UNIVERSITÉ PARIS 7 - DENIS DIDEROT – UFR D'INFORMATIQUE – ANNÉE 2002

Une étude des sommes fortes : isomorphismes et formes normales

THÈSE

pour l'obtention du diplôme de

docteur de l'université Paris 7, spécialité informatique

présentée et soutenue publiquement par

Vincent Balat

le 5 décembre 2002

DIRECTEUR DE THÈSE

Roberto Di Cosmo

JURY

M. Guy Cousineau	*président*
M. Andrea Asperti	*rapporteur*
M. Thomas Ehrhard	*rapporteur*
M. Roberto Di Cosmo	*directeur de thèse*
M. Olivier Danvy	
M. Marcelo Fiore	
M. Didier Galmiche	
M. Neil Jones	

version du 7 août 2003

Remerciements

Cette thèse est l'aboutissement d'années de travail riches en rebondissements, pleines de découvertes, parfois inattendues, mais aussi de moments de doutes. Roberto Di Cosmo m'a fait l'honneur d'accepter de diriger cette thèse, et a su me guider scientifiquement et me faire faire des rencontres très enrichissantes. J'ai beaucoup apprécié le fait de pouvoir travailler en étroite collaboration avec lui. Je voudrais le remercier aussi pour son amitié et pour tout ce qu'il m'a fait découvrir en dehors du travail de thèse, notamment pour la fabuleuse aventure DemoLinux.

Je tiens à souligner également le plaisir que j'ai eu a travailler avec Marcelo Fiore et Olivier Danvy. La très agréable collaboration avec chacun d'entre eux a été très fructueuse scientifiquement et a beaucoup influé sur l'orientation et les principaux résultats de ma thèse.

Je garde un excellent souvenir de mon stage de DEA avec Didier Galmiche et je suis heureux qu'il accepte d'être membre de mon jury. J'espère pouvoir collaborer de nouveau avec lui.

Je remercie Andrea Asperti et Thomas Ehrhard d'avoir accepté le lourd travail de rapporteurs, Guy Cousineau qui préside mon jury, et enfin Neil Jones, qui me fait l'honneur de venir de Copenhague pour ma soutenance.

Je voudrais également citer Xavier Leroy, Didier Rémy, Andrzej Filinski, Claude Kirchner et Sergei Soloviev avec qui j'ai eu quelques discussions très utiles.

Pendant ces années de thèse, j'ai eu la chance de pouvoir travailler dans des environnements scientifiques des plus prestigieux, d'abord au laboratoire d'informatique de l'École Normale Supérieure, puis au laboratoire *Preuves, Programmes et Systèmes* de l'université Paris 7. Je remercie tous leurs membres et les invités qui ont contribué à une vie scientifique riche, en particulier Pierre-Louis Curien.

Les thésards et chercheurs de PPS ont largement contribué à rendre ces quelques années agréables. J'ai beaucoup apprécié la compagnie des thésards du bureau 6C10, Jean-Vincent Loddo, Sylvain Baro, Emmanuel Polonovski et Anne-Gwenn Bosser, les discussions intéressantes avec Olivier Laurent, l'amitié des voisins d'en face, Raphaël Montelatici, Michel Hirschowitz et François Maurel, de Francisco Alberti, et de beaucoup d'autres membres du laboratoire dont je devrais citer le nom ici... Je tiens également à remercier Joëlle Isnard, Noëlle Delgado et Odile Ainardi pour leur sympathie et leur efficacité.

Je ne peux pas terminer cette page sans mentionner les noms de mes amis qui m'ont aidé dans les moments difficiles. Je tiens à exprimer un remerciement particulier à Sébastien Seguineau qui a été mon plus fidèle soutien et dont l'amitié m'est très chère. Merci aussi au « 3G » de l'École Centrale de m'avoir adopté, merci à Luc Jolivet et Sébastien Grellier. Merci enfin à mes parents, pour leur relecture et pour tout le reste.

Résumé

Le but de cette thèse est d'étudier la somme et le zéro dans deux principaux cadres : les isomorphismes de types et la normalisation de λ-termes. Les isomorphismes de type avaient déjà été étudiés dans le cadre du λ-calcul simplement typé avec paires surjectives mais sans somme. Pour aborder le cas avec somme et zéro, j'ai commencé par restreindre l'étude au cas des isomorphismes linéaires, dans le cadre de la logique linéaire, ce qui a conduit à une caractérisation remarquablement simple de ces isomorphismes, obtenue grâce à une méthode syntaxique sur les réseaux de preuve.

Le cadre plus général de la logique intuitionniste correspond au problème ouvert de la caractérisation des isomorphismes dans les catégories bi-cartésiennes fermées. J'ai pu apporter une contribution à cette étude en montrant qu'il n'y a pas d'axiomatisation finie de ces isomorphismes. Pour cela, j'ai tiré partie de travaux en théorie des nombres portant sur un problème énoncé par Alfred Tarski et connu sous le nom du « problème des égalités du lycée ».

Pendant tout ce travail sur les isomorphismes de types, s'est posé le problème de trouver une forme canonique pour représenter les λ-termes, que ce soit dans le but de nier l'existence d'un isomorphisme par une étude de cas sur la forme du terme, ou pour vérifier leur existence dans le cas des fonctions très complexes que j'étais amené à manipuler. Cette réflexion a abouti à poser une définition « extensionnelle » de forme normale pour le λ-calcul avec somme et zéro, obtenue par des méthodes catégoriques grâce aux relations logiques de Grothendieck, apportant ainsi une nouvelle avancée dans l'étude de la question réputée difficile de la normalisation de ce λ-calcul.

Enfin je montrerai comment il est possible d'obtenir une version « intentionnelle » de ce résultat en utilisant la normalisation par évaluation. J'ai pu ainsi donner une adaptation de la technique d'*évaluation partielle dirigée par les types* pour qu'elle produise un résultat dans cette forme normale, ce qui en réduit considérablement la taille et diminue aussi beaucoup le temps de normalisation dans le cas des isomorphismes de types considérés auparavant.

Mots-clés

λ-calcul, types, catégories, isomorphismes, somme, co-produit, zéro, objet initial, formes normales, modèles, logique linéaire multiplicative, égalités arithmétiques, problème des égalités du lycée de Tarski, relations logiques de Grothendieck, normalisation par évaluation, évaluation partielle dirigée par les types, opérateurs de contrôle, *Objective Caml*

Abstract

A study of strong sums: isomorphisms and normal forms

The goal of this thesis is to study the sum and the zero within two principal frameworks: type isomorphisms and the normalization of λ-terms. Type isomorphisms have already been studied within the framework of the simply typed λ-calculus with surjective pairing but without sums. To handle the case with sums and zero, I first restricted the study to the case of linear isomorphisms, within the framework of linear logic, which led to a remarkably simple characterization of these isomorphisms, obtained thanks to a syntactic method on proof-nets.

The more general framework of intuitionistic logic corresponds to the open problem of characterizing isomorphisms in bi-cartesian closed categories. I contributed to this study by showing that there is no finite axiomatization of these isomorphisms. To achieve this, I used some results in number theory regarding Alfred Tarski's so-called "high school algebra" problem.

The whole of this work brought about the problem of finding a canonical form to represent λ-terms, with the aim of either denying the existence of an isomorphism by a case study on the form of the term, or checking their existence in the case of the very complex functions I was brought to handle. This analysis led us to give an "extensional" definition of normal form for the λ-calculus with sums and zero, obtained by categorical methods using Grothendieck logical relations.

Finally, I could obtain an "intentional" version of this result by using normalization by evaluation. By adapting the technique of *type-directed partial evaluation*, it is possible to produce a result in the new normal form, reducing considerably its size in the case of the type isomorphisms considered before.

Keywords

λ-calculus, types, categories, isomorphisms, sum, co-product, zero, initial object, normal forms, models, multiplicative linear logic, arithmetic equalities, Tarski high school algebra problem, Grothendieck logical relations, normalisation by evaluation, type directed partial evaluation, control operators, *Objective Caml*

Table des matières

Résumé **5**

Abstract **7**

Introduction **13**

I Modèles du λ-calcul 23

1 Introduction aux isomorphismes dans les catégories **25**
1.1 Définitions de base . . . 25
1.2 Catégories bi-cartésiennes fermées . . . 27
1.2.1 Constructions de base . . . 27
1.2.2 Limites . . . 29
1.3 Foncteurs et catégories de foncteurs . . . 31
1.3.1 Définitions . . . 31
1.3.2 Pré-faisceaux . . . 32
1.3.3 Catégories bi-cartésiennes fermées et transformations naturelles . . . 34
1.4 Isomorphismes dans les biCCC . . . 35

2 Modèles du λ-calcul **43**
2.1 Le λ-calcul avec type somme . . . 44
2.1.1 Le λ-calcul pur . . . 44
2.1.2 Le λ-calcul simplement typé . . . 46
2.2 Sémantique du λ-calcul simplement typé . . . 53
2.2.1 Modèle des λ-termes . . . 54
2.2.2 Les biCCC modèles du λ-calcul avec somme . . . 57
2.3 Isomorphisme entre $\Lambda_{\times+10\beta\eta}$ et la biCCC libre . . . 58
2.3.1 Sans co-produit . . . 58
2.3.2 Avec co-produit . . . 67
2.3.3 Note sur la règle d'élimination du 0 . . . 72

II Isomorphismes de types avec somme 75

3 Isomorphismes linéaires en logique linéaire **77**
3.1 La logique linéaire . . . 78
3.1.1 Les règles de la logique linéaire . . . 78
3.1.2 Les réseaux de preuves . . . 79

3.2 Définitions et présentation des résultats 80
3.3 Réduction aux réseaux de preuves simples bipartites 82
3.4 Réduction à des formules non-ambiguës 84
3.5 Complétude pour les isomorphismes dans MLL 87
3.6 Prise en compte des constantes 89
3.6.1 Expansions des axiomes avec constantes : les réseaux simples identité revus et corrigés 90
3.6.2 Réduction aux isomorphismes entre formules simplifiées 90
3.6.3 Complétude avec constantes 90
3.7 Conclusions 91

4 Isomorphismes pour le λ-calcul avec type somme et type vide 93
4.1 Mise en relation des différentes théories 94
4.2 Le problème des égalités du lycée et les isomorphismes de types 95
4.2.1 Types produit et somme 98
4.2.2 Types produit, flèche, et somme 100
4.2.3 Type vide et type somme 102
4.3 Preuve du lemme 4.9 104
4.4 Remarques conclusives 108

III Normalisation en présence du type somme 111

5 Formes normales extensionnelles du λ-calcul avec somme 113
5.1 Survol de la preuve 114
5.2 Sémantique dans la biCCC des relations de Grothendieck 116
5.3 Normalisation dans les biCCC 121
5.3.1 Lemme de base 121
5.3.2 Termes neutres, termes normaux 123
5.3.3 Contextes contraints 127
5.3.4 Formes normales 131
5.4 Lemmes utilisés 133

6 Normalisation par évaluation du λ-calcul avec somme 139
6.1 Normalisation par évaluation 139
6.2 Évaluation partielle dirigée par les types 144
6.2.1 Introduction à l'évaluation partielle 144
6.2.2 De NBE à TDPE 144
6.2.3 TDPE et les types sommes 145
6.3 Implantation de TDPE en *Objective Caml* 146
6.3.1 Le problème 147
6.3.2 La solution de Filinski/Yang 148
6.3.3 La solution avec génération de bytecode 151
6.3.4 Normalisation *in situ* 152
6.4 Conclusion 153

7 Un normaliseur produisant des formes normales canoniques **155**
7.1 Le problème 155
7.2 Trois solutions pour répondre aux trois conditions 160
7.2.1 Condition ♠ 160
7.2.2 Condition ♣ 161
7.2.3 Condition ♢ 161
7.2.4 Les résultats 162
7.3 η-réduction 165
7.4 Insertion de `let` et mémoïsation 166

Conclusion **167**

Introduction

JUSTE APRÈS avoir appris à compter, à l'école primaire, les enfants apprennent l'addition. Ce n'est que bien après qu'est introduite la notion plus abstraite de multiplication. La somme et le produit réapparaissent ensuite sous de multiples formes dans l'étude des nombres à virgule, des nombres négatifs, des groupes et anneaux, des espaces vectoriels... En logique, ce sont respectivement les connecteurs « ou » et « et » ; en théorie des ensembles, l'union disjointe et le produit cartésien ; en théorie des catégories, le co-produit et le produit. En informatique aussi, les notions de somme et produit apparaissent dans les types ; par exemple en *C* ce sont les types `union` et `struct`. Mais très souvent, contrairement à l'école primaire, la somme est définie comme la notion « duale » du produit. Définie *après* le produit... Souvent même, dans les livres, dans les cours de DEA, on n'en parle pas, ou très peu. C'est la notion duale. Elle doit se comporter de la même façon...

Malheureusement ce n'est pas si simple. Dans bien des domaines, les systèmes avec somme posent des problèmes. En λ-calcul typé, la somme a de très mauvaises propriétés de confluence. Ce n'est que récemment qu'ont été prouvées deux propriétés fondamentales de la théorie équationnelle du λ-calcul avec somme, à savoir sa décidabilité, par Neil Ghani [47, 48], et sa complétude par rapport aux égalités de la catégorie des ensembles (Dougherty et Subrahmanyam [43]). En évaluation partielle dirigée par les types, nous verrons que la somme oblige à introduire des opérateurs de contrôle. Nous verrons aussi que même en arithmétique, elle n'a pas toujours des propriétés aussi simples que le produit.

La somme vient avec un objet étrange, son élément neutre, le zéro, l'ensemble vide, l'objet initial... Historiquement, il n'est apparu qu'au deuxième ou troisième siècle avant Jésus-Christ, à Babylone et en Grèce, puis fut adopté par les Arabes au huitième siècle, et en Europe au douzième siècle seulement ! Il n'est réellement devenu un nombre à part entière qu'au sixième siècle, en Inde. Cette notion aujourd'hui courante n'est donc pas si naturelle. En programmation, le zéro est un type mystérieux, quasiment jamais utilisé, qui n'a aucun élément... En logique, c'est une formule toujours fausse.

Nous allons étudier comment se comportent la somme et le zéro dans deux domaines précis liés plus particulièrement à l'informatique et à la logique, qui sont les isomorphismes de types et la normalisation du λ-calcul.

Le λ-calcul

Le λ-calcul est un modèle de représentation du calcul qui a donné naissance à un nouveau style de langages, dits fonctionnels (on peut citer *Lisp*, *Scheme*, *Haskell*, ou les dialectes de *ML* comme *SML* ou *Caml*). Sa syntaxe repose sur les notions de *variable*, d'*abstraction* ($\lambda x.\ t$) et d'*application* ($t\ \ u$). Intuitivement, si x est une variable et t et u des *λ-termes*, le terme $\lambda x.\ t$ re-

présente la fonction $x \mapsto t$ et l'application $(t \quad u)$ correspond à l'application d'une fonction t à une donnée u. L'exécution dans un langage fonctionnel correspond à la notion de ***β-réduction*** en λ-calcul, qui est définie par la règle suivante :

$$\beta_{\to} \qquad (\lambda x.\, t)\ u \quad \longrightarrow \quad t\,[^{u}/_{x}]$$

(où $\boldsymbol{t}\,[^{\boldsymbol{u}}/_{\boldsymbol{x}}]$ représente le terme t dans lequel chaque occurrence de la variable x a été remplacée par le terme u). On considérera souvent une autre règle, dite règle d'***η-réduction***, témoignant d'une propriété dite d'***extensionalité*** ou de ***catégoricité*** :

$$\eta_{\to} \qquad \lambda x.\, (M\ x) \quad \longrightarrow \quad M$$

Les règles $\beta_{\to}$ et $\eta_{\to}$ définissent une notion d'équivalence sur les termes du λ-calcul, appelée ***βη-équivalence***.

Il est possible de contraindre la formation des λ-termes en ajoutant une notion de ***typage***. Par exemple une application $(t \quad u)$ sera mal formée si t n'est pas de type « *fonction* ». Le λ-calcul peut être étendu à d'autres constructions comme le produit (produit cartésien) et la somme (union disjointe). Il suffit d'étendre sa syntaxe et de rajouter les règles β et η et les types correspondants. On peut également introduire le type unité `1` qui correspond au produit vide, et le type vide `0` qui correspond à la somme vide.

Une fois la syntaxe et les règles de réduction du langage définis, il faut en préciser la *sémantique*. Pour ce faire, on interprète les termes et les types du langage dans un modèle mathématique. Nous utiliserons pour cela des *catégories*.

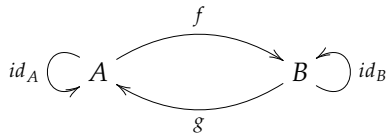
Les isomorphismes de types

L'étude des isomorphismes dans les catégories trouve une application pratique particulièrement intéressante en programmation, où elle permet de considérer les types « à isomorphisme près », c'est-à-dire sans se préoccuper de détails sans importance dans la représentation de données, comme par exemple l'ordre des paramètres d'une fonction. De nombreux travaux ont par exemple porté sur la recherche de fonctions dans une bibliothèque en utilisant le type comme clef de recherche [34, 35, 36, 71, 72, 73, 75] (par exemple l'outil *CamlSearch* écrit par Jérôme Vouillon, Julien Jalon et Roberto Di Cosmo pour le langage *Caml Light*) ou bien sur la recherche d'un module d'après une spécification [15, 33, 5], ou encore dans un assistant de preuve pour trouver des preuves dans des bibliothèques [33]. Une autre application est l'inter-opérabilité entre langages, dans le but par exemple d'utiliser une bibliothèque écrite dans un langage pour programmer dans un autre, qui ne dispose pas forcément des mêmes types. Cette idée a été utilisée en 1997 par IBM pour le projet *Mocking Bird*. Si ces deux applications peuvent paraître anecdotiques au premier abord, elles deviennent primordiales pour des fonctions au type très complexe ou bien pour les programmeurs de grands logiciels, qui doivent gérer des bibliothèques gigantesques de fonctions.

Formellement, deux types A et B sont dits ***isomorphes*** s'il existe une fonction $f : A \to B$ et une fonction $g : B \to A$ telles que les compositions $f \circ g$ et $g \circ f$ soient respectivement égales (pour une théorie donnée) à l'identité de B et de A. L'idée de base est qu'une donnée de type A peut être codée par une fonction en une donnée de type B et réciproquement, sans perte d'information.

On peut définir également une notion d'***isomorphisme sémantique*** : deux types sont dits isomorphes sémantiquement s'ils sont isomorphes dans tous les modèles du langage. Dans une

catégorie, deux objets A et B sont isomorphes si le diagramme suivant *commute* (c'est-à-dire si $f \circ g = id_B$ et $g \circ f = id_A$). :



Nous verrons que cette notion d'isomorphisme sémantique est équivalente à la définition d'isomorphisme énoncée juste avant[1].

Cette notion d'isomorphisme peut aussi être transportée en théorie de la démonstration, sous le nom d'***équivalence forte*** de propositions [68], grâce à la *correspondance de Curry-Howard*, qui établit un parallèle entre formule logique et type, et entre preuve et programme (voir page 48). Deux propositions A et B sont dites ***fortement équivalentes*** s'il existe une preuve de $A \Rightarrow B$ et une preuve de $B \Rightarrow A$, qui, composées dans un sens ou dans l'autre, conduisent après *élimination des coupures* (voir page 2.1.2) à une expansion de l'axiome identité.

Enfin nous verrons que la notion d'isomorphisme de types est également très proche de la notion d'égalité en théorie des nombres.

Un exemple célèbre d'isomorphisme de types non-trivial est celui de la ***curryfication*** de fonctions :

$$(A \times B) \to C \cong A \to (B \to C)$$

Autrement dit une fonction prenant deux paramètres de type A et B peut être transformée en une fonction prenant un paramètre a de type A et renvoyant une fonction de type $B \to C$, version « spécialisée » de la fonction pour le paramètre a. Cela correspond à une application du célèbre théorème S_n^m de Kleene de la théorie de la calculabilité.

Les exemples d'application mentionnés ci-dessus nous laissent entrevoir deux champs de recherche :

- la détection de types isomorphes,
- la construction, éventuellement automatisée, de fonctions de conversion entre types isomorphes.

Ce deuxième sujet est encore largement ouvert. Le premier a été déjà étudié dans de nombreux cadres, que je vais maintenant résumer.

Dès 1983, Sergei Soloviev a montré que les isomorphismes des catégories cartésiennes fermées étaient finiment axiomatisables grâce à la théorie $Th^1_{\times 1}$ (voir figure 1 page 17). Pour cela, il a montré que les isomorphismes d'une catégorie cartésienne fermée particulière (à savoir la catégorie des ordinaux finis et des applications entre eux) étaient axiomatisables par $Th^1_{\times 1}$, ce qui implique qu'il ne peut y avoir d'autres isomorphismes vrais dans toutes les catégories cartésiennes fermées que ceux engendrés par $Th^1_{\times 1}$ (Soloviev [77]).

L'article fondateur de la recherche sur les isomorphismes dans le cadre des types et du λ-calcul est dû à Kim Bruce et Giuseppe Longo, en 1985 [20], qui se sont intéressés à un λ-calcul sans paires. Le problème résolu par Sergei Soloviev a été étudié ensuite par Kim Bruce, Roberto Di Cosmo et Giuseppe Longo [19] en 1992 grâce à des méthodes purement syntaxiques, dans le cadre du λ-calcul avec paires surjectives et type unité.

Enfin en 1990, Roberto Di Cosmo a donné une caractérisation complète des isomorphismes

[1] Voir théorème 2.37 page 71

de types valides dans le λ-calcul de second ordre, avec paires surjectives et type unité, résultat résumé figure 1, qui inclut tous les systèmes étudiés auparavant (voir Di Cosmo [37, 36]). Le tableau ci-dessous donne la correspondance entre λ-calculs typés, systèmes logiques, modèles catégoriques et théories d'isomorphismes, en donnant les références aux travaux dans lesquels les premiers résultats de complétude ont été établis. Dans ce tableau, $CPI(c)$ dénote le calcul propositionnel intuitionniste sur l'ensemble de connecteurs c, $\boldsymbol{\lambda^1_{\beta\eta}}$ et $\boldsymbol{\lambda^1_{\times 1\beta\eta}}$ désignent respectivement le λ-calcul sans et avec type produit et unité, et $\boldsymbol{\lambda^2_{\beta\eta}}$ et $\boldsymbol{\lambda^2_{\times 1\beta\eta}}$ des extensions de ces calculs où l'on autorise les quantifications sur des variables de types.

λ-calcul	catégorie	logique	théorie	auteurs
$\lambda^1_{\beta\eta}$		$CPI(\Rightarrow)$	Th^1	Martin [67], Bruce-Longo [20]
$\lambda^1_{\times 1\beta\eta}$	CCC	$CPI(\top, \wedge, \Rightarrow)$	$Th^1_{\times 1}$	Soloviev [77], Bruce-Di Cosmo-Longo [19]
$\lambda^2_{\beta\eta}$		$CPI(\forall, \Rightarrow)$	Th^2	Bruce-Longo [20]
$\lambda^2_{\times 1\beta\eta}$		$CPI(\forall, \top, \wedge, \Rightarrow)$	$Th^2_{\times 1}$	Di Cosmo [37]

Enfin mentionnons le travail de Joseph Gil [49], qui a étudié récemment les isomorphismes de types dans un cadre adapté particulièrement aux langages impératifs, à savoir avec somme et produit mais sans flèche (ce qui correspond aux catégories distributives). Il s'est intéressé aux types algébriques et a prouvé que dans ce cadre, le sous-typage additif (inclusion de types) est indécidable.

•

Pour appréhender le problème des isomorphismes avec somme, la première étape a consisté à limiter la question aux isomorphismes dits « linéaires ». Sergei Soloviev a montré que la théorie constituée des axiomes 1,2,3,5 et 7 de la figure 1 est complète pour ces types. Plutôt qu'étudier ainsi les isomorphismes linéaires de formules intuitionnistes, nous allons nous intéresser dans le chapitre 3 au cas de ces isomorphismes dans le cadre de la logique linéaire multiplicative. Grâce à une étude de la forme des réseaux de preuves, nous allons montrer que ces isomorphismes sont remarquablement simples et correspondent exactement aux règles d'associativité de commutativité pour les connecteurs $\otimes$ et $\parr$, ainsi qu'aux règles pour leurs éléments neutres.

•

L'étude des isomorphismes de types dans les catégories cartésiennes fermées rejoint un problème de théorie des nombres énoncé par Alfred Tarski, et connu sous le nom de *problème des égalités du lycée* [41]. Il s'agissait de déterminer si les égalités sur les entiers apprises au lycée (associativité, la commutativité, distributivité, etc...) et présentées à la figure 2 sont suffisantes pour prouver toutes les égalités numériques possibles écrites avec des variables, l'exponentiation, la somme et le produit. Autrement dit, la théorie présentée figure 2 est-elle complète pour les équations sur les entiers naturels ?

Charles Martin a montré [67] que la réponse est affirmative si l'on se restreint aux équations sans somme, avec les quatre équations présentées en haut sur la figure 2. Ce résultat est valable également en présence de la constante 1 et des axiomes associés, à savoir :

$$1a = a \qquad a^1 = a \qquad 1^a = 1$$

Cette théorie coïncide exactement avec la théorie $Th^1_{\times 1}$ des isomorphismes de types sans somme ni type vide, lorsque l'on remplace le produit et la constante 1 par les types correspondants et l'exponentiation b^a par le type $a \to b$.

(**swap**) $A \to (B \to C) = B \to (A \to C)$ $\}\ Th^1$

1. $A \times B = B \times A$
2. $A \times (B \times C) = (A \times B) \times C$
3. $(A \times B) \to C = A \to (B \to C)$
4. $A \to (B \times C) = (A \to B) \times (A \to C)$
5. $A \times 1 = A$
6. $A \to 1 = 1$
7. $1 \to A = A$

$\}\ Th^1_{\times 1}$

8. $\forall X.\forall Y.A = \forall Y.\forall X.A$
9. $\forall X.A = \forall Y.A\left[{}^{Y}/_{X}\right]$ (X libre pour Y dans A, et $Y \notin FTV(A)$)
10. $\forall X.(A \to B) = A \to \forall X.B$ ($X \notin FTV(A)$)

$\}$ +**swap** $= Th^2$

11. $\forall X.A \times B = \forall X.A \times \forall X.B$
12. $\forall X.1 = 1$

split $\forall X.A \times B = \forall X.\forall Y.A \times (B\left[{}^{Y}/_{X}\right])$

$\}\ Th^2_{\times 1}$

$\}$ - 10, 11 $= Th^{ML}$

A, B, C sont des types quelconques, 1 est le type terminal `unit`.
L'axiome **swap** de Th^1 est dérivable de 1 et 3 dans $Th^1_{\times 1}$.
$FTV(A)$ est l'ensemble des variables de types libres dans A.
Th^{ML} désigne la théorie des isomorphismes pour le noyau de *ML*, c'est-à-dire pour $\lambda^2_{\times 1 \beta \eta}$ restreint aux types avec quantification préfixe et avec inférence de type.

FIG. 1 – Théories des isomorphismes pour divers λ-calculs.

$$
\begin{array}{rclcrcl}
ab & = & ba & & (ab)c & = & a(bc) \\
c^{ab} & = & (c^a)^b & & (ab)^c & = & a^c b^c \\
a+b & = & b+a & & (a+b)+c & = & a+(b+c) \\
a(b+c) & = & ab+ac & & c^{a+b} & = & c^a c^b
\end{array}
$$

FIG. 2 – « Égalités du lycée ».

Dans le cas des entiers avec somme et constante 1, Alex J. Wilkie a montré que la conjecture de Tarski était fausse, et R. Gurevič a même montré qu'il n'y a pas d'axiomatisation finie des égalités entre entiers naturels avec cette grammaire [54]. Pour cela, il a donné une suite infinie d'égalités et a montré qu'elle ne pouvait se déduire d'aucun ensemble fini d'axiomes.

Dans le cas des isomorphismes avec somme et type vide, nous pouvions espérer que le parallèle avec les entiers n'allait pas se poursuivre, étant donné que les arguments utilisés par Gurevič ne semblaient pas avoir de correspondant en λ-calcul. Une axiomatisation finie pour les types aurait permis de réaliser facilement des tests d'isomorphisme et l'écriture automatique des fonctions de conversion. Le problème est resté ouvert assez longtemps, et l'un des principaux résultats de la thèse énonce que ces isomorphismes des types ne sont pas finiment axiomatisables. Nous allons voir en effet que les égalités de la suite de Gurevič sont aussi des isomorphismes — ce qui est un résultat assez inattendu.

Nous verrons également que malgré cette propriété commune, la correspondance entre entiers et types n'est plus vraie en général.

Formes normales

Pour un λ-calcul simplement typé sans type produit ni somme ni les constantes associées, les propriétés de la β-réduction sont très simples. Il suffit d'appliquer la réduction autant de fois qu'il est possible et dans un ordre quelconque. Le processus termine toujours et le résultat, appelé ***forme normale***, est unique. Avec les types produits et sommes, ces propriétés ne sont pas toujours vérifiées.

Dans certains cas cependant, il serait intéressant de pouvoir trouver un représentant unique d'une classe de termes. Par exemple le travail sur les isomorphismes de types mentionné ci-dessus a conduit à utiliser des λ-termes compliqués dont il fallait prouver qu'ils étaient des isomorphismes. Il fallait donc les composer avec leur inverse supposé et montrer qu'ils étaient $\beta\eta$-équivalents à l'identité. La β-réduction associée à l'η-réduction ne permettent plus d'obtenir une forme normale unique. Il a donc fallu trouver une autre méthode.

•

La première étape a consisté à définir extensionnellement une notion de forme normale « canonique » pour le λ-calcul simplement typé avec somme, produit et constantes. En fait, nous ne pouvons pas parvenir à un représentant unique d'une classe d'équivalence modulo ηet β, essentiellement à causes de certaines équations dites « conversions commutatives » que l'on peut difficilement éviter. Cependant la forme que nous obtenons doit respecter des contraintes structurelles très fortes qui limitent de manière drastique les possibilités de formations des termes.

Ce travail a été effectué grâce à une étude des modèles catégoriques du λ-calcul, et en particulier grâce aux relations logiques de Grothendieck. Il vient à la suite de plusieurs articles sur le sujet. Tout d'abord, en 1993, Achim Jung et Jerzy Tiuryn ont donné une preuve de λ-définissabilité grâce aux relations logiques de Kripke dans les catégories cartésiennes fermées [58]. Marcelo Fiore a ensuite montré dans le premier cas que ce résultat de définissabilité pouvait être adapté pour donner un résultat de normalisation extensionnelle [45]. Le résultat de définissabilité a été étendu aux catégories bi-cartésiennes fermées par Marcelo Fiore et Alex Simpson en 1999 au cas des catégories bi-cartésiennes fermées grâce aux relations logiques de Grothendieck [46]. Le travail présenté ici étend le résultat de normalisation à ce cas.

•

Dans son article sur les formes normales du λ-calcul sans somme [45], Marcelo Fiore montre ensuite que l'algorithme de *normalisation par évaluation* est une version « intentionnelle » de ce résultat, c'est-à-dire qu'il donne une méthode effective de normalisation.

La normalisation par évaluation est une technique très astucieuse inventée par Ulrich Berger et Helmut Schwichtenberg [17] en 1991 permettant « d'inverser » la fonction d'évaluation pour obtenir la syntaxe d'un terme en forme normale à partir de sa sémantique. En 1996, Olivier Danvy redécouvrait cet algorithme pour faire de la normalisation de programme *ML* de manière très élégante [27]. Cette méthode est connue sous le nom d'*évaluation partielle dirigée par les types*. Elle utilise notamment les opérateurs de contrôle shift et reset pour traiter le cas du type somme.

Les deux derniers chapitres montrent une utilisation originale de cet algorithme : nous allons l'appliquer au problème de la vérification des isomorphismes de types. Malheureusement l'algorithme existant ne produit pas les termes dans notre forme normale canonique. De plus, deux termes $\beta\eta$-équivalents peuvent donner des résultats complètement différents, et nous verrons même que dans la cas des isomorphismes de la suite de Gurevič, on obtient des termes $\beta\eta$-équivalents à l'identité dont la normalisation produit des résultats de plusieurs milliers de lignes...

Heureusement, ce problème a été résolu. Le dernier chapitre montre comment on peut modifier l'algorithme d'évaluation partielle dirigée par les types pour qu'il produise un résultat répondant à la définition de nos formes normales, notamment grâce à un travail sur les opérateurs de contrôle. Cela aboutira dans les cas mentionnés ci-dessus à une drastique optimisation en taille et en temps de normalisation.

•

La thèse est divisée en trois parties. La première contient des rappels sur les catégories et le λ-calcul, la deuxième traite plus particulièrement des isomorphismes de type et la dernière s'attache aux problèmes de formes normales.

Le chapitre 1 contient des rappels de théorie des catégories, pour introduire les notions nécessaires à la compréhension de la suite. Il s'attarde en particulier sur l'étude des isomorphismes dans les catégories.

Dans le chapitre 2, je rappelle les définitions relatives au λ-calcul, ainsi que les problèmes de confluence. Ensuite j'introduis les modèles catégoriques du λ-calcul et je relie les questions d'isomorphismes entre types à celles sur les isomorphismes dans les catégories.

Le chapitre 3, qui décrit un travail réalisé avec Roberto Di Cosmo et publié à CSL 1999 (voir [12]), est l'étude des isomorphismes linéaires en logique linéaire multiplicative, d'abord sans puis avec constantes.

Le chapitre 4 s'intéresse quant à lui aux isomorphismes en logique intuitionniste et montre en particulier que les isomorphismes des catégories bi-cartésiennes fermées ne sont pas finiment axiomatisables, en reliant ce problème avec celui des égalités du lycée de Tarski. C'est un travail fait avec Marcelo Fiore et Roberto Di Cosmo, publié à LICS 2002 (voir [14]).

L'étude extensionnelle de la normalisation (réalisée avec Marcelo Fiore et Roberto Di Cosmo, pas encore publiée) est décrite dans le chapitre 5.

Enfin les chapitres 6 et 7 montrent l'application de l'évaluation partielle dirigée par les types au problème des isomorphismes de type. Je présente d'abord au chapitre 6 une implantation en *Objective Caml* de l'évaluateur partiel (travail réalisé sous la direction d'Olivier Danvy en 1997 et publié à TIC 1998 [9, 10]). Je montre ensuite les optimisations apportées à cet algorithme (chapitre 7) pour qu'il soit utilisable pour traiter le cas des isomorphismes de type du chapitre 4. Cela m'amène à utiliser des opérateurs de contrôle plus puissants. J'ai adapté récemment une petite partie de ce travail avec Olivier Danvy pour un langage avec constructions `let` pour obtenir un

évaluateur partiel complètement paresseux (qui ne calcule jamais deux fois la même chose) de programmes sans effets de bords. Cette adaptation a été publiée à GPCE 2002 [11].

Avertissement

Les sujets abordés sont souvent pointus et s'adressent à des spécialistes. Cependant, étant donné leur diversité, j'ai voulu faciliter la lecture pour un grand nombre de personnes en présentant le plus possible la thèse comme un ouvrage « auto-contenu ». J'ai donc essayé de définir au maximum les termes et notions employés et d'énoncer tous les théorèmes utilisés avec les notations et l'énoncé exact utilisés dans la suite. Ceci permet au lecteur de se reporter à des références précises sans avoir à se plonger dans des ouvrages extérieurs. Ce parti a été également pris pour des exigences de rigueur, pour éviter notamment les références à des hypothétiques « résultats bien connus ». Vous trouverez donc dans la suite des rappels de bases utiles à mon propos dans les domaines du λ-calcul, de la logique linéaire, de la théorie des catégories et de l'évaluation partielle. Cependant le but n'est pas de faire un cours sur ces notions et je renvoie à d'autres ouvrages pour des explications détaillées et d'autres développements de ces sujets, ainsi que pour les démonstrations. Ces parties pourront évidemment être ignorées par les connaisseurs. Les autres y trouveront un bref rappel des notions essentielles.

Présentation et conventions typographiques

Pour permettre de les repérer facilement, les nouvelles notions introduites sont composées en ***gras italique***, et sont référencées dans l'index. Les nouveaux symboles introduits seront également composés en gras. Un symbole en gras ne doit donc pas être distingué du même symbole en maigre. Les couleurs utilisées dans certaines versions de ce document ne servent qu'à mettre en valeur certaines portions du texte et n'ont aucune valeur sémantique. Enfin le symbole [▷ *translation*] donne la traduction anglaise de certains termes.

Terminologie

Nous utiliserons la plupart du temps la terminologie française pour les objets mathématiques. Notamment une ***fonction*** d'un ensemble E vers un ensemble F est une relation de E vers F telle que tout élément de E est en relation avec *au plus* un élément de F [▷ *partial function*]. On pourra donc parler d'***ensemble de définition*** d'une fonction. Si cet ensemble de définition est E tout entier, on parle d'***application*** (ou de ***fonction totale***). Enfin une ***paire*** de E est un ensemble de deux éléments de E, un n-uplet est une application de $\mathbb{N}_n = \{ 1, \ldots, n \}$ dans E. Un ***couple*** est un 2-uplet. On utilisera néanmoins l'anglicisme ***paire*** pour désigner un couple lorsque l'on parlera d'une structure de données en informatique. Une ***famille*** d'éléments de E indexée par un ensemble I est une application de I dans E (souvent notée $(e_i)_{i \in I}$).
$\mathbb{N} = \{ 0, 1, 2, \ldots \}$ est l'ensemble des ***entiers naturels***, $\mathbb{N}^*$ est l'ensemble des entiers naturels strictement positifs, et $\mathbb{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}$ est l'ensemble des ***entiers relatifs***.

Première partie

Modèles du λ-calcul

Chapitre 1

Introduction aux isomorphismes dans les catégories

Résumé

Ce chapitre présente de manière succincte les définitions et résultats classiques de théorie de catégories qui seront utiles par la suite. Il doit être vu comme un catalogue permettant de fixer les notations et auquel on pourra se référer pour trouver un énoncé exact. Il ne comporte aucun résultat nouveau et peut donc être ignoré par les connaisseurs. Pour plus de détails, nous renvoyons par exemple aux livres de Saunders Mac Lane [65] ou de Andrea Asperti et Giuseppe Longo [6]. Cependant certains petits résultats importants pour la suite sont démontrés ici. Il s'agit de propriétés concernant les isomorphismes dans les catégories et dont les preuves figurent rarement dans la littérature.

1.1 Définitions de base

Définition 1.1 (graphe orienté) Un ***graphe orienté*** est un quadruplet $\mathfrak{G}$ constitué :
- d'une collection notée $\boldsymbol{Obj_{\mathfrak{G}}}$ dont les éléments sont appelés ***objets*** de $\mathfrak{G}$,
- d'une collection notée $\boldsymbol{Mor_{\mathfrak{G}}}$ dont les éléments sont appelés ***morphismes*** ou ***flèches*** de $\mathfrak{G}$,
- de deux applications $\boldsymbol{dom_{\mathfrak{C}}}$ et $\boldsymbol{codom_{\mathfrak{C}}}$ qui associent à chaque morphisme f des objets A et B appelés respectivement ***domaine*** et ***co-domaine*** de f, ce que l'on notera $\boldsymbol{f : A \longrightarrow B}$,

La collection des flèches d'un objet A vers un objet B d'un graphe $\mathfrak{G}$ est notée $\boldsymbol{\mathfrak{G}(A,\ B)}$.

Définition 1.2 (catégorie) Une ***catégorie*** est un graphe orienté $\mathfrak{C}$ vérifiant les conditions suivantes :
- étant donné trois objets quelconques A, B et C de $\mathfrak{C}$, un morphisme $f : A \to B$ et un morphisme $g : B \to C$, il existe un morphisme particulier noté $\boldsymbol{g \circ f}$ et appelé ***composition*** de g et f, vérifiant $g \circ f : A \to C$
- pour chaque objet A de $\mathfrak{C}$, il existe un morphisme particulier $\boldsymbol{id_A} : A \to A$ appelé ***identité*** de A
- pour tous les objets A, B, C, D et morphismes $f : A \to B$, $g : B \to C$ et $h : C \to D$,
  - $id_B \circ f = f$ et $f \circ id_A = f$ (lois de l'identité)
  - $h \circ (g \circ f) = (h \circ g) \circ f$ (associativité)

La flèche identité d'un objet A sera souvent notée simplement $\boldsymbol{id}$ au lieu de id_A si cela n'entraîne pas de confusion.

Exemple : $\mathfrak{Ens}$, la catégorie des ensembles, dont les flèches sont les applications (fonctions totales) entre les ensembles. On notera $\mathfrak{Ens}_\mathfrak{f}$ la catégorie des ensembles finis.

Définition 1.3 (catégorie duale) On appellera ***catégorie duale*** d'une catégorie $\mathfrak{C}$ la catégorie notée $\boldsymbol{\mathfrak{C}^{op}}$ définie par
- $Obj_{\mathfrak{C}^{op}} = Obj_\mathfrak{C}$,
- $Mor_{\mathfrak{C}^{op}} = Mor_\mathfrak{C}$,
- $dom_{\mathfrak{C}^{op}} = codom_\mathfrak{C}$ et $codom_{\mathfrak{C}^{op}} = dom_\mathfrak{C}$

La composition de deux morphismes f et g de $\mathfrak{C}^{op}$ est donnée par $f \circ_{\mathfrak{C}^{op}} g = g \circ_\mathfrak{C} f$.

Définition 1.4 (catégorie produit) Le ***produit*** $\mathfrak{C} \times \mathfrak{D}$ de deux catégories $\mathfrak{C}$ et $\mathfrak{D}$ est la catégorie définie de la manière suivante :
- les objets sont les couples $(A,\ B)$, où A et B sont respectivement des objets de $\mathfrak{C}$ et $\mathfrak{D}$,
- les flèches entre $(A,\ B)$ et $(A',\ B')$ sont les couples $(f,\ g)$, où $f : A \to B$ et $g : A' \to B'$ sont respectivement des flèches de $\mathfrak{C}$ et $\mathfrak{D}$,
- $id_{(A,\,B)} = (id_A,\ id_B)$,
- $(f,\ g) \circ (f',\ g') = (f \circ f',\ g \circ g')$.

Définition 1.5 Soit $\mathfrak{C}$ une catégorie et $A, B \in Obj_\mathfrak{C}$.
- Un morphisme $f : A \to B$ est dit ***épimorphisme*** si pour tous g et h de domaine B $\quad g \circ f = h \circ f \implies g = h$
- Un morphisme $f : A \to B$ est dit ***monomorphisme*** si $\quad f \circ g = f \circ h \implies g = h$
- Un morphisme $f : A \to B$ est dit ***isomorphisme*** si il existe $g : B \to A$ tel que $\quad g \circ f = id$ et $f \circ g = id$

S'il existe un isomorphisme $f : A \to B$, on dira que A et B sont ***isomorphes***, ce que l'on notera $A \cong B$.

On utilisera la notation $A \rightarrowtail B$ pour représenter un monomorphisme de A vers B. Dans $\mathfrak{Ens}$, une flèche est un monomorphisme si et seulement si c'est une application injective. Notamment une fonction d'inclusion est un monomorphisme. De même, dans $\mathfrak{Ens}$, une flèche est un épimorphisme (resp. un isomorphisme) si et seulement si c'est une application surjective (resp. bijective). Cependant ces propriétés ne sont pas toujours vraies dans d'autres catégories.

Une catégorie $\mathfrak{C}$ est dite ***petite*** si $Mor_\mathfrak{C}$ (et donc $Obj_\mathfrak{C}$) sont des ensembles. Elle est dite ***localement petite*** si $\mathfrak{C}(A,\ B)$ est un ensemble (et non une classe) pour tous les objets A et B.

Définition 1.6 (catégorie libre) Étant donné un graphe orienté $\mathfrak{G}$, on appelle ***catégorie libre*** engendrée par $\mathfrak{G}$ la catégorie
- dont les objets sont les objets de $\mathfrak{G}$,
- et dont les flèches sont les suites finies de flèches « composables » de $\mathfrak{G}$, c'est-à-dire les suites de la forme $(f_1, \ldots, f_n)_{n\in\mathbb{N}}$ telles que $codom_{\mathfrak{G}}(f_i) = dom_{\mathfrak{G}}(f_{i+1})$ pour $1 \leq i < n$.

Les identités sont les suites vides ; la composition est la concaténation de suites.

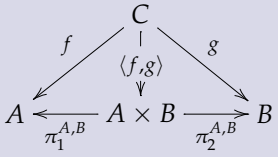
1.2 Catégories bi-cartésiennes fermées

Les propriétés des catégories sont souvent énoncées à l'aide de ***diagrammes***. Il est possible de donner une définition formelle de cette notion comme un homomorphisme de graphes entre un graphe quelconque et le graphe sous-jacent à une catégorie (voir par exemple dans le livre de Saunders Mac Lane [65]). Nous nous en tiendrons ici à une définition intuitive. Dans un diagramme, un morphisme f de $\mathfrak{C}(A,\ B)$ est représenté par une flèche de A vers B, étiquetée f. Nous dirons qu'un diagramme ***commute*** lorsqu'étant donné deux objets A et B du diagramme, toute composition de flèches commençant à A et se terminant à B donne le même résultat.

1.2.1 Constructions de base

Définition 1.7 (produit) Dans une catégorie $\mathfrak{C}$, on appelle ***produit*** de deux objets A et B un triplet composé :
- d'un objet noté $A \times B$,
- de deux flèches $\boldsymbol{\pi_1^{A,B}} : A \times B \to A$ et $\boldsymbol{\pi_2^{A,B}} : A \times B \to B$, appelées ***projections***,

tels que pour tout objet C et pour toutes flèches $f : C \to A$ et $g : C \to B$
il existe une unique flèche $\boldsymbol{\langle f, g \rangle} : C \to A \times B$ telle que le diagramme suivant commute :



Nous noterons la plupart du temps les projections $\boldsymbol{\pi_1}$ et $\boldsymbol{\pi_2}$ quels que soient A et B, au lieu de $\pi_1^{A,B}$ et $\pi_2^{A,B}$.

Définition 1.8 (objet terminal) Un objet 1 d'une catégorie $\mathfrak{C}$ est dit ***terminal*** si pour tout objet A de $\mathfrak{C}$ il existe une unique flèche de A vers 1, que l'on notera $\boldsymbol{!A}$.

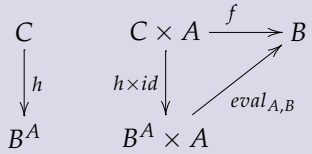
$$A \xrightarrow{!A} 1$$

Il est facile de montrer que si une catégorie a deux objets terminaux, alors ils sont isomorphes, et que si un objet est isomorphe à un objet terminal, il est lui-même terminal.

Définition 1.9 (catégorie cartésienne) Une catégorie est dite ***cartésienne*** si elle a
- un objet terminal
- et pour tout couple d'objets, un produit.

Définition 1.10 (exponentielle ou exposant) Dans une catégorie $\mathfrak{C}$, on appelle ***exponentielle*** ou ***exposant*** d'un objet B par un objet A un couple composé :
- d'un objet noté B^A,
- et d'un morphisme $\boldsymbol{eval_{A,B}} : B^A \times A \to B$

tels que pour tout objet C et pour toute flèche $f : C \times A \to B$
il existe une unique flèche $h : C \to B^A$ telle que le diagramme suivant commute :

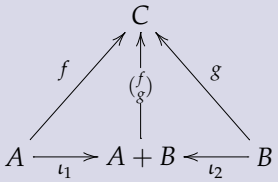


Dans la définition ci-dessus et dans toute la suite, nous notons $\boldsymbol{f \times g}$ la flèche $\langle f \circ \pi_1, f \circ \pi_2 \rangle$. Nous simplifierons encore une fois les notations en écrivant ***eval*** au lieu de $eval_{A,B}$, et nous noterons $\boldsymbol{\Lambda}(f)$ la flèche h ci-dessus, et $\boldsymbol{\Lambda^{-1}}(h)$ la flèche $eval \circ (h \times id)$. La flèche $\Lambda(f)$ est la ***curryfication*** de f, et $\Lambda^{-1}(h)$ la ***dé-curryfication*** de h. Il est facile de montrer que Λ est une bijection dont l'inverse est Λ^{-1}.

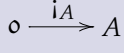
Définition 1.11 (catégorie cartésienne fermée) Une catégorie cartésienne $\mathfrak{C}$ est dite ***fermée*** s'il existe une exponentielle pour tout couple d'objets de $\mathfrak{C}$.

Définition 1.12 (co-produit) Dans une catégorie $\mathfrak{C}$, on appelle ***co-produit*** ou ***somme*** de deux objets A et B un triplet composé :
- d'un objet noté $A + B$,
- de deux flèches $\iota_1 : A \to A + B$ et $\iota_2 : B \to A + B$ appelées ***injections***,

tels que pour tout objet C et pour toutes flèches $f : A \to C$ et $g : B \to C$
il existe une unique flèche $\binom{f}{g} : A + B \to C$ telle que le diagramme suivant commute :



Définition 1.13 (objet initial) Un objet o d'une catégorie $\mathfrak{C}$ est dit ***initial*** si pour tout objet A de $\mathfrak{C}$ il existe une unique flèche de o vers A, que l'on notera $\boldsymbol{iA}$.



Il est facile de montrer que si une catégorie a deux objets initiaux, alors ils sont isomorphes, et que si un objet est isomorphe à un objet initial, il est lui-même initial.

Définition 1.14 (catégorie co-cartésienne) Une catégorie ***co-cartésienne*** est une catégorie avec co-produit et objet initial.

Définition 1.15 (catégorie bi-cartésienne) Une catégorie ***bi-cartésienne*** est une catégorie avec produit, co-produit, objet terminal et objet initial, c'est-à-dire une catégorie cartésienne et co-cartésienne.

Définition 1.16 (catégorie distributive) Une catégorie ***distributive*** est une catégorie bi-cartésienne $\mathfrak{C}$ telle que pour tout triplet (A, B, C) d'objets de $\mathfrak{C}$,

1. il existe un isomorphisme $\delta : (A \times B) + (A \times C) \to A \times (B + C)$,
2. et l'unique flèche $\alpha : \mathfrak{o} \to (A \times \mathfrak{o})$ est un isomorphisme.

En fait, la première propriété implique le seconde, même si ce n'est pas immédiat à prouver.

Définition 1.17 (catégorie bi-cartésienne fermée) Une catégorie $\mathfrak{C}$ est dite ***bi-cartésienne fermée*** si
- elle est cartésienne fermée,
- elle a un objet initial,
- et s'il existe un co-produit pour tout couple d'objets de $\mathfrak{C}$.

On utilisera les abréviations anglo-saxones ***CCC*** et ***biCCC*** pour parler respectivement des catégories cartésiennes fermées et des catégories bi-cartésiennes fermées.

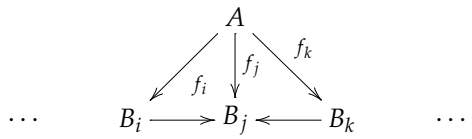
1.2.2 Limites

Nous allons maintenant généraliser les définitions de produit, co-produit, *etc.* en introduisant la notion de *limite*. Pour cela, nous avons besoin de quelques définitions :

Définition 1.18 (cône) Soit $\mathfrak{C}$ une catégorie et D un diagramme de $\mathfrak{C}$, d'objets $\{B_i\}_{i\in I}$. Un ***cône*** de ***base*** D est défini par :
- un objet A de $\mathfrak{C}$, appelé ***sommet***
- une famille de morphismes $\{f_i \in \mathfrak{C}(A,\ B_i)\}_{i\in I}$ telle que $\forall i, j \in I \quad \forall f \in \mathfrak{C}(B_i,\ B_j)$ dans D

$$f \circ f_i = f_j$$

Un cône peut être visualisé de la façon suivante :

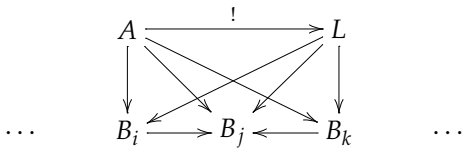


Dans une catégorie $\mathfrak{C}$ donnée, et pour un diagramme D de cette catégorie, les cônes de base D forment une catégorie, que l'on notera $\mathfrak{Cones}_{\mathfrak{C},D}$. Les morphismes entre un cône $(A,\ \{\ f_i\ \})$ et un cône $(A',\ \{\ f'_i\ \})$ sont les morphismes $g \in \mathfrak{C}(A,\ A')$ tels que $\forall i \in I \quad f'_i \circ g = f_i$.

Définition 1.19 (limite) Étant donnée une catégorie $\mathfrak{C}$ et un diagramme D de $\mathfrak{C}$, une ***limite*** (ou ***cône universel***) du diagramme D est un objet terminal dans $\mathfrak{Cones}_{\mathfrak{C},D}$.

Autrement dit, pour tout cône $(A,\ \{\ f_i\ \})$ sur le diagramme D, une limite $(L,\ \{\ g_i\ \})$ est un

cône de même base tel qu'il existe une unique flèche $h : A \to L$ vérifiant $\forall i \in I\ g_i \circ h = f_i$.



Exemples de limites :

▶ produit Le produit de deux objets A et B est une limite pour le diagramme constitué des seuls objets A et B (sans flèches).

▶ égaliseur Un ***égaliseur*** [▷ *equalizer*] est une limite pour un diagramme de la forme

$$A \underset{g}{\overset{f}{\rightrightarrows}} B$$

Ce qui signifie que pour tout cône $(C,\ \{\ c_1 : C \to A,\ c_2 : C \to B\ \})$,
l'égaliseur $(E,\ \{\ e_1 : E \to A,\ e_2 : E \to B\ \})$ est par définition un cône tel qu'il existe une unique flèche $h : C \to E$ vérifiant $e_1 \circ h = c_1$ et $e_2 \circ h = c_2$.
Or d'après la définition d'un cône, $c_2 = f \circ c_1 = g \circ c_1$ et $e_2 = f \circ e_1 = g \circ e_1$. Un cône sur ce diagramme est donc défini entièrement par la donnée d'une flèche entre le sommet et A. La propriété de l'égaliseur peut donc être lue de la façon suivante :
Un égaliseur de f et g est un objet E et une flèche $e : E \to A$ vérifiant :

- $f \circ e = g \circ e$
- pour toute flèche $c : C \to A$ vérifiant $f \circ c = g \circ c$, il existe une unique flèche $h : C \to E$ telle que $e \circ h = c$.

Cette dernière propriété implique qu'un égaliseur est toujours un monomorphisme.

$$E \rightarrowtail A \underset{g}{\overset{f}{\rightrightarrows}} B$$

Dans $\mathfrak{Ens}$, E est l'ensemble $\{\ x \in A \mid f(x) = g(x)\ \}$ et e est l'injection de ce sous-ensemble de A dans A.

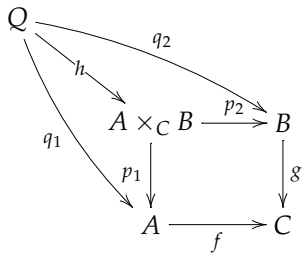
▶ produit fibré Un ***produit fibré*** [▷ *pullback*] est une limite sur un diagramme de la forme

$$A \xrightarrow{f} C \xleftarrow{g} B$$

Autrement dit c'est un objet P (que l'on notera souvent $A \times_C B$) et deux flèches $p_1 : P \to A$ et $p_2 : P \to B$ tels que

- $g \circ p_2 = f \circ p_1$
- et pour tout triplet $(Q,\ q_1 : Q \to A,\ q_2 : Q \to B)$ vérifiant $g \circ q_2 = f \circ q_1$, il existe une

unique flèche $h : Q \to P$ telle que $q_1 = p_1 \circ h$ et $q_2 = p_2 \circ h$.



Dans $\mathfrak{Ens}$, le produit fibré existe toujours ; c'est l'ensemble $\{ (a,\ b) \in A \times B \mid f(a) = g(b) \}$.

On peut bien entendu définir les notions duales : ***co-cône***, ***co-limite***, ***co-égaliseur***, et ***somme amalgamée*** [▷ *pushout*] (dual de produit fibré).

1.3 Foncteurs et catégories de foncteurs

1.3.1 Définitions

Définition 1.20 (foncteur) Un ***foncteur*** F entre deux catégories $\mathfrak{C}$ et $\mathfrak{D}$ est un couple d'applications $\boldsymbol{F_{obj}} : Obj_{\mathfrak{C}} \to Obj_{\mathfrak{D}}$ et $\boldsymbol{F_{mor}} : Mor_{\mathfrak{C}} \to Mor_{\mathfrak{D}}$ telles que,
$\forall\ A, B, C \in \mathfrak{C}, \quad \forall\ f : A \to B, \quad \forall\ g : B \to C$
- $F_{mor}(f) : F_{obj}(A) \to F_{obj}(B)$
- $F_{mor}(g \circ f) = F_{mor}(g) \circ F_{mor}(f)$
- $F_{mor}(id_A) = id_{F_{obj}(A)}$

On omettra souvent les indices obj et mor des foncteurs.

Définition 1.21 (foncteur pleinement fidèle) Soit $F : \mathfrak{C} \to \mathfrak{D}$ un foncteur. Pour tous objets A et B de $\mathfrak{C}$, appelons $F_{mor}(A,\ B)$ la restriction de F_{mor} à $\mathfrak{C}(A,\ B)$, vue comme une application de $\mathfrak{C}(A,\ B)$ dans $\mathfrak{D}(F(A),\ F(B))$.
- Le foncteur F est dit ***fidèle*** [▷ *faithful*] si $F_{mor}(A,\ B)$ est injective pour tout couple $(A,\ B)$ d'objets de $\mathfrak{C}$.
- Le foncteur F est dit ***plein*** [▷ *full*] si $F_{mor}(A,\ B)$ est surjective pour tout couple $(A,\ B)$ d'objets de $\mathfrak{C}$.
- Le foncteur F est dit ***pleinement fidèle*** [▷ *full and faithful*] si $F_{mor}(A,\ B)$ est bijective pour tout couple $(A,\ B)$ d'objets de $\mathfrak{C}$. On parlera aussi de ***plongement*** [▷ *embedding*].

On notera $F : A \hookrightarrow B$ un foncteur pleinement fidèle entre A et B.

On peut définir la composée de deux foncteurs très simplement, ce qui permet de définir la ***catégorie des catégories***, que l'on notera $\mathfrak{Cat}$. Deux catégories sont donc dites ***isomorphes*** si elles sont isomorphes en tant qu'objets de la catégorie $\mathfrak{Cat}$.

Définition 1.22 (hom-foncteur contravariant) Soit $\mathfrak{C}$ une petite catégorie, et B un objet de $\mathfrak{C}$. On notera $\mathfrak{C}(-,\ B) : \mathfrak{C} \to \mathfrak{Ens}$ le foncteur appelé ***hom-foncteur contravariant*** défini par :

- pour tout objet A de $\mathfrak{C}$, $\quad \mathfrak{C}(-,\ B)(C) = \mathfrak{C}(A,\ B)$
- pour toute flèche $h : A \to A'$, $\quad \mathfrak{C}(-,\ B)(h) \ : \ \mathfrak{C}(A',\ B) \ \to \ \mathfrak{C}(A,\ B)$
  $f \ \mapsto \ f \circ h$

  La flèche $\mathfrak{C}(-,\ B)(h)$ sera souvent notée $-\circ h$.

Bien noter le sens de la flèche $- \circ h$, qui justifie l'appellation *contravariant*.

Définition 1.23 (transformation naturelle) Étant donnés deux foncteurs F et G de même domaine $\mathfrak{C}$ et même co-domaine $\mathfrak{D}$, une ***transformation naturelle*** entre F et G est une *famille*

$$\varphi = \{\varphi_A : F(A) \to G(A)\}_{A \in Obj_{\mathfrak{C}}}$$

de *flèches de* $\mathfrak{D}$, telle que :
pour toute flèche $f : A \to B$ de $\mathfrak{C}$, le diagramme suivant commute.

$$\begin{array}{ccc} F(A) & \xrightarrow{\varphi_A} & G(A) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(B) & \xrightarrow{\varphi_B} & G(B) \end{array}$$

Une transformation naturelle est représentée de la manière suivante :

$$\mathfrak{C} \underset{G}{\overset{F}{\Downarrow \varphi}} \mathfrak{D}$$

La notion de transformation naturelle nous permet de considérer les foncteurs comme des objets d'une catégorie. Étant données deux catégories $\mathfrak{C}$ et $\mathfrak{D}$, on notera $\mathfrak{D}^{\mathfrak{C}}$ la catégorie dont les objets sont les foncteurs de $\mathfrak{C}$ dans $\mathfrak{D}$, et les morphismes les transformations naturelles entre ces foncteurs. La composition de deux transformations naturelles $\varphi : F \to G$ et $\psi : G \to H$ est la transformation naturelle $\psi \circ \varphi : F \to H$, appelée ***composée verticale***, et définie par $(\psi \circ \varphi)_A = \psi_A \circ \varphi_A$.

$$\mathfrak{C} \underset{\overset{G}{\Downarrow \psi}}{\overset{F}{\Downarrow \varphi}} \mathfrak{D}$$
$$H$$

On vérifie simplement que l'on définit bien ainsi une catégorie.

1.3.2 Pré-faisceaux

Définition 1.24 (pré-faisceau) À une catégorie localement petite $\mathfrak{C}$ est associée à la catégorie $\mathfrak{Ens}^{\mathfrak{C}^{op}}$ (des foncteurs de $\mathfrak{C}^{op}$ dans $\mathfrak{Ens}$) appelée ***catégorie des pré-faisceaux*** sur $\mathfrak{C}$.

Parmi de nombreuses autres propriétés, on peut démontrer que quelle que soit la catégorie $\mathfrak{C}$, la catégorie des pré-faisceaux sur $\mathfrak{C}$ est cartésienne fermée.

Définition 1.25 (plongement de Yoneda) Le ***plongement de Yoneda*** entre une catégorie $\mathfrak{C}$ et sa catégorie de pré-faisceaux est le foncteur $\mathbf{y}$ défini par

$$\begin{array}{rccc} \mathbf{y} & : & \mathfrak{C} & \to & \mathfrak{Ens}^{\mathfrak{C}^{op}} \\ & & A & \mapsto & \mathbf{y}_A = \mathfrak{C}(-,\ A) \end{array}$$

Autrement dit, le plongement de Yoneda est défini pour tout objet A de $\mathfrak{C}$ par la donnée du foncteur $\mathbf{y}_A : \mathfrak{C}^{op} \to \mathfrak{Ens}$ défini par :

- pour tout objet B de $\mathfrak{C}$, $\mathbf{y}_A(B) = \mathfrak{C}(B,\ A)$
- pour tout flèche $f : B \to C$ de $\mathfrak{C}$, $\begin{array}{rccc} \mathbf{y}_A(f) & : & \mathfrak{C}(C,\ A) & \to & \mathfrak{C}(B,\ A) \\ & & h & \mapsto & f \circ h \end{array}$

Théorème 1.26 Le foncteur $\mathbf{y}$ est pleinement fidèle.

Cette proposition justifie l'appellation de plongement. La preuve utilise le lemme de Yoneda énoncé ci-dessous.

Théorème 1.27 (Lemme de Yoneda) Soit A un objet d'une catégorie $\mathfrak{C}$, et P un élément quelconque de $\mathfrak{Ens}^{\mathfrak{C}^{op}}$. Il existe une *bijection* canonique

$$i(A, P) : P(A) \to \mathfrak{Ens}^{\mathfrak{C}^{op}}(\mathbf{y}_A, P)$$

qui associe à tout élément a de l'ensemble $P(A)$ la transformation naturelle $\alpha_a : \mathbf{y}_A \to P$ définie, pour tout $B \in Obj_{\mathfrak{C}}$ et pour tout $f \in \mathbf{y}_A(B)$ par

$$\alpha_{a,B}(f) = P(f)(a)$$

Une application intéressante du théorème 1.26 à l'étude des isomorphismes est la suivante :

Propriété 1.28 Deux objets A et B d'une catégorie $\mathfrak{C}$ sont isomorphes si et seulement si $\mathbf{y}_A$ est isomorphe à $\mathbf{y}_B$.

Supposons que l'on veuille montrer que deux objets A et B d'une catégorie petite $\mathfrak{C}$ sont isomorphes. Il suffira de montrer que $\mathbf{y}_A$ et $\mathbf{y}_B$ sont isomorphes. En d'autres termes, si nous arrivons à montrer que $\mathfrak{C}(X,\ A)$ est isomorphe à $\mathfrak{C}(X,\ B)$ « naturellement en X », nous pouvons en déduire que A et B sont isomorphes dans $\mathfrak{C}$. L'expression ***naturellement en X*** signifie qu'il existe un isomorphisme naturel φ entre $\mathbf{y}_A$ et $\mathbf{y}_B$.

$$\mathfrak{C}^{op} \underset{\mathfrak{C}(-,\ B)}{\overset{\mathfrak{C}(-,\ A)}{\Downarrow \varphi}} \mathfrak{Ens}$$

On doit donc vérifier que pour toute flèche $f : D \to C$ de $\mathfrak{C}^{op}$, le diagramme suivant commute :

$$\begin{array}{ccc} \mathfrak{C}(C,\ A) & \xrightarrow{\varphi_C} & \mathfrak{C}(C,\ B) \\ {\scriptstyle -\circ f}\downarrow & & \downarrow{\scriptstyle -\circ f} \\ \mathfrak{C}(D,\ A) & \xrightarrow{\varphi_D} & \mathfrak{C}(D,\ B) \end{array}$$

On a défini plus haut la composée verticale de transformations naturelles. Il existe une autre composition de transformations naturelles, appelée ***composée horizontale***, et définie de la manière suivante. Soient $\varphi : F \to F'$ et $\psi : G \to G'$ deux transformations naturelles, avec F et F' sont des foncteurs de $\mathfrak{C}$ dans $\mathfrak{D}$ et G et G' sont des foncteurs de $\mathfrak{D}$ dans $\mathfrak{E}$. La composée horizontale de ψ par φ est la transformation naturelle $(\psi\ \varphi) : G \circ F \to G' \circ F'$ définie par $(\psi\ \varphi)_A = \psi_{F'(A)} \circ G(\varphi_A)$. On peut montrer que $(\psi\ \varphi)_A = G'(\varphi_A) \circ \psi_{F(A)}$.

1.3.3 Catégories bi-cartésiennes fermées et transformations naturelles

Nous pouvons donner une nouvelle caractérisation des biCCC en utilisant des isomorphismes naturels (voir Asperti-Longo [6]). Elle nous permettra notamment de montrer plus facilement les isomorphismes usuels dans la section 1.4.

Proposition 1.29 (caractérisation des catégories cartésiennes) Une catégorie $\mathfrak{C}$ est cartésienne si et seulement si
- elle contient un objet terminal
- et pour tout couple $(A,\ B)$ d'objets de $\mathfrak{C}$, il existe un objet $A \times B$ et un isomorphisme naturel
$$\langle .,. \rangle : \mathfrak{C} \times \mathfrak{C}(-,\ (A,\ B)) \circ \Delta \to \mathfrak{C}(-,\ A \times B)$$
où $\boldsymbol{\Delta}$ est le ***foncteur diagonal*** est défini par $\begin{array}{rccc} \Delta & : & \mathfrak{C} & \to & \mathfrak{C} \times \mathfrak{C} \\ & & C & \mapsto & (C,\ C) \\ & & f & \mapsto & (f,\ f) \end{array}$

Proposition 1.30 (caractérisation des CCC) Une catégorie $\mathfrak{C}$ est cartésienne fermée si et seulement si elle est *cartésienne* et
pour tout couple $(A,\ B)$ d'objets de $\mathfrak{C}$, il existe un objet B^A et un isomorphisme naturel
$$\boldsymbol{\Lambda} : \mathfrak{C}(- \times A,\ B) \to \mathfrak{C}(-,\ B^A)$$

Proposition 1.31 (caractérisation des biCCC) Une catégorie $\mathfrak{C}$ est bi-cartésienne fermée si et seulement si
- elle est cartésienne fermée
- elle contient un objet initial
- et pour tout couple $(A,\ B)$ d'objets de $\mathfrak{C}$, il existe un objet $A + B$ et un isomorphisme naturel
$$(\dot{.}) : \mathfrak{C} \times \mathfrak{C}((A,\ B),\ -) \circ \Delta \to \mathfrak{C}(A + B,\ -)$$

Les trois dernières propositions nous permettent d'énoncer le corollaire suivant :

Proposition 1.32 Soit $\mathfrak{C}$ une biCCC, et A, B, et C des objets de $\mathfrak{C}$.
- $\mathfrak{C} \times \mathfrak{C}((C,\ C),\ (A,\ B))$ est isomorphe à $\mathfrak{C}(C,\ A \times B)$ naturellement en C
- $\mathfrak{C}(A \times B,\ C)$ est isomorphe à $\mathfrak{C}(A,\ C^B)$ naturellement en A

1.4 Isomorphismes dans les biCCC

Dans cette section, nous allons montrer les isomorphismes connus des catégories bi-cartésiennes fermées, en donnant le plus souvent possible explicitement les flèches réalisant ces isomorphismes. L'un des résultats principaux de la thèse sera de montrer dans le chapitre 4 que la théorie engendrée par ces isomorphismes n'est pas complète.

Remarque 1.33 Pour trouver les flèches réalisant les isomorphismes, il est souvent pratique de chercher d'abord les λ-termes réalisant les isomorphismes entre les types correspondants (qui sont beaucoup plus simple à trouver intuitivement), et d'utiliser la sémantique présentée dans la section 2.2.2 page 57.

Remarque 1.34 Les énoncés des propositions de cette section sont de la forme

Pour tous les objets A, B, C de $\mathfrak{C}$, $F(A,B,C)$ est isomorphe à $G(A,B,C)$

ce qui revient à dire

$F(A,B,C)$ est isomorphe à $G(A,B,C)$ naturellement en A, B, C

En effet, $F(A,B,C)$ peut être vu tour-à-tour comme chacun des foncteurs $F(-,B,C)$, $F(A,-,C)$, et $F(A,B,-)$, définis par

$$\begin{array}{rcccc} -\times B & : & \mathfrak{C} & \to & \mathfrak{C} \\ & & A & \mapsto & A\times B \\ & & f & \mapsto & f\times id \end{array} \qquad \begin{array}{rcccc} A\times - & : & \mathfrak{C} & \to & \mathfrak{C} \\ & & B & \mapsto & A\times B \\ & & f & \mapsto & id\times f \end{array}$$

$$\begin{array}{rcccc} -+B & : & \mathfrak{C} & \to & \mathfrak{C} \\ & & A & \mapsto & A+B \\ & & f & \mapsto & \binom{\iota_1\circ f}{\iota_2\circ id} \end{array} \qquad \begin{array}{rcccc} A+- & : & \mathfrak{C} & \to & \mathfrak{C} \\ & & B & \mapsto & A+B \\ & & f & \mapsto & \binom{\iota_1\circ id}{\iota_2\circ f} \end{array}$$

$$\begin{array}{rcccc} B^{(-)} & : & \mathfrak{C}^{op} & \to & \mathfrak{C} \\ & & A & \mapsto & B^A \\ & & f & \mapsto & \Lambda(eval_{A,B}\circ(id\times f)) \end{array} \qquad \begin{array}{rcccc} (-)^A & : & \mathfrak{C} & \to & \mathfrak{C} \\ & & B & \mapsto & B^A \\ & & f & \mapsto & \Lambda(f\circ eval_{A,B}) \end{array}$$

$$\begin{array}{ccc} B^A & & \\ \downarrow{\scriptstyle \Lambda(eval_{A,B}\circ(id\times f))} & & \\ B^{A'} & & \end{array} \qquad \begin{array}{ccccc} B^A\times A' & \xrightarrow{id\times f} & B^A\times A & \xrightarrow{eval_{A,B}} & B \\ \downarrow & & & \nearrow_{eval_{A',B}} & \\ B^{A'}\times A' & & & & \end{array}$$

$$\begin{array}{ccc} B^A & & \\ \downarrow{\scriptstyle \Lambda(f\circ eval_{A,B})} & & \\ C^A & & \end{array} \qquad \begin{array}{ccccc} B^A\times A & \xrightarrow{eval_{A,B}} & B & \xrightarrow{f} & C \\ \downarrow & & & \nearrow_{eval_{A,C}} & \\ C^A\times A & & & & \end{array}$$

Nous allons pouvoir utiliser le lemme suivant :

Lemme 1.35 Si $F:\mathfrak{C}\to\mathfrak{D}$, $F':\mathfrak{C}\to\mathfrak{D}$ sont des foncteurs isomorphes
et si $G:\mathfrak{D}\to\mathfrak{E}$ et $G':\mathfrak{D}\to\mathfrak{E}$ sont isomorphes,
alors leurs composées $G\circ F$ et $G'\circ F'$ sont isomorphes.

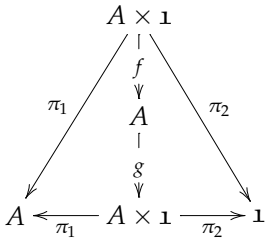
Pour démontrer ce lemme, il suffit de prendre la composée horizontale des isomorphismes naturels.

Proposition 1.36 Dans une catégorie cartésienne, pour tous les objets A, B, et C, les isomorphismes suivants sont vérifiés :

$$A \cong A \times \mathbb{1}$$
$$A \times B \cong B \times A$$
$$A \times (B \times C) \cong (A \times B) \times C$$

Démonstration :

▶ $A \cong A \times \mathbb{1}$ Prenons $f = \pi_1^{A,\mathbb{1}}$ et $g = \langle id_A, !A \rangle$. La composition $f \circ g$ est bien l'identité de A (propriété du produit). Pour montrer que l'autre composition est aussi l'identité, nous allons montrer qu'elle fait commuter le diagramme suivant :



On a $\pi_1 \circ g \circ f = \pi_1 \circ (\langle id_A, !A \rangle) \circ \pi_1 = \pi_1 \circ id_A = \pi_1$ et $\pi_2 \circ g \circ f = \pi_2$ (par unicité de la flèche entre $A \times \mathbb{1}$ et $\mathbb{1}$).

▶ $A \times B \cong B \times A$ On vérifie facilement que les flèches $\langle \pi_2^{A,B}, \pi_1^{A,B} \rangle$ et $\langle \pi_2^{B,A}, \pi_1^{B,A} \rangle$ forment l'isomorphisme recherché.

▶ $A \times (B \times C) \cong (A \times B) \times C$ L'isomorphisme et son inverse sont les flèches $\langle \pi_1 \circ \pi_1, \langle \pi_2 \circ \pi_1, \pi_2 \rangle \rangle$ et $\langle \langle \pi_1, \pi_1 \circ \pi_2 \rangle, \pi_2 \circ \pi_2 \rangle$. ■

Proposition 1.37 Dans une catégorie cartésienne fermée, les isomorphismes suivants sont vrais pour tous les objets A, B, C.

$$C^{A \times B} \cong (C^B)^A$$
$$(B \times C)^A \cong B^A \times C^A$$
$$\mathbb{1}^A \cong \mathbb{1}$$
$$A^{\mathbb{1}} \cong A$$

Démonstration :

▶ $C^{A \times B} \cong (C^B)^A$ Nous pouvons une fois de plus donner les flèches et vérifier que ce sont des isomorphismes. Pour changer, nous allons utiliser la propriété 1.28 page 33. Il suffit de montrer que $\mathbf{y}_{C^{A \times B}} \cong \mathbf{y}_{(C^B)^A}$.

Or $\mathbf{y}_{(C^B)^A}(X) = \mathfrak{C}(X,\ (C^B)^A)$ est isomorphe à $\mathfrak{C}(X \times A,\ C^B)$, naturellement en X, d'après la proposition 1.32 page 34.
De même $\mathfrak{C}(Y,\ C^B)$ est isomorphe à $\mathfrak{C}(Y \times B,\ C)$, naturellement en Y, d'où l'on peut déduire assez simplement que $\mathfrak{C}(X \times A,\ C^B)$ est isomorphe à $\mathfrak{C}((X \times A) \times B,\ C)$, naturellement en X.

Enfin $\mathbf{y}_{C^{A\times B}}(X) = \mathfrak{C}(X,\ C^{A\times B})$ est isomorphe à $\mathfrak{C}(X \times (A \times B),\ C)$, naturellement en X.
On peut conclure en utilisant l'isomorphisme d'associativité du produit et le lemme 1.35 page 35.

▶ $(B \times C)^A \cong B^A \times C^A$ En utilisant encore une fois la propriété 1.28, il suffit de montrer que $\mathbf{y}_{(B\times C)^A} \cong \mathbf{y}_{B^A\times C^A}$.
Or $\mathbf{y}_{(B\times C)^A}(X) = \mathfrak{C}(X,\ (B \times C)^A)$ est isomorphe à $\mathfrak{C}(X \times A,\ B \times C)$, naturellement en X.
L'ensemble $\mathbf{y}_{B^A\times C^A}(X) = \mathfrak{C}(X,\ B^A \times C^A)$ est isomorphe à $\mathfrak{C} \times \mathfrak{C}((X,\ X),\ (B^A,\ C^A))$, qui est isomorphe à $\mathfrak{C}(X,\ B^A) \times \mathfrak{C}(X,\ C^A)$ donc à $\mathfrak{C}(X \times A,\ B) \times \mathfrak{C}(X \times A,\ C)$ et à $\mathfrak{C}(X \times A,\ B \times C)$, toujours naturellement en X.

▶ $\mathfrak{1}^A \cong \mathfrak{1}$ Nous voulons montrer que $\mathfrak{1}^A$ est terminal.
Pour cela, prenons un objet B quelconque. Il existe une unique flèche entre $B \times A$ et $\mathfrak{1}$, notée $!(B \times A)$. La flèche $\Lambda(!(B \times A))$ va de B à $\mathfrak{1}^A$.
Reste à montrer l'unicité. Supposons que g va de B à $\mathfrak{1}^A$. On a $\Lambda^{-1}(g) = !(B \times A)$, donc $\Lambda(\Lambda^{-1}(g)) = \Lambda(!(B \times A))$, ce qui implique $g = \Lambda(!(B \times A))$, parce que Λ est une bijection.

▶ $A^{\mathfrak{1}} \cong A$ En utilisant la propriété 1.28 page 33, il suffit de montrer que $\mathbf{y}_A \cong \mathbf{y}_{A^{\mathfrak{1}}}$. Or $\mathbf{y}_{A^{\mathfrak{1}}}(X) = \mathfrak{C}(X,\ A^{\mathfrak{1}})$ est isomorphe à $\mathfrak{C}(X \times \mathfrak{1},\ A)$, qui lui même est isomorphe à $\mathfrak{C}(X,\ A)$ naturellement en X (car $X \times \mathfrak{1} \cong X$).

■

Proposition 1.38 Dans une catégorie co-cartésienne, pour tous les objets A, B, et C, les isomorphismes suivants sont vérifiés :

$$A + B \cong B + A$$
$$A + (B + C) \cong (A + B) + C$$
$$A + \mathfrak{o} \cong A$$

Démonstration :

▶ $A + B \cong B + A$ On vérifie que $\binom{\iota_2^{A,B}}{\iota_1^{A,B}}$ et $\binom{\iota_2^{B,A}}{\iota_1^{B,A}}$ sont les fonctions recherchées.

▶ $A + (B + C) \cong (A + B) + C$ Cette fois-ci, les témoins de l'isomorphisme sont :

$$\binom{\iota_1 \circ \iota_1}{\binom{\iota_1 \circ \iota_2}{\iota_2}} : A + (B + C) \to (A + B) + C$$

et

$$\binom{\binom{\iota_1}{\iota_2 \circ \iota_1}}{\iota_2 \circ \iota_2} : (A + B) + C \to A + (B + C)$$

▶ $A + \mathfrak{o} \cong A$ Les témoins sont ι_1 et $\alpha = \binom{id}{!_A}$. On vérifie d'abord que $\alpha \circ \iota_1 = id$ (propriété du co-produit). Ensuite on peut montrer que $\iota_1 \circ \alpha \circ \iota_1 = \iota_1$ en utilisant le résultat précédent, et $\iota_1 \circ \alpha \circ \iota_2 = !(A + \mathfrak{o}) = \iota_2$ par unicité. Donc $\iota_1 \circ \alpha = id$ (propriété du produit). ■

Proposition 1.39 Dans une catégorie cartésienne fermée avec
- un objet initial $\mathfrak{o}$
- et un isomorphisme entre $B \times \mathfrak{o}$ et $\mathfrak{o}$ pour tout objet B,

l'isomorphisme suivant est vérifié pour tout objet A :

$$A^{\mathfrak{o}} \cong \mathfrak{1}$$

Démonstration :

Nous voulons montrer que $A^{\mathfrak{o}}$ est terminal.
Pour cela, prenons un objet B quelconque. Nous savons que $B \times \mathfrak{o}$ est initial. Il existe donc une unique fonction f entre $B \times \mathfrak{o}$ et A.
L'existence de l'exponentielle $A^{\mathfrak{o}}$ nous donne l'existence d'une flèche $\Lambda(f)$ entre B et $A^{\mathfrak{o}}$.
Reste à montrer l'unicité. Soit $g : B \to A^{\mathfrak{o}}$. On a $\Lambda^{-1}(g) = f$ par unicité de f, et donc $\Lambda(\Lambda^{-1}(g)) = \Lambda(f)$. Ce qui implique $g = \Lambda(f)$ (Λ étant une bijection). ■

Toutes les catégories bi-cartésiennes ne sont pas distributives. En revanche, la distibutivité est vérifiée dans les catégories bi-cartésiennes *fermées*.

Proposition 1.40 Une catégorie bi-cartésienne fermée est distributive et les flèches

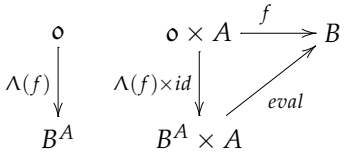
$$\delta = \binom{id_A \times \iota_1}{id_A \times \iota_2} : (A \times B) + (A \times C) \to A \times (B + C)$$

$$\gamma = \Lambda^{-1}\binom{\Lambda(\iota_1 \circ \langle \pi_2, \pi_1 \rangle)}{\Lambda(\iota_2 \circ \langle \pi_2, \pi_1 \rangle)} \circ \langle \pi_2, \pi_1 \rangle : A \times (B + C) \to (A \times B) + (A \times C)$$

sont les témoins de l'isomorphisme entre $A \times (B + C)$ et $(A \times B) + (A \times C)$.

Démonstration :

▶ Montrer que $A \times \mathfrak{o}$ est isomorphe à $\mathfrak{o}$, revient à montrer que $\mathfrak{o} \times A$ est initial. Or nous savons qu'il existe une unique flèche entre $\mathfrak{o}$ et B^A, notée $\mathfrak{i}B^A$. Donc il existe une flèche entre $\mathfrak{o} \times A$ et B qui est $\Lambda^{-1}(\mathfrak{i}B^A)$, et cette flèche est unique, car si $f : \mathfrak{o} \times A \to B$, alors d'après la propriété de l'exponentielle, $f = eval \circ (\Lambda(f) \times id)$, et, $\mathfrak{o}$ étant initial, $\Lambda(f) = \mathfrak{i}B^A$ indépendamment de f.



▶ Pour montrer la propriété de distributivité, calculons d'abord $\gamma \circ \delta$.
Les flèches $h_B = \Lambda(\iota_1 \circ \langle \pi_2, \pi_1 \rangle)$ et $h_C = \Lambda(\iota_2 \circ \langle \pi_2, \pi_1 \rangle)$, vérifient

$$eval \circ (h_B \times id_A) = \iota_1 \circ \langle \pi_2, \pi_1 \rangle$$

et

$$eval \circ (h_C \times id_A) = \iota_2 \circ \langle \pi_2, \pi_1 \rangle$$

On a :

$$\begin{aligned}
\gamma \circ \delta &= \Lambda^{-1}\begin{pmatrix} h_B \\ h_C \end{pmatrix} \circ \langle \pi_2, \pi_1 \rangle \circ \begin{pmatrix} id_A \times \iota_1 \\ id_A \times \iota_2 \end{pmatrix} \\
&= eval \circ (\begin{pmatrix} h_B \\ h_C \end{pmatrix} \times id) \circ \langle \pi_2, \pi_1 \rangle \circ \begin{pmatrix} id_A \times \iota_1 \\ id_A \times \iota_2 \end{pmatrix} \\
&= eval \circ \langle \begin{pmatrix} h_B \\ h_C \end{pmatrix} \circ \pi_2, \pi_1 \rangle \circ \begin{pmatrix} id_A \times \iota_1 \\ id_A \times \iota_2 \end{pmatrix} \\
&= eval \circ \begin{pmatrix} \langle \binom{h_B}{h_C} \circ \pi_2, \pi_1 \rangle \circ (id_A \times \iota_1) \\ \langle \binom{h_B}{h_C} \circ \pi_2, \pi_1 \rangle \circ (id_A \times \iota_2) \end{pmatrix} \\
&= eval \circ \begin{pmatrix} \langle \binom{h_B}{h_C} \circ \iota_1 \circ \pi_2, \pi_1 \rangle \\ \langle \binom{h_B}{h_C} \circ \iota_2 \circ \pi_2, \pi_1 \rangle \end{pmatrix} \\
&= eval \circ \begin{pmatrix} \langle h_B \circ \pi_2, \pi_1 \rangle \\ \langle h_C \circ \pi_2, \pi_1 \rangle \end{pmatrix} \\
&= \begin{pmatrix} eval \circ (h_B \times id_A) \circ \langle \pi_2, \pi_1 \rangle \\ eval \circ (h_C \times id_A) \circ \langle \pi_2, \pi_1 \rangle \end{pmatrix} \\
&= \begin{pmatrix} \iota_1 \\ \iota_2 \end{pmatrix} \\
&= id
\end{aligned}$$

▶ Pour calculer $\delta \circ \gamma$, nous aurons besoin du petit lemme suivant : Pour toutes flèches f et g,

$$g \circ \Lambda^{-1}(f) = \Lambda^{-1}(\Lambda(g \circ eval) \circ f)$$

En effet, le membre de droite est égal à $eval \circ (\Lambda(g \circ eval) \times id) \circ (f \times id)$, qui est lui même égal à $g \circ eval \circ (f \times id)$. Enfin il est facile de voir que le membre de gauche est lui aussi égal à cette dernière expression.
Nous avons donc

$$\begin{aligned}
\delta \circ \gamma &= \begin{pmatrix} id_A \times \iota_1 \\ id_A \times \iota_2 \end{pmatrix} \circ \Lambda^{-1}\begin{pmatrix} h_B \\ h_C \end{pmatrix} \circ \langle \pi_2, \pi_1 \rangle \\
&= \Lambda^{-1}\left(\Lambda\left(\begin{pmatrix} id_A \times \iota_1 \\ id_A \times \iota_2 \end{pmatrix} \circ eval\right) \circ \begin{pmatrix} h_B \\ h_C \end{pmatrix}\right) \circ \langle \pi_2, \pi_1 \rangle \text{ d'après le petit lemme ci-dessus} \\
&= \Lambda^{-1}(\begin{pmatrix} \Lambda(f) \circ h_B \\ \Lambda(f) \circ h_C \end{pmatrix}) \circ \langle \pi_2, \pi_1 \rangle
\end{aligned}$$

Or

$$\begin{aligned}
\Lambda^{-1}(\Lambda(f) \circ h_B) &= eval \circ (\Lambda(f) \times id) \circ (h_B \times id) \\
&= f \circ (h_B \times id) \\
&= \begin{pmatrix} id_A \times \iota_1 \\ id_A \times \iota_2 \end{pmatrix} \circ eval \circ (h_B \times id) \\
&= \begin{pmatrix} id_A \times \iota_1 \\ id_A \times \iota_2 \end{pmatrix} \circ \iota_1 \circ \langle \pi_2, \pi_1 \rangle \\
&= (id_A \times \iota_1) \circ \langle \pi_2, \pi_1 \rangle
\end{aligned}$$

Donc $\Lambda(f) \circ h_B = \Lambda((id_A \times \iota_1) \circ \langle \pi_2, \pi_1 \rangle)$.
De même $\Lambda(f) \circ h_C = \Lambda((id_A \times \iota_2) \circ \langle \pi_2, \pi_1 \rangle)$. On en déduit

$$\delta \circ \gamma = \Lambda^{-1}(\begin{pmatrix} \Lambda((id_A \times \iota_1) \circ \langle \pi_2, \pi_1 \rangle) \\ \Lambda((id_A \times \iota_2) \circ \langle \pi_2, \pi_1 \rangle) \end{pmatrix}) \circ \langle \pi_2, \pi_1 \rangle$$

Or

$$\begin{aligned} \Lambda((id_A \times \iota_1) \circ \langle \pi_2, \pi_1 \rangle) &= \Lambda(\langle \pi_2, \pi_1 \rangle \circ (\iota_1 \times id_A)) \\ &= \Lambda\left(\Lambda^{-1}(\Lambda\langle \pi_2, \pi_1 \rangle) \circ (\iota_1 \times id_A)\right) \\ &= \Lambda(eval \circ (\Lambda\langle \pi_2, \pi_1 \rangle \circ (\iota_1 \times id_A)) \\ &= \Lambda\left(\Lambda^{-1}(\Lambda\langle \pi_2, \pi_1 \rangle \circ \iota_1)\right) \\ &= \Lambda\langle \pi_2, \pi_1 \rangle \circ \iota_1 \end{aligned}$$

De même $\Lambda((id_A \times \iota_1) \circ \langle \pi_2, \pi_1 \rangle) = \Lambda\langle \pi_2, \pi_1 \rangle \circ \iota_2$.
Donc

$$\begin{aligned} \delta \circ \gamma &= \Lambda^{-1}\begin{pmatrix} \Lambda\langle \pi_2, \pi_1 \rangle \circ \iota_1 \\ \Lambda\langle \pi_2, \pi_1 \rangle \circ \iota_2 \end{pmatrix} \circ \langle \pi_2, \pi_1 \rangle \\ &= \Lambda^{-1}(\Lambda\langle \pi_2, \pi_1 \rangle) \circ \langle \pi_2, \pi_1 \rangle \\ &= id \end{aligned}$$

■

Proposition 1.41 Dans une catégorie bi-cartésienne fermée, l'isomorphisme suivant est vrai pour tous les objets A, B, C.

$$A^{(B+C)} \cong A^B \times A^C$$

Démonstration :

On vérifie que

$$\langle \Lambda(eval \circ \langle \pi_1, \iota_1 \circ \pi_2 \rangle), \Lambda(eval \circ \langle \pi_1, \iota_2 \circ \pi_2 \rangle) \rangle : A^{(B+C)} \to A^B \times A^C$$

et

$$\Lambda(\begin{smallmatrix} eval \circ \langle \pi_1 \circ \pi_1, \pi_2 \rangle \\ eval \circ \langle \pi_2 \circ \pi_1, \pi_2 \rangle \end{smallmatrix}) \circ \delta : A^B \times A^C \to A^{(B+C)}$$

sont mutuellement inverses.

Ces flèches correspondent respectivement aux interprétations des λ-termes

$$[\![f : B + C \to A \ \vdash \ (\lambda b.\ f\ \ (in_1\ \ b),\ \lambda c.\ f\ \ (in_2\ \ c)) : (B \to A) \times (C \to A)]\!]$$

et

$$[\![p : (B \to A) \times (C \to A) \ \vdash \ \lambda s.\ case\,(s,\ x.\ ((proj_1\ \ p)\ \ x),\ y.\ ((proj_2\ \ p)\ \ y)) : B + C \to A]\!]$$

(voir page 57)

La preuve dans les catégories est un peu fastidieuse. Il est beaucoup plus simple de faire la

preuve en utilisant les λ-termes (voir chapitre suivant). ■

La figure 1.1 récapitule les isomorphismes que nous venons de prouver dans les biCCC.

1.	$A \times B$	$\cong$	$B \times A$
2.	$A \times (B \times C)$	$\cong$	$(A \times B) \times C$
3.	$C^{A \times B}$	$\cong$	$(C^B)^A$
4.	$(B \times C)^A$	$\cong$	$B^A \times C^A$
5.	$A \times 1$	$\cong$	A
6.	1^A	$\cong$	1
7.	A^1	$\cong$	A
8.	$A + B$	$\cong$	$B + A$
9.	$A + (B + C)$	$\cong$	$(A + B) + C$
10.	$(A \times B) + (A \times C)$	$\cong$	$A \times (B + C)$
11.	$A^{(B+C)}$	$\cong$	$A^B \times A^C$
12.	$A \times 0$	$\cong$	0
13.	$A + 0$	$\cong$	A
14.	A^0	$\cong$	1

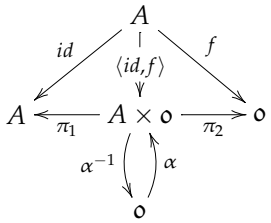
FIG. 1.1 – Quelques isomorphismes vrais dans les biCCC.

Proposition 1.42 Dans une catégorie distributive, s'il existe une flèche entre un objet A et un objet initial 0, alors A est isomorphe à 0. Notamment A est initial.

On dira qu'un objet initial est ***strict***.

Démonstration :

Rappelons que dans une catégorie distributive, il existe un isomorphisme $\alpha : 0 \to (A \times 0)$. Supposons $f : A \to 0$, et posons $g = \pi_1 \circ \alpha : 0 \to A$ et $h = \alpha^{-1} \circ \langle id_A, f \rangle : A \to 0$.



$h \circ g$ est une flèche de 0 vers 0. Il existe une seule flèche i_0 de 0 vers 0 qui est aussi égale à id_0. Pour l'autre composition :

$$g \circ h \quad = \quad \pi_1 \circ \alpha \circ \alpha^{-1} \circ \langle id_A, f \rangle$$

$$
\begin{aligned}
&= \pi_1 \circ \langle id_A, f \rangle \\
&= id_A
\end{aligned}
$$

■

Chapitre 2

Modèles du λ-calcul

Résumé

Ce chapitre rappelle d'abord succinctement les définitions et quelques résultats sur le λ-calcul simplement typé avec type somme. Le problème de normalisation en présence de sommes fortes est abordé. Ensuite, différents modèles catégoriques du λ-calcul sont présentés. À la fin du chapitre, je montrerai en détails que le modèle dit « des λ-termes » est isomorphe à la catégorie bi-cartésienne fermée libre engendrée à partir des types de base, ce qui permettra d'identifier le problème de la recherche des isomorphismes de types avec celui de la recherche des isomorphismes dans les catégories bi-cartésiennes fermées.

Le λ-CALCUL est un langage mathématique qui a donné naissance à une catégorie de langages de programmation comme *Lisp* ou *Objective Caml*, appelés langages fonctionnels. Il permet au théoricien des langages de programmation de raisonner sur un objet mathématique sans se préoccuper de toute la complexité d'un langage dédié à la production d'applications. Dans sa version de base, le λ-calcul est un langage extrêmement simple comprenant uniquement une notion de *variable*, une notion d'*abstraction* et une notion d'*application*. Tous les termes sont construits à partir de ces trois briques de base. Pour avoir une notion de *calcul*, ou encore d'*évaluation*, on définit une relation de *réécriture* en une seule règle appelée β-réduction. Malgré cette étonnante simplicité, ce langage a la puissance de calcul des machines de Turing, c'est-à-dire celle de n'importe quel langage de programmation. Il est en particulier possible de coder les entiers sous la forme de λ-termes, par exemple en utilisant ce que l'on appelle les ***entiers de Church***[1].

Cependant pour obtenir un langage réellement utilisable pour programmer, les langages fonctionnels ajoutent un grand nombre de constructions prédéfinies, par exemple les entiers et des fonctions pour les manipuler, ou encore des structures de données comme le produit ou la somme, et leurs constructeurs et destructeurs associés.

Enfin pour faciliter la tâche du programmeur, les langages évolués utilisent une notion de

[1] voir figure 6.7 page 152

typage permettant d'interdire tout ou partie des termes mal formés, qui produiraient une erreur à l'exécution.

Nous allons nous intéresser ici au λ-calcul simplement typé avec types produit et unité ainsi que types somme et zéro.

Une fois définies la syntaxe du calcul et la notion de réduction associée, il nous faut en définir la sémantique. Disons rapidement qu'il s'agit de « donner un sens » aux termes, compatible avec la notion de réduction, en leur associant une *interprétation* mathématique. À un même langage on peut donner plusieurs sémantiques différentes, en fonction des propriétés que l'on cherche à prouver.

Après avoir défini le λ-calcul et rappelé ses liens avec la logique, je présenterai dans ce chapitre ses sémantiques dans les catégories bi-cartésiennes fermées. Nous verrons que toute biCCC est un modèle du λ-calcul. Enfin, dans la section 2.3, je m'attacherai à prouver en détail, pour des raisons didactiques, que le modèle le moins *abstrait*, dit *modèle syntaxique*, est lui-même une biCCC qui est isomorphe à la biCCC libre engendrée sur les types de base (voir Lambek et Scott [60]). Ce résultat a une importance particulière pour l'étude qui nous intéresse puisqu'il permet de déduire que les isomorphismes de types sont exactement les isomorphismes vrais dans toutes les catégories bi-cartésiennes fermées.

2.1 Le λ-calcul avec type somme

2.1.1 Le λ-calcul pur

La syntaxe du λ-calcul est définie de la manière suivante :

Définition 2.1 (λ-terme) Étant donné un ensemble infini dénombrable dit ensemble de ***variables***, un ***λ-terme*** est
- soit une variable,
- soit un terme de la forme $(t_1 \ @ \ t_2)$, plus souvent noté $(t_1 \ \ t_2)$, appelé ***application*** de t_1 à t_2, (où t_1 et t_2 sont eux-mêmes des λ-termes),
- soit un terme de la forme $(\lambda x.\ t)$, appelé ***λ-abstraction***, (où x est une variable et t un λ-terme, appelé ***corps*** de l'abstraction).

Par exemple, $(\lambda x.\ \lambda y.\ (y \ \ x))$ et $(x \ \ (\lambda y.\ x))$ sont des λ-termes. Les parenthèses sont utilisées pour éviter toute ambiguïté dans la lecture.

Nous devrons faire attention à certains problèmes :
- tout d'abord, les termes sont définis modulo une relation d'équivalence sur les noms de variable, que l'on appellera ***α-équivalence***. Par exemple les termes $(\lambda x.\ x)$ et $(\lambda y.\ y)$ sont α-équivalents.
- ensuite les problèmes de ***capture*** de noms de variables, mis en évidence dans l'exemple suivant : $(\lambda x.\ \lambda x.\ x)$. Ici l'occurrence de x renvoie au deuxième λ. On pourra supposer pour simplifier que les λ sont suivis de noms de variables distincts ce qui évitera ces problèmes. Il faudra être vigilant sur ce problème lorsque l'on pratiquera la β-réduction et donc la substitution (voir ci-après).

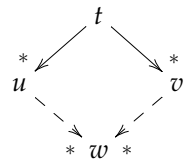
Pour terminer, nous allons définir quelques notions concernant les variables dans les λ-termes. Tout d'abord, nous dirons qu'une occurrence d'une variable x est ***liée*** si elle apparaît *sous un* λx (c'est-à-dire dans le corps de l'abstraction). Dans le cas contraire, l'occurrence sera dite ***libre***.

Une ***variable libre*** dans un terme t est une variable ayant au moins une occurrence libre dans t. Un terme sans variable libre est dit ***clos***. On notera $\boldsymbol{FV(t)}$ l'ensemble des variables libres d'un terme t.

Enfin on notera $\boldsymbol{t\,[^{u}/_{x}]}$ le terme obtenu en substituant dans t toutes les occurrences *libres* de la variable x par le terme u.

Réécriture

Rappelons d'abord quelques définition de réécriture. Pour définir un langage de programmation, nous devons définir une notion d'*exécution* dite aussi *réduction*. Pour cela, nous allons utiliser des *règles de réécriture*. Une ***règle de réécriture*** est un couple de termes $(t,\ u)$ noté sous la forme $t \to u$. Tout terme obtenu par instanciation des variables libres de la partie gauche d'une règle de réécriture est appelé un ***redex***. Un ensemble de règles de réécriture forme ce que l'on appelle un ***système de réécriture***, et définit une notion de ***réduction*** sur les termes du langage. On dira que le terme $\rho(t)$, obtenu par instanciation des variables de t, *se réduit* (en une étape) en $\rho(u)$, ce que l'on notera $\rho(t) \to \rho(u)$. Si $\rho(u)$ est lui-même un redex pour une autre règle (ou la même), on peut poursuivre le processus de réduction. On notera $t_1 \to^* t_2$ si le terme t_1 se réduit en t_2 en zéro, une ou plusieurs étape. Un système de réécriture est dit ***confluent*** si pour tous termes t, u et v tels que $t \to^* u$ et $t \to^* v$, alors il existe un terme w tel que $u \to^* w$ et $v \to^* w$.



Si un tel w n'existe pas, la paire de termes $\{\ u,\ v\ \}$ est appelée ***paire critique***.

Un système de réécriture définit une relation d'équivalence $=$ sur les termes par clôture transitive, réflexive et symétrique. On dira qu'un système vérifie la ***propriété de Church-Rosser*** si quels que soient deux termes u et v équivalents, il existe un terme w tel que $u \to^* w$ et $v \to^* w$ (autrement dit : pas de paire critique).

Un système de réécriture est dit ***fortement normalisant*** s'il n'existe aucune réduction infinie.

Un terme est dit en ***forme normale*** s'il ne contient aucun redex. Si un système est confluent et fortement normalisant, tout terme admet une forme normale unique, obtenue en appliquant les règles de réécriture dans un ordre quelconque tant qu'il reste des redex.

Lorsqu'un système de réécriture n'est pas confluent, on pourra parfois fixer une stratégie de réduction en imposant l'ordre d'application des règles de manière à obtenir une réduction confluente.

β-réduction

Au langage défini par les λ-termes on associe le système de réécriture formé par la seule règle suivante :

$$\beta_{\to} \qquad (\lambda x.\ t)\ \ u \quad \longrightarrow \quad t\,[^{u}/_{x}]$$

La réécriture associée à cette règle est appelée ***β-réduction***.

Proposition 2.2 La β-réduction dans le λ-calcul pur est confluente et a la propriété de Church-Rosser.

Cependant il est facile de voir que la β-réduction dans le λ-calcul pur ne termine pas toujours (considérer par exemple le terme $\Omega = (\Delta\ \ \Delta)$ avec $\Delta = \lambda x.\ (x\ \ x)$).

Extensionalité et η-réduction

Au système de réécriture ci-dessus, on ajoute souvent la règle $\eta_\to$ suivante :

$$\eta_\to \qquad \lambda x.\ (M\ \ x) \quad \longrightarrow \quad M$$

Cette règle est appelée ***η-réduction***. La règle orientée dans l'autre sens est appelée ***η-expansion***. Lorsque nous ne voudrons pas préciser le sens des règles, nous parlerons d'***η-conversion***. Le système de réécriture obtenu avec les deux règles $\beta_\to$ et $\eta_\to$ est appelé $\boldsymbol{\beta\eta}$.

Proposition 2.3 Le λ-calcul pur est confluent pour $\beta\eta$.

Puissance de calcul du λ-calcul pur

L'intérêt du λ-calcul réside en grande partie dans le fait qu'il permet de coder les entiers et n'importe quelle fonction récursive partielle de $\mathbb{N}^k$ dans $\mathbb{N}$. Autrement dit, il a la même puissance de calcul que les machines de Turing. Pour plus de détails là-dessus, voir par exemple le livre de Jean-Louis Krivine [59].

2.1.2 Le λ-calcul simplement typé

Afin de construire un langage de programmation plus évolué, nous devons d'une part associer au λ-calcul un système de *typage*, permettant d'exprimer des propriétés simples sur les termes et d'en interdire certains, et d'autre part ajouter à la syntaxe des constructions permettant par exemple de définir des couples de termes.

Nous allons donner dans cette section une présentation rapide du *λ-calcul simplement typé*, qui est obtenu en associant au λ-calcul pur un système de typage. On pourra se référer par exemple aux livres de Lambek et Scott [60] ou de Krivine [59] pour plus de détails. Dans ce système, on ne peut plus définir comme avant par des termes du λ-calcul les notions de couples de termes et de projections. Pour résoudre ce problème, on ajoute ces constructions au langage. Dans la plupart des ouvrages, le λ-calcul simplement typé est présenté avec un type produit cartésien, permettant de typer des couples de termes, mais sans la notion duale d'*union disjointe* (appelée aussi *somme disjointe*). La raison à cela est que les sommes sont beaucoup plus difficiles à manipuler que les produits car elles ne se comportent pas aussi bien vis-à-vis notamment de la β-réduction, comme nous le verrons plus loin. Notre but ici est d'étudier cette notion de somme, donc nous allons définir le λ-calcul simplement typé *avec* somme que nous distinguerons de celui *sans* type somme.

Types

Nous ne considérerons que le cas des sommes et produits *finis*, et pour cela nous définirons des constructeurs binaires. Le cas des constructeurs n-aires s'en déduit assez facilement . Nous nous intéresserons aussi au cas des sommes et produits vides. L'ensemble des types contiendra donc deux constantes, notées $\mathtt{1}$ et $\mathtt{0}$, pour dénoter respectivement le produit et la somme vides, ainsi qu'un ensemble dénombrable de types ***atomiques*** (appelés aussi ***types de base***). Enfin cet ensemble sera clos par les constructeurs de type flèche, produit et somme.

Nous pouvons résumer la syntaxe des types par la grammaire suivante :

$$\tau ::= \theta \mid \tau_1 \to \tau_2 \mid \mathtt{1} \mid \tau_1 \times \tau_2 \mid \mathtt{0} \mid \tau_1 + \tau_2$$

où θ représente un type de base. Le type `1` correspond au type `unit` de *Caml*.

Syntaxe

À chaque constructeur de type sont associées des constructions du langage, appelés ***destructeurs*** et ***constructeurs***. Ainsi la λ-abstraction est le constructeur associé au type flèche, alors que son destructeur est l'application. La syntaxe des termes du λ-calcul est donc étendue avec les constructeurs et destructeurs suivants :

- un élément noté $()$ (prononcez « nil ») seul élément (et constructeur) du type `1` — qui n'a pas de destructeur,
- un élément que l'on notera $\bot_{\boldsymbol{\tau}}(\boldsymbol{t})$, destructeur associé au type `0` — qui n'a pas de constructeur. Ici t est un λ-terme.
- $\boldsymbol{proj_i^{\tau_1,\tau_2}}$ est la i-ème projection, destructeur du type produit $\tau_1 \times \tau_2$. $\boldsymbol{(t_1,\ t_2)}$ est le constructeur associé à ce type,
- $\boldsymbol{in_1^{\tau_1,\tau_2}}$ et $\boldsymbol{in_2^{\tau_1,\tau_2}}$ sont les injections à gauche et à droite dans le type somme $\tau_1 + \tau_2$ (constructeur), et $\boldsymbol{case\,(t,\ x_1 : \tau_1.\ t_1,\ x_2 : \tau_2.\ t_2)}$ est le destructeur associé à ce type.

En résumé, la syntaxe des termes du λ-calcul simplement typé, en notation dite de Church, est donc la suivante :

$$\begin{aligned} t \quad ::= \quad & x \mid \lambda x : \tau.\, t \mid t_1\ t_2 \mid \\ & () \mid (t_1,\ t_2) \mid proj_1^{\tau_1,\tau_2}\ t \mid proj_2^{\tau_1,\tau_2}\ t \mid \\ & \bot_\tau(t) \mid in_1^{\tau_1,\tau_2}\ t \mid in_2^{\tau_1,\tau_2}\ t \mid case\,(t,\ x_1 : \tau_1.\ t_1,\ x_2 : \tau_2.\ t_2) \end{aligned}$$

où x (et x_1 et x_2) appartiennent à un ensemble dénombrable de variables.

Les symboles $in_1^{\tau_1,\tau_2}$ et $in_2^{\tau_1,\tau_2}$ seront souvent écrits plus simplement $\boldsymbol{in_1}$ et $\boldsymbol{in_2}$. De même, on écrira souvent $\boldsymbol{proj_i}$ quel que soit le type. Enfin, nous omettrons souvent les types dans l'écriture des abstractions et des *case*, utilisant ainsi la notation dite de Curry.

Remarquons que nos notations imposent que chaque branche d'un case soit une abstraction, empêchant par exemple d'écrire $case(x, f, g)$, où f et g sont des termes quelconques. Cette notation correspond au choix d'*Objective Caml* notamment. Cependant, pour simplifier les notations, nous écrirons souvent par exemple $case\,(t,\ f,\ g)$ pour
$case\,(t,\ x_1.\ f\ x_1,\ x_2.\ g\ x_2)$ avec $x_i \notin FV(f\ g)$, $(i = 1, 2)$.

Typage

Nous allons maintenant définir la notion de terme ***bien typé***. Pour cela, il nous faut d'abord associer un type aux variables libres d'un terme. Pour cela, nous utiliserons ce que l'on appelle un ***contexte de typage***, défini comme une suite finie de déclarations de types pour les variables, de la forme $x_1 : \tau_1,\ \ldots,\ x_n : \tau_n$ $(n \geq 0)$, où chaque variable ne peut être déclarée qu'une seule fois. On pourra voir un contexte comme une application des variables dans les types.

Un terme t est de type τ dans le contexte Γ (ce que l'on notera $\boldsymbol{\Gamma \vdash t : \tau}$) si le jugement $\Gamma \vdash t : \tau$ peut être dérivé des règles décrites à la figure 2.1 page suivante, c'est-à-dire si l'on peut construire avec ces règles un arbre dont la racine est ce jugement et les feuilles des règles axiomes (sans prémisse). À l'exception de la règle *ax*, les règles sont classées en deux catégories :

- règles d'***introduction*** (indicée par la lettre I) qui servent au typage des constructeurs,
- et règles d'***élimination*** (indicée par la lettre E), qui servent au typage des destructeurs.

$$\frac{}{\Gamma,\ x:\tau,\ \Gamma' \vdash x:\tau}\ ax \qquad \frac{}{\Gamma \vdash ():1}\ 1_I$$

$$\frac{\Gamma \vdash t_i:\tau_i \quad (i=1,2)}{\Gamma \vdash (t_1,\ t_2):\tau_1\times\tau_2}\ \times_I \qquad \frac{\Gamma \vdash t:\tau_1\times\tau_2}{\Gamma \vdash (proj_i\ t):\tau_i}\ \times_{Ei} \quad (i=1,2)$$

$$\frac{\Gamma,\ x:\tau_1 \vdash t:\tau}{\Gamma \vdash (\lambda x:\tau_1.\ t):\tau_1\to\tau}\ \to_I \qquad \frac{\Gamma \vdash t:\tau_1\to\tau \qquad \Gamma \vdash t_1:\tau_1}{\Gamma \vdash (t\ t_1):\tau}\ \to_E$$

$$\frac{\Gamma \vdash t:0}{\Gamma \vdash \bot_\tau(t):\tau}\ 0_E$$

$$\frac{\Gamma \vdash t:\tau_i}{\Gamma \vdash (in_i^{\tau_1,\tau_2}\ t):\tau_1+\tau_2}\ +_{Ii} \quad (i=1,2) \qquad \frac{\Gamma \vdash t:\tau_1+\tau_2 \qquad \Gamma,\ x_i:\tau_i \vdash t_i:\tau \quad (i=1,2)}{\Gamma \vdash case\,(t,\ x_1:\tau_1.\ t_1,\ x_2:\tau_2.\ t_2):\tau}\ +_E$$

FIG. 2.1 – Règles de typage du λ-calcul simplement typé avec sommes.

Notons en particulier la règle d'élimination du 0 : lorsque le contexte est ***inconsistant***, c'est-à-dire lorsqu'il permet de construire un terme t de type 0, alors pour n'importe quel type τ, nous pouvons construire un λ-terme de type τ de la forme $\bot_\tau(t)$. Remarquons qu'avec cette notation, nous conservons un témoin (le terme t) de l'inconsistance.

Dans la suite, l'expression ***λ-calcul simplement typé avec somme et zéro*** désignera le calcul que je viens de décrire (qui contient également le type produit et les types 1 et 0). On désignera par ***λ-calcul simplement typé sans somme ni zéro*** la version sans type somme ni 0, ni les constructeurs, destructeurs et règles de typage associés.

Correspondance de Curry-Howard

À un type peut-être associée de manière évidente et bijective une formule de la logique propositionnelle. Le type somme correspond au connecteur $\vee$ (« ou »), le type produit au $\wedge$ (« et ») et le type flèche à l'implication $\Rightarrow$. Enfin le type 1 est associé à la valeur booléenne « vrai » ($\top$) et le type 0 au « faux » ($\bot$). Les règles de la figure 2.1 sont exactement celles de la ***déduction naturelle*** utilisées pour faire des preuves en ***logique intuitionniste***[2]. Elles ont simplement été annotées par des λ-termes qui représentent les preuves. Ainsi, par exemple, une preuve d'une formule $E \wedge F$ est un couple $(u,\ v)$ où u est une preuve de E et v une preuve de F.

Dans cette optique, la règle d'élimination du 0 devient la règle dite « intuitionniste », qui dit que si le contexte est inconsistant, on peut prouver n'importe quelle formule.

Enfin la β-réduction correspond au processus d'***élimination des coupures*** dans les preuves. On rappelle qu'en déduction naturelle, une coupure est une règle d'élimination d'un symbole dont la prémisse principale est démontrée par une règle d'introduction de ce symbole.

[2]Ce n'est pas exactement la logique utilisée en mathématiques (appelée ***logique classique***) puisque la logique intuitionniste ne permet pas par exemple les démonstrations par l'absurde ou bien de montrer le tiers-exclus ($A \vee \neg A$). Une présentation alternative à la déduction naturelle est le ***calcul des séquents***. Nous en verrons un exemple lors des rappels sur la logique linéaire page 79.

$\beta\eta$-équivalence

Donnons maintenant les règles de ***β-réduction*** associées à ce calcul :

$$\begin{array}{lrcl}
\beta_{\to} & (\lambda x.\ t)\ u & \longrightarrow & t\left[{}^{u}/_{x}\right] \\
\beta_{\times 1} & proj_1\ (u,\ v) & \longrightarrow & u \\
\beta_{\times 2} & proj_2\ (u,\ v) & \longrightarrow & v \\
\beta_{+1} & case\,((in_1\ \ t),\ x_1.\ t_1,\ x_2.\ t_2) & \longrightarrow & t_1\left[{}^{t}/_{x_1}\right] \\
\beta_{+2} & case\,((in_2\ \ t),\ x_1.\ t_1,\ x_2.\ t_2) & \longrightarrow & t_2\left[{}^{t}/_{x_2}\right]
\end{array}$$

Comme dans le cas du λ-calcul pur, nous souhaitons ajouter des règles d'η-conversion. En particulier, nous parlerons de sommes ***fortes*** lorsque nous voudrons insister sur le fait que nous disposons de la règle d'η-conversion pour la somme. Comme nous le verrons juste après, le choix d'orientation de ces règles η n'est pas évident et conditionne beaucoup les propriétés du système. Énonçons-les donc d'abord sous la forme d'égalités :

$$\begin{array}{lrcl}
\eta_{\to} & \lambda x.\ (t\ \ x) & = & t \\
\eta_{\times} & (proj_1\ t,\ proj_2\ t) & = & t \\
\eta_{1} & t : 1 & = & () \\
\eta_{0} & t' & = & \bot_{\tau}(t) \quad (\text{où } t : 0 \text{ et } t' : \tau) \\
\eta_{+} & case\left(t,\ x_1.\ t'\left[{}^{(in_1\ \ x_1)}/_{x}\right],\ x_2.\ t'\left[{}^{(in_2\ \ x_2)}/_{x}\right]\right) & = & t'\left[{}^{t}/_{x}\right] \quad (\text{où } x_1, x_2 \notin FV(t'))
\end{array}$$

La règle $\eta_{\times}$ est souvent appelée ***SP*** (pour ***Surjective Pairing***). La règle η_1 dit simplement qu'il n'y a qu'un seul habitant du type 1. Enfin la règle η_0 dit que si l'on peut construire un terme t de type 0, alors tous les termes t' de type τ sont η-équivalents à $\bot_{\tau}(t)$.

Chacune des règles peut être orientée dans un sens ou dans l'autre. Pour $\eta_{\to}$, $\eta_{\times}$, et η_{+} nous appellerons ***η-réduction*** la règle orientée de gauche à droite et ***η-expansion*** la règle orientée de droite à gauche. Pour les règles η_1 et η_0, ces appellations sont moins évidentes. Pour la plupart de ces règles, des résultats ont été prouvés avec les deux orientations.

Normalisation

Le λ-calcul simplement typé ainsi défini a perdu une partie de son pouvoir expressif. Pour pouvoir définir toutes les fonctions récursives partielles, il faudrait par exemple ajouter un opérateur de point fixe, ce qui nous donnerait la possibilité de définir des fonctions récursivement.

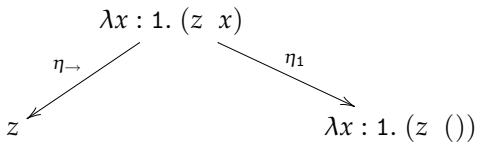
Cependant ce langage a le mérite d'être simple et bien compris, notamment grâce à la correspondance de Curry-Howard. Enfin il a une propriété essentielle :

Proposition 2.4 (normalisation forte) Le λ-calcul simplement typé avec somme et zéro est fortement normalisant pour la β-réduction.

Cependant, quelle que soit l'orientation des règles η, la confluence pose des problèmes, notamment à cause du type 1 ou des types sommes. Prenons par exemple les règles $\eta_{\to}$ et η_1 orientées de la façon suivante :

$$\begin{array}{lrcl}
\eta_{\to} & \lambda x.\ (t\ \ x) & \longrightarrow & t \\
\eta_{1} & t : 1 & \longrightarrow & ()
\end{array}$$

nous obtenons la paire critique suivante :



Nous verrons dans la suite de nombreux exemples de paires critiques faisant intervenir des *case*. Par exemple le lemme 2.6 page 52 montre une paire critique (pour η_+ orientée dans le sens de la réduction) entre les termes

$$case\,(case\,(x,\ x_1.\ in_1\ \ x_1,\ x_2.\ in_2\ \ z),\ y_1.\ y_1,\ y_2.\ f\ \ y_2)$$

et

$$case\,(x,\ z_1.\ z_1,\ z_2.\ f\ \ z)$$

qui sont $\beta\eta$-équivalents.

De nombreuses personnes ont tenté de montrer des propriétés de convergence, et de normalisation forte pour différents systèmes, notamment en utilisant des règles d'η-expansion conditionnelles à la place de l'η-réduction. [26, 38] Aucune d'entre elles ne parvient à résoudre à la fois le problème de la confluence et celui de la normalisation forte, dans le cas du λ-calcul avec somme forte. La $\beta\eta$-réduction ne permet plus d'obtenir une forme normale unique, quelle que soit l'orientation des règles η.

Nous allons donc proposer une autre approche dans les chapitres 5 à 7. Ne pouvant pas obtenir une forme normale unique grâce à des méthodes de réécriture, nous allons être amenés à définir extensionnellement une notion de forme normale, par un ensemble de règles de typage contraint. Le but cherché sera de réduire les possibilités d'η-conversion afin de trouver un représentant canonique de chaque classe d'équivalence de termes modulo $\beta\eta$. La normalisation par évaluation nous permettra d'obtenir effectivement ce représentant.

Je m'intéresserai donc surtout dans la suite à la théorie équationnelle engendrée par ces neuf règles $\beta\eta$. L'équivalence engendrée par cette théorie est appelée la ***βη-équivalence***, que l'on notera $=_{\beta\eta}$. Elle peut être résumée par l'ensemble de règles présenté figure 2.2 page suivante.

Quelques lemmes de $\beta\eta$-équivalence

Nous allons tout d'abord montrer l'équivalence du système de réécriture $\beta\eta$ présenté plus haut avec celui où la règle η_+ est remplacée par la règle η'_+ suivante, qui va nous permettre de simplifier un peu les notations :

$$\eta'_+ \qquad case\,(t,\ x_1.\ (h\ \ (in_1\ \ x_1)),\ x_2.\ (h\ \ (in_2\ \ x_2))) \quad = \quad h\ \ t$$

On appellera $=_{\beta\eta'}$ l'équivalence associée. Nous allons montrer que nous pourrons la plupart du temps utiliser indifféremment η_+ ou η'_+.

Proposition 2.5
- La règle de réécriture η'_+ se déduit de η_+ (même orientation)
- La règle de réécriture η_+ se déduit des règles η'_+ et $\beta_\to$ (même orientation pour η_+ et η'_+)

On peut donc en déduire que $t =_{\beta\eta} t'$ équivaut à $t =_{\beta\eta'} t'$ (où t et t' sont des termes quelconques)

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash t = t : \tau} \qquad \frac{\Gamma \vdash t = t' : \tau}{\Gamma \vdash t' = t : \tau} \qquad \frac{\Gamma \vdash t_1 = t_2 : \tau \qquad \Gamma \vdash t_2 = t_3 : \tau}{\Gamma \vdash t_1 = t_3 : \tau}$$

$$\frac{\Gamma \vdash t : 1}{\Gamma \vdash t = () : 1}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (proj_i \ (t_1, t_2)) = t_i : \tau_i} \ (i = 1,2) \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash t = ((proj_1 \ t), (proj_2 \ t)) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau \qquad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash ((\lambda x : \tau_1 . \, t) \ t_1) = t \left[t_1 / x \right] : \tau} \qquad \frac{\Gamma \vdash t : \tau_1 \to \tau}{\Gamma \vdash t = (\lambda x : \tau_1 . \, (t \ x)) : \tau_1 \to \tau} \ (x \notin FV(t))$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau \qquad \Gamma \vdash t_1 = t_1' : \tau_1}{\Gamma \vdash (t \ t_1) = (t \ t_1') : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash t = t' : \tau}{\Gamma \vdash (\lambda x : \tau_1 . \, t) = (\lambda x : \tau_1 . \, t') : \tau_1 \to \tau}$$

$$\frac{\Gamma \vdash t : 0 \qquad \Gamma \vdash t' : \tau}{\Gamma \vdash \bot_\tau(t) = t' : \tau}$$

$$\frac{\Gamma \vdash t : \tau_j \qquad \Gamma, x_i : \tau_i \vdash t_i : \tau \quad (i = 1,2)}{\Gamma \vdash case \left((in_j^{\tau_1, \tau_2} \ t), x_1 : \tau_1 . \, t_1, x_2 : \tau_2 . \, t_2 \right) = t_j \left[t / x_j \right] : \tau} \ (j = 1,2)$$

$$\frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 + \tau_2 \vdash t' : \tau}{\Gamma \vdash case \left(t, x_1 : \tau_1 . \, t' \left[(in_1^{\tau_1, \tau_2} \ x_1) / x \right], x_2 : \tau_2 . \, t' \left[(in_2^{\tau_1, \tau_2} \ x_2) / x \right] \right) = t' \left[t / x \right] : \tau}$$

FIG. 2.2 – Théorie équationnelle du λ-calcul simplement typé avec type somme et type vide.

Démonstration :

▶ $\eta_+ \Longrightarrow \eta'_+$: Il suffit de choisir $t' = h\ \ x$

▶ $\beta_{\to}\eta'_+ \Longrightarrow \eta_+$: Il suffit de prendre $h = \lambda x.\ t'$ ■

En faisant l'abus de notation présenté page 47, nous pouvons écrire également la règle η''_+ suivante :

$$\eta''_+ \qquad case\,(t,\ (h \circ in_1),\ (h \circ in_2)) \quad = \quad h\ \ t$$

Ici (comme après), $f \circ g$ représente le λ-terme $\lambda x.\ f\ \ (g\ \ x)$ (avec $x \notin FV(f\ \ g)$). Encore une fois, la théorie engendrée est trivialement la même. Dans la suite, nous ne distinguerons plus les règles η_+, η'_+ et η''_+ lorsque nous parlerons de règles équationnelles (non orientées).

Nous allons maintenant montrer quelques conséquences des règles de $\beta\eta$-équivalence. Il s'agit de montrer quelques formules qui nous seront utiles dans la suite. Remarquons qu'elles sont assez intuitives, mais leurs preuves à partir des règles $\beta\eta$ ne sont pas si évidentes. Ces exemples donnent une idée des difficultés pour écrire une fonction d'η-réduction ou bien un test d'η-équivalence.

Lemme 2.6

$$case\,(\mathbf{case\,(t,\ in_1 \circ h_1,\ in_2 \circ h_2)},\ f,\ g) =_{\beta\eta} case\,(t,\ f \circ h_1,\ g \circ h_2)$$

Démonstration :

Appliquons la règle η'_+ à $h = \lambda y.\ case\,(\mathbf{case\,(y,\ in_1 \circ h_1,\ in_2 \circ h_2)},\ f,\ g)$.
On obtient
$case\,(t,\ h \circ in_1,\ h \circ in_2) =_{\beta\eta} case\,(\mathbf{case\,(t,\ in_1 \circ h_1,\ in_2 \circ h_2)},\ f,\ g)$.

$$\begin{array}{lrll} \text{avec} & h \circ in_1 & = & \lambda z.\ case\,(\mathbf{case\,(in_1\ \ z,\ in_1 \circ h_1,\ in_2 \circ h_2)},\ f,\ g) \\ & & \to_{\beta_{+1}} & \lambda z.\ case\,(\mathbf{in_1\ \ (h_1\ \ z)},\ f,\ g) \\ & & \to_{\beta_{+1}} & \lambda z.\ f\ \ (h_1\ \ z) =_{\beta\eta} f \circ h_1 \end{array}$$

(de même $h \circ in_2 = g \circ h_2$) ■

Lemme 2.7

$$(f\ \ (case\,(t,\ x_1.\ t_1,\ x_2.\ t_2))) =_{\beta\eta} case\,(t,\ x_1.\ (f\ \ t_1),\ x_2.\ (f\ \ t_2))$$

$$((case\,(t,\ x_1.\ t_1,\ x_2.\ t_2))\ \ \mathbf{t'}) =_{\beta\eta} case\,(t,\ x_1.\ (t_1\ \ \mathbf{t'}),\ x_2.\ (t_2\ \ \mathbf{t'}))$$

Démonstration :

- Pour la première égalité, poser $h = \lambda x.\ (f\ \ (case\,(x,\ x_1.\ t_1,\ x_2.\ t_2)))$ et appliquer la règle η'_+, puis β-réduire.
- Pour la deuxième, procéder de façon analogue.

■

Lemme 2.8 Si $x_i \notin FV(t')\ \ (i = 1, 2)$:

$$case\,(t,\ x_1.\ \mathbf{t'},\ x_2.\ \mathbf{t'}) =_{\beta\eta} \mathbf{t'}$$

Démonstration :
C'est une conséquence directe de la règle η_+. ■

Lemme 2.9

$$case\,(t,\ x.\ case\,(t,\ x_1.\ t_1,\ x_2.\ t_2),\ y.\ u) =_{\beta\eta} case\left(t,\ x.\ t_1\left[{}^{x}/_{x_1}\right],\ y.\ u\right)$$

$$case\,(t,\ x.\ u,\ y.\ case\,(t,\ x_1.\ t_1,\ x_2.\ t_2)) =_{\beta\eta} case\left(t,\ x.\ u,\ y.\ t_2\left[{}^{y}/_{x_2}\right]\right)$$

Démonstration :
Pour la première égalité, il suffit d'appliquer η'_+ à $h = \lambda y.\ case\,(y,\ x.\ case\,(y,\ x_1.\ t_1,\ x_2.\ t_2),\ y.\ u)$ et de β-réduire. La preuve de la deuxième est analogue. ■

Lemme 2.10 Si $x \notin FV(t')$ et $x_i \notin FV(t)$ $(i = 1, 2)$:

$$case\,(t,\ x.\ case\,(t',\ x_1.\ u_1,\ x_2.\ u_2),\ y.\ u) =_{\beta\eta} case\,(t',\ x_1.\ case\,(t,\ x.\ u_1,\ y.\ u),\ x_2.\ case\,(t,\ x.\ u_2,\ y.\ u))$$

$$case\,(t,\ y.\ u,\ x.\ case\,(t',\ x_1.\ u_1,\ x_2.\ u_2)) =_{\beta\eta} case\,(t',\ x_1.\ case\,(t,\ y.\ u,\ x.\ u_1),\ x_2.\ case\,(t,\ y.\ u,\ x.\ u_2))$$

Démonstration :
Pour obtenir par exemple la première égalité, il suffit par exemple d'appliquer la règle η'_+ à $h = \lambda z.\ case\,(t',\ x_1.\ case\,(z,\ x.\ u_1,\ y.\ u),\ x_2.\ case\,(z,\ x.\ u_2,\ y.\ u))$, puis appliquer le lemme 2.8 ■

2.2 Sémantique du λ-calcul simplement typé

Tout programmeur connaît la nécessité pour un langage d'avoir une sémantique précise, que ce soit par exemple pour assurer un comportement prévisible aux programmes et réduire par là les possibilité d'erreurs, ou bien encore pour qu'un même programme ait le même comportement sur des systèmes différents ou s'il est généré par différents compilateurs.

Il existe essentiellement trois façons de définir la sémantique d'un langage :

- la ***sémantique opérationnelle*** décrit pas à pas l'exécution du programme appliqué à ses données,
- la ***sémantique dénotationnelle*** associe à chaque programme un objet mathématique,
- la ***sémantique axiomatique*** décrit les propriétés vérifiées par le programme à chaque étape de son exécution.

Pour l'instant nous allons nous intéresser à la sémantique dénotationnelle. Le principe consiste à donner une ***interprétation*** mathématique des termes et des types. L'image de cette fonction d'interprétation sera appelée ***modèle*** du langage.

Nous allons étudier ici plus précisément les modèles catégoriques du λ-calcul. Le principe consiste à interpréter un type τ par un objet $[\![\tau]\!]$, un contexte Γ comme le produit de ses types, et un jugement de typage $\Gamma \vdash t : \tau$ par une flèche entre $[\![\Gamma]\!]$ et $[\![\tau]\!]$, notée $[\![\Gamma \vdash t : \tau]\!]$. Une sémantique doit vérifier une condition essentielle, appelée ***correction*** [▷ *soundness*] :

correction : pour tout jugement de typage $\Gamma \vdash t : \tau$, si $t =_{\beta\eta} t'$ alors $[\![\Gamma \vdash t : \tau]\!] = [\![\Gamma \vdash t' : \tau]\!]$

Nous verrons que les catégories répondant à cette condition sont les catégories bi-cartésiennes fermées.

En fonction des propriétés à prouver sur le langage, il peut être intéressant de choisir le modèle le plus adapté. Pour ce faire, on choisit généralement une notion d'***equivalence observationnelle !équivalence observationnelle***, et l'on dira que le modèle est ***correct*** vis-à-vis de cette notion si l'égalité des interprétations de deux termes implique l'équivalence observationnelle. Réciproquement, si l'équivalence observationnelle implique l'égalité des interprétations, on dira que la sémantique est ***complète***. Une sémantique correcte et complète est dite ***complètement abstraite*** vis-à-vis de cette notion d'équivalence observationnelle.

2.2.1 Modèle des λ-termes

Pour obtenir une sémantique complète et correcte (pour la $\beta\eta$-équivalence), on peut tout simplement quotienter l'ensemble des λ-termes par la relation d'équivalence associée à la théorie $\beta\eta$, de sorte que tous les termes $\beta\eta$-équivalents soient interprétés par la même flèche. On parle parfois de ***modèle syntaxique***. Nous allons définir plus précisément cette catégorie et montrer que c'est une catégorie bi-cartésienne fermée.

Intéressons-nous d'abord au cas du λ-calcul sans somme. Nous allons l'interpréter dans une catégorie définie de la manière suivante :

Définition 2.11 ($\Lambda_{\times 1\beta\eta}$) Soit $\Lambda_{\times 1\beta\eta}$ le graphe dont les objets sont les types du λ-calcul simplement typé avec paires et unité, et dont les flèches entre deux objets A et B sont les classes d'équivalence modulo $\beta\eta$-équivalence des λ-termes de type $A \to B$ dans le contexte vide.

Notations : Pour tout entier n supérieur ou égal à 3, et pour tout n-uplet $(A_1, ..., A_n)$ d'objets d'une catégorie, on note $A_1 \times A_2 \times ... A_n$ l'objet $(...(A_1 \times A_2) \times ...A_n)$ (parenthésage à gauche). Les n-uplets seront donc également parenthésés à gauche.

On note $\Gamma^\times$ la conjonction des formules d'une liste Γ de formules. Si Γ est vide, $\Gamma^\times = 1$.

On note $[M]$ la classe d'équivalence du λ-terme M modulo $\beta\eta$-équivalence.

Nous allons interpréter un type par un objet de $\Lambda_{\times 1\beta\eta}$, un contexte Γ par l'interprétation du type $\Gamma^\times$ et un séquent $\Gamma \vdash t : \tau$ par la flèche $[\lambda x : \Gamma^\times.\ t]$. Notamment un séquent $\vdash t : \tau$ sera interprété par $[\lambda x : 1.\ t]$.

Proposition 2.12 $\Lambda_{\times 1\beta\eta}$ est une catégorie

Démonstration :

- Pour chaque objet A, il existe une flèche id_A de A vers lui-même qui est $[\lambda x.\ x]$
- Soient $f : C \to D$, $g : B \to C$ et $h : A \to B$ trois flèches de $\Lambda_{\times 1\beta\eta}$.
  Soient F et F' deux éléments de la classe d'équivalence f, G et G' deux éléments de g, et H un élément de h.
  La composition dans $\Lambda_{\times 1\beta\eta}$ est définie par $f \circ g = [F \circ G]$.
  On vérifie que $[F \circ G] = [F' \circ G']$.
- $id_D \circ f = [(\lambda x.\ x) \circ F] = [F] = f$
- de même : $f \circ id_C = f$
- composition :

$$\begin{aligned}(f \circ g) \circ h &= [F \circ G] \circ h \\ &= [(F \circ G) \circ H] \text{ car } F \circ G \text{ est un représentant de } f \circ g\end{aligned}$$

$$
\begin{aligned}
&= [F \circ (G \circ H)] \text{ (associativité en } \lambda\text{-calcul)} \\
&= f \circ (g \circ h)
\end{aligned}
$$

■

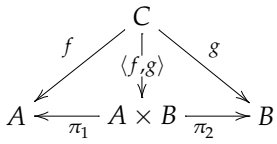
Nous allons montrer que ce modèle est une catégorie cartésienne fermée.

Théorème 2.13 $\Lambda_{\times 1 \beta\eta}$ est une catégorie cartésienne fermée.

Démonstration :

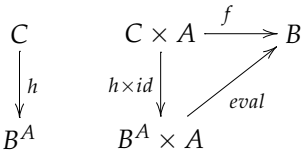
▶ Terminal : Le type 1 est terminal. Pour tout type A il existe une unique fonction $!A$ (à $\beta\eta$-équivalence près) de type $A \to 1$ qui est $!A = [\lambda x.\ ()]$ L'unicité vient de la règle η_1.

▶ Produit : Le produit est $\times$ (type produit), π_1 et π_2 sont les classes des projections $proj_1$ et $proj_2$.



On prend $< f, g >= [\lambda x.\ (F\ \ x,\ G\ \ x)]$ (que l'on notera h, F et G étant des représentants de f et g) et l'on vérifie que $\pi_1 \circ h = f$ et $\pi_2 \circ h = g$
Reste à montrer l'unicité d'un tel h. Pour cela supposons que la flèche $h' : C \to A \times B$ vérifie $\pi_1 \circ h' = f$ et $\pi_2 \circ h' = g$. On a alors $h = [\lambda x.\ ((proj_1 \circ H')\ \ x,\ (proj_2 \circ H')\ \ x))]$ et donc $h = h'$ grâce à la règle $\eta_\times$(où H' est un représentant de la classe h'.)

▶ Exponentielle :



Pour tous les objets A et B, l'objet B^A est le type $A \to B$ et la fonction *eval* est la classe de $\lambda p.\ ((proj_1\ \ p)\ \ (proj_2\ \ p))$. Pour h, on prend la classe de $\lambda y.\ \lambda z.\ (F\ \ (y,\ z))$ (où F est un représentant de la classe f). On vérifie que le diagramme commute :

$$
\begin{aligned}
&eval \circ (h \times id) \\
&= [(\lambda p.\ (proj_1\ p)\ \ (proj_2\ p)) \circ (\lambda x.\ ((\lambda y.\ \lambda z.\ F\ \ (y,\ z))\ \ (proj_1\ \ x),\ id\ \ (proj_2\ \ x)))] \\
&= [(\lambda p.\ (proj_1\ \ p)\ \ (proj_2\ \ p)) \circ (\lambda x.\ ((\lambda z.\ F\ \ ((proj_1\ \ x),\ z)),\ (proj_2\ \ x)))] \\
&= [\lambda x.\ (\lambda p.\ (proj_1\ \ p)\ \ (proj_2\ \ p))\ \ ((\lambda z.\ F\ \ ((proj_1\ \ x),\ z)),\ (proj_2\ \ x))] \\
&= [\lambda x.\ (\lambda z.\ F\ \ ((proj_1\ \ x),\ z))\ \ (proj_2\ \ x)] \\
&= [\lambda x.\ F\ \ ((proj_1\ \ x),\ (proj_2\ \ x))] \\
&= [\lambda x.\ F\ \ x] \text{ (règle } \eta_\times) \\
&= [F] = f
\end{aligned}
$$

Pour vérifier l'unicité de h, supposons que $h' : C \to B^A$ (dont un représentant est H') vérifie

$eval \circ (h' \times id) = f$. On a donc

$$\begin{aligned} f &= [(\lambda p.\ (proj_1\ p)\ (proj_2\ p)) \circ (\lambda x.\ (H'\ (proj_1\ x),\ (proj_2\ x)))] \\ &= [\lambda x.\ (H'\ (proj_1\ x))\ (proj_2\ x)] \end{aligned}$$

Donc

$$\begin{aligned} h &= [\lambda y.\ \lambda z.\ F\ (y,\ z)] \\ &= [\lambda y.\ \lambda z.\ (\lambda x.\ (H'\ (proj_1\ x))\ (proj_2\ x))\ (y,\ z)] \\ &= [\lambda y.\ \lambda z.\ (H'\ y\ z)] = h' \end{aligned}$$

■

Étendons maintenant ce résultat au λ-calcul avec somme :

Définition 2.14 On appelle $\Lambda_{\times+10\beta\eta}$ le graphe dont les objets sont les types du λ-calcul simplement typé avec produit, co-produit et unités, et dont les flèches entre deux objets A et B sont les classes d'équivalence des λ-termes de type $A \to B$ modulo $\beta\eta$-équivalence.

On gardera la même notation que précédemment pour les classes d'équivalence.

Proposition 2.15 $\Lambda_{\times+10\beta\eta}$ est une catégorie

Démonstration :

Même preuve que pour la proposition 2.12 page 54. ■

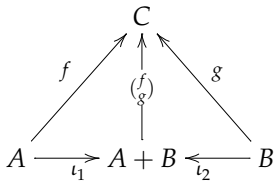
Théorème 2.16 $\Lambda_{\times+10\beta\eta}$ est une catégorie bi-cartésienne fermée.

Démonstration :

Il reste à vérifier les propriétés de l'objet initial et de la somme.

▶ Objet initial : Le type 0 est initial. $[\lambda x : 0.\ \bot_A(x)]$ est l'unique flèche entre 0 et A.

▶ Somme : La somme est $+$ (type somme), i_1 et i_2 sont les classes des injections in_1 et in_2.



On prend $\binom{f}{g} = [\lambda x.\ case\,(x,\ F,\ G)]$ (que l'on notera h, F et G étant des représentants de f et g.) et l'on a bien $h \circ [in_1] = f$ et $h \circ [in_2] = g$ d'après les règles β_{+1} et β_{+2}. Pour montrer l'unicité, supposons qu'une flèche $h' : A + B \to C$ vérifie $h' \circ [in_1] = f$ et $h' \circ [in_2] = g$. On a alors $h = [\lambda x.\ case\,(x,\ H' \circ in_1,\ H' \circ in_2)]$ et donc $h = h'$ d'après la règle η'_+(où H' est un représentant de la classe h'.) ■

2.2.2 Les biCCC modèles du λ-calcul avec somme

Nous avons trouvé une (bi-)CCC particulière modèle du λ-calcul. Nous allons maintenant montrer que n'importe quelle catégorie (bi-)cartésienne fermée peut être un modèle du λ-calcul.

Étant donnée une interprétation $\mathcal{I}(.)$ des types de base dans une biCCC $\mathfrak{C}$, on notera $\mathcal{I}[\![\tau]\!]$ l'interprétation d'un type τ induite par la structure bi-cartésienne fermée. À savoir :

$$\begin{gathered}\mathcal{I}[\![\theta]\!] = \mathcal{I}(\theta) \text{ (θ type de base)}\\ \mathcal{I}[\![1]\!] = \mathfrak{1}\,, \quad \mathcal{I}[\![\tau_1 \times \tau_2]\!] = \mathcal{I}[\![\tau_1]\!] \times \mathcal{I}[\![\tau_2]\!]\\ \mathcal{I}[\![\tau_1 \to \tau_2]\!] = \mathcal{I}[\![\tau_2]\!]^{\mathcal{I}[\![\tau_1]\!]}\\ \mathcal{I}[\![0]\!] = \mathfrak{o}\,, \quad \mathcal{I}[\![\tau_1 + \tau_2]\!] = \mathcal{I}[\![\tau_1]\!] + \mathcal{I}[\![\tau_2]\!]\end{gathered}$$

Étendons maintenant cette interprétation aux contextes, comme auparavant :

$$\mathcal{I}[\![\Gamma]\!] = \mathcal{I}[\![\Gamma^\times]\!]$$

et notons $\mathcal{I}[\![\Gamma \vdash t : \tau]\!]$ le morphisme $\mathcal{I}[\![\Gamma]\!] \to \mathcal{I}[\![\tau]\!]$ de $\mathfrak{C}$ interprétant le jugement $\Gamma \vdash t : \tau$. (Pour plus de détails, voir [60, 25, 79])

On peut donner une sémantique du λ-calcul simplement typé dans n'importe quelle biCCC de la manière suivante (voir le livre de Lambek et Scott [60]) :

$$\begin{aligned}[\![x_1 : \tau_1, \ldots x_n : \tau_n \vdash x_i : \tau_i]\!] &= \pi_i\\ [\![\Gamma \vdash \lambda x : \tau_1 . t : \tau_1 \to \tau_2]\!] &= \Lambda([\![\Gamma, x : \tau_1 \vdash t : \tau_2]\!])\\ [\![\Gamma \vdash t_1\ t_2 : \tau_2]\!] &= \mathit{eval} \circ \langle [\![\Gamma \vdash t_1 : \tau_1 \to \tau_2]\!], [\![\Gamma \vdash t_2 : \tau_1]\!] \rangle\end{aligned}$$

On peut étendre ceci en une sémantique du λ-calcul simplement typé avec produit et unité en ajoutant les clauses suivantes :

$$\begin{aligned}[\![\Gamma \vdash (t_1,\ t_2) : \tau_1 \times \tau_2]\!] &= \langle [\![\Gamma \vdash t_1 : \tau_1]\!], [\![\Gamma \vdash t_2 : \tau_2]\!] \rangle\\ [\![\Gamma \vdash \mathit{proj}_i\ t : \tau_i]\!] &= \pi_i \circ [\![\Gamma \vdash t : \tau_1 \times \tau_2]\!]\\ [\![\Gamma \vdash () : 1]\!] &= !\,\mathcal{I}[\![\Gamma]\!]\end{aligned}$$

Pour le λ-calcul avec type somme et produit, on peut étendre cette interprétation dans une biCCC en ajoutant :

$$\begin{aligned}[\![\Gamma \vdash \mathit{in}_i\ t : \tau_1 + \tau_2]\!] &= \iota_i \circ [\![\Gamma \vdash t : \tau_i]\!]\\ [\![\Gamma \vdash \mathit{case}\,(t,\ x_1.\, t_1,\ x_2.\, t_2) : \tau]\!] &= \begin{pmatrix}[\![\Gamma,\ x_1 : \tau_1 \vdash t_1 : \tau]\!]\\ [\![\Gamma,\ x_2 : \tau_2 \vdash t_2 : \tau]\!]\end{pmatrix}\\ &\quad \circ \gamma \circ \langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash t : \tau_1 + \tau_2]\!] \rangle\\ [\![\Gamma \vdash \bot_\tau(t) : \tau]\!] &= \mathfrak{i}\,\mathcal{I}[\![\tau]\!] \circ [\![\Gamma \vdash t : 0]\!]\end{aligned}$$

où $\gamma : (A \times (B + C)) \to ((A \times B) + (A \times C))$ est l'isomorphisme de distributivité.

Dans la section suivante, nous prouverons notamment que l'on définit bien ainsi une sémantique. Nous pouvons, en particulier, choisir la catégorie cartésienne fermée libre engendrée sur

les types de base. En fait, on peut même montrer que cette catégorie est isomorphe à $\Lambda_{\times+10\beta\eta}$, comme nous le verrons dans la section 2.3.

2.3 Isomorphisme entre $\Lambda_{\times+10\beta\eta}$ et la biCCC libre

Dans cette section, je vais montrer que le modèle des λ-termes est isomorphe à la catégorie bi-cartésienne fermée libre engendrée sur les types.

2.3.1 Sans co-produit

Définitions

Définition 2.17 On appelle $\mathfrak{L}$ la catégorie cartésienne fermée libre engendrée à partir de l'ensemble des types de base.

Remarque 2.18 On peut établir de manière triviale une bijection entre les objets de $\Lambda_{\times 1\beta\eta}$ et ceux de $\mathfrak{L}$. On considérera donc dans la suite que les objets des deux graphes sont les mêmes.

Les isomorphismes sont les mêmes

En décrivant le modèle des λ-termes page 54, nous avons en fait associé à chaque flèche de $\mathfrak{L}$ une flèche dans $\Lambda_{\times 1\beta\eta}$. Pour montrer l'isomorphisme entre $\mathfrak{L}$ et $\Lambda_{\times 1\beta\eta}$, il nous faut montrer que cette association est un foncteur bijectif. Nous allons définir plus formellement cette fonction (ou plutôt sa réciproque) puis nous allons montrer que c'est un foncteur, puis qu'elle est bijective.

Afin d'être plus proche des catégories et donc de simplifier un peu les calculs, nous allons donner la sémantique des jugements de typage en passant par l'intermédiaire des arbres de preuves de la déduction naturelle, et nous allons utiliser un système de règles de typage légèrement différent de celui de la page 48 : nous allons introduire des règles structurelles, affaiblissement et permutation, qui correspondent respectivement dans les catégories aux projections et aux isomorphismes de commutativité, comme nous le verrons plus loin. Plus précisément, nous gardons les règles de la figure 2.1 page 48 en remplaçant seulement la règle axiome par les trois règles suivantes :

$$\frac{}{x:\tau \vdash x:\tau}\ ax'$$

$$\frac{\Gamma \vdash t:\tau}{\Gamma,\ x:\sigma \vdash t:\tau}\ aff \qquad \frac{\Gamma,\ y:\tau,\ x:\sigma,\ \Gamma' \vdash t:C}{\Gamma,\ x:\sigma,\ y:\tau,\ \Gamma' \vdash t:C}\ perm$$

Il est facile de vérifier que le nouveau système d'inférence obtenu est équivalent au premier, dans le sens où il permet de typer exactement les mêmes termes. On peut remarquer que contrairement au premier, ce système permet d'écrire plusieurs dérivations de typage pour le même λ-terme, qui diffèrent uniquement par la position des règles structurelles. C'est cette propriété qui nous sera utile. Il sera notamment possible de placer toutes les règles structurelles immédiatement sous les axiomes.

On va donc commencer par définir une fonction (que l'on notera Φ) qui associe à un arbre de preuve du calcul des séquents de conclusion $\Gamma \vdash A$ une flèche de $\mathfrak{L}$ entre $\Gamma^\times$ et A. La définition de

la fonction entre $\Lambda_{\times 1\beta\eta}$ et $\mathfrak{L}$ découlera directement de cette définition. On la notera également Φ.

Si Γ est vide, la définition de Φ pose un problème. Pour le résoudre, on pourrait associer une flèche entre 1 et A aux séquents de la forme $\vdash A$ ce qui conduirait à séparer à chaque fois le cas où Γ est vide. On pourrait aussi associer à $A_1, \dots A_n \vdash A$ une flèche entre $1 \times A_1 \times \dots A_n$ et A. Mais ces deux solutions compliquent beaucoup les calculs. On va plutôt remarquer que si Γ n'est pas vide à la conclusion, alors il existe une preuve du même séquent dans laquelle tous les séquents ont au moins une hypothèse. Il suffit de « faire remonter » les règles d'affaiblissement. Nous verrons plus loin que les preuves qui nous intéressent ont une hypothèse à la racine. On ne s'intéressera donc dans les pages qui suivent qu'aux preuves dans lesquelles tous les séquents ont au moins une hypothèse.

Définition 2.19 Soit Π un arbre de preuve du calcul des séquents de conclusion $\Gamma \vdash A$, dans laquelle tous les séquents ont au moins une hypothèse. $\mathbf{\Phi(\Pi)}$ est une flèche de $\mathfrak{L}$ entre $\Gamma^\times$ et A définie par induction sur la structure de Π.

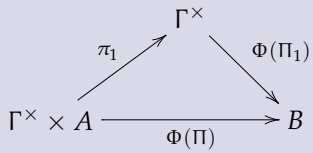
Pour les preuves de profondeur 1, on a deux possibilités :

- $\Phi(\overline{A \vdash A}) = id_A$
- $\Phi(\overline{\Gamma \vdash 1}) = !\Gamma^\times$

Pour les autres cas, on appellera Π_1 et éventuellement Π_2 et Π_3 les sous-arbres obtenus en enlevant la dernière règle de Π, dans l'ordre donné par la règle. On définit $\Phi(\Pi)$ en fonction de la dernière règle de Π :

- $\dfrac{\Gamma \vdash B}{\Gamma, A \vdash B}\ aff$

  $\Phi(\Pi_1)$ est une flèche entre $\Gamma^\times$ et B. On prend $\Phi(\Pi) = \Phi(\Pi_1) \circ \pi_1$

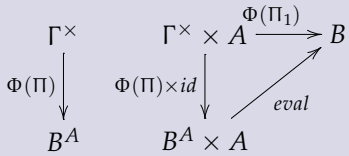


- $\dfrac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\ perm$

  On prend $\Phi(\Pi) = \Phi(\Pi_1) \circ i$, où i est l'isomorphisme canonique entre $\Gamma^\times \times A \times B \times \Delta^\times$ et $\Gamma^\times \times B \times A \times \Delta^\times$.

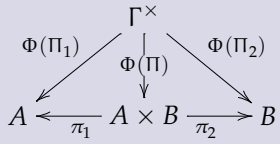
- $\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_I$

  $\Phi(\Pi_1)$ est une flèche entre $\Gamma^\times \times A$ et B. On prend $\Phi(\Pi) = \Lambda(\Phi(\Pi_1))$ l'unique flèche entre $\Gamma^\times$ et B^A qui fait commuter le diagramme de l'exponentielle.



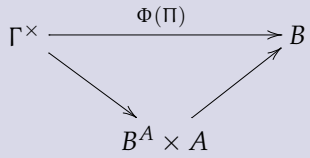
- $\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}\ \times_I$

  $\Phi(\Pi_1)$ est une flèche entre $\Gamma^\times$ et A et $\Phi(\Pi_2)$ une flèche entre $\Gamma^\times$ et B. On prend pour $\Phi(\Pi)$ l'unique flèche entre $\Gamma^\times$ et $A \times B$ qui fait commuter le diagramme du produit. $\Phi(\Pi) = \langle \Phi(\Pi_1), \Phi(\Pi_2) \rangle$



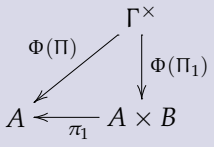
– $\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B} \to_E$

$\Phi(\Pi_1)$ est une flèche entre $\Gamma^\times$ et B^A et $\Phi(\Pi_2)$ une flèche entre $\Gamma^\times$ et A. On prend pour $\Phi(\Pi)$ la flèche $eval_{A,B} \circ \langle \Phi(\Pi_1), \Phi(\Pi_2) \rangle$ (entre $\Gamma^\times$ et B).



– $\dfrac{\Gamma \vdash A \times B}{\Gamma \vdash A} \times_{E1}$

$\Phi(\Pi_1)$ est une flèche entre $\Gamma^\times$ et $A \times B$. On prend pour $\Phi(\Pi)$ la flèche $\pi_1 \circ \Phi(\Pi_1)$ (entre $\Gamma^\times$ et A).



– $\pi_1 \circ \Phi(\Pi_1)$ pour $\times_{E2}$

Le lemme suivant nous permet de voir que l'on définit bien ainsi une interprétation pour les jugements de typage. Il est facile de voir que c'est exactement celle présentée à la section 2.2.2.

Lemme 2.20 Si Π et Π' sont deux arbres de typage du même λ-terme, alors $\Phi(\Pi) = \Phi(\Pi')$.

Démonstration :

Π et Π' ne diffèrent que par la position des règles structurelles (affaiblissement et permutation). On montre que l'on peut remonter les règles structurelles jusqu'en haut, et on utilise ensuite la canonicité de la flèche entre $\Gamma^\times \times A \times \Delta^\times$ et A. La preuve est exactement identique dans le cas des affaiblissements et des permutations. On notera Γ' le contexte obtenu à partir de Γ après affaiblissement ou permutation. On notera α la flèche π_1 (dans le cas d'un affaiblissement) ou i (dans le cas d'une permutation). Montrons que les règles structurelles « traversent » toutes les autres vers le haut sans changer l'image de la preuve par Φ :

▶ Règle $\to_I$:

$$\begin{array}{c} \dfrac{\dfrac{\Gamma,\ A \vdash B}{\Gamma \vdash A \to B}\to_I}{\Gamma' \vdash A \to B}\ \textit{struct} \\ \Phi(\Pi) = \Lambda(\Phi(\Pi_1)) \circ \alpha \end{array} \quad \longrightarrow \quad \begin{array}{c} \dfrac{\dfrac{\Gamma,\ A \vdash B}{\Gamma',\ A \vdash B}\ \textit{struct}}{\Gamma' \vdash A \to B}\to_I \\ \Phi(\Pi') = \Lambda(\Phi(\Pi_1 \circ (\alpha \times id))) \end{array}$$

Or $\Lambda^{-1}(\Lambda(\Phi(\Pi_1)) \circ \alpha) = eval \circ (\Lambda(\Phi(\Pi_1)) \times id) \circ (\alpha \times id) = \Lambda(\Phi(\Pi_1)) \circ (\alpha \times id)$ Donc $\Phi(\Pi) = \Phi(\Pi')$

▶ Règle $\times_I$: $\langle \Phi(\Pi_1), \Phi(\Pi_2)\rangle \circ \alpha = \langle \Phi(\Pi_1) \circ \alpha, \Phi(\Pi_2) \circ \alpha\rangle$

▶ Règle $\to_E$: $eval \circ \langle \Phi(\Pi_1), \Phi(\Pi_2)\rangle \circ \alpha = eval \circ \langle \Phi(\Pi_1) \circ \alpha, \Phi(\Pi_2) \circ \alpha\rangle$

▶ Règle $\times_{E1}$: $(\pi_1 \circ \Phi(\Pi_1)) \circ \alpha = \pi_1 \circ (\Phi(\Pi_1) \circ \alpha)$

▶ Règle $\times_{E2}$: idem ■

Lemme 2.21 Si Π et Π' sont deux arbres de preuve $\beta\eta$-équivalents, alors $\Phi(\Pi) = \Phi(\Pi')$.

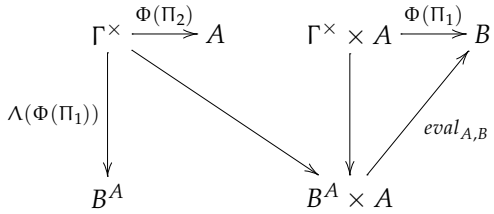
Démonstration :

Il s'agit de montrer que pour chaque règle, on obtient la même chose avant et après réduction.

▶ Règle $\beta_\to$: Cherchons la flèche associée à l'arbre Π suivant :

$$\dfrac{\dfrac{\Pi_1}{\Gamma \vdash (\lambda x.\, b) : A \to B}\ {\to_I} \qquad \Pi_2}{\Gamma \vdash (\lambda x.\, b)\ \ a : B}\ {\to_E}$$

La flèche associée à Π par la définition précédente est $\Phi(\Pi) = eval_{A,B} \circ \langle \Lambda(\Phi(\Pi_1)), \Phi(\Pi_2)\rangle$



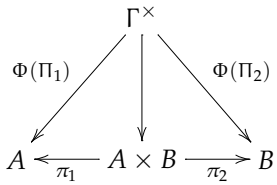
Il faut montrer que $\Phi(\Pi) = \Phi(\Pi_1) \circ \langle id_{\Gamma^\times}, \Phi(\Pi_2)\rangle$

Ce qui se déduit de l'égalité $\Phi(\Pi_1) = eval_{A,B} \circ (\Lambda(\Phi(\Pi_1)) \times id_A)$

▶ Règles $\beta_{\times 1}$ et $\beta_{\times 2}$: Cherchons la flèche associée à l'arbre Π suivant :

$$\dfrac{\dfrac{\Pi_1 \quad \Pi_2}{\Gamma \vdash (a, b) : A \times B}\ {\times_I}}{\Gamma \vdash proj_1(a, b) : A}\ {\times_E}$$

La flèche associée à Π par la définition précédente est $\pi_1 \circ \langle \Phi(\Pi_1), \Phi(\Pi_2)\rangle$, qui est égale à $\Phi(\Pi_1)$ (propriété du produit).



Le deuxième cas est quasiment identique.

▶ Règle $\eta_\rightarrow$: Cherchons la flèche associée à l'arbre Π suivant :

$$\dfrac{\dfrac{\dfrac{\overline{x : A \vdash x : A}}{x : A,\ \Gamma \vdash x : A}\ aff}{\Gamma,\ x : A \vdash x : A}\ perm \qquad \dfrac{\Pi_1}{\Gamma,\ x : A \vdash F : A \rightarrow B}\ \rightarrow_E}{\dfrac{\Gamma,\ x : A \vdash (F\ x) : B}{\Gamma \vdash \lambda x.\ (F\ x) : A \rightarrow B}\ \rightarrow_I}$$

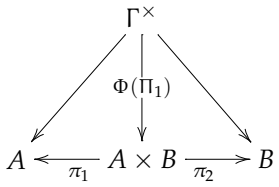
La flèche associée à Π par la définition précédente est :

$$\begin{aligned} \Lambda(eval_{A,B} \circ \langle \Phi(\Pi_1) \circ \pi_1, \pi_1 \circ \langle \pi_2, \pi_1 \rangle \rangle) &= \Lambda(eval_{A,B} \circ (\Phi(\Pi_1) \times id_A)) \\ &= \Lambda(\Lambda^{-1}(\Phi(\Pi_1))) \\ &= \Phi(\Pi_1) \end{aligned}$$

▶ Règle $\eta_\times$: Cherchons la flèche associée à l'arbre Π suivant :

$$\dfrac{\dfrac{\Pi_1}{\Gamma \vdash proj_1\ x : A}\ \times_E \qquad \dfrac{\Pi_1}{\Gamma \vdash proj_2\ x : B}\ \times_E}{\Gamma \vdash (proj_1\ x,\ proj_2\ x) : A \times B}\ \times_I$$

La flèche associée à Π par la définition précédente est $\Phi(\Pi) = \langle \pi_1 \circ \Phi(\Pi_1), \pi_2 \circ \Phi(\Pi_1) \rangle = \Phi(\Pi_1)$



▶ Règle η_1 : Si Π a pour conclusion $\Gamma \vdash t : 1$, la flèche $\Phi(\Pi)$ va de $\Gamma^\times$ à 1. Par unicité d'une telle flèche, $\Phi(\Pi) = \Phi(\overline{\Gamma \vdash () : 1})$ ■

Définition 2.22 Soit **Φ** la fonction à valeurs dans $\mathfrak{L}$ définie sur la catégorie $\Lambda_{\times 1\beta\eta}$ par :
- Pour tout objet A de $\Lambda_{\times 1\beta\eta}$, $\Phi(A) = A$
- Soit $f : A \rightarrow B$ une flèche de $\Lambda_{\times 1\beta\eta}$, et F un représentant de f, que l'on peut supposer de la forme $\lambda x.\ F'$ (quitte à faire une η-expansion).
  On appelle Π un arbre de preuve correspondant à F, sans la dernière règle ($\rightarrow_I$), et l'on pose $\Phi(f) = \Phi(\Pi)$.

Cette définition est valide parce que la conclusion de Π a au moins une hypothèse. Elle est indépendante du choix du représentant F d'après le lemme 2.21 page précédente.

Proposition 2.23 Φ est un foncteur entre $\Lambda_{\times 1\beta\eta}$ et $\mathfrak{L}$.

Démonstration :

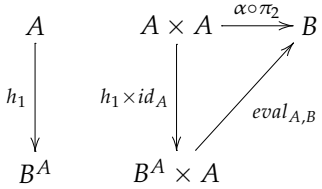
Soient $f : A \rightarrow B$ et $g : B \rightarrow C$ des flèches de $\Lambda_{\times 1\beta\eta}$ et F et G des représentants de ces deux classes respectivement, que l'on supposera être des λ-abstractions.

– On vérifie que $\Phi(f) : \Phi(A) \to \Phi(B)$ (trivial)
– $\Phi(id_A) = id_A$ (trivial)
– Cherchons la flèche associée à $g \circ f$. Par définition, $g \circ f = [G \circ F] = [\lambda x.\ G\ \ (F\ \ x)]$. On peut supposer F et G en forme normale de tête. L'arbre de preuve Π associé est le suivant :

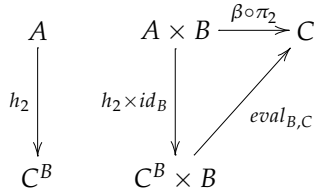
$$\dfrac{\dfrac{\dfrac{\Pi_2}{x : A,\ y : B\ \vdash\ G' : C}\ perm + aff}{x : A\ \vdash\ G : B \to C}\ \to_I \qquad \dfrac{}{x : A\ \vdash\ x : A} \qquad \dfrac{\dfrac{\Pi_1}{x : A,\ y : A\ \vdash\ F' : B}\ perm + aff}{x : A\ \vdash\ F : A \to B}\ \to_I}{x : A\ \vdash\ G\ \ (F\ \ x) : C}$$

où Π_1 et Π_2 sont les arbres de preuves associés respectivement à F et à G.
Notons $\alpha = \Phi(f)$ et $\beta = \Phi(g)$. Par définition de Φ, on a $\alpha = \Phi(\Pi_1)$ et $\beta = \Phi(\Pi_2)$. On veut montrer que $\Phi(\Pi) = \beta \circ \alpha$.

La flèche associée au sous-arbre $\overline{A\ \vdash\ A}$ est id_A.
La flèche associée au sous-arbre de racine $A\ \vdash\ A \to B$ est $h_1 = \Lambda(\alpha \circ \pi_2)$



La flèche associée au sous-arbre de racine $A\ \vdash\ B$ est la flèche $eval_{A,B} \circ \langle h_1, id_A \rangle = eval_{A,B} \circ (h_1 \times id_A) \circ \langle id, id \rangle = \alpha \circ \pi_2 \circ \langle id, id \rangle = \alpha$.
La flèche associée au sous-arbre de racine $A\ \vdash\ B \to C$ est la flèche $h_2 = \Lambda(\beta \circ \pi_2)$.



La flèche associée à Π est la flèche

$$\begin{aligned} eval_{B,C} \circ \langle h_2, \alpha \rangle &= eval_{B,C} \circ (h_2 \times id) \circ \langle id, \alpha \rangle \\ &= \beta \circ \pi_2 \circ \langle id, \alpha \rangle \\ &= \beta \circ \alpha \end{aligned}$$

■

Théorème 2.24 Les isomorphismes de $\Lambda_{\times 1\beta\eta}$ sont exactement ceux de $\mathfrak{L}$.

Démonstration :
$\Lambda_{\times 1\beta\eta}$ est une catégorie cartésienne fermée (d'après le théorème 2.13 page 55) donc tous

les isomorphismes des catégories cartésiennes fermées sont des isomorphismes de $\Lambda_{\times 1\beta\eta}$. La réciproque découle directement de l'existence du foncteur Φ. ■

Les catégories sont isomorphes

On peut même aller plus loin et montrer que les catégories sont isomorphes en montrant que Φ est une bijection. Pour cela, on définit une fonction Ψ en suivant la démonstration de la propriété 2.12 page 54 et du théorème 2.13 page 55 et on montrera que $\Phi \circ \Psi = id$ et $\Psi \circ \Phi = id$.

Définition 2.25 Soit Ψ la fonction définie par induction sur la structure de $\mathfrak{L}$ par :
- $\Psi(id_A) = [\lambda x : A.\ x]$
- Si $\Psi(f) = [F]$ et $\Psi(g) = [G]$ alors $\Psi(f \circ g) = [F \circ G]$
- $\Psi(!A) = [\lambda x.\ ()]$
- Soient $f : C \to A$ et $g : C \to B$ deux flèches de $\mathfrak{L}$. Si $\Psi(f) = [F]$ et $\Psi(g) = [G]$, on pose $\Psi(\langle f, g\rangle) = [\lambda x.\ (F\ \ x,\ G\ \ x)]$, $\Psi(\pi_1) = [proj_1]$ et $\Psi(\pi_2) = [proj_2]$.
- Soit $f : C \times A \to B$ une flèche de $\mathfrak{L}$. Si $\Psi(f) = [F]$, on pose $\Psi(\Lambda(f)) = [\lambda y.\ \lambda z.\ F\ \ (y,\ z)]$ et $\Psi(eval_{A,B}) = [\lambda p.\ ((proj_1\ \ p)\ \ (proj_2\ \ p))]$

Cette définition est valide parce que $\mathfrak{L}$ est une catégorie cartésienne fermée.
Notation : On pourra noter $\underline{\Psi}(f)$ un représentant quelconque de la classe $\Psi(f)$.

Proposition 2.26 Ψ est un foncteur.

Démonstration :

L'image d'une flèche de $\mathfrak{L}$ est une flèche de $\Lambda_{\times 1\beta\eta}$ entre les mêmes objets. La restriction de Ψ aux objets de $\mathfrak{L}$ est l'identité. L'identité est préservée. Reste à montrer que Ψ préserve aussi la composition, ce qui est vrai car $\Psi(f \circ g) = [F \circ G] = [F] \circ [G]$. ■

Notation : On notera $\boldsymbol{\lambda(x,\ y).\ M}$ le terme $\lambda t.\ M\left[(proj_1\ \ t)/_x,\ (proj_2\ \ t)/_y\right]$. Comme d'habitude, on parenthésera les n-uplets à gauche, et on étend la notation :
$\boldsymbol{\lambda((x_1, x_2), x_3 \ldots).\ M} = \lambda(x, x_3 ...).\ M\left[(proj_1\ \ x)/_{x_1},\ (proj_2\ \ x)/_{x_2}\right]$.

Théorème 2.27 Les catégories $\mathfrak{L}$ et $\Lambda_{\times 1\beta\eta}$ sont isomorphes.

Démonstration :

▶ Montrons $\boldsymbol{\Phi \circ \Psi = id}$ par induction sur la structure de $\mathfrak{L}$. (On garde les notations de la définition de Ψ.)
- $\Phi(\Psi(id_A)) = \Phi([\lambda x.\ x]) = id_A$
-
$$\begin{aligned}\Phi(\Psi(f \circ g)) &= \Phi([F \circ G]) \\ &= \Phi([F] \circ [G]) \\ &= \Phi([F]) \circ \Phi([G]) \text{ car } \Phi \text{ est un foncteur} \\ &= f \circ g \text{ par hypothèse d'induction}\end{aligned}$$
- $\Phi(\Psi(!A)) = \Phi([\lambda x.\ ()]) = !A$

– $\Phi(\Psi(\langle f, g\rangle)) = \Phi([\lambda x.\ (F\ \ x,\ G\ \ x)])$

$$\dfrac{\dfrac{}{x : C \vdash x : C}\quad \dfrac{\dfrac{\Pi_1}{x : C,\ y : C \vdash F' : A}}{x : C \vdash F : C \to A}\ perm + aff}{x : C \vdash F\ x : A} \qquad \dfrac{\dfrac{}{x : C \vdash x : C}\quad \dfrac{\dfrac{\Pi_2}{x : C,\ y : C \vdash G' : B}}{x : C \vdash G : C \to B}\ perm + aff}{x : C \vdash G\ x : B}$$

$$\dfrac{x : C \vdash F\ x : A \qquad x : C \vdash G\ x : B}{x : C \vdash (F\ x, G\ x) : A \times B}$$

La flèche associée à cette preuve Π est

$$\langle eval_{C,A} \circ \langle h_1, id_C\rangle, eval_{C,B} \circ \langle h_2, id_C\rangle\rangle$$

où h_1 vérifie $eval_{C,A} \circ (h_1 \times id_C) = \Phi(\Pi_1) \circ \pi_2$ et h_2 vérifie $eval_{C,A} \circ (h_2 \times id_C) = \Phi(\Pi_2) \circ \pi_2$.
Or $\pi_2 \circ \langle id, id\rangle = id$, donc en composant les deux précédentes égalités par $\langle id, id\rangle$ à droite, on obtient que la flèche associée à Π est $\langle \Phi(\Pi_1), \Phi(\Pi_2)\rangle$, qui est égale à $\langle f, g\rangle$ par hypothèse d'induction.

– $\Phi(\Psi(\pi_1)) = \Phi([proj_1]) = \pi_1$ et $\Phi(\Psi(\pi_2)) = \Phi([proj_2]) = \pi_2$
– $\Phi(\Psi(\Lambda(f))) = \Phi([\lambda y.\ \lambda z.\ F\ \ (y,\ z)])$

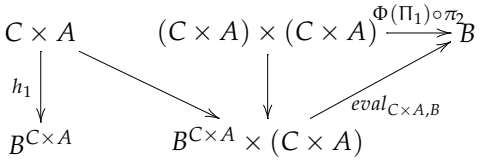
$$\dfrac{\dfrac{\dfrac{\Pi_1}{y : C,\ z : A,\ x : C \times A \vdash F' : B}\ perm + aff}{y : C,\ z : A \vdash F : C \times A \to B} \qquad \dfrac{\dfrac{\dfrac{\cdots}{y : C,\ z : A \vdash y : C}\ aff \qquad \dfrac{\cdots}{y : C,\ z : A \vdash z : A}\ perm + aff}{y : C,\ z : A \vdash (y,\ z) : C \times A}}{y : C,\ z : A \vdash F\ \ (y,\ z) : B}}{y : C \vdash \lambda z.\ F\ \ (y,\ z) : A \to B}$$

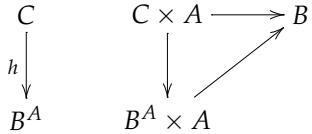
La flèche associée à cette preuve est l'unique h telle que

$$eval_{A,B} \circ (h \times id_A) = eval_{C\times A,B} \circ \langle h_1, id_{C\times A}\rangle$$

(car $\langle \pi_1, \pi_2\rangle = id_{C\times A}$) où h_1 est l'unique flèche vérifiant

$$eval_{C\times A,B} \circ (h_1 \times id_{C\times A}) = \Phi(\Pi_1) \circ \pi_2$$





Or $\pi_2 \circ \langle id_{C\times A}, id_{C\times A}\rangle = id_{C\times A}$ donc

$$\begin{aligned}\Phi(\Pi_1) &= eval_{C\times A,B} \circ (h_1 \times id_{C\times A}) \circ \langle id_{C\times A}, id_{C\times A}\rangle \\ &= eval_{C\times A,B} \circ \langle h_1, id_{C\times A}\rangle \\ &= eval_{A,B} \circ (h \times id_A)\end{aligned}$$

$$= \quad \Lambda^{-1}(h)$$

Par hypothèse d'induction $\Phi(\Pi_1) = f$, donc $h = \Lambda(\Phi(\Pi_1)) = \Lambda(f)$.

– $\Phi(\Psi(eval)) = \Phi([\lambda p.\ ((proj_1\ \ p)\ \ (proj_2\ \ p))]) = eval \circ \langle \pi_1, \pi_2 \rangle = eval$

▶ Montrons $\boldsymbol{\Psi \circ \Phi = id}$: Soit f une classe d'équivalence, et F un λ-terme de f. Si $F = proj_1$, $\Psi(\Phi(f)) = \Psi(\pi_1) = [proj_1] = f$. idem pour π_2. Sinon, soit Π l'arbre de dérivation de F sans la dernière règle ($\to_I$). On fait une induction sur la structure de Π (en gardant les notations de la définition 2.19 page 59) pour montrer que l'image par $\Psi \circ \Phi$ d'une preuve de conclusion $\Gamma \vdash M : A$ (où $\Gamma = (x_1 : A_1, \ldots, x_n : A_n)$) est la classe du λ-terme $\lambda(x_1, \ldots, x_n) : \Gamma^\times.\ M$.

Le cas où Γ est réduit à une seule formule correspond au résultat cherché.

– $\Psi(\Phi(\overline{A \vdash A})) = \Psi(id_A) = [\lambda x.\ x]$

– $\Psi(\Phi(\overline{\Gamma \vdash 1})) = \Psi(!\Gamma^\times) = [\lambda x.\ ()]$

– $\dfrac{\Gamma \vdash M : B}{\Gamma, A \vdash M : B}\ aff$

$$\begin{array}{rcl} \Psi(\Phi(\Pi)) & = & \Psi(\Phi(\Pi_1) \circ \pi_1) \\ & = & \Psi(\Phi(\Pi_1)) \circ \Psi(\pi_1) \text{ car } \Psi \text{ est un foncteur} \\ & = & \big[\lambda X : \Gamma^\times.\ M\big] \circ [proj_1] \text{ où } X = (x_1, ..., x_n) \\ & = & \big[\lambda (X, x) : \Gamma^\times \times A.\ M\big] \end{array}$$

– $\dfrac{\Gamma, A, B, \Delta \vdash M : C}{\Gamma, B, A, \Delta \vdash M : C}\ perm$

$$\begin{array}{rcl} \Psi(\Phi(\Pi)) & = & \Psi(\Phi(\Pi_1)) \circ \Psi(i) \\ & = & [\lambda(x_1, ..., x_i, x_{i+1}, ..., x_n).\ M] \circ \Psi(i) \\ & = & [\lambda(x_1, ..., x_{i+1}, x_i, ..., x_n).\ M] \end{array}$$

– $\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\ M : A \to B}\ \to_I$

$\Psi(\Phi(\Pi)) = [\lambda y.\ \lambda z.\ \underline{\Psi}(\Phi(\Pi_1))\ \ (y, z)]$.

Or $\Psi(\Phi(\Pi_1)) = [\lambda(X, x) : (\Gamma^\times \times A).\ M]$ par hypothèse d'induction, donc $\Psi(\Phi(\Pi)) = [\lambda y.\ \lambda z.\ M\,[{}^{y}/_{X},\ {}^{z}/_{x}]] = [\lambda X : \Gamma^\times.\ \lambda x : A.\ M]$

– $\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M,\ N) : A \times B}\ \times_I$

$$\begin{array}{rcl} \Psi(\Phi(\Pi)) & = & \Psi(\langle \Phi(\Pi_1), \Phi(\Pi_2) \rangle) \\ & = & [\lambda x.\ ((\underline{\Psi}(\Phi(\Pi_1))\ \ x),\ (\underline{\Psi}(\Phi(\Pi_2))\ \ x))] \\ & = & \big[\lambda x.\ (((\lambda X : \Gamma^\times.\ M)\ \ x),\ ((\lambda X : \Gamma^\times.\ N)\ \ x))\big] \text{ par hypothèse d'induction} \\ & = & \big[\lambda X : \Gamma^\times.\ (M,\ N)\big] \text{ par } \beta\text{-réduction et } \alpha\text{-conversion} \end{array}$$

– $\dfrac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B}\ \to_E$

$$\begin{array}{rcl} \Psi(\Phi(\Pi)) & = & \Psi(eval_{A,B} \circ \langle \Phi(\Pi_1), \Phi(\Pi_2) \rangle) \\ & = & \Psi(eval_{A,B}) \circ [\lambda x.\ ((\underline{\Psi}(\Phi(\Pi_1))\ \ x),\ (\underline{\Psi}(\Phi(\Pi_2))\ \ x))] \text{ car } \Psi \text{ est un foncteur} \\ & = & [\lambda p.\ ((proj_1\ \ p)\ \ (proj_2\ \ p))] \circ \big[\lambda X : \Gamma^\times.\ (M,\ N)\big] \\ & = & \big[\lambda p.\ ((proj_1\ \ p)\ \ (proj_2\ \ p)) \circ \lambda X : \Gamma^\times.\ (M,\ N)\big] \end{array}$$

$$= \quad [\lambda X : \Gamma^{\times}.\ (M\ \ N)]$$

– $\dfrac{\Gamma \vdash M : A \times B}{\Gamma \vdash proj_1\ \ M : A}\ \times_{E1}$

$$\begin{aligned}\Psi(\Phi(\Pi)) &= \Psi(\pi_1 \circ \Phi(\Pi_1)) \\ &= \Psi(\pi_1) \circ \Psi(\Phi(\Pi_1)) \text{ car } \Psi \text{ est un foncteur} \\ &= [proj_1] \circ [\lambda X : \Gamma^{\times}.\ M] \\ &= [\lambda X : \Gamma^{\times}.\ proj_1\ \ M]\end{aligned}$$

(même chose pour $\times_{E2}$)

2.3.2 Avec co-produit

Je vais maintenant prouver un résultat similaire en ajoutant le type somme et le type vide.

Définitions

Définition 2.28 On appelle $\mathfrak{L}_{o,+}$ la catégorie bi-cartésienne fermée engendrée à partir de l'ensemble des types de base.

Les isomorphismes sont les mêmes

Pour continuer, nous avons besoin de la proposition 1.40 page 38, qui définit deux isomorphismes

$$\delta = \binom{id_A \times \iota_1}{id_A \times \iota_2} : (A \times B) + (A \times C) \to A \times (B + C)$$

et

$$\gamma = \Lambda^{-1}\binom{\Lambda(\iota_1 \circ \langle \pi_2, \pi_1 \rangle)}{\Lambda(\iota_2 \circ \langle \pi_2, \pi_1 \rangle)} \circ \langle \pi_2, \pi_1 \rangle : A \times (B + C) \to (A \times B) + (A \times C)$$

Nous allons maintenant étendre la définition de Φ à tout $\Lambda_{\times+10\beta\eta}$. On commence encore une fois par définir d'abord Φ sur les dérivations de typage :

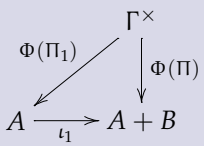
Définition 2.29 Soit Π un arbre de preuve du calcul des séquents de conclusion $\Gamma \vdash A$, dans laquelle tous les séquents ont au moins une hypothèse. $\boldsymbol{\Phi(\Pi)}$ est une flèche de $\mathfrak{L}_{o,+}$ entre $\Gamma^{\times}$ et A définie par induction sur la structure de Π.

Les premiers cas sont identiques à la définition 2.19 page 59. On ajoute les quatre cas suivants :

– $\dfrac{\Gamma \vdash 0}{\Gamma \vdash A}\ 0_E$ (axiome intuitionniste) On pose $\Phi(\Pi) = \mathfrak{i}A \circ \Phi(\Pi_1)$.

– $\dfrac{\Gamma \vdash A}{\Gamma \vdash A + B}\ +_{I1}$

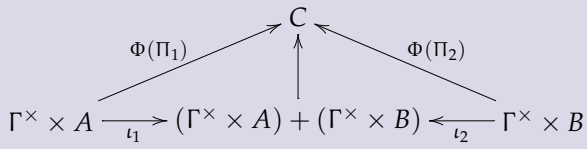
$\Phi(\Pi_1)$ est une flèche entre $\Gamma^{\times}$ et A. On pose $\Phi(\Pi) = \iota_1 \circ \Phi(\Pi_1)$.



– $\iota_2 \circ \Phi(\Pi_1)$ pour $+_{I2}$.

– $\dfrac{\Gamma,\ A \vdash C \quad \Gamma,\ B \vdash C \quad \Gamma \vdash A+B}{\Gamma \vdash C}\ +_E$

$\binom{\Phi(\Pi_1)}{\Phi(\Pi_2)}$ est une flèche entre $(\Gamma^\times \times A) + (\Gamma^\times \times B)$ et C.



La flèche

$$\begin{aligned}\gamma &= \Lambda^{-1}(\binom{\Lambda(\iota_1 \circ \langle \pi_2, \pi_1 \rangle)}{\Lambda(\iota_2 \circ \langle \pi_2, \pi_1 \rangle)}) \circ \langle \pi_2, \pi_1 \rangle \\ &= eval_{\Gamma^\times,(\Gamma^\times \times A)+(\Gamma^\times \times B)} \circ \langle \binom{\Lambda(\iota_1 \circ \langle \pi_2, \pi_1 \rangle)}{\Lambda(\iota_2 \circ \langle \pi_2, \pi_1 \rangle)} \circ \pi_2, \pi_1 \rangle\end{aligned}$$

va de $\Gamma^\times \times (A+B)$ à $(\Gamma^\times \times A) + (\Gamma^\times \times B)$
(Une catégorie bi-cartésienne fermée est distributive)
Enfin $\langle id_{\Gamma^\times}, \Phi(\Pi_3) \rangle$ est une flèche entre $\Gamma^\times$ et $\Gamma^\times \times (A+B)$.
On prend

$$\Phi(\Pi) = \binom{\Phi(\Pi_1)}{\Phi(\Pi_2)} \circ \gamma \circ \langle id_{\Gamma^\times}, \Phi(\Pi_3) \rangle$$

Lemme 2.30 Si Π et Π' sont deux arbres de typage du même λ-terme, alors $\Phi(\Pi) = \Phi(\Pi')$.

Démonstration :

Il faut rajouter les cas suivants à la démonstration du lemme 2.20 :

▶ Règle $+_{I1}$: $(\iota_1 \circ \Phi(\Pi_1)) \circ \alpha = \iota_1 \circ (\Phi(\Pi_1) \circ \alpha)$

▶ Règle $+_{I2}$: idem

▶ Règle $+_E$:

$$\Phi(\Pi) = \binom{\Phi(\Pi_1)}{\Phi(\Pi_2)} \circ \gamma \circ \langle id_{\Gamma^\times}, \Phi(\Pi_3) \rangle \circ \alpha$$

$$\Phi(\Pi') = \binom{\Phi(\Pi_1) \circ (\alpha \times id)}{\Phi(\Pi_2) \circ (\alpha \times id)} \circ \gamma \circ \langle id_{\Gamma^\times}, \Phi(\Pi_3) \circ \alpha \rangle$$

$$\begin{aligned}\Phi(\Pi') &= \binom{\Phi(\Pi_1)}{\Phi(\Pi_2)} \circ \gamma \circ \delta \circ \binom{\iota_1 \circ (\alpha \times id)}{\iota_2 \circ (\alpha \times id)} \circ \gamma \circ \langle id_{\Gamma^\times}, \Phi(\Pi_3) \circ \alpha \rangle \\ &= \binom{\Phi(\Pi_1)}{\Phi(\Pi_2)} \circ \gamma \circ (\alpha \times id) \circ \delta \circ \gamma \circ \langle id_{\Gamma^\times}, \Phi(\Pi_3) \circ \alpha \rangle \\ &= \binom{\Phi(\Pi_1)}{\Phi(\Pi_2)} \circ \gamma \circ \langle id_{\Gamma^\times}, \Phi(\Pi_3) \rangle \circ \alpha\end{aligned}$$

▶ Règle 0_E : $(\mathfrak{i}A \circ \Phi(\Pi_1)) \circ \alpha = \mathfrak{i}A \circ (\Phi(\Pi_1) \circ \alpha)$ ■

Lemme 2.31 Si Π et Π' sont deux arbres de preuve $\beta\eta$-équivalents, alors $\Phi(\Pi) = \Phi(\Pi')$.

Démonstration :

Il faut rajouter à la preuve du lemme 2.21 page 61 les cas suivants :

▶ Règle η_+ : Cherchons la flèche associée à l'arbre Π suivant :

$$\dfrac{\dfrac{\dfrac{\Pi_1}{\Gamma,\ x:A\ \vdash\ h:A+B\to C}\ \mathit{aff}\ \dfrac{}{\Gamma,\ x:A\ \vdash\ in_1\ \ x:A+B}}{\Gamma,\ x:A\ \vdash\ h\ \ (in_1\ \ x):C}\quad\dfrac{\ldots}{\Gamma,\ x:B\ \vdash\ h\ \ (in_2\ \ x):C}\quad\dfrac{\Pi_2}{\Gamma\ \vdash\ a:A+B}}{\Gamma\ \vdash\ case\,(a,\ x.\,h\ \ (in_1\ \ x),\ x.\,h\ \ (in_2\ \ x)):C}$$

$$\begin{array}{ccccc} & & \Gamma^\times\times A & & \\ & {\scriptstyle \Phi(\Pi_1)\circ\pi_1}\swarrow & \downarrow & \searrow{\scriptstyle \iota_1\circ\pi_2} & \\ C^{A+B} & \longleftarrow & C^{A+B}\times(A+B) & \longrightarrow & A+B \end{array}$$

$$\begin{array}{rcl} \Phi(\Pi) & = & \begin{pmatrix} eval_{A+B,C}\circ\langle\Phi(\Pi_1)\circ\pi_1,\iota_1\circ\pi_2\rangle \\ eval_{A+B,C}\circ\langle\Phi(\Pi_1)\circ\pi_1,\iota_2\circ\pi_2\rangle \end{pmatrix}\circ\gamma\circ\langle id_{\Gamma^\times},\Phi(\Pi_2)\rangle \\ & = & eval_{A+B,C}\circ(\Phi(\Pi_1)\times id_C)\circ\delta\circ\gamma\circ\langle id_{\Gamma^\times},\Phi(\Pi_2)\rangle \\ & = & eval_{A+B,C}\circ\langle\Phi(\Pi_1),\Phi(\Pi_2)\rangle \text{ car } \delta\circ\gamma = id \end{array}$$

Cherchons maintenant la flèche associée à l'arbre Π' suivant :

$$\dfrac{\dfrac{\Pi_1}{\Gamma\ \vdash\ h:A+B\to C}\quad\dfrac{\Pi_2}{\Gamma\ \vdash\ a:A+B}}{\Gamma\ \vdash\ (h\ \ a):C}$$

$$\Phi(\Pi') = eval_{A+B,C}\circ\langle\Phi(\Pi_1),\Phi(\Pi_2)\rangle$$

▶ Règle β_{+1} : Cherchons la flèche associée à l'arbre Π suivant :

$$\dfrac{\Pi_1\quad\Pi_2\quad\dfrac{\Pi_3}{\Gamma\ \vdash\ in_1\ \ a:A+B}\ +_I}{\Gamma\ \vdash\ case\,(in_1\ \ a,\ f,\ g):C}$$

$$\begin{array}{rcl} \Phi(\Pi) & = & \begin{pmatrix}\Phi(\Pi_1)\\ \Phi(\Pi_2)\end{pmatrix}\circ\gamma\circ\langle id_{\Gamma^\times},\iota_1\circ\Phi(\Pi_3)\rangle \\ & = & \begin{pmatrix}\Phi(\Pi_1)\\ \Phi(\Pi_2)\end{pmatrix}\circ eval\ \circ\langle\begin{pmatrix}\Lambda(\iota_1\circ\langle\pi_2,\pi_1\rangle)\\ \Lambda(\iota_2\circ\langle\pi_2,\pi_1\rangle)\end{pmatrix}\circ\iota_1\circ\Phi(\Pi_3),id_{\Gamma^\times}\rangle \\ & = & \begin{pmatrix}\Phi(\Pi_1)\\ \Phi(\Pi_2)\end{pmatrix}\circ eval\ \circ\langle\Lambda(\iota_1\circ\langle\pi_2,\pi_1\rangle)\circ\Phi(\Pi_3),id_{\Gamma^\times}\rangle \\ & = & \begin{pmatrix}\Phi(\Pi_1)\\ \Phi(\Pi_2)\end{pmatrix}\circ\Lambda^{-1}(\Lambda(\iota_1\circ\langle\pi_2,\pi_1\rangle))\circ\langle\Phi(\Pi_3),id_{\Gamma^\times}\rangle \\ & = & \begin{pmatrix}\Phi(\Pi_1)\\ \Phi(\Pi_2)\end{pmatrix}\circ\iota_1\circ\langle\pi_2,\pi_1\rangle\circ\langle\Phi(\Pi_3),id_{\Gamma^\times}\rangle \\ & = & \Phi(\Pi_1)\circ\langle id_{\Gamma^\times},\Phi(\Pi_3)\rangle \end{array}$$

▶ Règle β_{+2} : idem

▶ Règle η_0 : La flèche associée à l'arbre $\Pi = \dfrac{\Pi_1}{\Gamma \vdash \bot_A(t) : A}$ est $¡A \circ \Phi(\Pi_1)$. Soit Π' un arbre de preuve de conclusion $\overline{\Gamma \vdash t' : A}$. La flèche $\Phi(\Pi') \circ ¡\Gamma^\times$ a pour domaine 0 et co-domaine A. Par unicité d'une telle flèche, $¡A = \Phi(\Pi') \circ ¡\Gamma^\times$. Donc

$$¡A \circ \Phi(\Pi_1) = \Phi(\Pi') \circ ¡\Gamma^\times \circ \Phi(\Pi_1)$$

Or d'après le lemme 1.42 page 41, $¡\Gamma^\times \circ \Phi(\Pi_1) = id_{\Gamma^\times}$, ce qui donne le résultat recherché. ■

Définition 2.32 Soit Φ la fonction à valeurs dans $\mathfrak{L}_{o,+}$ définie sur la catégorie $\Lambda_{\times+10\beta\eta}$ par :
- Pour tout objet A de $\Lambda_{\times+10\beta\eta}$, $\Phi(A) = A$
- Soient $f : A \to B$ une flèche de $\Lambda_{\times+10\beta\eta}$, et F un représentant de f, que l'on peut supposer de la forme $\lambda x.\ F'$, quitte à faire une η-expansion.
  On appelle Π un arbre de preuve correspondant à F, sans la dernière règle $(\to_I)$, et l'on pose $\Phi(f) = \Phi(\Pi)$.

Cette définition est indépendante du choix du représentant F d'après la proposition précédente.

Proposition 2.33 Φ est un foncteur entre $\Lambda_{\times+10\beta\eta}$ et $\mathfrak{L}_{o,+}$.

Démonstration :
La même que celle de la proposition 2.23. ■

Théorème 2.34 Les isomorphismes de $\Lambda_{\times+10\beta\eta}$ sont exactement ceux de $\mathfrak{L}_{o,+}$.

Démonstration :
La même que celle du théorème 2.24. ■

Les catégories sont isomorphes

Définition 2.35 Soit Ψ la fonction définie en ajoutant les cas suivants à la définition 2.25 page 64 :
- Soient $f : A \to C$ et $g : B \to C$ deux flèches de $\mathfrak{L}_{o,+}$. Si $\Psi(f) = [F]$ et $\Psi(g) = [G]$, on pose $\Psi(\binom{f}{g}) = [\lambda x.\ case\ (x,\ F,\ G)]$, $\Psi(\iota_1) = [in_1]$ et $\Psi(\iota_2) = [in_2]$.
- $\Psi(¡A) = [\lambda x : 0.\ \bot_A(x)]$.

Cette définition est valide parce que $\mathfrak{L}_{o,+}$ est une catégorie bi-cartésienne fermée.

Proposition 2.36 Ψ est un foncteur.

Démonstration :
La même que celle de la proposition 2.26. ■

Théorème 2.37 Les catégories $\mathfrak{L}_{o,+}$ et $\Lambda_{\times+10\beta\eta}$ sont isomorphes.

Démonstration :

▶ Montrons $\boldsymbol{\Phi \circ \Psi = id}$: Il ne manque que les cas suivants à la preuve du théorème 2.27 :

– $\Phi(\Psi(\imath A)) = \Phi([\lambda x.\ \bot_A(x)]) = \imath A$

– Soient F et G des représentants en forme normale de tête de f et g.

$$\begin{aligned}
\Phi(\Psi(\begin{pmatrix} f \\ g \end{pmatrix})) &= \Phi([\lambda x.\ case\,(x,\ F,\ G)]) \\
&= \begin{pmatrix} \Phi([F]) \circ \pi_2 \\ \Phi([G]) \circ \pi_2 \end{pmatrix} \circ \gamma \circ \langle id_{A+B}, id_{A+B} \rangle \\
&= \begin{pmatrix} f \circ \pi_2 \\ g \circ \pi_2 \end{pmatrix} \circ \gamma \circ \langle id_{A+B}, id_{A+B} \rangle \\
&= \begin{pmatrix} \binom{f}{g} \circ \iota_1 \circ \pi_2 \\ \binom{f}{g} \circ \iota_2 \circ \pi_2 \end{pmatrix} \circ \gamma \circ \langle id_{A+B}, id_{A+B} \rangle \\
&= \begin{pmatrix} \binom{f}{g} \circ \pi_2 \circ (id \times \iota_1) \\ \binom{f}{g} \circ \pi_2 \circ (id \times \iota_2) \end{pmatrix} \circ \gamma \circ \langle id_{A+B}, id_{A+B} \rangle \\
&= \begin{pmatrix} f \\ g \end{pmatrix} \circ \pi_2 \circ \begin{pmatrix} id \times \iota_1 \\ id \times \iota_2 \end{pmatrix} \circ \gamma \circ \langle id_{A+B}, id_{A+B} \rangle \\
&= \begin{pmatrix} f \\ g \end{pmatrix} \circ \pi_2 \circ \delta \circ \gamma \circ \langle id_{A+B}, id_{A+B} \rangle \\
&= \begin{pmatrix} f \\ g \end{pmatrix}
\end{aligned}$$

$$\frac{\dfrac{\Pi_1}{x : A+B,\ y : A \ \vdash\ F' : C}\ {\scriptstyle aff+perm} \qquad \dfrac{\Pi_2}{x : A+B,\ y : B \ \vdash\ G' : C}\ {\scriptstyle aff+perm} \qquad \dfrac{}{x : A+B \ \vdash\ x : A+B}}{x : A+B \ \vdash\ case\,(x,\ F,\ G) : C}$$

– $\Phi(\Psi(\iota_1)) = \Phi([in_1]) = \iota_1$, de même pour ι_2.

▶ Montrons $\boldsymbol{\Psi \circ \Phi = id}$: Il ne nous manque que les cas suivants :

– $\dfrac{\Gamma,\ A \ \vdash\ f' : C \quad \Gamma,\ B \ \vdash\ g' : C \quad \Gamma \ \vdash\ a : A+B}{\Gamma \ \vdash\ case(a, f, g) : C}\ +_E$

$\Psi(\Phi(\Pi)) = \Psi(\binom{\Phi(\Pi_1)}{\Phi(\Pi_2)}) \circ \gamma \circ \langle id_{\Gamma\times}, \Phi(\Pi_3) \rangle)$

$= \Psi(\binom{\Phi(\Pi_1)}{\Phi(\Pi_2)})) \circ \Psi(\gamma) \circ \Psi(\langle id_{\Gamma\times}, \Phi(\Pi_3) \rangle)$ car Ψ est un foncteur

$= [\lambda x.\ case\,(x,\ \underline{\Psi}(\Phi(\Pi_1)),\ \underline{\Psi}(\Phi(\Pi_2)))]$

$\circ\, [\lambda p.\ ((proj_1\ \ p)\ \ (proj_2\ \ p))]$

$\circ \left[\lambda x.\ ((\underline{\Psi}(\binom{\Lambda(\iota_1 \circ \langle \pi_2,\ \pi_1 \rangle)}{\Lambda(\iota_2 \circ \langle \pi_2,\ \pi_1 \rangle)}) \circ \pi_2)\ x),\ (proj_1\ \ x))\right]$

$\circ\, [\lambda x.\ (x,\ \underline{\Psi}(\Phi(\Pi_3)))]$

$= \ldots$

$\circ \left[\lambda x.\ (\left((\underline{\Psi}(\binom{\Lambda(\iota_1 \circ \langle \pi_2,\ \pi_1 \rangle)}{\Lambda(\iota_2 \circ \langle \pi_2,\ \pi_1 \rangle)}))\ \ (\underline{\Psi}(\Phi(\Pi_3))\ x)\right),\ x)\right]$

$= \ldots$

$\circ \left[\lambda x.\ (\left((\lambda y.\ case \begin{pmatrix} y, \\ \underline{\Psi}(\Lambda(\iota_1 \circ \langle \pi_2,\ \pi_1 \rangle))), \\ \underline{\Psi}(\Lambda(\iota_2 \circ \langle \pi_2,\ \pi_1 \rangle)) \end{pmatrix}\right)\ \ (\underline{\Psi}(\Phi(\Pi_3))\ x)\right),\ x)\right]$

$= \ldots$

$$
\begin{aligned}
&\circ \left[\lambda x.\ (case \begin{pmatrix} \underline{\Psi}(\Phi(\Pi_3))\ x, \\ \underline{\Psi}(\Lambda(\iota_1 \circ \langle \pi_2,\ \pi_1 \rangle)), \\ \underline{\Psi}(\Lambda(\iota_2 \circ \langle \pi_2,\ \pi_1 \rangle)) \end{pmatrix},\ x)\right] \\
&= \ldots \\
&\circ \left[\lambda x.\ (case \begin{pmatrix} \underline{\Psi}(\Phi(\Pi_3))\ x, \\ \lambda y.\ \lambda z.\ \underline{\Psi}(\iota_1 \circ \langle \pi_2, \pi_1 \rangle)\ \ (y,\ z), \\ \lambda y.\ \lambda z.\ \underline{\Psi}(\iota_2 \circ \langle \pi_2, \pi_1 \rangle)\ \ (y,\ z) \end{pmatrix},\ x)\right] \\
&= \ldots \\
&\circ \left[\lambda x.\ (case \begin{pmatrix} \underline{\Psi}(\Phi(\Pi_3))\ x, \\ \lambda y.\ \lambda z.\ (in_1 \circ \lambda t.\ (proj_2\ \ t,\ proj_1\ \ t))\ \ (y,\ z), \\ \lambda y.\ \lambda z.\ (in_2 \circ \lambda t.\ (proj_2\ \ t,\ proj_1\ \ t))\ \ (y,\ z) \end{pmatrix},\ x)\right] \\
&= \ldots \\
&\circ \left[\lambda x.\ (case \begin{pmatrix} \underline{\Psi}(\Phi(\Pi_3))\ x, \\ \lambda y.\ \lambda z.\ (in_1\ \ (z,\ y)), \\ \lambda y.\ \lambda z.\ (in_2\ \ (z,\ y)) \end{pmatrix},\ x)\right] \\
&= [\lambda x.\ case\ (x,\ \underline{\Psi}(\Phi(\Pi_1)),\ \underline{\Psi}(\Phi(\Pi_2)))] \\
&\circ \left[\lambda x.\ (case \begin{pmatrix} \underline{\Psi}(\Phi(\Pi_3))\ x, \\ \lambda y.\ \lambda z.\ (in_1\ \ (z,\ y)), \\ \lambda y.\ \lambda z.\ (in_2\ \ (z,\ y)) \end{pmatrix}\ x)\right] \\
&= [\lambda x.\ case\ (x,\ \underline{\Psi}(\Phi(\Pi_1)),\ \underline{\Psi}(\Phi(\Pi_2)))] \\
&\circ \left[\lambda x.\ (case \begin{pmatrix} \underline{\Psi}(\Phi(\Pi_3))\ x, \\ h \circ in_1, \\ h \circ in_2 \end{pmatrix}\ x)\right]
\end{aligned}
$$

avec $h = \lambda t.\ \lambda z.\ case\ (t,\ w.\ in_1\ \ (z,\ w),\ w.\ in_2\ \ (z,\ w))$

$= [\lambda x.\ case\ (x,\ \underline{\Psi}(\Phi(\Pi_1)),\ \underline{\Psi}(\Phi(\Pi_2)))]$

$\circ\, [\lambda x.\ (h\ \ (\underline{\Psi}(\Phi(\Pi_3))\ \ x))\ \ x]$ d'après la règle η_+

$= [\lambda x.\ case\ ((h\ \ (\underline{\Psi}(\Phi(\Pi_3))\ \ x))\ \ x,\ \underline{\Psi}(\Phi(\Pi_1)),\ \underline{\Psi}(\Phi(\Pi_2)))]$

$= [\lambda x.\ case\ ((\underline{\Psi}(\Phi(\Pi_3))\ \ x)\ \ x,\ u.\ \underline{\Psi}(\Phi(\Pi_1))\ \ (x,\ u),\ u.\ \underline{\Psi}(\Phi(\Pi_2))\ \ (x,\ u))]$

d'après le lemme 2.6 page 52

D'après l'hypothèse d'induction,

$$\Psi(\Phi(\Pi_1)) = \lambda t : \Gamma^{\times} \times A.\ f' =_{\beta\eta} \lambda t : \Gamma^{\times}.\ f$$

$$\Psi(\Phi(\Pi_2)) = \lambda t : \Gamma^{\times} \times B.\ g' =_{\beta\eta} \lambda t : \Gamma^{\times}.\ g$$

$$\Psi(\Phi(\Pi_3)) = \lambda t : \Gamma^{\times}.\ a$$

Donc $\Psi(\Phi(\Pi)) = \lambda t : \Gamma^{\times}.\ case\ (a,\ f,\ g)$ après α-conversion et η-conversion.

– $\Psi \circ \Phi([in_1]) = \Psi(\iota_1) = [in_1]$ idem pour $[in_2]$.

– $\Psi(\Phi([\lambda x : 0.\ \bot_A(x)])) = \Psi(\mathfrak{i}A) = \lambda x : 0.\ \bot_A(x)$

■

2.3.3 Note sur la règle d'élimination du 0

Jusqu'à présent, nous avons utilisé la règle d'élimination suivante pour le type vide :

$$\frac{\Gamma \vdash t : 0}{\Gamma \vdash \bot_\tau(t) : \tau}$$

Il sera parfois utile, notamment dans le chapitre 5, de considérer une autre règle, dans laquelle le terme ne conserve pas de témoin de l'inconsistance du contexte :

$$\frac{\Gamma \vdash t : 0}{\Gamma \vdash \bot_\tau : \tau}$$

Cette règle nous permettra d'avoir une seule forme normale de type τ dans le cas d'un contexte inconsistant. Cependant elle ne vérifie pas toutes les propriétés habituelles des règles de typage, ce qui nous oblige à être prudent. Par exemple, si l'on a $\Gamma, x : \tau' \vdash t : \tau$ et que $x \notin FV(t)$, on ne peut plus en déduire que $\Gamma \vdash t : \tau$. Il suffit de considérer par exemple $x : 0 \vdash \bot_\tau : \tau$ qui est un séquent valide avec la nouvelle règle. Avec l'ancienne, on aurait $x : 0 \vdash \bot_\tau(x) : \tau$ et $x \in FV(\bot_\tau(x))$.

Nous pouvons vérifier que certains résultats importants montrés précédemment restent vrais avec cette nouvelle règle, notamment :

- Le lemme 2.31 page 69 est toujours vrai. En effet, d'après le lemme 1.42 page 41, pour Γ inconsistant, $\Gamma^\times$ est initial, donc la flèche $(\Phi(\Pi_1))$ entre $\Gamma^\times$ et 0 est unique.
- On pose $\Psi(\mathfrak{i}A) = [\lambda x : 0.\ \bot_A]$, et la démonstration du théorème final reste la même.

Deuxième partie

Isomorphismes de types avec somme

Chapitre 3

Isomorphismes linéaires en logique linéaire

Résumé

Ce chapitre donne une caractérisation remarquablement simple et élégante des isomorphismes linéaires dans le cadre de la logique linéaire multiplicative, en utilisant en particulier le critère de correction pour les réseaux de preuves dû à Jean-Yves Girard.

NOUS ALLONS commencer l'étude des isomorphismes avec somme par le cas plus simple des isomorphismes dits « linéaires », que nous allons étudier dans le cadre de la logique linéaire multiplicative. L'étude des isomorphismes linéaires a un intérêt par exemple pour la recherche dans les bibliothèques de composantes logicielles. Ils correspondent aux isomorphismes entre des objets de catégories sites symétriques monoïdales. On peut les voir également comme les isomorphismes de types dans le système du λ-calcul correspondant à la logique linéaire intuitionniste multiplicative.

Sergei Soloviev a montré dans [78] que la théorie constituée des axiomes 1, 2, 3, 5 et 7 de la figure 4.1 page 95 définit une relation d'équivalence sur les types qui coïncide avec la relation d'isomorphisme de types linéaire. Une preuve en théorie des modèles de ce résultat est présentée dans [42]. Enfin il existe un algorithme très efficace pour décider les isomorphismes linéaires, présenté dans [4].

Plutôt qu'étudier ainsi les isomorphismes linéaires de formules intuitionnistes, nous allons nous intéresser dans ce chapitre, aux isomorphismes linéaires à l'intérieur du fragment multiplicatif de la logique linéaire (MLL, Girard [50]), qui est le cadre naturel pour étudier les effets de la linéarité.

Nous verrons que les isomorphismes dans MLL ne sont pas les mêmes qu'en λ-calcul linéaire : MLL est un système plus riche, qui permet d'écrire des formules, comme $A^{\perp} \otimes A$, et des preuves qui n'ont pas de correspondance en λ-calcul linéaire. Nous entrons donc dans un monde plus fin. Nous verrons qu'une conséquence particulièrement intéressante de ce changement de point de vue est que les axiomes des isomorphismes linéaires sont réduits à une forme remarquablement

simple, à savoir l'associativité et la commutativité pour les connecteurs logiques $\otimes$ et $\parr$ de MLL, et les axiomes triviaux pour les constantes.

Par exemple, si nous interprétons l'implication $A \to B$ du λ-calcul linéaire par l'implication linéaire $A \multimap B$ de la logique linéaire, qui est équivalente à $A^\perp \parr B$, l'isomorphisme $\mathbb{1} \to A \cong A$ devient juste la règle de l'élément neutre $\bot \parr A \cong A$. De le même façon, la curryfication $(A \times B) \to C \cong A \to (B \to C)$ devient simplement l'associativité du connecteur $\parr$, à savoir $(A^\perp \parr B^\perp) \parr C \cong A^\perp (\parr B^\perp \parr C)$ et $(A \to (B \to C) \cong B \to (A \to C))$ devient juste $A^\perp \parr (B^\perp \parr C) \cong B^\perp \parr (A^\perp \parr C)$, qui est une conséquence évidente de l'associativité et la commutativité du $\parr$.

Nous commencerons par rappeler dans la section 3.1 les principales définitions et propriétés de la logique linéaire.

3.1 La logique linéaire

Je vais rappeler ici très brièvement quelques définitions de logique linéaire. Pour une introduction complète, on pourra se référer à l'article fondateur de Jean-Yves Girard [50].

3.1.1 Les règles de la logique linéaire

La logique linéaire permet de donner à la logique une gestion plus rigoureuse des ressources en interdisant la duplication et l'élimination d'hypothèses et de conclusions. La présentation habituelle de la logique linéaire utilise le calcul des séquents classique, duquel on a retiré les règles de contraction et d'affaiblissement. Cette modification a pour effet de faire disparaître l'équivalence entre les présentations dites ***additive*** et ***multiplicative*** des règles des connecteurs $\vee$ et $\wedge$, d'où la nécessité de distinguer un $\vee$ multiplicatif (« par », noté $\parr$), un $\vee$ additif (« avec », noté $\&$), un $\wedge$ multiplicatif (« tenseur », noté $\otimes$), et un $\wedge$ additif (« plus », noté $\oplus$). Les atomes sont de la forme A ou $A^\perp$. La définition de la négation $^\perp$ est étendue à toutes les formules par les règles suivantes :

$$(A \parr B)^\perp = (A^\perp \otimes B^\perp) \qquad (A \otimes B)^\perp = (A^\perp \parr B^\perp)$$
$$(A \oplus B)^\perp = (A^\perp \& B^\perp) \qquad (A \& B)^\perp = (A^\perp \oplus B^\perp)$$

Les symboles $\mathbf{1}$, $\mathbf{0}$, $\top$ et $\bot$ représentent respectivement les éléments neutres des connecteurs $\otimes$, $\oplus$, $\&$ et $\parr$. L'implication linéaire ($\multimap$) peut être définie par $A \multimap B = A^\perp \parr B$.

Des connecteurs ! et ? sont également utilisés pour réintroduire de manière contrôlée l'affaiblissement et la contraction. L'utilisation des connecteurs de la logique linéaire permet de décomposer l'implication intuitionniste en deux opérations plus simples : $A \to B$ correspond en fait à $!A \multimap B$. En fait, $!A$ signifie que l'on a A autant de fois que l'on veut. Il peut être réutilisé ; ce qui correspond à l'utilisation d'une hypothèse en mathématiques, ou à une mise en mémoire en informatique. Lorsque l'on écrit $A \multimap B$, cela signifie que l'on a B, mais en utilisant A. Ensuite A disparaît.

Les règles de la logique linéaire propositionnelle sont rappelées sur la figure 3.1. Nous appellerons MLL (sans constantes) le fragment constitué des seuls connecteurs multiplicatifs et MALL (sans constantes) le fragment constitué des connecteurs multiplicatifs et additifs, toujours sans les exponentielles. Nous allons nous intéresser ici au fragment MLL, d'abord sans, puis avec constantes.

$$\frac{}{\vdash A^\perp, A}\ ax \qquad \frac{\vdash A, \Delta \quad \vdash A^\perp \Delta'}{\vdash \Delta, \Delta'}\ cut$$

$$\frac{\vdash A, B, \Delta}{\vdash A \parr B, \Delta}\ \parr \qquad \frac{\vdash A, \Delta \quad \vdash B, \Delta'}{\vdash A \otimes B, \Delta, \Delta'}\ \otimes$$

$$\frac{\vdash A, \Delta \quad \vdash B, \Delta}{\vdash A \& B, \Delta}\ \& \qquad \frac{\vdash A, \Delta}{\vdash A \oplus B, \Delta}\ \oplus_1 \qquad \frac{\vdash B, \Delta}{\vdash A \oplus B, \Delta}\ \oplus_2$$

$$\frac{}{\vdash \mathbf{1}}\ \mathbf{1} \qquad \frac{\vdash \Delta}{\vdash \bot, \Delta}\ \bot$$

(pas de règle pour 0) $\qquad \dfrac{}{\vdash \top, \Delta}\ \top$

$$\frac{\vdash \Delta}{\vdash ?A, \Delta}\ \mathrm{w} \qquad \frac{\vdash ?A, ?A, \Delta}{\vdash ?A, \Delta}\ \mathrm{c}$$

$$\frac{\vdash A, \Delta}{\vdash ?A, \Delta}\ \mathrm{dir} \qquad \frac{\vdash A, ?\Delta}{\vdash !A, ?\Delta}\ \mathrm{pro}$$

FIG. 3.1 – Les règles de la logique linéaire.

3.1.2 Les réseaux de preuves

Les réseaux de preuve ont été introduits par Girard comme transposition en logique linéaire de la déduction naturelle de la logique intuitionniste. Un réseau est un graphe particulier dont les nœuds sont des formules de la logique linéaire.

Un réseau représente la preuve d'un séquent. Un avantage par rapport au calcul des séquents est de supprimer tous les problèmes de permutation de règles, ce qui réduit les indéterminismes lors de la construction de la preuve.

Nous allons nous intéresser d'abord aux réseaux multiplicatifs sans constantes. Définissons d'abord les *structures de preuves*.

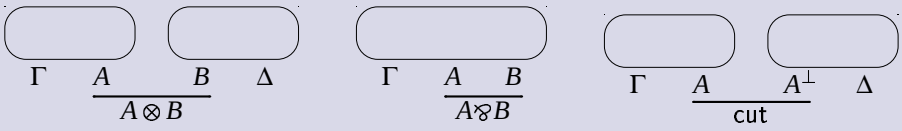
Définition 3.1 (structure de preuve) Une ***structure de preuve*** (ou ***pré-réseau***) est un graphe dont les sommets sont des formules, construit à partir des ***liens*** suivants :

$$\frac{\mathsf{ax}}{A \qquad A^\perp} \qquad \frac{A \qquad B}{A \parr B} \qquad \frac{A \qquad B}{A \otimes B} \qquad \frac{A \qquad B}{\mathsf{cut}}$$

(où A et B sont des formules quelconques), et vérifiant :
- toute formule est prémisse d'au moins un lien,
- toute formule est conclusion d'exactement un lien.

Les réseaux sont des structures de preuves particulières :

Définition 3.2 (réseau de preuve) Un ***réseau de preuve*** est une structure de preuve définie inductivement à partir de ***liens axiomes*** $\dfrac{\text{ax}}{A \quad A^\perp}$ grâce aux trois règles de construction suivantes :



Dans cette définition, chaque boîte représente un réseau de preuve. On peut déduire de cette définition que pour tout réseau ayant au moins une conclusion $\otimes$, il en existe une d'elles qui, si on l'enlève, déconnecte le réseau. On appellera un tel nœud un ***tenseur*** **split** .

Le théorème suivant fait le lien entre le calcul des séquents (séquentiels) et les réseaux de preuve (non séquentiels).

Théorème 3.3 Il existe un réseau de preuve de conclusions Γ si et seulement si il existe une preuve en calcul des séquents de conclusion $\vdash \Gamma$.

On peut définir une opération de réduction sur les réseaux, dite ***élimination des coupures***, qui correspond à la β-réduction dans le λ-calcul. S'il existe un réseau de conclusions Γ, alors il en existe un sans liens **cut** .

La procédure d'élimination des coupures est décrite figure 3.1.2.

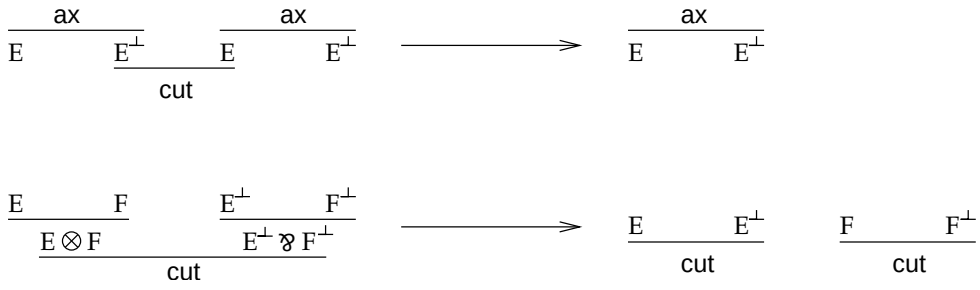


FIG. 3.2 – Élimination des coupures dans les réseaux MLL.

3.2 Définitions et présentation des résultats

Un isomorphisme de formules de MLL est défini formellement de la manière suivante :

Définition 3.4 Deux formules E et F sont dites ***isomorphes*** si
- E et F sont linéairement équivalentes, i.e. $\vdash E^\perp, F$ et $\vdash F^\perp, E$
- il existe des preuves de $\vdash E^\perp, F$ et de $\vdash F^\perp, E$ qui, lorsqu'elles sont composées par un lien **cut** , se réduisent après élimination des coupures en une preuve de $\vdash E^\perp, E$ (resp. $\vdash F^\perp, F$)), qui est l'η-expansion de l'axiome $\vdash E^\perp, E$ (resp $\vdash F^\perp, F$)

Nous allons montrer que deux formules E et F sont linéairement isomorphes si et seulement

si $AC(\otimes, \parr) \vdash E = F$ dans le cas de MLL sans constantes, et $ACI(\otimes, \parr) \vdash E = F$ dans le cas de MLL avec constantes où $AC(\otimes, \parr)$ et $ACI(\otimes, \parr)$ sont définis de la façon suivante :

Définition 3.5 ($AC(\otimes, \parr)$) Soit $AC(\otimes, \parr)$ l'ensemble constitué des quatre équations suivantes :

$$\begin{array}{rclcrcl} X \otimes Y & = & Y \otimes X & \qquad & (X \parr Y) \parr Z & = & X \parr (Y \parr Z) \\ X \parr Y & = & Y \parr X & & (X \otimes Y) \otimes Z & = & X \otimes (Y \otimes Z) \end{array}$$

$AC(\otimes, \parr) \vdash E = F$ signifie que $E = F$ appartient à la théorie équationnelle engendrée par $AC(\otimes, \parr)$ sur l'ensemble des formules de la logique linéaire ; en d'autres termes, cela signifie que les formules sont égales modulo associativité et commutativité de $\parr$ et $\otimes$.

Définition 3.6 ($ACI(\otimes, \parr)$) Soit $ACI(\otimes, \parr)$ l'ensemble constitué des équations de $AC(\otimes, \parr)$ auxquelles on ajoute :

$$X \otimes 1 = X \qquad X \parr \bot = X$$

$ACI(\otimes, \parr) \vdash E = F$ signifie que $E = F$ appartient à la théorie équationnelle engendrée par $ACI(\otimes, \parr)$.

Comme toujours, dans l'étude de théories d'objets isomorphes, la correction est facile à prouver :

Théorème 3.7 (Correction de la théorie) Si $ACI(\otimes, \parr) \vdash E = F$, alors E et F sont linéairement isomorphes.

Démonstration :

Il suffit d'exhiber les réseaux pour les axiomes et de montrer la clôture par contexte. ■

Toute la difficulté réside dans la preuve de la seconde implication, à savoir la complétude. Le reste du chapitre est consacré à cette preuve. La preuve que nous proposons ici utilise la notion de réseau de preuve.

Nous allons avoir besoin de la notion suivante :

Définition 3.8 (réseau simple) Un réseau est dit ***simple*** s'il ne contient que des liens axiomes atomiques (c'est-à-dire des liens axiomes entre formules atomiques).

Dans le cas de MLL, il est facile de prouver le résultat suivant :

Proposition 3.9 (η-expansion des réseaux de preuves MLL) Pour tout réseau S, il existe un réseau simple avec les mêmes conclusions, que nous noterons $\eta(S)$.

Démonstration :

Il suffit d'itérer une simple procédure d'η-expansion des liens non-atomiques, qui peuvent être remplacés par deux liens axiomes et un lien $\parr$ et un lien $\otimes$. ■

Cela signifie que, en ce qui concerne la prouvabilité, nous pouvons restreindre notre attention aux réseaux simples. Nous allons montrer que nous pouvons également le faire en ce qui concerne les isomorphismes.

Nous allons d'abord donner une caractérisation des formules isomorphes dans MLL sans

constantes, en réduisant l'isomorphisme à l'existence de deux réseaux de preuves particuliers appelés *réseaux de preuves simples bipartites.* . Ensuite, nous allons montrer que nous pouvons nous restreindre aux *formules non-ambiguës* (c'est-à-dire aux formules dans lesquelles les atomes apparaissent au plus une fois positivement et une fois négativement). Cette caractérisation en terme de réseaux permet de prouver la complétude de $AC(\otimes, \parr)$ pour MLL(sans constante) par une induction sur la taille du réseau.

En présence de constantes, nous simplifions d'abord les formules en enlevant tous les nœuds de la forme $\bot \parr E$ et $1 \otimes E$. Ensuite, nous remarquons que les isomorphismes pour les formules simplifiées sont très similaires au cas sans constantes. En utilisant une propriété remarquable des réseaux pour les formules simplifiées, nous pouvons en effet réduire la preuve de complétude au cas sans constante.

3.3 Réduction aux réseaux de preuves simples bipartites

Nous allons tout d'abord formaliser la réduction en réseaux simples.

Définition 3.10 (arbre d'une formule, réseau simple identité) Un réseau sans coupure S prouvant E est en fait composé de l'*arbre* de E, (noté $T(E)$), et d'un ensemble de liens axiomes sur des formules atomiques. Nous appellerons ***réseau simple identité*** de E le réseau simple sans coupure obtenu par une η-expansion complète du réseau $\overline{E \qquad E^\perp}$ (a priori pas simple). Ce réseau est constitué de $T(E)$, $T(E^\perp)$ et d'un ensemble de liens axiomes qui connectent des atomes de $T(E)$ avec des atomes de $T(E^\perp)$.
Notons que, dans les réseaux simples, l'axiome identité pour E est interprété par le réseau simple identité de E.

Remarquons tout d'abord que nous pouvons nous attacher aux témoins d'isomorphismes qui sont des réseaux de preuves simples, ce qui va simplifier le traitement des isomorphismes linéaires.

Lemme 3.11 (Réseaux simples/réseaux non-simples) Si un réseau (non-simple) S se réduit par élimination des coupures en S', alors le réseau simple $\eta(S)$ se réduit en $\eta(S')$.

Démonstration :

Il suffit de montrer que si $S \triangleright_1 S'$ (réduction en une étape), alors $\eta(S) \triangleright^* \eta(S')$ (réduction de longueur arbitraire). Si le redex R réduit dans S ne contient aucun lien axiome, alors exactement le même redex apparaît dans $\eta(S)$, et il peut donc être réduit en une étape en $\eta(S')$. Sinon R contient un lien axiome, qui peut être non atomique (s'il est atomique, alors le redex considéré est exactement le même dans S et dans $\eta(S)$ et la propriété est évidente). Si F est non atomique et $\underline{F \quad F^\perp}$ est le lien **cut** de R, soit n le nombre d'atomes de F (comptés avec leur multiplicité). Alors F a $n-1$ connecteurs, et $\eta(S)$ peut se réduire en $\eta(S')$ en $2n-1$ étapes ($n-1$ étapes pour propager le liens **cut** vers les formules atomiques et n étapes pour réduire chaque lien atomique produit de cette façon). ■

Théorème 3.12 (Réduction en réseaux simples) Deux formules E et F sont ***isomorphes*** si et seulement si il existe deux réseaux simples S de conclusions $E^\perp, F$ et S' de conclusions $F^\perp, E$ qui, lorsqu'ils sont composés par un lien **cut** sur F (resp. E), se réduisent après élimination des coupures en le réseau simple identité de E (resp. F).

Démonstration :

La partie « *si* » de la preuve est triviale, car un réseau de preuve représente une preuve et la réduction des coupures dans les réseaux correspond à l'élimination des coupures dans les preuves.
Pour la partie « *seulement si* », prenons deux preuves témoins de l'isomorphisme et construisons les réseaux de preuves associés S et S'. Ces réseaux ont pour conclusions $E^\perp, F$ (resp. $F^\perp, E$), et nous savons que après les avoir composés par un **cut** sur F (resp. E) et appliqué la réduction des coupures, on obtient un réseau axiome pour E (resp. F). Maintenant prenons les η-expansions complètes de S et S' pour les réseaux simples requis : le lemme 3.11 page ci-contre nous montre qu'ils se réduisent après composition sur F (resp. E) en le réseau simple identité de E (resp. F). ■

Nous allons maintenant montrer que si deux formules sont isomorphes, alors l'isomorphisme peut être prouvé par le moyen de réseaux de preuves de structure particulièrement simple.

Définition 3.13 (réseau de preuve simple bipartite) Un réseau simple sans coupure est dit ***bipartite*** s'il a exactement deux conclusions E et F, et s'il est constitué de $T(E)$, $T(F)$ et d'un ensemble de liens axiomes connectant des atomes de E à des atomes de F, mais pas d'atomes de E entre eux ni d'atomes de F entre eux (un exemple est présenté figure 3.3).

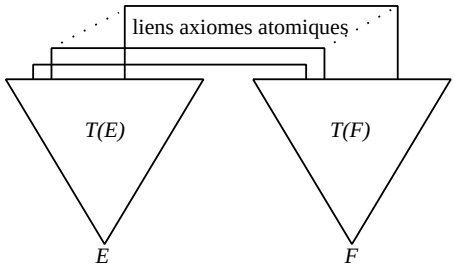


FIG. 3.3 – Forme d'un réseau de preuve bipartite.

Lemme 3.14 (Arbres et coupures) Soit S une structure (qui n'est *pas* un réseau de preuve) sans conclusion construit uniquement à partir de $T(E)$ et $T(E^\perp)$, sans lien axiome, et du lien **cut** $\underline{E \quad E^\perp}$. Alors la réduction des coupures appliquée sur S conduit à un résultat constitué uniquement d'un ensemble de liens **cut** atomiques $\underline{p_i \quad p_i^\perp}$ entres des atomes de E et des atomes de $E^\perp$.

Démonstration :

Par simple induction sur la taille de la structure. ■

Théorème 3.15 (Les isomorphismes sont bipartites) Soit S un réseau de preuve simple sans coupure de conclusions $E^\perp$ et F, et S' un réseau de preuve simple sans coupure de conclusions $F^\perp$ et E. Si leur composition par **cut** donne, respectivement les réseaux simples identités de E et F, alors S et S' sont bipartites.

Démonstration :

Nous allons montrer la contraposée : si S ou S' n'est pas bipartite, alors le résultat de l'élimination des coupures n'est pas bipartite, et donc n'est pas un réseau simple identité. Pour des raisons de symétrie, nous pouvons supposer que S n'est pas bipartite, et contient un lien axiome entre deux atomes de $E^{\perp}$. Nous allons montrer que la composition de S et S' par un **cut** sur F ne donne pas un réseau bipartite.
Comme S et S' sont sans coupure, leur composition ne contient qu'un seul lien **cut** (entre F et $F^{\perp}$). Comme S et S' sont simples, tout lien axiome de la composition est atomique. Donc chaque lien axiome (atomique) du réseau $\underline{S \quad S'}$ qui est réduit par élimination des coupures est connecté à un atome de F ou de $F^{\perp}$. En particulier, le lien axiome de S qui est est connecté à des atomes de E *n'est pas réduit*. Ce qui prouve que l'élimination des coupures dans $\underline{S \quad S'}$ ne donne pas un réseau bipartite.
Le théorème s'en déduit. ■

3.4 Réduction à des formules non-ambiguës

Pour prouver le théorème de correction, nous allons d'abord montrer que nous pouvons restreindre notre étude aux formules non-ambiguës, c'est-à-dire aux formules dans lesquelles chaque atome apparaît au plus une fois positivement et une fois négativement.

Définition 3.16 (formules non-ambiguës) Nous dirons qu'une formule E est ***non-ambiguë*** si chaque atome de E apparaît au plus un fois positivement et au plus une fois négativement.

Par exemple, $A \otimes B$ et $A \otimes A^{\perp}$ sont non-ambiguës, alors que $A \otimes A$ est ambiguë. Dans la suite, nous appellerons ***substitution*** l'opération usuelle $[G_1/A_1, ...G_n/A_n]$ de remplacement des atomes propositionnels A_i d'une formule par les formules G_i. Une substitution sera dénotée par une lettre grecque $\sigma, \tau, ...$ Nous considérerons également les substitutions étendues aux réseaux de preuves tout entiers : si R est un réseau, $\sigma(R)$ sera le réseau obtenu à partir de R en remplaçant toute formule A_j apparaissant dans R par $\sigma(A_j)$.

Nous allons également avoir besoin d'une notion plus faible, que l'on appellera ***renommage***, qui pourra remplacer, dans un réseau, *différentes occurrences* d'un *même* atome par différentes formules.

Définition 3.17 (renommage) Une application α de l'ensemble des *occurrences* des atomes d'un réseau de preuve R dans un ensemble d'atomes est appelée un ***renommage pour R*** si $\alpha(R)$ (le réseau obtenu par substitution de chaque occurrence p d'un atome de R par $\alpha(p)$) est un réseau de preuve correct.

Remarquons que si R est simple, la définition de α sur les seules occurrences d'atomes dans les *liens axiomes* est suffisante pour définir α sur toutes les occurrences dans R. Si R est simple et bipartite, alors la définition de α sur les seules occurrences d'atomes dans *une des conclusions* de R est suffisante pour définir α sur toute occurrence dans R.
Notons aussi que, si les conclusions de R sont des formules ambiguës, alors deux différentes occurrences du même atome peuvent être renommées différemment, contrairement à ce qu'il se passe dans le cas des substitutions.

Comme on pouvait l'espérer, une formule non-ambiguë ne peut être isomorphe qu'à une for-

mule non-ambiguë.

Lemme 3.18 (formules isomorphes non-ambiguës) Soient E et F des formules isomorphes, et α un renommage tel que $\alpha(E)$ est non-ambiguë, alors $\alpha(F^\perp)$ est non-ambiguë.

Démonstration :

Si E et F sont des formules isomorphes, alors (en utilisant le théorème 3.15 page 83) il existe un réseau de preuve bipartite de conclusions $E, F^\perp$. Comme α est un renommage, il existe aussi un *réseau de preuve bipartite de conclusions* $\alpha(E), \alpha(F^\perp)$. Donc, $\alpha(E)$ et $\alpha(F^\perp)$ ont exactement les mêmes atomes. Et comme $\alpha(E)$ est non-ambiguë, $\alpha(F^\perp)$ est aussi non-ambiguë. ■

Nous allons maintenant prouver que les isomorphismes sont invariants par renommage.

Théorème 3.19 (Le renommage préserve les isomorphismes) Si E et F sont isomorphes, soient R et R' les réseaux simples associés (de conclusions $E^\perp$, F et $F^\perp$, E respectivement). Si α est un renommage des (occurrences d') atomes de R, alors il existe α', renommage des atomes de R' tel que $\alpha'(E)$ et $\alpha(F)$ sont isomorphes, i.e. :

- $\alpha'(R')$ est un réseau de preuve correct,
- $\alpha(E^\perp) \equiv (\alpha'(E))^\perp$ et $\alpha'(F^\perp) \equiv (\alpha(F))^\perp$
- La composition de $\alpha(R)$ et $\alpha'(R')$ par une coupure sur $\alpha(F)$ (resp. $\alpha'(E)$) donne le réseau simple identité de $\alpha'(E)$ (resp. $\alpha(F)$).

Démonstration :

Nous devons commencer par définir α'. Comme, d'après le théorème 3.15 page 83, R' est bipartite, il suffit de définir α' sur les occurrences de $F^\perp$, c'est-à-dire définir $\alpha'(F^\perp)$. Nous pouvons poser : $\alpha'(F^\perp) \equiv (\alpha(F))^\perp$. La composition de $\alpha(R)$ et $\alpha'(R')$ par une coupure sur F est un réseau de preuve correct. Et puisque la réduction des réseaux ne dépend en rien des étiquettes des nœuds du réseau, la composition se réduit en un réseau identité de conclusions $\alpha(E^\perp)$ et $\alpha'(E)$. Une induction simple (sur le nombre de connecteurs de $\alpha(E)$) montre que l'η-réduction de ce réseau donne un lien axiome. On a donc $\alpha(E^\perp) \equiv (\alpha'(E))^\perp$.
Mais alors, la composition de $\alpha'(R')$ avec $\alpha(R)$ par une coupure sur $\alpha'(E)$ est un réseau de preuve correct, qui se réduit en un réseau identité (puisque la réduction des réseaux ne dépend pas des étiquettes), et ce réseau simple identité est celui de $\alpha(F)$.
On en déduit que $\alpha'(E)$ et $\alpha(F)$ sont isomorphes. ■

Nous pouvons maintenant montrer qu'il est possible de limiter notre attention aux formules non-ambiguës.

Lemme 3.20 (formules isomorphes ambiguës) Soient E et F des formules isomorphes, telles que E est ambiguë. Il existe une substitution σ et des formules E' et F' *non-ambiguës*, telles que E' et F' sont isomorphes et vérifient $E \equiv \sigma(E')$ et $F \equiv \sigma(F')$.

Démonstration :

Soient R et R' des réseaux bipartites de conclusions $F^\perp$, E et $E^\perp$, F respectivement, associés à l'isomorphisme entre E et F. Puisqu'il est suffisant de définir α seulement sur les occurrences d'atomes de E, on peut définir un renommage α tel que $\alpha(E)$ a uniquement des atomes distincts (i.e. aucun atome de $\alpha(E)$ n'apparaît plusieurs fois dans $\alpha(E)$, même une fois positivement et une fois négativement). En particulier, $\alpha(E)$ est non-ambiguë. Le théorème 3.19 donne alors un algorithme pour définir un renommage α' tel que $\alpha(E)$ et $\alpha'(F)$ sont isomorphes : en particulier $\alpha'(E^\perp) \equiv (\alpha(E))^\perp$ et $\alpha(F^\perp) \equiv (\alpha'(F))^\perp$.

Soient $E' \equiv \alpha(E)$ et $F' \equiv \alpha'(F)$. D'après le théorème 3.19 page précédente, E' et F' sont isomorphes et $\alpha(R)$ a pour conclusions E' et ${F'}^{\perp}$.

Nous pouvons définir sur $\alpha(R)$ un renommage α^{-1} tel que $\alpha^{-1}(E') \equiv E$, et donc $\alpha^{-1}({F'}^{\perp}) \equiv F^{\perp}$.

Puisque $\alpha(R)$ est bipartite, il est équivalent de définir α^{-1} sur les occurrences de R, ou seulement sur les occurrences des atomes de E'. Mais comme tous les atomes de E' sont distincts, deux occurrences distinctes d'atomes de E' correspondent à des atomes distincts de E'. Nous pouvons donc définir une substitution σ sur les atomes de E' par : $\sigma(p) \equiv \alpha^{-1}(Occ(p))$ où $Occ(p)$ est la *seule* occurrence de l'atome p dans E'.

Ainsi, $R \equiv \alpha^{-1}(\alpha(R)) \equiv \sigma(\alpha(R))$. En particulier $\sigma(E') \equiv E$ et $\sigma({F'}^{\perp}) \equiv \sigma(\alpha(F^{\perp})) \equiv \alpha^{-1}(\alpha(F^{\perp})) \equiv F^{\perp}$, donc $\sigma(F') \equiv F$.

Enfin, E' et F' sont des formules non-ambiguës isomorphes telles que $\sigma(E') \equiv E$ et $\sigma(F') \equiv F$. ■

Corollaire 3.21 (réduction à des formules non-ambiguës) L'ensemble des couples de formules isomorphes est l'ensemble des instances (par substitution sur les atomes) de couples de formules isomorphes *non-ambiguës*.

Démonstration :

Nous montrons chacune des deux inclusions séparément.

Soient E et F deux formules isomorphes. D'après le lemme 3.20 page précédente, E et F sont des instances de deux formules isomorphes non-ambiguës E' et F'.

À l'inverse, soient C et D des formules isomorphes et σ une substitution sur les atomes C (et donc aussi les atomes de D). Soient R et R' deux réseaux de preuves bipartites associés à C et D. La substitution σ définit sur R un renommage α (chaque substitution peut être vue comme un renommage). Soit α' le renommage défini sur R', associé à α comme dans le théorème 3.19 page précédente. Comme $\sigma(C^{\perp}) \equiv \alpha(C^{\perp}) \equiv (\alpha'(C))^{\perp}$, α' est aussi le renommage induit par σ sur R'. D'après le théorème 3.19 page précédente, $\alpha(C)$ et $\alpha(D)$ sont isomorphes. Donc $\sigma(C)$ et $\sigma(D)$ sont isomorphes. ■

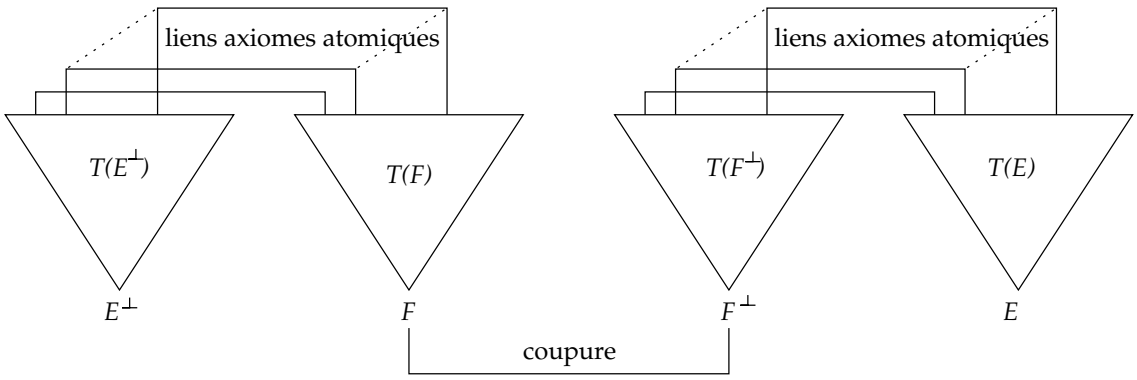
Nous pouvons donc, dans ce qui suit, nous intéresser uniquement aux formules non-ambiguës.

Nous sommes maintenant en mesure de montrer que pour deux formules non-ambiguës, l'existence même de réseaux simples bipartites implique l'isomorphisme.

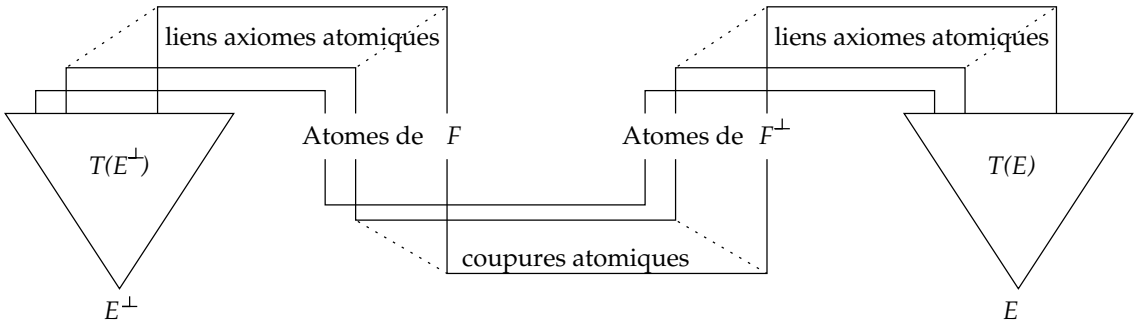
Théorème 3.22 (réseaux simples bipartites pour des formules non-ambiguës) Soit S un réseau simple bipartite sur $E^{\perp}$ et F, et S' un réseaux simple bipartite sur $F^{\perp}$ et E. Leur composition par une coupure sur F se réduit au réseau simple identité de E (resp. leur composition par une coupure sur E se réduit au réseau simple identité sur F).

Démonstration :

Considérons la composition de S et S' par une coupure sur F (voir figure).



D'après le lemme 3.14 page 83, l'élimination des coupures dans le réseau $\underline{T(F) \quad T(F^\perp)}$ aboutit à un ensemble de liens **cut** atomiques entre atomes de F et atomes de $F^\perp$. Comme S et S' sont bipartites, chacun de ces *liens* **cut** *atomiques* est connecté à un *lien axiome atomique* entre un atome de $E^\perp$ et un atome de F, et à un *lien axiome atomique* entre un atome de $F^\perp$ et un atome de E. Le réseau contient désormais uniquement des redex atomiques composés de coupures et de liens axiomes. La réduction de ces redex donne l'arbre identité de E (puisqu'il n'y a aucun lien axiome reliant des atomes de E — resp. $({}^\perp E)$ entre eux).



Les théorèmes 3.22 page ci-contre et 3.15 page 83 ont la conséquence fondamentale suivante :

Corollaire 3.23 Deux formules E et F sont isomorphes si et seulement si il existe des réseaux simples bipartites de conclusions $E^\perp, F$, et $F^\perp, E$.

3.5 Complétude pour les isomorphismes dans MLL

En utilisant le résultat de la section ci-dessus et le simple lemme suivant, nous sommes en mesure de prouver le résultat principal du chapitre, à savoir la complétude de $AC(\otimes, \parr)$ pour MLL sans constante.

Lemme 3.24 (formules isomorphes) Si E et F sont linéairement isomorphes, elles sont soit toutes deux des formules $\parr$, soit toutes deux des formules $\otimes$.

Démonstration :

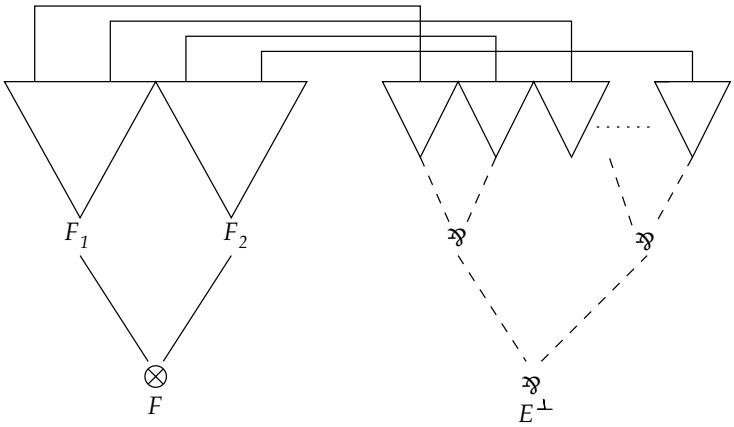
Effectivement, une formule $\parr$ et une formule $\otimes$ ne peuvent pas être isomorphes. Si $E_1 \parr E_2$ et $E_3 \otimes E_4$ étaient isomorphes, il existerait un réseaux simple bipartite de conclusions $({E_1}^\perp \otimes {E_2}^\perp), (E_3 \otimes E_4)$, ce qui est impossible, car un tel réseau n'a pas de tenseur *split* terminal. En effet, enlever un des deux tenseurs terminaux ne déconnecte pas le graphe car il est bipartite. Donc deux formules isomorphes sont forcément soit toutes deux des $\parr$ soit des $\otimes$. ■

Théorème 3.25 (Complétude) Si E et F sont linéairement isomorphes, elles vérifient $AC(\otimes, \parr) \vdash E = F$.

Démonstration :

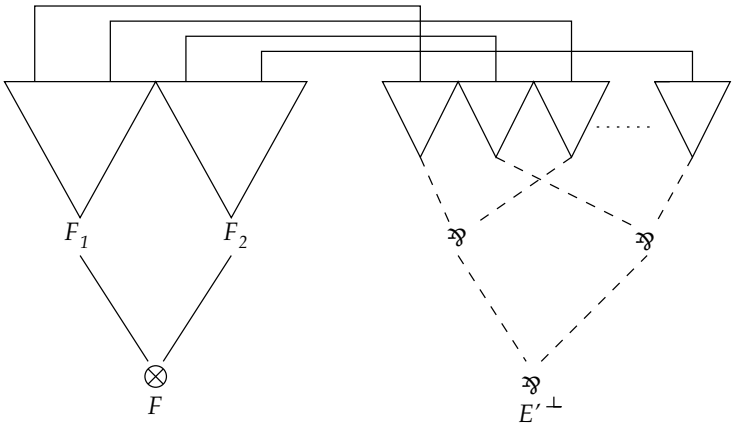
La preuve se fait par induction sur la taille du réseau simple bipartite de conclusions $E^\perp, F$ donné par l'isomorphisme.
Si E et F sont atomiques, la propriété est évidente.
Sinon les formules $E^\perp$ et F sont *toutes deux* non atomiques (puisque le réseau est bipartite, elles doivent contenir le même nombre d'atomes). De plus, (d'après le lemme 3.24 page précédente) puisque E et F sont isomorphes, l'une des formules $E^\perp$, F est un $\parr$ et l'autre un $\otimes$. Nous pouvons supposer par exemple que F est un $\otimes$, et $E^\perp$ un $\parr$.

Enlevons maintenant tous les nœuds $\parr$ « pendants » dans le réseau de conclusions $E^\perp, F$. On obtient un réseau de preuve correct dont les conclusions sont de la forme ${E_1}^\perp, \ldots, {E_k}^\perp, F_1 \otimes F_2$. Si l'une des formules ${E_1}^\perp, \ldots, {E_k}^\perp$ est un nœud tenseur, il n'est pas possible d'obtenir deux graphes déconnectés en enlevant ce lien, puisque (étant donné que le réseau initial est bipartite) chaque atome de ${E_1}^\perp, \ldots, {E_k}^\perp$ est relié à la formule $F_1 \otimes F_2$.

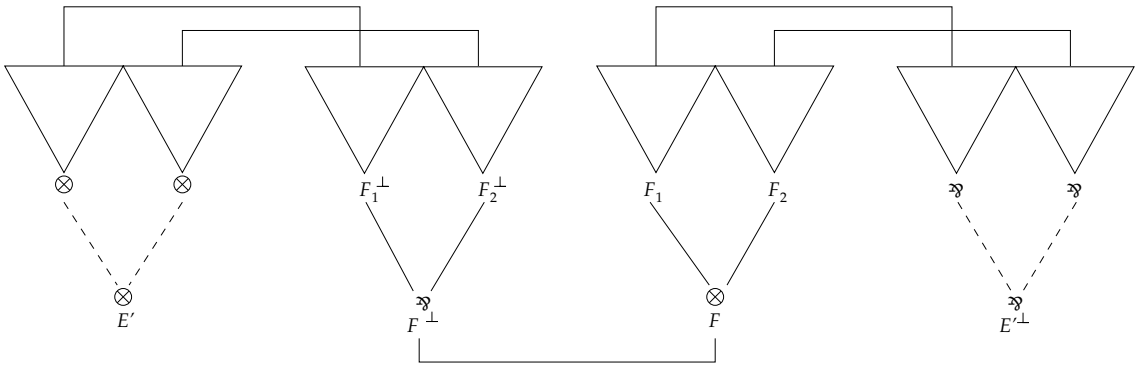


Le nœud tenseur *split* (qui doit exister d'après le critère de correction de Girard) est nécessairement le nœud $F_1 \otimes F_2$. On obtient, en le supprimant, deux réseaux déconnectés, qui sont toujours simples puisque les liens axiomes n'ont pas été modifiés.

Nous pouvons reconstruire à partir de ces réseaux deux réseaux de preuves simples et bipartites (un réseau bipartite doit avoir exactement deux conclusions) en ajoutant des liens $\parr$ sous les E_i (grâce à $AC(\otimes, \parr)$, l'ordre des E_i n'est pas important). Ceci construit deux formules ${E'_1}^\perp$ (reliée à F_1) et ${E'_2}^\perp$ (reliée à F_2). Les réseaux obtenus contiennent au moins un lien de moins que le réseau de départ, et donc sont strictement plus petits. Il ne nous reste plus qu'à vérifier que F_1 et E'_1 sont isomorphes, et que F_2 et E'_2 sont isomorphes, de façon à pouvoir appliquer l'hypothèse d'induction.



Pour ceci, nous utilisons le fait que les deux formules initiales sont isomorphes. Si nous remettons en place le $\parr$ et $\otimes$ de départ, nous obtenons deux formules E' et F, isomorphes (d'après le théorème 3.7 page 81). Il existe donc un réseau simple bipartite de conclusions E' et $F^{\perp}$. Comme les formules sont non-ambiguës, nous pouvons extraire de ce réseau deux réseaux simples bipartites ; l'un de conclusions E'_1 et ${F_1}^{\perp}$, l'autre de conclusions E'_2 et ${F_2}^{\perp}$. Ainsi, d'après le théorème 3.22 page 86, E'_1 et F_1 sont isomorphes, et E'_2 et F_2 sont isomorphes.



Par hypothèse d'induction, $AC(\otimes, \parr) \vdash E'_1 = F_1$ et $AC(\otimes, \parr) \vdash E'_2 = F_2$, et donc $AC(\otimes, \parr) \vdash E' = F$. Nous concluons en utilisant le fait que $AC(\otimes, \parr) \vdash E = E'$. ■

3.6 Prise en compte des constantes

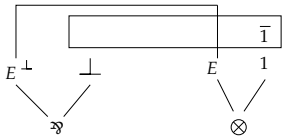
Nous avons montré ci-dessus des résultats de correction et de complétude pour la théorie des isomorphismes de MLL.. Cela correspond aux isomorphismes des catégories *-autonomes, qui est un sur-ensemble des catégories Symétriques Monoïdales Fermées *sans constantes*. Maintenant, si nous voulons un résultat intéressant également en termes de modèles, et prendre en compte les catégories Symétriques Monoïdales Fermées, nous devons aussi traiter le cas des constantes.

3.6.1 Expansions des axiomes avec constantes : les réseaux simples identité revus et corrigés

En présence des constantes **1** et $\bot$, les réseaux de preuve deviennent souvent beaucoup plus complexes, car $\bot$ force à introduire la notion de *boîte* .

Rappelons ici les règles de formation des réseaux pour les constantes :

L'expansion d'un axiome peut maintenant contenir des boîtes, si la formule fait intervenir des constantes. Par exemple, l'axiome $\vdash (E \otimes \mathbf{1})^{\bot}, (E \otimes \mathbf{1})$ est complètement η-expansé en :



3.6.2 Réduction aux isomorphismes entre formules simplifiées

Remarquons d'abord qu'une formule de la forme $1 \otimes E$ est toujours isomorphe à E, et que $\bot \parr E$ est isomorphe à E.

Définition 3.26 (formules simplifiées) Une formule est dite ***simplifiée*** si elle n'a aucune sous-formule de la forme $1 \otimes E$ ou $\bot \parr E$ (où E est une formule quelconque). À chaque formule E, on associe la formule $s(E)$ obtenue en normalisant E d'après le système de réécriture canonique suivant (confluent et fortement normalisant) :

$$1 \otimes E \to E \qquad E \otimes 1 \to E \qquad \bot \parr E \to E \qquad E \parr \bot \to E$$

Nous pouvons nous limiter aux formules simplifiées en utilisant la proposition suivante :

Proposition 3.27 Deux formules E et F sont linéairement isomorphes si et seulement si $s(E)$ et $s(F)$ sont linéairement isomorphes.

Démonstration :

Il suffit de montrer d'abord que $1 \otimes E$ est isomorphe à E, et que $\bot \parr E$ est isomorphe à E, et d'utiliser le fait que la notion d'isomorphisme linéaire est une congruence. ■

3.6.3 Complétude avec constantes

Pour des formules simplifiées, la preuve de complétude est très similaire à celle du cas sans constante. Nous avons juste à étendre la définition de réseaux de preuves simple bipartite :

Définition 3.28 (réseau simple bipartite) Un réseau de preuve simple sans coupure est dit ***bipartite*** s'il a exactement deux conclusions E et F, et est constitué de
- $T(E)$, $T(F)$,
- un ensemble de liens axiomes reliant des atomes de E à des atomes de F, mais pas d'atomes de E entre eux, ni d'atomes de F entre eux,
- et de boîtes avec au moins une conclusion dans $T(E)$ et au moins une dans $T(F)$.

Proposition 3.29 Soient E et F deux formules simplifiées. Soit S un réseau simple sans coupure de conclusions $E^\perp$ et F, et S' un réseau simple sans coupure de conclusions $F^\perp$ et E. Toutes les boîtes de S et S' ne contiennent que la constante 1.

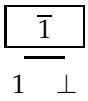
Démonstration :

Dans cette preuve, notons n_1^X le nombre de 1 et $n_\perp^X$ le nombre de $\perp$ dans le réseau de preuve X.
- Le cas dans lequel E ou F est une constante est trivial.
- Sinon, toutes les occurrences de la constante 1 dans les deux réseaux sont un des deux sous-termes d'un lien $\parr$. La seule façon d'avoir ces réseaux connectés (ce qui est nécessaire pour faire un réseau de preuve) est de mettre le 1 dans une boîte. Chacune de ces boîtes correspond à un $\perp$ distinct. Nous pouvons en déduire que $n_\perp^S \geq n_1^S$, et $n_\perp^{S'} \geq n_1^{S'}$.
- Mais $n_\perp^S = n_1^{S'}$, $n_\perp^{S'} = n_1^S$. Donc $n_\perp^S = n_1^S$. Ainsi il n'y a aucune boîte contenant autre chose que 1 dans S. C'est la même chose pour S'.

■

Il est facile de montrer que les boîtes de cette forme se comportent comme des liens axiomes lors de l'élimination des coupures. Il suffit de remarquer que le seul cas possible d'élimination des coupures faisant intervenir des boîtes dans un tel réseau est le suivant :

qui se réduit en



Les constantes peuvent donc être vues dans ce cas exactement comme des atomes, et nous pouvons montrer, en procédant exactement comme dans le cas sans constantes, que

Proposition 3.30 Si E et F sont linéairement isomorphes, alors $AC(\otimes, \parr) \vdash s(E) = s(F)$.

Et donc

Théorème 3.31 (Complétude avec constantes) Si E et F sont linéairement isomorphes, alors $ACI(\otimes, \parr) \vdash E = F$.

3.7 Conclusions

Nous avons montré que dans MLL, les seuls isomorphismes de formules sont donnés par les axiomes les plus intuitifs, à savoir l'associativité et la commutativité, et seulement ceux-ci. Au

delà de l'intérêt du résultat lui-même, cela donne une interprétation symétrique très élégante des isomorphismes linéaires dans le λ-calcul linéaire, et justifie le fait, observé plusieurs fois par le passé, que la curryfication en programmation fonctionnelle correspond à un sorte d'associativité (cela se produit dans l'implantation de machines abstraites, ou bien dans le codage de λ-termes dans les CDS de Berry [18]). Notre résultat confirme une fois encore que la logique linéaire est une paire de lunettes à travers lesquelles les propriétés fondamentales du calcul fonctionnel apparaissent simplifiées et plus symétriques. Il est aussi intéressant de remarquer que les réseaux étaient vraiment l'outil qui manquait pour comprendre la linéarité. Pour prouver le résultat, nous avons utilisé essentiellement les propriétés topologiques des réseaux de preuves de la logique linéaire, ce qui a simplifié grandement notre tâche (par exemple la réduction aux formules non-ambiguës lorsque l'on travaille directement sur des λ-termes est beaucoup plus complexe qu'ici, où les liens axiomes nous permettent de donner une preuve élégante).

Il serait intéressant d'essayer d'étendre ce résultat à MALL. Cependant la méthode utilisée ici se heurte au fait que les réseaux de preuves pour MALL connus à ce jour gardent tous une trace de séquentialité, que ce soient les réseaux à poids de Jean-Yves Girard [51], ou les réseaux à boîtes de Lorenzo Tortora [80].

Olivier Laurent a réussi à résoudre le problème des isomorphismes dans la logique linéaire polarisée LLP [63] et le $\lambda\mu$-calcul [69], en utilisant une sémantique des jeux [62]. Cette fois encore, les isomorphismes sont remarquablement simples. Il a montré que ce sont exactement les isomorphismes de types que pour le λ-calcul si l'on se restreint aux types produits et flèche. Cependant nous ne pouvons pas en déduire le résultat pour le λ-calcul simplement typé avec sommes et zéro car la disjonction de LLP n'est pas un co-produit (catégories de contrôle et non biCCC).

Chapitre 4

Isomorphismes pour le λ-calcul avec type somme et type vide

Résumé

Les égalités arithmétiques apprises au lycée sont-elles complètes pour montrer toutes les égalités valides pour les entiers naturels ? La réponse à cette question est positive pour le langage des expressions arithmétiques contenant la constante 1 *et les opérations produit et exponentiation. La théorie axiomatique correspondante caractérise également les isomorphismes en* λ*-calcul typé, où la constante* 1*, le produit et l'exponentiation sont remplacés respectivement par les types unité, produit et flèche. Ce chapitre étudie les isomorphismes en* λ*-calcul avec type vide et type somme de ce point de vue. Il clôt notamment un problème ouvert en montrant que la théorie des isomorphismes de types en présence de produit, flèche, somme (et avec ou sans le type unité) n'est pas finiment axiomatisable. Nous observerons aussi que pour les théories de types avec flèche, somme et type vide, la correspondance entre isomorphismes et égalités arithmétiques n'est plus vraie en général, mais qu'elle existe toujours dans quelques cas particuliers.*

NOUS ALLONS étudier dans ce chapitre les isomorphismes en λ-calcul typé avec type somme et type vide, en reliant comme d'habitude ce cas à la théorie des langages de programmation, à la théorie des catégories et à la logique.

Nous allons d'abord énoncer dans la section 4.1 un résultat permettant de mettre en relation les différentes théories de types considérées, qui dit que les foncteurs d'extension entre elles sont pleinement fidèles. On pourra en déduire que les théories équationnelles des isomorphismes de types sont conservatives.

Ensuite nous allons montrer que la théorie des isomorphismes de types en présence des types produit, flèche et somme n'est pas finiment axiomatisable. Pour cela, nous allons relier le pro-

blème au problème des *égalités du lycée* de Tarski : pour le cas des types utilisant les constructeurs produit, flèche et unité, deux types sont isomorphes si et seulement si les expressions arithmétiques associées sont égales dans le modèle standard des entiers naturels. (On obtient cette traduction en interprétant simplement le type unité par le nombre 1, le produit par la multiplication, et la flèche par l'exponentiation). Dans ce cas, les isomorphismes de types (et les égalités numériques) sont finiment axiomatisables et décidables ; et c'est donc vrai également pour les isomorphismes des CCC.

Nous allons nous poser la question de savoir si une telle correspondance est limitée ou non à ce cas, ou si au contraire elle peut être étendue à des types plus problématiques contenant les constructeurs somme et vide. D'un point de vue pratique, il est intéressant de savoir si les isomorphismes de types en présence de sommes sont finiment axiomatisables — ce qui serait pratique pour implanter des procédures de décision dans des outils de recherche dans des bibliothèques comme ceux présentés dans [72, 36]. La section 4.2 montre que ce n'est pas le cas, et qu'en cela, encore une fois, les isomorphismes se comportent comme les entiers naturels.

On pourrait donc espérer que le parallèle se poursuive plus loin en établissant que les isomorphismes de types sont exactement les égalités entre entiers. Nous allons montrer dans la section 4.2.3 que c'est vrai dans certains cas particuliers mais que dans le cas général cette coïncidence n'est plus observée.

4.1 Mise en relation des différentes théories

Dans cette section, je vais énoncer un résultat dû à Marcelo Fiore qui permet de mettre en relation les différentes théories de types qui nous intéressent dans la suite. Des détails de la preuve peuvent être trouvés dans [14].

Pour cela, notons
$\mathfrak{L}[\mathfrak{C}]$ la CCC libre engendrée par $\mathfrak{C}$
$\mathfrak{L}_0[\mathfrak{C}]$ la CCC libre avec objet initial engendrée par $\mathfrak{C}$
$\mathfrak{L}_+[\mathfrak{C}]$ la CCC libre avec co-produits binaires engendrée par $\mathfrak{C}$
$\mathfrak{L}_{0,+}[\mathfrak{C}]$ la biCCC libre engendrée par $\mathfrak{C}$

Le résultat qui nous intéresse est le suivant :

Théorème 4.1 Les foncteurs préservant la structure suivants :

$$\begin{array}{ccc} \mathfrak{L}[\mathfrak{C}] & \longrightarrow & \mathfrak{L}_0[\mathfrak{C}] \\ \downarrow & & \downarrow \\ \mathfrak{L}_+[\mathfrak{C}] & \longrightarrow & \mathfrak{L}_{0,+}[\mathfrak{C}] \end{array} \tag{4.1}$$

étendant les inclusions universelles $\mathfrak{C} \to \mathfrak{L}[\mathfrak{C}]$, $\mathfrak{C} \to \mathfrak{L}_0[\mathfrak{C}]$, $\mathfrak{C} \to \mathfrak{L}_+[\mathfrak{C}]$, et $\mathfrak{C} \to \mathfrak{L}_{0,+}[\mathfrak{C}]$ sont pleinement fidèles.

La même méthode de preuve fonctionne pour tous les foncteurs et d'autres variations, ce qui indique qu'il existe une théorie générale sous-jacente au théorème. Nous avons par exemple le résultat suivant :

Théorème 4.2 Les foncteurs canoniques préservant la structure allant de la catégorie libre distributive sur une petite catégorie vers la biCCC libre sur la même petite catégorie est pleinement fidèle.

Une conséquence de ces théorèmes est la conservativité des différentes théories de types. Donc, non seulement les isomorphismes de types vrais dans une théorie des types sont également vrais dans ses extensions, mais aussi les types non-isomorphes dans la théorie des types originale restent non-isomorphes dans la théorie étendue.

4.2 Le problème des égalités du lycée et les isomorphismes de types

La figure 4.1 rappelle l'axiomatisation des isomorphismes dans les catégories cartésiennes fermées. Si nous lisons ces isomorphismes en voyant le nombre 1 à la place de l'objet terminal 1, des

$$
\left.
\begin{array}{lrcl}
1. & A \times B & \cong & B \times A \\
2. & A \times (B \times C) & \cong & (A \times B) \times C \\
3. & C^{A \times B} & \cong & (C^B)^A \\
4. & (B \times C)^A & \cong & B^A \times C^A \\
5. & A \times 1 & \cong & A \\
6. & A^1 & \cong & A \\
7. & 1^A & \cong & 1
\end{array}
\right\} Th_{CCC}
$$

FIG. 4.1 – Théorie des isomorphismes vrais dans les CCC.

variables numériques à la place des variables de types, le produit $A \cdot B$ de deux expressions numériques au lieu du type $A \times B$, et enfin l'exposant B^A à la place du type $A \to B$, les isomorphismes ci-dessus deviennent des égalités numériques bien connues apprises au lycée. Cela est dû au fait que les entiers naturels peuvent être utilisés pour construire une catégorie bi-cartésienne fermée : à chaque entier n on associe un ensemble fini à n éléments, et les morphismes entre deux objets m et n sont les n^m applications (au sens de la théorie de ensembles) entre ces ensembles finis[1]. Dans cette catégorie, l'objet terminal est le singleton associé à 1, et le produit et l'exponentielle correspondent respectivement au produit d'entiers et à l'exponentiation. En utilisant cela, il est facile de montrer que deux objets sont isomorphes *dans la catégorie des ensembles finis* si et seulement si ils sont égaux en tant qu'expressions numériques. Cela signifie que le problème de la caractérisation des types isomorphes *dans la catégorie des ensembles finis* peut être rapporté à celui de trouver toutes les équations numériques valides entre des expressions construites à l'aide du produit, de la puissance, de la constante 1 et de variables.

C'est un cas particulier du fameux ***problème des égalités du lycée*** de Tarski [▷ *Tarski's high school algebra problem*] que nous allons rappeler ici, pour lequel la solution est exactement l'ensemble des équations de la figure 4.1 donnant les isomorphismes des CCC[2].

[1] On peut utiliser la catégorie des ***ordinaux finis***.

[2] Cette analogie remarquable entre isomorphismes et égalités en théorie des nombres fonctionne bien également pour certaines variantes naturelles (les isomorphismes avec flèche seulement coïncident avec les égalités numériques entre expressions utilisant uniquement l'exponentiation, etc.), du moment que l'on limite le langage à une combinaison de l'exponentiation, du produit et du 1.

Le problème des égalités du lycée En 1969, Alfred Tarski [41] a posé la question suivante : la théorie équationnelle $\mathcal{E}$ des identités arithmétiques usuelles apprises au lycée (voir figure 4.2) est-elle complète pour le modèle standard $\langle \mathbb{N}^*, 1, \cdot, \uparrow, + \rangle$ des entiers naturels strictement positifs ? Autrement dit peut-on avec elles-seules montrer toute égalité arithmétique[3] ? Il a conjecturé qu'elles l'étaient[4], mais n'a pas réussi à montrer ce résultat. Son étudiant Charles Martin [67] a montré que l'identité $(\mathcal{E}_3)$ à elle-seule est complète pour le modèle standard $\langle \mathbb{N}^*, \uparrow \rangle$ des entiers naturels positifs avec exponentiation, et que la théorie constituée par les identités $(\mathcal{E}_1)$, $(\mathcal{E}_2)$, $(\mathcal{E}_3)$, et $(\mathcal{E}_4)$ est complète pour le modèle standard $\langle \mathbb{N}^*, \cdot, \uparrow \rangle$ des entiers naturels avec multiplication et exponentiation. De plus, il a exhibé une l'égalité

$$(x^u + x^u)^v \cdot (y^v + y^v)^u = (x^v + x^v)^u \cdot (y^u + y^u)^v$$

qui n'est *pas* prouvable dans $\mathcal{E}$ si l'on se restreint au langage sans la constante 1... Cela montre déjà qu'en présence de la somme, tout n'est pas si simple[5]... Cependant la question n'était pas complètement résolue par ce contre-exemple, car c'est un contre-exemple uniquement pour le langage ne comportant pas la constante 1, que Tarski considérait clairement nécessaire, même s'il ne la mentionnait pas explicitement dans sa conjecture. En présence de la constante 1, les équations $(\mathcal{E}_5)$, $(\mathcal{E}_7)$, et $(\mathcal{E}_6)$ apparaissent, et nous permettent de montrer facilement l'identité de Martin[6]. Ce problème a attiré l'attention de nombreux autres mathématiciens, comme Leon Henkin, qui s'est attardé sur les égalités valides dans $\langle \mathbb{N}, 0, + \rangle$, et a montré la complétude des axiomes usuels (commutativité, associativité de la somme et axiome du zéro), et a donné une intéressante présentation du sujet [56].

Alex J. Wilkie [83, 84] a été le premier à établir que la conjecture de Tarski était fausse. Par une analyse en théorie de la preuve, il a montré que l'égalité

$$(A^x + B^x)^y \cdot (C^y + D^y)^x = (A^y + B^y)^x \cdot (C^x + D^x)^y$$

$$\begin{array}{lll}
(\mathcal{E}_1) & x \cdot y = y \cdot x & (\mathcal{E}_2) \quad (x \cdot y) \cdot z = x \cdot (y \cdot z) \\
(\mathcal{E}_3) & x^{y \cdot z} = (x^y)^z & (\mathcal{E}_4) \quad (x \cdot y)^z = x^z \cdot y^z \\
(\mathcal{E}_5) & 1 \cdot x = x & \\
(\mathcal{E}_6) & 1^x = 1 & (\mathcal{E}_7) \quad x^1 = x \\
(\mathcal{E}_8) & x + y = y + x & \\
(\mathcal{E}_9) & (x + y) + z = x + (y + z) & \\
(\mathcal{E}_{10}) & x \cdot (y + z) = x \cdot y + x \cdot z & \\
(\mathcal{E}_{11}) & x^{(y+z)} = x^y \cdot x^z &
\end{array}$$

FIG. 4.2 – Égalités du lycée.

[3] Nous avons vu au chapitre 1 que les quatre règles utilisant l'addition sont aussi des isomorphismes dans les biCCC, tout comme les identités usuelles utilisant le zéro.

[4] En fait, il a fait une conjecture plus forte, à savoir que $\mathcal{E}$ est complète pour $\langle \mathbb{N}^*, Ack(n, \_, \_) \rangle$, les entiers naturels équipés d'une famille d'opérateurs binaires plus généraux $Ack(n, \_, \_)$ (proches de la célèbre fonction d'Ackermann) qui étendent la somme $+$, le produit $\cdot$ et l'exponentiation $\uparrow$. Dans la définition de Tarski , $Ack(0, \_, \_)$ est la somme, $Ack(1, \_, \_)$ la multiplication, et $Ack(2, \_, \_)$ est l'exponentiation (pour les autres cas, voir par exemple [74]).

[5] Il a aussi montré qu'il n'y a que des équations triviales pour $\langle \mathbb{N}^*, Ack(n, \_, \_) \rangle$ si $n > 2$.

[6] Il suffit d'écrire $(x^u + x^u) = (1 \cdot x^u + 1 \cdot x^u) = (x^u \cdot 1 + x^u \cdot 1) = x^u \cdot (1 + 1)$, *etc.*

où

$$\begin{array}{rclcrcl} A & = & 1+x\,, & & B & = & 1+x+x^2 \\ C & = & 1+x^3\,, & & D & = & 1+x^2+x^4 \end{array}$$

n'est pas prouvable dans $\mathcal{E}$.

Gurevič a construit plus tard un contre-modèle *ad hoc* [54] permettant de le prouver, et surtout, il a montré qu'il n'y a pas d'axiomatisation finie des équations valides dans le modèle standard $\langle \mathbb{N}^*, 1, \cdot, \uparrow, + \rangle$ des entiers naturels avec 1, multiplication, exponentiation, et addition [55]. Pour cela, il a donné une suite infinie d'équations telles que, pour tout ensemble fini d'axiomes, on peut montrer qu'une des équations ne s'en déduit pas. Les identités de Gurevič, qui généralisent celles de Wilkie, sont les suivantes :

$$(A^x + {B_n}^x)^{2^x} \cdot\ ({C_n}^{2^x} + {D_n}^{2^x})^x = \ (A^{2^x} + {B_n}^{2^x})^x \cdot\ ({C_n}^x + {D_n}^x)^{2^x} \quad (n \geq 3 \text{ impair}) \tag{4.2}$$

où

$$\begin{array}{rcl} A & = & 1+x \\ B_n & = & 1+x+\cdots+x^{n-1} \ = \ \sum_{i=0}^{n-1}\ x^i \\ C_n & = & 1+x^n \\ D_n & = & 1+x^2+\cdots+x^{2(n-1)} \ = \ \sum_{i=0}^{n-1}\ x^{2i} \end{array}$$

Cependant, l'égalité dans ces structures, même non-finiment axiomatisables, a été montrée décidable [66, 54][7].

Comme souvent en théorie des nombres, ces derniers résultats utilisent des outils largement plus complexes que de simples raisonnements arithmétiques, comme dans le cas de [57], où la théorie de Nevanlinna est utilisée pour identifier une sous-classe des expressions numériques pour lesquelles les axiomes usuels de $+$, $\cdot$, $\uparrow$ et 1 sont complets.

Isomorphismes de types Les équations de $\mathcal{E}$, ainsi que les deux suivantes

$$(\mathcal{D}_1) \quad x \cdot 0 = 0 \qquad (\mathcal{D}_2) \quad x + 0 = x$$

ont une interprétation combinatoire très claire, qui est rendue évidente lorsqu'on les interprète comme des isomorphismes de la catégorie des ensembles finis $\mathfrak{Ens}_{\mathfrak{f}}$, ou en fait dans n'importe quelle biCCC, à l'aide de la traduction évidente donnée ici :

$$\overline{x} = x \text{ (x variable)}\,, \qquad \overline{1} = \mathfrak{1}$$

$$\overline{e_1 \cdot e_2} = \overline{e_1} \times \overline{e_2}\,, \qquad \overline{e_1{}^{e_2}} = \overline{e_1}^{\overline{e_2}} \quad \text{(que l'on notera parfois } \overline{e_2} \to \overline{e_1}\text{)}$$

$$\overline{0} = \mathfrak{o}\,, \qquad \overline{e_1 + e_2} = \overline{e_1} + \overline{e_2}$$

Les isomorphismes réalisant les traductions des équations de $\mathcal{E}$ sont donnés dans le cadre des catégories au chapitre 1. Notons que l'équation

$$x \cdot 0^x = 0$$

qui correspond à l'isomorphisme de types

$$\theta \times (\theta \to 0) \cong 0$$

[7] Pour les curieux, voici l'idée de la preuve : à partir de la taille de l'équation à vérifier, il est possible de calculer une borne supérieure ; si les deux membres de l'équation coïncident pour toutes les valeurs de variables jusqu'à cette borne, alors elles coïncident partout.

n'a pas de signification combinatoire évidente, bien qu'elle corresponde à la tautologie intuitionniste $(p \wedge \neg p) \Leftrightarrow \bot$.

Pour des expressions arithmétiques e_1 et e_2 dans le langage avec la constante 1, les opérations $\cdot$, $\uparrow$, $+$, et les inconnues dans un ensemble U, nous avons donc la chaîne d'implications suivante :

$$\begin{array}{rcl} \mathcal{E} \vdash e_1 = e_2 & \Longrightarrow & \overline{e_1} \cong \overline{e_2} \text{ in } \mathfrak{L}_+[U] \\ & \Longrightarrow & \mathfrak{Ens}_{\mathfrak{f}} \models \overline{e_1} = \overline{e_2} \\ & \Longrightarrow & \mathbb{N}^* \models e_1 = e_2 \end{array} \tag{4.3}$$

où, pour des types τ_1 et τ_2 et une catégorie $\mathfrak{S}$, on note $\mathfrak{S} \models \tau_1 = \tau_2$ lorsque l'identité $\tau_1 = \tau_2$ est valide en tant qu'isomorphisme de $\mathfrak{S}$; c'est-à-dire si pour toutes les interprétations $\mathcal{I}(.)$ des types de base dans $\mathfrak{S}$, nous avons $\mathcal{I}[\![\tau_1]\!] \cong \mathcal{I}[\![\tau_2]\!]$ dans $\mathfrak{S}$. (Notons que les expressions $\tau_1 \cong \tau_2$ dans $\mathfrak{L}_{o,+}[U]$ et $\mathfrak{L}_{o,+}[U] \models \tau_1 = \tau_2$ correspondent aux isomorphismes de type dans la théorie équationnelle du λ-calcul typé avec type vide et type somme.) Il est facile d'établir la dernière implication de (4.3) en observant que des ensembles finis sont isomorphes si et seulement s'ils ont la même cardinalité, et que les constructeurs de types sur les ensembles finis coïncident avec l'arithmétique cardinale.

Sergei Soloviev, ainsi que Kim Bruce, Roberto Di Cosmo et Giuseppe Longo ont montré que les implications de (4.3) sont des équivalences dans le cas des catégories cartésiennes fermées [77, 19]. Le reste du chapitre est consacré à étudier dans quelle mesure ces implications peuvent être inversées dans les théories des types avec type vide et/ou type somme.

4.2.1 Types produit et somme

Considérons d'abord le cas des catégories distributives, qui correspondent à la théorie des types avec type unité, type vide, produit, et somme.

Soit $\mathcal{D}$ la théorie équationnelle constituée des équations $(\mathcal{E}_1)$, $(\mathcal{E}_2)$, $(\mathcal{E}_5)$, $(\mathcal{E}_8)$, $(\mathcal{E}_9)$, $(\mathcal{E}_{10})$ et $(\mathcal{D}_1)$, $(\mathcal{D}_2)$. Nous avons le résultat suivant qui peut être établi en remarquant que $\mathcal{D}$ permet de réduire l'égalité des expressions arithmétiques à l'égalité de polynômes, et que l'égalité de polynômes est juste l'identité dans $\mathbb{N}$.

Proposition 4.3 ($\mathcal{D}$ est correcte et complète) Pour des expressions arithmétiques e_1 et e_2 dans le langage utilisant 1, 0, $\cdot$, et $+$ et avec des inconnues dans un ensemble U, les affirmations suivantes sont équivalentes :

1. $\mathcal{D} \vdash e_1 = e_2$
2. $\overline{e_1} \cong \overline{e_2}$ in $\mathcal{D}[U]$
3. $\mathfrak{Ens}_{\mathfrak{f}} \models \overline{e_1} = \overline{e_2}$
4. $\mathbb{N} \models e_1 = e_2$

où $\mathcal{D}[U]$ dénote la catégorie distributive libre engendrée sur l'ensemble U, et où la dernière affirmation exprime la validité dans le modèle standard des entiers naturels.

La chaîne d'implications $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (4)$ est triviale, et l'implication $(4) \Rightarrow (1)$ correspond à la complétude de $\mathcal{D}$ pour le modèle standard des entiers naturels. Ce résultat est connu (voir, par exemple, [21]) et nous le rappelons ci-dessous.

Lemme 4.4 La théorie équationnelle $\mathcal{D}$ est complète pour le modèle standard $\langle N_0, 1, 0, \cdot, + \rangle$ des entiers naturels.

Démonstration :

Toute expression dans le langage des variables, 1, 0, $\cdot$, et $+$ peut être réécrite, en utilisant les équations de $\mathcal{D}$, en une forme polynômiale canonique, que l'on peut rendre unique.

Maintenant, $N_0 \models e_1 = e_2$ si et seulement si, pour les polynômes canoniques p_1 et p_2 correspondant respectivement à e_1 et e_2, $N_0 \models p_1 = p_2$. Ceci revient à dire que p_1 et p_2 sont le même polynôme (*i.e.*, égal syntaxiquement) comme le polynôme $p_1 - p_2$ à coefficients rationnels a une infinité de zéros et donc est nul.

On en déduit le résultat de la manière suivante : si l'égalité $e_1 = e_2$ est vraie dans le modèle standard, alors, en utilisant $\mathcal{D}$, e_1 peut être transformé en sa forme canonique polynomiale, qui peut elle-même être transformée en e_2. ■

Notons que l'argument de la preuve ci-dessus montre aussi que $\mathcal{D}$ est décidable (voir aussi Gil [49]). Comme corollaire, nous avons la propriété de simplification suivante :

Corollaire 4.5 Pour tout type non vide τ (à savoir $\tau \not\cong 0$) de $\mathcal{D}[T]$, nous avons

$$\tau_1 \times \tau \cong \tau_2 \times \tau \text{ dans } \mathcal{D}[T] \implies \tau_1 \cong \tau_2 \text{ dans } \mathcal{D}[T]$$

pour tout couple $(\tau_1,\ \tau_2)$ de types de $\mathcal{D}[T]$.

Il est intéressant de remarquer qu'en présence d'exponentielles, la propriété de simplification ci-dessus n'est pas toujours vraie. En effet, pour tout type τ de $\mathcal{L}_{0,+}[T]$, nous avons l'isomorphisme

$$(\tau \to 0) \times \tau \cong 0 \times \tau \text{ dans } \mathcal{L}_{0,+}[T]\ ,$$

mais la simplification par τ ne donne pas toujours un isomorphisme (par exemple, $\theta \to 0 \not\cong 0$ pour tout $\theta \in T$).

Lemme 4.6 Toutes les égalités de $\mathcal{D}$ sont des isomorphismes valides dans les catégories distributives.

Voir la démonstration dans le chapitre 1

Théorème 4.7 $\mathcal{D}$ est complète pour les isomorphismes des catégories distributives.

Démonstration :

Toutes les équations de $\mathcal{D}$ sont valides dans toutes les catégories distributives. Remarquons que la catégorie des ensembles finis est une catégorie distributive particulière pour laquelle les seuls isomorphismes valides sont ceux qui sont dérivables de $\mathcal{D}$, et donc aucun autre isomorphisme ne peut être valide dans toutes les catégories distributives.

Maintenant, comme rappelé dans la section 4.2, dans la catégorie des ensembles finis, deux objets sont isomorphes si et seulement si ils ont le même cardinal, et le calcul des cardinalités est le même dans ce cadre que les règles de formation des objets ($card(AB) = card(A)card(B)$, $card(\emptyset) = 0$, $card(1) = 1$, *etc.*). Donc deux objets A et B sont isomorphes si et seulement si, lus comme des expressions numériques, $A = B$ est vrai dans $\langle \mathbb{N}, +, \cdot, 1, 0 \rangle$; ce qui, d'après le lemme 4.4, se produit uniquement si $A = B$ est dérivable de $\mathcal{D}$. ■

4.2.2 Types produit, flèche, et somme

Dans le but de comprendre les isomorphismes de types en présence de types produit, flèche et somme, il est naturel de se demander si les équations de Gurevič sont aussi des isomorphismes de types. Comme d'habitude, on peut d'abord vérifier que ce sont bien des équivalences, en utilisant un logiciel de preuve en logique intuitionniste, comme *Strip* (Galmiche-Larchey [61]).

Ensuite nous allons continuer en présentant deux manières de prouver les identités (4.2) pour les entiers naturels, dues respectivement à Wilkie et Gurevič.

► méthode de Wilkie Utiliser le fait que $C_n = A \cdot E_n$ et $D_n = B_n \cdot E_n$ pour $E_n = 1 - x + \cdots + x^{n-1} = \sum_{i=0}^{n-1} (-1)^i x^i$.

► méthode de Gurevič Multiplier le côté gauche de (4.2) par $({D_n}^x)^{2^x}$ et utiliser l'égalité $A \cdot D_n = B_n \cdot C_n$ deux fois pour montrer que l'on obtient le côté droit de (4.2) multiplié par $({D_n}^{2^x})^x$; conclure (4.2) en simplifiant par $({D_n}^x)^{2^x} = ({D_n}^{2^x})^x$:

$$\begin{aligned} {D_n}^{2^x x} \cdot (A^{2^x} + {B_n}^{2^x})^x \cdot ({C_n}^x + {D_n}^x)^{2^x} &= (A^{2^x} {D_n}^{2^x} + {B_n}^{2^x} {D_n}^{2^x})^x \cdot ({C_n}^x + {D_n}^x)^{2^x} \\ &= ({B_n}^{2^x} {C_n}^{2^x} + {B_n}^{2^x} {D_n}^{2^x})^x \cdot ({C_n}^x + {D_n}^x)^{2^x} \\ &= {B_n}^{2^x x} ({C_n}^{2^x} + {D_n}^{2^x})^x \cdot ({C_n}^x + {D_n}^x)^{2^x} \end{aligned}$$

De la même façon,

$${D_n}^{2^x x} \cdot (A^x + {B_n}^x)^{2^x} \cdot ({C_n}^{2^x} + {D_n}^{2^x})^x = {B_n}^{2^x x} ({C_n}^{2^x} + {D_n}^{2^x})^x \cdot ({C_n}^x + {D_n}^x)^{2^x}$$

Donc, ${D_n}^{2^x x} \cdot (A^{2^x} + {B_n}^{2^x})^x \cdot ({C_n}^x + {D_n}^x)^{2^x} = {D_n}^{2^x x} \cdot (A^x + {B_n}^x)^{2^x} \cdot ({C_n}^{2^x} + {D_n}^{2^x})^x$.
Ensuite, en simplifiant par ${D_n}^{2^x x}$, on obtient le résultat cherché.

Comme les deux méthodes exposées ci-dessus utilisent, respectivement, des entiers négatifs (qui n'ont pas de correspondant en théorie des types) et la simplification (qui n'est pas correcte en théorie des types), nous avons d'abord conjecturé que les identités de Gurevič ne sont pas des isomorphismes. Nous avons donc commencé par chercher à prouver qu'aucun terme du type correspondant aux équations de Gurevič n'est un isomorphisme, en étudiant avec attention les formes normales du λ-calcul avec type vide et type somme. Pour cela, nous avons fait une étude précise de ces formes normales qui sera présentée dans le chapitre 5 (et Balat-Di Cosmo-Fiore [13]).

Les identités de Wilkie-Gurevič généralisées Pour simplifier l'étude des formes normales entre les types induits par les expressions de (4.2), nous avons introduit les identités de Wilkie-Gurevič généralisées suivantes, qui n'ont pas de constante.

$$\begin{aligned} &(A^u + {B_n}^u)^v \cdot ({C_n}^v + {D_n}^v)^u \\ &\quad = (A^v + {B_n}^v)^u \cdot ({C_n}^u + {D_n}^u)^v \end{aligned} \qquad (n \geq 3 \text{ impair}) \tag{4.4}$$

où

$$\begin{aligned} A &= y + x \\ B_n &= y^{n-1} + y^{n-2}x + \cdots + x^{n-1} \\ &= \textstyle\sum_{i=0}^{n-1} y^{n-(i+1)} x^i \\ C_n &= y^n + x^n \\ D_n &= y^{2(n-1)} + y^{2(n-2)}x^2 + \cdots + x^{2(n-1)} \\ &= \textstyle\sum_{i=0}^{n-1} y^{2(n-(i+1))} x^{2i} \end{aligned} \tag{4.5}$$

Remarquons qu'en remplaçant y par 1, u par x, et v par 2^x dans (4.4) on obtient les équations (4.2). Le résultat de non-fini-axiomatisabilité ne dépend donc pas de la présence de constantes dans le langage, et nous pouvons donc énoncer le résultat suivant :

Proposition 4.8 La théorie équationnelle de $\langle \mathbb{N}, \cdot, \uparrow, + \rangle$ n'est pas finiment axiomatisable.

Les identités de Wilkie-Gurevič sont des isomorphismes de types En analysant les formes normales des termes dont le type correspond aux identités de Wilkie-Gurevič généralisées (4.4) dans le cas $n = 3$ nous avons fini, contre toute attente, par trouver un isomorphisme. Nous allons présenter ci-après la construction générale.

Lemme 4.9 Pour des types $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$, et des termes clos mutuellement inverses $\varphi : \mathcal{A} \times \mathcal{D} \to \mathcal{C} \times \mathcal{B}$ et $\phi : \mathcal{C} \times \mathcal{B} \to \mathcal{A} \times \mathcal{D}$ du λ-calcul simplement typé avec type somme et type vide, les termes suivants (où l'on utilise la notation τ^σ pour $\sigma \to \tau$) définis figure 4.3 sont mutuellement inverses :

$$t_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{U},\mathcal{V}}[\varphi,\phi] \ : \ (\mathcal{A}^{\mathcal{U}} + \mathcal{B}^{\mathcal{U}})^{\mathcal{V}} \times (\mathcal{C}^{\mathcal{V}} + \mathcal{D}^{\mathcal{V}})^{\mathcal{U}} \to (\mathcal{A}^{\mathcal{V}} + \mathcal{B}^{\mathcal{V}})^{\mathcal{U}} \times (\mathcal{C}^{\mathcal{U}} + \mathcal{D}^{\mathcal{U}})^{\mathcal{V}}$$

$$t_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{V},\mathcal{U}}[\varphi,\phi] \ : \ (\mathcal{A}^{\mathcal{V}} + \mathcal{B}^{\mathcal{V}})^{\mathcal{U}} \times (\mathcal{C}^{\mathcal{U}} + \mathcal{D}^{\mathcal{U}})^{\mathcal{V}} \to (\mathcal{A}^{\mathcal{U}} + \mathcal{B}^{\mathcal{U}})^{\mathcal{V}} \times (\mathcal{C}^{\mathcal{V}} + \mathcal{D}^{\mathcal{V}})^{\mathcal{U}}$$

Démonstration :
Voir section 4.3. ∎

$$t_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{U},\mathcal{V}}[\varphi,\phi] \stackrel{\text{def}}{=} \lambda h : (\mathcal{A}^{\mathcal{U}} + \mathcal{B}^{\mathcal{U}})^{\mathcal{V}} \times (\mathcal{C}^{\mathcal{V}} + \mathcal{D}^{\mathcal{V}})^{\mathcal{U}}.$$
$$(\ p_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{U},\mathcal{V}}[\varphi,\phi,proj_1 \ h, proj_2 \ h] \ , \ p_{\mathcal{C},\mathcal{D},\mathcal{A},\mathcal{B},\mathcal{V},\mathcal{U}}[\phi,\varphi,proj_2 \ h, proj_1 \ h] \ )$$

où

$$p_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{U},\mathcal{V}}[\varphi,\phi,f,g] \stackrel{\text{def}}{=} \lambda u : \mathcal{U}.\, case(g(u),$$
$$g_1 : \mathcal{C}^{\mathcal{V}}.\, in_1 \ (\lambda v : \mathcal{V}.\, case(f(v),$$
$$f_1 : \mathcal{A}^{\mathcal{U}}.\, f_1(u),$$
$$f_2 : \mathcal{B}^{\mathcal{U}}.\, proj_1 \ (\phi(g_1(v), f_2(u)) \ )$$
$$)$$
$$),$$
$$g_2 : \mathcal{D}^{\mathcal{V}}.\, in_2 \ (\lambda v : \mathcal{V}.\, case(f(v),$$
$$f_1 : \mathcal{A}^{\mathcal{U}}.\, proj_2 \ (\varphi(f_1(u), g_2(v) \ ),$$
$$f_2 : \mathcal{B}^{\mathcal{U}}.\, f_2(u)$$
$$)$$
$$)$$
$$)$$

FIG. 4.3 – Définition de $t_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{U},\mathcal{V}}[\varphi,\phi]$
de type $(\mathcal{A}^{\mathcal{U}} + \mathcal{B}^{\mathcal{U}})^{\mathcal{V}} \times (\mathcal{C}^{\mathcal{V}} + \mathcal{D}^{\mathcal{V}})^{\mathcal{U}} \to (\mathcal{A}^{\mathcal{V}} + \mathcal{B}^{\mathcal{V}})^{\mathcal{U}} \times (\mathcal{C}^{\mathcal{U}} + \mathcal{D}^{\mathcal{U}})^{\mathcal{V}}$.

Pour les expressions (4.5), l'identité $A \cdot D_n = C_n \cdot B_n$ peut être prouvée à l'aide des axiomes

standards. Il y a donc des termes mutuellement inverses $\varphi_n : \mathcal{A} \times \mathcal{D}_n \to \mathcal{C}_n \times \mathcal{B}_n$ et $\phi_n : \mathcal{C}_n \times \mathcal{B}_n \to \mathcal{A} \times \mathcal{D}_n$, (où $\mathcal{A}$ est mis pour $\overline{A}$, $\mathcal{B}_n$ pour $\overline{B_n}$, $\mathcal{C}_n$ pour $\overline{C_n}$, et $\mathcal{D}_n$ pour $\overline{D_n}$). Il s'ensuit que, $t_{\mathcal{A},\mathcal{B}_n,\mathcal{C}_n,\mathcal{D}_n,\mathcal{U},\mathcal{V}}[\varphi_n, \phi_n]$ est un isomorphisme dont l'inverse est $t_{\mathcal{A},\mathcal{B}_n,\mathcal{C}_n,\mathcal{D}_n,\mathcal{V},\mathcal{U}}[\varphi_n, \phi_n]$. Nous avons donc le résultat suivant :

Corollaire 4.10 La théorie équationnelle des isomorphismes de types dans les catégories cartésiennes fermées avec co-produits binaires n'est pas finiment axiomatisable.

4.2.3 Type vide et type somme

En présence de flèche et des types vide *et* somme, on observe que tous les isomorphismes de la catégorie des ensembles finis ne sont pas des isomorphismes en théorie des types.

Par exemple, en écrivant $\neg\tau$ pour $\tau \to 0$, nous avons

$$\mathfrak{Ens}_{\mathfrak{f}} \models \neg\neg\theta \to \theta = \theta + \neg\theta$$

mais comme la formule $(\neg\neg p \Rightarrow p) \Rightarrow (\neg p \vee p)$ est une tautologie classique qui n'est pas valide intuitionnistiquement, il n'existe pas de terme de type $(\neg\neg\theta \to \theta) \to (\neg\theta + \theta)$. Une autre équation avec cette propriété, dérivée de la précédente, obtenue en remplaçant les types de base par leur négation et en utilisant le fait que $1 \cong \neg\neg\neg\theta \to \neg\theta$, est

$$\mathfrak{Ens}_{\mathfrak{f}} \models 1 = \neg\theta + \neg\neg\theta \quad . \tag{4.6}$$

Donc, en général, $\mathfrak{Ens}_{\mathfrak{f}} \models \tau_1 = \tau_2$ n'implique pas $\tau_1 \cong \tau_2$ dans $\mathfrak{L}_{o,+}[T]$. Cependant, nous avons le résultat suivant :

Proposition 4.11 Pour tout couple (τ_1, τ_2) de types en λ-calcul typé avec type somme et type vide sur un ensemble de types de base T, les affirmations suivantes sont équivalentes :

1. $\neg\tau_1 \cong \neg\tau_2$ dans $\mathfrak{L}_{o,+}[T]$.
2. La formule $\neg\overline{\tau_1} \Leftrightarrow \neg\overline{\tau_2}$ est une tautologie intuitionniste, où
$$\overline{\theta} = \theta\,(\theta \in T)\,, \qquad \overline{1} = \top\,,$$
$$\overline{\tau_1 \times \tau_2} = \overline{\tau_1} \wedge \overline{\tau_2}\,, \qquad \overline{\tau_1 \to \tau_2} = \overline{\tau_1} \Rightarrow \overline{\tau_2}\,,$$
$$\overline{0} = \bot\,, \qquad \overline{\tau_1 + \tau_2} = \overline{\tau_1} \vee \overline{\tau_2}\,.$$
3. Pour toute interprétation $\mathcal{I}(.)$ des types de base dans la catégorie des ensembles finis $\mathfrak{Ens}_{\mathfrak{f}}$, $\mathcal{I}[\![\tau_1]\!] = 0$ iff $\mathcal{I}[\![\tau_2]\!] = 0$.
4. $\mathfrak{Ens}_{\mathfrak{f}} \models \neg\tau_1 = \neg\tau_2$.
5. $N_0 \models 0^{e_1} = 0^{e_2}$, où $\overline{e_1} = \tau_1$ et $\overline{e_2} = \tau_2$.

Démonstration :

Les équivalences (1)⇔(2) et (3)⇔(4)⇔(5), ainsi que l'implication (1) ⇒ (4) sont évidentes.

Pour établir l'implication (3) ⇒ (2) nous allons utiliser la traduction négative (de Gödel-

Gentzen) $\tau^\star$ des types τ définie par

$$\theta^\star = \neg\neg\theta \ (\theta \in T)$$
$$1^\star = 1\,, \quad (\tau_1 \times \tau_2)^\star = {\tau_1}^\star \times {\tau_2}^\star\,,$$
$$(\tau_1 \to \tau_2)^\star = {\tau_1}^\star \to {\tau_2}^\star\,,$$
$$0^\star = 0\,, \quad (\tau_1 + \tau_2)^\star = \neg(\neg{\tau_1}^\star \times \neg{\tau_2}^\star)$$

ainsi que les faits suivants :

(i) Pour tout type τ et toute interprétation $\mathcal{I}(.)$ des types de base dans la catégorie des ensembles finis, nous avons

$$\mathcal{I}[\![\tau^\star]\!] \cong \mathbf{0} \text{ où } \mathcal{I}[\![\tau^\star]\!] \cong \mathbf{1}$$

et

$$\begin{aligned} \mathcal{I}[\![\tau^\star]\!] \cong \mathbf{1} &\iff \overline{\tau^\star} \text{ est une tautologie classique} \\ &\iff \overline{\tau^\star} \text{ est une tautologie intuitionniste} \quad . \end{aligned}$$

(ii) Pour tout type τ, la formule $\overline{\tau^\star} \Leftrightarrow \neg\neg\overline{\tau}$ est une tautologie intuitionniste.

En effet, en supposant (3) on peut montrer grâce à (i) que $(\overline{\tau_1^\star} \to \overline{\tau_2^\star}) = \overline{(\tau_1 \to \tau_2)^\star}$ est une tautologie intuitionniste. Nous en déduisons (grâce à (ii)) que $\neg\neg\overline{\tau_1} \Rightarrow \neg\neg\overline{\tau_2}$ est une tautologie intuitionniste, d'où l'on peut conclure que $\neg\overline{\tau_2} \Rightarrow \neg\overline{\tau_1}$ aussi. De la même façon, on voit que $\neg\overline{\tau_1} \Rightarrow \neg\overline{\tau_2}$ est aussi une tautologie intuitionniste, ce qui finit la démonstration. ■

Corollaire 4.12 Pour tout type τ de $\mathfrak{L}_{0,+}[T]$ et toute expression arithmétique e telle que $\overline{e} = \tau$,

$$\tau \cong 0 \text{ dans } \mathfrak{L}_{0,+}[T] \iff \mathfrak{Ens}_\mathfrak{f} \models \tau = 0 \iff N_0 \models e = 0 \quad .$$

Démonstration :

$$\begin{aligned} \tau \cong 0 \text{ dans } \mathfrak{L}_{0,+}[T] &\iff \neg\tau \cong \neg 0 \text{ dans } \mathfrak{L}_{0,+}[T] \\ &\iff \mathfrak{Ens}_\mathfrak{f} \models \neg\tau = \neg 0 \\ &\iff \mathfrak{Ens}_\mathfrak{f} \models \tau = 0 \end{aligned}$$

■

Corollaire 4.13 Le problème de savoir si deux négations de types sont isomorphes dans la théorie des biCCC est décidable.

4.3 Preuve du lemme 4.9

Nous allons montrer que dans la théorie équationnelle du λ-calcul typé avec type somme et type vide, la composée

$$\begin{aligned}
&t_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{V},\mathcal{U}}[\varphi,\phi] \circ t_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{U},\mathcal{V}}[\varphi,\phi] \\
&\quad = \lambda h.\ t_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{V},\mathcal{U}}[\varphi,\phi](t_{\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{U},\mathcal{V}}[\varphi,\phi](h)) \\
&\quad = \lambda h.\ t[\varphi,\phi]((p[\varphi,\phi,proj_1\ \ h, proj_2\ \ h],\ p[\phi,\varphi,proj_2\ \ h, proj_1\ \ h])) \\
&\quad = \lambda h.\ (p[\varphi,\phi,p[\varphi,\phi,proj_1\ \ h, proj_2\ \ h],\ p[\phi,\varphi,proj_2\ \ h, proj_1\ \ h]], \\
&\qquad\quad p[\phi,\varphi,p[\phi,\varphi,proj_2\ \ h, proj_1\ \ h],\ p[\varphi,\phi,proj_1\ \ h, proj_2\ \ h]])
\end{aligned}$$

est égale à l'identité du type $(\mathcal{A}^{\mathcal{U}} + \mathcal{B}^{\mathcal{U}})^{\mathcal{V}} \times (\mathcal{C}^{\mathcal{V}} + \mathcal{D}^{\mathcal{V}})^{\mathcal{U}}$. Pour cela, nous allons prouver que les première et seconde composantes de la paire du dernier terme sont respectivement égales aux termes $proj_1\ \ (h)$ et $proj_2\ \ (h)$ dans le contexte $h : (\mathcal{A}^{\mathcal{U}} + \mathcal{B}^{\mathcal{U}})^{\mathcal{V}} \times (\mathcal{C}^{\mathcal{V}} + \mathcal{D}^{\mathcal{V}})^{\mathcal{U}}$.

En effet, par un calcul direct, on voit que la deuxième composante de la paire

$$p[\phi,\varphi,p[\phi,\varphi,proj_2\ \ h, proj_1\ \ h], p[\varphi,\phi,proj_1\ \ h, proj_2\ \ h]]$$

est égale au terme

$$\begin{aligned}
&\lambda u : \mathcal{U}. \\
&\quad case((proj_2\ \ h)\ \ u, \\
&\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ in_1(\lambda v : \mathcal{V}. \\
&\qquad\qquad case((proj_1\ \ h)\ \ v, \\
&\qquad\qquad\quad f_1 : \mathcal{A}^{\mathcal{U}}.\ (g_1\ \ v) \\
&\qquad\qquad\quad f_2 : \mathcal{B}^{\mathcal{U}}.\ proj_2\ \ (\varphi(\phi((f_2\ \ u),(g_1\ \ v)))) \\
&\qquad\qquad\quad) \\
&\qquad\qquad), \\
&\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ in_2(\lambda v : \mathcal{V}. \\
&\qquad\qquad case((proj_1\ \ h)\ \ v, \\
&\qquad\qquad\quad f_1 : \mathcal{A}^{\mathcal{U}}.\ proj_2\ \ (\phi(\varphi((f_1\ \ u),(g_2\ \ v)))), \\
&\qquad\qquad\quad f_2 : \mathcal{B}^{\mathcal{U}}.\ (g_2\ \ v) \\
&\qquad\qquad\quad) \\
&\qquad\qquad) \\
&\qquad)
\end{aligned}$$

et, comme φ et ϕ sont mutuellement inverses, le terme ci-dessus est égal au suivant :

$$\begin{array}{l}\lambda u : \mathcal{U}.\, case((proj_2\ \ h)\ \ u,\\ \qquad g_1 : \mathcal{C}^{\mathcal{V}}.\, in_1\ (\lambda v : \mathcal{V}.\, case((proj_1\ \ h)\ \ v,\\ \qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\, (g_1\ \ v)\\ \qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\, (g_1\ \ v)\\ \qquad\qquad)\\ \qquad),\\ \qquad g_2 : \mathcal{D}^{\mathcal{V}}.\, in_2\ (\lambda v : \mathcal{V}.\, case((proj_1\ \ h)\ \ v,\\ \qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\, (g_2\ \ v),\\ \qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\, (g_2\ \ v)\\ \qquad\qquad)\\ \qquad)\\ )\end{array}$$

qui, par extensionalité, est égal à $(proj_2\ \ h)$.

L'autre identité

$$p[\varphi, \phi, p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], p[\phi, \varphi, proj_2\ \ h, proj_1\ \ h]] = (proj_1\ \ h)$$

est établie de la même façon.

Les détails du calcul sont présentés ci-après :

$$\varphi : \mathcal{A}.\mathcal{D} \to \mathcal{B}.\mathcal{C}, \phi : \mathcal{B}.\mathcal{C} \to \mathcal{A}.\mathcal{D} \vdash t_{\mathcal{U},\mathcal{V}}[\varphi, \phi] : (\mathcal{A}^{\mathcal{U}} + \mathcal{B}^{\mathcal{U}})^{\mathcal{V}}.(\mathcal{C}^{\mathcal{V}} + \mathcal{D}^{\mathcal{V}})^{\mathcal{U}} \to (\mathcal{A}^{\mathcal{V}} + \mathcal{B}^{\mathcal{V}})^{\mathcal{U}}.(\mathcal{C}^{\mathcal{U}} + \mathcal{D}^{\mathcal{U}})^{\mathcal{V}}$$

$$t_{\mathcal{U},\mathcal{V}}[\varphi, \phi] \stackrel{\text{def}}{=} \lambda h : (\mathcal{A}^{\mathcal{U}} + \mathcal{B}^{\mathcal{U}})^{\mathcal{V}}.(\mathcal{C}^{\mathcal{V}} + \mathcal{D}^{\mathcal{V}})^{\mathcal{U}}.\ (\ p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h],\ q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h]\ )$$

$$\begin{array}{l}p[\varphi, \phi, f, g] \stackrel{\text{def}}{=} \lambda u : \mathcal{U}.\, case((g\ \ u),\\ \qquad\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\, in_1\ \ (\ p_1[\phi, f, g_1, u]\ ),\\ \qquad\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\, in_2\ \ (\ p_2[\varphi, f, g_2, u]\ )\\ \qquad\qquad)\end{array}$$

$$\begin{array}{l}p_1[\phi, f, g_1, u] = \lambda v : \mathcal{V}.\, case((f\ \ v),\\ \qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\, (f_1\ \ u),\\ \qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\, proj_1\ \ (\phi((f_2\ \ u), (g_1\ \ v)))\\ \qquad\qquad)\end{array}$$

$$\begin{array}{l}p_2[\varphi, f, g_2, u] = \lambda v : \mathcal{V}.\, case((f\ \ v),\\ \qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\, proj_1\ \ (\varphi((f_1\ \ u), (g_2\ \ v))),\\ \qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\, (f_2\ \ u)\\ \qquad\qquad)\end{array}$$

$$\begin{array}{l}q[\varphi, \phi, f, g] \stackrel{\text{def}}{=} \lambda v : \mathcal{V}.\, case((f\ \ v),\\ \qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\, in_1\ \ (\ q_1[\varphi, f_1, g, v]\ ),\\ \qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\, in_2\ \ (\ q_2[\phi, f_2, g, v]\ )\\ \qquad\qquad)\end{array}$$

$$\begin{array}{l}q_1[\varphi, f_1, g, v] \stackrel{\text{def}}{=} \lambda u : \mathcal{U}.\, case((g\ \ u),\\ \qquad\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\, (g_1\ \ v),\\ \qquad\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\, proj_2\ \ (\varphi((f_1\ \ u), (g_2\ \ v)))\\ \qquad\qquad)\end{array}$$

$$
\begin{array}{l}
q_2[\phi, f_2, g, v] \stackrel{\text{def}}{=} \lambda u : \mathcal{U}.\ case((g\ \ u),\\
\qquad\qquad\qquad\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ proj_2\ \ (\phi((f_2\ \ u), (g_1\ \ v))),\\
\qquad\qquad\qquad\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ (g_2\ \ v)\\
\qquad\qquad\qquad\qquad)
\end{array}
$$

$$
\begin{array}{l}
\lambda h.\ t_{\mathcal{V},\mathcal{U}}[\varphi, \phi](t_{\mathcal{U},\mathcal{V}}[\varphi, \phi](h))\\
= \lambda h.\ t_{\mathcal{V},\mathcal{U}}[\varphi, \phi](p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h],\ q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h])\\
= \lambda h.\ (\ \ p[\varphi, \phi, p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h]],\\
\qquad\quad q[\varphi, \phi, p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h]]\ )
\end{array}
$$

$$
\begin{array}{l}
p[\varphi, \phi, p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h]]\\
= \lambda v : \mathcal{V}.\ case(q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h](v),\\
\qquad\qquad g_1 : \mathcal{C}^{\mathcal{U}}.\ in_1\ \ (\ p_1[\phi, p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], g_1, v]\ ),\\
\qquad\qquad g_2 : \mathcal{D}^{\mathcal{U}}.\ in_2\ \ (\ p_2[\varphi, p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], g_2, v]\ )\\
\qquad\qquad)\\
= \lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ in_1\ \ (\ p_1[\phi, p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], q_1[\varphi, f_1, proj_2\ \ h, v], v]\ ),\\
\qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ in_2\ \ (\ p_2[\varphi, p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], q_2[\phi, f_2, proj_2\ \ h, v], v]\ )\\
\qquad\qquad)\\
= \lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ in_1\ (\lambda u : \mathcal{U}.case(p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h](u),\\
\qquad\qquad\qquad\qquad\qquad\qquad f_1' : \mathcal{A}^{\mathcal{V}}.\ f_1'(v),\\
\qquad\qquad\qquad\qquad\qquad\qquad f_2' : \mathcal{B}^{\mathcal{V}}.\ proj_1\ \ (\phi(f_2'(v), q_1[\varphi, f_1, proj_2\ \ h, v](u)))\\
\qquad\qquad\qquad\qquad\qquad\qquad)\\
\qquad\qquad\qquad\qquad),\\
\qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ in_2\ (\lambda u : \mathcal{U}.case(p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h](u),\\
\qquad\qquad\qquad\qquad\qquad\qquad f_1' : \mathcal{A}^{\mathcal{V}}.\ proj_1\ \ (\varphi(f_1'(v), q_2[\phi, f_2, proj_2\ \ h, v](u))),\\
\qquad\qquad\qquad\qquad\qquad\qquad f_2' : \mathcal{B}^{\mathcal{V}}.\ f_2'(v)\\
\qquad\qquad\qquad\qquad\qquad\qquad)\\
\qquad\qquad\qquad\qquad)\\
\qquad\qquad)\\
= \lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ in_1\ (\lambda u : \mathcal{U}.\ case((proj_2\ \ h)\ \ u,\\
\qquad\qquad\qquad\qquad\qquad\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ p_1[\phi, proj_1\ \ h, g_1, u](v),\\
\qquad\qquad\qquad\qquad\qquad\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ proj_1\ \ (\phi(\ \ p_2[\varphi, proj_1\ \ h, g_2, u](v),\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad q_1[\varphi, f_1, proj_2\ \ h, v](u)))\\
\qquad\qquad\qquad\qquad\qquad\qquad)\\
\qquad\qquad\qquad\qquad),\\
\qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ in_2\ (\lambda u : \mathcal{U}.case((proj_2\ \ h)\ \ u,\\
\qquad\qquad\qquad\qquad\qquad\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ proj_1\ \ (\varphi(\ \ p_1[\phi, proj_1\ \ h, g_1, u](v),\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad q_2[\phi, f_2, proj_2\ \ h, v](u)),)\\
\qquad\qquad\qquad\qquad\qquad\qquad g_2 : \mathcal{C}^{\mathcal{V}}.\ p_2[\phi, proj_1\ \ h, g_2, u](v)\\
\qquad\qquad\qquad\qquad\qquad\qquad)\\
\qquad\qquad\qquad\qquad)\\
\qquad\qquad)
\end{array}
$$

$$
\begin{array}{l}
= \lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ in_1\ (\lambda u : \mathcal{U}.case((proj_2\ \ h)\ \ u,\\
\qquad\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ (f_1\ \ u),\\
\qquad\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ proj_1\ \ (\phi(\ \ proj_1\ \ (\varphi(f_1\ \ u, g_2\ \ v)),\ \ )\\
\qquad\qquad\qquad\qquad\qquad\qquad proj_2\ \ (\varphi(f_1\ \ u, g_2\ \ v)))\\
\qquad\qquad)\\
\qquad),\\
\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ in_2\ (\lambda u : \mathcal{U}.case((proj_2\ \ h)\ \ u,\\
\qquad\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ proj_1\ \ (\varphi(\ \ proj_1\ \ (\phi(f_2 u, g_1 v)),\ \ )),\\
\qquad\qquad\qquad\qquad\qquad\qquad proj_2\ \ (\phi(f_2 u, g_1 v)))\\
\qquad\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ (f_2\ \ u)\\
\qquad\qquad)\\
\qquad)\\
\qquad)\\
= \lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ in_1\ (\lambda u : \mathcal{U}.case((proj_2\ \ h)\ \ u,\\
\qquad\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ (f_1\ \ u),\\
\qquad\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ proj_1\ \ (\phi(\varphi(f_1\ \ u, g_2\ \ v)))\\
\qquad\qquad)\\
\qquad),\\
\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ in_2\ (\lambda u : \mathcal{U}.case((proj_2\ \ h)\ \ u,\\
\qquad\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ proj_1\ \ (\varphi(\phi(f_2 u, g_1 v))),\\
\qquad\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ (f_2\ \ u)\\
\qquad\qquad)\\
\qquad)\\
\qquad)\\
q[\varphi, \phi, p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h]]\\
= \lambda u : \mathcal{U}.\ case(p[\varphi, \phi, proj_1\ \ h, proj_2\ \ h](u),\\
\qquad f_1' : \mathcal{A}^{\mathcal{V}}.\ in_1(\ q_1[\varphi, f_1', q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], u]\ ),\\
\qquad f_2' : \mathcal{B}^{\mathcal{V}}.\ in_2(\ q_2[\phi, f_2', q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], u]\ )\\
\qquad)\\
= \lambda u : \mathcal{U}.\ case((proj_2\ \ h)\ \ u,\\
\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ in_1\ \ (\ q_1[\varphi, p_1[\phi, proj_1\ \ h, g_1, u], q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], u]\ ),\\
\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ in_2\ \ (\ q_2[\phi, p_2[\varphi, proj_1\ \ h, g_2, u], q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h], u]\ )\\
\qquad)\\
= \lambda u : \mathcal{U}.\ case((proj_2\ \ h)\ \ u,\\
\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ in_1\ (\lambda v : \mathcal{V}.\ case(q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h](v),\\
\qquad\qquad g_1' : \mathcal{C}^{\mathcal{U}}.\ g_1'(u)\\
\qquad\qquad g_2' : \mathcal{D}^{\mathcal{U}}.\ proj_2\ \ (\varphi(p_1[\phi, proj_1\ \ h, g_1, u](v), g_2'(u)))\\
\qquad\qquad)\\
\qquad),\\
\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ in_2\ (\lambda v : \mathcal{V}.\ case(q[\varphi, \phi, proj_1\ \ h, proj_2\ \ h](v),\\
\qquad\qquad g_1' : \mathcal{C}^{\mathcal{U}}.\ proj_2\ \ (\phi(p_2[\varphi, proj_1\ \ h, g_2, u](v), g_1'(u))),\\
\qquad\qquad g_2' : \mathcal{D}^{\mathcal{U}}.\ g_2'(u)\\
\qquad\qquad)\\
\qquad)\\
\qquad)
\end{array}
$$

$$
\begin{array}{l}
= \lambda u : \mathcal{U}.\ case((proj_2\ \ h)\ \ u,\\
\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ in_1\ (\lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ q_1[\varphi, f_1, proj_2\ \ h, v](u)\\
\qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ proj_2\ \ (\varphi(\ \ p_1[\phi, proj_1\ \ h, g_1, u](v),\\
\qquad\qquad\qquad\qquad q_2[\varphi, f_2, proj_2\ \ h, v](u)))\\
\qquad\qquad)\\
\qquad\quad),\\
\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ in_2\ (\lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ proj_2\ \ (\phi(\ \ p_2[\varphi, proj_1\ \ h, g_2, u](v),\ \ ),\\
\qquad\qquad\qquad\qquad q_1[\varphi, f_1, proj_2\ \ h, v](u))\\
\qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ q_2[\varphi, f_2, proj_2\ \ h, v](u)\\
\qquad\qquad)\\
\qquad\quad)\\
\qquad)
\end{array}
$$

$$
\begin{array}{l}
= \lambda u : \mathcal{U}.\ case((proj_2\ \ h)\ \ u,\\
\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ in_1\ (\lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ (g_1\ \ v)\\
\qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ proj_2\ \ (\varphi(\ \ proj_1\ \ (\phi((f_2\ \ u), (g_1\ \ v))),\ \ )\\
\qquad\qquad\qquad\qquad proj_2\ \ (\phi((f_2\ \ u), (g_1\ \ v))))\\
\qquad\qquad)\\
\qquad\quad),\\
\qquad g_2 : \mathcal{D}^{\mathcal{V}}.\ in_2\ (\lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ proj_2\ \ (\phi((\ \ proj_1\ \ (\varphi((f_1\ \ u), (g_2\ \ v))),\ \ )),\\
\qquad\qquad\qquad\qquad proj_2\ \ (\varphi((f_1\ \ u), (g_2\ \ v))))\\
\qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ (g_2\ \ v)\\
\qquad\qquad)\\
\qquad\quad)\\
\qquad)
\end{array}
$$

$$
\begin{array}{l}
= \lambda u : \mathcal{U}.\ case((proj_2\ \ h)\ \ u,\\
\qquad g_1 : \mathcal{C}^{\mathcal{V}}.\ in_1\ (\lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ (g_1\ \ v)\\
\qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ proj_2\ \ (\varphi(\phi((f_2\ \ u), (g_1\ \ v))))\\
\qquad\quad),\\
\qquad)g_2 : \mathcal{D}^{\mathcal{V}}.\ in_2\ (\lambda v : \mathcal{V}.\ case((proj_1\ \ h)\ \ v,\\
\qquad\qquad f_1 : \mathcal{A}^{\mathcal{U}}.\ proj_2\ \ (\phi(\varphi((f_1\ \ u), (g_2\ \ v)))),\\
\qquad\qquad f_2 : \mathcal{B}^{\mathcal{U}}.\ (g_2\ \ v)\\
\qquad\qquad)\\
\qquad\quad)\\
\qquad)
\end{array}
$$

4.4 Remarques conclusives

Les résultats présentés dans ce chapitre sont les premières avancées significatives dans l'étude des isomorphismes de types en présence de type vide et type somme.

De nombreuses questions restent ouvertes, comme par exemple celle de l'existence d'équations arithmétiques dans le langage constitué de $1, \cdot, \uparrow, 0$ ou de $1, \cdot, \uparrow, +$ qui ne correspondent pas à des isomorphismes de types. Nous avons conjecturé que le résultat de Gurevič [55] établissant la

non-fini-axiomatisabilité de la théorie équationnelle du modèle des entiers naturels $\langle \mathbb{N}^*, 1, \cdot, \uparrow, + \rangle$ peut être généralisé au cas du modèle des entiers naturels $\langle \mathbb{N}, 1, 0, \cdot, \uparrow, + \rangle$, et donc, grâce aux résultats de ce chapitre, que la théorie équationnelle des isomorphismes de types dans les catégories bi-cartésiennes fermées n'est pas finiment axiomatisable. La question de la décidabilité de la théorie équationnelle des isomorphismes de types dans les extensions du λ-calcul typé avec type vide et/ou type somme est encore ouverte. Enfin, les observations de la section 4.2.3 suggèrent que le cadre approprié pour caractériser les isomorphismes de types valides dans la catégorie des ensembles finis avec type flèche, type vide, et type somme pourraient être des calculs pour des logiques classiques ou intermédiaires.

La preuve du lemme 4.9 a été volontairement détaillée pour montrer l'intérêt d'automatiser ce calcul, ce qui sera l'objet des chapitres 6 et 7.

Troisième partie

Normalisation en présence du type somme

Chapitre 5

Formes normales extensionnelles du λ-calcul avec somme

Résumé

Ce chapitre montre comment il est possible de définir par des règles d'inférence une notion de forme normale canonique pour les λ-termes avec somme et zéro. Pour montrer le résultat de normalisation, nous utiliserons la notion de relation logique de Grothendieck.

ÉTANT DONNÉE une sémantique $[\![.]\!]$ du λ-calcul simplement typé dans une catégorie $\mathfrak{S}$, le problème de ***λ-définissabilité*** est le suivant : quelles sont les flèches $f : [\![\sigma]\!] \to [\![\tau]\!]$ de $\mathfrak{S}$ pour lesquelles il existe un λ-terme t tel que $f = [\![\vdash t : \sigma \to \tau]\!]$? En 1980, Gordon Plotkin a donné une caractérisation des termes définissables de la « *Full Type Hierarchy* » [70]. Pour cela, il a introduit la notion de ***relation logique de Kripke***. En 1993, Achim Jung et Jerzy Tiuryn ont généralisé cette notion, en en faisant varier l'« ***arité*** » et ont donné une caractérisation des éléments définissables de n'importe quel modèle de Henkin [58]. Marcelo Fiore a récemment montré que ce résultat de définissabilité pouvait être adapté en un résultat de normalisation pour le λ-calcul simplement typé sans somme [45]. Pour cela, il définit une notion de forme normale à l'aide d'un système de règles d'inférence, et montre que l'ensemble des flèches λ-définissables est inclus dans l'ensemble des flèches définissables par des termes dans cette forme normale. En choisissant la sémantique canonique dans la catégorie cartésienne fermée libre engendrée par les types de base, cela implique que tout λ-terme simplement typé est $\beta\eta$-équivalent à un terme en forme normale. Ce résultat ne donne pas de moyen d'obtenir la forme normale à partir du terme initial (en particulier il n'est pas basé sur la réécriture), d'où la dénomination « ***forme normale extensionnelle*** ».

Dans ce chapitre, nous allons étendre ce résultat au λ-calcul avec type somme et type vide, en nous appuyant essentiellement sur la preuve de λ-définissabilité étendue à ce cadre par Marcelo Fiore et Alex Simpson [46]. Cela va nous permettre d'associer un représentant canonique à chaque classe d'équivalence de termes modulo $\beta\eta$, sans utiliser la réécriture (qui, nous l'avons vu dans le chapitre 2, n'a pas de bonnes propriétés de normalisation/confluence). Nous n'obtenons pas

exactement l'unicité, en particulier à cause des « *conversions commutatives* » (voir définition 5.11) que l'on peut difficilement éviter. Cependant la forme des termes obtenus est très contrainte et les possibilités de $\beta\eta$-conversion sont très limitées. Le travail le plus proche de celui-ci est un article de Thorsten Altenkirch, Peter Dybjer, Martin Hofmann et Philip Scott [2]. Grâce à des techniques proches des nôtres dans les faisceaux, ils parviennent à une notion de forme normale unique pour un λ-calcul avec *case* n-aires dans lequel les branches ne sont pas ordonnées. Notre preuve présente l'avantage de donner une construction plus explicite, et correspond au cas des *case* binaires qui nous intéresse.

Ce résultat de normalisation a un intérêt pour l'étude des isomorphismes de types, car il réduit de manière drastique les possibilités de construction des λ-termes. Il va nous permettre de raisonner sur la structure des λ-termes. En réalité, c'est grâce à ces formes normales canoniques que j'ai pu trouver le terme présenté au chapitre 4 (figure 4.3 page 101), plus exactement en essayant de prouver la conjecture inverse, à savoir qu'un tel isomorphisme ne pouvait exister. Nous verrons dans les chapitres 6 et 7 un moyen pour obtenir cette forme normale canonique pour un terme donné. Cela nous donnera un moyen simple de vérifier qu'un terme est bien un isomorphisme, en comparant la normalisation de la composition de ce terme avec son inverse supposé, avec celle de l'identité.

5.1 Survol de la preuve

Commençons par exposer brièvement le cas du λ-calcul sans somme (voir Fiore [45]). Soit $[\![.]\!]$ une sémantique du λ-calcul simplement typé dans une catégorie cartésienne fermée $\mathfrak{S}$, obtenue comme à la page 57 à partir d'une interprétation quelconque des types de base. On définit l'ensemble $\mathcal{D}_\tau(\Gamma)$ des flèches ***définissables*** de $\mathfrak{S}$ de la manière suivante :

$$\mathcal{D}_\tau(\Gamma) = \{\ [\![\Gamma \vdash t : \tau]\!] \mid \Gamma \vdash t : \tau\ \}$$

$\mathcal{D}_\tau(\Gamma)$ est un sous-ensemble de $\mathfrak{S}([\![\Gamma]\!],\ [\![\tau]\!])$.

Considérons maintenant la catégorie $\mathbf{C}$ dite ***catégorie des contextes***, dont les objets sont les contextes et les morphismes sont les renommages injectifs de contextes préservant les types ; c'est-à-dire les fonctions $\rho : \mathrm{dom}(\Gamma) \to \mathrm{dom}(\Gamma')$ telles que si $x : \tau \in \Gamma$, alors $\rho(x) : \tau \in \Gamma'$. On peut étendre facilement $[\![.]\!]$ en un foncteur de $\mathbf{C}$ dans $\mathfrak{S}$ de la manière suivante :

$$[\![\rho]\!] = \langle [\![\Gamma' \vdash \rho(x) : \tau]\!] \rangle_{(x:\tau)\in\Gamma}$$

Définissons les ensembles $\mathcal{R}_\tau(\Gamma) \subseteq \mathfrak{S}([\![\Gamma]\!],\ [\![\tau]\!])$ pour tout $\Gamma \in \mathbf{C}$ et pour tout type τ de la manière suivante :

$$\begin{array}{rcll} \mathcal{R}_\theta(\Gamma) & = & \mathcal{D}_\theta(\Gamma) & \text{pour } \theta \text{ type de base} \\ \mathcal{R}_1(\Gamma) & = & \top(\Gamma) & \\ \mathcal{R}_{\tau_1\times\tau_2}(\Gamma) & = & (\mathcal{R}_{\tau_1} \wedge \mathcal{R}_{\tau_2})(\Gamma) & \\ \mathcal{R}_{\sigma\to\tau}(\Gamma) & = & (\mathcal{R}_\sigma \supset \mathcal{R}_\tau)(\Gamma) & \end{array}$$

où

$$\top(\Gamma) \quad = \quad !\,[\![\Gamma]\!]$$

$$
\begin{array}{rcl}
(\mathcal{R}_{\tau_1} \wedge \mathcal{R}_{\tau_2})(\Gamma) & = & \{\, f \in \mathfrak{S}([\![\Gamma]\!],\, [\![\tau_1 \times \tau_2]\!]) \mid \pi_1 \circ f \in \mathcal{R}_{\tau_1}(\Gamma) \text{ et } \pi_2 \circ f \in \mathcal{R}_{\tau_2}(\Gamma) \,\} \\
(\mathcal{R}_{\sigma} \supset \mathcal{R}_{\tau})(\Gamma) & = & \{\, f \in \mathfrak{S}([\![\Gamma]\!],\, [\![\sigma \to \tau]\!]) \mid \forall g \in \mathcal{R}_{\sigma}(\Gamma') \quad \forall \rho : \Gamma' \to \Gamma \quad \mathit{eval} \circ \langle f \circ [\![\rho]\!], g \rangle \in \mathcal{R}_{\tau}(\Gamma') \,\}
\end{array}
$$

On peut vérifier que les objets $\mathcal{D}_\tau$ et $\mathcal{R}_\tau$ sont des ***relations de Kripke***[1]. Les $\mathcal{R}_\tau$ sont appelés ***relations logiques de Kripke***.

Le résultat de définissabilité (dû à Jung et Tiuryn) énonce que pour tout type τ, on a l'égalité $\mathcal{D}_\tau = \mathcal{R}_\tau$. Pour le montrer, il suffit de prouver que

$$
\begin{array}{rcl}
\mathcal{D}_1 & = & \top \\
\mathcal{D}_{\tau_1 \times \tau_2} & = & \mathcal{D}_{\tau_1} \wedge \mathcal{D}_{\tau_2} \\
\mathcal{D}_{\sigma \to \tau} & = & \mathcal{D}_{\sigma} \supset \mathcal{D}_{\tau}
\end{array}
$$

Pour obtenir le résultat de normalisation, Marcelo Fiore a défini deux systèmes d'inférence de types $\vdash_{\mathcal{N}}$ et $\vdash_{\mathcal{M}}$ permettant de définir respectivement les termes ***normaux*** et ***neutres***. On peut définir les ***flèches normales*** et ***neutres*** suivant le même principe que les flèches définissables ci-dessus, à savoir, respectivement :

$$
\begin{array}{rcl}
\mathcal{N}_\tau(\Gamma) & = & \{\, [\![\Gamma \vdash t : \tau]\!] \mid \Gamma \vdash_{\mathcal{N}} t : \tau \,\} \\
\mathcal{M}_\tau(\Gamma) & = & \{\, [\![\Gamma \vdash t : \tau]\!] \mid \Gamma \vdash_{\mathcal{M}} t : \tau \,\}
\end{array}
$$

$\mathcal{M}_\tau$ et $\mathcal{N}_\tau$ sont également des relations de Kripke.

On peut montrer que si des familles de relations de Kripke $(\mathcal{L}_\tau)$ et $(\mathcal{U}_\tau)$ vérifient les inclusions suivantes (pour tous types τ, σ, τ_1, τ_2) :

$$
\begin{array}{c}
\mathcal{U}_1 = \top \\
\mathcal{L}_{\tau_1 \times \tau_2} \subseteq \mathcal{L}_{\tau_1} \wedge \mathcal{L}_{\tau_2} \qquad \mathcal{U}_{\tau_1} \wedge \mathcal{U}_{\tau_2} \subseteq \mathcal{U}_{\tau_1 \times \tau_2} \\
\mathcal{L}_{\sigma \to \tau} \subseteq \mathcal{U}_\sigma \supset \mathcal{L}_\tau \qquad \mathcal{L}_\sigma \supset \mathcal{U}_\tau \subseteq \mathcal{U}_{\sigma \times \tau} \\
\mathcal{L}_\theta \subseteq \mathcal{R}_\theta \subseteq \mathcal{U}_\theta \quad (\theta \text{ type de base})
\end{array}
$$

alors elles vérifient, pour tout type τ :

$$\mathcal{L}_\tau \subseteq \mathcal{R}_\tau \subseteq \mathcal{U}_\tau$$

En prenant $\mathcal{L}_\tau = \mathcal{U}_\tau = \mathcal{D}_\tau$, on retrouve le résultat de définissabilité ($\mathcal{D}_\tau = \mathcal{R}_\tau$). Mais si l'on prend $\mathcal{L}_\tau = \mathcal{M}_\tau$ et $\mathcal{U}_\tau = \mathcal{N}_\tau$, on arrive à prouver les inclusions ci-dessus, et donc

$$\mathcal{M}_\tau \subseteq \mathcal{R}_\tau \subseteq \mathcal{N}_\tau$$

Ce qui montre que toute flèche de $\mathcal{R}_\tau(\Gamma)$ est définissable par une forme normale de type τ dans le contexte Γ.

Pour en conclure que tout terme bien typé est égal à un terme en forme normale, il ne reste plus qu'à montrer que $\mathcal{R}_\tau(\Gamma)$ contient l'interprétation de tous les termes bien typés de type τ dans le contexte Γ.

Nous allons maintenant étendre ce résultat au λ-calcul avec somme et type vide. Le résultat de λ-définissabilité de Achim Jung et Jerzy Tiuryn a été étendu au λ-calcul avec somme et zéro par Marcelo Fiore et Alex Simpson [46]. Pour cela, ils ont eu besoin de placer les contextes dans une catégorie dotée d'un *topologie de Grothendieck*, obtenant ainsi une généralisation de la notion

[1] Pour la définition de relation de Kripke, voir page 117 et [45]

de relation de Kripke appelée *relation de Grothendieck*.

Le reste de ce chapitre montre en détails comment adapter ce résultat pour obtenir un résultat de normalisation pour le λ-calcul avec somme et zéro.

Il nous faut des notions de relations logiques duales de $\mathcal{R}_{\tau_1} \wedge \mathcal{R}_{\tau_2}$ et $\top$, que l'on notera $\mathcal{R}_{\tau_1} \vee \mathcal{R}_{\tau_2}$ et $\bot$. Pour les définir, nous allons utiliser la notion de *recouvrement* de la topologie de Grothendieck.

Remarque 5.1 La règle habituelle d'élimination du 0 est la suivante :

$$\frac{\Gamma \vdash t : 0}{\Gamma \vdash \bot_\tau(t) : \tau}$$

Ici, nous n'utiliserons pas cette règle, mais la suivante :

$$\frac{\Gamma \vdash t : 0}{\Gamma \vdash \bot_\tau : \tau}$$

Cette nouvelle règle nous permettra de n'avoir qu'une seule forme normale de type τ (à savoir $\bot_\tau$) dans un contexte Γ inconsistant. Nous avons vu à la section 2.3.3 page 72 que ce choix ne pose pas de problème.

5.2 Sémantique dans la biCCC des relations de Grothendieck

Pour traiter le cas du co-produit, nous allons introduire la notion de *recouvrement* et de *topologie de Grothendieck* (voir Fiore-Simpson [46]).

Définition 5.2 (topologie de Grothendieck) Soit $\mathfrak{C}$ une catégorie petite. Une (base d'une) ***topologie de Grothendieck*** K sur $\mathfrak{C}$, est définie par la donnée, pour chaque objet a de $\mathfrak{C}$ d'une collection $K(a)$ de ***recouvrements*** [$\triangleright$ *cover*] de a (une famille de morphismes de $\mathfrak{C}$ de codomaine a), satisfaisant les conditions suivantes :

- (Identité) Pour tout $a \in Obj_{\mathfrak{C}}$, $K(a)$ contient la famille constituée du seul morphisme id_a.
- (Stabilité) Pour tout $\{\phi_i : a_i \to a\}_{i\in I} \in K(a)$, et tout morphisme $\psi : b \to a$, il existe une famille $\{\phi'_j : b_j \to b\}_{j\in J} \in K(b)$ telle que toute composition $\psi \circ \phi'_j : b_j \to a$ se factorise par l'intermédiaire d'un ϕ_i (*i.e.* pour tout $j \in J$ il existe $i \in I$ et $\gamma_{ij} : b_j \to a_i$ tels que $\psi \circ \phi'_j = \phi_i \circ \gamma_{ij}$).
- (Transitivité) Pour tout $\{\phi_i : a_i \to a\}_{i\in I} \in K(a)$, et pour tous $\{\gamma_{ij}\}_{j\in J_i} \in K(a_i)$ $(i \in I)$, la famille $\{\phi_i \circ \gamma_{ij}\}_{i\in I, j\in J_i}$ appartient à $K(a)$.

Une catégorie petite munie d'une topologie de Grothendieck est appelée un ***site***.

Les relations de Grothendieck généralisent la notion de relation de Kripke afin de prendre en compte la topologie de Grothendieck.

Définition 5.3 (relation de Grothendieck) Étant donné un site $(\mathfrak{C}, K)$ et un foncteur $\boldsymbol{s} : \mathfrak{C} \to \mathfrak{S}$, une ***$(\mathfrak{C}, K)$-relation de Grothendieck*** R d'***arité*** $\boldsymbol{s}$ sur $A \in Obj_{\mathfrak{S}}$ est une famille $\{ R(c) \subseteq \mathfrak{S}(\boldsymbol{s}(c), A) \}_{c \in Obj_{\mathfrak{C}}}$ vérifiant les deux propriétés suivantes :

► (Monotonie) Pour toute flèche $\psi : c' \to c$ de $\mathfrak{C}$, et toute flèche $x : \boldsymbol{s}(c) \to A$ de $\mathfrak{S}$, si $x \in R(c)$, alors $x \circ \boldsymbol{s}(\psi) \in R(c')$.

► (Caractère local) Pour tout recouvrement $\{ \phi_i : c_i \to c \}_{i \in I} \in K(c)$ et pour tout $x : \boldsymbol{s}(c) \to A$ de $\mathfrak{S}$, si $x \circ \boldsymbol{s}(\phi_i) \in R(c_i)$ pour tout $i \in I$, alors $x \in R(c)$.

La définition de relation de Kripke est la même sans la propriété de caractère local.

Nous utiliserons comme foncteur d'arité un foncteur « sémantique » allant d'une catégorie de contextes (contraints) dans $\mathfrak{S}$. Une flèche de c' vers c dans $\mathfrak{C}$ peut être vue comme une « extension de contexte » (ou une flèche vers un « monde futur »). On demande à une relation de Kripke d'être stable par cette extension. Autrement dit, nous devons être capables de transformer toute flèche de $R(c)$ en une flèche de de $R(c')$, qui est sa « ré-interprétation » dans le nouveau contexte.

Pour obtenir une relation de Grothendieck, on demande non seulement de pouvoir ré-interpréter les flèches après changement de contexte, mais également que lorsque nous avons une flèche x de l'interprétation d'un contexte dans A, s'il existe un recouvrement de ce contexte qui « envoie » x dans la relation (pour les composants du recouvrement), alors x est aussi dans la relation pour le contexte original !

Nous allons maintenant définir la catégorie des relations de Grothendieck.

Définition 5.4 (catégorie des relations de Grothendieck) Étant donné un site $(\mathfrak{C}, K)$ et un foncteur $\boldsymbol{s} : \mathfrak{C} \to \mathfrak{S}$, la ***catégorie des relations de Grothendieck*** $\underline{\mathrm{G}}(\mathfrak{C}, K, \boldsymbol{s})$ est définie de la manière suivante :

- Les objets sont des couples (A, R) constitués d'un objet A de $\mathfrak{S}$ et d'une $(\mathfrak{C}, K)$-relation de Grothendieck R d'arité $\boldsymbol{s}$ sur A ;
- Les morphismes $(A, R) \to (A', R')$ sont les morphismes $f : A \to A'$ de $\mathfrak{S}$ tels que pour tout objet c de $\mathfrak{C}$ et pour tous morphismes $x : \boldsymbol{s}(c) \to A$ de $R(c)$, la composée $f \circ x$ est dans $R'(c)$.

Nous allons maintenant définir quelques notations.

Définition 5.5 Soit $\underline{\mathrm{G}}(\mathfrak{C}, K, \boldsymbol{s})$ une catégorie de relations de Grothendieck. Pour tout $c \in Obj_{\mathfrak{C}}$, pour $A, B \in Obj_{\mathfrak{S}}$, et pour R_A et R_B relations de Grothendieck sur A et B respectivement, on pose :

$$
\begin{array}{rcl}
\top(c) & = & \{!\boldsymbol{s}(c)\} \\
(R_A \wedge R_B)(c) & = & \{ f : \boldsymbol{s}(c) \to A \times B \mid \pi_1 \circ f \in R_A(c) \text{ et } \pi_2 \circ f \in R_B(c) \} \\
(R_A \supset R_B)(c) & = & \{ f : \boldsymbol{s}(c) \to B^A \mid \\
 & & \qquad \forall a \in R_A(c') \quad \forall \psi : c' \to c \quad eval \circ \langle f \circ \boldsymbol{s}(\psi), a \rangle \in R_B(c') \} \\
\bot(c) & = & \begin{cases} \emptyset & \text{si la famille vide n'est pas un recouvrement de } K(c) \\ \{ f : \boldsymbol{s}(c) \to \mathfrak{o} \} & \text{si la famille vide est un recouvrement de } K(c) \end{cases}
\end{array}
$$

$$(R_A \vee R_B)(c) = \{ f : \mathbf{s}(c) \to A + B \mid \text{il existe un recouvrement } \{ \phi_i : c_i \to c \}_{i\in I} \in K(c) \text{ tel que pour tout } i \in I :$$
$$\text{– soit } f \circ \mathbf{s}(\phi_i) = \iota_1 \circ a : \mathbf{s}(c_i) \to A \text{ pour un } a : \mathbf{s}(c_i) \to A \text{ de } R_A(c_i)$$
$$\text{– soit } f \circ \mathbf{s}(\phi_i) = \iota_2 \circ a' : \mathbf{s}(c_i) \to B \text{ pour un } a' : \mathbf{s}(c_i) \to B \text{ de } R_B(c_i) \}$$

Proposition 5.6 Ces constructions définissent des relations de Grothendieck.

Démonstration :

Il faut vérifier pour chacune d'elles les propriétés de monotonie et caractère local.

▶ $\top(c)$

Monotonie Soit $\psi : c' \to c$ et $x : \mathbf{s}(c) \to 1$ dans $\top(c)$. On a $x = !\mathbf{s}(c)$, et par unicité, $x \circ \mathbf{s}(\psi) = !\mathbf{s}(c')$, donc appartient à $\top(c')$.

Caractère local Soit $x : \mathbf{s}(c) \to 1$. Nécessairement $x = !\mathbf{s}(c)$ donc $x \in \top(c)$.

▶ $(R_A \wedge R_B)(c)$

Monotonie Supposons $x \in (R_A \wedge R_B)(c)$. Nous avons $\pi_1 \circ x \in R_A(c)$ donc en utilisant la propriété de monotonie de la relation de Grothendieck R_A, on peut déduire que

$$\pi_1 \circ x \circ \mathbf{s}(\psi) \in R_A(c')$$

De même

$$\pi_2 \circ x \circ \mathbf{s}(\psi) \in R_B(c')$$

Et donc $x \circ \mathbf{s}(\psi) \in (R_A \wedge R_B)(c')$

Caractère local Soit $\{ \phi_i : c_i \to c \}_{i\in I} \in K(c)$, et $x : \mathbf{s}(c) \to A \times B$. Supposons $x \circ \mathbf{s}(\phi_i) \in (R_A \wedge R_B)(c_i)$ pour tout $i \in I$. On a donc $\pi_1 \circ x \circ \mathbf{s}(\phi_i) \in R_A(c_i)$. En utilisant la propriété de caractère local de R_A, on déduit que $\pi_1 \circ x \in R_A(c)$. De même $\pi_2 \circ x \in R_B(c)$. Donc $x \in (R_A \wedge R_B)(c)$

▶ $(R_A \supset R_B)(c)$

Monotonie Supposons $x \in (R_A \supset R_B)(c)$. Soient $a \in R_A(c'')$ et $\psi' : c'' \to c'$.
La flèche $eval \circ \langle x \circ \mathbf{s}(\psi) \circ \mathbf{s}(\psi'), a\rangle = eval \circ \langle x \circ \mathbf{s}(\psi \circ \psi'), a\rangle$ est dans $R_B(c'')$ d'après la monotonie appliquée à x.

Caractère local Soit $\{ \phi_i : c_i \to c \}_{i\in I} \in K(c)$, et $x : \mathbf{s}(c) \to B^A$. Supposons $x \circ \mathbf{s}(\phi_i) \in (R_A \supset R_B)(c_i)$ pour tout $i \in I$.
Soient $a \in R_A(c')$ et $\psi : c' \to c$.
Par stabilité, il existe un recouvrement $\left\{ \phi'_j : c'_j \to c' \right\}_{j\in J}$ dans $K(c')$ tel que pour tout $j \in J$ il existe $i \in I$ et $\gamma_{ij} : c'_j \to c_i$ tels que $\psi \circ \phi'_j = \phi_i \circ \gamma_{ij}$.

$$\begin{aligned} eval \circ \langle x \circ \mathbf{s}(\psi), a\rangle \circ \mathbf{s}(\phi'_j) &= eval \circ \langle x \circ \mathbf{s}(\psi) \circ \mathbf{s}(\phi'_j), a \circ \mathbf{s}(\phi'_j)\rangle \\ &= eval \circ \langle x \circ \mathbf{s}(\psi \circ \phi'_j), a \circ \mathbf{s}(\phi'_j)\rangle \\ &= eval \circ \langle x \circ \mathbf{s}(\phi_i \circ \gamma_{ij}), a \circ \mathbf{s}(\phi'_j)\rangle \end{aligned}$$

$$= \quad eval \circ \langle x \circ \boldsymbol{s}(\phi_i) \circ \boldsymbol{s}(\gamma_{ij}), a \circ \boldsymbol{s}(\phi'_j) \rangle$$

Par monotonie, $a \circ \boldsymbol{s}(\phi'_j) \in R_A(c'_j)$.
Et $x \circ \boldsymbol{s}(\phi_i) \in (R_A \supset R_B)(c_i)$, donc comme $\gamma_{ij} : c'_j \to c_i$, on a

$$eval \circ \langle x \circ \boldsymbol{s}(\phi_i) \circ \boldsymbol{s}(\gamma_{ij}), a \circ \boldsymbol{s}(\phi'_j) \rangle \in R_B(c'_j)$$

Donc

$$eval \circ \langle x \circ \boldsymbol{s}(\psi), a \rangle \circ \boldsymbol{s}(\phi'_j) \in R_B(c'_j)$$

pour tout $j \in J$.
En utilisant la propriété de caractère local pour R_B, on déduit que

$$eval \circ \langle x \circ \boldsymbol{s}(\psi), a \rangle \in R_B(c)$$

et donc que $x \in (R_A \supset R_B)(c)$

▶ $\bot(c)$

Monotonie

- Premier cas : la famille vide n'est pas dans $K(c)$. Il n'existe aucun x dans $\bot(c)$ donc la monotonie est vraie.
- Deuxième cas : la famille vide est dans $K(c)$. Par stabilité, elle est aussi dans $K(c')$. Supposons $x \in \bot(c)$. La flèche $x \circ \boldsymbol{s}(\psi)$ va de $\boldsymbol{s}(c')$ dans $\mathfrak{o}$, donc appartient à $\bot(c')$.

Caractère local

- Premier cas : la famille vide n'est pas dans $K(c)$.
  Soit $\{ \phi_i : c_i \to c \}_{i \in I} \in K(c)$, et $x : \boldsymbol{s}(c) \to \mathfrak{o}$. Supposons $x \circ \boldsymbol{s}(\phi_i) \in \bot(c_i)$ pour tout $i \in I$.
  Cela implique que $\bot(c_i)$ n'est pas vide, donc la famille vide est dans $K(c_i)$.
  Par transitivité, la famille vide appartient aussi à $K(c)$, ce qui est en contradiction avec les hypothèses. On en déduit que la propriété $\forall i \in I.\ x \circ \boldsymbol{s}(\phi_i) \in \bot(c_i)$ n'est jamais vérifiée, et donc que le caractère local est vrai.
- Deuxième cas : la famille vide est dans $K(c)$.
  x est une flèche entre $\boldsymbol{s}(c)$ et $\mathfrak{o}$, donc $x \in \bot(c)$.

▶ $(R_A \vee R_B)(c)$

Monotonie Supposons $x \in (R_A \vee R_B)(c)$. La définition nous donne un recouvrement $\{ \phi_i : c_i \to c \}_{i \in I} \in K(c)$. Par stabilité, il existe un recouvrement $\left\{ \phi'_j : c'_j \to c' \right\}_{j \in J}$ dans $K(c')$ tel que pour tout $j \in J$ il existe $i \in I$ et $\gamma_{ij} : c'_j \to c_i$ tels que $\psi \circ \phi'_j = \phi_i \circ \gamma_{ij}$.
Nous avons

$$\begin{aligned} x \circ \boldsymbol{s}(\psi) \circ \boldsymbol{s}(\phi'_j) &= x \circ \boldsymbol{s}(\psi \circ \phi'_j) \\ &= x \circ \boldsymbol{s}(\phi_i \circ \gamma_{ij}) \\ &= x \circ \boldsymbol{s}(\phi_i) \circ \boldsymbol{s}(\gamma_{ij}) \\ &= \iota_k \circ a \circ \boldsymbol{s}(\gamma_{ij}) \end{aligned}$$

avec k et a donnés par la définition de $(R_A \vee R_B)(c)$. Le domaine de la flèche $a \circ \boldsymbol{s}(\gamma_{ij})$ est $\boldsymbol{s}(c'_j)$, donc elle appartient à $R_A(c'_j)$ si $a \in R_A(c_i)$ et à $R_B(c'_j)$ si $a \in R_B(c_i)$.

Caractère local Soit $\{ \phi_i : c_i \to c \}_{i \in I} \in K(c)$, et $x : \boldsymbol{s}(c) \to A + B$. Supposons $x \circ \boldsymbol{s}(\phi_i) \in (R_A \vee R_B)(c_i)$ pour tout $i \in I$.

Pour tout i, la définition de $(R_A \vee R_B)(c_i)$ nous donne un recouvrement $\{ \phi_{ij} : c_{ij} \to c_i \}_{j \in J_i} \in K(c_i)$.
Considérons le recouvrement $\{ \phi_i \circ \phi_{ij} : c_{ij} \to c \}_{i \in I,\, j \in J_i} \in K(c)$.
$x \circ s(\phi_i \circ \phi_{ij}) = x \circ s(\phi_i) \circ s(\phi_{ij}) = \iota_k \circ a_{ij}$
où k et $a_{ij} : s(c_{ij}) \to A$ ou $a_{ij} : s(c_{ij}) \to B$ sont donnés par la définition de $(R_A \vee R_B)(c_i)$.

■

Proposition 5.7 (préservation de la structure bi-cartésienne fermée) Pour un site $(\mathfrak{C}, K)$ et un foncteur $s : \mathfrak{C} \to \mathfrak{S}$ dans une catégorie bi-cartésienne fermée donnés, la catégorie des relations de Grothendieck $\underline{G}(\mathfrak{C}, K, s)$ est bi-cartésienne fermée et le foncteur d'oubli $\underline{G}(\mathfrak{C}, K, s) \to \mathfrak{S}$ préserve la structure bi-cartésienne fermée.

Démonstration :

Donnons les constructions pour les objets terminal, produit, exponentielle, initial et co-produit.

▶ terminal L'objet terminal est $(1, \top)$.

En effet, soit (A, R) dans $\underline{G}(\mathfrak{C}, K, s)$. Les flèches de la catégorie des relations de Grothendieck sont un sous-ensemble des flèches de $\mathfrak{S}$. Nous savons que 1 est terminal, donc il existe une unique flèche $!A$ de A dans 1. Nous voulons montrer que c'est une flèche de la catégorie des relations de Grothendieck.

Soit $x \in R_A(c)$. La flèche $!A \circ x$ va de $s(c)$ dans 1, et donc appartient à $\top(c)$. On en déduit que $!A$ est dans $\underline{G}(\mathfrak{C}, K, s)$.

▶ produit Le produit de (A, R_A) et (B, R_B) est $(A \times B, R_A \wedge R_B)$.

Pour le montrer, notons tout d'abord que les projections π_1 et π_2 de $\mathfrak{S}$ sont des flèches de la catégorie des relations de Grothendieck, car pour tout $x \in (R_A \wedge R_B)(c)$, $\pi_1 \circ x \in R_A(c)$ par définition de $R_A \wedge R_B$.
Ensuite, nous vérifions que le diagramme du produit commute. Soit (D, R_D) dans $\underline{G}(\mathfrak{C}, K, s)$, $f : (D, R_D) \to (A, R_A)$, et $g : (D, R_D) \to (B, R_B)$. Nous savons que $f : D \to A$ et $g : D \to B$ sont des flèches de $\mathfrak{S}$. Prouvons que $\langle f, g \rangle$ appartient à $\underline{G}(\mathfrak{C}, K, s)$.
Soit $x \in R_D(c)$. Nous savons que $f \circ x \in R_A(c)$ et que $g \circ x \in R_B(c)$, c'est-à-dire : $\pi_1 \circ \langle f, g \rangle \circ x \in R_A(c)$ et $\pi_2 \circ (f, g) \circ x \in R_B(c)$. Donc $\langle f, g \rangle \circ x \in (R_A \wedge R_B)(c)$.

▶ exponentielle L'exponentielle de (A, R_A) et (B, R_B) est l'objet $(B^A, R_A \supset R_B)$.

Pour montrer cela, montrons d'abord que *eval* appartient à $\underline{G}(\mathfrak{C}, K, s)$. Soit $x \in ((R_A \supset R_B) \wedge R_A)(c)$. Nous savons que $\pi_1 \circ x \in (R_A \supset R_B)(c)$ et $\pi_2 \circ x \in R_A(c)$. Par définition de $(R_A \supset R_B)$, en prenant $\psi = \mathrm{id}$, nous avons $eval \circ \langle \pi_1 \circ x, \pi_2 \circ x \rangle = eval \circ x \in R_B(c)$.

Soit $f : (C \times A, R_C \wedge R_A) \to (B, R_B)$. Soit h l'unique flèche de C vers B^A faisant commuter le diagramme de l'exponentielle dans $\mathfrak{S}$.

Nous voulons maintenant montrer que h appartient à $\underline{G}(\mathfrak{C}, K, s)$. Soit $x \in R_C(c)$. La flèche $h \circ x$ appartient-elle à $(R_A \supset R_B)(c)$?

Soit $\psi : c' \to c$ et $g \in R_A(c')$. Par commutation du diagramme de l'exponentielle dans $\mathfrak{S}$, nous avons $eval \circ \langle h \circ x \circ s(\psi), g \rangle = f \circ \langle x \circ s(\psi), g \rangle$. La flèche $x \circ s(\psi)$ appartient à $R_C(c')$ par monotonie, et g appartient à $R_A(c')$, donc $\langle x \circ s(\psi), g \rangle$ appartient à $(R_C \wedge R_A)(c')$. Nous pouvons donc déduire que $eval \circ \langle h \circ x \circ s(\psi), g \rangle$ appartient à $R_B(c')$, puisque f est une flèche de $\underline{G}(\mathfrak{C}, K, s)$.

▶ initial L'objet initial est $(0, \bot)$.

En effet, soit (A, R_A) dans $\underline{G}(\mathfrak{C}, K, s)$. $\mathfrak{S}$ est une biCCC, donc nous savons qu'il existe une

unique flèche $¡A$ de $\mathfrak{o}$ vers A.

Si $\emptyset \notin K(c)$, nous avons $\bot(c) = \emptyset$, et donc $¡A \in \underline{\mathrm{G}}(\mathfrak{C}, K, \boldsymbol{s})$.

Si $\emptyset \in K(c)$, nous utilisons la propriété de caractère local avec le recouvrement vide pour montrer que $¡A \circ x \in R_A(c)$ (où $x \in \bot(c)$).

▶ co-produit Le co-produit de (A_1, R_{A_1}) et (A_2, R_{A_2}) est $(A_1 + A_2, R_{A_1} \vee R_{A_2})$.

Pour montrer cela, nous devons vérifier que $\iota_1 : A_1 \to A_1 + A_2$ et $\iota_2 : A_2 \to A_1 + A_2$ sont des flèches de la catégorie des relations de Grothendieck.

Soit x dans $R_{A_1}(c)$. Prenons le recouvrement $\{id_c\} \in K(c)$. $\iota_1 \circ x \circ \boldsymbol{s}(id_c) = \iota_1 \circ x$ est de la forme requise pour que ι_1 appartienne à $(R_{A_1} \vee R_{A_2})(c)$. De la même façon, ι_2 appartient à $(R_{A_1} \vee R_{A_2})(c)$.

Considérons un objet (D, R_D) et deux flèches $f_1 : (A_1, R_{A_1}) \to (D, R_D)$ et $f_2 : (A_2, R_{A_2}) \to (D, R_D)$.

$A_1 + A_2$ est un co-produit, donc il existe une unique flèche $\binom{f_1}{f_2} : A_1 + A_2 \to D$ telle que $\binom{f_1}{f_2} \circ \iota_i = f_i$ $(i = 1, 2)$.

C'est le seul candidat pour devenir $\binom{f_1}{f_2}$ dans la catégorie des relations de Grothendieck. Montrons donc qu'il appartient à cette catégorie.

Soit $x \in (R_{A_1} \vee R_{A_2})(c)$. Nous voulons montrer que $\binom{f_1}{f_2} \circ x \in R_D(c)$. Il existe un recouvrement $\{\ \phi_i : c_i \to c\ \}_{i \in I} \in K(c)$ tel que $\forall i \in I.\ \exists j_i, x_i \in R_{A_{j_i}}(c_i). x \circ \boldsymbol{s}(\phi_i) = \iota_{j_i} \circ x_i$.

Pour tout $i \in I$, comme $x_i \in R_{A_{j_i}}(c_i)$, nous pouvons montrer, en utilisant que f_1 et f_2 sont dans la catégorie des relations de Grothendieck, que

$$\binom{f_1}{f_2} \circ x \circ \boldsymbol{s}(\phi_i) = \binom{f_1}{f_2} \circ \iota_{j_i} \circ x_i = f_{j_i} \circ x_i \in R_D(c_i)$$

Donc, par caractère local, $\binom{f_1}{f_2} \circ x \in R_D(c)$. ■

5.3 Normalisation dans les biCCC

5.3.1 Lemme de base

Définition 5.8 (relation logique de Grothendieck) Soit $(\mathfrak{C}, K)$ un site, $\boldsymbol{s} : \mathfrak{C} \to \mathfrak{S}$ un foncteur vers une biCCC, et $\mathcal{I}(.)$ une interprétation des types de base dans $\mathfrak{S}$.
Pour une famille de relations de Grothendieck $\{(\mathcal{I}[\![\theta]\!], \mathcal{R}_\theta)\}_\theta$ dans $\underline{\mathrm{G}}(\mathfrak{C}, K, \boldsymbol{s})$ indexées par les types de base, soit $\{(\mathcal{I}[\![\tau]\!], \mathcal{R}_\tau)\}_\tau$ la famille de relations de Grothendieck indexées par les types induite par la structure bi-cartésienne fermée de $\underline{\mathrm{G}}(\mathfrak{C}, K, \boldsymbol{s})$:

$$\begin{aligned} \mathcal{R}_0 &= \bot \\ \mathcal{R}_1 &= \top \\ \mathcal{R}_{\sigma\times\tau} &= \mathcal{R}_\sigma \wedge \mathcal{R}_\tau \\ \mathcal{R}_{\sigma+\tau} &= \mathcal{R}_\sigma \vee \mathcal{R}_\tau \\ \mathcal{R}_{\sigma\to\tau} &= \mathcal{R}_\sigma \supset \mathcal{R}_\tau \end{aligned}$$

Ces relations sont appelées ***relations logiques de Grothendieck***.

Marcelo Fiore et Alex Simpson ont démontré le lemme suivant [46] :

Lemme 5.9 (lemme fondamental des relations logiques de Grothendieck) Soit $(\mathfrak{C}, K)$ un site, $s : \mathfrak{C} \to \mathfrak{S}$ un foncteur vers une biCCC, et $\mathcal{I}(.)$ une interprétation des types de base dans $\mathfrak{S}$.
Soit $\{(\mathcal{I}[\![\tau]\!], \mathcal{R}_\tau)\}_\tau$ une famille de relations *logiques* de Grothendieck de $\underline{\mathrm{G}}(\mathfrak{C}, K, s)$.
Pour tout terme $\Gamma \vdash t : \tau$, l'interprétation $\mathcal{I}[\![\Gamma \vdash t : \tau]\!]$ est une flèche de la catégorie des relations de Grothendieck, entre $(\mathcal{I}[\![\Gamma]\!],\ \mathcal{R}_\Gamma)$ et $(\mathcal{I}[\![\tau]\!],\ \mathcal{R}_\tau)$.

Démonstration :
La preuve découle directement de la proposition 5.7. ∎

Le résultat de normalisation repose sur le lemme suivant :

Lemme 5.10 (lemme de base) Soit $(\mathfrak{C}, K)$ un site, $s : \mathfrak{C} \to \mathfrak{S}$ un foncteur vers une biCCC, et $\mathcal{I}(.)$ une interprétation des types de base dans $\mathfrak{S}$.
Soient $\{(\mathcal{I}[\![\tau]\!], \mathcal{L}_\tau)\}_\tau$ et $\{(\mathcal{I}[\![\tau]\!], \mathcal{U}_\tau)\}_\tau$ deux familles de relations de Grothendieck de $\underline{\mathrm{G}}(\mathfrak{C}, K, s)$ indexées par les types, telles que

$$\begin{array}{ll} \mathcal{L}_0 = \bot & \mathcal{U}_1 = \top \\ \mathcal{L}_{\sigma\times\tau} \subseteq \mathcal{L}_\sigma \wedge \mathcal{L}_\tau & \mathcal{U}_\sigma \wedge \mathcal{U}_\tau \subseteq \mathcal{U}_{\sigma\times\tau} \\ \mathcal{L}_{\sigma+\tau} \subseteq \mathcal{L}_\sigma \vee \mathcal{L}_\tau & \mathcal{U}_\sigma \vee \mathcal{U}_\tau \subseteq \mathcal{U}_{\sigma+\tau} \\ \mathcal{L}_{\sigma\to\tau} \subseteq \mathcal{U}_\sigma \supset \mathcal{L}_\tau & \mathcal{L}_\sigma \supset \mathcal{U}_\tau \subseteq \mathcal{U}_{\sigma\to\tau} \end{array}$$

Soit $\{(\mathcal{I}[\![\tau]\!], \mathcal{R}_\tau)\}_\tau$ une famille de relations logiques de Grothendieck.

Si $\mathcal{L}_\theta \subseteq \mathcal{R}_\theta \subseteq \mathcal{U}_\theta$ pour tous les types de base θ, alors

$$\mathcal{L}_\tau \subseteq \mathcal{R}_\tau \subseteq \mathcal{U}_\tau \text{ pour tous les types } \tau$$

Démonstration :
La preuve se fait par induction sur les types. Les cas de base sont dans les hypothèses.

Pour des relations de Grothendieck R_A et R'_A nous avons $R_A \subseteq R'_A$ si et seulement si $\mathrm{id}_A : (A, R_A) \to (A, R'_A)$. Nous pouvons en déduire, par fonctorialité des constructeurs de types catégoriques, que si $R_A \subseteq R'_A$ et $R_B \subseteq R'_B$, alors

$$\begin{array}{c} R_A \wedge R_B \subseteq R'_A \wedge R'_B \\ R_A \vee R_B \subseteq R'_A \vee R'_B \\ R'_A \supset R_B \subseteq R_A \supset R'_B \end{array}$$

Par exemple pour la première inclusion il suffit de voir que $\pi_1 \circ id_{A\times B} = id_A \circ \pi_1$ et $\pi_2 \circ id_{A\times B} = id_B \circ \pi_2$.

Et enfin, grâce à l'hypothèse d'induction, nous avons

$$\begin{array}{c} \mathcal{L}_\sigma \wedge \mathcal{L}_\tau \subseteq \mathcal{R}_\sigma \wedge \mathcal{R}_\tau \subseteq \mathcal{U}_\sigma \wedge \mathcal{U}_\tau \\ \mathcal{L}_\sigma \vee \mathcal{L}_\tau \subseteq \mathcal{R}_\sigma \vee \mathcal{R}_\tau \subseteq \mathcal{U}_\sigma \vee \mathcal{U}_\tau \\ \mathcal{U}_\sigma \supset \mathcal{L}_\tau \subseteq \mathcal{R}_\sigma \supset \mathcal{R}_\tau \subseteq \mathcal{L}_\sigma \supset \mathcal{U}_\tau \end{array}$$

Nous voulons appliquer ce lemme de base pour obtenir un résultat de normalisation sémantique.

5.3.2 Termes neutres, termes normaux

Nous allons définir, grâce aux règles de la figure 5.1, deux sous-ensembles de l'ensemble des termes bien typés : les termes ***normaux***, qui nous donneront les formes normales canoniques, sont ceux pour lesquels le jugement $\Gamma \vdash_{\mathcal{N}} t : \tau$ est dérivable. Les termes ***neutres***, appelés ainsi car ils correspondent aux termes neutres des preuves de normalisation forte, sont ceux pour lesquels le jugement $\Gamma \vdash_{\mathcal{M}} t : \tau$ est dérivable. Notre but sera ensuite de montrer que ces ensembles vérifient les conditions du lemme de base 5.10.

Définition 5.11

1. Un contexte Γ est dit ***inconsistant*** s'il existe un terme t tel que $\Gamma \vdash t : 0$.
2. Nous noterons $\approx$ la relation d'équivalence engendrée par

$$case\,(\boldsymbol{M},\ x.\ case\,(\boldsymbol{M_1},\ x_1.\ N_1,\ x_2.\ N_2),\ y.\ N) \approx case\,(\boldsymbol{M_1},\ x_1.\ case\,(\boldsymbol{M},\ x.\ N_1,\ y.\ N),\ x_2.\ case\,(\boldsymbol{M},\ x.\ N_2,\ y.\ N))$$

$$case\,(\boldsymbol{M},\ y.\ N,\ x.\ case\,(\boldsymbol{M_1},\ x_1.\ N_1,\ x_2.\ N_2)) \approx case\,(\boldsymbol{M_1},\ x_1.\ case\,(\boldsymbol{M},\ y.\ N,\ x.\ N_1),\ x_2.\ case\,(\boldsymbol{M},\ y.\ N,\ x.\ N_2))$$

lorsque $x \notin FV(M_1)$ et $x_i \notin FV(M)$ $(i = 1, 2)$

$$\frac{N_i \approx N_i' \quad (i = 1, 2)}{case\,(M,\ x_1.\ N_1,\ x_2.\ N_2) \approx case\,(M,\ x_1.\ N_1',\ x_2.\ N_2')}$$

Les deux lemmes suivants sont cruciaux pour la démonstration des résultats qui suivront. Ils donnent une idée de l'algorithme qui peut être utilisé pour trouver la forme normale d'un terme.

Lemme 5.12

1. Pour tout terme neutre $\Gamma \vdash_{\mathcal{M}} M : \tau_1 \times \tau_2$ il existe des termes neutres $\Gamma \vdash_{\mathcal{M}} M_1 : \tau_1$ et $\Gamma \vdash_{\mathcal{M}} M_2 : \tau_2$ tels que $\Gamma \vdash proj_i\ M = M_i : \tau_i$ $(i = 1, 2)$.
2. Pour tout terme neutre $\Gamma \vdash_{\mathcal{M}} M : \tau_1 \to \tau$ et tout terme normal $\Gamma \vdash_{\mathcal{N}} N : \tau_1$, il existe un terme neutre $\Gamma \vdash_{\mathcal{M}} M' : \tau$ tel que $\Gamma \vdash M\ N = M' : \tau$.

Démonstration :

1. Soit M un terme neutre de type $\tau_1 \times \tau_2$. Nous allons faire la preuve par induction sur la profondeur de $case\,(\_,\ \_,\ \_)$.
   - Premier cas : $\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 \times \tau_2$ Soit $M_1 = proj_1\ M$. Nous avons $\Gamma \vdash_{\mathcal{M}_0} M_1 : \tau_1$, donc $\Gamma \vdash_{\mathcal{M}} M_1 : \tau_1$. Idem pour M_2.
   - Deuxième cas : $M = case\,(A,\ x_1,\ A^1)\,x_2 A^2 : \tau_1 \times \tau_2$ où $\Gamma \vdash_{\mathcal{M}_0} A : \sigma_1 + \sigma_2$ et $\Gamma,\ x_i : \sigma_i \vdash_{\mathcal{M}} A^i : \tau_1 \times \tau_2$ $(i = 1, 2)$.
     Par hypothèse d'induction sur A^i, nous savons qu'il existe des termes neutres $\Gamma \vdash_{\mathcal{M}} A_1^i : \tau_1$ et $\Gamma \vdash_{\mathcal{M}} A_2^i : \tau_2$ tels que $\Gamma \vdash proj_j\ A^i = A_j^i : \tau_j$ $(j = 1, 2)$.
     Il est facile de vérifier que $M_i = case\,(A,\ x_1.\ A_i^1,\ x_2.\ A_i^2)$ a les bonnes

$$\frac{}{\Gamma \vdash_{\mathcal{M}_0} x : \tau} \ ((x:\tau) \text{ in } \Gamma)$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 \times \tau_2}{\Gamma \vdash_{\mathcal{M}_0} proj_i \ M : \tau_i} \ (i = 1,2) \qquad \frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 \to \tau \qquad \Gamma \vdash_{\mathcal{N}_0} N : \tau_1}{\Gamma \vdash_{\mathcal{M}_0} M \ N : \tau}$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau}{\Gamma \vdash_{\mathcal{M}} M : \tau}$$

$$\frac{}{\Gamma \vdash_{\mathcal{M}} \bot_\tau : \tau} \ (\Gamma \text{ inconsistant})$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 + \tau_2 \qquad \Gamma, \ x_i : \tau_i \vdash_{\mathcal{M}} M_i : \tau \ (i = 1,2)}{\Gamma \vdash_{\mathcal{M}} case\,(M, \ x_1.\ M_1, \ x_2.\ M_2) : \tau}$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \theta}{\Gamma \vdash_{\mathcal{N}_0} M : \theta} \ (\theta \text{ type de base})$$

$$\frac{}{\Gamma \vdash_{\mathcal{N}_0} () : 1} \qquad \frac{\Gamma \vdash_{\mathcal{N}_0} N_i : \tau_i \ (i = 1,2)}{\Gamma \vdash_{\mathcal{N}_0} (N_1, \ N_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash_{\mathcal{N}_0} N : \tau_i}{\Gamma \vdash_{\mathcal{N}_0} in_i \ N : \tau_1 + \tau_2} \ (i = 1,2)$$

$$\frac{\Gamma \vdash_{\mathcal{N}_0} N : \tau}{\Gamma \vdash_{\mathcal{N}} N : \tau}$$

$$\text{gardes}(N) \stackrel{\text{def}}{=} \emptyset$$

$$\frac{\Gamma, \ x : \tau_1 \vdash_{\mathcal{N}} N : \tau}{\Gamma \vdash_{\mathcal{N}_0} \lambda x : \tau_1.\ N : \tau_1 \to \tau} \ (x \in FV(C) \text{ pour tout } C \in \text{gardes}(N)) \qquad \diamondsuit$$

$$\frac{}{\Gamma \vdash_{\mathcal{N}} \bot_\tau : \tau} \ (\Gamma \text{ inconsistant})$$

$$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 + \tau_2 \qquad \Gamma, \ x_i : \tau_i \vdash_{\mathcal{N}} N_i : \tau \ (i = 1,2)}{\Gamma \vdash_{\mathcal{N}} case\,(M, \ x_1.\ N_1, \ x_2.\ N_2) : \tau}$$

$M \notin \text{gardes}(x_1.\ N_1) \cup \text{gardes}(x_2.\ N_2)$, ♠

et $N_1 \not\approx N_2$ lorsque $x_1 \notin FV(N_1)$ et $x_2 \notin FV(N_2)$ ♣

où $\text{gardes}(x_i.\ N_i) \stackrel{\text{def}}{=} \{\ C \in \text{gardes}(N_i) \mid x_i \notin FV(C)\ \}$,

$\text{gardes}(case\,(M, \ x_1.\ N_1, \ x_2.\ N_2)) \stackrel{\text{def}}{=} \{\ M\ \} \cup \bigcup_{i=1,2} \text{gardes}(x_i.\ N_i)$

FIG. 5.1 – Termes neutres et termes normaux.
(Le contexte Γ est supposé consistant sauf mention du contraire.)

propriétés, en utilisant le fait que $proj_j\ \ case\,(A,\ x_1.\ A_i^1,\ x_2.\ A_i^2) = case\,(A,\ x_1.\ proj_j\ \ A_i^1,\ x_2.\ proj_j\ \ A_i^2)$.
- Troisième cas : $M = \bot_{\tau_1\times\tau_2}$ Prendre $M_i = \bot_{\tau_i}$. Le terme $proj_i\ \ M$ est de type τ_i dans le contexte Γ, et Γ est inconsistant, donc $\Gamma\ \vdash\ proj_i\ \ M = M_i : \tau_i$.

2. Soit M un terme neutre de type $\tau_1 \to \tau$ et N un terme normal de type τ_1. Nous allons faire la preuve par induction sur la somme des profondeurs de M et N.
   - Premier cas : $\Gamma\ \vdash_{\mathcal{M}_0}\ M : \tau_1 \to \tau$ et $\Gamma\ \vdash_{\mathcal{N}_0}\ N : \tau_1$. Prendre $M' = (M\ \ N)$.
   - Deuxième cas : $N = case\,(A,\ x_1.\ N_1,\ x_2.\ N_2)$. L'hypothèse d'induction sur N_1 et N_2 nous donne des termes neutres $A_1 = M\ \ N_1$ et $A_2 = M\ \ N_2$.
     Prenons $M' = case\,(A,\ x_1.\ A_1,\ x_2.\ A_2)$. Il est facile de vérifier que $M\ \ N = M'$.
   - Troisième cas : $N = \bot_{\tau_1}$. Le terme $(M\ \ \ N)$ est de type τ dans le contexte Γ qui est inconsistant donc $\Gamma\ \vdash\ \bot_\tau = M\ \ N : \tau$. Donc $M' = \bot_\tau$ convient.
   - Quatrième cas : $M = case\,(A,\ x_1.\ M_1,\ x_2.\ M_2)$. L'hypothèse d'induction nous donne des termes neutres $A_1 = M_1\ \ N$ et $A_2 = M_2\ \ N$.
     Prenons $M' = case\,(A,\ x_1.\ A_1,\ x_2.\ A_2)$. Il est facile de vérifier que $M\ \ N = M'$.
   - Cinquième cas : $M = \bot_{\tau_1\to\tau}$. Encore une fois $M' = \bot_\tau$ convient.

■

Lemme 5.13

1. Pour tout terme $\Gamma\ \vdash_{\mathcal{N}_1}\ C : \tau$ dérivable d'après les règles suivantes

$$\frac{\Gamma\ \vdash_{\mathcal{N}_0}\ N : \tau}{\Gamma\ \vdash_{\mathcal{N}_1}\ N : \tau}\ (\Gamma \text{ consistant}) \qquad \frac{}{\Gamma\ \vdash_{\mathcal{N}_1}\ \bot_\tau : \tau}\ (\Gamma \text{ inconsistant})$$

$$\frac{\Gamma\ \vdash_{\mathcal{M}_0}\ M : \tau_1 + \tau_2 \qquad \Gamma,\ x_i : \tau_i\ \vdash_{\mathcal{N}_1}\ C_i : \tau \quad (i = 1,2)}{\Gamma\ \vdash_{\mathcal{N}_1}\ case\,(M,\ x_1.\ C_1,\ x_2.\ C_2) : \tau}\ (\Gamma \text{ consistant})$$

   il existe un terme normal $\Gamma\ \vdash_{\mathcal{N}}\ N : \tau$ tel que $\Gamma\ \vdash\ C = N : \tau$.
2. Pour tout couple de termes normaux $\Gamma\ \vdash_{\mathcal{N}}\ N_i : \tau_i \quad (i = 1,2)$, il existe un terme normal $\Gamma\ \vdash_{\mathcal{N}}\ N : \tau_1 \times \tau_2$ tel que $\Gamma\ \vdash\ (N_1,\ N_2) = N : \tau_1 \times \tau_2$.
3. Pour tout terme normal $\Gamma\ \vdash_{\mathcal{N}}\ N : \tau_i \quad (i \in \{1,2\})$, il existe un terme normal $\Gamma\ \vdash_{\mathcal{N}}\ N' : \tau_1 + \tau_2$ tel que $\Gamma\ \vdash\ in_i\ \ N = N' : \tau_1 + \tau_2$.
4. Pour tout terme normal $\Gamma,\ x : \tau_1\ \vdash_{\mathcal{N}}\ N_1 : \tau$, il existe un terme normal $\Gamma\ \vdash_{\mathcal{N}}\ N : \tau_1 \to \tau$ tel que $\Gamma\ \vdash\ \lambda x : \tau_1.\ N_1 = N : \tau_1 \to \tau$.

Démonstration :

1. Nous allons d'abord définir une procédure Φ qui va « *aplatir* » le terme C, dans le sens où elle va construire un terme équivalent vérifiant les conditions ♠. La procédure est la suivante :
   - Si C n'a pas de gardes, $\Phi(C) = C$
   - Si

$$C \approx case\,(M,\ x_1.case\,(M,\ x_2.\ldots case\,(M,\ x_k.C_0,\ y_k.\ldots)\ldots,\ y_2.\ldots),\ y_1.\ldots)$$

avec $x_i \notin FV(M)$ $(1 \leq i \leq k)$ et $M \notin \text{gardes}(C_0)$ et

$$C' \approx case\,(M,\ y_1 \ldots,\ x'_1.case\,(M,\ y_2 \ldots,\ x'_2 \ldots case\,(M,\ y'_\ell \ldots,\ x'_\ell.C'_0)\ldots))$$

avec $x'_i \notin FV(M)$ $(1 \leq i \leq k)$ et $M \notin \text{gardes}(C'_0)$,
on pose

$$\Phi(case\,(M,\ x.C,\ x'.C')) = case \begin{pmatrix} M, \\ x.\ \Phi(C_0 \left[x/x_k\right] \ldots \left[x/x_1\right]), \\ x'.\ \Phi(C'_0 \left[x'/x'_\ell\right] \ldots \left[x'/x'_1\right] \end{pmatrix}$$

Notons qu'à chaque appel récursif de Φ, la cardinalité de l'ensemble des gardes du terme décroît, et donc que Φ est bien définie.
Nous avons donc, pour $\Gamma \vdash_{\mathcal{N}_1} C : \tau$,

$$C \approx \Phi(C)$$

où $\Phi(C)$ vérifie la condition ♠.
Maintenant, nous allons simplifier encore ce terme afin qu'il vérifie la condition ♣. Nous définissons pour cela la fonction Σ de la manière suivante :

$$\Sigma(N) = N \qquad \text{si } C \text{ n'a pas de gardes}$$

et

$$\Sigma(case\,(M,\ x.C,\ x'.C')) \quad = \quad \begin{cases} C_1 & \text{si } x \notin FV(C_1), x' \notin FV(C'_1), \text{et } C_1 \approx C'_1 \\ case\,(M,\ x.C_1,\ x'.C'_1) & \text{sinon} \end{cases}$$
$$\text{où } C_1 = \Sigma(C) \text{ et } C'_1 = \Sigma(C')$$

Nous obtenons l'égalité

$$C = \Sigma(\Phi(C))$$

où $\Sigma(\Phi(C))$ est un terme normal, ce qui termine la preuve de le première partie du lemme.

Pour finir, donnons un algorithme permettant de calculer Φ :

$$\Phi(N) = N \qquad \text{si } N \text{ n'a pas de gardes}$$

et

$$\Phi(case\,(M,\ x.C,\ x'.C')) = case \begin{pmatrix} M, \\ x.\ \Phi(\Phi_1^{M,x}(C)), \\ x.\ \Phi(\Phi_2^{M,x'}(C')) \end{pmatrix} \qquad (x, x' \notin FV(M))$$

où

$$\Phi_i^{M,x}(N) = N \qquad \text{si } N \text{ n'a pas de gardes}$$

et

$$\Phi_i^{M,x}(case\,(M',\ x_1.C_1,\ x_2.C_2)) = \begin{cases} \Phi_i^{M,x}(C_i)\left[{}^{x}/_{x_i}\right] & \text{si } M' = M \\ case\left(M',\ x_1.\Phi_i^{M,x}(C_1),\ x_2.\Phi_i^{M,x}(C_2)\right) & \text{sinon} \end{cases} \qquad (x_1, x_2 \notin FV(M))$$

De plus $\approx$ est décidable, donc la fonction Σ peut également être calculée.

2. Soient $\Gamma \vdash_{\mathcal{N}} N_1 : \tau_1$ et $\Gamma \vdash_{\mathcal{N}} N_2 : \tau_2$ deux termes normaux. Nous faisons la preuve par induction sur la somme des profondeurs des *case* des hypothèses. Nous allons d'abord contruire un terme N de $\mathcal{N}_1$ puis nous obtiendrons un terme normal en utilisant la première partie du lemme.
   – Si $\Gamma \vdash_{\mathcal{N}_0} N_1 : \tau_1$ et $\Gamma \vdash_{\mathcal{N}_0} N_2 : \tau_2$ il suffit de prendre $N = (N_1,\ N_2)$.
   – Deuxième cas : si $N_1 = case\,(M,\ x'.\ N',\ x''.\ N'')$, on prend

   $$case\left(M,\ x'.\ (N',\ N_2),\ x''.\ (N'',\ N_2)\right)$$

   (en faisant attention aux problèmes de capture de variables), puis on applique l'hypothèse d'induction à chacune des branches du *case*.
   – Troisième cas : si $N_2 = case\,(M,\ x'.\ N',\ x''.\ N'')$, on procède de la même façon.
   – Quatrième cas : si $N_1 = \bot_{\tau_1}$ (donc Γ est inconsistant). $N = \bot_{\tau_1 \times \tau_2}$ convient.
   – Cinquième cas : si $N_2 = \bot_{\tau_2}$ *idem*.
3. Soit $\Gamma \vdash_{\mathcal{N}} N : \tau_1$ un terme normal. Nous faisons la preuve par induction sur la somme des profondeurs des *case* des hypothèses.
   – Si $\Gamma \vdash_{\mathcal{N}_0} N : \tau_1$ il suffit de prendre $N' = in_1\ N$.
   – Deuxième cas : si $N = case\,(M,\ x_1.\ N_1,\ x_2.\ N_2)$, on prend

   $$N' = case\left(M,\ x_1.\ N_1',\ x_2.\ N_2'\right)$$

   (en faisant attention aux problèmes de capture de variables), où N_1' et N_2' nous sont fournis par l'hypothèse d'induction.
   – Troisième cas : si $N = \bot_{\tau_1}$ (donc Γ est inconsistant). $N' = \bot_{\tau_1 + \tau_2}$ convient.
   On procède bien sûr de la même façon pour $i = 2$.
4. Soit $\Gamma,\ x : \tau_1 \vdash_{\mathcal{N}} N_1 : \tau$ un terme normal. Posons $N' = \lambda x : \tau_1.\ N_1$. On a bien

   $$\Gamma \vdash \lambda x : \tau_1.\ N_1 = N' : \tau_1 \to \tau$$

   Mais N' ne vérifie pas nécessairement la condition $\diamondsuit$. Si elle n'est pas vérifiée, N_1 est de la forme $case\,(M,\ x.\ A,\ y.\ B)$, et l'on fait la même opération sur A et B.

■

5.3.3 Contextes contraints

Pour appliquer le lemme de base, il nous faut d'abord choisir un site. Nous allons avoir besoin de poser des contraintes sur les contextes. Pour cela, nous définissons la notion de *contexte contraint* $\Gamma \mid \Xi$ (l'intuition est que nous voulons considérer un contexte Γ uniquement s'il satisfait les contraintes dans Ξ) (voir Fiore-Simpson [46]).

Syntaxe

Définition 5.14 (contexte contraint) Les ***contextes contraints*** sont définis par les règles suivantes :

$$\frac{}{() \mid ()} \qquad \frac{\Gamma \mid \Xi}{\Gamma,\ x : \tau \mid \Xi,\ x =_\tau x}$$

$$\frac{\Gamma \mid \Xi \qquad \Gamma \vdash_{\mathcal{M}_0} M : \tau_1 + \tau_2}{\Gamma,\ x : \tau_i \mid \Xi,\ (in_i\ x) =_{\tau_1+\tau_2} M} \ (\Gamma \text{ consistant}, i = 1, 2)$$

$$\frac{\Gamma \mid \Xi \qquad \Gamma \vdash t : \tau_1 + \tau_2}{\Gamma,\ x : \tau_i \mid \Xi,\ (in_i\ x) =_{\tau_1+\tau_2} t} \ (\Gamma \text{ inconsistant}, i = 1, 2)$$

où $x \notin \mathrm{dom}(\Gamma)$.

Notons que seuls les termes neutres sont autorisés dans les contraintes de contextes consistants.

Définition 5.15 (catégorie des contextes contraints) La catégorie des contextes contraints $\mathbf{C}$ a pour objets les contextes contraints et pour morphismes $\Gamma' \mid \Xi' \to \Gamma \mid \Xi$ les ***renommages*** injectifs $\rho : \mathrm{dom}(\Gamma) \rightarrowtail \mathrm{dom}(\Gamma')$ qui préservent le typage (*i.e.* si $x : \tau \in \Gamma$, alors $\rho(x) : \tau \in \Gamma'$) et les contraintes (*i.e.* si $t =_\tau t' \in \Xi$, alors $t[\rho] =_\tau t'[\rho] \in \Xi'$).

Dans cette définition, il faut bien remarquer que le sens des flèches est inversé : un morphisme de $\Gamma' \mid \Xi'$ dans $\Gamma \mid \Xi$ est une flèche de $\mathrm{dom}(\Gamma)$ dans $\mathrm{dom}(\Gamma')$.

Définition 5.16 La famille des recouvrements $\mathsf{K}(\Gamma \mid \Xi)$ d'un contexte contraint $\Gamma \mid \Xi$ est définie par les règles suivantes :

$$\frac{}{\emptyset \in \mathsf{K}(\Gamma \mid \Xi)} \ (\Gamma \text{ inconsistant})$$

$$\frac{}{\left\{\ id_{\mathrm{dom}(\Gamma)}\ \right\} \in \mathsf{K}(\Gamma \mid \Xi)}$$

$$\frac{\left\{\ \rho_j\ \right\}_{j\in J} \cup \left\{\ \rho : \Gamma' \mid \Xi' \to \Gamma \mid \Xi\ \right\} \in \mathsf{K}(\Gamma \mid \Xi) \qquad \Gamma \vdash t : \tau_1 + \tau_2}{\left\{\ \rho_j\ \right\}_{j\in J} \cup \left\{\ \rho \circ \zeta_i : \Gamma'_i \mid \Xi'_i \to \Gamma \mid \Xi\ \right\}_{i=1,2} \in \mathsf{K}(\Gamma \mid \Xi)}$$

où, pour $i = 1, 2$, les contextes contraints $\Gamma'_i \mid \Xi'_i$ sont de la forme

$$(\Gamma',\ x'_i : \tau_i \mid \Xi',\ (in_i\ x'_i) =_{\tau_1+\tau_2} t)$$

et les renommages ζ_i sont les inclusions $\Gamma'_i \mid \Xi'_i \rightarrowtail \Gamma' \mid \Xi'$.

Proposition 5.17 Le couple $(\mathbf{C}, \mathsf{K})$ est un site.

Démonstration :

▶ Identité Par définition, $\left\{\ id_{\mathrm{dom}(\Gamma)}\ \right\} \in \mathsf{K}(\Gamma \mid \Xi)$

▶ Stabilité Soit $\Gamma \mid \Xi$ un contexte contraint, et $\left\{\ \phi_i : \Gamma_i \mid \Xi_i \to \Gamma \mid \Xi\ \right\}_{i\in I} \in \mathsf{K}(\Gamma \mid \Xi)$. Considérons un morphisme $\psi : \Gamma^* \mid \Xi^* \to \Gamma \mid \Xi$. Nous allons faire la preuve par induction sur la forme

du recouvrement :

- Si $I = \emptyset$ (recouvrement vide, Γ inconsistant) alors Γ^* est inconsistant, et le recouvrement vide convient.
- Si le recouvrement contient uniquement la flèche identité, alors le recouvrement $\left\{ id_{\mathrm{dom}(\Gamma^*)} \right\}$ de $K(\Gamma^* \mid \Xi^*)$ convient.
- Enfin si le recouvrement est obtenu par la troisième règle de la définition 5.16, notons

$$\{ \phi_i \}_{i \in I} = \{ \rho_k \}_{k \in K} \cup \{ \rho \circ \zeta_1,\ \rho \circ \zeta_2 \}$$

avec $\rho : \Gamma' \mid \Xi' \to \Gamma \mid \Xi$.
Par hypothèse d'induction, il existe un recouvrement $\left\{ \rho_j^* \right\}_{j \in J} \cup \{ \rho^* \}$ telle que toute composition $\psi \circ \rho_j^*$ se factorise par l'intermédiaire d'un ρ_i, et $\psi \circ \rho^* = \rho \circ \gamma$, où $\gamma : \Gamma^{*\prime} \mid \Xi^{*\prime} \to \Gamma' \mid \Xi'$.
Pour conclure, nous allons remarquer que le diagramme suivant commute :

$$\begin{array}{ccc}
(\Gamma^{*\prime},\ x^{*\prime}_i : \tau_i \mid \Xi^{*\prime},\ (in_i\ \ x^{*\prime}_i) =_{\tau_1+\tau_2} t) & \xrightarrow{\gamma[x'_i \mapsto x^{*\prime}_i]} & (\Gamma',\ x'_i : \tau_i \mid \Xi',\ (in_i\ \ x'_i) =_{\tau_1+\tau_2} t) \\
\Big\downarrow{\scriptstyle \zeta'_i} & & \Big\downarrow{\scriptstyle \zeta_i} \\
(\Gamma^{*\prime} \mid \Xi^{*\prime}) & \xrightarrow[\gamma]{} & (\Gamma' \mid \Xi')
\end{array}$$

(pour tout $x^{*\prime}_i \notin \mathrm{dom}(\Gamma^{*\prime})$)

▶ Transitivité Découle directement de la définition de $\mathsf{K}(\Gamma \mid \Xi)$. ■

Sémantique

Nous allons restreindre notre attention aux ***interprétations stables*** des types ; c'est-à-dire les interprétations $\mathcal{I}(.)$ des types de base dans une biCCC telles que, pour tous les couples de types $(\tau_1,\ \tau_2)$, le co-produit $\mathcal{I}[\![\tau_1]\!] + \mathcal{I}[\![\tau_2]\!]$ est stable par produits fibrés.

Définition 5.18 (co-produit stable) Dans une catégorie quelconque, un co-produit $A_1 + A_2$ est dit ***stable*** si, pour toute flèche $f : X \to A_1 + A_2$ il existe des produits fibrés X_1 et X_2

$$\begin{array}{ccc}
X_1 & \xrightarrow{x_1} & X \\
\downarrow & & \downarrow{\scriptstyle f} \\
A_1 & \xrightarrow[\iota_1]{} & A_1 + A_2
\end{array}
\qquad\qquad
\begin{array}{ccc}
X_2 & \xrightarrow{x_2} & X \\
\downarrow & & \downarrow{\scriptstyle f} \\
A_2 & \xrightarrow[\iota_2]{} & A_1 + A_2
\end{array}$$

tels que le diagramme $X_1 \xrightarrow{x_1} X \xleftarrow{x_2} X_2$ est aussi un co-produit.

Les topos élémentaires ou les algèbres de Heyting sont des exemples de catégories bi-cartésiennes fermées avec sommes stables.

Pour une interprétation stable $\mathcal{I}(.)$, nous définissons la sémantique $\mathcal{I}[\![\Gamma \mid \Xi]\!]$ d'un contexte contraint $\Gamma \mid \Xi$, par induction sur la structure des contextes. Pour les cas avec contraintes triviales, nous avons la sémantique habituelle d'un contexte comme produit des sémantiques des types des variables du contexte. Pour prendre en compte le cas des contraintes non-triviales, nous avons

besoin de définir simultanément et inductivement un monomorphisme de $\mathcal{I}[\![\Gamma \mid \Xi]\!]$ dans $\mathcal{I}[\![\Gamma]\!]$. La sémantique $\mathcal{I}[\![\Gamma \mid \Xi]\!]$ est définie comme le domaine de ce monomorphisme.

Définition 5.19 (sémantique de $\Gamma \mid \Xi$) Étant donnée une interprétation stable $\mathcal{I}(.)$ des types de base dans une biCCC, on associe à chaque contexte contraint $\Gamma \mid \Xi$ un objet $\mathcal{I}[\![\Gamma \mid \Xi]\!]$ et un monomorphisme $m_{\Gamma \mid \Xi} : \mathcal{I}[\![\Gamma \mid \Xi]\!] \rightarrowtail \mathcal{I}[\![\Gamma]\!]$ définis comme suit :

1. $m_{() \mid ()} \stackrel{\text{def}}{=} id_1 : 1 \rightarrowtail 1$.
2. $m_{\Gamma,\, x:\tau \mid \Xi,\, x=_\tau x} \stackrel{\text{def}}{=} m_{\Gamma \mid \Xi} \times \mathrm{id}_{\mathcal{I}[\![\tau]\!]} : \mathcal{I}[\![\Gamma \mid \Xi]\!] \times \mathcal{I}[\![\tau]\!] \rightarrowtail \mathcal{I}[\![\Gamma]\!] \times \mathcal{I}[\![\tau]\!]$.
3. $m_{\Gamma,\, x_i:\tau_i \mid \Xi,\, (in_i\ x_i)=_{\tau_1+\tau_2} M} \stackrel{\text{def}}{=} \langle m_{\Gamma \mid \Xi} \circ p_i, q_i \rangle$:
$\mathcal{I}[\![\Gamma,\ x_i : \tau_i \mid \Xi,\ (in_i\ x_i) =_{\tau_1+\tau_2} M]\!] \rightarrowtail \mathcal{I}[\![\Gamma]\!] \times \mathcal{I}[\![\tau_i]\!]$
où le diagramme suivant est un produit fibré :

$$\begin{array}{ccc}
\mathcal{I}[\![\Gamma,\ x_i : \tau_i \mid \Xi,\ (in_i\ x_i) =_{\tau_1+\tau_2} M]\!] & \xrightarrow{\ p_i\ } & \mathcal{I}[\![\Gamma \mid \Xi]\!] \\
\Big\downarrow{\scriptstyle q_i} & & \Big\downarrow{\scriptstyle m_{\Gamma \mid \Xi}} \\
\mathcal{I}[\![\tau_i]\!] & \xrightarrow[\ \iota_i\ ]{} & \mathcal{I}[\![\tau_1]\!] + \mathcal{I}[\![\tau_2]\!]
\end{array}
\qquad
\Big\downarrow \mathcal{I}[\![\Gamma \vdash M : \tau_1+\tau_2]\!]$$

Par stabilité, la famille

$$\left\{ \mathcal{I}[\![\Gamma,\ x_i : \tau_i \mid \Xi,\ (in_i\ x_i) =_{\tau_1+\tau_2} M]\!] \xrightarrow{\ p_i\ } \mathcal{I}[\![\Gamma \mid \Xi]\!] \right\}_{i=1,2}$$

est un co-produit, et pour tout $\Gamma \mid \Xi = (x_1 : \tau_1,\ \ldots,\ x_n : \tau_n \mid t_1 =_{\tau'_1} t'_1,\ \ldots,\ t_n =_{\tau'_n} t'_n)$, nous avons un diagramme d'égaliseur :

$$\mathcal{I}[\![\Gamma \mid \Xi]\!] \xrightarrow{m_{\Gamma \mid \Xi}} \mathcal{I}[\![\Gamma]\!] \underset{\langle \mathcal{I}[\![\Gamma \vdash t'_i : \tau'_i]\!] \rangle_{1 \le i \le n}}{\overset{\langle \mathcal{I}[\![\Gamma \vdash t_i : \tau'_i]\!] \rangle_{1 \le i \le n}}{\rightrightarrows}} \mathcal{I}[\![\tau'_1]\!] \times \ldots \times \mathcal{I}[\![\tau'_n]\!]$$

Le foncteur d'arité est défini comme dans [46] :

Définition 5.20 Le foncteur d'arité $\boldsymbol{\mathcal{I}} : \mathbf{C} \rightarrow \mathfrak{S}$ est défini de la manière suivante :

► sur les objets : $\boldsymbol{\mathcal{I}}(\Gamma \mid \Xi) \stackrel{\text{def}}{=} \mathcal{I}[\![\Gamma \mid \Xi]\!]$

► sur les morphismes : pour $\rho : \Gamma' \mid \Xi' \rightarrow \Gamma \mid \Xi$, on définit $\boldsymbol{\mathcal{I}}(\rho) : \mathcal{I}[\![\Gamma' \mid \Xi']\!] \rightarrow \mathcal{I}[\![\Gamma \mid \Xi]\!]$ comme l'unique flèche telle que

$$\begin{array}{ccc}
\mathcal{I}[\![\Gamma \mid \Xi]\!] & \xrightarrow{m_{\Gamma \mid \Xi}} & \mathcal{I}[\![\Gamma]\!] = \Pi_{x \in \mathrm{dom}(\Gamma)} \mathcal{I}[\![\Gamma(x)]\!] \\
\Big\uparrow{\scriptstyle \boldsymbol{\mathcal{I}}(\rho)} & & \Big\uparrow{\scriptstyle (\pi_{\rho(x)})_{x \in \mathrm{dom}(\Gamma)}} \\
\mathcal{I}[\![\Gamma' \mid \Xi']\!] & \xrightarrow[m_{\Gamma' \mid \Xi'}]{} & \mathcal{I}[\![\Gamma']\!] = \Pi_{x' \in \mathrm{dom}(\Gamma')} \mathcal{I}[\![\Gamma'(x')]\!]
\end{array}$$

On peut voir que la flèche $\boldsymbol{\mathcal{I}}(\rho)$ existe et est unique en utilisant la propriété universelle de l'égaliseur $m_{\Gamma \mid \Xi}$ (voir page précédente), et le fait que ρ préserve les contraintes.

5.3.4 Formes normales

Étant donnée une interprétation stable $\mathcal{I}(.)$ des types de base dans une biCCC $\mathfrak{S}$, posons :

$$\mathcal{M}_\tau(\Gamma \mid \Xi) = \left\{ \mathcal{I}[\![\Gamma \vdash M : \tau]\!] \circ m_{\Gamma \mid \Xi} \mid \Gamma \vdash_{\mathcal{M}} M : \tau \right\} \subseteq \mathfrak{S}(\mathcal{I}[\![\Gamma \mid \Xi]\!], \mathcal{I}[\![\tau]\!])$$

$$\mathcal{N}_\tau(\Gamma \mid \Xi) = \left\{ \mathcal{I}[\![\Gamma \vdash N : \tau]\!] \circ m_{\Gamma \mid \Xi} \mid \Gamma \vdash_{\mathcal{N}} N : \tau \right\} \subseteq \mathfrak{S}(\mathcal{I}[\![\Gamma \mid \Xi]\!], \mathcal{I}[\![\tau]\!])$$

Proposition 5.21 Soit $\mathcal{I}(.)$ une interprétation stable des types de base dans une biCCC. Pour tout type τ, les couples $(\mathcal{I}[\![\tau]\!],\ \mathcal{M}_\tau)$ et $(\mathcal{I}[\![\tau]\!],\ \mathcal{N}_\tau)$ sont des relations de Grothendieck de $\underline{G}(\mathbf{C}, \mathsf{K}, \boldsymbol{\mathcal{I}})$.

Démonstration :

▶ Monotonie Soit $\psi : \Gamma' \mid \Xi' \to \Gamma \mid \Xi$, et $x : \boldsymbol{\mathcal{I}}(\Gamma \mid \Xi) \to \mathcal{I}[\![\tau]\!]$. Supposons x dans $\mathcal{M}_\tau(\Gamma \mid \Xi)$. Montrons

$$x \circ \boldsymbol{\mathcal{I}}(\psi) \in \mathcal{M}_\tau(\Gamma' \mid \Xi')$$

Soit M un terme neutre tel que $x = \mathcal{I}[\![\Gamma \vdash M : \tau]\!] \circ m_{\Gamma \mid \Xi}$.
On peut vérifier facilement que

$$\mathcal{I}[\![\Gamma \vdash M : \tau]\!] \circ (\pi_{\psi(x)})_{x \in \mathrm{dom}(\Gamma)} = \mathcal{I}[\![\Gamma' \vdash M[\psi] : \tau]\!]$$

Or $x \circ \mathcal{I}[\![\psi]\!] = \mathcal{I}[\![\Gamma \vdash M : \tau]\!] \circ (\pi_{\psi(x)})_{x \in \mathrm{dom}(\Gamma)} \circ m_{\Gamma' \mid \Xi'}$ (voir définition 5.20).
D'où le résultat.

▶ Caractère local Prenons un recouvrement de $K(\Gamma \mid \Xi)$. Nous allons faire la preuve par induction sur la forme de ce recouvrement.

- Si c'est le recouvrement vide (Γ inconsistant), toute flèche $x : \mathcal{I}[\![\Gamma \mid \Xi]\!] \to \mathcal{I}[\![\tau]\!]$ est dans $\mathcal{N}_\tau(\Gamma \mid \Xi)$ et $\mathcal{M}_\tau(\Gamma \mid \Xi)$.
- Si le recouvrement ne contient que l'identité, la propriété est évidente.
- Supposons le recouvrement obtenu par le troisième règle de la définition 5.16. Soit $x : \mathcal{I}[\![\Gamma \mid \Xi]\!] \to \mathcal{I}[\![\tau]\!]$. Notons ce recouvrement $\{\ \rho_j\ \}_{j \in J} \cup \{\ \rho \circ \zeta_1,\ \rho \circ \zeta_2\ \}$ comme dans la définition 5.16.
  Supposons $x \circ \mathcal{I}[\![\rho \circ \zeta_i]\!] \in \mathcal{M}_\tau(\Gamma'_i \mid \Xi'_i)$ pour $i = 1, 2$. Il existe des termes neutres M_1 et M_2 tels que $x \circ \mathcal{I}[\![\rho \circ \zeta_i]\!] = \mathcal{I}[\![\Gamma'_i \vdash M_i : \tau]\!] \circ m_{\Gamma'_i \mid \Xi'_i}$ (pour $i = 1, 2$).
  Posons $M' = case\,(t,\ x'_1.\ M_1,\ x'_2.\ M_2)$. Nous avons
  $\mathcal{I}[\![\Gamma' \vdash M' : \tau]\!] \circ m_{\Gamma' \mid \Xi'} \circ \mathcal{I}[\![\zeta_i]\!]$

$$\begin{aligned}
&= \mathcal{I}[\![\Gamma' \vdash M' : \tau]\!] \circ (\pi_{\zeta_i(x)})_{x \in \mathrm{dom}(\Gamma')} \circ m_{\Gamma'_i \mid \Xi'_i} \\
&= \begin{pmatrix} \mathcal{I}[\![\Gamma'_1 : \tau_1 \vdash M_1 : \tau]\!] \\ \mathcal{I}[\![\Gamma'_2 : \tau_2 \vdash M_2 : \tau]\!] \end{pmatrix} \circ \delta \circ \langle id_{\mathcal{I}[\![\Gamma']\!]}, \mathcal{I}[\![\Gamma' \vdash t : \tau_1 + \tau_2]\!] \rangle \\
&\quad \circ (\pi_{\zeta_i(x)})_{x \in \mathrm{dom}(\Gamma')} \circ m_{\Gamma'_i \mid \Xi'_i} \\
&= \begin{pmatrix} \mathcal{I}[\![\Gamma'_1 : \tau_1 \vdash M_1 : \tau]\!] \\ \mathcal{I}[\![\Gamma'_2 : \tau_2 \vdash M_2 : \tau]\!] \end{pmatrix} \circ \delta
\end{aligned}$$

$$
\begin{aligned}
& \circ\langle(\pi_{\zeta_i(x)}) \circ m_{\Gamma_i' \mid \Xi_i'},\ \mathcal{I}[\![\Gamma_i' \ \vdash\ \boldsymbol{t}[\zeta_i] : \tau_1 + \tau_2]\!] \circ m_{\Gamma_i' \mid \Xi_i'}\rangle \\
= \ & \begin{pmatrix} \mathcal{I}[\![\Gamma_1' : \tau_1 \ \vdash\ M_1 : \tau]\!] \\ \mathcal{I}[\![\Gamma_2' : \tau_2 \ \vdash\ M_2 : \tau]\!] \end{pmatrix} \circ\ \delta \\
& \circ\langle(\pi_{\zeta_i(x)}) \circ m_{\Gamma_i' \mid \Xi_i'},\ \mathcal{I}[\![\Gamma_i' \ \vdash\ (\boldsymbol{in_i}\ \ \boldsymbol{x_i'}) : \tau_1 + \tau_2]\!] \circ m_{\Gamma_i' \mid \Xi_i'}\rangle \\
& \text{car } m_{\Gamma_i' \mid \Xi_i'} \text{ est un égaliseur de } \mathcal{I}[\![\Gamma_i' \ \vdash\ (\boldsymbol{in_i}\ \ \boldsymbol{x_i'}) : \tau_1 + \tau_2]\!] \text{ et } \mathcal{I}[\![\Gamma_i' \ \vdash\ \boldsymbol{t}[\zeta_i] : \tau_1 + \tau_2]\!] \\
= \ & \mathcal{I}[\![\Gamma_i' : \tau_i \ \vdash\ M_i : \tau]\!] \circ m_{\Gamma_i' \mid \Xi_i'} \\
= \ & x \circ \mathcal{I}[\![\rho]\!] \circ \mathcal{I}[\![\zeta_i]\!]
\end{aligned}
$$

Or ζ_1 et ζ_2 sont les injections d'un co-produit donc

$$\mathcal{I}[\![\Gamma' \ \vdash\ M' : \tau]\!] \circ m_{\Gamma' \mid \Xi'} = x \circ \mathcal{I}[\![\rho]\!]$$

D'où l'on peut déduire que $x \circ \mathcal{I}[\![\rho]\!] \in \mathcal{M}_\tau(\Gamma' \mid \Xi')$.
On applique ensuite l'hypothèse d'induction pour déduire que $x \in \mathcal{M}_\tau(\Gamma \mid \Xi)$.

Le cas des termes normaux est identique.

■

Théorème 5.22 Les relations de Grothendieck des termes neutres et normaux vérifient les propriétés de clôture suivantes :

$$
\begin{array}{ll}
\mathcal{M}_0 = \bot & \mathcal{N}_1 = \top \\
\mathcal{M}_{\sigma\times\tau} \subseteq \mathcal{M}_\sigma \wedge \mathcal{M}_\tau & \mathcal{N}_\sigma \wedge \mathcal{N}_\tau \subseteq \mathcal{N}_{\sigma\times\tau} \\
\mathcal{M}_{\sigma+\tau} \subseteq \mathcal{M}_\sigma \vee \mathcal{M}_\tau & \mathcal{N}_\sigma \vee \mathcal{N}_\tau \subseteq \mathcal{N}_{\sigma+\tau} \\
\mathcal{M}_{\sigma\to\tau} \subseteq \mathcal{N}_\sigma \supset \mathcal{M}_\tau & \mathcal{M}_\sigma \supset \mathcal{N}_\tau \subseteq \mathcal{N}_{\tau^\sigma}
\end{array}
$$

et

$$\mathcal{M}_\theta \subseteq \mathcal{N}_\theta \qquad (\theta \text{ type de base})$$

Ce théorème sera prouvé en détails dans la section 5.4. Il nous permet d'appliquer le lemme de base, qui nous montre, en posant $\mathcal{R}_\theta = \mathcal{M}_\theta$ pour les types de base θ, que pour tout type τ

$$\mathcal{M}_\tau \subseteq \mathcal{R}_\tau \subseteq \mathcal{N}_\tau$$

Étant donné un contexte contraint $\Gamma \mid \Xi$, tout morphisme de $\mathcal{R}_\tau(\Gamma \mid \Xi)$ est donc définissable par un terme normal de type τ dans le contexte Γ. Pour en conclure que tout terme bien typé est égal à un terme en forme normale, nous devons encore montrer que $\mathcal{R}_\tau(\Gamma \mid \Xi)$ contient l'interprétation de tous les termes bien typés de type τ dans le contexte Γ. Enfin, il nous faudra montrer que le terme obtenu est bien $\beta\eta$-équivalent au terme initial.

Résultat sémantique

Pour $\Gamma = (x_1 : \tau_1,\ \ldots,\ x_n : \tau_n)$ $(1 \le i \le n)$, la projection $\pi_i = \mathcal{I}[\![\Gamma \ \vdash\ x_i : \tau_i]\!] : \mathcal{I}[\![\Gamma]\!] \to \mathcal{I}[\![\tau_i]\!]$ est un morphisme neutre de $\mathcal{M}_{\tau_i}(\Gamma \mid \Delta_\Gamma)$ où $\Delta_\Gamma \stackrel{\text{def}}{=} (x_1 =_{\tau_1} x_1,\ \ldots,\ x_n =_{\tau_n} x_n)$ (on a $\mathcal{I}[\![\Gamma \mid \Delta_\Gamma]\!] = \mathcal{I}[\![\Gamma]\!]$).
C'est donc également un morphisme de $\mathcal{R}_{\tau_i}(\Gamma \mid \Delta_\Gamma)$ (car $\mathcal{M}_{\tau_i} \subseteq \mathcal{R}_{\tau_i}$).

Posons $g = \langle \pi_1, \ldots, \pi_n \rangle$. Comme $\mathcal{R}_\Gamma = \mathcal{R}_{\tau_1} \wedge \ldots \wedge \mathcal{R}_{\tau_n}$, nous avons $g = id_{\mathcal{I}[\![\Gamma]\!]} \in \mathcal{R}_\Gamma(\Gamma \mid \Delta_\Gamma)$.

D'après le lemme fondamental, $\mathcal{I}[\![\Gamma \vdash t : \tau]\!]$ est une flèche de la catégorie des relations logiques de Grothendieck entre $(\mathcal{I}[\![\Gamma]\!], \mathcal{R}_\Gamma)$ et $(\mathcal{I}[\![\tau]\!], \mathcal{R}_\tau)$, et donc (en appliquant la définition des morphismes de cette catégorie)

$$\mathcal{I}[\![\Gamma \vdash t : \tau]\!] \circ g = \mathcal{I}[\![\Gamma \vdash t : \tau]\!] \in \mathcal{R}_\tau(\Gamma \mid \Delta_\Gamma)$$

En appliquant l'inclusion $\mathcal{R}_\tau \subseteq \mathcal{N}_\tau$, nous avons donc le corollaire suivant :

Corollaire 5.23 Soit $\mathcal{I}(.)$ une interprétation stable des types de base dans une biCCC $\mathfrak{S}$. Pour tout terme $\Gamma \vdash t : \tau$, il existe un terme normal $\Gamma \vdash_{\mathcal{N}} N : \tau$ tel que $\mathcal{I}[\![\Gamma \vdash t : \tau]\!] = \mathcal{I}[\![\Gamma \vdash N : \tau]\!] : \mathcal{I}[\![\Gamma]\!] \to \mathcal{I}[\![\tau]\!]$ dans $\mathfrak{S}$.

Résultat syntaxique

Il nous reste à montrer que le terme normal N que nous avons trouvé avec la même sémantique que le terme t de départ, est vraiment *égal* à t dans la théorie équationnelle $\beta\eta$. Pour ce faire, nous allons *essentiellement* prendre comme catégorie sémantique la biCCC libre (ou de manière équivalente le modèle des λ-termes). Dans cette catégorie, l'interprétation de t est la classe d'équivalence $[t]$ de t modulo $\beta\eta$. Le fait que $[t]$ soit « définissable » par un N en forme normale signifie juste que l'interprétation de N est $[t]$, c'est-à-dire que $N = t$ dans la théorie équationnelle $\beta\eta$.

Cependant le modèle des λ-termes n'a pas de sommes stables. Mais Marcelo Fiore a montré que l'on pouvait le plonger dans un modèle avec sommes stables, qui convient pour notre preuve[2].

5.4 Lemmes utilisés

Cette section contient la preuve du théorème 5.22.

Lemme 5.24 $\mathcal{M}_0 = \bot$.

Démonstration :

Par définition nous avons

$$\mathcal{M}_0(\Gamma \mid \Xi) = \left\{ \mathcal{I}[\![\Gamma \vdash M : 0]\!] \circ m_{\Gamma \mid \Xi} \mid \Gamma \vdash_{\mathcal{M}} M : 0 \right\}$$

Donc si Γ est consistant, $\mathcal{M}_0(\Gamma \mid \Xi) = \emptyset$ et la famille vide n'est pas un recouvrement de $\mathsf{K}(\Gamma \mid \Xi)$, donc $\mathcal{M}_0(\Gamma \mid \Xi) = \bot(\Gamma \mid \Xi)$.
Enfin si Γ est inconsistant $\mathcal{I}[\![\Gamma \vdash M : 0]\!] \circ m_{\Gamma \mid \Xi}$ est une flèche entre $\mathcal{I}[\![\Gamma \mid \Xi]\!]$ et $\mathfrak{o}$. Réciproquement, soit $f : \mathcal{I}[\![\Gamma \mid \Xi]\!] \to \mathfrak{o}$. Le recouvrement vide appartient à $\mathsf{K}(\Gamma \mid \Xi)$ donc d'après le caractère local des relations de Grothendieck $f \in \mathcal{M}_0(\Gamma \mid \Xi)$. On a donc donc bien $\mathcal{M}_0(\Gamma \mid \Xi) = \bot(\Gamma \mid \Xi)$. ∎

Lemme 5.25 $\mathcal{N}_1 = \top$.

[2] Communication personnelle – référence à venir

Démonstration :

Par définition, nous avons

$$\mathcal{N}_1(\Gamma \mid \Xi) = \left\{ \mathcal{I}[\![\Gamma \vdash N : 1]\!] \circ m_{\Gamma \mid \Xi} \mid \Gamma \vdash_{\mathcal{N}} N : 1 \right\}$$

Or $\mathcal{I}[\![\Gamma \vdash N : 1]\!] \circ m_{\Gamma \mid \Xi}$ est une flèche entre $\mathcal{I}[\![\Gamma \mid \Xi]\!]$ et $\mathcal{I}[\![1]\!] = 1$ donc par unicité c'est $!\mathcal{I}(\Gamma \mid \Xi)$. ■

Lemme 5.26 $\mathcal{M}_{\tau_1 \times \tau_2} \subseteq \mathcal{M}_{\tau_1} \wedge \mathcal{M}_{\tau_2}$.

Démonstration :

Par définition de $\mathcal{M}_{\tau_1 \times \tau_2}$, nous avons $f \in \mathcal{M}_{\tau_1 \times \tau_2}(\Gamma \mid \Xi)$ si et seulement si

$$f = \mathcal{I}[\![\Gamma \vdash M : \tau_1 \times \tau_2]\!] \circ m_{\Gamma \mid \Xi} \text{ pour un terme neutre } \Gamma \vdash_{\mathcal{M}} M : \tau_1 \times \tau_2.$$

Par propriété des paires surjectives, et d'après la définition de la fonction d'interprétation, nous avons

$$f = \langle \mathcal{I}[\![\Gamma \vdash \pi_1(M) : \tau_1]\!], \mathcal{I}[\![\Gamma \vdash \pi_2(M) : \tau_2]\!] \rangle \circ m_{\Gamma \mid \Xi}$$

qui est égale, d'après le lemme 5.12 (1), à

$$\langle \mathcal{I}[\![\Gamma \vdash M_1 : \tau_1]\!], \mathcal{I}[\![\Gamma \vdash M_2 : \tau_2]\!] \rangle \circ m_{\Gamma \mid \Xi}$$

pour un terme neutre $\Gamma \vdash_{\mathcal{M}} M_i : \tau_1$ $(i = 1, 2)$.

Donc, pour $i = 1, 2$, nous avons $\pi_i(f) \in \mathcal{M}_{\tau_i}(\Gamma \mid \Xi)$ et donc $f \in (\mathcal{M}_{\tau_1} \wedge \mathcal{M}_{\tau_2})(\Gamma \mid \Xi)$. ■

Lemme 5.27 $\mathcal{N}_{\tau_1} \wedge \mathcal{N}_{\tau_2} \subseteq \mathcal{N}_{\tau_1 \times \tau_2}$.

Démonstration :

Par définition de $\mathcal{N}_{\tau_1} \wedge \mathcal{N}_{\tau_2}$, nous avons $f \in (\mathcal{N}_{\tau_1} \wedge \mathcal{N}_{\tau_2})(\Gamma \mid \Xi)$ si et seulement si, pour $i = 1, 2$,

$$\pi_i \circ f = \mathcal{I}[\![\Gamma \vdash N_i : \tau_i]\!] \circ m_{\Gamma \mid \Xi} \text{ pour des termes } \Gamma \vdash_{\mathcal{N}} N_i : \tau_i.$$

Par la propriété des paires surjectives et la définition de la fonction d'interprétation, nous avons

$$f = \mathcal{I}[\![\Gamma \vdash (N_1, N_2) : \tau_1 \times \tau_2]\!] \circ m_{\Gamma \mid \Xi}$$

qui, d'après le lemme 5.13 (2), est égal à

$$\mathcal{I}[\![\Gamma \vdash N : \tau_1 \times \tau_2]\!] \circ m_{\Gamma \mid \Xi}$$

pour un terme normal $\Gamma \vdash_{\mathcal{N}} N : \tau_1 \times \tau_2$. ■

Lemme 5.28 $\mathcal{M}_{\tau_1 + \tau_2} \subseteq \mathcal{M}_{\tau_1} \vee \mathcal{M}_{\tau_2}$.

Démonstration :

Supposons que $f \in \mathcal{M}_{\tau_1 + \tau_2}(\Gamma \mid \Xi)$. Par définition, nous avons

$$f = \mathcal{I}[\![\Gamma \vdash M : \sigma_1 + \sigma_2]\!] \circ m_{\Gamma \mid \Xi} \text{ pour un terme } \Gamma \vdash_{\mathcal{M}} M : \tau_1 + \tau_2.$$

Nous voulons montrer qu'il existe un recouvrement $\{ \rho_i : \Gamma_i \mid \Xi_i \to \Gamma \mid \Xi \}_{i \in I} \in \mathsf{K}(\Gamma \mid \Xi)$ tel que, pour tout $i \in I$, il existe $j_i \in \{ 1, 2 \}$ et $f_i \in \mathcal{M}_{\tau_{j_i}}(\Gamma_i \mid \Xi_i)$, tels que $f \circ \boldsymbol{\mathcal{I}}(\rho_i) = \iota_{j_i} \circ f_i$.

Nous allons procéder par induction sur la dérivation de $\Gamma \vdash_{\mathcal{M}} M : \tau_1 + \tau_2$.

► Cas de base :

1. $$\frac{}{\Gamma \vdash_{\mathcal{M}} \bot_\tau : \tau} \text{ (Γ inconsistant)}$$

   Nous avons $\emptyset \in \mathsf{K}(\Gamma \mid \Xi)$ et donc la propriété est démontrée.

2. $$\frac{\Gamma \vdash_{\mathcal{M}_0} M : \tau_1 + \tau_2}{\Gamma \vdash_{\mathcal{M}} M : \tau_1 + \tau_2}$$

   Pour le recouvrement d'inclusions

   $$\{ \rho_i : \Gamma_i \mid \Xi_i \to \Gamma \mid \Xi \}_{i=1,2}$$

   où $\Gamma_i \mid \Xi_i = (\Gamma,\ x_i : \tau_i \mid \Xi,\ (in_i\ \ x_i) =_{\tau_1+\tau_2} M)$ nous montrons que, pour $i = 1, 2$, le diagramme suivant commute :

   $$\begin{array}{ccc}
   \mathcal{I}[\![\Gamma_i \mid \Xi_i]\!] & \xrightarrow{\boldsymbol{\mathcal{I}}(\rho_1)} & \mathcal{I}[\![\Gamma \mid \Xi]\!] \\
   \downarrow{\scriptstyle m_{\Gamma_i \mid \Xi_i}} & & \downarrow{\scriptstyle m_{\Gamma \mid \Xi}} \\
   \mathcal{I}[\![\Gamma]\!] \times \mathcal{I}[\![\tau_i]\!] & & \mathcal{I}[\![\Gamma]\!] \\
   \downarrow{\scriptstyle \pi_2} & & \downarrow{\scriptstyle \mathcal{I}[\![\Gamma \vdash M : \tau_1+\tau_2]\!]} \\
   \mathcal{I}[\![\tau_i]\!] & \xrightarrow[\iota_i]{} & \mathcal{I}[\![\tau_1]\!] + \mathcal{I}[\![\tau_2]\!]
   \end{array}$$

   où $\pi_2 \circ m_{\Gamma_i \mid \Xi_i} = \mathcal{I}[\![\Gamma,\ x_i : \tau_i \ \vdash\ x_i : \tau_i]\!] \circ m_{\Gamma_i \mid \Xi_i} \in \mathcal{M}_{\tau_i}(\Gamma_i \mid \Xi_i)$.
   Remarquons tout d'abord que, par définition de l'interprétation de l'inclusion de contexte ρ_i (voir définition 5.20), le diagramme suivant commute :

   $$\begin{array}{ccc}
   \mathcal{I}[\![\Gamma_i \mid \Xi_i]\!] & \xrightarrow{\boldsymbol{\mathcal{I}}(\rho_i)} & \mathcal{I}[\![\Gamma \mid \Xi]\!] \\
   \downarrow{\scriptstyle m_{\Gamma_i \mid \Xi_i}} & & \downarrow{\scriptstyle m_{\Gamma \mid \Xi}} \\
   \mathcal{I}[\![\Gamma]\!] \times \mathcal{I}[\![\tau_i]\!] & \xrightarrow[\pi_1]{} & \mathcal{I}[\![\Gamma]\!]
   \end{array}$$

   Il est donc suffisant de montrer que le diagramme qui suit commute :

   $$\begin{array}{ccc}
   \mathcal{I}[\![\Gamma_i \mid \Xi_i]\!] & & \\
   \downarrow{\scriptstyle m_{\Gamma_i \mid \Xi_i}} & & \\
   \mathcal{I}[\![\Gamma]\!] \times \mathcal{I}[\![\tau_i]\!] & \xrightarrow[\pi_1]{} & \mathcal{I}[\![\Gamma]\!] \\
   \downarrow{\scriptstyle \pi_2} & + & \downarrow{\scriptstyle \mathcal{I}[\![\Gamma \vdash M : \tau_1+\tau_2]\!]} \\
   \mathcal{I}[\![\tau_i]\!] & \xrightarrow[\iota_i]{} & \mathcal{I}[\![\tau_1]\!] + \mathcal{I}[\![\tau_2]\!]
   \end{array}$$

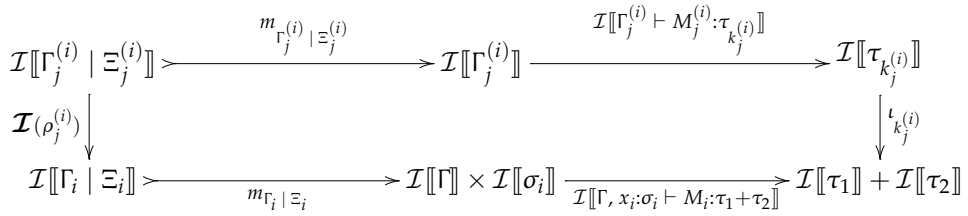
   ce qui est vrai puisque $m_{\Gamma_i \mid \Xi_i}$ égalise $\mathcal{I}[\![\Gamma,\ x_i : \tau_i \ \vdash\ in_i\ \ x_i : \tau_1 + \tau_2]\!] = \iota_1 \circ \pi_2$ et

$\mathcal{I}[\![\Gamma,\ x_i : \tau_i \ \vdash\ M : \tau_1 + \tau_2]\!] = \mathcal{I}[\![\Gamma \ \vdash\ M : \tau_1 + \tau_2]\!] \circ \pi_1$, voir (5.3.3).

► Étape d'induction :

$$\frac{\Gamma \ \vdash_{\mathcal{M}_0}\ M : \sigma_1 + \sigma_2 : \qquad \Gamma,\ x_i : \sigma_i \ \vdash_{\mathcal{M}}\ M_i : \tau_1 + \tau_2\ (i = 1, 2)}{\Gamma \ \vdash_{\mathcal{M}}\ case\,(M,\ x_1.\ M_1,\ x_2.\ M_2) : \tau_1 + \tau_2}$$

Pour $i = 1, 2$, posons $\Gamma_i \mid \Xi_i = (\Gamma,\ x_i : \sigma_i \mid \Xi,\ in_i\ \ x_i =_{\sigma_1+\sigma_2} M)$. Par induction, pour $i = 1, 2$, il existe des recouvrements $\left\{ \rho_j^{(i)} : \Gamma_j^{(i)} \mid \Xi_j^{(i)} \to \Gamma_i \mid \Xi_i \right\}_{j \in J^{(i)}} \in \mathsf{K}$ tels que, pour tout $j \in J^{(i)}$, le diagramme

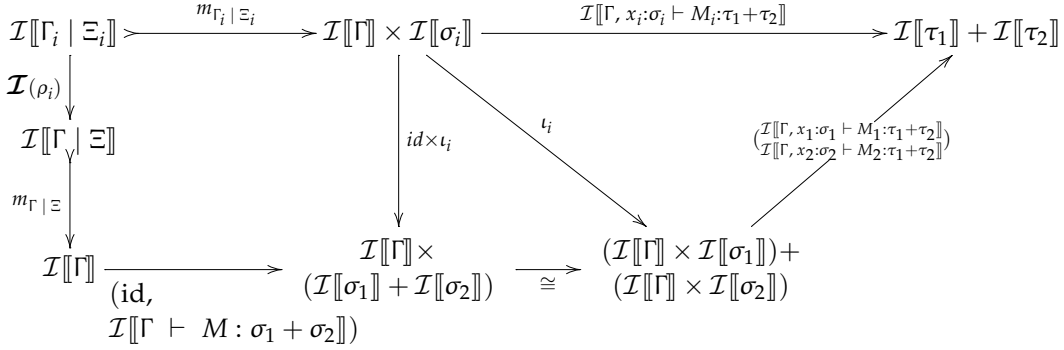


commute pour un terme neutre $\Gamma_j^{(i)} \ \vdash_{\mathcal{M}}\ M_j^{(i)} : \tau_{k_j^{(i)}}$.

Donc, pour le recouvrement

$$\left\{ \rho_i \circ \rho_j^{(i)} : \Gamma_j^{(i)} \mid \Xi_j^{(i)} \to \Gamma \mid \Xi \right\}_{i \in \{1,2\}, j \in J^{(i)}}$$

où $\rho_i : \Gamma_i \mid \Xi_i \to \Gamma \mid \Xi$ est l'inclusion, nous sommes dans la situation suivante :



En mettant ensemble les deux diagrammes, le lemme est prouvé. ■

Lemme 5.29 $\mathcal{N}_{\tau_1} \vee \mathcal{N}_{\tau_2} \subseteq \mathcal{N}_{\tau_1+\tau_2}$.

Démonstration :

Soit f dans $(\mathcal{N}_{\tau_1} \vee \mathcal{N}_{\tau_2})(\Gamma \mid \Xi)$. Il existe un recouvrement $\{ \rho_i : \Gamma_i \mid \Xi_i \to \Gamma \mid \Xi \}_{i \in I} \in \mathsf{K}(\Gamma \mid \Xi)$ tel que pour tout i de I, il existe $j_i \in \{ 1, 2 \}$ et un terme normal $\Gamma_i \ \vdash_{\mathcal{N}}\ N_i : \tau_{j_i}$ pour lequel $f \circ \boldsymbol{\mathcal{I}}(\rho_i) = \iota_{j_i} \circ \mathcal{I}[\![\Gamma_i \ \vdash\ N_i : \tau_{j_i}]\!] \circ m_{\Gamma_i \mid \Xi_i}$.

Si $I = \emptyset$, alors Γ est inconsistant et $f = \mathcal{I}[\![\Gamma \ \vdash\ \ \perp_{\tau_1+\tau_2} : \tau_1 + \tau_2]\!] \circ m_{\Gamma \mid \Xi}$ est dans $\mathcal{N}_{\tau_1+\tau_2}(\Gamma \mid \Xi)$. Sinon, pour $i \in I$, en utilisant le lemme 5.13 (3), il existe un terme normal $\Gamma_i \vdash_{\mathcal{N}} N_i' : \tau_1 + \tau_2$ tel que $\Gamma_i \ \vdash\ \ N_i' = \iota_{j_i}(N_i) : \tau_1 + \tau_2$; donc, $f \circ \boldsymbol{\mathcal{I}}(\rho_i) = \mathcal{I}[\![\Gamma_i \ \vdash$

$N_i' : \tau_1 + \tau_2]\!] \circ m_{\Gamma_i \mid \Xi_i}$ est dans $\mathcal{N}_{\tau_1+\tau_2}(\Gamma_i \mid \Xi_i)$, et par caractère local, f est dans $\mathcal{N}_{\tau_1+\tau_2}(\Gamma \mid \Xi)$. ■

Lemme 5.30 $\mathcal{M}_{\tau_1\to\tau} \subseteq \mathcal{N}_{\tau_1} \supset \mathcal{M}_\tau$.

Démonstration :

Soit f une flèche de $\mathcal{M}_{\tau_1\to\tau}(\Gamma \mid \Xi)$; donc, par définition, il existe un terme neutre $\Gamma \vdash_{\mathcal{M}} M : \tau_1 \to \tau$ tel que

$$f = \mathcal{I}[\![\Gamma \vdash M : \tau_1 \to \tau]\!] \circ m_{\Gamma \mid \Xi}$$

Pour montrer que $f \in (\mathcal{N}_{\tau_1} \supset \mathcal{M}_\tau)(\Gamma \mid \Xi)$, nous devons montrer que, pour toute injection de contexte $\rho : \Gamma' \mid \Xi' \to \Gamma \mid \Xi$ dans $\mathbf{C}$, et pour toute flèche $a \in \mathcal{N}_{\tau_1}(\Gamma' \mid \Xi')$, nous avons

$$eval \circ (f \circ \boldsymbol{\mathcal{I}}(\rho), a) \in \mathcal{M}_\tau(\Gamma' \mid \Xi') \quad .$$

Mais $f \circ \boldsymbol{\mathcal{I}}(\rho)$ est égal à $\mathcal{I}[\![\Gamma \vdash M : \tau_1 \to \tau]\!] \circ m_{\Gamma \mid \Xi} \circ \boldsymbol{\mathcal{I}}(\rho)$, et, d'après la définition 5.20 page 130 et les remarques suivantes, c'est égal à $\mathcal{I}[\![\Gamma' \vdash M[\rho] : \tau_1 \to \tau]\!] \circ m_{\Gamma' \mid \Xi'}$ qui est dans $\mathcal{M}_{\tau_1\to\tau}(\Gamma' \mid \Xi')$. Ensuite, par définition de $\mathcal{N}_{\tau_1}(\Gamma' \mid \Xi')$, il existe un terme normal $\Gamma' \vdash_{\mathcal{N}} N : \tau_1$ tel que

$$a = \mathcal{I}[\![\Gamma' \vdash N : \tau_1]\!] \circ m_{\Gamma' \mid \Xi'} \quad .$$

Donc,

$$\begin{aligned} eval \circ (f \circ \boldsymbol{\mathcal{I}}(\rho), a) &= eval \circ (\mathcal{I}[\![\Gamma' \vdash M[\rho] : \tau_1 \to \tau]\!], \mathcal{I}[\![\Gamma' \vdash N : \tau_1]\!]) \circ m_{\Gamma' \mid \Xi'} \\ &= \mathcal{I}[\![\Gamma' \vdash (M[\rho]\ N) : \tau]\!] \circ m_{\Gamma' \mid \Xi'} \end{aligned}$$

et comme, d'après le lemme 5.12 (2), il existe un terme neutre $\Gamma' \vdash_{\mathcal{M}} M' : \tau$ tel que $\Gamma' \vdash M' = (M[\rho]\ N) : \tau$, la preuve est finie. ■

Lemme 5.31 $\mathcal{M}_{\tau_1} \supset \mathcal{N}_\tau \subseteq \mathcal{N}_{\tau_1\to\tau}$.

Démonstration :

Soir $f \in (\mathcal{M}_{\tau_1} \supset \mathcal{N}_\tau)(\Gamma \mid \Xi)$. Par définition, f est tel que pour tout renommage de contexte $\rho : \Gamma' \mid \Xi' \to \Gamma \mid \Xi$ de $\mathbf{C}$, et pour tout $a \in \mathcal{M}_{\tau_1}(\Gamma' \mid \Xi')$, nous avons

$$eval \circ (f \circ \boldsymbol{\mathcal{I}}(\rho), a) \in \mathcal{N}_\tau(\Gamma' \mid \Xi') \quad .$$

Nous devons montrer que f est dans $\mathcal{N}_{\tau_1\to\tau}(\Gamma \mid \Xi)$; autrement dit qu'il existe un terme normal $\Gamma \vdash_{\mathcal{N}} N : \tau_1 \to \tau$ tel que $f = \mathcal{I}[\![\Gamma \vdash N : \sigma \to \tau]\!] \circ m_{\Gamma \mid \Xi}$.

Pour l'extension de contexte donnée par l'inclusion $\rho : (\Gamma,\ x : \tau_1 \mid \Xi, x =_{\tau_1} x) \to \Gamma \mid \Xi$, et pour $\mathcal{I}[\![\Gamma,\ x : \tau_1 \vdash x : \tau_1]\!] \circ m_{(\Gamma,\ x:\tau_1 \mid \Xi,\ x=_{\tau_1} x)} = \pi_2 \circ (m_{\Gamma \mid \Xi} \times \mathcal{I}[\![\tau_1]\!]) = \pi_2$ dans $\mathcal{M}_{\tau_1}(\Gamma,\ x : \tau_1 \mid \Xi,\ x =_{\tau_1} x)$, nous avons

$$eval \circ \langle f \circ \boldsymbol{\mathcal{I}}(\rho), \pi_2 \rangle \in \mathcal{N}_\tau(\Gamma,\ x : \tau_1 \mid \Xi,\ x =_{\tau_1} x)$$

Donc, comme $\boldsymbol{\mathcal{I}}(\rho) = \pi_1$, nous pouvons déduire $eval \circ (f \times id_{\mathcal{I}[\![\tau_1]\!]}) = \mathcal{I}[\![\Gamma,\ x : \tau_1 \vdash_{\mathcal{N}} N : \tau]\!] \circ (m_{\Gamma \mid \Xi} \times \mathrm{id}_{\mathcal{I}[\![\tau_1]\!]})$ pour un terme normal $\Gamma,\ x : \tau_1 \vdash_{\mathcal{N}} N : \tau$. Donc,

$$f = \Lambda(\mathcal{I}[\![\Gamma,\ x : \tau_1 \vdash N : \tau]\!]) \circ m_{\Gamma \mid \Xi} = \mathcal{I}[\![\Gamma \vdash \lambda x : \tau_1.\, N : \tau_1 \to \tau]\!] \circ m_{\Gamma \mid \Xi} \quad .$$

D'après le lemme 5.13 (4), il existe un terme normal $\Gamma \vdash_{\mathcal{N}} N' : \tau_1 \to \tau$ tel que $\Gamma \vdash N' = \lambda x : \tau_1.\, N : \tau_1 \to \tau$; donc $f = \mathcal{I}[\![\Gamma \vdash N' : \tau_1 \to \tau]\!] \circ m_{\Gamma \mid \Xi}$ est dans $\mathcal{N}_{\tau_1 \to \tau}(\Gamma \mid \Xi)$, ce qui montre le résultat. ■

Lemme 5.32 $\mathcal{M}_\theta \subseteq \mathcal{N}_\theta$ pour θ type de base.

Démonstration :

C'est une conséquence directe des règles. ■

Chapitre 6

Normalisation par évaluation du λ-calcul avec somme

Résumé

Ce chapitre décrit comment il est possible d'appliquer la technique dite de normalisation par évaluation *et plus particulièrement d'*évaluation partielle dirigée par les types *pour normaliser des λ-termes, et optimiser des programmes Caml. Je vais présenter ici ces techniques et décrire une implantation que j'ai réalisée en Objective Caml.*

DANS LE CHAPITRE PRÉCÉDENT, nous avons prouvé que tout terme du λ-calcul avec somme et zéro était $\beta\eta$-équivalent à un terme normal, mais sans donner de moyen effectif de réaliser cette normalisation. Nous avons vu dans le chapitre 2 qu'il n'existait pas de méthode de réduction satisfaisante basée sur la réécriture. Nous allons donc utiliser une autre technique de normalisation connue sous le nom de ***normalisation par évaluation*** [▷ *normalisation by evaluation*] (On utilisera l'abréviation ***NBE***).

Dans ce chapitre, je décrirai cette méthode et je montrerai comment Olivier Danvy l'a utilisée pour faire de l'évaluation partielle dans des langages de la famille ML, en utilisant des opérateurs de contrôle pour traiter le cas des types sommes. L'algorithme obtenu est appelé ***évaluation partielle dirigée par les types*** [▷ *type-directed partial evaluation*] (souvent dans la suite ***TDPE***). Je décrirai ensuite mon implantation en *Objective Caml*, qui utilise des techniques de génération de code au moment de l'exécution pour normaliser *in situ* des valeurs *Caml* compilées.

L'adaptation de ces techniques au problème des isomorphismes de types sera étudiée au chapitre suivant.

6.1 Normalisation par évaluation

Les langages fonctionnels comme *Objective Caml* implantent ce que l'on appelle la ***β-réduction faible***, ce qui veut dire qu'ils ne réduisent pas le corps des abstractions tant qu'elles ne sont

pas appliquées à des paramètres. Le résultat obtenu est dit en ***forme normale de tête faible***[1]. Un langage permettant des effets de bord[2] doit par exemple avoir cette stratégie faible pour que ces effets se produisent au moment voulu. On appelle ***valeur*** un terme en forme normale de tête faible.

La ***normalisation forte***, qui consiste à mettre un terme sous une forme ne contenant plus aucun redex, se heurte aussi au problème de non-terminaison possible dans beaucoup de systèmes, dont les langages de programmation (et non-décidables). Cependant, elle est utile dans bien des domaines. En théorie de la preuve, elle est utilisée par exemple pour obtenir des preuves sans coupures. En programmation, elle peut servir à faire de l'***évaluation partielle***, technique que je détaillerai plus loin, servant à spécialiser une fonction pour des valeurs données de certains de ses arguments. Dans le monde des isomorphismes de types, elle est utile pour vérifier qu'une fonction est bien un isomorphisme, comme nous l'avons fait manuellement au chapitre 4, mais une méthode automatique de normalisation pourrait être utile également pour simplifier des fonctions de conversion entre types, générées automatiquement.

Une notion essentielle en normalisation est le repérage des redex, autrement dit des endroits où une réduction peut avoir lieu. Nous commencerons donc généralement par décorer le terme initial sous la forme d'un terme à deux niveaux : l'un pour représenter les parties à réduire, l'autre utilisé pour les sous-expressions déjà en forme normale. J'utiliserai dans la suite le vocabulaire issu du monde de l'évaluation partielle, en parlant de niveau ***statique*** pour les expressions pouvant être réduites au moment de la compilation et de niveau ***dynamique*** pour celles qui ne pourront être réduites qu'au moment de l'exécution, lorsque la fonction à normaliser sera appliquée à tous ses paramètres. Si le terme est bien annoté, la réduction des parties statiques doit toujours pouvoir se faire, en aboutissant à la fin à un terme complètement dynamique. Il faudra faire attention à la confusion que peut engendrer cette terminologie lorsque l'on parle de normalisation de λ-termes. Il faudra toujours avoir en tête que l'exécution d'un programme produit une réduction (*dynamique*) du terme appliqué à ses paramètres alors que nous voulons ici nous intéresser à un problème de normalisation *statique* d'un terme non appliqué, c'est-à-dire une normalisation au moment de la compilation.

Nous allons définir précisément le langage à deux niveaux de la manière suivante :

[1] En mémoire, il est stocké sous forme de ***code natif***, propre au micro-processeur, ou bien en « ***bytecode*** » (indépendant du processeur), destiné à être évalué par une ***machine abstraite***.

[2] En programmation fonctionnelle, on appelle ***effets de bord*** les actions ayant lieu en marge du calcul (affichage, modification ou lecture d'une case mémoire ou sur un périphérique, etc.)

Définition 6.1 (λ-calcul à deux niveaux) Un λ-terme à deux niveaux (en notation à la Curry) est un terme de la forme t donnée par la grammaire suivante :

$$\begin{array}{rcl} t & ::= & d \mid s \\ s & ::= & x \mid \lambda x.\, t \mid t \mathbin{@} t \mid \\ & & () \mid \mathrm{paire}(t,\, t) \mid proj_1\ t \mid proj_2\ t \mid \\ & & in_1\ t \mid in_2\ t \mid case\,(t,\, x_1.\, t,\, x_2.\, t) \\ d & ::= & \underline{x} \mid \underline{\lambda x.\, t} \mid t \mathbin{\underline{@}} t \mid \\ & & \underline{()} \mid \underline{\mathrm{paire}}(t,\, t) \mid \underline{proj_1}\ t \mid \underline{proj_2}\ t \mid \\ & & \underline{in_1}\ t \mid \underline{in_2}\ t \mid \underline{case}\,(t,\, x_1.\, t,\, x_2.\, t) \end{array}$$

où x appartient à un ensemble infini dénombrable dit de ***variables statiques***, et $\underline{x}$ appartient à un ensemble infini dénombrable dit de ***variables dynamiques***. Les termes de la forme s sont dits ***statiques***, ceux de la forme d sont dits ***dynamiques***.

Dans la suite, nous verrons que nous pourrons contraindre davantage la syntaxe des termes dynamiques puisque nous voulons leur imposer d'être en forme normale.

Les différentes techniques de normalisation

Il existe essentiellement deux approches pour faire de la normalisation forte : la plus évidente est dite « ***offline*** ». Elle s'appuie sur une analyse statique du terme à normaliser, afin de repérer les parties statiques et dynamiques, et utilise souvent des méthodes de substitutions explicites pour effectuer syntaxiquement les réductions possibles.

La deuxième approche est dite « ***online*** ». Elle consiste à tirer partie des propriétés de normalisation d'un langage de programmation fonctionnel, pour éviter d'avoir à écrire un programme de β-réduction syntaxique. Le λ-terme à normaliser est donc représenté par un programme dans ce langage, et la normalisation est basée sur l'exécution du programme. Le test sur les parties statiques et dynamiques est réalisé au vol pendant la normalisation. Cette idée se heurte néanmoins à deux difficultés de taille :

- les langages fonctionnels comme *Objective Caml* compilent leurs valeurs en forme normale de tête faible, alors que nous voulons normaliser fortement,
- les langages produisent des valeurs compilées, ce qui peut parfois être ce que l'on désire si l'on souhaite par exemple faire de l'optimisation de programme par évaluation partielle, mais dans beaucoup d'autres cas, comme celui qui nous intéresse plus particulièrement, nous souhaitons obtenir le résultat sous forme syntaxique.

Je citerai deux exemples de normaliseurs *online* :

- les machines abstraites de réduction fortes, technique utilisée par exemple par Pierre Crégut [24, 23] pour faire de l'évaluation forte paresseuse, et par Xavier Leroy et Benjamin Grégoire pour faire de l'inférence de type pour un système avec types dépendants dans l'assistant de preuves *Coq* [64]. Ils résolvent le premier des deux problèmes énoncés ci-dessus en modifiant la machine abstraite du langage pour la rendre capable de travailler sur des termes non-clos, et de produire ainsi des valeurs compilées en forme normale. Elle nécessite donc de modifier l'implantation du langage.
- La deuxième méthode, que je vais détailler ici, est bien sûr l'évaluation partielle dirigée par les types, basée sur l'algorithme de normalisation par évaluation. Elle peut fonctionner

sans qu'il soit nécessaire de toucher au langage[3]. L'idée principale consiste à donner les paramètres manquants aux fonctions, afin que la réduction de tête ait lieu. Pour ce faire, elle utilise principalement une η-expansion dans le langage à deux niveaux défini ci-dessus. En jouant sur les deux niveaux du calcul, l'évaluation combinée avec l'η-expansion produit un terme sous forme normale syntaxique (non compilé).

Bref historique

L'algorithme de normalisation par évaluation pour le λ-calcul simplement typé a été décrit pour la première fois par Ulrich Berger et Helmut Schwichtenberg dans un article intitulé *An inverse of the evaluation functional for types λ-calculus* [17]. L'idée était donc clairement d'*inverser* la fonction d'évaluation, autrement dit de donner une technique permettant de retrouver le terme initial (la syntaxe) à partir du terme évalué (ou la sémantique), et qui plus est, en forme normale.

Cette technique, présentée d'abord du point de vue de la théorie de la preuve a depuis été utilisée dans de nombreux domaines : en logique, par exemple par Ulrich Berger [16], en théorie des types par Catarina Coquand [22], en théorie des catégories notamment par Thorsten Altenkirch, Martin Hofmann, et Thomas Streicher [2, 3], par Djordje Čubrić, Peter Dybjer, Phil Scott [81], et récemment par Marcelo Fiore [45] , et enfin en évaluation partielle, notamment par Olivier Danvy [27].

Remarque sur la terminologie L'évaluation partielle dirigée par les types est donc une application de l'algorithme de normalisation par évaluation au problème de l'évaluation partielle. Elle utilise diverses techniques permettant d'implanter effectivement la normalisation par évaluation pour obtenir le code du programme en forme normale à partir du programme compilé dans un langage en appel par valeur[4] comme *ML*. Elle a notamment un mécanisme d'insertion d'instructions `let...in...`, permettant de préserver l'équivalence observationnelle avec le programme initial, et utilise des opérateurs de contrôle, notamment pour gérer le type somme. Dans le cas présenté ici, la distinction entre TDPE et NBE ne sera pas si claire puisque le but du chapitre suivant est d'utiliser les techniques développées pour l'évaluation partielle afin de faire de la normalisation en λ-calcul. J'utiliserai parfois le terme TDPE pour désigner un algorithme de normalisation forte utilisant les techniques de TDPE, notamment les opérateurs de contrôle.

L'algorithme

L'application à l'évaluation partielle permet de se faire une idée intuitive de la manière dont l'algorithme fonctionne. Comme nous l'avons vu plus haut, TDPE transforme l'évaluation de tête en évaluation forte en appliquant les fonctions à des paramètres pour permettre à toutes les réductions d'avoir lieu. Pour ne pas changer la sémantique du résultat, ce passage de paramètres est fait par l'intermédiaire d'une η-expansion. Je vais d'abord présenter l'algorithme d'***η-expansion dirigée par les types***. Il s'agit de faire une η-expansion du terme en se basant sur l'information contenue dans le type :

$$\begin{aligned} |^{\theta}\, t &= t \\ |^{\mathbf{1}}\, t &= () \end{aligned}$$

[3] en tout cas si le langage dispose d'opérateurs de contrôle, ce qui n'est pas le cas de *Objective Caml*...

[4] ***appel par valeur*** : les paramètres sont évalués avant l'application
appel par nom : l'application est évaluée d'abord (et donc la variable de l'abstraction est substituée par le paramètre non-réduit)

$$
\begin{aligned}
|^{\tau_1\to\tau_2}\ t &= \lambda x.\ |^{\tau_2}\ (t\ @\ |^{\tau_1}\ x) \qquad (x \text{ variable neuve})\\
|^{\tau_1\times\tau_2}\ t &= (|^{\tau_1}\ (proj_1\ \ t),\ |^{\tau_2}\ (proj_2\ \ t))\\
|^{\tau_1+\tau_2}\ t &= case\,(t,\ x_1.\ in_1\ \ (|^{\tau_1}\ x_1),\ x_2.\ in_2\ \ (|^{\tau_2}\ x_2))
\end{aligned}
$$

Cependant une telle η-expansion ne résout pas le problème puisqu'elle transforme par exemple une abstraction en une autre. Pour permettre l'évaluation à l'intérieur du corps des abstractions, il faut utiliser une η-expansion à deux niveaux, résumée à la figure 6.1. Les parties statiques (qui doivent être réduites à la compilation) sont écrites en *ML*, les parties dynamiques qui représentent le résultat final sont écrites grâce à une structure de données *ML*. Ainsi le corps d'une abstraction dynamique sera évalué. Une autre conséquence sera que le terme final complètement dynamique sera exprimé dans cette structure de données, donc sous forme syntaxique. Le cas du type somme posant un problème, nous le traiterons plus loin.

$$
\begin{aligned}
\downarrow^{\theta}\ v &= v\\
\downarrow^{1}\ v &= ()\\
\downarrow^{\sigma\to\tau}\ v &= \underline{\lambda x}.\ \downarrow^{\tau}\ (v\ @\ \uparrow^{\sigma}\ \underline{x}) \qquad (\underline{x} \text{ variable neuve})\\
\downarrow^{\tau_1\times\tau_2}\ v &= \underline{\text{paire}}(\downarrow^{\tau_1}\ (proj_1\ \ v),\ \downarrow^{\tau_2}\ (proj_2\ \ v))\\
\\
\uparrow^{\theta}\ e &= e\\
\uparrow^{1}\ e &= ()\\
\uparrow^{\tau\to\sigma}\ e &= \lambda x.\ \uparrow^{\sigma}\ (e\ \underline{@}\ \downarrow^{\tau}\ x)\\
\uparrow^{\sigma_1\times\sigma_2}\ e &= \text{paire}(\uparrow^{\sigma_1}\ (\underline{proj_1}\ \ e),\ \uparrow^{\sigma_2}\ (\underline{proj_2}\ \ e))
\end{aligned}
$$

FIG. 6.1 – Normalisation par évaluation.

Pour un type τ, la fonction $\downarrow^{\tau}$, que l'on appellera ***réification*** (et qui est parfois appelée à tort « ***quote*** » ; il s'agit plutôt du « ***quasi-quote*** » de *Lisp*, *Scheme*...) transforme un terme dynamique (valeur compilée) en une expression en forme normale ; pour un type σ donné, la fonction $\uparrow^{\sigma}$, que l'on appellera ***réflexion*** (appelée parfois « ***unquote*** ») transforme un terme neutre (dans le sens du chapitre 5) en terme dynamique.
L'algorithme final de normalisation (ou ***résidualisation***) pour un λ-terme clos est donc le suivant :

1. Écrire le terme à normaliser de manière complètement statique. Notre but étant de permettre au terme de s'évaluer complètement (en lui passant des paramètres), cette décoration est justifiée.
2. Appliquer l'algorithme de réification $\downarrow$ de la figure 6.1,
3. Réduire les parties statiques.

Théorème 6.2 (NBE sans somme) L'algorithme ci-dessus produit un terme complètement dynamique en forme β-normale $\beta\eta$-équivalent au terme initial.

Démonstration :
Je donnerai juste un aperçu de cette preuve :

Il est facile de voir que le terme final est $\beta\eta$-équivalent au terme initial, étant obtenu par une η-expansion suivie d'une β-réduction. On peut se convaincre qu'il est en forme β-normale par des arguments de typage en restreignant la grammaire des parties dynamiques à celle des formes normales canoniques vue au chapitre 5, qui ne permet pas d'écrire de β-redex. Pour plus de détails sur le typage de l'algorithme, voir la section 6.3.1 ci-après. ■

Il est intéressant de voir que les six lignes de la figure 6.1 à elles-seules donnent non seulement un algorithme de normalisation d'une valeur compilée, mais extraient aussi un représentant syntaxique de la forme normale de cette valeur ! C'est donc un décompilateur à peu de frais, que l'on peut écrire sans avoir la moindre idée du bytecode du langage ou de l'assembleur !

6.2 Évaluation partielle dirigée par les types

6.2.1 Introduction à l'évaluation partielle

Olivier Danvy a redécouvert la méthode décrite ci-dessus dans le but d'écrire un évaluateur partiel pour un langage utilisant une stratégie d'appel par valeur, comme *ML*. Avant de parler de mon implantation, je vais résumer la présentation de TDPE que l'on peut trouver dans ses articles (Danvy [27] et [29]).

L'***évaluation partielle*** est une transformation de programme visant à les spécialiser pour certaines données. Étant donnés un programme et une partie de ses données, elle va construire un programme « résiduel » (en forme normale), qui, appliqué au reste des données produira le même résultat que le programme initial appliqué à l'ensemble des données. Il s'agit donc d'une version effective du théorème S_n^m de Kleene connu en théorie de la calculabilité.

Considérons l'exemple classique de la fonction puissance :

```
let rec puissance n x =
          if n=0
            then 1
            else x * (puissance (n-1) x);;
```

La spécialisation de cette fonction pour `n` $= 3$ donne la fonction suivante :

```
let rec puissance3 x = x * x * x * 1
```

Pour des exemples plus gros, cette technique peut devenir très intéressante et faire gagner beaucoup. Supposons par exemple que nous avons écrit une fonction d'interprétation pour un langage quelconque. Cette fonction prend en paramètres le programme à interpréter et les données sur lesquelles l'évaluer. Une évaluation partielle de l'interpréteur sur un programme donné (sans ses données) va produire une version spécialisée de l'interpréteur pour ce programme, en forme normale. Ce résultat peut lui-même être compilé pour obtenir une version compilée du programme source ! En deux mots, nous avons obtenu un compilateur à partir d'un autre compilateur, en associant un évaluateur partiel à un simple interpréteur... (première projection de Futamura)

Dans d'autres exemples néanmoins, la normalisation peut entraîner une augmentation (voire une explosion) de la taille du code.

6.2.2 De NBE à TDPE

Pour appliquer l'algorithme de normalisation par évaluation à l'évaluation partielle de programme *ML*, nous allons écrire l'algorithme de la figure 6.1 en utilisant le langage *ML* pour les

parties statiques, et en représentant les parties dynamiques dans un type de données que l'on appellera `normal`, qui utilise la grammaire des termes normaux canoniques du chapitre 5. (Le type *Caml* utilisé est présenté à la figure 6.5.)

En résumé, il faut tenir compte des points suivants :

- Nous voulons appliquer la fonction de réification ($\downarrow^\tau$) à des valeurs compilées du langage. Il est donc hors de question de regarder la forme de t, notamment pour gérer les types sommes.
- *ML* est une extension du λ-calcul qui comporte des types prédéfinis (entiers, chaînes de caractères,...), des fonctions récursives, des effets de bord...
- *ML* utilise une stratégie d'évaluation en appel par valeurs, alors que NBE est prévu pour l'appel par nom. Par exemple, la fonction

  ```
  fun f x -> (fun y -> x) (f x)
  ```

  qui peut boucler est normalisée en

  ```
  fun f x -> x
  ```

  qui termine toujours (exemple tiré de Danvy [29]).

Pour les détails sur la façon de résoudre ces problèmes, je renvoie à l'article d'Olivier Danvy [29]. Je donnerai juste quelques éléments de réponse :

- Les effets de bords de la fonction à normaliser se produisent au moment de la normalisation, ce qui en général n'est pas ce que l'on souhaite. Il n'y a pas à ma connaissance de moyen de corriger ce défaut pour l'instant.
- Normaliser une fonction récursive risque de faire boucler indéfiniment la normalisation.
- Les programmes à normaliser doivent être complètement polymorphes, ce que l'on peut obtenir en travaillant sur des termes complètement clos. En fait, on peut autoriser des types non-polymorphes lorsqu'ils apparaissent avec une polarité positive dans le type.
- Enfin l'introduction de constructions `let...in...` permet de résoudre le problème lié à la stratégie d'évaluation de *ML* exposé ci-dessus, et évite certaines duplications de code.

Les défauts mentionnées ci-dessus obligent à utiliser TDPE avec prudence. Néanmoins c'est un évaluateur partiel très ingénieux qui peut être utilisé dans des cas concrets.

6.2.3 TDPE et les types sommes

Étendre l'algorithme de TDPE aux types sommes et booléens n'est pas évident. Pour comprendre le problème, essayons simplement d'appliquer la même technique d'η-expansion à deux niveaux. On obtiendrait les définitions suivantes :

$$\downarrow^{\tau_1+\tau_2} t \quad \stackrel{?}{=} \quad case\,(t,\ x_1.\ \underline{in_1}\ \ (\downarrow^{\tau_1} x_1),\ x_2.\ \underline{in_2}\ \ (\downarrow^{\tau_2} x_2)) \tag{6.1}$$

$$\uparrow^{\sigma_1+\sigma_2} e \quad \stackrel{?}{=} \quad \underline{case}\,(e,\ x_1.\ in_1\ \ (\uparrow^{\sigma_1} x_1),\ x_2.\ in_2\ \ (\uparrow^{\sigma_2} x_2)) \tag{6.2}$$

Essayons d'appliquer cette solution à la normalisation de l'exemple suivant :

$$v = \lambda x.\ case\,(x,\ y.\ y,\ z.\ z) : \theta+\theta\to\theta$$

On obtient :

$$\begin{aligned}
\downarrow^{\theta+\theta\to\theta} v &= \underline{\lambda x}.\ \downarrow^{\theta} (v\ @\ \uparrow^{\theta+\theta} \underline{x}) \qquad (\underline{x} \text{ variable neuve}) \\
&= \underline{\lambda x}.\ \downarrow^{\theta} (v\ @\ \underline{case}\left(\underline{x},\ x_1.\ in_1\ \ (\uparrow^{\theta} x_1),\ x_2.\ in_2\ \ (\uparrow^{\theta} x_2)\right)) \\
&= \underline{\lambda x}.\ (v\ @\ \underline{case}\,(\underline{x},\ x_1.\ (in_1\ \ x_1),\ x_2.\ (in_2\ \ x_2)))
\end{aligned}$$

Mais ce terme ne peut plus être réduit, puisque v attend un terme de type somme alors qu'il est appliqué à un terme de type *normal*. Nous voyons que les définitions 6.1 et 6.2 sont mal typées. Nous voudrions obtenir le terme suivant :

$$\underline{\lambda x}.\ \underline{case}\ (\underline{x},\ x_1.\ (v\ @\ (in_1\ \ x_1)),\ x_2.\ (v\ @\ (in_2\ \ x_2)))$$

Il faudrait donc extraire le *case* pour le placer sous le λ précédent. On vérifie facilement que ce nouveau terme est $\beta\eta$-équivalent au premier et qu'il se réduit en un terme complètement statique en forme normale.

Nous ne pouvons pas poser une définition de $\downarrow^{\tau_1+\tau_2} t$ dépendante de t, car il s'agit d'une valeur compilée. Il nous faut trouver un moyen de déplacer les *case* sous le précédent λ. TDPE fait ceci grâce aux opérateurs de contrôle shift et reset.

shift et reset

Les opérateurs de contrôle shift et reset ont été introduits en 1990 par Olivier Danvy et Andrzej Filinski (voir notamment [30, 31]). Je me contenterai ici de donner une intuition de leur fonctionnement.

L'opérateur reset est utilisé pour *délimiter* un contexte d'évaluation. Ensuite shift *abstrait* ce contexte dans une fonction. Ainsi l'exemple suivant

$$1 + \mathsf{reset}\ (\mathbf{2} + \mathsf{shift}\ c.\ (3\ +\ (c\ 4)\ +\ (c\ 5)))$$

se réduit en

$$1\ +\ 3\ +\ (\mathbf{2}+4)+(\mathbf{2}+5)$$

Le reset délimite le contexte $2\ +\ \square$ (où $\square$ est appelé un « trou », voir par exemple [32]), qui est abstrait en une fonction c, puis 4 et 5 sont successivement insérés dans ce contexte, et l'expression résultante est évaluée.

shift/reset au secours du *case*

Nous pouvons maintenant écrire l'algorithme de normalisation dirigé par le typage grâce à shift/reset comme expliqué à la figure 6.2.

6.3 Implantation de TDPE en *Objective Caml*

Je vais maintenant résumer mon implantation de TDPE en *Objective Caml*, en insistant particulièrement sur ce dont nous avons besoin pour le normaliseur de λ-termes dont il sera question dans le chapitre suivant. Pour plus de détails, notamment sur les applications à l'évaluation partielle et sur la génération de code à l'exécution, il convient de se référer aux articles sur le sujet (Balat [9] et Balat-Danvy [10]).

Le premier problème auquel on est confronté pour implanter cet algorithme de normalisation en *Objective Caml* est bien sûr l'implantation des opérateurs de contrôle. Andrzej Filinski a donné une implantation de shift et reset pour le langage *SML/NJ*, autre dialecte de *ML*, grâce à l'opérateur call/cc [44]. Malheureusement le langage *Objective Caml* ne dispose pas à l'heure actuelle de call/cc, mais Xavier Leroy m'a gentiment fourni une implantation basique de call/cc pour *Objective Caml*, écrite en *C*. Elle se contente de faire des copies entières de la pile, ce qui est évidemment très lourd et inefficace sur de gros exemples, mais qui permet néanmoins de tester le programme.

$$
\begin{array}{rcl}
\downarrow^{\theta} t & = & t \\
\downarrow^{1} t & = & \underline{()} \\
\downarrow^{\sigma\to\tau} t & = & \underline{\lambda x}.\ \mathsf{reset}\ \downarrow^{\tau} (t\ @\ \uparrow^{\sigma} \underline{x}) \qquad (\underline{x} \text{ variable neuve}) \\
\downarrow^{\tau_1\times\tau_2} t & = & \underline{\mathrm{paire}}(\downarrow^{\tau_1} (proj_1\ t), \downarrow^{\tau_2} (proj_2\ t)) \\
\downarrow^{\tau_1+\tau_2} t & = & case\ (t,\ x_1.\ \underline{in_1}\ (\downarrow^{\tau_1} x_1),\ x_2.\ \underline{in_2}\ (\downarrow^{\tau_2} x_2)) \\
\uparrow^{\theta} e & = & e \\
\uparrow^{1} e & = & () \\
\uparrow^{\tau\to\sigma} e & = & \lambda x.\ \uparrow^{\sigma} (e\ \underline{@}\ \downarrow^{\tau} x) \\
\uparrow^{\sigma_1\times\sigma_2} e & = & \mathrm{paire}(\uparrow^{\sigma_1} (\underline{proj_1}\ e),\ \uparrow^{\sigma_2} (\underline{proj_2}\ e)) \\
\uparrow^{\sigma_1+\sigma_2} e & = & \mathsf{shift}\ c.\ \underline{case} \begin{pmatrix} e, \\ x_1.\ \mathsf{reset}\ (c\ @\ in_1\ (\uparrow^{\sigma_1} x_1)), \\ x_2.\ \mathsf{reset}\ (c\ @\ in_2\ (\uparrow^{\sigma_2} x_2)) \end{pmatrix}
\end{array}
$$

FIG. 6.2 – Normalisation dirigée par le typage en appel par valeur.

Nous allons voir maintenant que malgré l'apparente simplicité de l'algorithme, l'implantation dans un langage fortement typé n'est pas évidente. Je présenterai d'abord une solution due à Andrzej Filinski et Zhe Yang [28, 86], puis je montrerai une alternative à cette solution, basée sur la génération de code au moment de l'exécution. Cette méthode m'a permis d'écrire une fonction de normalisation *in situ* de valeurs *Caml* compilées.

6.3.1 Le problème

En *Scheme*, l'implantation de TDPE est très simple. Il suffit d'utiliser les mécanismes de retardement de l'exécution rendus possibles par la « *quasi-quotation* » pour simuler les deux niveaux de λ-calcul. La figure 6.3 montre cette implantation pour les types de base et fonctions (Danvy [27]). La fonction `residualize` prend en paramètre une valeur et un type (sous forme de donnée structurée). (Ici `gensym !` est une fonction générant des variables neuves).

Cependant en *ML*, il n'est pas possible de faire la même chose, pour des raisons de typage. En effet, supposons que `reify` et `reflect` sont des fonctions *Caml* qui prennent en argument d'abord le type puis une valeur (resp. une expression). Alors, par exemple, l'expression :

```
reflect Arrow(Base, Base) e1
```

est de type `'a -> normal`, et

```
reflect Arrow(Base, Prod(Base, Base)) e2
```

est de type `'a -> (normal * normal)`.
(Dans les exemples précédents, `Arrow(Base, Base)` et `Arrow(Base, Prod(Base, Base))` sont les

```
(define residualize
  (lambda (v t)
    (letrec ([reify
              (lambda (t v)
                (case-record t
                  [(Base)
                   v]
                  [(Func t1 t2)
                   (let ([x1 (gensym!)])
                     `(lambda (,x1)
                        ,(reify t2 (v (reflect t1 x1)))))]))]
             [reflect
              (lambda (t e)
                (case-record t
                  [(Base)
                   e]
                  [(Func t1 t2)
                   (lambda (v1)
                     (reflect t2 `(,e ,(reify t1 v1))))]))])
      (begin
        (reset-gensym!)
        (reify (tdpe_parse-type t) v)))))
```

FIG. 6.3 – TDPE en *Scheme*.

expressions utilisées respectivement pour représenter les types $\theta \to \theta$ et $\theta \to (\theta \times \theta)$, et `normal` est le type utilisé pour représenter les termes dynamiques.)

Or *ML* utilise des types simples, et donc le type de la fonction `reflect` ne peut dépendre d'une donnée...

6.3.2 La solution de Filinski/Yang

Pour résoudre ce problème, Andrzej Filinski et Zhe Yang [28, 86] ont proposé (chacun de leur côté) la même astuce, qui permet de construire très facilement la fonction de résidualisation associée à un type. Il suffit de remarquer que l'on peut toujours construire le couple de fonctions $(\downarrow^{\tau_1 * \tau_2}, \uparrow^{\sigma_1 * \sigma_2})$, ou $*$ est un constructeur de type quelconque, connaissant $(\downarrow^{\tau_1}, \uparrow^{\sigma_1})$ et $(\downarrow^{\tau_2}, \uparrow^{\sigma_2})$. Il suffit donc d'écrire une telle fonction pour chaque constructeur de type, et une fonction `residualise` prenant en paramètres un couple de la forme $(\downarrow^{\tau}, \uparrow^{\sigma})$ et une valeur v, appliquant $\downarrow^{\tau}$ à v. Dans les exemples qui suivront, nous appellerons respectivement `base`, `**->`, `produit` et `somme` les fonctions *Caml* associées respectivement de cette manière aux types de base, flèche, produit et somme. La figure 6.4 montre leur implantation. Les types utilisés pour cette implantation sont montrés figure 6.5 et l'interface du module implantant shift et reset est montrée à la figure 6.6. `Names` est un module permettant la génération de noms de variables neuves. Remarquons que le type `normal` suit la syntaxe des formes canoniques du chapitre 5 (page 124).

Voici un exemple d'utilisation du programme :

```
# let toto = (fun f g -> ((fun h a -> h (fun b -> f a b)) g));;
val toto : ('a -> 'b -> 'c) -> (('b -> 'c) -> 'd) -> 'a -> 'd = <fun>

# residualise ((base **->(base **-> base))**->
              (((base **-> base)**-> base) **->
              (base **-> base))) toto;;
- : Controls.ans = Normal0 (Lam ("v0", Normal0 (Lam ("v1",
```

```
let base = ((fun x -> x),
            (fun x -> (Neutral0 x)))

let unit = ((fun c x -> Nil),
            (fun c x -> ()))

let ( **->) (down1,up1) (down2,up2) =
  ((fun v ->
    let var = Names.new_name ()
    in Lam
      (var,
         (reset
            (fun () ->
              Normal0 (down2 (v (up1 (Var var))))))),
   (fun e v1 -> up2 (Appl(e, down1 v1))))

let produit ((down1,up1),(down2,up2)) =
  ((fun v ->
    Pair (down1 (fst v),down2 (snd v))),
   (fun v ->
     ((up1 (Proj1 v),up2 (Proj2 v)))))

let sum ((down1,up1),(down2,up2)) =
  ((fun v ->
    match v with
      Gauche a -> Inl (down1 a)
    | Droite a -> Inr (down2 a)),
   (fun e ->
     let var = Names.new_name ()
     in (shift
       (fun k ->
          (Match(
           e,
           (var, (reset (fun () -> k (Gauche (up1 (Var var))))),
           (var, (reset (fun () -> k (Droite (up2 (Var var))))))))))
   )

let residualise (down,up) v =
  Names.init (); (run (fun () -> Normal0 (down v)))
```

FIG. 6.4 – TDPE en *Objective Caml*.

```
type ('a,'b) sum = Gauche of 'a | Droite of 'b;;

type var = string

type normal =
    Match of neutral0 * (var * normal) * (var * normal)
  | Normal0 of normal0
and normal0 =
    Lam of var * normal
  | Pair of normal0 * normal0
  | Nil
  | Inl of normal0
  | Inr of normal0
  | Neutral0 of neutral0
and neutral0 =
    Var of var
  | Proj1 of neutral0
  | Proj2 of neutral0
  | Appl of neutral0 * normal0
```

FIG. 6.5 – Types.

```
module type SHIFTRESET =
  sig
    type ans
    val run : (unit -> ans) -> ans
    val reset : (unit -> ans) -> ans
    val shift : (('a -> ans) -> ans) -> 'a
  end;;
```

FIG. 6.6 – Interface du module `Controls` implantant shift/reset.

```
Normal0 (Lam ("v2", Normal0 (Neutral0 (Appl (Var "v1", Lam ("v3",
 Normal0 (Neutral0 (Appl (Appl (Var "v0", Neutral0 (Var "v2")),
Neutral0 (Var "v3"))))))))))))))
```

Pour être plus lisible, le résultat peut être affiché avec la syntaxe de *Objective Caml* en utilisant un « *pretty-printer* » :

```
# #install_printer affiche_normal;;
# residualise ((base **->(base **-> base))**->
              (((base **-> base)**-> base) **->
              (base **-> base))) toto;;
- : Controls.ans = (fun v0 v1 v2 -> (v1 (fun v3 -> ((v0 v2) v3))))
```

6.3.3 La solution avec génération de bytecode

La solution de Filinski/Yang utilise donc une fonction de résidualisation construite « à la main » pour chaque type. Il est bien sûr possible de simplifier ce travail en utilisant une syntaxe proche de celle utilisée pour les types du langage (voire même identique). Cette solution présente l'avantage d'être très facilement implantable et c'est celle que j'utiliserai dans le chapitre 7. Cependant j'ai voulu essayer de me débarrasser de cet encombrant paramètre pour obtenir un résultat plus simple d'utilisation.

Pour cela, j'utiliserai une structure de données permettant de représenter un λ-calcul à deux niveaux. Le niveau statique n'est donc plus le langage *Caml* lui-même. Commençons par étudier la solution naïve suivante :

1. Étant donné un type τ, construisons la représentation dans cette structure de données de la fonction de réification.
2. Affichons maintenant ce résultat de la manière suivante :
   - la partie statique avec la syntaxe de *Caml*,
   - la partie dynamique avec la syntaxe des formes normales (type `normal`).
3. Compilons (par copier/coller !) le terme obtenu appliqué à la valeur à résidualiser. On obtient un terme de type `normal`.

Cette méthode en deux étapes d'exécution peut être simplifiée en faisant de la ***génération de bytecode à l'exécution*** [▷ *runtime code generation*]. Pour cela, nous avons besoin de compiler au vol une fonction *Caml*, et de charger en mémoire, également au moment de l'exécution, le bytecode généré. Dans ce but, j'ai utilisé le générateur de bytecode de *Objective Caml*, et j'ai comparé le résultat obtenu avec une générateur de bytecode que j'ai écrit. Celui-ci, beaucoup plus simple et spécialisé pour ma structure de données, se révèle beaucoup plus rapide.

Cependant, pour faire cela, j'ai bien sûr eu besoin de modifier légèrement le code d'*Objective Caml* :

- des points d'accès aux fonctions utiles ont été ajoutés dans le module `Topdirs` (qui est un des modules chargés par défaut par l'interpréteur de commandes) ;
- certaines interfaces (`.cmi`) définissant des types utilisés par le compilateur doivent être rendues accessibles en étant copiées dans le répertoire contenant les bibliothèques d'*Objective Caml*.

Les principales fonctions dont j'ai eu besoin sont :

- des fonctions utilisées pour les différentes étapes de la compilation ;
- des fonctions nécessaires au chargement du code en mémoire
- une fonction retournant l'environnement courant, pour pouvoir avoir les informations nécessaires sur les valeurs chargées en mémoire ;

– des fonctions permettant de retrouver des objets dans cet environnement.

Enfin on peut également se donner le droit d'accéder directement au type inféré par *Caml* pour la fonction à optimiser, ce qui évite d'avoir à le passer en paramètre. L'utilisation est donc rendue très simple. Voici l'exemple de la page 148 avec cette nouvelle version :

```
 # let toto = (fun f g -> ((fun h a -> h (fun b -> f a b)) g));;
val toto : ('a -> 'b -> 'c) -> (('b -> 'c) -> 'd) -> 'a -> 'd = <fun>

 # residualise_rtcg¨ ¨toto;;
- : Controls.ans = (fun v0 v1 v2 -> (v1 (fun v3 -> ((v0 v2) v3))))
```

6.3.4 Normalisation *in situ*

Maintenant que nous pouvons générer et charger en mémoire le bytecode au moment de l'exécution, il devient possible de remplacer directement les valeurs par leur version normalisée, en compilant au vol le résultat obtenu.

Voyons maintenant un exemple simple d'utilisation. La figure 6.7 montre un module définissant l'entier de Church zéro (`cz`), le successeur d'un entier de Church (`cs`), l'addition de deux entiers de Church (`cadd`), ainsi que des fonctions de conversion d'entiers vers entiers de Church (`n2cn`), et vice-versa (`cn2n`).

```
module ChurchNumbers =
  struct

    let cz s z = z

    let cs n s z = n s (s z)

    let cadd a b = a cs b

    let rec n2cn n = if n = 0 then cz else (cs (n2cn (n-1)))
    let cn2n n = n (fun i -> i+1) 0

  end;;
```

FIG. 6.7 – Entiers de Church en *Objective Caml*.

Définissons l'opération « ajouter 1000 » de la manière suivante :

```
 # let cs1000 a = cadd (n2cn 1000) a;;
val cs1000 : (('a -> 'a) -> 'b -> 'a) -> ('a -> 'a) -> 'b -> 'a = <fun>
```

L'application de `csn` à `cz` nécessite $O(\mathtt{n})$ β-réductions qui auraient pu être réalisées statiquement auparavant.

```
 # cn2n (cs1000 cz);;
- : int = 1000
```

La résidualisation de cette valeur s'effectue simplement en tapant la commande suivante :

```
 # residualise_rtcg "cs1000";;
- : unit = ()
```

D'après les tests effectués, le résultat est 4800 fois plus rapide que l'original.

En résumé, cette technique d'implantation a le grand avantage d'aboutir à un résultat très simple à utiliser. Cependant elle nécessite quelques modifications légères d'*Objective Caml*, et n'est pas très simple à implanter. J'ai également fait une version de l'évaluateur partiel qui fonctionne pour le langage de modules d'*Objective Caml*, ce qui permet de réaliser la clôture des termes (nécessaire pour TDPE) élégamment.

6.4 Conclusion

Nous avons vu dans ce chapitre que l'évaluation partielle dirigée par les types permet de construire à peu de frais un outil de normalisation forte pour un langage fonctionnel. La technique d'implantation utilisant la génération de code à l'exécution permet d'intégrer pleinement l'évaluateur au langage et de rendre son utilisation très simple.

La normalisation forte peut aboutir dans certains cas à une sérieuse accélération de l'exécution du programme, mais elle n'est pas toujours souhaitable (augmentation possible de la taille du code, problème des effets de bord,...). De plus TDPE ne peut être appliqué à n'importe quel programme *ML* sans modification.

Nous allons voir dans le chapitre suivant que TDPE est en revanche parfaitement adapté pour résoudre notre problème de normalisation d'isomorphismes de types, et même que l'on peut encore l'améliorer dans ce but.

Chapitre 7

Un normaliseur produisant des formes normales canoniques

Résumé

Dans ce chapitre, je vais appliquer l'algorithme vu au chapitre précédent au problème de la recherche des isomorphismes de types. Cela mettra en évidence la nécessité d'une définition de forme normale plus contrainte pour l'η-conversion, comme celle du chapitre 5. Nous verrons comment modifier l'algorithme de TDPE pour obtenir un résultat sous cette forme canonique. Je montrerai que ces modifications entraînent des optimisations non négligeables pour la normalisation des termes présentés au chapitre 4, optimisations dont on peut tirer partie en évaluation partielle. Nous verrons également que ce travail soulève des questions intéressantes concernant les opérateurs de contrôle.

L'UTILISATION de l'algorithme de normalisation présenté au chapitre précédent pour vérifier les isomorphismes de types du chapitre 4 a mis en évidence la nécessité d'optimiser la technique pour obtenir un résultat plus compact. Je montrerai comment j'ai pu adapter la technique pour produire des résultats dans la forme normale canonique présentée au chapitre 5, obtenant ainsi dans notre cas des termes d'une taille plusieurs dizaines de fois inférieure !

7.1 Le problème

Au chapitre 4, j'ai présenté une suite d'isomorphismes permettant de prouver qu'une axiomatisation finie des isomorphismes de types avec somme n'est pas possible. Cette fonction, écrite en *Caml*, est présentée dans le cas $n = 3$ à la figure 7.1. Comme nous l'avons vu, prouver que cette fonction est un isomorphisme n'est pas évident (voir pages 104 à 108). J'ai voulu tirer partie du travail sur la normalisation par évaluation pour réduire automatiquement la composition de

```
let f3 =
  fun (a1,a2) -> (
    (fun u ->
     match (a2 u) with
       Gauche b -> Gauche (fun v ->
         match (a1 v) with
           Gauche c -> (c u)
         | Droite c -> (match (c u),(b v) with
             (Gauche d), (Gauche e) -> (* y2,y3 *) Gauche (fst d)
           | (Gauche d), (Droite e) -> (* y2,x3 *) Droite (fst e)
           | (Droite (Gauche d)), (Gauche e) -> (* yx,y3 *) Droite (snd d)
           | (Droite (Gauche d)), (Droite e) -> (* yx,x3 *) Gauche (fst d)
           | (Droite (Droite d)), (Gauche e) -> (* x2,y3 *) Gauche (fst e)
           | (Droite (Droite d)), (Droite e) -> (* x2,x3 *) Droite (fst d))
                 )
     | Droite b -> Droite (fun v ->
         match (a1 v) with
         | Gauche c -> (match (c u),(b v) with
             (Gauche d), (Gauche e) -> (* y,y4 *) Gauche (d,(fst e))
           | (Gauche d), (Droite (Gauche e)) -> (* y,y2x2 *) Droite (Droite (snd (snd e)))
           | (Gauche d), (Droite (Droite e)) -> (* y,x4 *) Droite (Gauche ((d,fst e)))
           | (Droite d), (Gauche e) -> (* x,y4 *) Droite (Gauche ((fst e,d)))
           | (Droite d), (Droite (Gauche e)) -> (* x,y2x2 *) Gauche (fst e,(fst (snd e)))
           | (Droite d), (Droite (Droite e)) -> (* x,x4 *) Droite (Droite (d,fst e)))
         | Droite c -> (c u)
    )),
    (fun v ->
     match (a1 v) with
       Gauche c -> Gauche (fun u ->
         match (a2 u) with
           Gauche b -> (b v)
         | Droite b -> (match (c u),(b v) with
             (Gauche d), (Gauche e) -> (* y,y4 *) Gauche (snd e)
           | (Gauche d), (Droite (Gauche e)) -> (* y,y2x2 *) Gauche (d,((fst e),(fst (snd e))))
           | (Gauche d), (Droite (Droite e)) -> (* y,x4 *) Droite (snd e)
           | (Droite d), (Gauche e) -> (* x,y4 *) Gauche (snd e)
           | (Droite d), (Droite (Gauche e)) -> (* x,y2x2 *) Droite (d,(snd (snd e)))
           | (Droite d), (Droite (Droite e)) -> (* x,x4 *) Droite (snd e))
                 )
     | Droite c -> Droite (fun u ->
         match (a2 u) with
         | Gauche b -> (match (c u),(b v) with
             (Gauche d), (Gauche e) -> (* y2,y3 *) Gauche ((snd d),e)
           | (Gauche d), (Droite e) -> (* y2,x3 *) Droite (Gauche (fst d,(snd d,snd e)))
           | (Droite (Gauche d)), (Gauche e) -> (* yx,y3 *) Gauche ((fst d),e)
           | (Droite (Gauche d)), (Droite e) -> (* yx,x3 *) Droite (Droite ((snd d),e))
           | (Droite (Droite d)), (Gauche e) -> (* x2,y3 *) Droite (Gauche ((fst (snd e)),
                                                                  ((snd (snd e)),d)))
           | (Droite (Droite d)), (Droite e) -> (* x2,x3 *) Droite (Droite ((snd d),e)))
         | Droite b -> (b v)
  )));;

val f3 : ('a -> ('b -> ('c, 'd) sum, 'b -> ('c * 'c, ('c * 'd, 'd * 'd) sum) sum) sum) *
('b -> ('a -> ('c * ('c * 'c), 'd * ('d * 'd)) sum,
'a -> ('c * ('c * ('c * 'c)), ('c * ('c * ('d * 'd)),
'd * ('d * ('d * 'd))) sum) sum) sum)
->
('b -> ('a -> ('c, 'd) sum, 'a -> ('c * 'c, ('c * 'd, 'd * 'd) sum) sum) sum) *
('a -> ('b -> ('c * ('c * 'c), 'd * ('d * 'd)) sum,
'b -> ('c * ('c * ('c * 'c)), ('c * ('c * ('d * 'd)),
'd * ('d * ('d * 'd))) sum) sum) sum) = <fun>
```

FIG. 7.1 – Isomorphisme dans le cas $n = 3$.

cette fonction avec son inverse supposée (en fait c'est une involution), et prouver qu'il s'agit bien de l'identité. Le résultat de cette résidualisation est présenté en partie à la figure 7.2. La version complète fait environ 1200 lignes (alors que `f3` fait seulement 52 lignes).

```
 #  let comp3 x = f3 (f3 x);;

 #  residualise (((prod ((base **-> (sum ((base **-> (sum (base,base))),(base **-> (sum ((prod (ba
se, base)), (sum ((prod (base, base)), (prod (base, base))))))))))), (base **-> (sum ((base **-> (s
um ((prod (base, (prod (base, base)))), (prod (base, (prod (base, base)))))))), (base **-> (sum ((p
rod (base, (prod (base, (prod (base, base)))))), (sum ((prod (base, (prod (base, (prod (base, base
)))))), (prod (base, (prod (base, (prod (base, base))))))))))))))))),Names2.Fixname "a") **=> (prod
 (((base,Names2.Fixname "u") **=> (sum (((base,Names2.Fixname "v") **=> (sum (base,base))), ((base
,Names2.Fixname "v") **=> (sum ((prod (base, base)), (sum ((prod (base, base)), (prod (base, base)
)))))))), ((base,Names2.Fixname "v") **=> (sum (((base,Names2.Fixname "u") **=> (sum ((prod (base
, (prod (base, base)))), (prod (base, (prod (base, base)))))))), ((base,Names2.Fixname "u") **=> (s
um ((prod (base, (prod (base, (prod (base, base)))))), (sum ((prod (base, (prod (base, (prod (base
, base)))))), (prod (base, (prod (base, (prod (base, base))))))))))))))))) comp3;;

- : Controls.ans =
(fun a ->
 ((fun u ->
   (match ((proj1 a) u)
    with
    | Gauche v160 ->
        (Gauche (fun v ->
           (match ((proj2 a) v)
            with
            | Gauche v241 -> (match ((proj1 a) u)
                with
                | Gauche v313 -> (match (v313 v)
                   with
                   | Gauche v319 -> (Gauche v319)
                   | Droite v319 -> (Droite v319))

                | Droite v313 -> (match (v313 v)
                   with
                   | Gauche v314 -> (match (v241 u) with
                                     | Gauche v318 -> (Gauche (proj1 v314))
                                     | Droite v318 -> (Droite (proj1 v318)))

                   | Droite v314 -> (match v314 with
                                     | Gauche v315 -> (match (v241 u) with
                                                       | Gauche v317 -> (Droite (proj2 v315))
                                                       | Droite v317 -> (Gauche (proj1 v315)))

                                     | Droite v315 -> (match (v241 u) with
                                                       | Gauche v316 -> (Gauche (proj1 v316))
                                                       | Droite v316 -> (Droite (proj1 v315)))
                                   )
                               )
                           )

            | Droite v241 -> (match ((proj1 a) u)
                with
                | Gauche v242 -> (match (v242 v) with
                                  ...
...
```

FIG. 7.2 – Résidualisation de la composition de `f3` avec elle-même (petit extrait).

En observant ce résultat, nous pouvons voir qu'il ne respecte pas les conditions de la figure 5.1, à savoir :

Dans $\lambda x.\ N$:

$$\text{la variable } x \text{ vérifie } x \in FV(C) \text{ pour tout } C \in \text{gardes}(N) \qquad \diamondsuit$$

Dans $case\,(M,\ x_1.\ N_1,\ x_2.\ N_2)$:

$$M \notin \bigcup_{i=1,2} \text{gardes}(x_i.\ N_i), \qquad \spadesuit$$

$$\text{et si } x_1 \notin FV(N_1) \text{ et } x_2 \notin FV(N_2) \text{ alors } N_1 \not\approx N_2 \qquad \clubsuit$$

Regardons quelques exemples plus simples.

Exemple 7.1 La résidualisation de la fonction suivante ne vérifie pas la condition $\diamondsuit$, puisque `(v2 v0)` ne contient pas la variable `v4` :

```
# let f t x g = match g x with
              | Gauche c -> (fun y -> Gauche y)
              | _ -> (fun y -> (g t));;

# residualise (base **-> (base **-> ((base **-> (sum (base ,base)))
**-> ((base **-> (sum (base ,base))))))) f;;
- : Controls.ans =
(fun v0 v1 v2 -> (match (v2 v1) with
                | Gauche v3 -> (fun v6 -> (Gauche v6))
                | Droite v3 -> (fun v4 -> (match (v2 v0) with
                                         | Gauche v5 -> (Gauche v5)
                                         | Droite v5 -> (Droite v5)))))
```

Exemple 7.2 Testons maintenant la fonction de résidualisation sur un exemple suggéré par Andrzej Filinski[1]. Il est possible de montrer que pour toute fonction booléenne f, on a $f \circ f \circ f = f$.

```
# let fff f x = f (f (f x));;
val fff : ('a -> 'a) -> 'a -> 'a = <fun>
```

Définissons la fonction `bool` de la manière suivante :

```
# let bool = sum (unit,unit);;

# residualise ((bool **-> bool) **-> (bool **-> bool)) fff;;
- : Controls.ans =
(fun v0 v1 ->
 (match v1
  with
  | Gauche
```

[1] Copenhague, FLoC 2002

```
   v2 -> (match (v0 (Gauche ()))
          with
          | Gauche v10 -> (match (v0 (Gauche ())) with
                           | Gauche v14 -> (match (v0 (Gauche ())) with
                                           | Gauche v16 -> (Gauche ())
                                           | Droite v16 -> (Droite ()))

                           | Droite v14 -> (match (v0 (Droite ())) with
                                           | Gauche v15 -> (Gauche ())
                                           | Droite v15 -> (Droite ()))
                            )

          | Droite v10 -> (match (v0 (Droite ())) with
                           | Gauche v11 -> (match (v0 (Gauche ())) with
                                           | Gauche v13 -> (Gauche ())
                                           | Droite v13 -> (Droite ()))

                           | Droite v11 -> (match (v0 (Droite ())) with
                                           | Gauche v12 -> (Gauche ())
                                           | Droite v12 -> (Droite ()))
                            )
            )

 | Droite
   v2 -> (match (v0 (Droite ()))
          with
          | Gauche v3 -> (match (v0 (Gauche ())) with
                          | Gauche v7 -> (match (v0 (Gauche ())) with
                                          | Gauche v9 -> (Gauche ())
                                          | Droite v9 -> (Droite ()))

                          | Droite v7 -> (match (v0 (Droite ())) with
                                          | Gauche v8 -> (Gauche ())
                                          | Droite v8 -> (Droite ()))
                           )

          | Droite v3 -> (match (v0 (Droite ())) with
                          | Gauche v4 -> (match (v0 (Gauche ())) with
                                          | Gauche v6 -> (Gauche ())
                                          | Droite v6 -> (Droite ()))

                          | Droite v4 -> (match (v0 (Droite ())) with
                                          | Gauche v5 -> (Gauche ())
                                          | Droite v5 -> (Droite ()))
                           )
            )
   )
)
```

On voit clairement que la condition ♠ n'est pas vérifiée, puisque l'on a par exemple deux `match (v0 (Gauche ()))` imbriqués. Enfin on a `fff` $=_{\beta\eta}$ `id`, mais la résidualisation de l'identité avec le même type ne produit pas le même code :

```
 # let id x = x;;
val id : 'a -> 'a = <fun>
 # residualise ((bool **-> bool) **-> (bool **-> bool)) id;;
- : Controls.ans = (fun v0 v1 -> (match v1 with
                                | Gauche v2 -> (match (v0 (Gauche ())) with
                                                | Gauche v4 -> (Gauche ())
                                                | Droite v4 -> (Droite ()))

                                | Droite v2 -> (match (v0 (Droite ())) with
                                                | Gauche v3 -> (Gauche ())
                                                | Droite v3 -> (Droite ()))
                                 )
                   )
```

7.2 Trois solutions pour répondre aux trois conditions

Cette section décrit les optimisations permettant de prendre en compte les contraintes sur les formes normales décrites au chapitre 5.

7.2.1 Condition ♠

En étudiant rapidement le code produit par TDPE sur la fonction `comp3` (voir figure 7.2), on s'aperçoit tout de suite que de nombreuses branches du terme ne sont jamais atteintes. Pour résoudre ce problème, nous pouvons utiliser le lemme 2.9 page 53.

Pour cela, remarquons que le programme résiduel est un arbre de syntaxe abstraite construit en profondeur d'abord, de droite à gauche, l'évaluation se faisant en appel par valeur. L'idée consiste à maintenir une table globale tenant à jour les branches conditionnelles traversées depuis la racine du programme résiduel jusqu'au point courant de la construction.

Cette table associe un drapeau (G ou D) et une variable à une expression de la manière suivante :

$$\uparrow^{\sigma_1+\sigma_2} e \quad = \quad \begin{cases} in_1 \ (\uparrow^{\sigma_1} z) & \text{si } e \text{ est globalement associé à (G, } z\text{)} \\ in_2 \ (\uparrow^{\sigma_2} z) & \text{si } e \text{ est globalement associé à (D, } z\text{)} \\ \mathsf{shift}\ c.\ \underline{case} \begin{pmatrix} e, \\ x_1.\ \mathsf{reset}\ in_1\ \uparrow^{\sigma_1} x_1, \\ x_2.\ \mathsf{reset}\ in_2\ \uparrow^{\sigma_2} x_2 \end{pmatrix} & \text{sinon} \\ \text{où } x_1 \text{ et } x_2 \text{ sont neuves} \end{cases}$$

Si e n'est associé à rien dans la table, alors nous lui associons $(G,\ x_1)$ en entrant dans la première branche du *case*, et $(D,\ x_2)$ en entrant dans la seconde.

7.2.2 Condition ♣

Pour gérer la condition ♣, nous devons implanter un test de l'équivalence $\approx$ définie page 123, que l'on appellera égalité modulo ***conversions commutatives***. Cela correspond à l'application du lemme 2.8 page 52. Pour cela, nous devons d'abord écrire une fonction vérifiant l'égalité de deux termes dynamiques (de type `normal`) modulo α-conversion, ainsi qu'un test d'appartenance aux variables libres. Ces fonctions ne posent pas de problème théorique.

Ensuite nous allons écrire une fonction testant l'égalité modulo conversions commutatives de deux termes normaux, en supposant qu'ils vérifient déjà tous deux la condition ♣. Commençons par démontrer le lemme suivant :

Lemme 7.3 Si N_1 et N_2 sont deux termes normaux vérifiant la condition ♣ et tels que

$$N_1 \approx N_2$$

alors $\text{gardes}(N_1) = \text{gardes}(N_2)$.

Démonstration :

$\text{gardes}(\mathit{case}\ (\boldsymbol{M},\ x.\ \mathit{case}\ (\boldsymbol{M_1},\ x_1.\ N_1,\ x_2.\ N_2),\ y.\ N))$
$\quad = \{\ \boldsymbol{M}\ \} \cup \text{gardes}(x.\ \mathit{case}\ (\boldsymbol{M_1},\ x_1.\ N_1,\ x_2.\ N_2)) \cup \text{gardes}(y.\ N)$
$\quad = \{\ \boldsymbol{M},\ \boldsymbol{M_1}\ \} \cup \text{gardes}(x.\ x_1.\ N_1) \cup \text{gardes}(x.\ x_2.\ N_2) \cup \text{gardes}(y.\ N)$
$\text{gardes}(\mathit{case}\ (\boldsymbol{M_1},\ x_1.\ \mathit{case}\ (\boldsymbol{M},\ x.\ N_1,\ y.\ N),\ x_2.\ \mathit{case}\ (\boldsymbol{M},\ x.\ N_2,\ y.\ N)))$
$\quad = \{\ \boldsymbol{M_1}\ \} \cup \text{gardes}(x_1.\ \mathit{case}\ (\boldsymbol{M},\ x.\ N_1,\ y.\ N)) \cup \text{gardes}(x_2.\ \mathit{case}\ (\boldsymbol{M},\ x.\ N_2,\ y.\ N))$
$\quad = \{\ \boldsymbol{M},\ \boldsymbol{M_1}\ \} \cup \text{gardes}(x.\ x_1.\ N_1) \cup \text{gardes}(x.\ x_2.\ N_2) \cup \text{gardes}(y.\ N)$
Le deuxième cas est identique. ■

La première étape de notre fonction peut donc être le test de l'égalité des deux ensembles de gardes. S'ils sont différents, les termes ne sont pas équivalents. Sinon, on continue suivant la méthode suivante. À chaque garde correspond deux choix possibles, soit en tout 2^n choix (où n est le nombre de gardes). Pour chacun de ces choix, on peut facilement trouver la branche du terme concernée, et vérifier que c'est la même pour les deux termes (modulo α-équivalence). On n'a pas besoin de vérifier récursivement la condition à l'intérieur de ces branches puisque l'on sait que N_1 et N_2 vérifient déjà la condition ♣, TDPE construisant le terme en profondeur d'abord.

7.2.3 Condition ♢

Pour obtenir des termes dans la forme normale du chapitre 5, nous devons aussi vérifier une condition concernant les gardes du corps des abstractions (condition ♢).

Pour cela, regardons l'exemple 7.1 page 158. Nous voulons introduire le « `match (v2 v0)` » au-dessus de « `fun v4` »... Or un shift renvoie toujours au reset précédent. Il faudrait pouvoir nommer les reset et choisir le meilleur au moment d'introduire le `match`. C'est ce que permettent de faire les opérateurs de contrôle cupto/set.

Utilisation de cupto/set

Les opérateurs de contrôle set et cupto ont été introduits en 1998 par Carl A. Gunter, Didier Rémy et Jon G. Riecke [53]. Ils généralisent les exceptions et les continuations (alors que shift et reset ne permettent pas de coder les exceptions). Cependant ils ne sont pas tirés d'une étude sur les continuations, contrairement à shift/reset, qui en tirent leur légitimité.

Pour le détail de leur sémantique opérationnelle, je renvoie à l'article mentionné ci-dessus. Je me contenterai encore une fois de donner l'intuition de leur fonctionnement sur un exemple.

Le fonctionnement de cupto/set repose sur la notion de ***prompt***, permettant de marquer les occurrences de set. Il est possible de créer à la demande des nouveaux prompts. Si p_1 et p_2 sont deux prompts, on peut écrire par exemple l'expression suivante :

$$1 + \mathsf{set}\ p_1\ \mathsf{in}\ \mathbf{2} + \mathsf{set}\ p_2\ \mathsf{in}\ \mathbf{3} + \mathsf{cupto}\ p_1\ \mathsf{as}\ c\ \mathsf{in}\ (\mathbf{4} + (c\ \mathbf{5}))$$

se réduit en :

$$1 + 4 + (\mathbf{2} + \mathbf{3} + 5)$$

Application à TDPE

Pour utiliser ces opérateurs de contrôle à notre problème, nous devons créer un nouveau prompt à chaque λ dynamique créé (remise à zéro de l'ensemble des gardes). Nous maintenons une liste globale associant à chaque prompt un ensemble de variables. Pour introduire un nouveau *case*, il suffit de chercher toutes les variables libres de sa condition, et d'aller chercher dans cette liste le dernier prompt introduit associé à l'une de ces variables. Le terme étant construit en profondeur d'abord et de droite à gauche, cela assure que l'on obtient bien de cette façon un terme clos.

Nous modifions donc l'algorithme de la figure 6.2 de la manière suivante :

$$\downarrow^{\sigma\to\tau} t \quad = \quad \underline{\lambda x}.\ \mathsf{set}\ p\ \mathsf{in}\ \downarrow^{\tau} (t\ @\ \uparrow^{\sigma} \underline{x}) \qquad (\underline{x}\ \text{variable neuve},\ p\ \text{nouveau prompt})$$

$$\uparrow^{\sigma_1+\sigma_2} e \quad = \quad \mathsf{cupto}\ m\ \mathsf{as}\ c\ \mathsf{in}\ \underline{case} \begin{pmatrix} e, \\ x_1.\ \mathsf{set}\ m\ \mathsf{in}\ (c\ @\ in_1\ \ (\uparrow^{\sigma_1} x_1)), \\ x_2.\ \mathsf{set}\ m\ \mathsf{in}\ (c\ @\ in_2\ \ (\uparrow^{\sigma_2} x_2)) \end{pmatrix}$$

où m est le meilleur prompt pour e.

Notons que tel quel, l'algorithme peut boucler sur certains exemples[2]. Mais en associant cette optimisation à celle pour la condition ♠, nous obtenons un algorithme qui termine toujours. L'algorithme final est présenté à la figure 7.3.

Discussion sur les opérateurs de contrôle

Le nouvel algorithme n'utilise pas toute la puissance des opérateurs cupto/set ; en particulier nous ne nous servons pas du codage des exceptions grâce à eux. Nous pourrions donc n'utiliser qu'une version restreinte de ces opérateurs. Il existe par exemple une version hiérarchisée de shift/reset, permettant d'avoir plusieurs niveaux de contrôle (voir Danvy-Filinski [30]). Mais ils nécessitent de connaître à l'avance la profondeur maximale nécessaire, ce qui est impossible dans notre cas. Après de multiples discussion avec Olivier Danvy, Andrzej Filinski et Didier Rémy, une implantation avec des shift/reset (hiérarchisés ou non) ne semble pas évidente, même si elle paraît possible théoriquement.

7.2.4 Les résultats

Observons le code produit pour les exemples 7.1 et 7.2 page 158.

```
# residualise2 (base **-> (base **-> ((base **-> (sum (base ,base))) **->
((base **-> (sum (base ,base))))))) f;;
```

[2] En fait le seul exemple faisant boucler l'algorithme que j'ai trouvé pour l'instant est l'isomorphisme pour les formules de Wilkie-Gurevič généralisées.

$$
\begin{array}{rcl}
\downarrow^{\theta} t & = & t \\
\downarrow^{1} t & = & \underline{()} \\
\downarrow^{\sigma\to\tau} t & = & \underline{\lambda x}.\ \mathsf{set}\ p\ \mathsf{in}\ \downarrow^{\tau}(t\ @\ \uparrow^{\sigma}\underline{x}) \qquad (\underline{x}\ \text{variable neuve},\ p\ \text{nouveau prompt}) \\
\downarrow^{\tau_1\times\tau_2} t & = & \underline{\text{paire}}(\downarrow^{\tau_1}(proj_1\ t),\ \downarrow^{\tau_2}(proj_2\ t)) \\
\downarrow^{\tau_1+\tau_2} t & = & case\ (t,\ x_1.\ \underline{in_1}\ (\downarrow^{\tau_1} x_1),\ x_2.\ \underline{in_2}\ (\downarrow^{\tau_2} x_2)) \\
\uparrow^{\theta} e & = & e \\
\uparrow^{1} e & = & () \\
\uparrow^{\tau\to\sigma} e & = & \lambda x.\ \uparrow^{\sigma}(e\ \underline{@}\ \downarrow^{\tau} x) \\
\uparrow^{\sigma_1\times\sigma_2} e & = & \text{paire}(\uparrow^{\sigma_1}(\underline{proj_1}\ e),\ \uparrow^{\sigma_2}(\underline{proj_2}\ e))
\end{array}
$$

$$
\uparrow^{\sigma_1+\sigma_2} e \ = \ \begin{cases}
in_1\ (\uparrow^{\sigma_1} z) & \text{si } e \text{ est globalement associé à (G, } z\text{)} \\
in_2\ (\uparrow^{\sigma_2} z) & \text{si } e \text{ est globalement associé à (D, } z\text{)} \\
\mathsf{cupto}\ m\ \mathsf{as}\ c\ \mathsf{in} \left|\begin{array}{rcl} \mathsf{let} & n_1 & = \ \mathsf{set}\ m\ \mathsf{in}\ (c\ @\ in_1\ (\uparrow^{\sigma_1} x_1)) \\ \mathsf{and} & n_2 & = \ \mathsf{set}\ m\ \mathsf{in}\ (c\ @\ in_2\ (\uparrow^{\sigma_2} x_2)) \\ \mathsf{in} & \multicolumn{2}{l}{\begin{cases} n_1 & \text{si } x_1 \notin FV(n_1),\ x_2 \notin FV(n_2) \text{ et } n_1 \approx n_2 \\ \underline{case}\,(e,\ x_1.\ n_1,\ x_2.\ n_2) & \text{sinon} \end{cases}} \end{array}\right. \\
\multicolumn{2}{l}{\text{sinon, où } m \text{ est le meilleur prompt pour e.}} \\
\multicolumn{2}{l}{\text{Si e n'est associé à rien dans la table, alors nous lui associons}} \\
\multicolumn{2}{l}{(G,\ x_1) \text{ en entrant dans la première branche du } \underline{case}\text{,}} \\
\multicolumn{2}{l}{\text{et } (D,\ x_2) \text{ en entrant dans la seconde.}}
\end{cases}
$$

FIG. 7.3 – Normalisation dirigée par les types optimisée.

```
- : normal =
(fun v0 v1 v2 -> (match (v2 v1) with
                | Gauche v3 -> (fun v5 -> (Gauche v5))
                | Droite v4 -> (match (v2 v0) with
                                | Gauche v7 -> (fun v6 -> (Gauche v7))
                                | Droite v8 -> (fun v6 -> (Droite v8)))
               )
)

 # residualise2 ((bool **-> bool) **-> (bool **-> bool)) fff;;
- : normal =
(fun v0 -> (match (v0 (Gauche ())) with
           | Gauche v4 -> (match (v0 (Droite ())) with
                           | Gauche v6 -> (fun v1 -> (Gauche ()))
                           | Droite v7 -> (fun v1 -> (match v1 with
                                                      | Gauche v2 -> (Gauche ())
                                                      | Droite v3 -> (Droite ()))
                                                     ))

           | Droite v5 -> (match (v0 (Droite ())) with
                           | Gauche v10 -> (fun v1 -> (match v1 with
                                                       | Gauche v2 -> (Droite ())
                                                       | Droite v3 -> (Gauche ()))
                                                      )
                           | Droite v11 -> (fun v1 -> (Droite ())))
             )
)
```

Cette fois-ci, le résultat est identique à la résidualisation de l'identité :

```
 # residualise2 ((bool **-> bool) **-> (bool **-> bool)) id;;
- : normal =
(fun v0 -> (match (v0 (Gauche ())) with
           | Gauche v4 -> (match (v0 (Droite ())) with
                           | Gauche v6 -> (fun v1 -> (Gauche ()))
                           | Droite v7 -> (fun v1 -> (match v1 with
                                                      | Gauche v2 -> (Gauche ())
                                                      | Droite v3 -> (Droite ()))
                                                     ))

           | Droite v5 -> (match (v0 (Droite ())) with
                           | Gauche v8 -> (fun v1 -> (match v1 with
                                                      | Gauche v2 -> (Droite ())
                                                      | Droite v3 -> (Gauche ()))
                                                     )
                           | Droite v9 -> (fun v1 -> (Droite ())))
             )
)
```

La figure 7.4 montre la résidualisation de la fonction `comp3` avec le nouveau normaliseur. Comparée au résultat présenté à la figure 7.2, le résultat est environ 48 fois plus petit (25 lignes au lieu

de 1200, et environ 250 sans tenir compte de la condition ♣).

```
 #  residualise2 (((prod ((base **-> (sum ((base **-> (sum (base,base))),(base **-> (sum ((prod (b
ase, base)), (sum ((prod (base, base)), (prod (base, base))))))))))), (base **-> (sum ((base **-> (
sum ((prod (base, (prod (base, base)))), (prod (base, (prod (base, base)))))))), (base **-> (sum ((
prod (base, (prod (base, (prod (base, base)))))), (sum ((prod (base, (prod (base, (prod (base, bas
e)))))), (prod (base, (prod (base, (prod (base, base))))))))))))))),Names2.Fixname "a") **=> (pro
d (((base,Names2.Fixname "u") **=> (sum (((base,Names2.Fixname "v") **=> (sum (base,base))), ((bas
e,Names2.Fixname "v") **=> (sum ((prod (base, base)), (sum ((prod (base, base)), (prod (base, base
))))))))))), ((base,Names2.Fixname "v") **=> (sum (((base,Names2.Fixname "u") **=> (sum ((prod (bas
e, (prod (base, base)))), (prod (base, (prod (base, base)))))))), ((base,Names2.Fixname "u") **=> (
sum ((prod (base, (prod (base, (prod (base, base)))))), (sum ((prod (base, (prod (base, (prod (bas
e, base)))))), (prod (base, (prod (base, (prod (base, base))))))))))))))) comp3;;

- : normal =
(fun a ->
 ((fun u -> (match ((proj1 a) u) with
    | Gauche v32 -> (Gauche (fun v -> (match (v32 v) with
        | Gauche v36 -> (Gauche v36)
        | Droite v37 -> (Droite v37))
                  ))
    | Droite v33 -> (Droite (fun v -> (match (v33 v) with
        | Gauche v50 -> (Gauche ((proj1 v50) , (proj2 v50)))
        | Droite v51 -> (match v51 with
                         | Gauche v54 -> (Droite (Gauche ((proj1 v54) , (proj2 v54))))
                         | Droite v55 -> (Droite (Droite ((proj1 v55) , (proj2 v55)))))
         ))))
  ) ,
  (fun v ->
   (match ((proj2 a) v) with
    | Gauche v0 -> (Gauche (fun u -> (match (v0 u) with
       | Gauche v4 -> (Gauche ((proj1 v4) , ((proj1 (proj2 v4)) , (proj2 (proj2 v4)))))
       | Droite v5 -> (Droite ((proj1 v5) , ((proj1 (proj2 v5)) , (proj2 (proj2 v5))))))
                  ))
    | Droite v1 -> (Droite (fun u -> (match (v1 u) with
       | Gauche v20 -> (Gauche ((proj1 v20) , ((proj1 (proj2 v20)) ,
                                ((proj1 (proj2 (proj2 v20))) , (proj2 (proj2 (proj2 v20)))))))
       | Droite v21 -> (match v21 with
                        | Gauche v22 -> (Droite (Gauche ((proj1 v22) , ((proj1 (proj2 v22)),
                                 ((proj1 (proj2 (proj2 v22))) , (proj2 (proj2 (proj2 v22))))))))
                        | Droite v23 -> (Droite (Droite ((proj1 v23) , ((proj1 (proj2 v23)) ,
                                ((proj1 (proj2 (proj2 v23))) , (proj2 (proj2 (proj2 v23)))))))))
        )))))))
```

FIG. 7.4 – Résidualisation de la composition de `f3` avec elle-même, avec le nouvel algorithme.

7.3 η-réduction

Les formes normales produites par le nouveau normaliseur sont des formes normales η-longues, comme présentées au chapitre 5. Contrairement au cas général, il est possible pour les termes de cette forme, d'écrire une fonction d'η-réduction qui va réduire notre terme en une identité non η-expansée.

Cette fonction d'η-réduction parcourt le terme en profondeur d'abord en repérant les motifs correspondant aux règles η (à α-équivalence près), et en les remplaçant par leur réduction. Nous avons vu au chapitre 2 que cette technique ne peut pas faire « toutes » les η-réductions, et même

qu'il n'est pas facile de définir une notion de forme η-réduite en présence de la somme. En particulier cette fonction d'η-réduction appliquée au terme de la figure 7.2 ne produit pas le terme $\lambda x.\ x$ (en fait il comporte toujours des centaines de lignes).

La figure 7.5 montre qu'avec le nouveau normaliseur, nous obtenons bien l'identité non η-expansée.

```
# etared (
 residualise2 (((prod ((base **-> (sum ((base **-> (sum (base,base))),(base **-> (sum ((prod (bas
e, base)), (sum ((prod (base, base)), (prod (base, base))))))))))), (base **-> (sum ((base **-> (su
m ((prod (base, (prod (base, base)))), (prod (base, (prod (base, base)))))))), (base **-> (sum ((pr
od (base, (prod (base, (prod (base, base)))))), (sum ((prod (base, (prod (base, (prod (base, base)
))))), (prod (base, (prod (base, (prod (base, base)))))))))))))))),Names2.Fixname "a") **=> (prod
(((base,Names2.Fixname "u") **=> (sum (((base,Names2.Fixname "v") **=> (sum (base,base))), ((base,
Names2.Fixname "v") **=> (sum ((prod (base, base)), (sum ((prod (base, base)), (prod (base, base))
)))))))), ((base,Names2.Fixname "v") **=> (sum (((base,Names2.Fixname "u") **=> (sum ((prod (base,
 (prod (base, base)))), (prod (base, (prod (base, base)))))))), ((base,Names2.Fixname "u") **=> (su
m ((prod (base, (prod (base, (prod (base, base)))))), (sum ((prod (base, (prod (base, (prod (base,
 base)))))), (prod (base, (prod (base, (prod (base, base))))))))))))))))) comp3);;

- : normal = (fun a -> a)
```

FIG. 7.5 – η-réduction du résultat de la résidualisation de `comp3`.

7.4 Insertion de `let` et mémoïsation

J'ai présenté avec Olivier Danvy dans [11] une application de l'optimisation basée sur la condition ♠ en utilisant le procédé d'insertion d'instructions `let`. Grâce à l'utilisation de mémo-fonctions, nous obtenons un évaluateur partiel « complètement paresseux », qui n'évalue jamais deux fois le même sous-terme. Ce travail est bien entendu valable uniquement pour un langage sans effets de bord, puisque si l'application d'une fonction produit un effet, l'application doit avoir lieu autant de fois dans le programme résiduel que dans l'original.

Conclusion

Le travail exposé dans cette thèse se place dans le cadre de la recherche effectuée dans le domaine des isomorphismes de types, thématique qui forme le fil conducteur de toute la thèse.

L'idée de base qui sous-tend ce domaine de recherche est très simple : on cherche à savoir quand une donnée a de type A peut être « codée » par un programme (ou fonction) f sous la forme d'une donnée $f(a)$ de type B « sans perte d'information », c'est à dire qu'il doit être possible après de « décoder » $f(a)$ à l'aide d'un programme (ou fonction) g tel que $g(f(a))$ soit a lui-même.

Cette notion a plusieurs applications qui deviennent de plus en plus intéressantes avec la masse grandissante d'information disponible en ligne : la recherche de composantes dans des bibliothèques logicielles [73, 36, 85], la recherche de théorèmes dans des bibliothèques de preuves [33, 1], la génération automatique de code d'adaptation [5, 7] entre composantes logicielles, le changement de représentation des données pour faciliter l'écriture de programmes [82].

Pour pouvoir répondre à ces besoins, il est important de connaître et savoir décider les isomorphismes de types dans le cadre des systèmes de types complexes qui sont utilisés dans les langages modernes : les types fonctionnels, les produits et enregistrements, les types variants, les types récursifs.

Pourtant, nos connaissances étaient remarquablement limitées en dehors du cas heureux des catégories cartésiennes fermées, et notamment, on ne savait rien à propos des difficultés introduites par le type somme et le type vide. Le résultat de non-fini-axiomatisabilité de Gurevič pour les entiers était connu depuis dix ans, mais on ne savait pas s'il pouvait ou pas se transposer dans le théorie des types : il n'était pas évident de construire les termes inversibles correspondant à ces égalités, en l'absence de tout système de réécriture pour le lambda calcul avec types sommes, et en l'absence même d'une notion raisonnable de forme normale pour les termes.

La quête d'une solution nous a donc amené à développer des outils théoriques pour définir une notion de forme normale de termes, et pratiques, pour pouvoir calculer efficacement ces formes normales, dont l'intérêt dépasse le cadre initial que nous nous étions posés. D'un coté la fructueuse collaboration avec Marcelo Fiore a débouché sur une notion de forme normale canonique pour le λ-calcul avec sommes binaires. De l'autre, la dimension des termes qui ressortent des égalités de Gurevič a motivé le développement d'une application originale de l'évaluation partielle dirigée par les types, qui incorpore les idées de ces formes canoniques pour simplifier énormement les termes produits par l'approche traditionnelle de l'évaluation partielle des types sommes. Ces idées pourront également apporter des optimisations non négligeables dans certains cas pour l'évaluation partielle de programmes sans effets de bords.

On aurait pu espérer que le remarquable parallèle avec les identités entières, qui était valable sans la somme, s'arrêterait là en nous permettant de trouver une axiomatisation finie, qui aurait eu une utilité pratique immédiate, puisqu'elle nous aurait permis de déterminer simplement si deux types sont isomorphes ou non. Mais nous avons montré que la situation est bien plus complexe : d'un coté, les isomorphismes de types avec somme ne sont pas finiment axiomatisables, parce que les identités de Gurevič ont un contenu calculatoire qui est capturé par les termes inversibles exhibés au chapitre 4, alors que le parallèle avec les entiers n'est plus vrai en présence du zéro ; cependant, la question reste ouverte pour le cas de la somme et sans le zéro.
Beaucoup de travail reste à faire pour clarifier la relation entre la longue série d'articles de théorie des nombres, initiée par la conjecture des égalités du lycée de Tarski, et les isomorphismes de types dans les catégories bi-cartésiennes fermées, et plus en général le domaine de l'« Objective Number Theory » [76], mais nous croyons que cette thèse apporte une prémière contribution significative dans cette direction.

En même temps, le résultat négatif sur l'axiomatisabilité des isomorphismes avec somme donne une justification supplémentaire à la recherche de sous-systèmes comme celui défini par les isomorphismes linéaires. Dans ce domaine, nous avons montré comment l'utilisation des réseaux de preuve de la logique linéaire permet d'obtenir d'une façon remarquablement simple et élégante une preuve de complétude pour la théorie des isomorphismes de MLL. Le jour où l'on disposera d'une notion de réseau additif qui ne gardent pas de trace de séquentialité, ce qui n'est pas encore connu, cette technique devrait pouvoir s'étendre au cas des connecteurs additifs et l'on peut conjecturer que les isomorphismes de MALL sont obtenus par des règles de commutativité, associativité, distributivité et éléments neutres.

Bibliographie

[1] Mathematical knowledge management. see `http ://www.cs.unibo.it/MKM03/`.

[2] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. Normalization by evaluation for typed lambda calculus with coproducts. In Joseph Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[3] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.

[4] A. Andreev and Sergei Soloviev. A deciding algorithm for linear isomorphism of types with complexity $o(nlog^2(n))$. In Eugenio Moggi and Giuseppe Rossolini, editors, *Category Theory and Computer Science*, number 1290 in LNCS, pages 197–210, 1997.

[5] Maria-Virginia Aponte and Roberto Di Cosmo. Type isomorphisms for module signatures. In *Programming Languages Implementation and Logic Programming (PLILP)*, volume 1140 of *Lecture Notes in Computer Science*, pages 334–346. Springer-Verlag, 1996.

[6] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures*. The MIT Press, 1991.

[7] Joshua Auerbach and Mark C. Chu-Carroll. The mockingbird system : A compiler-base approach to maximally interoperable distributed programming. *IBM Research Report RC20718*, 1997.

[8] V. Balat and D. Galmiche. *Labelled Deduction*, volume 17 of *Applied Logic Series*, chapter Labelled Proof Systems for Intuitionistic Provability. Kluwer Academic Publishers, 2000.

[9] Vincent Balat. Évaluation partielle dirigé par les types en Objective Caml. Rapport de stage de maîtrise, École normale supérieure de Lyon / BRICS (Århus, Danemark), 1997.

[10] Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998. Springer-Verlag.

[11] Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In *ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GCSE/-SAIG)*, number 2487 in Lecture Notes in Computer Science, Pittsburgh, USA, October 2002.

[12] Vincent Balat and Roberto Di Cosmo. A linear logical view of linear type isomorphisms. In *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 250–265. Springer-Verlag, 1999.

[13] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation for typed lambda calculus with sums via Grothendieck logical relations. Manuscript, 2002.

[14] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In Gordon D. Plotkin, editor, *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 2002. IEEE Computer Society Press.

[15] Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001*, number 2030 in Lecture Notes in Computer Science, pages 57–71, Genova, Italy, April 2001. Springer-Verlag.

[16] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[17] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[18] Gérard Berry and Pierre-Louis Curien. Theory and practice of sequential algorithms : the kernel of the applicative language CDS. In M. Nivat and J. Reynolds, editors, *Algebraic methods in semantics*, pages 35–87. Cambridge University Press, 1985.

[19] Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2) :231–247, 1992.

[20] Kim Bruce and Giuseppe Longo. Provable isomorphisms and domain equations in models of typed languages. *ACM Symposium on Theory of Computing (STOC 85)*, 1985.

[21] Stanley Burris and Simon Lee. Tarski's high school identities. *American Mathematical Monthly*, 100(3) :231–236, 1993.

[22] Catarina Coquand. From semantics to rules : a machine assisted analysis. In *CSL'94*, volume 832 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[23] Pierre Crégut. An abstract machine for lambda-terms normalization. In *Lisp and Functional Programming*, pages 333–340. ACM Press, 1990.

[24] Pierre Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. PhD thesis, Université Paris 7, 1991.

[25] R. Crole. *Categories for Types*. Cambridge University Press, 1994.

[26] Pierre-Louis Curien and Roberto Di Cosmo. A confluent reduction system for the λ-calculus with surjective pairing and terminal object. In Leach, Monien, and Artalejo, editors, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, volume 510 of *Lecture Notes in Computer Science*, pages 291–302. Springer-Verlag, July 1991.

[27] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[28] Olivier Danvy. A simple solution to type specialization. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Conference on Automata, Languages and Programming (ICALP)*, number 1443 in Lecture Notes in Computer Science, pages 908–917, 1998.

[29] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory ; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[30] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[31] Olivier Danvy and Andrzej Filinski. Representing control, a study of the cps transformation. In *Mathematical Structures in Computer Science*, number 2(4), pages 361–191, December 1992.

[32] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In *European Symposium On Programming*, pages 224–242, 1999.

[33] D. Delahaye, Roberto Di Cosmo, and B. Werner. Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes. In *PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, 1997.

[34] Roberto Di Cosmo. Type isomorphisms in a type assignment framework. In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 200–210, Albuquerque, New Mexico, January 1992. ACM Press.

[35] Roberto Di Cosmo. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, 3(3) :485–525, 1993.

[36] Roberto Di Cosmo. *Isomorphisms of types : from λ-calculus to information retrieval and language design*. Birkhauser, 1995. ISBN-0-8176-3763-X.

[37] Roberto Di Cosmo. Second order isomorphic types. A proof theoretic study on second order λ-calculus with surjective pairing and terminal object. *Information and Computation*, pages 176–201, June 1995.

[38] Roberto Di Cosmo and Delia Kesner. A confluent reduction for the extensional typed λ-calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, editor, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, volume 700 of *Lecture Notes in Computer Science*, pages 645–656. Springer-Verlag, July 1993.

[39] Roberto Di Cosmo and Delia Kesner. Combining first order algebraic rewriting systems, recursion and extensional lambda calculi. In Serge Abiteboul and Eli Shamir, editors, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, volume 820 of *Lecture Notes in Computer Science*, pages 462–472. Springer-Verlag, July 1994.

[40] Roberto Di Cosmo and Delia Kesner. Simulating expansions without expansions. *Mathematical Structures in Computer Science*, 4 :1–48, 1994.

[41] John Doner and Alfred Tarski. An extended arithmetic of ordinal numbers. *Fundamenta Mathematica*, 65 :95–127, 1969.

[42] Kosta Dosen and Zoran Petric. Isomorphic objects in symmetric monoidal closed categories. *Mathematical Structures in Computer Science*, 7(6) :639–662, 1997.

[43] Daniel Dougherty and Ramesh Subrahmanyam. Equality between functionals in the presence of coproducts. volume 157, pages 52–83, 2000. *Information and Computation*. An earlier version appeared in *Proceedings of the tenth Annual IEEE Symposium on Logic in Computer Science*.

[44] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the twenty-first annual ACM Symposium on Principles Of Programming Languages*, pages 446–457. ACM Press, September 1994.

[45] Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *4th International Conference on Principles and Practice of Declarative Programming (PPDP 2002)*. ACM Press, 2002.

[46] Marcelo Fiore and Alex Simpson. Lambda-definability with sums via Grothendieck logical relations. In *Typed Lambda Calculus and Applications*, number 1581 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[47] Neil Ghani. *Adjoint rewriting*. PhD thesis, University of Edinburgh, 1995.

[48] Neil Ghani. $\beta\eta$-equality for coproducts. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculus and Applications*, volume 902 of *Lecture Notes in Computer Science*, April 1995.

[49] Joseph Gil. Subtyping arithmetical types. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 276–289, London, UK, 2001.

[50] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1) :1–102, 1987.

[51] Jean-Yves Girard. Proof nets : the parallel syntax for proof theory. *Logic and Algebra*, 1995.

[52] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, June 1995.

[53] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. Return types for functional continuations. A preliminary version appeared as [52], 1998.

[54] R. Gurevič. Equational theory of positive numbers with exponentiation. *Proceedings of the American Mathematical Society*, 94(1) :135–141, May 1985.

[55] R. Gurevič. Equational theory of positive numbers with exponentiation is not finitely axiomatizable. *Annals of Pure and Applied Logic*, 49 :1–30, 1990.

[56] Leon Henkin. The logic of equality. *American Mathematical Monthly*, 84 :597–612, October 1977.

[57] C. W. Henson and L. A. Rubel. Some applications of nevanlinna theory to mathematical logic : Identities of exponential functions. *Trans. American Mathematical Society*, 282(1) :1–32, March 1984.

[58] Achim Jung and Jerzy Tiuryn. A new characterization of lambda-definability. In J. Groote M. Bezem, editor, *Typed Lambda Calculus and Applications*, number 664 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[59] Jean-Louis Krivine. *Lambda-calcul, types et modèles*. Masson, 1990.

[60] J. Lambek and P. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1986.

[61] Dominique Larchey-Wendling, Daniel Mery, and Didier Galmiche. Strip : Structural sharing for efficient proof-search. In *International Joint Conference on Automated Reasoning, IJCAR 2001*, volume 2083 of *LNAI*, pages 696–700, 2001.

[62] Olivier Laurent. A game semantics approach to isomorphisms of types. First *Workshop on Isomorphisms of Types*, Toulouse, France, (paper in preparation), November 2002.

[63] Olivier Laurent. *Étude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, March 2002.

[64] Xavier Leroy and Benjamin Grégoire. A compiled implementation of strong reduction. In *International Conference on Functional Programming*, 2002.

[65] Saunders Mac Lane. *Categories for the working mathematician*, volume 5 of *GTM*. Springer, 1971.

[66] A. Macintyre. The laws of exponentiation. In C. Berline, K. McAloon, and J.-P. Ressayre, editors, *Model Theory and Arithmetic*, volume 890 of *Lecture Notes in Mathematics*, pages 185–197. Springer-Verlag, 1981.

[67] Charles F. Martin. Axiomatic bases for equational theories of natural numbers. *Notices of the Am. Math. Soc.*, 19(7) :778, 1972.

[68] Simone Martini. Provable isomorphisms, strong equivalence and realizability. In Marchetti-Spaccamela et al., editor, *Proceedings of the Fourth Italian Conference on Theoretical Computer Science*, pages 258–268. Word Scientific Publishing Co, 1992.

[69] Michel Parigot. $\lambda - \mu$-calculus : an algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Deduction*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201, 1992.

[70] Gordon Plotkin. Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry : Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, 1980.

[71] Mikael Rittri. Retrieving library identifiers by equational matching of types. In Mark E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, number 449 in Lecture Notes in Computer Science, pages 603–617, Kaiserslautern, Germany, July 1990. Springer-Verlag.

[72] Mikael Rittri. *Searching program libraries by type and proving compiler correctness by bisimulation*. PhD thesis, University of Göteborg, Göteborg, Sweden, 1990.

[73] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1) :71–89, 1991.

[74] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. The MIT Press, Cambridge, Massachusetts ; London, England, second edition, 1988.

[75] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. *Journal of Functional Programming*, 1(2) :191–211, 1991.

[76] Stephen H. Schanuel. Objective number theory and the retract chain condition. *Journal of Pure and Applied Algebra*, 154 :295–298, 2000.

[77] Sergei V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3) :1387–1400, 1983.

[78] Sergei V. Soloviev. A complete axiom system for isomorphism of types in closed categories. In A. Voronkov, editor, *Logic Programming and Automated Reasoning, 4th International Conference*, volume 698 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 360–371, St. Petersburg, Russia, 1993. Springer-Verlag.

[79] P. Taylor. *Practical Foundations of Mathematics*, volume 59 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1999.

[80] Lorenzo Tortora de Falco. The additive multiboxes. *Annals of Pure and Applied Logic*, 120 :65–102, January 2003.

[81] D. Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. In *Mathematical Structures in Computer Science*, volume 8, pages 153–192, 1997.

[82] Philip Wadler. Views : a way for pattern matching to cohabit with data abstraction. In *Proceedings of 14th ACM Symposium on Principles of Programming Languages*, Munich, Germany, 1987. Association for Computing Machinery.

[83] A. J. Wilkie. On exponentiation — A solution to Tarski's high school algebra problem. Math. Inst. Oxford University (preprint), 1981.

[84] Alex J. Wilkie. On exponentiation – a solution to Tarski's high school algebra problem. *Quaderni di Matematica*, 2001. To appear. Mathematical Institute, University of Oxford (preprint).

[85] Boris Yakobowski. Matching de modules modulo isomorphismes. Rapport de stage, École Normale Supérieure de Lyon / INRIA Rocquencourt, 2002.

[86] Zhe Yang. Encoding types in ML-like languages. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300. ACM Press, 1998. Extended version available as the technical report BRICS RS-98-9.

Index des symboles

ι_i (injection dans les catégories) 28
$AC(\otimes, \parr)$ **81**
$ACI(\otimes, \parr)$ **81**
Φ **62**
$eval_{A,B}$ **28**
$- \circ h$ **32**
$\lambda^1_{\beta\eta}$ **16**
$\lambda^1_{\times 1\beta\eta}$ **16**
$\lambda^2_{\beta\eta}$ **16**
$\lambda^2_{\times 1\beta\eta}$ **16**
$\bot_\tau(t)$ **47**
$case\,(t,\ x_1 : \tau_1.\ t_1,\ x_2 : \tau_2.\ t_2)$ **47**
$\lambda((x_1, x_2), x_3 \ldots).\ M$ **64**
$\lambda(x,\ y).\ M$ **64**
$\langle f, g\rangle$ **27**
$\mathfrak{L}_+[\mathfrak{C}]$ (CCC libre avec co-produits engendrée sur $\mathfrak{C}$) **94**
$\mathfrak{L}_o[\mathfrak{C}]$ (CCC libre avec objet initial engendrée sur $\mathfrak{C}$) **94**
$\mathfrak{L}[\mathfrak{C}]$ (CCC libre engendrée sur $\mathfrak{C}$) **94**
$Th^2_{\times 1}$,Th^2,$Th^1_{\times 1}$,Th^1 **16**
$=_{\beta\eta}$ ($\beta\eta$-équivalence) **50**
$\mathfrak{L}_{o,+}[\mathfrak{C}]$ (biCCC libre engendrée sur $\mathfrak{C}$) **94**
Λ **28, 34**
¡A **28**
!A **27**
F_{mor} **31**
F_{obj} **31**
Λ^{-1} **28**
$f \times g$ **28**
$\binom{f}{g}$ **28**
$\Gamma^\times$ **54**
$=_{\beta\eta'}$ **50**
$\Lambda_{\times+10\beta\eta}$ **56**
$\mathfrak{L}_{o,+}$ **67**
$\mathfrak{L}$ **58**
$(\lambda x.\ t)$ (abstraction) **44**
$(t_1\ \ t_2)$ (application) **44**
$(t_1\ @\ t_2)$ (application) **44**
biCCC (catégorie bi-cartésienne fermée) **29**
CCC (catégorie cartésienne fermée) **29**
$\mathfrak{D}^{\mathfrak{C}}$ (catégorie de foncteurs) **32**
$\mathfrak{Cat}$ (catégorie des catégories) **31**
$\mathfrak{Ens}^{\mathfrak{C}^{op}}$ (catégorie des pré-faisceaux) **32**
$\mathfrak{C}^{op}$ (catégorie duale) **26**
$[M]$ (classe d'équivalence) **54**
$codom_{\mathfrak{C}}(f)$ (co-domaine de f) **25**
$f \circ g$ (composition de λ-termes) **52**
$f \circ g$ (composition de flèches) **26**
$\Gamma \mid \Xi$ (contexte contraint) **127**
$(t_1,\ t_2)$ (couple de termes) **47**
$dom_{\mathfrak{C}}(f)$ (domaine de f) **25**
$\wedge$ (et) **48**
$\mathfrak{G}(A,\ B)$ (flèches de A vers B) **25**
Δ (foncteur diagonal) **34**
$\mathfrak{C}(-,\ B)$ (hom-foncteur contravariant) **32**
id_A, id (identité dans une catégorie) **26**
$\Rightarrow$ (implication intuitionniste) **48**
$in_i^{\tau_1,\tau_2}$, in_i (injection dans le λ-calcul) **47**
$\cong$ (isomorphisme dans une catégorie) **26**
$\Gamma \vdash t : \tau$ (jugement de typage) **47**
$\Lambda_{\times 1\beta\eta}$ (modèle de λ-termes) **54**
$\rightarrowtail$ (monomorphisme) **26**
$Mor_{\mathfrak{G}}$ (morphismes de $\mathfrak{G}$) **25**
$()$ (nil) **47**
$Obj_{\mathfrak{G}}$ (objets de $\mathfrak{G}$) **25**
$\vee$ (ou) **48**
$\mathbf{y}$ (plongement de Yoneda) **33**
$\hookrightarrow$ (plongement) **31**
$A_1 \times A_2 \times \ldots A_n$ (produit n-aire) **54**
$proj_i^{\tau_1,\tau_2}$, $proj_i$ (projection dans le λ-calcul) **47**
$\pi_i^{A,B}$, π_i (projection dans les catégories) **27**
$t\,[u/x]$ (substitution) **14, 45**
$\mathtt{1}$ (type unité – produit vide) **46**
$\mathtt{0}$ (type vide – somme vide) **46**
$FV(t)$ (variables libres) **45**
$(t_1,\ t_2)$ **47**
$Th^2_{\times 1}$,Th^2,$Th^1_{\times 1}$,Th^1,Th^{ML} 17
& (« avec ») **78**

⅋ (« par »)..... **78**
⊕ (« plus »)..... **78**
⊗ (« tenseur »)..... **78**

Index

$(\mathfrak{C}, K)$-relation de Grothendieck **117**
α-équivalence **44**
β-réduction faible **139**
β-réduction **14**, **45**, 49, **49**
$\beta\eta$ **46**
$\beta\eta$-équivalence **14**, **50**
η-conversion **46**
η-expansion **46**, **49**
η-expansion dirigée par les types **142**
η-réduction **14**, **46**, **49**
λ-abstraction **44**
λ-calcul 44, 47
λ-calcul simplement typé
  avec somme et zéro **48**
  sans somme ni zéro **48**
λ-définissabilité **113**
λ-terme **44**

A

abstraction 13, 43, 44
affaiblissement 58
appel
  par nom **142**
  par valeur **142**
application 13, **21**, 26, 43, **44**
arbre de preuve 58
arité **113**, **117**
avec **78**

B

base (d'un cône) **29**
bipartite **83**, **91**
bytecode **140**

C

calcul des séquents **48**
Caml 13
capture **44**
case 47
catégoricité **14**
catégorie **26**
  bi-cartésienne **29**
  bi-cartésienne fermée **29**
  cartésienne **27**
  des catégories **31**
  des contextes **114**
  des ordinaux finis **95**
  des pré-faisceaux **32**
  des relations de Grothendieck **117**
  distributive **29**, 98, 99
  duale **26**
  libre **27**
  localement petite **26**
  petite **26**
  produit **26**
Church-Rosser 45
co-cartésienne **28**
co-cône **31**
co-domaine **25**
co-égaliseur **31**
co-limite **31**
co-produit **28**, 47
  stable **129**
code natif **140**
commute **27**
complète **54**
complètement abstraite **54**
composition **26**
composée
  horizontale **34**
  verticale **32**
cône **29**
  universel **29**
confluent **45**
constructeurs **47**
contexte
  inconsistant **48**, **123**
contexte contraint **128**

contexte de typage **47**
conversions commutatives **161**
corps d'une abstraction **44**
correct **54**
correction **53**
correspondance de Curry-Howard 15, 48
couple **21**, 47
cover 116
curryfication **15**
curryfication/dé-curryfication **28**

D
déduction naturelle **48**
destructeurs **47**
diagrammes **27**
domaine **25**
dynamique **140, 141**

E
effet de bord **140**
égaliseur **30**, 130
égalité
de Martin 96
de Wilkie 97
égalités
de Gurevič 97
de Wilkie-Gurevič généralisées 100
du lycée 17, 96
élimination des coupures **48, 80**
embedding 31
ensemble de définition **21**
entiers de Church **43**, 152
entiers naturels **21**
entiers relatifs **21**
épimorphisme **26**
equalizer 30
équivalence forte **15**
equivalence observationnelle
équivalence observationnelle **54**
eval **28**
évaluation 43
évaluation partielle **140, 144**
évaluation partielle
dirigée par les types **139**
exponentielle **28**
exposant **28**
extensionalité **14**

F
faithful 31
famille **21**
fermée **28**
flèche
définissable **114**
neutre **115**
normale **115**
flèches **25**
foncteur **31**
fidèle **31**
plein **31**
pleinement fidèle **31**
foncteur diagonal **34**
fonction **21**
fonction totale **21**
forme normale **18, 45**
de tête faible **140**
extensionnelle **113**
formules
isomorphes **80**
isomorphes (logique linéaire) **82**
fortement normalisant **45**
full 31
full and faithful 31

G
génération de bytecode à l'exécution **151**
graphe orienté **25**

H
Haskell 13
hom-foncteur contravariant **32**

I
identité **26**
inconsistant **48**
initial **28**
injection **28**, 47
interprétation 44, **53**
stable **129**
isomorphisme
de types 14
entre catégories **31**
entre objets d'une catégorie **26**
sémantique **14**

J

jugement de typage ... 47

L

langage
    fonctionnel ... 13, 43
lien
    axiome ... **80**
liens ... **79**
limite ... **29**
Lisp ... 13
logique
    classique ... **48**
    intuitionniste ... **48**
    linéaire ... 78

M

machine abstraite ... **140**
Martin ... 96
ML ... 13
modèle ... **53**
    biCCC ... 57
    abstrait ... 44
    des λ-termes ... 54
    syntaxique ... 44, **54**
monomorphisme ... **26**
morphismes ... **25**

N

$\mathbb{N}$ ... **21**
$\mathbb{N}^*$ ... **21**
naturellement en X ... **33**
NBE ... **139**
nil ... 47
niveau
    dynamique ... 140
niveau
    statique ... 140
non-ambiguë ... **84**
`normal` ... 145, **150**
normalisation ... 49
    forte ... **140**
    par évaluation ... **139**
    « offline » ... **141**
    « online » ... **141**
normalisation by evaluation ... 139
notation de Church du λ-calcul ... 47
notation de Curry du λ-calcul ... 47

O

objet (d'une catégorie) ... **25**
Objective Caml ... 139, 141, 142, 146, 149, 151
occurrence
    libre ... **44**
    liée ... **44**

P

paire ... **21**, 47
paire critique ... **45**
par ... **78**
partial function ... 21
permutation ... 58
plongement ... **31**
plongement de Yoneda ... **33**
plus ... **78**
problème des égalités du lycée ... **95**
produit ... **27**, 47
    fibré ... **30**, 130
projection
    (λ-calcul) ... 47
    (catégories) ... **27**
prompt ... **162**
propriété de Church-Rosser ... **45**
pré-réseau ... **79**
pullback ... 30
pushout ... 31

Q

quasi-quote ... **143**
quote ... **143**

R

recouvrement ... 116, **116**
redex ... **45**
réduction ... **45**
réécriture ... 43, 45
réflexion ... **143**
règle
    de réécriture ... **45**
règles
    additives ... **78**

d'introduction **47**
d'élimination **47**
multiplicatives **78**
structurelles 58
réification **143**
relation de Grothendieck 116
relation logique de Kripke **113**
relations de Kripke **115**
relations logiques de Grothendieck **121**
relations logiques de Kripke **115**
renommage **84**
renommage pour R **84**
renommages **128**
réseau
de preuve **80**
simple **81**
simple identité **82**
reset 19, **146**
résidualisation **143**
runtime code generation 151

S

Scheme 13, 147
sémantique
axiomatique **53**
dénotationnelle **53**
opérationnelle **53**
séquent 48, 54
shift 19, **146**
simplifiée **90**
site **116**
somme **28**
amalgamée **31**
forte **49**
stable 129
sommet **29**
soundness 53
SP **49**
statique **140, 141**
strict **41**
structure de preuve **79**
substitution 14, 45, **84**
Surjective Pairing **49**
système de réécriture **45**

T

Tarski 96
Tarski's high school algebra problem 95
TDPE **139**
tenseur **78**
split **80**
terme
bien typé **47**
clos **45**
dynamique 141
neutre **115, 123**
normal **115, 123**
statique 141
terminal **27**
théorème S_n^m de Kleene 15, 144
topologie de Grothendieck 115, **116**
transformation naturelle **32**
translation 21
typage **14**, 44
type
atomique **46**
de base **46**
initial 47
terminal 47
unité 47
vide 47
type-directed partial evaluation 139
types
isomorphes **14**

U

unquote **143**

V

valeur **140**
variable 13, 43, **44**
dynamique **141**
libre **45**
statique **141**

Z

$\mathbb{Z}$ **21**

Une étude des sommes fortes : isomorphismes et formes normales

Le but de cette thèse est d'étudier la somme et le zéro dans deux principaux cadres : les isomorphismes de types et la normalisation de λ-termes. Les isomorphismes de type avaient déjà été étudiés dans le cadre du λ-calcul simplement typé avec paires surjectives mais sans somme. Pour aborder le cas avec somme et zéro, j'ai commencé par restreindre l'étude au cas des isomorphismes linéaires, dans le cadre de la logique linéaire, ce qui a conduit à une caractérisation remarquablement simple de ces isomorphismes, obtenue grâce à une méthode syntaxique sur les réseaux de preuve.

Le cadre plus général de la logique intuitionniste correspond au problème ouvert de la caractérisation des isomorphismes dans les catégories bi-cartésiennes fermées. J'ai pu apporter une contribution à cette étude en montrant qu'il n'y a pas d'axiomatisation finie de ces isomorphismes. Pour cela, j'ai tiré partie de travaux en théorie des nombres portant sur un problème énoncé par Alfred Tarski et connu sous le nom du « problème des égalités du lycée ».

Pendant tout ce travail s'est posé le problème de trouver une forme canonique pour représenter les λ-termes, que ce soit dans le but de nier l'existence d'un isomorphisme par une étude de cas sur la forme du terme, ou pour vérifier leur existence dans le cas des fonctions très complexes que j'étais amené à manipuler. Cette réflexion a abouti à poser une définition « extensionnelle » de forme normale pour le λ-calcul avec somme et zéro, obtenue par des méthodes catégoriques grâce aux relations logiques de Grothendieck.

Enfin, j'ai pu obtenir une version « intentionnelle » de ce résultat en utilisant la normalisation par évaluation : en adaptant la technique d'*évaluation partielle dirigée par les types*, il est possible de produire un résultat dans cette forme normale, ce qui en réduit considérablement la taille dans le cas des isomorphismes de types considérés auparavant.

MOTS-CLÉS : λ-calcul, types, catégories, isomorphismes, somme, co-produit, zéro, objet initial, formes normales, modèles, logique linéaire multiplicative, égalités arithmétiques, problème des égalités du lycée de Tarski, relations logiques de Grothendieck, normalisation par évaluation, évaluation partielle dirigée par les types, opérateurs de contrôle, *Objective Caml*

ENGLISH TITLE: A study of strong sums: isomorphisms and normal forms
ENGLISH ABSTRACT: see page 7

DISCIPLINE : Informatique
LABORATOIRE : Laboratoire *Preuves, Programmes, Systèmes*
Université Paris 7 - Denis Diderot
Case 7014
2, place Jussieu
75251 PARIS CEDEX 05