda x.f\ x \leq max\_list\_nat\ (map\ f\ l))\ g$. On raisonne par coinduction. L'hypothèse de coinduction est

$$CH : \forall g, Gall\ (element\_of\ l)\ g \Rightarrow Gall\ (\lambda x.f\ x \leq max\_list\_nat\ (map\ f\ l))\ g$$

L'hypothèse H_2 nous donne deux nouvelles hypothèses :

$$H_3 : element\_of_R\ l\ g \quad H_4 : iall\ (Gall\ (element\_of_R\ l))\ (sons\ g)$$

H_3 nous donne g' tel que $H_5 : g' \in l$ et $H_6 : Geq_R\ g\ g'$. On applique la Définition 5.10 à notre but et on doit montrer que :

1. $f\ g \leq max\_list\_nat(map\ f\ l)$: comme on a supposé que f est un morphisme pour *Geq* on peut réécrire l'hypothèse H_6 et on doit prouver que : $f\ g' \leq max\_list\_nat(map\ f\ l)$. On applique le Lemme 5.28, et il nous suffit de prouver que $f\ g' \in map\ f\ l$. En appliquant un résultat bien connu sur *map*, il nous suffit en fait de prouver que $g' \in l$, ce qui est H_5.
2. $\forall i, Gall\ (\lambda x.f\ x \leq max\_list\_nat\ (map\ f\ l))\ (fct\ (sons\ g)\ i)$: on applique CH et il nous suffit de prouver que : $Gall\ (element\_of_R\ l)\ (fct\ (sons\ g)\ i)$ ce qu'on fait avec H_4 appliqué à i.

□

A.2.1.13 Preuve du Lemme 5.33

Rappel de l'énoncé. $\forall n, GinG_{eq}\ Infinite\_Graph_{n+1}\ Infinite\_Graph_n$

Démonstration. On applique la règle dir de la Définition 5.4 ($Infinite\_Graph_n$ a pour fils $Infinite\_Graph_{n+1}$). On doit prouver que :

$$Geq_{eq}\ Infinite\_Graph_{n+1}\ (fct\ (sons\ Infinite\_Graph_n)\ (first\ 0))$$

c'est-à-dire $Geq_{eq}\ Infinite\_Graph_{n+1}\ Infinite\_Graph_{n+1}$, ce qu'on montre par réflexivité de *Geq*.

□

A.2.1.14 Preuve du Lemme 5.34

Rappel de l'énoncé. $\forall n\ m, n < m \Rightarrow GinG_{eq}\ Infinite\_Graph_m\ Infinite\_Graph_n$

Démonstration. Soit $H_1 : n < m$. On raisonne par induction sur m.

[Cas 0**]** Ici $H_1 : n < 0$ ce qui est faux.

[Cas $m+1$**]** L'hypothèse d'induction est

$$IH : n < m \Rightarrow GinG_{eq}\ Infinite\_Graph_m\ Infinite\_Graph_n$$

On veut prouver que $GinG_{eq}\ Infinite\_Graph_{m+1}\ Infinite\_Graph_n$. Pour pouvoir utiliser IH on doit comparer plus finement n et m. D'après $H_1 : n < m+1$ on a deux cas :

[Cas $H_2 : n = m$**]** Ici on veut donc prouver que $GinG_{eq}\ Infinite\_Graph_{n+1}\ Infinite\_Graph_n$, ce qui est vrai d'après le Lemme 5.33.

[Cas $H_2 : n < m$**]** On applique la transitivité de *GinG* (Lemme 5.10) et il nous suffit de prouver que :

1. $GinG_{eq}\ Infinite\_Graph_{m+1}\ Infinite\_Graph_m$: c'est vrai d'après le Lemme 5.33.
2. $GinG_{eq}\ Infinite\_Graph_m\ Infinite\_Graph_n$: on applique *IH* avec H_2.

□

A.2.2 Vers une représentation plus souple

A.2.2.1 Preuve du Lemme 6.10

Rappel de l'énoncé. $\forall g_1\ g_2, GPerm\_imp_R\ g_1\ g_2 \Leftrightarrow GPerm\_mend_R\ g_1\ g_2$

Démonstration.

[Direction $\Rightarrow$**]** On raisonne par coinduction et on a $CH : GPerm\_imp_R \subseteq GPerm\_mend_R$. Grâce au Lemme 6.3, l'hypothèse $GPerm\_imp_R\ g_1\ g_2$ nous donne

$$H_1 : R\ (label\ g_1)\ (label\ g_2) \quad H_2 : iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2)$$

On termine la preuve avec la Définition 6.3 appliquée à CH, H_1 et H_2,

[Direction $\Leftarrow$**]** Soit $H : GPerm\_mend_R\ g_1\ g_2$. D'après le Lemme 6.2 avec $\mathcal{R} := GPerm\_mend_R$, il nous suffit de montrer que :

1. $\forall g_1'\ g_2', GPerm\_mend_R\ g_1'\ g_2' \Rightarrow \begin{array}{ll} & R\ (label\ g_1')\ (label\ g_2') \\ \wedge & iperm\_ind_{GPerm\_mend_R}\ (sons\ g_1')\ (sons\ g_2') \end{array}$:
   l'hypothèse $GPerm\_mend_R\ g_1'\ g_2'$ nous donne $\mathcal{R}$ telle que

   $$H_1 : \mathcal{R} \subseteq GPerm\_mend_R \qquad H_2 : R\ (label\ g_1')\ (label\ g_2')$$
   $$H_3 : iperm\_ind_{\mathcal{R}}\ (sons\ g_1')\ (sons\ g_2')$$

   On doit prouver que :

   (a) $R\ (label\ g_1')\ (label\ g_2')$: c'est H_2.

   (b) $iperm\_ind_{GPerm\_mend_R}\ (sons\ g_1')\ (sons\ g_2')$: on le prouve grâce au Lemme 4.34 appliqué à H_1 et H_3.
2. $GPerm\_mend\ g_1\ g_2$: c'est H.

□

A.2.2.2 Preuve du Lemme 6.11

Rappel de l'énoncé. $\forall g_1\ g_2, GeqPath_R\ g_1\ g_2 \Rightarrow lg\ (sons\ g_1) = lg\ (sons\ g_2)$

Démonstration. Soient $H : GeqPath_R\ g_1\ g_2$ et t_1, n_1, ln_1, t_2, n_2 et ln_2 tels que $g_1 = mk\_Graph\ t_1\ \langle n_1, ln_1\rangle$ et $g_2 = mk\_Graph\ t_2\ \langle n_2, ln_2\rangle$. On veut prouver que $n_1 = n_2$. Pour pouvoir se servir de H, on va essayer de trouver des listes d'entiers qui nous permettent d'avancer. En particulier, on va chercher des entiers inférieurs à n_1 et n_2. Pour cela, analysons les différentes valeurs que peuvent prendre n_1 et n_2 :

[Cas 0 **et** 0**]** Ici on veut prouver que $0 = 0$ ce qui est vrai par réflexivité.

[Cas 0 **et** $n_2 + 1$**]** Ici on veut prouver que $0 = n_2 + 1$. Ceci est manifestement faux, nous devons donc trouver une contradiction dans nos hypothèses. On a

$$H : \forall l, RelOp_R\ (rpath\ l\ (mk\_Graph\ t_1\ \langle 0, ln_1\rangle))\ (rpath\ l\ (mk\_Graph\ t_2\ \langle n_2 + 1, ln_2\rangle))$$

Appliquons H à la liste $l := [n_2]$. Cela nous donne :

$$H_1 : RelOp_R\ (rpath\ [n_2]\ (mk\_Graph\ t_1\ \langle 0, ln_1\rangle))\ (rpath\ [n_2]\ (mk\_Graph\ t_2\ \langle n_2 + 1, ln_2\rangle))$$

On a de façon immédiate que $H_2 : 0 \leq n_2$ et $H_3 : n_2 < n_2 + 1$. En utilisant la Définition 6.4 on a :

$$H_1 : RelOp_R\ None\ (rpath\ []\ (ln_2\ (code\ H_3)))$$

Et encore :

$$H_1 : RelOp_R\ None\ (Some(label(ln_2(code\ H_3))))$$

Ce qui donne *False* d'après la Définition 6.5.

[Cas $n_1 + 1$ **et** 0**]** La preuve est ici absolument similaire à celle du cas précédent. Nous ne la détaillons pas.

[Cas $n_1 + 1$ **et** $n_2 + 1$**]** On veut ici montrer que $n_1 + 1 = n_2 + 1$ ou encore $n_1 = n_2$. Pour cela, nous aurons besoin des deux lemmes suivants, qui se déduisent aisément des Définitions 6.4 et 6.5.

Lemme A.5. $\forall g\ n\ t, RelOp_R\ (rpath\ [n]\ g)\ (Some\ t) \Rightarrow n < lg(sons\ g)$

Et de la même façon pour l'autre argument :

Lemme A.6. $\forall g\ n\ t, RelOp_R\ (Some\ t)\ (rpath\ [n]\ g) \Rightarrow n < lg(sons\ g)$

On a :

$$H : \forall l, RelOp_R\ (rpath\ l\ (mk\_Graph\ t_1\ \langle n_1 + 1, ln_1\rangle))\ (rpath\ l\ (mk\_Graph\ t_2\ \langle n_2 + 1, ln_2\rangle))$$

Appliquons H à la liste $l := [n_1]$. Cela nous donne :

$$H_1 : RelOp_R\ (rpath\ [n_1]\ mk\_Graph\ t_1\ \langle n_1 + 1, ln_1\rangle)\ (rpath\ [n_1]\ mk\_Graph\ t_2\ \langle n_2 + 1, ln_2\rangle)$$

On a de façon immédiate que $H_2 : n_1 \leq n_1 + 1$. En utilisant la Définition 6.4 deux fois (comme précédemment) on a :

$$H_1 : RelOp_R\ (Some(label(ln_2(code\ H_2))))\ (rpath\ [n_1]\ (mk\_Graph\ t_2\ \langle n_2 + 1, ln_2\rangle))$$

Du Lemme A.6 appliqué à H_1 on déduit $H_3 : n_1 < n_2 + 1$ ou encore $H_3 : n_1 \leq n_2$

On fait de même avec $l := [n_2]$ et en utilisant le même processus (mais on applique cette fois-ci le Lemme A.5), on obtient $H_4 : n_2 \leq n_1$.

De H_3 et H_4 on déduit que $n_1 = n_2$.

□

A.2.2.3 Preuve de la Propriété 6.1

Rappel de l'énoncé.

$$\begin{array}{rcl} \forall g_1\ g_2, g_1 \equiv_{R,0} g_2 & \Leftrightarrow & R\ (label\ g_1)\ (label\ g_2) \\ \forall g_1\ g_2, g_1 \equiv_{R,n+1} g_2 & \Leftrightarrow & R\ (label\ g_1)\ (label\ g_2) \\ & \wedge & iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2) \end{array}$$

Démonstration.

[Propriété 6.1.1] On veut montrer que : $g_1 \equiv_{R,0} g_2 \Leftrightarrow R\ (label\ g_1)\ (label\ g_2)$. Ou encore, d'après la Définition 6.8 :

$$TPerm_R\ (mk\_iTree\ (label\ g_1)\ [\![]\!])\ (mk\_iTree\ (label\ g_2)\ [\![]\!]) \Leftrightarrow R\ (label\ g_1)\ (label\ g_2)$$

[Direction $\Rightarrow$] Soit $H : TPerm_R\ (mk\_iTree\ (label\ g_1)\ [\![]\!])\ (mk\_iTree\ (label\ g_2)\ [\![]\!])$. On veut montrer que $R\ (label\ g_1)\ (label\ g_2)$. Grâce à la Définition 6.9, H nous donne $H_1 : R\ (label\ g_1)\ (label\ g_2)$. Ce qu'on voulait démontrer.

[Direction $\Leftarrow$] Soit $H : R\ (label\ g_1)\ (label\ g_2)$. D'après la Définition 6.9, il nous suffit de montrer que

1. $R\ (label\ g_1)\ (label\ g_2)$: ce qui est vrai d'après H
2. $iperm\_ind_{TPerm_R}\ [\![]\!]\ [\![]\!]$: ce qui est vrai d'après la Définition 4.17.

[Propriété 6.1.2] On veut montrer que :

$$g_1 \equiv_{R,n+1} g_2 \Leftrightarrow R\ (label\ g_1)\ (label\ g_2) \wedge iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$$

Ou encore, d'après la Définition 6.8 :

$$\begin{array}{rl} & TPerm_R\ \ (mk\_iTree\ (label\ g_1)\ (imap\ (G2iT\ n)\ (sons\ g_1))) \\ & \qquad\quad (mk\_iTree\ (label\ g_2)\ (imap\ (G2iT\ n)\ (sons\ g_2))) \\ \Leftrightarrow & R\ (label\ g_1)\ (label\ g_2) \wedge iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2) \end{array}$$

[Direction $\Rightarrow$] Soit $H : \begin{array}{rl} TPerm_R & (mk\_iTree\ (label\ g_1)\ (imap\ (G2iT\ n)\ (sons\ g_1))) \\ & (mk\_iTree\ (label\ g_2)\ (imap\ (G2iT\ n)\ (sons\ g_2))) \end{array}$

Grâce à H et à la Définition 6.9, on obtient :

$$\begin{array}{l} H_1 : R\ (label\ g_1)\ (label\ g_2) \\ H_2 : iperm\_ind_{TPerm_R}\ (imap\ (G2iT\ n)\ (sons\ g_1))\ (imap\ (G2iT\ n)\ (sons\ g_2)) \end{array}$$

On veut prouver que :

1. $R\ (label\ g_1)\ (label\ g_2)$: c'est H_1.
2. $iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$: d'après le Lemme 4.30 cela revient à montrer que : $iperm\_ind_{TPerm_R}\ (imap\ (G2iT\ n)\ (sons\ g_1))\ (imap\ (G2iT\ n)\ (sons\ g_2))$. C'est H_2.

[Direction $\Leftarrow$] Soient $H_1 : R\ (label\ g_1)\ (label\ g_2)$ et $H_2 : iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$. On veut montrer que :

$$\begin{array}{rl} & TPerm_R\ \ (mk\_iTree\ (label\ g_1)\ (imap\ (G2iT\ n)\ (sons\ g_1))) \\ & \qquad\quad (mk\_iTree\ (label\ g_2)\ (imap\ (G2iT\ n)\ (sons\ g_2))) \end{array}$$

D'après la Définition 6.9, il nous suffit de démontrer que :

1. $R\ (label\ g_1)\ (label\ g_2)$: ce qui est vrai d'après H_1.
2. $iperm\_ind_{TPerm_R}\ (imap\ (G2iT\ n)\ (sons\ g_1))\ (imap\ (G2iT\ n)\ (sons\ g_2))$: c'est-à-dire, d'après le Lemme 4.30 : $iperm\_ind_{\lambda t_1.\lambda t_2.TPerm_R\ (G2iT\ n\ t_1)\ (G2iT\ n\ t_2)}\ (sons\ g_1)\ (sons\ g_2)$. Ou encore d'après la Définition 6.10 : $iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$. Ce qui est vrai d'après H_2.

□

A.2.2.4 Preuve du Lemme 6.19

Rappel de l'énoncé. $Dec\ R \Rightarrow Dec \equiv_{R,n}$

Démonstration. Soit $H : Dec\ R$. On va raisonner par induction sur n puis par analyse de cas en se servant de H et de l'hypothèse d'induction. On veut montrer que

$$g_1 \equiv_{R,n} g_2 \vee \neg(g_1 \equiv_{R,n} g_2)$$

[Cas 0**]** Comparons les labels de g_1 et g_2 à l'aide de H :

[Cas $H_1 : R\ (label\ g_1)\ (label\ g_2)$**]** Dans ce cas on choisit de montrer que $g_1 \equiv_{R,0} g_2$. Ceci est vrai d'après la Propriété 6.1.1 et H_1.

[Cas $H_1 : \neg(R\ (label\ g_1)\ (label\ g_2))$**]** Ce cas est analogue au précédent et on montre que $\neg(g_1 \equiv_{R,0} g_2)$.

[Cas $n+1$**]** L'hypothèse d'induction est $IH : Dec \equiv_{R,n}$. Comparons les labels de g_1 et g_2 à l'aide de H :

[Cas $H_1 : R\ (label\ g_1)\ (label\ g_2)$**]** Ici, les labels sont équivalents. Pour savoir dans quelle situation nous sommes (graphes équivalents ou non), nous devons aller voir plus loin. Pour cela, comparons les fils de g_1 et g_2 à l'aide du Lemme 4.31 et de IH.

[Cas $H_2 : iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$**]** Dans ce cas tout est équivalent, on choisit donc de montrer que $g_1 \equiv_{R,n+1} g_2$. On le fait à l'aide de la Propriété 6.1.2 appliquée à H_1 et H_2.

[Cas $H_2 : \neg(iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2))$**]** Ce cas est analogue au précédent et on montre que $\neg(g_1 \equiv_{R,n+1} g_2)$.

[Cas $H_1 : \neg(R\ (label\ g_1)\ (label\ g_2))$**]** Ce cas est analogue au précédent et on montre que$\neg(g_1 \equiv_{R,n+1} g_2)$.

□

A.2.2.5 Preuve du Lemme 6.20

Rappel de l'énoncé. $\forall g_1\ g_2\ n, g_1 \equiv_{R,n+1} g_2 \Rightarrow g_1 \equiv_{R,n} g_2$

Démonstration. Soit $H_1 : g_1 \equiv_{R,n+1} g_2$. D'après la Propriété 6.1.2, H_1 nous donne :

$$H_2 : R\ (label\ g_1)\ (label\ g_2) \quad H_3 : iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$$

On raisonne par induction sur n.

[Cas 0**]** D'après la Propriété 6.1.1, il nous suffit de montrer que $R\ (label\ g_1)\ (label\ g_2)$, ce qui est vrai d'après H_2 .

[Cas $n+1$] L'hypothèse d'induction est $IH : \equiv_{R,n+1} \subseteq \equiv_{R,n}$. On a maintenant H_3 : $iperm\_ind_{\equiv_{R,n+1}}\ (sons\ g_1)\ (sons\ g_2)$. On veut montrer que $g_1 \equiv_{R,n+1} g_2$. D'après la Propriété 6.1.2, il nous suffit de montrer que :

1. $R\ (label\ g_1)\ (label\ g_2)$: c'est vrai d'après H_2.
2. $iperm\_ind_{\equiv_{R,n}}\ (sons\ g_1)\ (sons\ g_2)$: ce qui est vrai d'après le Lemme 4.34 appliqué à IH et à H_3.

□

A.2.2.6 Preuve du Lemme 6.24

Rappel de l'énoncé. $DNE \Rightarrow \forall P, \neg(\forall t, \neg(Pt)) \Rightarrow \exists t, P\ t$

Démonstration. On va faire une succession de raisonnements par l'absurde. Soient $H_1 : DNE$ et $H_2 : \neg(\forall t, \neg(Pt))$. On veut montrer que $\exists t, P\ t$. D'après H_1, il nous suffit de montrer que $\neg\neg(\exists t, P\ t)$. Supposons que $H_3 : \neg(\exists t, P\ t)$ et montrons que nous arrivons à une contradiction. En l'occurrence, nous allons montrer que : $\forall t, \neg(Pt)$. On veut donc montrer que $\neg(Pt)$. Supposons donc encore que $H_4 : P\ t$ on peut alors démontrer que $\exists t, P\ t$, ce qui est en contradiction avec H_3. On a donc bien montré que $\forall t, \neg(Pt)$ ce qui est en contradiction avec H_2. □

A.2.2.7 Preuve du Lemme 6.26

Rappel de l'énoncé.

$$\forall m, (\forall R : Fin\ m \to \mathbb{N} \to Prop, (\forall x \exists y, R\ x\ y) \Rightarrow \exists f : Fin\ m \to \mathbb{N}, (\forall x, R\ x\ (f\ x)))$$

Démonstration. Soit $H_1 : \forall x \exists y, R\ x\ y$. On raisonne par induction sur m.

[Cas 0] Comme $Fin\ 0$ est vide, on peut prendre ce que l'on veut. On prend $f := \lambda x.0$ et on doit montrer que $\forall x, R\ x\ (f\ x))$. Ceci est vrai puisque $Fin\ 0$ est vide.

[Cas $m+1$] L'hypothèse d'induction est

$$IH : \forall R : Fin\ m \to \mathbb{N} \to Prop, (\forall x \exists y, R\ x\ y) \Rightarrow \exists f : Fin\ m \to \mathbb{N}, (\forall x, R\ x\ (f\ x))$$

Soit R' la relation définie par $\forall x, R'\ x := R\ (succ\ x)$. R' est de type $Fin\ m \to \mathbb{N} \to Prop$. Pour pouvoir utiliser IH, démontrons que $H_2 : \forall x \exists y, R'\ x\ y$. C'est-à-dire : $\forall x \exists y, R\ (succ\ x)\ y$. On le montre grâce à H_1 appliqué à $succ\ x$.
IH appliqué à R' et H_2 nous donne $f : Fin\ m \to \mathbb{N}$ tel que $H_3 : \forall x, R'\ x\ (f\ x)$. On veut montrer que

$$\exists f' : Fin\ (m+1) \to \mathbb{N}, (\forall x, R\ x\ (f'\ x))$$

H_1 appliqué à $first\ m$ nous donne y_0 tel que $H_4 : R\ (first\ m)\ y_0$.
On va prendre pour f' la fonction définie par les assertions suivantes :

$$\begin{array}{lll} f'\ \ (first\ m) & := & y_0 \\ f'\ \ (succ\ i) & := & f\ i \end{array}$$

On doit montrer que $\forall x, R\ x\ (f'\ x)$. On a deux cas pour x :

1. $x = first\ m$: d'après la définition de f', $R\ x\ (f'\ x)$ nous donne $R\ (first\ m)\ y_0$ ce qui est vrai d'après H_4.

2. $x = succ\ i$: d'après la définition de f', $R\ x\ (f'\ x)$ nous donne $R\ (succ\ i)\ (f\ i)$ ce qui est vrai d'après H_3.

□

A.2.2.8 Preuve de la Propriété 6.2

Rappel de l'énoncé. $\forall n\ (f : Fin\ n \to \mathbb{N}), (\forall i, MaxFin\ f \geq f\ i)$

Démonstration. On veut montrer que : $max\_list\_nat\ (map\ f\ (makeListFin\ n)) \geq f\ i$. D'après le Lemme 5.28, il nous suffit de montrer que : $f\ i \in map\ f\ (makeListFin\ n)$. Ou encore, d'après un résultat bien connu sur *map* que $i \in makeListFin\ n$, ce qui se démontre aisément par définition de *makeListFin*. □

A.2.2.9 Preuve du Lemme 6.28

Rappel de l'énoncé. $\forall (f : T \to U)\ (g : U \to T), bij\ f\ g \wedge IPPGen\ T \Rightarrow IPPGen\ U$

Démonstration. Soient H_1 : $bij\ f\ g$ et H_2 : $IPPGen\ T$. On a $P : \mathbb{N} \to U \to Prop$ tel que $H_3 : \forall n \exists s, P\ n\ s$ et on veut montrer que :

$$\exists s_0, (\forall n \exists n', n' \geq n \wedge P\ n'\ s_0)$$

H_1 nous donne $H_{11} : \forall t, g(f\ t) = t$ et $H_{12} : \forall u, f(g\ u) = u$.

Pour pouvoir utiliser H_2, on veut montrer que $H_4 : \forall n \exists a, P\ n\ (f\ a)$. On fixe n. H_3 nous donne b tel que H_3' : $P\ n\ b$. On instancie l'existencielle avec $g\ b$ et on doit prouver que $P\ n\ (f(g\ b))$, ou encore, d'après H_{12}, $P\ n\ b$, ce qui est vrai d'après H_3'. On a donc H_4 : $\forall n \exists a, P\ n\ (f\ a)$.

H_2 utilisé avec H_4 nous donne a_0 tel que $H_5 : \forall n \exists n', n' \geq n \wedge P\ n'\ (f\ a_0)$. On veut toujours montrer que :

$$\exists s_0, (\forall n \exists n', n' \geq n \wedge P\ n'\ s_0)$$

On instancie l'existentielle avec $f\ a_0$ et on doit montrer que $\forall n \exists n', n' \geq n \wedge P\ n'\ (f\ a_0)$, ce qui est vrai d'après H_5. □

A.2.2.10 Preuve du Lemme 6.30

Rappel de l'énoncé. $TiersEx \Rightarrow DNE$

Démonstration. Soit $H_1 : \forall P, P \vee \neg P$. On a P tel que $H_2 : \neg\neg P$ et on veut montrer que P est vraie. D'après H_1, on a deux solutions pour P :

[Cas $H_3 : P$**]** On veut montrer que P ce qui est directement vrai par H_3.

[Cas $H_3 : \neg P$**]** H_3 est en contradiction avec H_2.

□

A.2.2.11 Preuve du Lemme 6.36

Rappel de l'énoncé. $\forall g_1\ g_2, GPerm\_bij_R\ g_1\ g_2 \Leftrightarrow GPerm\_bij\_mend_R\ g_1\ g_2$

Démonstration.

[Direction $\Rightarrow$] On raisonne par coinduction avec CH : $GPerm\_bij_R \subseteq GPerm\_bij\_mend_R$. D'après la Définition 6.17, l'hypothèse $GPerm\_bij_R\ g_1\ g_2$ nous donne :

$$H_1 : R\ (label\ g_1)\ (label\ g_2) \quad H_2 : iperm\_bij_{GPerm\_bij_R}\ (sons\ g_1)\ (sons\ g_2)$$

On applique alors simplement la Définition 6.18 en prenant $\mathcal{R} := GPerm\_bij_R$. On doit alors montrer que :

1. $GPerm\_bij_R \subseteq GPerm\_bij\_mend_R$: c'est CH.
2. $R\ (label\ g_1)\ (label\ g_2)$: c'est H_1.
3. $iperm\_bij_{GPerm\_bij_R}\ (sons\ g_1)\ (sons\ g_2)$: c'est H_2.

[Direction $\Leftarrow$] On raisonne par coinduction avec CH : $GPerm\_bij\_mend_R \subseteq GPerm\_bij_R$. D'après les Définitions 6.18 et 4.25, l'hypothèse $GPerm\_bij\_mend_R\ g_1\ g_2$ nous donne $\mathcal{R}$, f et g tels que :

$$H_1 : \mathcal{R} \subseteq GPerm\_bij\_mend_R \quad H_2 : R\ (label\ g_1)\ (label\ g_2) \quad H_3 : bij\ f\ g$$
$$H_4 : \forall i, \mathcal{R}\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ (f\ i))$$

D'après la Définition 6.17, il nous suffit de montrer que :

1. $R\ (label\ g_1)\ (label\ g_2)$: c'est H_2.
2. $iperm\_bij_{GPerm\_bij_R}\ (sons\ g_1)\ (sons\ g_2)$: d'après la Définition 4.25 appliquée à H_3, il nous suffit de prouver que :

$$\forall i, GPerm\_bij_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ (f\ i))$$

D'après CH, il nous suffit de prouver que :

$$GPerm\_bij\_mend_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ (f\ i))$$

Et d'après H_1 :

$$R'\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ (f\ i))$$

Ce qui est vrai d'après H_4.

□

A.2.2.12 Preuve du Lemme 6.37

Rappel de l'énoncé. $\forall g_1\ g_2, GPerm\_mend_R\ g_1\ g_2 \Leftrightarrow GPerm\_bij\_mend_R\ g_1\ g_2$

Démonstration.

[Direction $\Rightarrow$] On raisonne par coinduction avec CH : $GPerm\_mend_R \subseteq GPerm\_bij\_mend_R$. D'après la Définition 6.3, l'hypothèse $GPerm\_mend_R\ g_1\ g_2$ nous donne $\mathcal{R}$ telle que :

$$H_1 : \mathcal{R} \subseteq GPerm\_mend_R \qquad H_2 : R\ (label\ g_1)\ (label\ g_2)$$
$$H_3 : iperm\_ind_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2)$$

D'après la Définition 6.17 avec $\mathcal{R} := GPerm\_mend_R$, il nous suffit de montrer que :

1. $GPerm\_mend_R \subset GPerm\_bij\_mend_R$: c'est CH.
2. $R\ (label\ g_1)\ (label\ g_2)$: c'est H_2.
3. $iperm\_bij_{GPerm\_mend_R}\ (sons\ g_1)\ (sons\ g_2)$: d'après le Théorème 4.2, il nous suffit de prouver que :
$$iperm\_ind_{GPerm\_mend_R}\ (sons\ g_1)\ (sons\ g_2)$$
D'après le Lemme 4.34 utilisé avec H_1, il nous suffit de prouver que
$$iperm\_ind_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2)$$
Ce qui est vrai d'après H_3.

[Direction $\Leftarrow$] On raisonne par coinduction avec $CH : GPerm\_bij\_mend_R \subseteq GPerm\_mend_R$. D'après la Définition 6.18, l'hypothèse $GPerm\_bij\_mend_R\ g_1\ g_2$ nous donne $\mathcal{R}$ telle que :
$$H_1 : \mathcal{R} \subseteq GPerm\_bij\_mend_R \qquad H_2 : R\ (label\ g_1)\ (label\ g_2)$$
$$H_3 : iperm\_bij_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2)$$
D'après la Définition 6.3 avec $\mathcal{R} := GPerm\_bij\_mend_R$, il nous suffit de montrer que :
1. $GPerm\_bij\_mend_R \subseteq GPerm\_mend_R$: c'est CH.
2. $R\ (label\ g_1)\ (label\ g_2)$: c'est H_2.
3. $iperm\_ind_{GPerm\_bij\_mend_R}\ (sons\ g_1)\ (sons\ g_2)$: d'après le Théorème 4.2, il nous suffit de prouver que :
$$iperm\_bij_{GPerm\_bij\_mend_R}\ (sons\ g_1)\ (sons\ g_2)$$
D'après le Lemme 4.48 utilisé avec H_1, il nous suffit de prouver que
$$iperm\_bij_{\mathcal{R}}\ (sons\ g_1)\ (sons\ g_2)$$
Ce qui est vrai d'après H_3.

□

A.2.2.13 Preuve du Lemme 6.40

Rappel de l'énoncé. $\forall g_1\ g_2, GinGP_R\ g_1\ g_2 \Rightarrow \forall i_1, GinGP_R\ (fct\ (sons\ g_1)\ i_1)\ g_2$

Démonstration. Soit $H_1 : GinGP_R\ g_1\ g_2$. On raisonne par induction sur H_1.

[Cas de base] On a maintenant $H_1 : GPerm\_imp_R\ g_1\ g_2$. Le Lemme 6.3 appliqué à H_1 nous donne
$$H_2 : R\ (label\ g_1)\ (label\ g_2) \quad H_3 : iperm\_ind_{GPerm\_imp_R}\ (sons\ g_1)\ (sons\ g_2)$$
Et le Lemme 4.21 nous donne i' tel que $H_4 : GPerm\_imp_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ i')$. Selon la règle indirG de la Définition 5.5, il nous suffit de prouver que
$$GinGP_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ i')$$
On applique alors la règle dirG de la Définition 5.5 avec H_4.

[Cas inductif] On a i' tel que $H_1 : GinGP_R\ g_1\ (fct\ (sons\ g_2)\ i')$ et l'hypothèse d'induction est :
$$IH : GinGP_R\ (fct\ (sons\ g_1)\ i)\ (fct\ (sons\ g_2)\ i')$$
On peut donc directement appliquer la règle indirG de la Définition 5.5 avec IH.

□

A.2.2.14 Preuve du Lemme 6.41

Rappel de l'énoncé.

$$R \text{ transitive} \Rightarrow (\forall g_1\ g_2\ g_3, GinGP_R\ g_1\ g_2 \wedge GinGP_R\ g_2\ g_3 \Rightarrow GinGP_R\ g_1\ g_3)$$

Démonstration. Soient $H_1 : GinGP_R\ g_1\ g_2$ et $H_2 : GinGP_R\ g_2\ g_3$. On va raisonner par induction sur H_1.

[Cas de base] On a maintenant $H_1 : GPerm\_imp_R\ g_1\ g_2$. On va raisonner par induction sur H_2 :

- **[Cas de base]** On a $H_2 : GPerm\_imp_R\ g_2\ g_3$. On doit prouver que $GinGP_R\ g_1\ g_3$. Selon la règle dirG de la Définition 5.5, il nous suffit de prouver que $GPerm\_imp_R\ g_1\ g_3$. On montre cela par transitivité de $GPerm\_imp_R$ (Lemme 6.8) avec H_1 et H_2.
- **[Cas inductif]** On a i_2 tel que $H_2 : GinGP_R\ g_2\ (fct\ (sons\ g_3)\ i_2)$ et l'hypothèse d'induction est $IH_2 : GinGP_R\ g_1\ (fct\ (sons\ g_3)\ i_2)$. On veut montrer que $GinGP_R\ g_1\ g_3$. Pour cela, on applique simplement la règle indirG de la Définition 5.5 avec IH_2.

[Cas inductif] On a i_1 tel que $H_1 : GinGP_R\ g_1\ (fct\ (sons\ g_2)\ i_1)$ et l'hypothèse d'induction est $IH_1 : GinGP_R\ (fct\ (sons\ g_2)\ i_1)\ g_3 \Rightarrow GinGP_R\ g_1\ g_3$. Donc, selon IH_1 il nous suffit de prouver que $GinGP_R\ (fct\ (sons\ g_2)\ i_1)\ g_3$. On termine la preuve en utilisant le Lemme 6.40 avec H_2.

□

B Correspondances

Nous présentons ici une liste de correspondances entre les éléments apparaissant dans la thèse et leurs équivalents dans les fichiers Coq. Chaque élément de la thèse est donné avec la section dans laquelle il apparaît, sa référence (si elle existe), la page où on peut le trouver (colonne **p.**) et son nom le cas échéant. Pour son correspondant dans le code, on fournit son nom, le fichier (.v) dans lequel il se trouve et la ligne à laquelle il se situe (colonne **l.**).

Pour des raisons de place, nous avons utilisé des abbréviations. Ainsi, Déf. correspond à Définition, Lem. à Lemme, Thm. à Théorème, Prop. à Propriété, Cor. à Corollaire et Ax. à Axiome. Dans la colonne **Fichier**, IPPJust renvoie au fichier IPPJustification, EquivCont à EquivContejean et stdlib, à la librairie standard (lorsqu'il s'agit de définitions ou de résultats que nous énonçons mais qui font partie de la libraire standard). Finalement, il faut préciser que dans la colonne **Dans les scripts/Nom** (cinquième colonne), certains noms ont dû être coupés pour tenir dans la case, mais ils sont toujours d'une seule pièce dans les scripts.

Dans la thèse				**Dans les scripts**		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
1.2.1	Déf. 1.4	10	*list_rel*	`ListEq`	ListEq	24
3.1.1.1	Déf. 3.1	44	*Fin*	`Fin`	Fin	25
	Déf. 3.2	45	*makeListFin*	`makeListFin`	Fin	674
	Lem. 3.2	45		`makeListFin_ nb_elem_ok`	Fin	693
	Lem. 3.3	45		`all_Fin_n_in_ makeListFin`	Fin	705
		45	*makeListFin* énumération	`enum_makeListFin`	CardFin	325
	Lem. 3.4	45		`Fin_0_empty`	Fin	230
3.1.1.2	Déf. 3.3	45	*decode*	`decode_Fin`	Fin	58
	Lem. 3.5	46		`decode_Fin_inf_n`	Fin	63
	Déf. 3.4	46	*code*	`code_Fin1`	Fin	122
	Lem. 3.6	46		`code_Fin1_proofirr`	Fin	201
	Lem. 3.7	46		`code1_decode_Id`	Fin	308
	Lem. 3.8	46		`decode_code1_Id`	Fin	322
	Déf. 3.5	47	*NatSeg*	`NatSeg`	NatSeg	20
	Lem. 3.9	47		`decode_Fin_unique`	Fin	552
3.1.1.3	Lem. 3.10	47		`Fin_inj`	Fin	959
	Déf. 3.6	47	*bij*	`Bijective`	Tools	290
	Lem. 3.11	48		`Bij_sym`	Tools	293
	Lem. 3.12	48		`Bij_trans`	Tools	300

Dans la thèse				Dans les scripts		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
	Lem. 3.13	48		`bij_inj`	Tools	313
	Lem. 3.14	48		`Fin_inj_aux`	Fin	925
	Déf. 3.7	49	*getcons*	`get_cons`	Fin	592
	Prop. 3.1.1	49		`get_cons_ok1`	Fin	660
	Prop. 3.1.2	49		`get_cons_ok2`	Fin	666
	Déf. 3.8	49	*transfoFun*	`FSnFSn'_FnFn'`	Fin	840
	Lem. 3.15	50		`FSnFSn'_FnFn'_aux_ bij`	Fin	891
	Lem. 3.16	50		`FSnFSn'_FnFn'_bij`	Fin	917
	Rq. 3.12	50		`Fin_inj_alt`	CardFin	342
3.1.2	Déf. 3.9	50	*ilistn*	`ilistn`	Ilist	68
	Déf. 3.10	51	*ilist*	`ilist`	Ilist	75
		51	*lg*	`lgti`	Ilist	82
		51	*fct*	`fcti`	Ilist	84
3.2.1		52	*conv*	`rewriteFins`	Fin	784
	Prop. 3.2	52		`decode_Fin_match'`	Fin	788
	Déf. 3.11	52	*ilist_rel*	`ilist_rel`	Ilist	110
	Lem. 3.17	52		`ilist_rel_nil`	Ilist	1346
	Lem. 3.18	52		`ilist_rel_refl`	Ilist	120
	Lem. 3.19	52		`ilist_rel_sym`	Ilist	129
	Lem. 3.20	53		`ilist_rel_trans`	Ilist	141
	Lem. 3.21	53		`ilist_relRel`	Ilist	153
	Lem. 3.22	53		`ilist_rel_mon`	Ilist	194
3.2.2	Lem. 3.23	54		`ilist_rel_dec`	Ilist	1360
3.2.3	Lem. 3.24	56		`nth_makeListFin_def`	Fin	736
	Déf. 3.13	56	*ilist2list*	`ilist2list`	Ilist	216
	Lem. 3.25	56		`ilist_rel_eq`	Ilist	827
	Déf. 3.14	56	*list2FinT*	`list2Fin_T`	Ilist	310
	Prop. 3.3	57		`list2Fin_T_first`	Ilist	429
	Prop. 3.4	57		`list2Fin_T_succ`	Ilist	435
	Lem. 3.26	57		`list2Fin_T_map`	Ilist	488
	Lem. 3.27	57		`list2Fin_T_ makeListFin`	Ilist	520
	Déf. 3.15	57	*list2ilist*	`list2ilist`	Ilist	317
	Lem. 3.28	57		`list2ilist_nth2`	Ilist	392
	Lem. 3.29	57		`ilist2list_nth'`	Ilist	261
	Thm. 3.1	58		`list2ilist_ ilist2list_id`	Ilist	416
	Thm. 3.2	58		`ilist2list_ list2ilist_id`	Ilist	463
	Prop. 3.5	58		`map_map`	stdlib	
3.2.4	Déf. 3.16	59	*ifilter*	`ifilterB`	Ilist	483
3.2.4.1	Déf. 3.17	60	*imap*	`imap`	Ilist	96
	Lem. 3.32	60		`imap_apply`	Ilist	1229
3.2.4.2	Déf. 3.18	60	*rightFin*	`rightFin`	Ilist	
	Déf. 3.20	61	*iappend*	`iappend`	Ilist	648

Dans la thèse				Dans les scripts		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
	Prop. 3.6.1	61		`iappend_lgti`	Ilist	658
	Prop. 3.6.2	61		`iappend_left`	Ilist	664
	Prop. 3.6.3	61		`iappend_right`	Ilist	679
	Lem. 3.33	61		`iappend_append`	Ilist	701
	Lem. 3.34	61		`append_iappend`	Ilist	729
	Lem. 3.35	61		`eq_nth`	Tools	22
	Cor. 3.36	62		`eq_nth_cor`	Tools	40
3.2.4.3	Déf. 3.21	62	*iall*	`iall`	Ilist	100
3.2.5		62	$iniln_0$	`iniln`	Ilist	999
	Déf. 3.22	62	*inil*	`inil`	Ilist	1005
	Déf. 3.23	62	*iconsn*	`iconsn`	Ilist	1007
	Déf. 3.24	63	*icons*	`icons`	Ilist	1015
	Lem. 3.37	63		`inil_nil`	Ilist	1159
	Lem. 3.38	63		`icons_cons`	Ilist	1171
	Lem. 3.39	63		`nil_inil`	Ilist	1166
	Lem. 3.40	63		`cons_icons`	Ilist	1192
	Lem. 3.41	63		`cons_icons'`	Ilist	1217
	Déf. 3.25	63	*ihead*	`ihead`	Ilist	1049
	Déf. 3.26	63	*itail*	`itail`	Ilist	1056
		63	Assertion	`ihead_itail_ok`	Ilist	1069
3.2.6	Déf. 3.28	64	*ileft*	`left_sib`	Ilist	1237
	Déf. 3.30	64	*iright*	`right_sib`	Ilist	1249
	Prop. 3.7.1	64		`left_sib_lgti`	Ilist	1244
	Prop. 3.7.2	64		`right_sib_lgti`	Ilist	1259
	Prop. 3.7.3	64		`left_sib_right_sib_lgti`	Ilist	1265
	Lem. 3.42	65		`left_sib_right_sib`	Ilist	1273
	Cor. 3.43	65		`left_sib_right_sib_cor`	Ilist	1333
3.3.2	Déf. 3.31	66	*PropMult*	`PropMult`	IlistMult	33
	Déf. 3.32	66	*ilistnMult*	`ilistnMult`	IlistMult	40
	Déf. 3.33	66	*ilistMult*	`ilistMult`	IlistMult	71
		66	*fctM*	`fctiMult`	IlistMult	78
		66	*lgM*	`lgtiMult`	IlistMult	75
	Déf. 3.34	67	*iM_rel*	`imeq`	IlistMult	124
		67	*iM_rel* préserve l'équivalence	`imeqRel`	IlistMult	172
4.2	Déf. 4.8	71	*nbocc*	`count_occ`	IlistPerm	37
	Déf. 4.9	72	*iperm_occ*	`IlistPerm`	IlistPerm	74
	Lem. 4.2	72		`ilist_rel_finer_IlistPerm`	IlistPerm	193
4.3.1.1	Déf. 4.10	73	*weakFin*	`weakFin`	Extroduce	177
	Lem. 4.3	73		`weakFin_ok`	Extroduce	181
	Déf. 4.12	73	*remEl*	`extroduce`	Extroduce	20
	Prop. 4.1.1	73		`extroduce_lgti`	Extroduce	30
	Prop. 4.1.2	73		`extroduce_ok2'`	Extroduce	201

Dans la thèse				Dans les scripts		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
	Prop. 4.1.3	73		`extroduce_ok3'`	Extroduce	226
	Lem. 4.4	74		`extroduce_ok_cor`	Extroduce	236
	Lem. 4.5	74		`extroduce_ilist_rel`	Extroduce	246
	Lem. 4.6	74		`extroduce_imap`	Extroduce	321
	Lem. 4.7	74		`left_right_sib_ extroduce`	Extroduce	789
	Cor. 4.8	74		`left_right_sib_ extroduce_bis`	Extroduce	827
4.3.1.2	Déf. 4.14	75	*addEl*	`introduce`	Introduce	21
	Prop. 4.2.1	76		`introduce_lgti`	Introduce	34
	Prop. 4.2.2	76		`introduce_ok2'`	Introduce	53
	Prop. 4.2.3	76		`introduce_ok1'`	Introduce	41
	Prop. 4.2.4	76		`introduce_ok3'`	Introduce	81
	Lem. 4.9	76		`introduce_ extroduce_id`	Introduce	97
	Lem. 4.10	76		`extroduce_ introduce_id`	Introduce	140
4.3.1.3	Déf. 4.15	76	*indexInRemEl*	`index_in_extroduce`	Extroduce	430
	Prop. 4.3.1	77		`index_in_extroduce _decode1`	Extroduce	444
	Prop. 4.3.2	77		`index_in_extroduce _decode2`	Extroduce	460
	Prop. 4.3.3	77		`index_in_extroduce _weakFin`	Extroduce	612
	Prop. 4.3.4	77		`index_in_extroduce _weakFin2`	Extroduce	634
	Prop. 4.3.5	77		`index_in_extroduce _succ2`	Extroduce	647
	Prop. 4.3.6	77		`index_in_extroduce _succ`	Extroduce	622
	Lem. 4.11	77		`index_in_extroduce _ok_cor`	Extroduce	527
	Déf. 4.16	77	*indexFromRemEl*	`extroduce_Fin`	Extroduce	347
	Lem. 4.12	77		`extroduce_Fin_ok_ cor`	Extroduce	403
	Lem. 4.13	77		`extroduce_Fin_not_ fex`	Extroduce	355
	Lem. 4.14	78		`index_in_from_ extroduce`	Extroduce	682
	Lem. 4.15	78		`index_from_in_ extroduce`	Extroduce	657
4.3.1.4	Lem. 4.16	78		`extroduce_ interchange_eq`	Extroduce	765
4.3.2	Déf. 4.17	79	*iperm_ind*	`IlistPerm3`	IlistPerm	225
	Déf. 4.18	79	*iperm_ind′*	`IlistPerm4`	IlistPerm	232
	Déf. 4.19	79	*iperm_ind″*	`IlistPerm5`	IlistPerm	260

Dans la thèse				Dans les scripts		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
	Lem. 4.17	79		`IlistPerm3_lgti`	IlistPerm	286
	Lem. 4.18	79		`IlistPerm4nil`	IlistPerm	397
	Lem. 4.19	79		`IlistPerm3_ ilist_rel_eq`	IlistPerm	663
	Lem. 4.20	79		`IlistPerm3_ ilist_rel_eq_snd`	IlistPerm	679
	Thm. 4.1	80	$iperm\_ind_R \Rightarrow iperm\_ind'_R$	`IlistPerm3_ IlistPerm4_eq`	IlistPerm	930
	Thm. 4.1	80	$iperm\_ind'_R \Rightarrow iperm\_ind_R$	`IlistPerm4_ IlistPerm3_eq`	IlistPerm	298
	Thm. 4.1	80	$iperm\_ind''_R \Rightarrow iperm\_ind_R$	`IlistPerm5_ IlistPerm3_eq`	IlistPerm	317
	Lem. 4.21	80		`IlistPerm3_ exists_rec`	IlistPerm	843
4.3.3.1	Lem. 4.22	83		`IlistPerm3_refl_ refl`	IlistPerm	373
	Lem. 4.23	84		`IlistPerm3_flip`	IlistPerm	425
	Lem. 4.24	84		`IlistPerm3_sym_sym`	IlistPerm	456
	Déf. 4.20	85	*transAt*	`TransitiveAt`	IlistPerm	468
	Lem. 4.25	85		`IlistPerm4_trans_ refined`	IlistPerm	471
	Lem. 4.26	86		`IlistPerm4_trans_ trans`	IlistPerm	509
	Lem. 4.27	86		`IlistPerm4_trans_ special`	IlistPerm	1021
	Lem. 4.28	86	*iperm_ind*	`IlistPerm3Rel`	IlistPerm	981
	Lem. 4.28	86	*iperm_ind'*	`IlistPerm4Rel`	IlistPerm	1005
4.3.3.2	Lem. 4.29	86		`ilist_rel_finer_ IlistPerm3`	IlistPerm	722
	Lem. 4.30	87	$\Rightarrow$	`IlistPerm3_imap_bis`	IlistPerm	802
	Lem. 4.30	87	$\Leftarrow$	`IlistPerm3_imap_ back`	IlistPerm	820
4.3.3.3	Lem. 4.31	89		`IlistPerm3_dec`	IlistPerm	1302
	Lem. 4.32	89		`extroduce_ IlistPerm4`	IlistPerm	1134
	Lem. 4.33	89		`exists_eq_Ilist`	IlistPerm	1197
4.3.3.4	Lem. 4.34	91		`IlistPerm3_mon`	IlistPerm	1052
4.3.4	Déf. 4.21	91	*skel_type*	`IlistPerm3Cert_list`	IlistPerm	1314
	Lem. 4.35	92		`IlistPerm3Cert_aux2`	IlistPerm	1319
	Déf. 4.22	92	*convSkel*	`rewriteIlistPerm3 Cert_list`	IlistPerm	1327
	Déf. 4.23	92	*skel_type_aux*	`IlistPerm3Cert_aux3`	IlistPerm	1342
	Lem. 4.36	92		`rewriteIlistPerm3 Cert_list_proofirr`	IlistPerm	1360
	Lem. 4.37	93		`IlistPerm3Cert_ aux3_proofirr`	IlistPerm	1401

Dans la thèse				Dans les scripts		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
	Déf. 4.24	93	*iperm_ind_skel*	`IlistPerm3Cert`	IlistPerm	1415
	Lem. 4.38	93		`IlistPerm3Cert_ proofirr`	IlistPerm	1458
	Lem. 4.39	93	$\Leftarrow$	`IlistPerm3_ IlistPerm3Cert`	IlistPerm	1434
	Lem. 4.39	93	$\Rightarrow$	`IlistPerm3Cert_ IlistPerm3`	IlistPerm	1425
	Lem. 4.40	94		`IlistPerm3Cert_mon`	IlistPerm	1447
	Lem. 4.41	95		`IlistPerm3Cert_ inter`	IlistPerm	1470
	Déf. 4.25	95	*iperm_bij*	`IlistPerm7`	IlistPerm	1787
	Lem. 4.42	95		`IlistPerm7_refl`	IlistPerm	1790
	Lem. 4.43	95		`IlistPerm7_sym`	IlistPerm	1795
4.4	Lem. 4.44	96		`IlistPerm7_trans`	IlistPerm	1807
	Lem. 4.45	96		`IlistPerm7Rel`	IlistPerm	1819
	Lem. 4.46	96		`IlistPerm7_lgti`	IlistPerm	1834
	Lem. 4.48	96		`IlistPerm7_mon`	IlistPerm	1825
	Thm. 4.2	96		`IlistPerm3_ IlistPerm7`	IlistPerm	1848
	Déf. 4.26	97	*skel_type_fun*	`IlistPerm3Cert_ list_function`	IlistPerm	1534
4.5	Déf. 4.27	97	*skel_type_inv*	`IlistPerm3Cert_ list_inv`	IlistPerm	1545
	Lem. 4.49	97		`IlistPerm3Cert_ list_inv_inv_id`	IlistPerm	1556
	Lem. 4.50	97		`IlistPerm3Cert_ list_function_inv`	IlistPerm	1553
	Lem. 4.51	100		`equiv_list_permut_ IlistPerm3`	EquivCont	30
4.6	Lem. 4.52	102		`equiv_ilist_ IlistPerm3_permut`	EquivCont	87
	Lem. 4.53	103		`equiv_list_ IlistPerm3_permut'`	EquivCont	119
	Lem. 4.54	103		`equiv_ilist_permut_ IlistPerm3`	EquivCont	129
4.8		105	Permutations sur les listes		PermsLists	
	Déf. 5.1	109	*Graph*	`Graph`	Graphs	27
	Déf. 5.2	109	*applyF2G*	`applyF2G`	Graphs	127
5.1		109	*label*	`label`	Graphs	32
		109	*sons*	`sons`	Graphs	33
	Lem. 5.1	109		`label_sons_OK`	Graphs	36
	Déf. 5.3	110	*Geq*	`Geq`	Graphs	49
5.2	Lem. 5.2	110		`finite_example_Geq_ finite_example_ unfolded`	Graphs	650

Dans la thèse				Dans les scripts		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
	Lem. 5.3	111		`Geq_refl`	Graphs	54
	Lem. 5.4	111		`Geq_sym`	Graphs	83
	Lem. 5.5	112		`Geq_trans`	Graphs	65
	Lem. 5.6	112		`GeqRel`	Graphs	99
5.3.1.1	Déf. 5.4	113	*GinG*	`Graph_in_Graph`	Graphs	157
	Lem. 5.7	113		`Graph_in_GraphM`	Graphs	163
	Lem. 5.8	113		`Graph_in_GraphM2`	Graphs	175
	Lem. 5.9	114		`GinG_sons_in_Graph`	Graphs	194
	Lem. 5.10	114		`Graph_in_Graph_trans`	Graphs	205
5.3.1.2	Déf. 5.5	114	*GinG**	`Graph_in_Graph_Gene`	Graphs	220
	Déf. 5.6	114	*GinG'*	`GinG'`	Graphs	247
	Lem. 5.11	114		`GinG_GinG'`	Graphs	249
5.3.2.1	Déf. 5.7	115	*isCycle*	`is_cycle`	Graphs	1086
	Déf. 5.8	115	*hasCycle*	`hasCycle`	Graphs	1096
	Déf. 5.9	115	*hasCycle'*	`hasCycle'`	Graphs	1229
	Lem. 5.12	115	$\Leftarrow$	`hasCycle_hasCycle'`	Graphs	1242
	Lem. 5.12	115	$\Rightarrow$	`hasCycle'_hasCycle`	Graphs	1261
5.3.2.2	Lem. 5.13	116		`JustOneLeaf_not_cycle`	Graphs	1108
	Lem. 5.14	116		`finite_example_has_cycle`	Graphs	1100
5.3.3.1	Déf. 5.10	117	*Gall*	`G_all`	Graphs	308
	Déf. 5.11	117	*element_of*	`P_Finite`	Graphs	312
	Déf. 5.12	117	*G_finite*	`G_finite`	Graphs	316
	Lem. 5.15	117		`G_all_G_in_G_P`	Graphs	466
	Lem. 5.16	117		`G_all_Geq_eq`	Graphs	360
	Lem. 5.17	117		`P_FiniteM`	Graphs	341
	Lem. 5.18	118		`G_Finite_Geq_eq`	Graphs	410
	Lem. 5.19	118		`P_Finite_monotone`	Graphs	380
	Lem. 5.20	118		`G_all_monotone`	Graphs	390
	Lem. 5.21	118		`G_all_P_Finite_monotone`	Graphs	400
	Lem. 5.22	118		`collectLists_G'`	Graphs	417
	Lem. 5.23	119		`GinG_finite_ci`	Graphs	444
	Lem. 5.24	119		`G_all_G_in_G_P`	Graphs	466
5.3.3.2	Lem. 5.25	119		`JustOneLeaf_finite`	Graphs	1114
	Lem. 5.26	120		`finite_example_finite`	Graphs	621
5.3.3.3	Lem. 5.27	121		`finite_bounded`	Graphs	513
	Déf. 5.13	121	*max_list_nat*	`max_list_nat`	Tools	91
	Lem. 5.28	121		`max_list_max`	Tools	137
	Lem. 5.29	121		`infinite_unbounded`	Graphs	575
	Lem. 5.30	121		`unbounded_infiniteGraph`	Graphs	614

Dans la thèse				Dans les scripts		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
	Lem. 5.31	121		`ilist_unbounded_infiniteGraph`	Graphs	598
	Lem. 5.32	121		`infinite_example_infinite`	Graphs	717
	Lem. 5.33	122		`infinite_graph_gene_Sn_in_n`	Graphs	682
	Lem. 5.34	122		`infinite_example_gene_n_inc_all`	Graphs	690
6.1.1	Déf. 6.1	126	*GPerm*	`GeqPerm0`	GPerm	306
6.1.2.1	Déf. 6.2	127	*GPerm_imp*	`GeqPerm`	GPerm	311
	Lem. 6.2	128		`GeqPerm_coind`	GPerm	321
	Lem. 6.3	128		`GeqPerm_out`	GPerm	331
	Lem. 6.4	128		`GeqPerm_intro`	GPerm	343
	Lem. 6.5	129		`GeqPerm0_GeqPerm`	GPerm	357
	Lem. 6.6	129		`GeqPerm_refl`	GPerm	641
	Lem. 6.7	129		`GeqPerm_sym`	GPerm	664
	Lem. 6.8	130		`GeqPerm_trans`	GPerm	690
	Lem. 6.9	130		`GeqPermRel`	GPerm	720
6.1.2.2	Déf. 6.3	130	*GPerm_mend*	`GeqPerm1`	GPerm	317
	Lem. 6.10	130	$\Leftarrow$	`GeqPerm_GeqPerm1`	GPerm	385
	Lem. 6.10	130	$\Rightarrow$	`GeqPerm1_GeqPerm`	GPerm	377
6.1.3.1	Déf. 6.4	131	*rpath*	`label_path`	Paths	25
	Déf. 6.5	131	*RelOp*	`RelOp`	Tools	248
		131	*RelOp* préserve l'équivalence	`RelOpRel`	Tools	280
	Déf. 6.6	132	*GeqPath*	`GeqPath`	Paths	78
	Lem. 6.11	132		`GeqPath_lgti`	Paths	139
	Lem. 6.12	132		`GeqPath_Geq`	Paths	198
6.1.3.2	Déf. 6.7	136	*iTree*	`TreeG`	GPerm	26
		136	*labeliT*	`labelT`	GPerm	29
		136	*sonsiT*	`sonsT`	GPerm	30
	Lem. 6.13	136		`labelT_sonsT_ok`	GPerm	32
	Déf. 6.8	136	*G2iT*	`Graph2TreeG`	GPerm	114
	Déf. 6.9	136	*TPerm*	`TeqPerm`	GPerm	155
	Lem. 6.14	136		`TeqPerm_refl`	GPerm	179
	Lem. 6.15	137		`TeqPerm_sym`	GPerm	194
	Lem. 6.16	137		`TeqPerm_trans`	GPerm	204
	Lem. 6.17	137		`TeqPermRel`	GPerm	219
	Déf. 6.10	138	$\equiv$	`TeqPermn`	GPerm	394
	Lem. 6.18	138		`TeqPermnRel`	GPerm	411
	Prop. 6.1.1	138		`TeqPermn_0`	GPerm	417
	Prop. 6.1.2	138	$\Leftarrow$	`TeqPermn_Sn_back`	GPerm	445
	Prop. 6.1.2	138	$\Rightarrow$	`TeqPermn_Sn`	GPerm	432
	Lem. 6.19	138		`TeqPermn_dec`	GPerm	487
	Lem. 6.20	138		`TeqPermn_Sn_n`	GPerm	462
	Lem. 6.21	138		`TeqPermn_antitone`	GPerm	478

Dans la thèse				Dans les scripts		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
	Déf. 6.11	139	*GTPerm*	`TeqPerm_gene`	GPerm	515
	Lem. 6.22	139		`TeqPerm_geneRel`	GPerm	537
	Thm. 6.1	139		`GeqPerm_TeqPerm`	GPerm	557
	Lem. 6.23	140		`TeqPerm_GeqPerm`	GPerm	615
	Ax. 1	141		Paramètre de `TeqPerm_GeqPerm`	GPerm	615
	Déf. 6.12	142	*DNE*	`DNE`	IPPJust	162
	Lem. 6.24	142		`DeMorganExists`	IPPJust	173
	Déf. 6.13	142	*IPPGen*	`IPPGen`	IPPJust	256
	Déf. 6.14	142	*IPPFin*	`IPPFin`	IPPJust	213
	Lem. 6.25	142		`DNEImpIPPFin`	IPPJust	216
	Lem. 6.26	143		`FunctionalChoiceFin`	IPPJust	137
	Déf. 6.15	143	*MaxFin*	`MaxFin'`	IPPJust	204
	Prop. 6.2	143		`MaxFin'Ok`	IPPJust	206
6.1.3.2	Lem. 6.27	144		`IlistPerm3Cert_list_bij_Fin`	IPPJust	463
		144	*FnmFnFm*	`FmFnFmn`	IPPJust	287
		144	*FnFmFnm*	`FmnFmFn`	IPPJust	305
	Prop. 6.3.1	145		`FmnFmFn_ok1`	IPPJust	315
	Prop. 6.3.2	145		`FmnFmFn_ok2`	IPPJust	329
	Prop. 6.3.3	145		`FmFnFmn_ok1`	IPPJust	343
	Prop. 6.3.4	145		`FmFnFmn_ok2`	IPPJust	355
	Prop. 6.3.5	145		`Fin_bij_mult`	IPPJust	419
	Lem. 6.28	145		`IPPGen_bij`	IPPJust	264
	Lem. 6.29	145		`IPPJustification`	IPPJust	503
	Déf. 6.16	145	*TiersEx*	`excluded_middle`	stdlib	
	Lem. 6.30	146		`ExclMiddleImpDNE`	IPPJust	164
	Lem. 6.31	146		`IPPJustification'`	IPPJust	516
	Déf. 6.17	146	*GPerm_bij*	`GeqPerm2`	GPermBij	28
	Lem. 6.32	146		`GeqPerm2_refl`	GPermBij	32
	Lem. 6.33	147		`GeqPerm2_sym`	GPermBij	43
	Lem. 6.34	147		`GeqPerm2_trans`	GPermBij	57
	Lem. 6.35	148		`GeqPerm2Rel`	GPermBij	72
	Déf. 6.18	148	*GPerm_bij_mend*	`GeqPerm1'`	GPermBij	79
6.1.4	Lem. 6.36	148	$\Leftarrow$	`GeqPerm2_GeqPerm1'`	GPermBij	101
	Lem. 6.36	148	$\Rightarrow$	`GeqPerm1'_GeqPerm2`	GPermBij	90
	Lem. 6.37	148	$\Leftarrow$	`GeqPerm1_GeqPerm1'`	GPermBij	132
	Lem. 6.37	148	$\Rightarrow$	`GeqPerm1'_GeqPerm1`	GPermBij	122
	Lem. 6.38	148	$\Leftarrow$	`GeqPerm2_GeqPerm`	GPermBij	148
	Lem. 6.38	148	$\Rightarrow$	`GeqPerm_GeqPerm2`	GPermBij	142
	Lem. 6.39	149		`GeqPerm0_GeqPerm2`	GPermBij	117
	Déf. 6.19	150	*GinGP*	`Graph_in_Graph_Perm`	GPerm	725
6.2.1	Lem. 6.40	150		`GinGP_sons`	GPerm	727
	Lem. 6.41	151		`Graph_in_Graph_Perm_trans`	GPerm	741

Dans la thèse				Dans les scripts		
Section	**Référence**	**p.**	**Nom**	**Nom**	**Fichier**	**l.**
6.2.2	Déf. 6.20	151	*GeqPerm*	`GPPerm`	GPerm	791
	Lem. 6.42	151		`GPPerm_refl`	GPerm	794
	Lem. 6.43	151		`GPPerm_sym`	GPerm	799
	Lem. 6.44	151		`GPPerm_trans`	GPerm	806
	Lem. 6.45	151		`GPPermRel`	GPerm	815
6.2.3.1	Lem. 6.46	152		`GPPerm_g012_g021`	GPerm	824
6.2.3.2	Lem. 6.47	153		`GPPerm_G01'_G10'`	GPerm	867
6.2.3.3	Lem. 6.48	153		`not_GPPerm_g3_g4`	GPerm	981
6.3.1	Déf. 6.21	154	*AllGraph*	`AllGraph`	allGraphs	29
	Déf. 6.22	155	*AGeq*	`AGeq`	allGraphs	31
	Déf. 6.23	155	*G2AG*	`G2AG`	allGraphs	65
6.3.2	Déf. 6.24	156	*ForestGraph*	`ForestGr`	allGraphs	63
	Déf. 6.25	156	*FGeq*	`FGeq`	allGraphs	69
A.1.1.10	Lem. A.1	169		`nth_indep`	stdlib	
A.1.1.12	Lem. A.2	170		`app_nth1`	stdlib	
	Lem. A.3	170		`app_nth2`	stdlib	
A.2.1.5	Lem. A.4	191		`hasCycle'_sons`	Graphs	1232
A.2.2.2	Lem. A.5	196		`label_path_inf_n_rel`	Paths	119
	Lem. A.6	196		`label_path_inf_n_rel_sym`	Paths	129

Bibliographie

[1] Michael ABBOTT, Thorsten ALTENKIRCH, Neil GHANI et Conor MCBRIDE : Constructing polymorphic programs with quotient types. *In* Dexter KOZEN et Carron SHANKLAND, éditeurs : *MPC*, volume 3125 de *Lecture Notes in Computer Science*, pages 2–15. Springer, 2004.
Cité pages 69 et 162.

[2] Andreas ABEL : *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. Thèse de doctorat, Ludwig-Maximilians-Universität München, 2006.
Cité page 36.

[3] Jean-Raymond ABRIAL : *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.
Cité page 1.

[4] Jirí ADÁMEK, Filippo BONCHI, Mathias HÜLSBUSCH, Barbara KÖNIG, Stefan MILIUS et Alexandra SILVA : A coalgebraic perspective on minimization and determinization. *In Proceedings of the Fifteenth International Conference on Foundations of Software Science and Computation structures (FoSSaCS 2012)*, Lecture Notes in Computer Science, 2012.
Cité page 161.

[5] Agda. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.
Cité page 1.

[6] Thorsten ALTENKIRCH : A formalization of the strong normalization proof for system F in LEGO. *In* BEZEM et GROOTE [17], pages 13–28.
Cité page 44.

[7] Emilie BALLAND et Pierre-Etienne MOREAU : Term-graph rewriting via explicit paths. *In* Andrei VORONKOV, éditeur : *RTA : International Conference on Rewriting Techniques and Applications RTA Lecture Notes in Computer Science*, volume 5117 de *Lecture Notes in Computer Science*, pages 32–47, Hagenberg Autriche, 2008. Springer.
Cité pages 123 et 161.

[8] Henk BARENDREGT et Tobias NIPKOW, éditeurs. *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 de *Lecture Notes in Computer Science*. Springer, 1994.
Cité pages 217 et 219.

[9] Falk BARTELS : Generalised coinduction. *Mathematical Structures in Computer Science*, 13(2):321–348, 2003.
Cité page 37.

[10] Gilles BARTHE, Maria João FRADE, E. GIMÉNEZ, Luis PINTO et Tarmo UUSTALU : Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.
Cité page 36.

[11] Stefano BERARDI, Ferruccio DAMIANI et Ugo DE'LIGUORO, éditeurs. *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, volume 5497 de *Lecture Notes in Computer Science*. Springer, 2009.
Cité pages 216 et 220.

[12] Ulrich BERGER : From coinductive proofs to exact real arithmetic : theory and applications. *Logical Methods in Computer Science*, 7(1), 2011.
Cité page 162.

[13] Yves BERTOT : Filters on coinductive streams, an application to Eratosthenes' sieve. *In* Pawel URZYCZYN, éditeur : *TLCA*, volume 3461 de *Lecture Notes in Computer Science*, pages 102–115. Springer, 2005.
Cité page 26.

[14] Yves BERTOT et Pierre CASTÉRAN : *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004. `http://www.labri.fr/publications/l3a/2004/BC04`.
Cité page 12.

[15] Yves BERTOT et Ekaterina KOMENDANTSKAYA : Inductive and coinductive components of corecursive functions in Coq. *CoRR*, abs/0807.1524, 2008.
Cité page 123.

[16] Yves BERTOT et Ekaterina KOMENDANTSKAYA : Using structural recursion for corecursion. *In* BERARDI *et al.* [11], pages 220–236.
Cité pages xii, 26, 27, 36 et 43.

[17] Marc BEZEM et Jan Friso GROOTE, éditeurs. *Typed Lambda Calculi and Applications, International Conference, TLCA 1993, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 de *Lecture Notes in Computer Science*. Springer, 1993.
Cité pages 215 et 220.

[18] Richard S. BIRD : Maximum marking problems. *J. Funct. Program.*, 11(4):411–424, 2001.
Cité page 30.

[19] Sylvain BOULMÉ : Specifying in Coq inheritance used in computer algebra. Research report, LIP6, 2000. `http://www.lip6.fr/reports/lip6.2000.013.html`.
Cité page 162.

[20] Achim D. BRUCKER et Burkhart WOLFF : An extensible encoding of object-oriented data models in HOL. *Journal of Automated Reasoning*, 41(3-4):219–249, 2008.
Cité page 161.

[21] Adam CHLIPALA : Dans la discussion "is Coq being too conservative ?" du Coq club, janvier 2010. `https://sympa.inria.fr/sympa/arc/coq-club/2010-01/msg00089.html`.
Cité page 43.

[22] CIRC. http://fsl.cs.uiuc.edu/index.php/Circ.
Cité page 12.

[23] Edmund M. CLARKE, Orna GRUMBERG et Doron PELED : *Model checking*. MIT Press, 2001.
Cité page 1.

[24] Evelyne CONTEJEAN : Modeling permutations in Coq for Coccinelle. *In* Hubert COMON-LUNDH, Claude KIRCHNER et Hélène KIRCHNER, éditeurs : *Rewriting, Computation and Proof*, volume 4600 de *Lecture Notes in Computer Science*, pages 259–269. Springer, 2007.
Cité pages 71 et 105.

[25] Coq. http://coq.inria.fr/.
Cité page 1.

[26] Thierry COQUAND : Infinite objects in type theory. *In* BARENDREGT et NIPKOW [8], pages 62–78.
Cité pages 22 et 27.

[27] Thierry COQUAND et Gérard P. HUET : The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.
Cité page 12.

[28] Bruno COURCELLE : Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983.
Cité pages xi, xii, 29 et 33.

[29] Chris DAMS : Dans la discussion "is Coq being too conservative ?" du Coq club, janvier 2010. https://sympa.inria.fr/sympa/arc/coq-club/2010-01/msg00085.html.
Cité pages xii et 37.

[30] Nils Anders DANIELSSON : Beating the productivity checker using embedded languages. *In* Ana BOVE, Ekaterina KOMENDANTSKAYA et Milad NIQUI, éditeurs : *PAR*, volume 43 de *EPTCS*, pages 29–48, 2010.
Cité pages xii et 40.

[31] Nils Anders DANIELSSON et Thorsten ALTENKIRCH : Subtyping, declaratively. *In* Claude BOLDUC, Jules DESHARNAIS et Béchir KTARI, éditeurs : *Mathematics of Program Construction (MPC'10)*, volume 6120 de *LNCS*, pages 100–118. Springer, 2010.
Cité pages xii et 40.

[32] Equipe de développement de COQ : Bibliothèque standard de l'assistant de preuve Coq. http://coq.inria.fr/stdlib/.
Cité pages xiii, 69, 70 et 71.

[33] Equipe de développement de COQ : FAQ de l'assistant de preuve Coq. http://coq.inria.fr/V8.1/faq.html.
Cité page 127.

[34] Equipe de développement de COQ : Manuel de référence de l'assistant de preuve Coq. http://coq.inria.fr.
Cité pages 1, 18 et 44.

[35] Edsger W. DIJKSTRA : On the productivity of recursive definitions. circulated privately, http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD749.PDF, 1980.
Cité page 26.

[36] Métamodèle Ecore. http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/package-summary.html.
Cité page 162.

[37] Abbas EDALAT et Peter John POTTS : A new representation for exact real numbers. *Electr. Notes Theor. Comput. Sci.*, 6:119–132, 1997.
Cité page 26.

[38] Martin ERWIG : Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, 2001.
Cité pages xii, 31, 32 et 33.

[39] Herman GEUVERS : Inductive and coinductive types with iteration and recursion. *In Proceedings of the Workshop on Types for Proofs and Programs, Bastad*, pages 193–217, 1992.
Cité pages 39 et 127.

[40] Pietro Di GIANANTONIO et Marino MICULAN : A unifying approach to recursive and co-recursive definitions. *In* Herman GEUVERS et Freek WIEDIJK, éditeurs : *TYPES*, volume 2646 de *Lecture Notes in Computer Science*, pages 148–161. Springer, 2002.
Cité pages 26 et 35.

[41] Pietro Di GIANANTONIO et Marino MICULAN : Unifying recursive and co-recursive definitions in sheaf categories. *In* Igor WALUKIEWICZ, éditeur : *FoSSaCS*, volume 2987 de *Lecture Notes in Computer Science*, pages 136–150. Springer, 2004.
Cité page 35.

[42] Eduardo GIMÉNEZ : Codifying guarded definitions with recursive schemes. *In* Peter DYBJER, Bengt NORDSTRÖM et Jan M. SMITH, éditeurs : *TYPES*, volume 996 de *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.
Cité page 26.

[43] Eduardo GIMÉNEZ : *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1996.
Cité pages 2 et 22.

[44] Eduardo GIMÉNEZ et Pierre CASTÉRAN : A tutorial on [co-]inductive types in Coq. http://www.labri.fr/perso/casteran/RecTutorial.pdf, 2007.
Cité page 27.

[45] Lars HALLNÄS : An intensional characterization of the largest bisimulation. *Theor. Comput. Sci.*, 53:335–343, 1987.
Cité page 37.

[46] Peter HANCOCK et Anton SETZER : Guarded induction and weakly final coalgebras in dependent type theory. *In* L. CROSILLA et P. SCHUSTER, éditeurs : *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, pages 115 – 134, Oxford, 2005. Clarendon Press.
Cité page 37.

[47] Jason J. HICKEY : Formal objects in type theory using very dependent types. *In* Kim BRUCE et Giuseppe LONGO, éditeurs : *Informal proceedings of Third Workshop on Foundations of Object-Oriented Languages (FOOL 3)*, 1999.
Cité page 161.

[48] John HOPCROFT : An $n \log n$ algorithm for minimizing states in a finite automaton. *In Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971)*, pages 189–196. Academic Press, New York, 1971.
Cité page 161.

[49] Bart JACOBS et Jan RUTTEN : A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.
Cité page 37.

[50] Nils KLARLUND et Michael I. SCHWARTZBACH : Graph types. *In In Proc. 20th ACM POPL*, pages 196–205. ACM Press, 1993.
Cité pages 123 et 161.

[51] Alexander KURZ, Marina LENISA et Andrzej TARLECKI, éditeurs. *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*, volume 5728 de *Lecture Notes in Computer Science*. Springer, 2009.
Cité pages 219 et 221.

[52] François LECLERC et Christine PAULIN-MOHRING : Programming with streams in Coq - a case study : the sieve of Eratosthenes. *In* BARENDREGT et NIPKOW [8], pages 191–212.
Cité page 127.

[53] Xavier LEROY : A formally verified compiler back-end. *CoRR*, abs/0902.2137, 2009.
Cité page 2.

[54] Dorel LUCANU, Eugen-Ioan GORIAC, Georgiana CALTAIS et Grigore ROSU : Circ : a behavioral verification tool based on circular coinduction. *In* KURZ *et al.* [51], pages 433–442.
Cité page 12.

[55] Dorel LUCANU et Grigore ROSU : Circ : a circular coinductive prover. *In* Till MOSSAKOWSKI, Ugo MONTANARI et Magne HAVERAAEN, éditeurs : *CALCO*, volume 4624 de *Lecture Notes in Computer Science*, pages 372–378. Springer, 2007.
Cité page 12.

[56] Ralph MATTHES : Tarski's fixed-point theorem and lambda calculi with monotone inductive types. *Synthese*, 133:107–129, 2002. `http://dx.doi.org/10.1023/A:1020831825964`.
Cité page 8.

[57] Maude. `http://maude.cs.uiuc.edu/download/`.
Cité page 12.

[58] Conor MCBRIDE et James MCKINNA : The view from the left. *J. Funct. Program.*, 14(1): 69–111, 2004.
Cité page 44.

[59] N. P. MENDLER : Recursive types and type constraints in second-order lambda calculus. *In LICS*, pages 30–36. IEEE Computer Society, 1987.
Cité page 36.

[60] N. P. MENDLER : Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic*, 51(1-2):159–172, 1991.
Cité page 36.

[61] Keiko NAKATA et Tarmo UUSTALU : Resumptions, weak bisimilarity and big-step semantics for while with interactive i/o : an exercise in mixed induction-coinduction. *In* Luca ACETO et Pawel SOBOCINSKI, éditeurs : *SOS*, volume 32 de *EPTCS*, pages 57–75, 2010.
Cité pages xii, 39 et 130.

[62] Keiko NAKATA, Tarmo UUSTALU et Marc BEZEM : A proof pearl with the fan theorem and bar induction - walking through infinite trees with mixed induction and coinduction. *In* Hongseok YANG, éditeur : *APLAS*, volume 7078 de *LNCS*, pages 353–368. Springer, 2011.
Cité page 10.

[63] Wolfgang NARASCHEWSKI et Markus WENZEL : Object-oriented verification based on record subtyping in higher-order logic. *In Theorem Proving in Higher Order Logics*, 1998.
Cité page 161.

[64] Tobias NIPKOW, Lawrence PAULSON et Markus WENZEL : *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002. `http://isabelle.in.tum.de`.
Cité page 1.

[65] Milad NIQUI : Coinductive field of exact real numbers and general corecursion. *Electr. Notes Theor. Comput. Sci.*, 164(1):121–139, 2006.
Cité page 26.

[66] Milad NIQUI : Coalgebraic reasoning in Coq : bisimulation and the lambda-coiteration scheme. *In* BERARDI *et al.* [11], pages 272–288.
Cité pages xii, 37 et 162.

[67] Christine PAULIN-MOHRING : Inductive definitions in the system Coq - rules and properties. *In* BEZEM et GROOTE [17], pages 328–345.
Cité page 13.

[68] Christine PAULIN-MOHRING : Circuits as streams in Coq : verification of a sequential multiplier. *In* Stefano BERARDI et Mario COPPO, éditeurs : *TYPES*, volume 1158 de *Lecture Notes in Computer Science*, pages 216–230. Springer, 1995.
Cité page 127.

[69] Celia PICARD et Ralph MATTHES : Coinductive graph representation : the problem of embedded lists. *In* Rachid ECHAHED, Annegret HABEL et Mohamed MOSBAH, éditeurs : *Graph Computation Models, Enschede - The Netherlands*, 2010.
Cité page 160.

[70] Celia PICARD et Ralph MATTHES : Coinductive graph representation : the problem of embedded lists. *ECEASST*, 39, 2011.
Cité page 160.

[71] Celia PICARD et Ralph MATTHES : Formalisation en Coq, 2012. http://www.irit.fr/~Celia.Picard/These/.
Cité page 3.

[72] Celia PICARD et Ralph MATTHES : Permutations in Coinductive Graph Representation. *In* Dirk PATTINSON et Lutz SCHRÖDER, éditeurs : *Coalgebraic Methods in Computer Science, Tallinn, Estonie*, volume 7399 de *Lecture Notes In Computer Science*. Springer, 2012. http://www.irit.fr/~Celia.Picard/Papers/Permutations.pdf.
Cité page 160.

[73] Detlef PLUMP, Robin SURI et Ambuj SINGH : Minimizing finite automata with graph programs. *ECEASST*, 39, 2011.
Cité page 161.

[74] Iman POERNOMO : Proofs-as-model-transformations. *In* Antonio VALLECILLO, Jeff GRAY et Alfonso PIERANTONIO, éditeurs : *International Conference on Model Transformation, ICMT 2008*, volume 5063 de *Lecture Notes in Computer Science*, pages 214–228. Springer, 2008.
Cité page 162.

[75] Rawle PRINCE, Neil GHANI et Conor MCBRIDE : Proving properties about lists using containers. *In* Jacques GARRIGUE et Manuel V. HERMENEGILDO, éditeurs : *FLOPS*, volume 4989 de *Lecture Notes in Computer Science*, pages 97–112. Springer, 2008.
Cité pages 3 et 43.

[76] Grigore ROSU et Dorel LUCANU : Circular coinduction : a proof theoretical foundation. *In* KURZ *et al.* [51], pages 127–144.
Cité page 12.

[77] Ben A. SIJTSMA : On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst.*, 11(4):633–649, 1989.
Cité page 26.

[78] Matthieu SOZEAU : *Coq 8.2 Reference Manual*, chapitre User defined equalities and relations. INRIA TypiCal, 2008.
Cité pages xi et 22.

[79] Ssreflect. http://www.msr-inria.inria.fr/Projects/math-components.
Cité page 45.

[80] Alfred TARSKI : A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.
Cité page 8.

[81] Tarmo UUSTALU et Varmo VENE : Least and greatest fixed points in intuitionistic natural deduction. *Theoretical Computer Science*, 272:315–339, 2002.
Cité pages xii et 39.

[82] G. C. WRAITH : A note on categorical datatypes. *In* David H. PITT, David E. RYDEHEARD, Peter DYBJER, Andrew M. PITTS et Axel POIGNÉ, éditeurs : *Category Theory and Computer Science*, volume 389 de *Lecture Notes in Computer Science*, pages 118–127. Springer, 1989.
Cité page 127.

Liste des figures

Index

Celia Picard

COINDUCTIVE GRAPH REPRESENTATION

Thesis Advisor : Ralph Matthes, *C.N.R.S.*
PhD defended June 15, 2012 at IRIT - Université Paul Sabatier

Abstract

We are interested in graph representation in the theorem prover Coq. We have chosen to represent graphs using coinductive types. We wanted to explore their use in Coq. Indeed, they make the graph representation succinct and elegant. Moreover, navigability is ensured by construction. We had to overcome the guardedness condition whose objective is to ensure validity of all operations made on coinductive objects. Its implementation in Coq is restrictive and sometimes forbids definitions, even semantically correct ones. A canonical formalization of graphs thus surmounts Coq's direct expressivity. We have designed a solution respecting these limitations. Then, we have defined a relation on graphs close to the notion of equivalence obtained on a classical representation, keeping however the advantages offered by coinduction. We show that this relation is equivalent to another one based on finite observations of the graphs.

Keywords : coinduction, theorem prover, graphs, lists, mixing induction and coinduction.

Informatique - Sûreté du logiciel et calcul de haute performance

---

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31062 TOULOUSE cedex 4

Celia Picard

REPRÉSENTATION COINDUCTIVE DES GRAPHES

Directeur de thèse : Ralph Matthes, *CNRS*
Thèse soutenue le 15 juin 2012 à l'IRIT - Université Paul Sabatier

Résumé

Nous nous intéressons à la représentation de graphes dans le prouveur Coq. Nous avons choisi de les représenter par des types coinductifs dont nous voulions explorer l'utilisation. Ceux-ci permettent de rendre succincte et élégante la représentation et d'obtenir la navigabilité par construction. Nous avons dû contourner la condition de garde dont le but est d'assurer la validité des opérations effectuées sur les objets coinductifs. Son implantation dans Coq est restrictive et interdit parfois des définitions sémantiquement correctes. Une formalisation canonique des graphes dépasse ainsi l'expressivité directe de Coq. Nous avons donc proposé une solution respectant ces limitations, puis nous avons défini une relation sur les graphes nous permettant d'obtenir la même notion d'équivalence qu'avec une représentation classique tout en gardant les avantages de la coinduction. Nous montrons qu'elle est équivalente à une relation basée sur des observations finies.

Mots-clés : coinduction, prouveur de théorèmes, graphes, listes, mélange induction et coinduction.

Informatique - Sûreté du logiciel et calcul de haute performance

---

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31062 TOULOUSE cedex 4