Nancy-Université
Université Henri Poincaré

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

Toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

➢ Contact SCD Nancy 1 : theses.sciences@scd.uhp-nancy.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

Nancy-Université

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Automatisation de la Construction Sémantique dans le Lambda Calcul Simplement Typé avec plusieurs Types de base

THÈSE

présentée et soutenue publiquement le 21 octobre 2008

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Sébastien Hinderer

Composition du jury

Président :	Claude Godart	Professeur, Université Henri Poincaré, Nancy, France
Rapporteurs :	Christian Retoré	Professeur, Université Bordeaux 1, Bordeaux, France
	Jacques Jayez	Professeur, Ecole Normale Supérieure des Lettres et Sciences Humaines, Lyon, France
Examinateurs :	Patrick Blackburn (directeur)	Directeur de Recherche, INRIA, Nancy, France
	Nicholas Asher	Directeur de Recherche, CNRS-IRIT, Toulouse, France
	Reinhard Muskens	Professeur, Université de Tilburg, Tilburg, Pays-Bas
	Philippe de Groote	Directeur de Recherche, INRIA, Nancy, France

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria

À tous ceux, innombrables, sans qui cette thèse n'aurait pas vu le jour.

Remerciements

Environnement professionnel

En premier lieu, je voudrais remercier Patrick Blackburn, sous la direction duquel cette thèse a été menée à bien. Travailler avec Patrick a été pour moi à la fois plaisant et enrichissant. Ce fut plaisant grâce à toutes les qualités humaines dont Patrick a su faire preuve dans sa façon d'encadrer la thèse. Celles de ses qualités auxquelles j'ai été le plus sensible sont sa générosité dans sa façon de partager avec moi ses idées, la franchise et l'authenticité dont il a continuellement fait preuve lorsqu'il me donnait son avis sur mon travail, une façon de voir la réalité qui n'a cessé de me surprendre par sa justesse et son optimisme rafraîchissant. Le travail avec Patrick fut également enrichissant, notamment parce que Patrick m'a appris, non seulement des méthodes de travail (comment rédiger une communication scientifique, comment la présenter à l'oral...), mais surtout il m'a appris à m'affranchir de certaines attitudes et habitudes qui sont à mon sens des obstacles à la créativité. Pour remercier Patrick de cet apprentissage, mais aussi pour inspirer, peut-être, ceux qui liront ces lignes, je voudrais citer quelques-unes de ces attitudes que Patrick m'a aidé à reconnaître comme des obstacles.

1. La croyance en l'état de l'art comme pré-requis indispensable. J'ai appris avec Patrick que la recherche ne se fait pas systématiquement en commençant par l'état de l'art et en poursuivant par des contributions personnelles, mais que les secondes peuvent être entreprises en premier, cette inversion pouvant rendre le travail beaucoup plus intéressant et permettant de mieux cibler les travaux qui doivent être étudiés.
2. Expérimenter seulement pour vérifier une hypothèse. J'ai appris qu'il n'est pas nécessaire pour entreprendre une expérimentation d'avoir au préalable formulé une hypothèse à vérifier. Il me semble même qu'une expérimentation entreprise hors du contexte de la vérification d'hypothèses peut se révéler féconde.
3. Ne pas s'intéresser aux choses apparemment simples. J'ai appris que même les tâches qui paraissent simples méritent d'être traitées avec sérieux et que cette attitude humble peut être des plus profitables.
4. L'excès de perfectionnisme. Il m'est souvent arrivé de rester bloqué parce que je voulais trop bien faire. J'ai appris qu'une possibilité pour sortir de ce type d'ornière consiste à se fixer des objectifs moins ambitieux, et par conséquent plus atteignables et plus réalistes.

Après Patrick, la personne avec qui j'ai le plus travaillé pendant cette thèse est Sylvain Pogodalla. Sans son aide, cette thèse ne contiendrait vraisemblablement pas de chapitre sur les grammaires catégorielles abstraites. Je voudrais ici remercier Sylvain pour son incroyable disponibilité, la patience avec laquelle il m'a expliqué certaines notions, certaines preuves, parfois plusieurs fois lorsque c'était nécessaire. Merci également pour les relectures du chapitre en question ainsi que d'autres chapitres de la thèse, relectures toujours attentives, utiles et constructives.

Merci aussi pour m'avoir aidé à maintes reprises à améliorer des présentations orales. Merci enfin pour nos nombreuses discussions, tant scientifiques que non scientifiques et toujours pleines de sens.

Je voudrais ensuite remercier tous les membres du projet Talaris (anciennement Langue et Dialogue) au sein duquel cette thèse a été rédigée, ainsi que tous les membres du projet Calligramme et ceux de l'équipe Paréo (anciennement Prothéo) dans laquelle j'ai effectué mon stage de DEA et avec laquelle j'ai gardé pendant la thèse des relations privilégiées. Que chacun des membres de ces trois projets que j'ai côtoyés pendant le stage de DEA et la thèse soit remercié pour la chaleur de l'accueil qui m'a été réservé, les innombrables marques de gentillesse qui m'ont été témoignées.

À ces remerciements généraux, je voudrais ajouter quelques « mentions spéciales » et remercier quelques personnes en particulier :

- Claude Kirchner : sans lui, peut-être n'aurais-je pas obtenu de bourse de thèse, c'est dire à quel point je lui suis redevable et reconnaissant. Merci à lui de m'avoir soutenu bien que la thèse ne se fasse pas au sein du projet Prothéo qu'il dirigeait alors.
- Carlos Areces : merci de m'avoir aidé à rédiger l'article présenté à la session étudiante d'ESSLLI 2005. Merci à Carlos de m'avoir apporté son aide bien que l'article traitât d'une thématique qui n'était pas la sienne. Merci aussi à Carlos pour sa disponibilité, ainsi que pour la gentillesse et l'efficacité avec laquelle il m'a aidé.
- Claire Gardent : merci de m'avoir permis d'aller présenter mon travail à Varsovie. Je suis profondément reconnaissant à Claire de m'avoir donné l'occasion de me rendre en Pologne, me permettant ainsi d'avoir des échanges scientifiquement inspirants et humainement riches et instructifs.
- Anna Młynarczyk-Alstein : merci pour ses patientes explications quant aux subtilités du système aspectuel des verbes polonais, explications indispensables au développement de tous les travaux présentés dans le chapitre 4.
- Sarah Maarek : merci de m'avoir lu et dicté des choses parfois très complexes, notamment des λ-termes que je n'aurais pu appréhender seul. Merci à Sarah de m'avoir initié aux origamis, ce fut parfois une distraction bien agréable, voire indispensable, pendant la rédaction. Merci enfin à Sarah pour la douceur de sa présence, sa patience d'ange et son incroyable et inspirante capacité d'attention aux autres et à leur bien-être.
- Sylvain Schmitz : il a non seulement relu plusieurs chapitres de la thèse, mais a en outre largement contribué à la mise en page de celle-ci, tant du corps du document que des références bibliographiques. Merci à lui pour de nombreuses discussions portant tantôt sur la recherche, tantôt sur la programmation, et qui ont toujours été riches d'enseignements. Merci enfin pour sa gentillesse, sa discrétion et son ouverture d'esprit qui rendent sa présence agréable.
- Cody Roux : merci à lui pour son aide dans l'étude des systèmes de réécriture présentés au chapitre 6.
- Jackie Lai : merci à elle pour sa présence en B224, et en particulier pour son dynamisme, son humour, sa joie de vivre, sa simplicité, autant de qualités qui ont rendu les journées de travail dans le bureau très agréables.
- Eric Kow : merci à lui pour un nombre incalculable de discussions sur des sujets très variés et qui ont toutes été extrêmement instructives et sources d'évolutions positives. Merci à

Eric de m'avoir fait découvrir « Getting Things Done », un livre qui m'a permis de mettre en place des méthodes de travail efficaces et qui, j'en suis sûr, m'aideront énormément dans les années à venir. Enfin merci à Eric pour son amitié, sa disponibilité, son humour...

- Yannick Parmentier : merci à lui pour son indéfectible amitié, sa réconfortante présence à mes côtés dans les moments difficiles, sa disponibilité à toute épreuve et sa gentillesse exemplaire.
- Dmitry Sustretov : merci à lui pour son humour toujours rafraîchissant et décalé, les trop rares mais passionnantes discussions philosophiques que nous avons eues, en particulier pendant la rédaction, la nuit, sur messagerie instantanée. Merci à Dmitry pour son enthousiasme concernant les mathématiques, la physique et bien d'autres sujets. Merci à Dmitry d'avoir su réveiller ma sensibilité à une certaine beauté abstraite, sensibilité qui avait tendance à s'émousser avec le temps et faute d'être stimulée.
- Guillaume Hoffmann : merci pour les discussions toujours amusantes, ainsi que les références de livres particulièrement intéressants partagées.
- Paul Bedaride : merci à lui pour sa gentillesse et sa générosité surprenante de spontanéité. Merci aussi pour la machine à chocolats chauds apportée au bureau et grâce à laquelle j'ai pu passer plusieurs soirées à rédiger la thèse sans être le moins du monde tiraillé par la faim.
- Corinna Anderson : merci à elle pour la justesse de sa présence, sa gentillesse constante et sa discrétion. Merci aussi pour nos nombreuses discussions, portant là encore sur des sujets très divers et qui furent toujours passionnantes et instructives, grâce à la grande ouverture d'esprit de Corinna.
- Luciana Benotti : merci à elle pour sa gentillesse et son sourire. Merci pour toute l'aide apportée lors de notre participation à EALing 2008, merci pour sa présence à cette occasion, tant de moments agréables partagés.

Jury

Je voudrais ensuite remercier les membres du jury de cette thèse, en commençant par les deux rapporteurs, Christian Retoré et Jacques Jayez, d'une part d'avoir accepté le rôle de rapporteur pour cette thèse, d'autre part pour leur lecture attentive et bienveillante du manuscrit. Merci à eux pour leurs remarques constructives, et en particulier à Christian Retoré pour de nombreux et inspirants échanges téléphoniques.

Je remercie ensuite Claude Godart, président du jury, ainsi que les autres examinateurs : Reinhard Muskens, Nicholas Asher, Philippe de Groote et Patrick Blackburn. Merci à eux d'avoir accepté de faire partie du jury de cette thèse, et merci pour l'intérêt ainsi témoigné au travail qui y est présenté.

Enfin, je souhaite remercier à la fois les rapporteurs et les examinateurs, d'une part pour avoir rendue possible l'organisation de la soutenance dans des délais particulièrement serrés, d'autre part pour avoir posé pendant la soutenance des questions qui m'ont paru particulièrement pertinentes et qui, il me semble, pourraient constituer d'intéressants points de départ pour des recherches ultérieures.

Pour terminer, je voudrais remercier les personnes des services support. À mon sens, leur diligence a largement contribué à la qualité de l'environnement de travail qui m'a été offert

au sein du Loria. Je voudrais donc ici remercier les quelques personnes avec lesquelles j'ai été amené à interagir fréquemment. Ces « mentions spéciales » ne doivent cependant pas faire oublier que mes remerciements s'adressent à toutes les personnes des services support : assistantes, moyens informatiques, services généraux, hôtesses d'accueil et personnels du CROUS et d'Avenance présents au restaurant et à la cafétéria du laboratoire. Que toutes ces personnes soient remerciées pour la promptitude et l'efficacité avec lesquelles elles ont toujours répondu à mes requêtes, ainsi que pour leur constant souci de se montrer utiles. Je voudrais en outre remercier plus particulièrement :

- Nadine Beurné, qui n'a cessé de voler à mon secours, depuis mon arrivée à Nancy et jusqu'à son départ en retraite (et même un peu après…). Merci à elle pour maints services rendus au quotidien, pour son aide à l'obtention du financement initial de cette thèse ainsi qu'au prolongement de celui-ci.
- Isabelle Blanchard. Merci à elle de ne m'avoir jamais refusé son aide. Non seulement Isabelle m'a toujours aidé lorsque j'en ai eu besoin, mais elle l'a de plus fait avec le sourire, avec discrétion et efficacité et avec le souci constant de me rendre la vie aussi simple que possible. Pour ne donner qu'un seul exemple, je citerai la préparation de la soutenance de la thèse, qu'Isabelle a accepté de gérer en dépit de son propre emploi du temps déjà bien chargé.
- Jozefina Sadowska. Merci pour sa prodigieuse efficacité pour scanner des documents. Quel dommage de ne pas avoir eu plus tôt l'idée de recourir à ses précieux services, ainsi qu'à ceux de toutes les autres documentalistes du Loria, tant elles ont su se montrer efficaces dans le traitement de toutes mes demandes de recherches bibliographiques. Je termine cette thèse avec l'impression que le service documentaire du Loria n'est pas suffisamment exploité, ce que je trouve bien dommage compte tenu de sa grande qualité et de son professionnalisme. Pour en revenir à Jozefina, je voudrais également la remercier pour la leçon de polonais qu'elle a bien voulu me donner avant le voyage à Varsovie. Une seule leçon, certes, mais d'une densité incroyable et dont le contenu, partiellement défini par Jozefina elle-même, s'est révélé en parfaite adéquation avec les besoins rencontrés sur place.
- Céline Simon. Nous n'avons que peu interagi, mais ce fut à des moments cruciaux. C'est en effet grâce à Céline que deux des membres du jury qui se trouvaient à l'étranger ont pu être contactés, sans quoi la soutenance n'aurait pu avoir lieu à la date souhaitée. Merci à Céline pour sa détermination, pour son joyeux enthousiasme à être utile et à apporter son aide.
- Chantal Llorens. Merci à elle pour sa présence, son humour, sa patience avec le stagiaire de DEA d'alors, peu familier avec le fonctionnement des équipes de recherche et le rôle des assistantes au sein de celles-ci.
- Josiane Reffort, pour son aide lors de la dernière ligne droite, c'est-à-dire la rédaction du dernier rapport d'avancement et la préparation de la soutenance. Merci à elle pour sa présence.
- Jeanine Souquières, Dominique Méry, Janine Perreau, dont les signatures étaient indispensables pour que la soutenance puisse avoir lieu et qui ont accepté d'accélérer considérablement la procédure. Merci à eux trois pour leurs efforts et leur compréhension.

Environnement extra-professionnel

Je voudrais remercier ici quelques autres personnes dont la réconfortante présence m'a été précieuse :

- Mes parents et grands parents, dont les encouragements n'ont cessé de m'accompagner et de me porter. Il est arrivé à plusieurs reprises que les rendre heureux et fiers de moi soit ma motivation principale pour terminer cette thèse, je veux donc les remercier d'avoir été à l'origine d'une si grande énergie.
- De nombreux amis qui se reconnaîtront et dont l'expérience, les conseils et l'affection furent des plus bénéfiques.
- Jean-Sébastien (non, pas Bach, bien que lui aussi pourrait être remercié), qui m'a donné l'envie d'aller à l'université et dont la compagnie ne cesse de stimuler ma curiosité.
- Élise, dont la nourriture bio, les conseils, l'affection, m'ont à maintes reprises rendu les forces et le sourire qui me manquaient.
- Laure, d'abord pour ses précieux conseils, ensuite pour son amitié, non moins précieuse.
- Aude, qui m'a appris à être plus heureux et qui continue de m'accompagner.
- Martine, dont la bienveillance et la sagesse m'inspirent et qui m'accompagne, elle aussi. Merci à elle pour son étonnante clairvoyance.

Table des matières

1 Introduction **5**
- 1.1 Les origines de la sémantique formelle . . . 6
- 1.2 Quatre développements subséquents aux travaux de Montague . . . 10
  - 1.2.1 Entités abstraites . . . 11
  - 1.2.2 TYn en sémantique formelle . . . 13
  - 1.2.3 Théorie de la Représentation des Discours (DRT) . . . 18
  - 1.2.4 Grammaires Catégorielles Abstraites . . . 23
- 1.3 Sémantique computationnelle . . . 25
- 1.4 Plan de la thèse . . . 28

2 Nessie, un outil pour automatiser le calcul du sens **29**
- 2.1 Introduction à `OCaml` . . . 31
  - 2.1.1 Inférence de type et langage fonctionnel . . . 32
  - 2.1.2 Types de données . . . 33
  - 2.1.3 Modules . . . 38
  - 2.1.4 Foncteurs ou modules paramétrés . . . 42
- 2.2 Description de `Nessie` . . . 46
  - 2.2.1 Termes . . . 46
  - 2.2.2 Lexiques . . . 57
  - 2.2.3 Arbres . . . 64
  - 2.2.4 Algorithme de construction sémantique . . . 67
  - 2.2.5 Fonctionnement . . . 76
  - 2.2.6 Conclusion . . . 77

3 Intégration de Nessie à Curt **79**
- 3.1 Procédure de test . . . 80
- 3.2 Phrases simples . . . 81
  - 3.2.1 Conception d'un lexique pour `Nessie` . . . 81
  - 3.2.2 Construction d'arbres syntaxiques . . . 83
  - 3.2.3 Equivalence de représentations sémantiques . . . 87
  - 3.2.4 Mise en place du test . . . 88
  - 3.2.5 Résultats . . . 89
- 3.3 Coordinations . . . 91
- 3.4 Questions . . . 95
  - 3.4.1 Construction de représentations α-équivalentes à celles de `Curt` . . . 96
  - 3.4.2 Vers des représentations plus pertinentes . . . 99

3.5 Conclusion 102

4 Construction de représentations temporelles pour le polonais 103
4.1 Quelques pré-requis sur le polonais 104
4.1.1 La notion d'aspect 104
4.1.2 Expression de l'aspect et du temps en polonais 105
4.1.3 Groupes nominaux 106
4.2 La proposition de Aalstein (Młynarczyk) 106
4.3 Sémantique des verbes polonais dans TYn 110
4.3.1 Typage et représentation sémantiques des verbes 111
4.3.2 Passage de TY3 à TY4 115
4.3.3 Implantation 116
4.4 Axiomatisation des événements 122
4.4.1 Présentation des axiomes 123
4.4.2 Raffinement de la sélection des axiomes 129
4.5 Génération de modèles non minimaux 134

5 Des formalismes pour calculer la sémantique des Discours 139
5.1 La DRT compositionnelle 140
5.1.1 Extensions du langage des DRSs 141
5.1.2 Dénotation des formules du premier ordre 143
5.1.3 Dénotation des DRSs 145
5.1.4 Calculer des DRS compositionnellement 148
5.1.5 Quelques remarques 155
5.2 Traitement Compositionnel de la Dynamicité 160
5.2.1 Contextes 160
5.2.2 Calcul effectif de représentations sémantiques 163
5.2.3 Liens avec la DRT 171
5.2.4 Liens avec `Nessie` 172

6 Calculer automatiquement la représentation des discours 175
6.1 La grammaire et les arbres syntaxiques associés 176
6.1.1 Amélioration des arbres pour les coordinations 176
6.1.2 Pronoms 179
6.2 Implantation de la DRT compositionnelle de Muskens 180
6.3 Implantation du traitement compositionnel de la dynamicité 188
6.3.1 Lexique 188
6.3.2 Calcul de représentations du sens 190
6.4 Résolution d'anaphores dans TYn 193
6.4.1 Ce que présuppose la proposition de de Groote 193
6.4.2 Une architecture logicielle pour la résolution d'anaphores 194
6.4.3 Approche algorithmique 196
6.4.4 Axiomatisation des relations anaphoriques 199
6.5 Conclusion 201

7 Nessie vu comme une grammaire catégorielle abstraite du second ordre 203
7.1 Définition et propriétés des grammaires catégorielles abstraites 204
7.1.1 Définitions 204
7.1.2 Quelques propriétés remarquables des ACGs 208
7.1.3 Problématique abordée 209
7.2 Épuration de `Nessie` 209
7.2.1 Non prise en compte des constantes d'occurrences 209
7.2.2 Élimination des alias et des macros 210
7.2.3 Élimination des arbres unaires et n-aires 210
7.3 Construction de l'ACG et preuve de la propriété de simulation 211
7.3.1 Notations 211
7.3.2 Construction de $\mathscr{G}$ 212
7.3.3 Preuve d'équivalence 214
7.3.4 Exemple 215
7.4 Conclusion 217

8 Conclusion 219
8.1 Contributions 219
8.2 Pistes pour des recherches ultérieures 223

Bibliographie 227

Chapitre 1

Introduction

Cette thèse étudie une famille de systèmes de types pour la logique d'ordre supérieur appelée TYn. Il s'agit en particulier de savoir si ces systèmes sont utiles pour la construction automatique de représentations sémantiques pour les langues [1]. Les systèmes de la famille TYn (dont le nom signifie théorie des types à n sortes) sont basés sur la logique classique ; ils sont en effet des généralisations par n sortes de l'approche utilisée par Henkin pour la logique d'ordre supérieur. Comme nous allons le voir plus loin, cela signifie que les systèmes TYn sont très différents non seulement des logiques d'ordre supérieur de Richard Montague (et en particulier de sa logique intentionnelle IL), mais aussi du langage des « boîtes » utilisé dans la théorie de la représentation des discours de Hans Kamp, que nous noterons désormais DRT, pour Discourse Representation Theory. Ces deux formalismes (IL et DRT) ont en commun d'avoir été tous deux motivés par la sémantique formelle des langues, tandis que, comme nous le verrons, TYn a émergé à partir de considérations logiques et mathématiques et n'était pas conçu, initialement, pour être utilisé dans des applications en sémantique des langues. Néanmoins, plusieurs sémanticiens considèrent désormais TYn comme le meilleur langage pour représenter la sémantique des langues. En particulier, Reinhard Muskens a expliqué avec éloquence que TYn permet d'exprimer tout ce qui peut être exprimé dans IL ou la DRT et que, de surcroît, il permet de le faire d'une façon plus propre et mieux fondée mathématiquement. Par la suite, d'autres arguments en faveur de l'utilisation de TYn ont été proposés, certains ayant des fondements logiques, tandis que d'autres ont des fondements plus en lien avec les spécificités de la sémantique des langues.

Comme nous l'avons indiqué plus haut, cette thèse traite des conséquences de l'utilisation de TYn sur l'automatisation de la construction de représentations sémantiques, un sujet qui avait été peu étudié jusqu'ici. Afin d'évaluer l'intérêt de TYn pour la construction automatique de représentations sémantiques, nous développons un système appelé `Nessie` qui permet à l'utilisateur de spécifier une forme particulière de TYn, puis de construire des représentations sémantiques dans le langage ainsi spécifié. En utilisant `Nessie`, nous serons en mesure d'évaluer l'intérêt computationnel de TYn, notamment en montrant que les représentations sémantiques exprimées dans ce formalisme sont plus intéressantes que leurs homologues exprimées dans le λ-calcul non typé, dans la mesure où il est possible de produire à partir des informations de typage des axiomes qui se révèlent des plus précieux par exemple lors de la construction de modèles (voir chapitre 4). Nous comparerons ensuite deux stratégies pour la modélisation de la sémantique du discours dans le λ-calcul. Cette implantation en parallèle nous permet d'une

[1] Nous utiliserons systématiquement le terme « langue » pour faire référence à ce que certains appellent « langue naturelle ». Le terme « langage » sera quant à lui utilisé uniquement pour faire référence à des langages formels tels que la logique ou les langages de programmation.

part d'illustrer l'adéquation de `Nessie` avec la tâche pour laquelle il a été conçu, d'autre part d'étudier la pertinence computationnelle de chacune de ces théories. Enfin, nous montrerons que l'expressivité permise par `Nessie` peut être décrite à l'aide d'une classe particulière de grammaires catégorielles abstraites, qui sont un formalisme récent permettant de modéliser l'interface syntaxe-sémantique.

Tous ces points seront développés dans les chapitres suivants de cette thèse. Le but de ce chapitre est de présenter rapidement les pré-requis nécessaires à la compréhension des chapitres suivants. La section 1.1 présente les premiers travaux en sémantique formelle des langues, des origines jusqu'aux travaux de Richard Montague, ces derniers étant également présentés. Dans cette section, l'accent sera mis sur les outils logiques utilisés par Montague, ainsi que sur les idées générales qu'il a introduites. Puis, dans la section 1.2, nous introduisons les quatre développements en sémantique formelle consécutifs aux travaux du Montague et qui sont les plus pertinents pour cette thèse, à savoir l'utilisation d'entités abstraites, l'utilisation de la famille de logiques TYn, la DRT et les Grammaires Catégorielles Abstraites (ACGs). Dans la section 1.3, nous présentons les approches computationnelles de la sémantique qui s'inscrivent dans la tradition de Montague, avant de donner dans la section 1.4 le plan de cette thèse, chapitre par chapitre.

1.1 Les origines de la sémantique formelle

La sémantique formelle est une branche de la linguistique. Il est difficile de résumer toute une branche de la linguistique en quelques mots, mais la déclaration suivante décrit bien ce qui en est selon nous l'idée centrale :

> Essayer de modéliser le sens de textes en donnant des méthodes systématiques permettant de les traduire en des représentations logiques.

Cette idée est extrêmement importante. Le sens des langues est évidemment une chose difficile à modéliser. La logique, quant à elle, est une branche précise et bien maîtrisée de la recherche mathématique. Donc, si une façon systématique de traduire les expressions de la langue en des formules logiques pouvait être trouvée, alors nous obtiendrions un moyen d'accéder à au moins certains des aspects du sens de façon précise. Bien sûr, cette déclaration implique que la sémantique formelle sera l'une des branches les plus abstraites de la linguistique. De plus, le fait qu'elle s'appuie sur des formalismes logiques a pour conséquence que cette discipline sera éminemment technique. L'objectif de cette section est de montrer comment cette idée de la sémantique formelle a vu le jour.

L'histoire de la sémantique formelle est intimement liée à celle de la logique. En effet, la sémantique formelle moderne ainsi que la logique moderne se souchent toutes deux sur les travaux de Gottlob Frege (1848-1925). Pour ce qui est de la logique, Frege introduisit l'utilisation de quantificateurs liant les variables. En sémantique, il introduisit plusieurs concepts (tels que la distinction entre le « sens » et la « référence » d'une expression) qui demeurent importants aujourd'hui ; voir à ce sujet (Frege, 1892; Martinich, 1996). L'influence des travaux de Frege en logique et de ses idées quant aux fondements des mathématiques n'a cessé de croître depuis environ 1900, et ce mouvement se poursuit encore aujourd'hui. Ses travaux en sémantique mirent pour leur part plus longtemps à porter leurs fruits. Bien que des travaux dans ce domaine aient

été proposés, de façon relativement précoce (par exemple par le philosophe Bertrand Russell), il a fallu attendre le milieu du vingtième siècle pour voir apparaître dans ce domaine des avancées notables. Cette longue période sans avancées significatives n'est pas due au hasard. Elle s'explique par l'absence de deux outils indispensables : la notion d'interprétation dans un modèle, et la notion d'une « colle » permettant de construire des représentations logiques. Le premier outil a été proposé en 1933 par Alfred Tarski (1902–1983). C'est en effet lui (Tarski, 1935) qui introduisit la notion de modèles, c'est-à-dire de structures simples et abstraites qui peuvent être comprises comme des images du monde. Il montra également comment interpréter la logique du premier ordre dans de telles structures (une logique du premier ordre est un langage qui utilise des connecteurs booléens tels que $\neg$, $\wedge$, $\vee$ et $\rightarrow$ et dans lequel les quantificateurs $\forall$ et $\exists$ décrivent l'ensemble des individus présents dans un modèle).

Le travail de Tarski était important pour plusieurs raisons. D'une part, il rendit la notion de vérité pour les langages logiques mathématiquement précise. Étant donnés une formule logique et un modèle, la question de savoir si cette formule affirmait quelque chose de vrai quant à ce modèle n'était plus sujette à discussions, c'était devenu une question mathématiquement précise. D'autre part, le travail de Tarski a permis de définir une notion sémantique naturelle d'implication. En d'autres termes, son travail permet de savoir quand est-ce qu'un ensemble de formules de logique du premier ordre Σ a pour conséquence sémantique une formule du premier ordre ϕ, ce que l'on note $\Sigma \models \phi$. Tarski a défini $\Sigma \models \phi$ de la façon suivante : $\Sigma \models \phi$ est vrai si tout modèle satisfaisant toutes les formules de Σ satisfait également ϕ. Pour dire les choses de façon plus intuitive : à chaque fois que l'on parvient à rendre toutes les formules de Σ vraies dans un modèle on est sûr que ϕ est également vraie dans ce modèle.

Une raison plus générale expliquant l'importance des travaux de Tarski tient au caractère si simple et si général des modèles en tant que structures mathématiques. Il suffit en effet pour définir un modèle du premier ordre de se donner un ensemble non vide et une collection de relations sur cet ensemble, ce qui explique pourquoi les modèles sont parfois appelés des structures relationnelles. Ainsi, toutes les structures courantes en mathématiques (telles que les groupes, les anneaux, les corps *etc.*) peuvent être considérées comme des classes particulières de modèles, et la théorie des modèles est rapidement devenue une branche à part entière des mathématiques. D'autres avancées étaient cependant nécessaires avant que les travaux de Tarski puissent être appliqués avec profit à la sémantique des langues.

L'avancée suivante est due à Alonzo Church et Leon Henkin. Leur travail consiste essentiellement à étendre les résultats de Tarski à la logique d'ordre supérieur. Comme nous l'avons dit plus haut, en logique du premier ordre les quantificateurs $\forall$ et $\exists$ portent sur l'ensemble des individus des modèles. En logique d'ordre supérieur, ils peuvent porter sur des entités d'ordre supérieur telles que les fonctions des individus vers les valeurs propositionnelles. En 1940, Alonzo Church (Church, 1940) introduisit une approche à la logique d'ordre supérieur appelée théorie des types simple. Cette approche utilisait l'opérateur λ de Church pour définir toutes les fonctions d'ordre supérieur qui étaient nécessaires. Une décennie plus tard, Leon Henkin (Henkin, 1950), l'un des doctorants de Church, donna une sémantique utilisant la théorie des modèles de Tarski du système de Church en utilisant pour cela une suite récurrente de fonctions sur un ensemble sous-jacent (nous verrons une version plus générale de la sémantique de Henkin lorsque nous introduirons TYn, plus tard dans ce chapitre).

Les outils développés par Tarski, Church et Henkin qui viennent d'être présentés constituent l'essentiel de l'infrastructure logique utilisée en sémantique formelle. La personne qui a su voir l'intérêt et la pertinence de ces outils pour l'étude de la sémantique des langues est Richard Montague (1930–1971). Bien que Montague n'ait écrit que quelques articles sur ce sujet, ceux-ci ont considérablement influencé la recherche qui a eu lieu après leur publication. En particulier, dans ses articles « English as a Formal Language » (Montague, 1970a), « Universal Grammar » (Montague, 1970c), et « The Proper Treatment of Quantification in Ordinary English » (Montague, 1973), Montague dévoile ce que l'on s'accorde désormais à appeler la méthode des fragments. Dans ces articles, Montague a en effet défini des grammaires pour de petits fragments de l'anglais (c'est-à-dire des grammaires qui n'engendrent que quelques phrases) et il a montré comment les phrases engendrées par ces grammaires pouvaient être interprétées dans des modèles. Dans « English as a Formal Language » il a interprété le fragment d'anglais considéré directement (c'est-à-dire sans utiliser une traduction préalable dans une représentation logique intermédiaire) alors que dans « Universal Grammar » et « The Proper Treatment of Quantification in Ordinary English » il a d'abord traduit les phrases en logique d'ordre supérieur. Cette différence demeure cependant marginale lorsqu'on la compare à ce que ces articles ont en commun : dans les trois articles le processus d'interprétation utilisé est complètement explicite. En résumé, les trois articles donnent des algorithmes d'interprétation pour des fragments de l'anglais. Le travail de Montague ne couvre pas tous les aspects de ce que nous aimerions appeler le sens, et les fragments qu'il étudiait étaient relativement réduits, se limitant à quelques constructions syntaxiques. De telles considérations ne devraient cependant pas nous empêcher de remarquer le caractère novateur de ces articles, qui réside dans le mécanisme systématique sous-jacent à la sémantique qu'ils mettent en évidence, tout à fait dans l'esprit de la définition de la sémantique formelle des langues proposée au début de cette section.

Examinons d'un peu plus près la méthode des fragments. Comme nous l'avons dit, elle montre comment associer systématiquement une sémantique aux phrases générées à partir d'un lexique et d'une grammaire. Le rôle du lexique est de répertorier les mots que l'on peut utiliser, ainsi que des informations les concernant, en particulier des indications permettant de leur associer une représentation sémantique. Le rôle de la grammaire est de nous permettre de réaliser l'analyse syntaxique des phrases utilisant les mots du lexique. En d'autres termes, le rôle de la grammaire est d'associer à chaque phrase une structure d'arbre qui peut être utilisée pour guider le processus de construction sémantique. L'une des intuitions majeures de Montague était que le λ-calcul pouvait être utilisé comme langage d'assemblage (« glue » en anglais) pour la mise en œuvre de la construction sémantique. Les représentations sémantiques seraient construites en assemblant les λ-termes fournis dans le lexique grâce aux deux opérations fondamentales du λ-calcul que sont l'application de fonctions et la β-réduction.

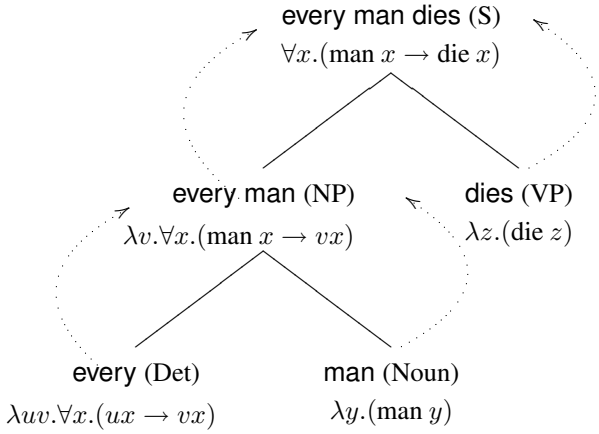
Considérons un exemple. Nous allons construire une représentation sémantique pour « Every man dies ». Nous n'utiliserons ni les représentations originelles de Montague, ni ses notations, mais des représentations et notations simplifiées qui mettront en évidence les idées principales. Supposons que le lexique associe aux mots de notre exemple les λ-termes suivants :

every : $\lambda uv.\forall x.(ux \rightarrow vx)$

man : $\lambda y.(\text{man } y)$

dies : $\lambda z.(\text{die } z)$

De plus, supposons que notre grammaire analyse cette phrase (S) « Every man dies » comme étant constituée du groupe nominal (NP) « every man » suivi d'un groupe verbal (VP) « dies », et qu'elle analyse le groupe nominal « every man » comme étant constitué du déterminant « every » suivi du nom (N) « man ». Compte tenu de ces informations, le processus de construction sémantique peut être illustré par l'arbre suivant. Cet arbre montre la structure syntaxique de la phrase, tandis que les lignes en pointillés montrent comment les représentations sémantiques sont combinées par les applications fonctionnelles et la β-réduction lorsque l'arbre est parcouru des feuilles vers la racine :



Dans ce diagramme, les β-radicaux ont déjà été réduits. Il peut néanmoins être utile de s'intéresser à la β-réduction qui a lieu au nœud racine (S). D'abord nous associons au nœud S l'application dont le foncteur est la représentation du groupe nominal NP et dont l'argument est la représentation du groupe verbal VP :

« every man dies » : $\lambda v.\forall x(\text{man } x \rightarrow vx)\lambda z.\text{die } z$

Par β-réduction, on obtient :

« every man dies » : $\forall x\text{man } x \rightarrow \lambda z.\text{die } zx$

On peut alors β-réduire l'expression $(\lambda z.\text{die } z)x$:

« every man dies » : $\forall x.(\text{man } x \rightarrow \text{die } x)$

qui n'est autre que la représentation apparaissant à la racine de l'arbre vu précédemment. Il est clair que cette représentation est correcte d'un point de vue sémantique. On peut en outre remarquer qu'il s'agit d'une représentation du premier ordre. Autrement dit, elle ne contient aucun résidu d'expressions d'ordre supérieur tels que des λ-abstractions ou des applications fonctionnelles. Cette situation est, en fait, assez représentative. Nous pouvons en effet dire que l'approche de Montague fait un détour par la logique d'ordre supérieur pour pouvoir assembler toutes les pièces du puzzle. Il n'en demeure pas moins que le résultat de cet assemblage est le plus souvent une authentique formule du premier ordre.

Nous n'entrerons pas davantage dans les détails de l'approche de Montague ; beaucoup de ces détails ne sont plus pertinents pour la sémantique formelle moderne. Il nous paraît cependant important d'essayer de résumer quelques-unes des idées générales portées par les travaux

de Montague, car celles-ci sont, encore aujourd'hui, au cœur de la démarche utilisée pour la construction sémantique.

- **Compositionalité.** La compositionalité affirme que « le sens du tout est fonction du sens de ses parties », affirmation à laquelle il est souvent fait référence par l'appellation « principe de Frege ». La méthode des fragments de Montague, dans laquelle les λ-termes donnés dans le lexique sont combinés étape par étape en remontant le long de l'arbre syntaxique au moyen de l'application fonctionnelle et de la β-réduction, a été la première proposition concrète montrant comment le principe de Frege pouvait être mis en œuvre. Le modèle qu'elle fournit demeure fondamental pour la sémantique moderne.
- **Les représentations sont éliminables.** Il est important de comprendre que, pour Richard Montague, ce n'était pas la représentation logique construite par traduction qui était importante. Pour Montague, ce qui était important était que cette représentation logique ait une interprétation au sens de la théorie des modèles. Ainsi, en composant la fonction de traduction et la fonction d'interprétation, on obtient une interprétation dans la théorie des modèles de l'expression d'origine. Montague pensait que, bien que les représentations logiques puissent jouer un rôle intermédiaire utile, elles sont éliminables. Cette idée fut par la suite considérée comme fondamentale par les sémanticiens pendant des années ; comme nous le verrons plus tard dans ce chapitre, l'apparition de la DRT a remis en question cette vision des choses.

Enfin, nous nous devons de dire un mot de la logique utilisée par Montague comme langage intermédiaire. Dans son article le plus célèbre, « The Proper Treatment of Quantification in Ordinary English » (Montague, 1973), Montague a introduit une logique appelée IL, ce qui signifie Intentional Logic. Il s'agissait d'une logique d'ordre supérieur qui utilisait les idées qui constituaient le cœur de l'approche de Henkin-Church mais qui, comme son nom l'indique, utilisait également des opérateurs intentionnels et une sémantique intentionnelle. En d'autres termes, IL combinait la logique d'ordre supérieur classique et ce que l'on appelle de nos jours la logique modale (Blackburn et al., 2001). La logique modale est très différente de la logique classique, tant du premier ordre que d'ordre supérieur. En fait, Montague a intégré cinq opérateurs modaux dans sa logique : $\Box$ (pour gérer la possibilité), F et P pour gérer le futur et le passé, et deux nouveaux opérateurs $^\vee$ et $^\wedge$ pour créer des intentions et des extensions, respectivement. Cette combinaison d'opérateurs modaux et de logique d'ordre supérieur était certes ingénieuse et techniquement habile, mais le système qui a résulté de cette combinaison était complexe, à la fois au niveau syntaxique et au niveau sémantique. Cette complexité a été acceptée pendant bon nombre d'années ; les innovations de Montague étaient intéressantes et leur motivation linguistique était évidente. Mais, comme nous le verrons dans peu de temps, une génération de chercheurs plus tardive a choisi de s'orienter progressivement vers plus de simplicité et de souplesse, à savoir celles offertes par la théorie des types avec n sortes.

1.2 Quatre développements subséquents aux travaux de Montague

Les développements de la sémantique formelle consécutifs aux travaux de Montague sont si nombreux et si variés qu'il n'est pas possible d'en donner ici un compte-rendu détaillé. Nous

renvoyons le lecteur à (Partee, 1997b) et à (Partee, 1997a) pour des discussions intéressantes sur les travaux de Montague et les recherches qu'ils ont inspirées. Quatre de ces développements sont cependant particulièrement importants pour cette thèse, à savoir les entités abstraites, TYn, la DRT et les Grammaires Catégorielles abstraites (ACGs). Nous allons présenter tour à tour chacun de ces développements, en soulignant leurs interactions et leurs liens avec les travaux de Montague.

1.2.1 Entités abstraites

L'un des thèmes dont l'importance n'a cessée de croître dans les travaux consécutifs à ceux de Montague est celui des entités abstraites. Montague lui-même était parfaitement conscient de l'importance de telles entités : ses articles « Pragmatics » (Montague, 1968), « On the Nature of Certain Philosophical Entities » (Montague, 1969) et « Pragmatics and Intensional Logic" (Montague, 1970b) développent des logiques pour gérer certaines classes d'entités abstraites, à savoir les instants du temps et les mondes possibles. Ces entités ne constituent cependant que la partie émergée de l'iceberg pour ce qui est des entités abstraites et de leur utilisation dès que l'on s'intéresse aux langues, et d'une certaine façon ces entités sont les moins surprenantes : la nécessité de disposer d'un modèle du temps apparaît par le biais de la physique, tandis que l'utilisation des mondes possibles a une longue histoire philosophique. De nos jours, un grand nombre d'entités abstraites sont utilisées en sémantique des langues. Pour n'en donner qu'un seul exemple, nous allons considérer celui des événements que nous utiliserons au chapitre 4. Notre propos n'est cependant pas tant de parler d'une entité abstraite particulière que de de montrer à quel point le besoin de pouvoir manipuler des entités abstraites et donc de disposer d'une logique où cela est faisable simplement est prégnant. Le but de cette sous-section est donc de contribuer à motiver la transition vers la théorie des types à n sortes.

Pour comprendre pourquoi on pourrait avoir besoin de manipuler des événements, considérons les phrases suivantes :

– John eats a burger
– John eats a burger with his hands for lunch.
– John eats a burger with his hands for lunch in the room where the suitcase has been hidden.

Comment représenter la sémantique de toutes ces phrases ? Nous pourrions essayer de le faire en utilisant un long prédicat eat disposant de suffisamment d'arguments pour stocker toutes les informations dont nous avons besoin. Par exemple :

$$\textit{eat}(\textit{agent}, \textit{patient}, \textit{instrument}, \textit{meal}, \textit{location})$$

Alors, pour représenter les phrases qui n'ont pas besoin de toutes ces informations, nous pouvons simplement quantifier existentiellement les arguments qui ne sont pas utilisés. Par exemple :

– John eats a burger with his hands for lunch in the room where the suitcase has been hidden. eat(john, burger, hands, lunch, room-suitcase-hidden)
– John eats a burger with his hands for lunch $\exists x$. eat(john, burger, hands, lunch, x)
– John eats a burger with his hands. $\exists y.\exists x$. eat(john, burger, hands, y, x)
– John eats a burger. $\exists z.\exists y.\exists x$. eat(john, burger, z, y, x)
– John eats. $\exists w.\exists z.\exists y.\exists x$. eat(john, w, z, y, x)

Le problème est que nous ne sommes pas assurés que l'arité choisie pour le prédicat eat est suffisante pour faire face à tous les besoins. Et en réalité, le prédicat tel qu'il a été proposé est probablement insuffisant. Il faudrait lui ajouter au moins un argument pour tenir compte du temps. Pis encore, les modifications adverbiales prouvent qu'il est peu vraisemblable que l'on parvienne à déterminer quels sont, exactement, les arguments requis. Chacune des phrases qui ont été utilisées peut en effet être modifiée par des adverbes tels que greedily, slowly, rapidly, piggily, fastidiously, ceremoniously, ravenously, surreptitiously, ... Une façon naturelle de traiter ce problème est de supposer que l'on dispose d'entités abstraites (appelées des événements) et dont les propriétés peuvent être représentées à l'aide de prédicats. En supposant que de telles entités existent (c'est-à-dire en se donnant le droit de quantifier existentiellement sur de telles entités), on simplifie considérablement les représentations. Notre exemple peut ainsi être traité directement. Par exemple, on a que :

– John eats : $\exists e$. eat(e) $\wedge$ agent(e,john)
– John eats a burger : $\exists e$. eat(e) $\wedge$ agent(e,john) $\wedge$ patient(e,burger)

De plus, nous pouvons ajouter à souhait d'autres informations, telles que des indications temporelles, des modifications adverbiales :

– John greedily ate a big Kahuna burger at three o'clock in the room where the suitcase was hidden :

$$\exists e.\text{eat}(e) \wedge \text{greedy}(e) \wedge \text{agent}(e, \text{vincent}) \\ \wedge \text{patient}(e, \text{big-k-burger}) \wedge \text{time}(e, 3, 00) \wedge \text{location}(e, \text{room-suitcase-hidden})$$

Il n'est alors plus nécessaire de se préoccuper de l'ordre des arguments, ou de déterminer à l'avance combien d'arguments seront nécessaires ; on ajoute simplement de nouveaux conjoints au gré des besoins. De plus, on obtient ainsi automatiquement les implications sémantiques entre les phrases. Par exemple, si Vincent mange un burger goulûment, alors Vincent mange :

$$\exists e.\text{eat}(e) \wedge \text{agent}(e, \text{vincent}) \wedge \text{patient}(e, \text{burger}) \wedge \text{greedy}(e) \\ \models \exists e.\text{eat}(e) \wedge \text{agent}(e, \text{vincent})$$

Les inférences nécessaires découlent directement des propriétés sémantiques du quantificateur existentiel et de la conjonction.

Il reste beaucoup de choses à dire à propos des événements et de leur pertinence en sémantique des langues ; le traitement classique des événements dans un cadre montagovien est donné dans (Dowty, 1979). Mais notre intérêt porte sur la méthode de modélisation, pas sur les détails. Comme nous l'avons dit, nous avons présenté les événements surtout pour donner un exemple de ce qu'on appelle la réification. La réification consiste rendre concret un concept abstrait. Nous avons considéré une entité abstraite et l'avons traitée comme un individu ordinaire sur lequel il est possible de quantifier. Procéder ainsi a conduit à une simplification des représentations utilisées et nous permet de gérer les adverbes.

Depuis les travaux de Montague, les sémanticiens ont introduit un grand nombre d'entités abstraites dans les représentations sémantiques. De même qu'il est naturel d'utiliser des événements, on peut montrer qu'il est tout aussi naturel d'utiliser des entités plurielles. Il est par exemple peu vraisemblable qu'une phrase telle que « John and Mary lifted the piano », signifie

que soit John, soit Mary a soulevé le piano. Elle signifie plutôt qu'une sorte d'entité composite incluant à la fois John et Mary a soulevé le piano. La nécessité de travailler avec une grande variété d'entités en sémantique formelle a été défendue par des sémanticiens importants tels que Emmon Bach (Bach, 1989) et Nicholas Asher (Asher, 1993).

Or, si l'on souhaite être en mesure de travailler avec des entités abstraites, nous aurons besoin de pouvoir les manipuler au sein de notre logique. En d'autres termes, nous avons besoin de pouvoir gérer différents types d'entités abstraites, de spécifier comment elles devraient se comporter, *etc.*Comme nous l'avons dit, la logique IL de Montague est en elle-même relativement complexe. Y aurait-il des systèmes logiques auxquels il serait plus aisé d'incorporer les entités abstraites ? Comme nous allons le voir maintenant, c'est l'une des raisons pour lesquelles les sémanticiens ont commencé à s'intéresser à TYn.

1.2.2 TYn en sémantique formelle

Comme nous l'avons déjà mentionné, la logique IL de Montague, présentée dans le désormais célèbre article (Montague, 1973), était une combinaison de théorie des types classique avec des idées provenant de la logique modale. Pendant la décennie qui a suivi son introduction, IL a été largement acceptée et considérée l'outil standard de la théorie des types pour modéliser la sémantique des langues. Mais, avec le temps, un certain nombre d'objections – issues de la logique et de la sémantique des langues – ont été faites et une famille de logiques alternative appelée TYn ou théorie des types à n sortes est devenue la théorie de prédilection pour plusieurs sémanticiens. C'est cette famille de systèmes qui se trouve au cœur de cette thèse. Le membre le plus précoce (l'aîné) de cette famille, à savoir TY2 (c'est-à-dire une théorie des types à deux sortes), a été introduit par Daniel Gallin, l'un des doctorants de Richard Montague (voir (Gallin, 1975)). La définition de la famille TYn est une extension directe de la définition de TY2 donnée par Gallin (cette dernière étant elle-même une extension directe de l'approche de Henkin à la logique d'ordre supérieur). Voici cette définition, qui inclut celle de la syntaxe de TYn ainsi que celle de sa sémantique.

Définition 1 (Types de TYn)**.** *Pour tout $n \geq 0$, les types de TYn sont construits à partir de $n+1$ types de base. L'un de ces types de base est appelé le type des propositions ou des valeurs de vérité. Ce type est particulier et appartient à toutes les variantes de TYn. On le notera t, pour « truth values ». De plus, nous sommes libres de choisir $n \geq 0$ autres types. Si $n = 0$ (c'est-à-dire si l'on ne choisit aucun type) le système obtenu est appelé TY0 ; ce système est le plus simple de tus, puisque son seul type de base est celui des valeurs de vérité t. Par ailleurs si l'on choisit $n \geq 1$, alors on dispose, en plus de t, de n autres types de base notés $\tau_1, \ldots, \tau_n$. Un ensemble arbitraire de n types de base ayant été fixé, on définit les types de TYn sur ces types de base de la façon suivante :*

1. *Tous les types de base sont des types de TYn.*
2. *Si α et β sont des types de TYn, alors le type (α, β) est un type de TYn. Ce type sera noté $\alpha \rightarrow \beta$ dans tout le reste de cette thèse, pour rappeler sa sémantique fonctionnelle, qui sera définie ci-dessous.*
3. *Rien d'autre n'est un type de TYn.*

L'ensemble des types ayant été défini, l'étape suivante consiste à construire le langage logique utilisant ces types. Cette construction est elle aussi directe. On suppose que l'on dispose d'ensembles infinis dénombrables de variables et de constantes de chaque type, ainsi que des connecteurs habituels de la logique d'ordre supérieur de Henkin (tels que $\neg$, $\wedge$, $\forall$, $=$, λ). On définit alors :

Définition 2 (Termes de TYn)**.** *Étant donnés les types de TYn (sur un ensemble fixe mais arbitraire de types de base) on définit les termes de TYn de la façon suivante :*

1. *Toute constante ou variable de type α est un terme de type α. Nous écrirons c_α (resp. x_α) pour désigner les constantes (resp. les variables) de type α.*
2. *Si ϕ et ψ sont des termes de type t, alors $\neg\phi$ et $\phi et \psi$ sont des termes de type t. Les termes de type t (c'est-à-dire les propositions) sont aussi appelés des formules.*
3. *Si ϕ est une formule et x une variable d'un type quelconque, alors $\forall x.\phi$ est une formule.*
4. *Si A et B sont des termes de même type, alors $A = B$ est une formule.*
5. *Si A et un terme de type $\alpha \to \beta$ et B un terme de type α, alors (AB) est un terme de type β.*
6. *Si x est une variable de type α et M est un terme de type β, alors $\lambda x : \alpha.M$ est un terme de type $\alpha \to \beta$.*
7. *Rien d'autre n'est un terme de TYn.*

Avant de poursuivre, faisons quelques remarques sur la syntaxe. D'abord, notre choix quant aux opérateurs primitifs (à savoir $\neg$, $\wedge$ et $\forall$, $=$, λ) est arbitraire. Nous aurions tout aussi bien pu considérer moins d'opérateurs primitifs (il est par exemple bien connu que $\neg$ $\wedge$ et $\forall$ peuvent être définis à partir de $=$ et λ) et d'autres connecteurs logiques familiers (tels que $\exists$, $\vee$, $\to$, *etc.*) peuvent être définis de la façon habituelle. Qu'un opérateur soit primitif ou défini n'est pas, ici, d'une très grande importance. Nous supposerons tout simplement que nous avons tous les opérateurs habituels dont nous avons besoin à notre disposition. De la même façon, nous ne donnerons pas les définitions classiques des notions de variables libres et liées, ni des substitutions – ces définitions sont toutes standard et nous les supposons connues. Dans cette section nous utiliserons la notation $[B/x_\alpha]A$ pour faire référence au résultat du remplacement de toutes les occurrences libres de la variable x de type α dans A par B.

La remarque la plus importante concernant la définition qui vient d'être donnée est que le système TY1 n'est autre que la théorie des types simple de Henkin (Henkin, 1950). Et, comme nous l'avons déjà mentionné, TY2 est l'extension à deux sortes du système de Henkin introduite par Gallin. Plus généralement, un système TYn arbitraire est une version à n sortes de la théorie des types simple de Henkin. De plus, cette correspondance s'étend à la sémantique : comme nous allons le voir maintenant, la sémantique de TYn est simplement une version à n sortes de la sémantique explorée dans le travail de Henkin.

Définition 3 (Cadres sur TYn)**.** *Étant donnée une collection de types de TYn (sur un ensemble fixé mais arbitraire de types de base) on définit un cadre $\mathcal{F}$ pour TYn comme une structure de la forme*

$$\mathcal{F} = \{D_\alpha \mid \alpha \text{ est un type de TY}n\}$$

tel que $D_t = \{0, 1\}$*, pour tout* $n \geq 1$ $D_{\tau_n} \neq \emptyset$*, et pour tout type de la forme* $\alpha \rightarrow \beta$ *on a que*

$$D_{\alpha\rightarrow\beta} \subseteq \{F \mid F : D_\alpha \longrightarrow D_\beta\}$$

Un cadre standard est un cadre où $D_{\alpha\rightarrow\beta} = \{F \mid F : D_\alpha \longrightarrow D_\beta\}$.

Nous pouvons maintenant donner une interprétation dans la théorie des modèles et ressemblant à celle de Tarski de notre logique d'ordre supérieure typée. Étant donné un cadre $\mathcal{F}$, on dit que $\mathcal{I}$ est une fonction d'interprétation sur $\mathcal{F}$ si pour chaque constante c_α dans TYn on a que $\mathcal{I}(c_\alpha) \in D_\alpha$. Un modèle pour TY$n$ est un couple $\langle \mathcal{F}, \mathcal{I}\rangle$ où $\mathcal{F}$ est un cadre sur TYn et $\mathcal{I}$ est une fonction d'interprétation sur $\mathcal{F}$. Un modèle $\langle \mathcal{F}, \mathcal{I}\rangle$ est dit standard si sa fonction d'interprétation $\mathcal{I}$ est standard. Comme d'habitude dans les définitions sémantiques tarskiennes, il nous faut définir en plus de la notion de modèle celle de valuation qui permet de gérer les variables libres. On dit qu'une fonction g est une valuation si pour toute variable x_α dans TYn on a $g(x_\alpha) \in D_\alpha$. Et, si g' est une valuation qui coïncide avec g sur toutes les variables sauf peut-être en x_α (c'est-à-dire si $y_\beta \neq x_\alpha$ implique implique que $g'(y_\beta) = g(y_\beta)$), alors on dit que g' est une x_α-variante de g et l'on note $g[x]g'$.

Définition 4 (Interprétation des termes de TYn dans un modèle de Henkin)**.** *Étant donné un modèle* $\mathcal{M} = \langle \mathcal{F}, \mathcal{I}\rangle$ *et une valuation* g *sur* $\mathcal{M}$ *on définit la valeur* $V_g^{\mathcal{M}}$ *d'un terme de la façon suivante :*

$$\begin{array}{lcl}
V_g^{\mathcal{M}}(x_\alpha) & = & g(x_\alpha) \\
V_g^{\mathcal{M}}(c_\alpha) & = & \mathcal{I}(c_\alpha) \\
V_g^{\mathcal{M}}(\neg\phi) & = & \textit{0 si } V_g^{\mathcal{M}}(\phi) = 1\textit{, et 1 si } V_g^{\mathcal{M}}(\phi) = 0 \\
V_g^{\mathcal{M}}(\phi \wedge \psi) & = & \textit{min}\{V_g^{\mathcal{M}}(\phi), V_g^{\mathcal{M}}(\psi)\} \\
V_g^{\mathcal{M}}(\forall x_\alpha \phi) & = & \textit{1 si pour tout } x_\alpha\textit{-variante } g' \textit{ de } g \textit{ on a} \\
 & & \textit{que } V_{g'}^{\mathcal{M}}(\phi) = 1\textit{. 0 sinon.} \\
V_g^{\mathcal{M}}((A_{\alpha\rightarrow\beta}B_\alpha)) & = & V_g^{\mathcal{M}}(A_{\alpha\rightarrow\beta})(V_g^{\mathcal{M}}(B_\beta)) \\
V_g^{\mathcal{M}}(\lambda x_\alpha(B_\alpha)) & = & \textit{la fonction } F \textit{ de } D_\beta \textit{ dont la valeur en} \\
 & & d \in D_\beta \textit{ est égale à } V_g^{\mathcal{M}}(B_\alpha)
\end{array}$$

Définition 5 (Conséquence sémantique standard dans TYn)**.** *Soit* Σ *un ensemble de formules de TY*n *et* ϕ *une formule de TY*n*. On dit que* ϕ *est la conséquence sémantique standard et on note* $\Sigma \models_s \phi$ *si pour tout modèle* $\mathcal{M}$ *et pour toute valuation* g *sur* $\mathcal{M}$ *telle que* $V_g^{\mathcal{M}}(\sigma) = 1$ *pour tout* $\sigma \in \Sigma$*, on a que* $V_g^{\mathcal{M}}(\phi) = 1$ *également. On dit qu'une telle formule* ϕ *est standard-valide et l'on note* $\models_s \phi$ *si* $\emptyset \models_s \phi$*.*

On peut citer des exemples de formules standard-valides :

$$\forall x_\alpha (Ax = Bx) \rightarrow A = B,$$

qui signifie que TYn est une logique extensionnelle, et

$$(\lambda x_\alpha (A) B) = [B/x_\alpha]A,$$

qui signifie que la β-réduction fonctionne comme habituellement.

Maintenant que nous savons ce qu'est TYn, penchons-nous plus attentivement sur les raisons qui ont motivé son apparition. Pour commencer, examinons celles qui ont conduit Daniel Gallin pour introduire TY2. Gallin s'intéressait à la logique IL de Montague, logique qu'il a axiomatisé avec succès. Cependant, comme le montre un examen du chapitre 1 de (Gallin, 1975), la preuve de complétude de Gallin est longue et laborieuse. Cette complexité tient à la nécessité de gérer l'interaction entre la logique classique d'ordre supérieur et les opérateurs modaux de Montague. C'est pourquoi, dans le chapitre 2 de sa thèse, Gallin introduit TY2 en justifiant ainsi cette innovation :

> ... the cap operator $^\wedge$ acts as a functional abstractor over indices, although the grammar of IL lacks variables over indices since s alone is not a basic type. This omission is reasonable, since IL was intended as a formal logic with intensional features close to those of natural language, and in natural language we do not refer explicitly to contexts of use ... From a formal point of view, however, it is natural to consider investigating IL in an extensional theory of types having two sorts of individuals. We call this logic Two-Sorted Type Theory, and we denote it by TY2 (Gallin, 1975, page 58).

Gallin a donc développé TY2 uniquement pour des raisons logiques : il accepte l'idée que IL est bien motivée du point de vue de la sémantique formelle des langues. Comme nous allons le voir bientôt, cette déclaration sera plus tard remise en cause par Reinhard Muskens.

Peu de temps après, un autre problème technique de IL fut mis en évidence : Friedman et Warren (Friedman et Warren, 1979) ont en effet montré que IL ne vérifiait pas la propriété dite du diamant, ce qui signifie qu'elle n'est pas confluente. En d'autres termes, ils ont montré que certains termes de IL n'ont pas une forme normale unique. Une fois de plus, le problème était dû à l'interaction entre la logique d'ordre supérieur classique et les opérateurs modaux.

Cependant, tout comme Gallin, Friedman et Warren semblent avoir accepté que, en tant qu'outil pour modéliser la sémantique des langues naturelles, IL était un bon choix. En particulier, ils prouvent un résultat positif, à savoir que la confluence est disponible pour un sous-ensemble des termes de IL appelé formules modalement closes, ce qui signifie que toutes ces formules ont une unique forme normale. Les termes modalement clos sont des termes dont la dénotation est indépendante du point de référence auquel ils sont évalués (la définition précise de cet ensemble est donnée à la page 314 de (Friedman et Warren, 1979)). Les auteurs remarquent que Montague n'utilise que des termes modalement clos dans sa modélisation de l'anglais, ce qui signifie que dans la grammaire de Montague, aucun problème de confluence ne peut se produire. La conclusion de leur article est donc positive :

> The main theorems have immediate applications in computer processing of the expressions obtained as translations of English phrases from the PTQ fragment. Their unique normal form can be obtained by contraction of minimal parts, and this process will always terminate. The resulting form is easier to comprehend than direct translation and is an appropriate form for display or for evaluation. (Friedman et Warren, 1979, page 323)

Cependant, peu de temps après ces travaux, les sémanticiens ont commencé à s'intéresser à TYn. Les premiers à avoir exploré cette piste semblent avoir été Jeroen Groenendijk et Martin Stokhof dans leur exploration classique de la pragmatique et de la sémantique des questions (voir (Groenendijk et Stokhof, 1984, 1997)). Mais le sémanticien qui argumenta le plus en faveur de l'importance de TYn d'un point de vue sémantique est sans aucun doute Reinhard Muskens. Comme le travail de Muskens joue un rôle important dans cette thèse, voyons pourquoi il trouve TYn si important. Muskens prend pour la première fois position contre IL et en faveur de TYn dans (Muskens, 1996a). Certains de ses arguments sont logiques ; par exemple, il rappelle le résultat de non confluence dû à Friedman et Warren. Il remarque également qu'un certain nombre de formules qui sont en général des tautologies ne le sont pas dans TYn. Par exemple, nous avons vu plutôt une tautologie reflétant le fait que la β-réduction fonctionne correctement dans TYn. Cette formule n'est pas une tautologie dans IL, car la β-réduction ne fonctionne pas correctement dans la logique de Montague, mais seulement pour certaines classes de formules modalement closes.

Au-delà de ces critiques, Muskens insiste sur le fait que TY2 offre un meilleur cadre pour la sémantique. Le point de vue de Muskens peut être résumé de la façon suivante : pourquoi s'embarrasser de la machinerie modale ? Il est beaucoup plus simple de modéliser directement toutes les entités abstraites dont on a besoin, en ajoutant autant de sortes que nécessaire ainsi que des axiomes pour spécifier le comportement des entités introduites.

Voici un exemple simple. Supposons que nous sommes à la recherche d'une logique pour représenter la sémantique temporelle. Une façon d'y parvenir serait d'utiliser une logique modale intentionnelle dans le style de celle proposée par Montague, c'est-à-dire utilisant les opérateurs F et P. Une autre possibilité consiste simplement à ajouter un nouveau type pour la structure dont nous avons besoin. Supposons donc que nous avons décidé de travailler avec les instants du temps. La première étape consiste à ajouter un nouveau type (appelons-le i, pour instants) et des entités D_i de type i représentant les instants du temps. On ne peut cependant pas s'en tenir là. Nous voulons en effet que les éléments de D_i se comportent comme un continuum temporel. Cela signifie que l'on aimerait au moins que ce continuum soit irréflexif et transitif. Nous introduisons donc une constante $<$ de type $i \to i \to t$, puis nous ajoutons les axiomes suivants :

$$\begin{array}{ll} \text{Irréflexivité :} & \forall x_i \neg x_i < x_i \\ \text{Transitivité :} & \forall x_i \forall y_i \forall z_i (x_i < y_i \wedge y_i < z_i \to x_i < z_i) \end{array}$$

(où nous avons écrit $x_i < y_j$ au lieu de $((< x_i)y_j)$, qui est moins lisible.) Et voilà tout ce que nous avons à faire. Ayant ajouté une sorte pour les instants du temps, nous disposons de toute la puissance d'expressivité de la théorie des types classique pour travailler avec ces instants. Nous n'avons pas besoin d'opérateurs modaux.

En résumé, le message fondamental sous-jacent à la plupart des travaux de Muskens est : ne compliquons pas les définitions sémantiques. Au lieu de cela, ajoutons simplement de nouvelles

sortes à chaque fois que l'on souhaite utiliser des entités abstraites et utilisons des axiomes pour en réguler le comportement.

Muskens a montré que cette stratégie était fructueuse. Par exemple, dans (Muskens, 1995), il propose une approche intéressante pour la sémantique temporelle. Mais il a également montré autre chose : que TYn est utile non seulement parce qu'il offre une façon simple de gérer le genre d'entités abstraites dont on a besoin en sémantique formelle (les événements, les situations, les instants du temps *etc.*) mais aussi et peut-être surtout parce que l'on peut y exprimer les idées fondamentales de la DRT en résolvant ainsi plusieurs problèmes de ce formalisme de représentation des discours. Le travail de Muskens sur la DRT est l'un des thèmes centraux de cette thèse (nous y reviendrons en détails aux chapitres 5 et 6 et le comparerons à un travail plus récent dû à de Groote). C'est afin de préparer le lecteur à cette discussion que nous passons maintenant à une brève présentation de la DRT.

1.2.3 Théorie de la Représentation des Discours (DRT)

La DRT (Discourse Representation Theory, voir (Heim, 1982; Kamp, 1984; Kamp et Reyle, 1993)) est le développement de la sémantique formelle qui est à la fois peut-être le plus important depuis les travaux de Montague, et l'un des plus pertinents pour la sémantique computationnelle. La DRT a permis au programme de Montague d'être étendu du niveau des phrases à celui de discours entiers, et a aussi permis à la sémantique d'effectuer quelques percées dans le domaine de la pragmatique (par exemple, l'algorithme de van der Sandt introduit dans (Van der Sandt, 1992) pour traiter les présuppositions utilise la DRT de façon intensive).

Les travaux de Montague en sémantique souffrent d'une limitation évidente : les méthodes qu'il a développées pour construire des représentations sémantiques ne fonctionnent que pour des phrases et ne s'appliquent pas à des textes constitués de plusieurs phrases. Ceci s'explique par le fait qu'un authentique texte ne peut en aucun cas se réduire à une séquence de phrases arbitraires. Les phrases entretiennent entre elles des relations qui peuvent être de beaucoup de types différents, comme par exemple des relations temporelles ou rhétoriques. C'est pourquoi, s'il n'y a pas de mécanisme pour rendre compte de l'information qui est exprimée au travers de ces relations, le sens du texte ne pourra être saisi correctement.

Les structures les plus simples permettant d'établir ainsi des relations entre phrases sont les pronoms, et notre traitement des discours au chapitre 6 s'y intéressera tout particulièrement. La fonction d'un pronom anaphorique est d'établir un lien avec une expression de référence apparaissant plus tôt dans le texte. Considérons par exemple le texte suivant : « Vincent looks at Mia. She dances. » Nous voyons que le pronom « she » doit être lié au nom « Mia ». Pour utiliser la terminologie linguistique, on dit que l'anaphore « she » doit être liée à son antécédent « Mia ». Il y a deux problèmes principaux à prendre en compte dans le traitement des pronoms. Le premier est le problème de la résolution. Il s'agit de savoir comment, étant donné un pronom, trouver son antécédent. Dans l'exemple cité ci-dessus, la réponse était claire : « she » doit être lié à un antécédent féminin, et l'on ne pourrait donc pas lier « she » à Vincent. Dans le cas général, pourtant, le problème de la résolution des anaphores s'avère extrêmement complexe, à tel point que toute une branche de la recherche en linguistique lui est consacrée. Beaucoup d'algorithmes pour la résolution de pronoms sont désormais disponibles. Une bonne vue d'ensemble de ce domaine peut être trouvée dans (Mitkov, 2002).

Le second problème concernant le traitement des pronoms est celui de leur représentation. Il s'agit de trouver une logique permettant de représenter des textes de façon que la représentation choisie pour les pronoms ait un sens. Une bonne façon d'appréhender la difficulté de cette question est d'essayer d'utiliser la logique du premier ordre pour représenter la sémantique de textes où figurent des pronoms. On peut par exemple se demander ce qu'il se passe si l'on procède à la construction sémantique phrase par phrase, les représentations sémantiques obtenues étant ensuite jointes par des conjonctions. Considérons l'exemple suivant : « A woman snorts. She collapses. » Si nous traduisons tour à tour chaque phrase en utilisant une approche montagovienne, et si nous lions ensuite les représentations sémantiques obtenues, nous aboutissons au résultat suivant :

$$\exists x.(\text{woman } x) \wedge (\text{snort } x) \wedge (\text{collapse } y).$$

Cette représentation n'est pas satisfaisante, parce qu'il n'y a pas de lien entre le pronom (représenté par la variable y) et son antécédent (ici « a woman »). Et il n'y a pas de façon simple de résoudre ce problème. Par exemple, ajouter l'équation $x = y$ à la conjonction ne suffit pas, parce que l'occurrence de x qui apparaît dans cette équation ne serait alors pas liée par le quantificateur existentiel.

Il n'y a d'ailleurs même pas besoin de considérer des discours à plusieurs phrases pour s'apercevoir que les pronoms posent un problème. Considérons par exemple le fameux exemple des « donkey sentences » dont il a été fait mention pour la première fois dans (Geach, 1962) : « If a farmer owns a donkey, he beats it ». Ici, le problème n'est pas qu'il n'est pas possible d'écrire une formule du premier ordre représentant cette phrase. Voici en effet une formule exprimant le sens de cette phrase :

$$\forall x.\forall y.[((\text{farmer } x) \wedge (\text{donkey } y) \wedge (\text{own } xy)) \rightarrow (\text{beat } xy)].$$

Le véritable problème ici est plutôt que l'on ne peut construire cette représentation systématiquement. Si nous essayons de le faire en utilisant les λ-termes habituels, nous obtiendrons quelque chose comme

$$\exists x.[(\text{farmer } x) \wedge \exists y.[(\text{donkey } y) \wedge (\text{own } xy)]] \rightarrow (\text{beat } xy).$$

Malheureusement, cette formule n'exprime pas le sens de la phrase initiale. On constate ainse que, lorsque l'on s'intéresse aux pronoms, il devient évident que la portée des variables et leur liaison dans les langues ont un comportement très différent de celui qui est courant dans les langages formels.

La DRT a été conçue pour être une logique plus proche de la langue dans sa façon de traiter la quantification et la portée des variables. Elle est construite à partir d'un nouveau langage logique, le langage des DRSs (Discourse Representation Structures) communément appelées « boîtes ». La DRT ne se donne pas pour mission d'apporter une solution au problème de la résolution des anaphores qui a été évoqué plus haut (elle permet néanmoins de restreindre quelque peu l'ensemble des antécédents possibles pour un pronom). Autrement dit, dans un système réel basé sur la DRT, la DRT devrait être couplée à un algorithme de résolution des anaphore pronominales, pour ne parler que d'elles. La DRT est cependant une bonne solution au problème de la

représentation. Elle est en effet de nos jours l'un des formalismes de représentation sémantique les plus largement utilisé. Aussi, voyons de plus près à quoi elle ressemble.

Les langages à base de DRSs réutilisent beaucoup des ingrédients présents dans les langages du premier ordre. Comme eux, ils contiennent les symboles $\neg$, $\vee$, $=$ et $\Rightarrow$ (plutôt que $\rightarrow$). Ils contiennent les symboles x, y, z *etc.*, mais dans le contexte des langages à base de DRSs ces symboles sont appelés des référents de discours et non plus des variables. Il y a cependant entre les langages du premier ordre et les langages à base de DRSs une différence cruciale : les langages à base de DRSs ne contiennent ni le quantificateur universel $\forall$, ni le quantificateur existentiel $\exists$. C'est d'ailleurs précisément pour cette raison (ils traitent les concepts de quantification et de variable d'une façon très différentes des langages du premier ordre) que les DRSs sont intéressantes. Donnons maintenant la définition du langage des DRSs, le fameux langage des « boîtes » :

Définition 6 (Syntaxe des DRSs et conditions)**.** *Les DRSs sont construites comme suit (leur définition et celle des conditions sont mutuellement inductives) :*

1. *Si $x_1, \ldots, x_n, n \geq 0$ sont des référents de discours et $\gamma_1, \ldots, \gamma_m, m \geq 0$ sont des conditions, alors*

$x_1, \ldots, x_n$
γ_1 $\vdots$ γ_m

   est une DRS. Par la suite, nous utiliserons pour représenter cette DRS la notation linéarisée suivante : $[x_1, \ldots, x_n | \gamma_1, \ldots, \gamma_m]$.
2. *Si R est une relation d'arité n et $x_1, \ldots, x_n$ sont des référents de discours, alors*

   $$R(x_1, \ldots, x_n)$$

   est une condition.
3. *Si t_1 et t_2 sont des référents de discours ou des constantes, alors $t_1 = t_2$ est une condition.*
4. *Si K_1 et K_2 sont des DRSs, alors $K_1 \Rightarrow K_2$ est une condition.*
5. *Si K_1 et K_2 sont des DRSs, alors $K_1 \vee K_2$ est une condition.*
6. *Si K est une DRS, alors $\neg K$ est une condition.*
7. *Rien n'est une DRS ni une condition, à moins que cela puisse être montré à l'aide des clauses précédentes.*

Pour comprendre comment les DRSs sont utilisées, revenons à notre exemple précédent : « A woman snorts. She collapses. » Supposons qu'avant le début du traitement de ce petit discours, aucune information n'est disponible. Nous partons donc d'une DRS vide, c'est-à-dire

que nous représentons par $[|]$ dans notre notation linéarisée, la seule que nous utiliserons désormais. Puis, l'analyse syntaxique et sémantique de la première phrase est effectuée. Dans l'approche

classique de Kamp et Reyle, un algorithme descendant parcourt l'arbre syntaxique et aboutit à la représentation suivante : $[x|(\text{woman } x), (\text{snort } x)]$. L'algorithme reconnaît que « A woman » est une expression à laquelle on peut vouloir faire référence et ajoute par conséquent l'individu x à la partie supérieure de la DRS, c'est-à-dire celle qui apparaît avant la barre verticale dans notre notation linéarisée. Il ajoute ensuite les deux informations dont nous disposons sur cet individu, à savoir qu'elle est une femme qui sniffe). Voyons maintenant ce qu'il se passe lorsque nous poursuivons avec la seconde phrase, « She collapses ». L'intuition essentielle ici est la dynamicité. On continue d'enrichir la même boîte en y ajoutant l'information contenue dans la phrase suivante. En particulier, nous allons :

1. Ajouter un nouveau référent de discours, par exemple y pour tenir compte du fait que le pronom "she" est une expression référente.
2. Ajouter la condition $(\text{collapse} y)$, puisque c'est l'information dont nous disposons à propos de « she ».
3. Ajouter une équation incomplète $y =?$ pour indiquer qu'un référent de discours doit être résolu.

Nous obtenons ainsi la représentation suivante :

$$[x, y|(\text{woman } x), (\text{snort } x), (\text{collapse } y), y =?]$$

On poursuit en essayant de résoudre les référents de discours qui ont besoin d'un antécédent. Ici il n'y a qu'un référent de discours non résolu, à savoir y et seulement un autre référent qui peut lui servir d'antécédent, à savoir x. On peut donc ajouter l'équation $y = x$ et obtenir ainsi :

$$[x, y|(\text{woman } x), (\text{snort } x), (\text{collapse } y), y = x]$$

La DRT impose des contraintes sur les antécédents possibles pour les pronoms ; nous n'entrerons pas dans les détails ici. Comme nous l'avons dit plus haut, la DRT ne cherche pas à apporter une solution au problème de la résolution d'anaphores. Elle devrait plutôt être comprise comme une logique qui nous permet de représenter toutes les interprétations, cela de façon sous-spécifiée. En d'autres termes : en l'absence d'un module spécialisé dans la résolution d'anaphores, on peut envisager l'étape de résolution dans la DRT comme un algorithme non-déterministe qui génère des représentations possibles.

Nous espérons que notre discussion informelle de la DRT a mis en évidence deux points. Le premier est qu'il s'agit d'un formalisme intuitif, capable de construire des représentations qui résolvent les problèmes de représentation posés par les pronoms d'une façon plutôt satisfaisante. Le second point est que le processus de construction de DRSs qui vient d'être décrit se présente très différemment de l'exemple de construction sémantique montagovienne, mise en œuvre à l'aide du λ-calcul et qui avait été donné en section 1.1. Lorsque nous avons présenté la méthode des fragments, nous avons souligné son fonctionnement résolument compositionnel : nous avions spécifié la contribution de chaque mot à la représentation sémantique finale, puis combiné ces représentations par des β-réductions successives qui remontaient l'arbre syntaxique, des feuilles vers la racine. Dans l'exemple qui vient d'être donné, nous n'avons rien fait de tout cela. Aucune sémantique lexicale n'a été spécifiée, et l'algorithme de construction sémantique utilisé était descendant plutôt qu'ascendant.

En fait, dans les premiers temps de la DRT, beaucoup de chercheurs ont exprimé les mêmes interrogations. Ils se demandaient si la DRT était authentiquement compositionnelle. Elle semblait faire quelque chose de sensé, mais offrait-elle une analyse authentiquement compositionnelle ? Une autre interrogation s'ajoutait à celle-ci, cette fois à propos de la nature représentationnelle de la DRT. Il est manifeste dans l'exemple ci-dessus que la DRT est intimement liée aux représentations qu'elle manipule. Considérons par exemple la toute dernière étape dans l'exemple que nous venons de donner. Nous avons dit que l'équation $y = x$ devait être ajoutée. Cela signifie que nous devons scruter la représentation que nous sommes en train de construire et chercher explicitement dans cette représentation une variable convenable pour construire une équation. Si nous faisons une utilisation aussi intensive des détails de nos représentations, de telles représentations peuvent-elles réellement être éliminables ?

En effet, même la seconde étape du processus ci-dessus semble être représentationnelle. L'algorithme spécifie que l'information de la seconde phrase doit être « incorporé » à la boîte préexistante. On utilise à nouveau ici des éléments spécifiques de la représentation ; en effet, dans (Kamp et Reyle, 1993) l'algorithme de construction sémantique descendant fonctionne en plaçant d'abord l'arbre syntaxique de la $n + 1$-ième phrase dans la DRS produite par la n-ième phrase, puis en extrayant ensuite l'information sémantique contenue dans l'arbre, information qui se retrouve de fait intégrée à la DRS préexistante. Il n'est pas du tout évident qu'une telle approche soit compatible avec le principe d'éliminabilité des représentations de Montague.

En bref, la DRT semblait aller à l'encontre des deux principes méthodologiques centraux de la sémantique formelle : elle n'était pas *a priori* compositionnelle, et elle semblait avoir besoin d'accéder aux représentations. Ainsi, bien que la DRT fût prisée pour son succès empirique, elle était pour le moins controversée.

Il y eut deux types de réponses techniques à ces problèmes. Le premier chronologiquement a consisté à essayer de « combiner la DRT avec les lambdas ». Il s'agissait essentiellement d'étendre le langage de la DRT en y ajoutant la λ-abstraction et d'autres constructions, puis d'utiliser le formalisme résultant selon les principes montagoviens. Plusieurs formalismes fonctionnant sur ce principe ont été proposés, tels que la DRT ascendante de (Asher, 1993) et la λ-DRT de (Bos et al., 1994). L'approche plus récente appelée SDRT et qui étend la DRT pour mieux prendre en compte la structure des discours (voir (Asher et Lascarides, 2003)) appartient également à cette tradition.

La seconde réponse a été apportée un peu plus tard dans (Muskens, 1996c). Elle consiste en un formalisme appelé la CDRT (pour Compositional DRT) « in which the construction process that sends (parsed) sentences to boxes consists of a Montague style compositional translation of trees into the lambda box » (Muskens, 1996b, p.145). L'innovation principale de la CDRT était que, contrairement aux approches mentionnées précédemment, elle n'a pas été obtenue en enrichissant le langage de la DRT par des constructions du λ-calcul. Muskens a plutôt montré que les boîtes pouvaient être *définies* dans TYn. En fait, ce que fait Muskens dans cet article est très similaire à ce que fait un sémanticien lorsqu'il modélise des événements ou un autre genre d'entités abstraites dans TYn. En effet, Muskens introduit des objets abstraits appelés *états*, *entités* et *registres* et montre comment ces nouvelles entités abstraites peuvent être utilisées pour créer des interprétations des boîtes dans la théorie des modèles, ces interprétations se comportant exactement comme des DRSs. Cette idée est intéressante parce que, dans une certaine mesure, elle ne demande la création de rien de nouveau. On travaille tout simplement dans TYn, avec

l'aide de quelques axiomes. L'article est un classique de la sémantique formelle, et est de plus élégant et convainquant.

L'histoire ne s'arrête cependant pas là. En 2006, Philippe de Groote a publié un article proposant un autre traitement compositionnel de la dynamicité (voir (de Groote, 2006)). Alors que Muskens s'était fixé comme but de rendre la DRT compositionnelle, il n'en va pas ainsi pour de Groote. Ce dernier a en effet plutôt cherché à proposer un traitement générique et compositionnel de la dynamicité. Ce traitement peut certes être utilisé pour simuler la DRT, mais il a une portée beaucoup plus générale et cette utilisation n'en n'est qu'un cas particulier. Cette approche ne nécessite aucun axiome supplémentaire, à strictement parler cette approche n'a pas besoin de la théorie des types et pourrait tout aussi bien être présentée dans le _lc non-typé. Aors que les approches précédentes ajoutent des lambdas aux boîtes, l'approche de de Groote pourrait être envisagée comme montrant que même le λ-calcul « pur » contient les boîtes. Les chapitres 5 et 6 sont consacrés à l'exploration de ces deux dernières propositions. Théoriquement, les deux approches (qui fonctionnent très différemment) semblent offrir des solutions élégantes aux problèmes posés par les premières versions de la DRT. Mais qu'en est-il sur le plan computationnel ?

1.2.4 Grammaires Catégorielles Abstraites

Richard Montague ne fut pas seulement le fondateur de la sémantique formelle appliquée, il fut aussi le fondateur de son versant théorique. Bien que « The Proper Treatment of Quantification in Ordinary English » (Montague, 1973) soit probablement son article le plus célèbre dans la communauté des linguistes (cet article est en grande partie consacré au développement d'une sémantique précise pour un fragment d'anglais), c'est son article « Universal Grammar » (Montague, 1970c) qui est probablement le plus connu des logiciens et des philosophes. L'importance de cet article tient à la précision de la modélisation qu'il propose de la sémantique formelle. Montague y présente en effet l'interface syntaxe-sémantique comme un homomorphisme entre une algèbre syntaxique (c'est-à-dire, en essence, des arbres syntaxiques) et un domaine sémantique. Dans ce cadre formel, il est en mesure de donner, en utilisant le langage de l'algèbre universelle, une définition précise de concepts tels que l'éliminabilité des représentations. Le travail de Montague fut aussi influent pour d'autres raisons. Notamment, il n'a pas été influencé par l'approche chomskyenne de la linguistique (il avait en réalité plutôt tendance à être quelque peu indifférent à cette approche) et a utilisé un cadre syntaxique qui, à bien des égards, est beaucoup plus naturel pour les linguistes et les informaticiens, à savoir celui des grammaires catégorielles (Adukiewicz, 1935; Bar-Hillel et al., 1960; Lambek, 1958). Les grammaires catégorielles constituent une approche relativement simple et hautement lexicalisée à l'écriture de grammaires. De plus, l'idée d'appliquer un foncteur syntaxique à un argument syntaxique qui y est utilisé correspond parfaitement bien à ce qu'il se passe dans la sémantique de Montague, où les foncteurs sémantiques sont appliqués à des arguments sémantiques. Les travaux de Montague, joints à la compréhension du fait que l'approche des grammaires catégorielles proposée par Lambek (Lambek, 1958, 1961, 1988) était intéressante tant logiquement que mathématiquement, a contribué à faire des grammaires catégorielles l'une des approches les plus influentes à la syntaxe.

Puis, en 1982, Johan van Benthem (van Benthem, 1986) fit une observation intéressante : si l'on travaille avec le calcul de Lambek non dirigé, alors le lien entre les représentations sé-

mantiques montagoviennes et le calcul de Lambek peut être envisagé comme une instance de l'isomorphisme de Curry-Howard-de Bruijn [2]. Ceci prouvait, une fois de plus, que l'approche de la construction sémantique basée sur le λ-calcul n'était pas arbitraire, mais, au contraire, théoriquement bien motivée. L'intuition de Van Bethem n'a cependant pas vraiment attiré beaucoup l'attention, probablement parce qu'elle utilisait un calcul non dirigé. Ces calculs ne sont en effet pas toujours adaptés aux besoins des linguistes, notamment parce que les langages qu'ils reconnaissent sont clos par permutation (c'est-à-dire que si l'on peut reconnaître une chaîne donnée, alors on peut aussi reconnaître toutes ses permutations), ce qui n'est bien sûr pas souhaitable dans le contexte de la langue.

Plus récemment, des recherches combinant et généralisant ces observations théoriques ont eu lieu. Elles ont commencé avec les travaux de Oehrle (Oehrle, 1994, 1995) qui a décidé de prendre en compte les problèmes posés par les calculs non dirigés en utilisant des λ-termes non seulement pour représenter la sémantique, mais aussi et surtout pour représenter la syntaxe. Puis, plus ou moins simultanément, Reinhard Muskens et Philippe de Groote (les pionniers des traitements compositionnels de la dynamicité) ont exprimé cette idée dans un cadre mathématique général pour modéliser l'interface syntaxe-sémantique. La formulation de Muskens porte le nom de « λ-grammars » (Muskens, 2003), tandis que celle proposée par de Groote s'appelle « Abstract Categorial Grammars » (de Groote, 2003, 2001). Dans cette thèse, nous nous appuierons sur les grammaires catégorielles abstraites de de Groote et parlerons donc des ACGs plutôt que des grammaires λ de Muskens.

Au chapitre 7, nous présenterons les idées sous-jacentes aux ACGs en détail, et nous montrerons les liens qu'elles entretiennent avec le système `Nessie`. Au moins deux raisons rendent les ACGs importantes pour cette thèse. La première est qu'elles offrent un cadre abstrait dans lequel l'expressivité peut être évaluée et caractérisée avec précision. L'outil `Nessie` est central pour cette thèse. Mais quelles en sont, précisément, les capacités ? Nous donnons quelques réponses dans cette thèse (en montrant comment il peut être intégré à un système construisant des représentations temporelles pour le polonais, par exemple), mais la petite taille des exemples considérés les rend quelque peu insatisfaisants. Pour pouvoir mener à bien une évaluation sérieuse, nous ne voyons que deux possibilités. La première consiste à montrer que `Nessie` peut être combiné à une grammaire à large couverture pour une langue donnée. Cette première possibilité nous paraît tout à fait digne d'intérêt, au point que nous pensons qu'elle pourrait faire l'objet d'une thèse à part entière. Elle pose en effet ses propres questions méthodologiques, par exemple sur la démarche à adopter pour savoir si les représentations sémantiques construites sont toujours celles qui sont souhaitées. La seconde possibilité consiste à donner une caractérisation mathématique des capacités de `Nessie`. Les ACGs sont intéressantes parce qu'elles fournissent un cadre permettant de mesurer l'expressivité d'un système de construction sémantique, et c'est précisément cette possibilité qui sera exploitée au chapitre 7.

Une seconde raison, plus générale que la première, nous pousse à nous intéresser aux ACGs. Comme nous l'avons indiqué plus haut, Montague n'a pas seulement été à l'origine de la sémantique formelle appliquée. Il en a aussi fondé le versant plus théorique. Or, cette thèse porte

[2] L'isomorphisme de Curry-Howard-De Bruijn est l'isomorphisme entre les preuves de la logique intuitionniste et les types du λ-calcul simplement typé. En informatique, il est souvent fait référence à cet isomorphisme en disant qu'une preuve est un programme et que la proposition qu'elle prouve est le type du programme. On pourra se référer à (de Groote, 1995) qui regroupe beaucoup d'articles fondamentaux sur ce sujet.

davantage sur la sémantique computationnelle que sur la sémantique formelle. Il n'y a cependant pas de raison pour que la sémantique computationnelle néglige les aspects théoriques. En effet, compte tenu du fait que la sémantique computationnelle est une discipline encore jeune, il est important qu'elle sache tirer parti de toute la connaissance accumulée par des branches plus anciennes de l'informatique : de solides fondations théoriques sont importantes. Les ACGs offrent une vision abstraite du processus de construction sémantique qui utilise des outils fondamentaux de l'informatique théorique (telles que l'isomorphisme de Curry-Howard-De Bruijn), et il nous semble qu'il est extrêmement sage de ne pas hésiter à faire appel à de tels outils.

1.3 Sémantique computationnelle

Les idées les plus importantes constituant l'arrière-plan de cette thèse devraient maintenant être claires. En particulier, on devrait bien comprendre maintenant pourquoi les chercheurs en sémantique formelle considèrent TYn comme un outil important. Mais le but de cette thèse est d'examiner TYn en adoptant une perspective computationnelle. En d'autres termes, cette thèse n'est pas une thèse en sémantique formelle, mais en sémantique *computationnelle*. C'est pourquoi, avant d'aller plus loin, il nous paraît plus que souhaitable de dire quelques mots de sémantique computationnelle.

L'expression « sémantique computationnelle_fg peut être lue soit dans un sens relativement large, soit avec une acception plus restreinte. Prise au sens large, cette expression fait référence à tout travail en linguistique computationnelle qui se rapporte au sens. Mais cette interprétation est probablement trop large pour être vraiment utile. Elle ne permet pas de dégager un champ de recherche cohérent. Ainsi l'expression « sémantique computationnelle » est souvent utilisée avec un sens plus restreint. Elle fait alors référence à des tentatives pour modéliser computationnellement la langue en utilisant l'approche logique introduite par Montague et développé par une importante communauté depuis lors. Dans ce sens plus restrictif, la sémantique computationnelle pourrait être décrite comme étant la rencontre de la sémantique formelle avec la linguistique computationnelle. C'est un domaine de recherche encore petit, et relativement récent. Il a fallu un certain temps pour que cette rencontre entre sémantique formelle et linguistique computationnelle ait lieu. En effet, jusqu'aux années 90, il semble qu'il n'y a eu que peu d'entreprises de la part des chercheurs en linguistique computationnelle (et en intelligence artificielle) pour établir le contact avec les idées de la sémantique formelle. Quant aux tentatives de chercheurs en sémantique formelle pour établir le contact avec des idées en provenance de la linguistique formelle ou de l'intelligence artificielle, elles semblent plus marginales encore. Ceci dit, il y a des exemples intéressants de travaux plus anciens en linguistique computationnelle qui correspondent à notre définition la plus restrictive de la sémantique computationnelle.

Hoobs, dans un article intitulé « Making computational Sense of Montague's Intensional Logic » (voir (Hobbs et Rosenschein, 1978)), suggère que la sémantique de Montague peut être envisagée avec profit en termes de sémantique procédurale plutôt que dans le cadre de la théorie des modèles. Pour donner plus d'aplomb à cette idée, l'article contient une traduction de la logique intentionnelle de Montague dans le langage de programmation fonctionnelle Lisp. Dans (Schubert et Pelletier, 1982), les auteurs définissent une traduction simple d'un fragment de l'anglais (spécifié en utilisant une grammaire hors contexte) dans ce qu'ils appellent la logique

conventionnelle, l'objectif étant de pouvoir utiliser ces traductions dans un système d'inférence. Dans (Landsbergen, 1982), d'autre part, le contexte est celui de la traduction automatique. La logique de Montague est utilisée comme langage intermédiaire : la langue source est traduite vers ce langage, et les expressions logiques qui résultent de cette traduction sont utilisées pour construire des phrases dans la langue cible. Dans (Main et Benson, 1983), les idées de la sémantique de Montague sont utilisées dans un système dont la tâche est de répondre à des questions. Un autre travail datant de la même période est (Gunji, 1981), une thèse de doctorat qui n'est pas seulement une contribution pionnière à la sémantique computationnelle, mais aussi à la pragmatique computationnelle. Gunji, réalisant qu'une base de données peut-être considérée comme un modèle, définit un système dans lequel les phrases d'entrée sont traduites dans une logique ressemblant à celle de Montague. Bien que la base de donnée soit essentiellement de la mémoire en lecture seule, en tout cas pour autant que ses procédures sémantiques sont concernées (c'est-à-dire que les formules logiques sont évaluées dans la base de données sans l'altérer, à la manière des requêtes et bases de données standard) il y a également des procédures pragmatiques qui peuvent modifier le modèle (ou comme le dit Gunji, induire des changements de contextes) qui peut très bien affecter l'évaluation sémantique de phrases futures.

Mais c'est pendant les années 90 que la sémantique computationnelle au sens étroit a commencé à émerger plus distinctement. Les raisons pour cette émergence que d'aucuns pourraient qualifier de tardives ne sont pas difficiles à comprendre. La sémantique, qu'elle soit computationnelle ou formelle, présuppose l'analyse syntaxique, et ce n'est qu'à partir des années 90 que des grammaires à large couverture, des analyseurs syntaxiques robustes, ainsi que d'autres outils et ressources ont véritablement commencé à être disponibles. Pendant cette décennie, beaucoup de travaux ont porté sur les formalismes de sous-spécification (c'est-à-dire des façons d'encoder les représentations qui permettent de retarder la résolution des ambiguïtés) ; voir par exemple (Copestake et al., 1995; Althaus et al., 2003; Koller et al., 2003; Bos, 1996).

La mise en place du *International Workshop in Computational Semantics* en 1995 et la publication du livre de cours (Blackburn et Bos, 2005b) ont contribué à donner au domaine une direction, mais peut-être plus encore, une identité. L'année 2005 est aussi celle qui a vu l'apparition d'un système mettant en œuvre l'analyse sémantique profonde pour l'anglais, basé sur un formalisme de représentation sémantique standard (la DRT) et utilisant une grammaire robuste et à large couverture. Il s'agit du système BOXER de Johan Bos, décrit dans (Bos, 2005; Bos et Markert, 2006). Ce sont ces développements qui constituent le contexte de cette thèse.

Le système `Curt` (pour Clever Use of Reasonning Tools), introduit dans (Blackburn et Bos, 2005b) pour illustrer les techniques qui y sont présentées est utilisé de façon intensive aux chapitres 3, 4 et 6. Il nous paraît donc souhaitable d'en donner ici une brève présentation, qui n'a pas pour but d'en donner une description exhaustive, mais plutôt d'aider le lecteur à comprendre les travaux présentés par la suite.

L'un des objectifs poursuivis dans (Blackburn et Bos, 2005b) était de montrer que des outils logiques (aussi appelés outils d'inférence) tels que des prouveurs de théorèmes et des constructeurs de modèles peuvent être utilisés pour construire des représentations sémantiques plus pertinentes. On peut par exemple les utiliser pour vérifier qu'une représentation est cohérente avec une théorie donnée, cette théorie pouvant être constituée d'axiomes représentant divers types de connaissances, ou de représentations de phrases préalablement analysées. Une autre utilisation possible des outils d'inférence est le test d'informativité, qui consiste à déterminer si une repré-

sentation sémantique apporte réellement une information nouvelle, ou si elle ne fait qu'affirmer une conséquence sémantique de la théorie considérée. Le système `Curt` est construit progressivement, chaque nouvelle version étant obtenue en ajoutant à la précédente de nouvelles utilisations des outils d'inférence. La version la plus simple du système, appelée `Baby Curt`, est la seule à n'utiliser aucun outil d'inférence, se contentant de construire des représentations sémantiques de façon compositionnelle, suivant les principes montagoviens. Les autres versions de `Curt` comportent quant à elles deux parties, l'une concernant la construction de représentations sémantiques et l'autre concernant l'inférence. La construction sémantique est mise en œuvre à l'aide d'une DCG [3] qui est utilisée à la fois pour réaliser l'analyse syntaxique des phrases et pour guider le processus de construction sémantique. Le langage utilisé pour exprimer la représentation des phrases est celui de la logique du premier ordre, tandis que le langage utilisé pour assembler les différentes représentations est le λ-calcul non typé.

Au début du chapitre 2, nous expliquerons en quoi la mise en œuvre de la construction sémantique telle qu'elle vient d'être résumée nous paraît limitée. Nous proposerons une approche alternative qui s'appuie directement sur les concepts d'arbre et de lexique introduits par Montague et permet de séparer l'analyse syntaxique de la construction sémantique de façon plus nette que ce qui est fait dans `Curt`. Cette séparation ayant été établie, il devient naturel d'envisager un outil dédié spécifiquement à la construction sémantique, et c'est cet outil, appelé `Nessie`, que nous présenterons au chapitre 2. Il ne s'agit cependant pas d'abandonner complètement l'architecture de `Curt`. Nous la réutiliserons, au contraire, dans la suite de cette thèse, d'abord, au chapitre 3, pour valider les résultats de `Nessie`, puis pour entreprendre de nouvelles expérimentations en construction sémantique, aux chapitres 4 et 6. Plus précisément, nous conserverons la DCG de `Curt` ainsi que sa composante chargée de l'inférence (en les modifiant légèrement au gré de nos besoins), et nous externaliserons la construction sémantique pour la confier à `Nessie`. En combinant ces ingrédients, nous serons par exemple capables, au chapitre 6, de construire *automatiquement* la représentation de la phrase « A farmer who owns a donkey beats it » utilisée comme exemple final dans (de Groote, 2006). La représentation fournie par `Nessie` est la suivante :

```
lam(E,lam(Phi,
  and(all(X,not(some(X__1,and(donkey(X__1),and(
  own(X,X__1),and(farmer(X),
  not(and(beat(X,selit(cons(X,cons(X__1,E)))),
  top))))))))),app(Phi,E))
)).
```

qui n'est rien d'autre que la représentation `Prolog` du terme indiqué dans (de Groote, 2006).

[3] Les DCGs, pour Definite Clause Grammars, sont un mécanisme fourni par `Prolog` pour faciliter l'écriture de grammaires. Ce mécanisme a pendant longtemps été considéré comme l'un des plus efficaces pour la linguistique computationnelle, ce qui explique sans doute le succès rencontré par `Prolog` dans ce domaine. Pour de plus amples informations sur `Prolog` et les DCGs, on pourra se reporter à (Blackburn et al., 2007).

1.4 Plan de la thèse

Beaucoup de chercheurs en sémantique formelle ont affirmé que TYn était le langage le plus adapté à la construction sémantique pour des raisons tant logiques que sémantiques. L'objectif de cette thèse est de déterminer si TYn est aussi intéressant d'un point de vue computationnel. L'outil informatique qui a été développé et exploré pendant cette thèse est le programme `Nessie`. Il permet à l'utilisateur de spécifier un système de type de TYn, et met ensuite en œuvre la construction sémantique au sein de ce système de type. Le plan de la thèse est le suivant :

- **Chapitre 2.** Nous présentons `Nessie` et son implantation en `OCaml`. Comme nous le verrons, la programmation fonctionnelle constitue un paradigme idéal pour l'implantation d'un système tel que TYn.
- **Chapitre 3.** Nous validons `Nessie` en l'utilisant pour re-construire les représentations sémantiques construites par `Curt`, l'outil développé à des fins pédagogiques par Blackburn et Bos.
- **Chapitre 4.** Nous proposons une théorie du temps simple et basée sur les événements, conçue pour modéliser la sémantique temporelle du polonais. Nous utilisons `Nessie` pour créer un système simple dans l'esprit de `Curt` pour le polonais et montrons comment la construction de modèles temporels peut être intégrée à cette architecture.
- **Chapitre 5.** Nous présentons la DRT compositionnelle de Muskens et le traitement compositionnel de la dynamicité de de Groote. Nous étudions les différences entre ces deux théories et utilisons la réécriture d'ordre supérieur pour proposer un cadre dans lequel les propriétés de l'encodage de Muskens peuvent être étudiées finement.
- **Chapitre 6.** Nous mettons en œuvre les outils présentés au chapitre précédent. Pour la CDRT, nous montrons les conséquences sur le plan pratique, des remarques faites au chapitre précédent. Le traitement compositionnel de la dynamicité est quant à lui utilisé comme base pour mener à bien une réflexion sur les différentes façons dont pourrait être mise en œuvre la résolution d'anaphores dans TYn.
- **Chapitre 7.** Nous nous y interrogeons sur l'expressivité de `Nessie`, nous demandant en particulier avec quels types d'analyseurs syntaxiques il pourrait être utilisé. Nous apportant à cette question une réponse théorique, en montrant que `Nessie` est équivalent à une grammaire catégorielle abstraite du second ordre. Les conséquences de ce résultat sont également étudiées.
- **Chapitre 8.** Nous résumons la thèse en mettant l'accent sur ses contributions, puis nous proposons quelques pistes pour des recherches ultérieures.

Chapitre 2

Nessie, un outil pour automatiser le calcul du sens

À la section 1.3, nous avons donné une brève présentation de `Curt`, le système introduit dans (Blackburn et Bos, 2005b) pour illustrer l'utilisation de l'inférence en construction sémantique. Dans cette présentation, nous avons souligné plusieurs caractéristiques du processus de construction sémantique tel qu'il est mis en œuvre dans `Curt`. Nous aimerions ici reprendre ces caractéristiques et montrer en quoi elles peuvent être vues comme des limitations dont on peut souhaiter s'affranchir. Plus précisément, nous ne cherchons pas réellement à critiquer `Curt`. Ce système a été développé à des fins pédagogiques et il est à notre avis tout à fait adapté au but pour lequel il a été conçu. Notre propos est plutôt de montrer que la valeur pédagogique certaine de `Curt` ne suffit pas à en faire un outil bien adapté à la recherche en sémantique computationnelle qui nous intéresse ici. Examinons donc les caractéristiques de `Curt` précédemment citées.

D'abord, nous avions fait remarquer que la construction de représentations sémantiques complexes à partir de celles des éléments lexicaux était faite dans le cadre du λ-calcul non typé. Or, le fait d'utiliser le λ-calcul *non typé* peut être une source de problème. En effet, en théorie en tout cas, il est possible que le processus de construction sémantique aboutisse à des termes qui n'ont pas de forme normale par β-réduction. L'exemple classique pour illustrer ce problème est le terme $(\lambda x.xx)(\lambda y.yy)$ qui se réduit vers lui-même. On pourrait certes faire remarquer qu'il est peu vraisemblable que de tels termes apparaissent en pratique, mais cela ne change rien au fait que, sans les types qui restreignent l'ensemble des termes acceptables, on ne peut garantir que ce problème ne se produira pas. En outre, toujours sur le plan théorique, nous avons signalé que les représentations sémantiques construites pour les phrases utilisent la logique du premier ordre. Il est cependant important de préciser que les seuls termes manipulables par `Curt` (en tout cas pour ce qui est de l'inférence) sont les constantes et les variables. Il n'est donc pas possible d'utiliser des fonctions. Comme on le verra au chapitre 4, ceci oblige à modéliser ces dernières à l'aide de relations, ce qui implique d'axiomatiser pour chacune des relations modélisant une fonction des propriétés telles que l'unicité de l'image.

Ensuite, sur un plan plus pratique, nous avons expliqué à la section 1.3 que la DCG de `Curt` est utilisée à la fois pour réaliser l'analyse syntaxique, et pour guider la construction sémantique. Concrètement, cela signifie que les règles de DCG utilisées pour reconnaître les différents syntagmes font appel à des règles `Prolog` capables de construire la représentation du syntagme en combinant celles de ses constituants. Les « recettes sémantiques » ne peuvent donc être écrites qu'en `Prolog`, ce qui limite à notre avis considérablement les possibilités d'expérimenter d'autres formalismes sémantiques que ceux présents dans `Curt`. Le fait de rendre obligatoire

l'utilisation de `Prolog` pour la spécification des recettes sémantiques nous paraît d'autant plus regrettable que `Prolog` est, à notre avis, un langage assez peu adapté à cette tâche, notamment parce que, en l'absence d'un véritable système de types, peu d'erreurs de programmation peuvent être détectées et reportées.

Les premiers éléments qui peuvent être utilisés pour résoudre les problèmes qui viennent d'être évoqués ont été donnés au chapitre précédent. D'une part, nous avons introduit TYn, une logique d'ordre supérieur basée sur le λ-calcul simplement typé. Si ce formalisme est utilisé à la place de la logique du premier ordre et du λ-calcul non typé qui sont utilisés dans `Curt`, alors les problèmes théoriques mentionnés précédemment (absence de forme normale et impossibilité d'utiliser des fonctions) disparaissent. D'autre part, nous avons vu en section 1.1 que la proposition de Montague pour aborder la construction sémantique est de recourir à des lexiques pour associer à chaque mot sa représentation sémantique, et à des arbres pour guider le processus de construction sémantique. Bien que ces structures aient été introduites dans le cadre de la sémantique formelle, nous les croyons bien adaptées aussi aux besoins de la sémantique computationnelle, et nous allons donc les utiliser ici comme entrées pour la construction sémantique. Nous serons cependant amenés à les raffiner quelque peu, de sorte à en augmenter l'efficacité computationnelle. Nous verrons ainsi qu'il est souhaitable de tenir compte, dans les lexiques, des notions de familles fermées et ouvertes, bien connues des linguistes. Pour ce qui est des arbres, nous prendrons un peu de distance avec la proposition originale de Montague : alors que ceux qu'il utilisait étaient des arbres syntaxiques, les nôtres seront abstraits, ce qui signifie qu'ils ne devront pas nécessairement refléter la structure syntaxique du texte (il pourra cependant nous arriver d'y faire référence en parlant d'arbres syntaxiques ; il s'agira alors d'un abus de langage, certes un peu imprécis mais suffisamment commode pour que nous le conservions). Nous parviendrons ainsi à une notion d'arbre qui ne dépend pas d'un formalisme syntaxique spécifique et dont l'expressivité est équivalente à celle d'une grammaire catégorielle abstraite du second ordre, ce que nous prouverons au chapitre 7.

Nous allons donc nous intéresser dans ce chapitre au processus de construction sémantique proprement dit, prenant comme entrées un lexique et un arbre. Nous mettons en œuvre ce processus de construction sémantique au sein d'un outil appelé `Nessie` et développé en `OCaml`, un langage de programmation fonctionnel, c'est-à-dire basé sur le λ-calcul. Lors de la présentation de `Nessie`, nous nous attacherons à montrer les bénéfices que l'on peut tirer de l'utilisation d'un langage fonctionnel tel que `Caml` pour des applications en sémantique computationnelle. Nous pouvons d'ores et déjà en donner une intuition, en constatant que, dans un langage basé sur le λ-calcul, les problèmes inhérents à la manipulation de λ-termes sont centraux et donc particulièrement bien étudiés par les développeurs et utilisateurs d'un tel langage. Tout y est fait pour que la conception de programmes manipulant des termes soit rapide, efficace et sûre. Dans la section 2.1, nous donnons une brève présentation de `Caml`, ce qui nous permettra, dans la section 2.2 consacrée à la présentation de `Nessie`, de montrer de façon plus concrète comment se traduit l'intuition qui vient d'être donnée.

2.1 Introduction à OCaml

Comme nous allons l'expliquer dans la suite de cette section, le langage OCaml est un langage fonctionnel avec inférence de types. En plus d'un compilateur permettant de traduire les programmes OCaml en programmes exécutables, le langage peut être utilisé de façon interactive grâce à une boucle d'interaction (top-level) qui permet d'évaluer des expressions de façon interactive. Voici un exemple de session OCaml interactive simple :

```
$ ocaml
        Objective Caml version 3.10.1

# 1+1;;
- : int = 2
# let pi = 3.141592;;
val pi : float = 3.141592
# pi *. pi;;
- : float = 9.86960029446400178
# let e = 2.718281 in e *. e;;
- : float = 7.38905609643502093
# e;;
Unbound value e
```

Chaque commande se termine par `;;` et est appelée une *phrase*. La première phrase dénote une expression à laquelle n'est associé aucun nom. La réponse du système à cette première phrase peut se lire de la façon suivante :

- `-` signifie que l'expression qui vient d'être entrée n'a été liée à aucun nom, autrement dit qu'elle est anonyme ;
- `int` est le type de l'expression et `2` est sa valeur.

La seconde phrase définit elle aussi une expression, mais cette fois, un nom lui est associé. Ceci permet de réutiliser la valeur de l'expression dans des expressions futures, comme en témoigne la troisième phrase. Dans la quatrième phrase, le nom `e` est lié à la valeur `2.718281`, mais seulement pendant le calcul de l'expression qui suit le mot réservé `in`. Dès que l'évaluation de cette expression se termine, l'association entre le nom `e` et sa valeur est oubliée, ce qui explique la réponse `Unbound value` donnée par le système lorsqu'on lui demande la valeur de `e` à la phrase suivante. La quatrième phrase illustre donc la construction `let ... in` qui permet d'associer un nom à une expression pendant le calcul d'une autre expression. On peut enchaîner autant de `let ... in ...` qu'on le souhaite :

```
# let x = 3 in let y = x+1 in 2 * x + y;;
- : int = 10
```

En plus de ne lier une expression à un nom que pendant le calcul d'une autre expression, la construction `let e = expr1 in expr2` garantit que l'expression `expr1` sera toujours évaluée *avant* `expr2`, ce qui peut s'avérer très utile lorsque l'ordre dans lequel des calculs sont effectués a de l'importance.

2.1.1 Inférence de type et langage fonctionnel

Dans les exemples que nous avons montrés précédemment, nous avons pu constater que le système `OCaml` est capable d'afficher le type des expressions, et cela bien que les phrases que nous avons saisies ne comportent aucune information de typage explicite. Ceci est rendu possible par le fait que le langage `OCaml` fournit un mécanisme d'*inférence de types*, ce qui signifie que `OCaml` est capable d'associer de lui-même un type à toute expression syntaxiquement bien formée. Par exemple, si nous déclarons :

```
# let square = fun x -> x * x;;
```

la réponse du système est la suivante :

```
val square : int -> int = <fun>
```

Ici, le système a inféré que l'expression associée au nom `square` est une fonction (ce qu'indique la valeur `<fun>`) prenant en argument un entier et renvoyant un entier, comme en témoigne son type `int -> int`. Pour utiliser cette fonction dans un calcul, il suffit d'écrire quelque chose comme

```
# square 4;;
- : int = 16
```

L'exemple précédent nous apprend aussi que la syntaxe `fun var -> expr` représente la fonction qui à `var` associe `expr`, où `expr` est une expression qui dépend généralement de `var`. Il est important de souligner que `fun var -> expr` est une expression à part entière, et qu'à ce titre elle peut aussi être évaluée sans qu'il y ait besoin de lui associer nécessairement un nom. Par exemple :

```
# fun x -> x * x * x;;
- : int -> int = <fun>
```

Ici, nous avons évalué une expression qui est une fonction, sans pour autant lui donner un nom. Cette expression peut aussi apparaître comme sous-expression d'expressions plus complexes, par exemple :

```
# (fun x -> x * x * x) 3;;
- : int = 27
```

Dans cet exemple, la fonction qui à `x` associe son cube est appliquée à l'argument `3` et c'est de cette application que provient la valeur `27` de type `int` renvoyée par le système.

Ces exemples illustrent l'une des raisons qui vaut à `OCaml` son appellation de langage *fonctionnel*. En effet, ils montrent qu'en `OCaml` les fonctions sont des expressions comme les autres (on les appelle aussi « citoyens de première classe »), qui n'ont pas besoin d'être associées à un nom pour être utilisées dans des expressions plus complexes. Ceci constitue une différence importante avec les langages impératifs comme `C`, dans lesquels une fonction ne peut être utilisée sans avoir été déclarée au préalable, c'est-à-dire sans qu'un nom lui ait d'abord été associé.

L'autre raison pour laquelle `OCaml` est dit être un langage fonctionnel est qu'il est possible de construire facilement une fonction à partir d'une autre, comme le montre l'exemple suivant :

```
# let sum = fun x -> fun y -> x + y;;
val sum : int -> int -> int = <fun>
# let succ = sum 1;;
val succ : int -> int = <fun>
# succ 5;;
- : int = 6
```

La première phrase de cet exemple déclare une fonction `sum` à deux arguments. Ces deux arguments sont des entiers, dont la fonction renvoie la somme, également entière. La deuxième phrase construit la fonction `succ` en appliquant la fonction `sum` à son premier argument seulement. Il résulte de cette application partielle de la fonction `sum` une nouvelle fonction à un argument de type entier et qui renvoie cet entier auquel a été ajouté `1`, c'est-à-dire son successeur.

Comme on peut le constater, il est donc possible en `OCaml` de construire des fonctions en appliquant des fonctions préexistantes à une partie de leurs arguments, ce qui constitue une autre différence importante avec les langages impératifs, où ce genre de construction n'est pas possible de façon systématique.

Dans le contexte du développement d'un programme tel que `Nessie`, l'inférence de types et l'aspect fonctionnel de `OCaml` se révèlent toutes deux des outils des plus précieux. En effet, les facilités offertes par `OCaml` pour écrire des fonctions incitent le programmeur à en utiliser le plus possible, ce qui aboutit non seulement à une plus grande factorisation du code, mais aussi à des programmes dans lesquels les fonctions n'ont pas d'effets de bord, ce qui les rend plus faciles à utiliser et réduit les risques d'erreurs. L'inférence de types, quant à elle, dispense souvent d'expliciter les types des objets manipulés, ce qui augmente la lisibilité des programmes. On peut par ailleurs remarquer que la vérification du typage des programmes `Caml` se fait à la compilation, et non à l'exécution. Dans la pratique, on peut constater que ce typage statique permet d'éviter beaucoup d'erreurs de programmation. Il rend en effet la phase de compilation plus exigeante, augmentant les chances pour un programme qui se compile correctement de ne pas contenir d'erreurs.

Introduisons pour terminer cette sous-section une notation permettant de définir des fonctions plus aisément que ne le permet la construction `fun` utilisée jusqu'ici :

```
let f = fun x1 -> fun x2 -> ... -> fun xn -> expr;;
```

peut aussi s'écrire

```
let f x1 x2 ... xn = expr;;
```

et dans la pratique, c'est cette notation qui sera utilisée pour définir des fonctions auxquelles un nom est associé, la notation `fun` n'étant utilisée en général que pour les fonctions anonymes apparaissant dans des expressions plus complexes.

2.1.2 Types de données

Comme la plupart des langages de programmation, `OCaml` fournit à la fois des types prédéfinis (entiers, flottants, booléens, chaînes de caractères...) et des opérations permettant de construire des types plus complexes à partir des types prédéfinis ou de types définis précédemment. Dans

cette sous-section, nous décrivons quelques opérations disponibles pour construire de nouveaux types.

Types produit Un tuple est un type permettant de regrouper un certain nombre de données, ces données pouvant être de types différents. Ces types peuvent être utilisés lorsqu'il est nécessaire de passer simultanément plusieurs objets à une fonction, lorsqu'une fonction a besoin de renvoyer plusieurs valeurs, ou tout simplement parce que cela correspond à ce que l'on souhaite modéliser. Les points du plan constituent un bon exemple de cette dernière situation. En `OCaml`, le type des points à coordonnées entières peut être déclaré de la façon suivante :

```
# type point = int * int;;
type point = int * int
```

Le point dont l'abscisse est 1 et l'ordonnée 2 peut alors être noté comme suit :

```
# (1,2);;
- : int * int = (1, 2)
```

Remarquons que nous nous trouvons ici dans une situation où le type inféré par `OCaml`, bien que correct, n'est pas forcément celui auquel nous nous attendions. Nous pouvons aider le système à inférer le « bon » type en utilisant la notation `(expr : type)` comme suit :

```
# ((1,2) : point);;
- : point = (1, 2)
```

Et si nous souhaitions manipuler des points pondérés, dans le contexte d'une application en physique par exemple, nous pourrions définir un type représentant ces points de la façon suivante :

```
# type weightedPoint = point * float;;
type weightedPoint = point * float
```

Ce type permet de manipuler comme une seule entité un objet qui contient à la fois un point et son poids. Il illustre aussi le fait qu'un type prodit peut rassembler des objets de types différents.

Types somme Il arrive souvent que l'on ait à manipuler des valeurs appartenant à un ensemble fini et connu à l'avance. Un exemple de cette situation est donné par les mois de l'année. C'est pour ce genre d'exemples que les types somme sont utiles. Voici par exemple comment pourrait être défini un type représentant les différents mois d'une année :

```
# type month = Jan | Feb | Mar | Apr | May | Jun
  | Jul | Aug | Sep | Oct | Nov | Dec;;
type month = Jan | Feb | Mar | Apr | May | Jun
  | Jul | Aug | Sep | Oct | Nov | Dec
```

D'après cette définition de type, il y a exactement 12 valeurs de type `month`, à savoir `Jan`, `Feb`, ..., `Dec`. Ces valeurs sont souvent appelées des *constructeurs* car elles permettent de

construire des valeurs du type auquel elles appartiennent. Les constructeurs peuvent appartenir à au plus un type, ce qui facilite l'inférence de type en présence de constructeurs. Notons enfin que les noms des constructeurs commencent nécessairement par une lettre majuscule.

La technique utilisée pour traiter des valeurs de types tels que `month` est appelée *filtrage de motif* ou *pattern matching* en anglais. Pour illustrer cette technique, voici comment la fonction `days` renvoyant le nombre de jours dans un mois peut être définie :

```
let days m = match m with
  | Jan -> 31
  | Feb -> 28
  ...
  | Dec -> 31
```

ou, de façon un peu plus concise :

```
let days = function
  | Jan -> 31
  | Feb -> 28
  ...
  | Dec -> 31
```

Dans la seconde version, l'abstraction n'apparaît plus explicitement : on a en effet remplacé

```
let days m = match m with
```

par

```
let days = function
```

On évite ainsi de donner un nom à l'argument que l'on souhaite examiner à l'aide du filtrage de motif. Ceci s'avère souvent pertinent, dans la mesure où il arrive fréquemment qu'on n'ait pas besoin de faire référence à la valeur que l'on filtre.

Les types somme permettent aussi d'énumérer au sein d'un même type des valeurs de types différents. Par exemple, et bien que cela puisse sembler quelque peu artificiel, voici la définition d'un type `number` qui peut contenir soit un entier, soit un flottant.

```
type number =
  | Float of float
  | Int of int
```

Ici encore, le traitement des valeurs de type `number` se fait grâce au filtrage de motif. Voici comment écrire une fonction `sum` pour un tel type :

```
# let sum = function
  | (Int x, Int y) -> Int (x+y)
  | (Float x, Float y) -> Float (x +. y)
  | (Int x, Float y) -> Float ((float_of_int x) +. y)
  | (Float x, Int y) -> Float (x +. (float_of_int y))
val sum : number * number -> number = <fun>
```

Remarquons qu'ici, nous avons fait le choix de passer à `sum` ses arguments sous la forme d'une paire de nombres, comme en témoigne le type `number * number` de l'unique argument de `sum`. Nous aurions tout aussi bien pu définir `sum` comme une fonction à deux arguments de type `number`. Il nous aurait suffit pour cela de remplacer la première ligne de la définition précédente par :

```
let sum n1 n2 = match (n1,n2) with
```

ce qui aurait conduit le système `OCaml` à afficher le type suivant :

```
val sum : number -> number -> number = <fun>
```

Ce changement n'est cependant pas très intéressant car le couple de nombres est tout de même construit. La seule chose qui change par rapport à la première version, c'est que la construction du couple se fait dans la fonction et n'est donc plus à la charge de l'appelant. Il est possible d'éviter complètement de recourir à un couple en ne procédant pas au filtrage de motif sur les deux nombres en même temps, mais le code que l'on obtient est un peu moins lisible que celui que nous avons présenté.

Pour conclure ce paragraphe sur les types unions, il nous paraît important de mentionner qu'il est possible de réunir au sein d'un même type des constructeurs sans arguments comme ceux de `month` et des constructeurs avec arguments comme ceux de `number`. Voici par exemple un type `line` qui pourrait être utilisé par une fonction de lecture depuis un fichier pour renvoyer son résultat :

```
type line =
  | Nothing
  | Something of string
```

Une fonction de lecture depuis un fichier texte pourrait par exemple prendre en argument le nom du fichier à lire et renvoyer `Something` avec en argument la prochaine ligne non lue du fichier, et `Nothing` si la fin du fichier est atteinte.

L'idée que nous venons de présenter ici est généralisée par le type polymorphe `option` que nous introduirons un peu plus loin. Dans la pratique, c'est ce dernier type qui serait utilisé dans une situation comme celle que nous venons d'évoquer, plutôt qu'un type *ad hoc*.

Enregistrements (structures) Les structures ou enregistrements fournis par `OCaml` sont similaires à ceux fournis par d'autres langages de programmation. Voici par exemple une définition alternative du type `point` vu précédemment utilisant une structure à la place d'un produit cartésien :

```
# type point = {
    x : int;
    y : int
  };;
type point = { x : int; y : int; }
```

Un point `p` peut alors être défini à partir de ses coordonnées

```
# let p = { x = 1; y = 2 };;
val p : point = {x = 1; y = 2}
```

ou à partir d'un point existant :

```
# let p' = { p with y = 0 };;
val p' : point = {x = 1; y = 0}
```

Enfin, on accède aux champs des structures grâce à la notation `structure.champs` commune à beaucoup de langages de programmation.

Types récursifs et types polymorphes Comme leur nom l'indique, les types récursifs sont utilisés pour décrire des données dont la structure est récursive, comme des listes ou des arbres. Voici par exemple comment définir des listes d'entiers :

```
# type ilist = INil | ICons of int * ilist
type ilist = INil | ICons of int * ilist
```

comme on peut le constater, cette définition est similaire à celles des types unions. Et en fait, le type `ilist` que nous venons de définir *est* un type union, mais où le type que l'on est en train de définir apparaît aussi comme argument de l'un de ses constructeurs. Par conséquent, les valeurs de types similaires à celui que nous venons de déclarer peuvent elles aussi être traitées à l'aide de fonctions procédant par filtrage de motif, la seule différence avec les fonctions vues précédemment étant que, pour manipuler des données dont le type est récursif, on peut avoir besoin de fonctions récursives. Voici par exemple une fonction calculant la longueur de listes de type `ilist` :

```
# let rec ilength = function
  | INil -> 0
  | ICons (_,xs) -> 1 + (ilength xs);;
val ilength : ilist -> int = <fun>
```

Mis à part le caractère _ utilisé dans les motifs lorsqu'on ne souhaite pas donner de nom explicite à un argument extrait par filtrage, cette définition ne devrait poser aucun problème de lecture. Nous pouvons donc ainsi développer toute une bibliothèque de fonctions sur les listes d'entiers. Cependant, si l'on souhaite disposer d'une bibliothèque équivalente sur les listes de caractères, le seul moyen de l'obtenir sera de copier la première bibliothèque et de l'adapter, l'on devra faire ainsi pour chaque type pour lequel on voudra construire des listes d'éléments.

Le langage `OCaml` offre cependant une solution plus élégante à ce problème, à savoir les types polymorphes. Il s'agit de types particulièrement adaptés à la création de structures de données génériques puisqu'ils font intervenir dans leur définition des variables de types qui peuvent être par la suite instanciées par n'importe quel type. Voici la définition des listes fournie par la bibliothèque standard de `OCaml` :

```
# type 'a list = Nil | Cons of 'a * 'a list;;
type 'a list = Nil | Cons of 'a * 'a list
```

La variable `'a` est une variable de type appelée à être instanciée par un type concret ultérieurement. La définition des listes que nous venons de donner se lit alors de la façon suivante : « étant donné un type `'a`, une liste d'éléments de type `'a` est soit une liste vide (constructeur `Nil`), soit une liste non vide (constructeur `Cons`) constituée d'un élément de type `'a` (premier argument de `Cons`, appelé *tête* de la liste) et d'une autre liste d'éléments de type `'a` (deuxième argument de `Cons`, appelé *queue* de la liste) ». Un type `ilist2` équivalent au type `ilist` définit précédemment peut alors être obtenu en remplaçant `'a` par `int` dans le type que nous venons de définir :

```
# type ilist2 = int list;;
type ilist2 = int list
```

Bien que ce genre de définition soit possible, on n'y recourt pas en général, car on en a en général pas besoin. En outre, pour faciliter l'utilisation des listes, `OCaml` fournit les notations abrégées suivantes :

- `Nil` s'écrit `[]` ;
- `Cons (x,l)` s'écrit `x::l` ;
- `Cons (x,Cons (y, Nil))` s'écrit `[x;y]`.

Comme autre exemple de type polymorphe, citons le type `'a option` qui généralise le type `line` introduit précédemment. Ce type permet, comme son nom l'indique, de représenter une valeur optionnelle. Voici sa définition :

```
# type 'a option = None | Some of 'a;;
type 'a option = None | Some of 'a
```

Remarquons que ce type est polymorphe mais non récursif.

Comme le montre cette présentation des mécanismes offerts par `OCaml` pour définir des types, ce langage est particulièrement bien adapté à la modélisation de définitions mathématiques récursives et de structures de données. Comme nous le verrons par la suite, il est extrêmement aisé et naturel en `Caml` de définir des types pour représenter des λ-termes, des arbres ou des lexiques, types dont nous ferons une utilisation abondante par la suite. Que l'on compare cette facilité de modélisation aux possibilités offertes en `Prolog`, et l'on comprendra aisément pourquoi nous pensons que `OCaml` est un langage bien plus adapté que `Prolog` au développement d'outils tels que `Nessie`.

2.1.3 Modules

Lors du développement de programmes de taille importante, on ressent souvent le besoin de regrouper dans un même endroit des définitions de types de données et de fonctions ayant entre elles un lien logique. C'est précisément l'une des raisons d'être des modules de `OCaml` que nous allons présenter ici. Pour donner sans plus tarder un premier exemple, nous pouvons citer les listes polymorphes que nous avons introduites précédemment. Les définitions d'un certain nombre de fonctions permettant de travailler avec des listes sont regroupées au sein du module `List` de la bibliothèque standard de `Caml`. Comme tous les autres modules en `Caml`, le module `List` est composé de deux parties. Une première partie, appelée indifféremment *interface*, *signature* ou *type* du module, déclare l'ensemble des types, valeurs et fonctions fournis par ce

module au reste du programme. La seconde, appelée en général *implémentation* du module, définit les objets déclarés dans l'interface et peut aussi définir d'autres objets, ceux-ci n'étant pas visibles de l'extérieur du module [1]. Dans le reste de cette sous-section, nous allons présenter d'abord les différentes façons d'écrire un module, puis nous montrerons comment accéder aux éléments définis dans un module depuis le reste du programme.

Écriture d'un module

Il existe deux manières de définir un module en Caml. L'une consiste à écrire son interface et son implémentation dans deux fichiers distincts, l'autre permet de définir l'interface et l'implémentation dans un même fichier.

Interface et implémentation dans des fichiers distincts Étant donné un module nommé M, il s'agit ici de définir son interface dans un fichier m.mli et son implémentation dans un fichier m.ml. C'est de cette façon qu'est défini le module List que nous avons mentionné précédemment. Voici un extrait du fichier list.mli tel qu'il apparaît dans la bibliothèque standard de Caml :

```
val length : 'a list -> int
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
...
```

Et voici la partie correspondante de list.ml :

```
let rec length_aux len = function
    [] -> len
    | a::l -> length_aux (len + 1) l
let length l = length_aux 0 l
let hd = function
    [] -> failwith "hd"
    | a::l -> a
let tl = function
    [] -> failwith "tl"
  | a::l -> l
```

Ces extraits de la bibliothèque standard de Caml appellent plusieurs remarques. En premier lieu, on peut noter que la fonction length_aux définie dans list.ml ne l'est pas dans list.mli, donnant ainsi un exemple de valeur (ici une fonction) définie dans l'implémentation d'un module mais non visible du reste du programme [2]. La seconde remarque concerne les appels à la fonction failwith apparaissant dans les fonctions hd et tl. Sans entrer dans les détails, disons que cette fonction reporte une erreur due au fait que les fonctions qui l'appellent ont été appelées sur une liste vide, liste qui n'a ni queue ni tête. Enfin, remarquons que le type

[1] ...

[2] Cette définition de length à l'aide d'une fonction auxiliaire est plus complexe que celle que nous avions donnée, mais plus efficace à l'exécution car récursive terminale, c'est-à-dire utilisant un espace constant dans la pile, quelle que soit la taille de la liste donnée en entrée.

`'a list` dont nous avions donné la définition à la section précédente n'est pas défini par le module `List`. Ceci est lié au statut particulier des listes en `Caml`, ce statut nécessitant que le type des listes soit connu du système avant même le chargement du module correspondant. Il s'agit là d'une situation tout à fait exceptionnelle qui ne se reproduit pas pour les autres modules, tous devant déclarer les types qu'ils souhaitent manipuler. Notons cependant qu'il existe plusieurs façons de déclarer un type dans l'interface. Une première possibilité consiste à faire coïncider la déclaration de type figurant dans l'interface avec celle apparaissant dans l'implémentation. La définition du type est alors visible depuis l'extérieur du programme, et les objets de ce type peuvent être manipulées sans qu'il soit nécessaire pour cela d'utiliser les fonctions fournies par le module. Une deuxième possibilité consiste à ne donner dans l'interface que le nom du type. Sa définition est alors privée, et seules les fonctions du module y ont accès. Cette technique peut paraître plus restrictive, mais elle est souvent préférée à la première parce qu'elle permet de changer la définition d'un type sans avoir pour autant à modifier d'autres parties du programme que le module où ce type est défini. On profite ainsi pleinement des possibilités d'abstraction offertes par les modules. Enfin, signalons une troisième façon pour une interface de déclarer un type. Le type peut être déclaré à l'aide du mot réservé `private`. Avec cette déclaration, généralement utilisée pour les types unions ou les structures, les valeurs de ces types sont accessibles « en lecture » par le reste du programme, mais il n'est pas possible de les construire sans recourir aux fonctions fournies par le module. Les types privés peuvent être compris comme un compromis entre les deux méthodes vues précédemment, dans la mesure où ils permettent une « demi » exportation du type, intermédiaire entre l'exportation de sa définition et l'exportation de son nom seulement. Comme nous le verrons à la section suivante, le type utilisé pour représenter les termes dans `Nessie` est un type privé, ce qui a permis d'utiliser le filtrage de motif sur les termes dans tout le code de `Nessie`, tout en garantissant les propriétés désirées pour les termes considérés, puisque seules les fonctions du module `Term` ont le droit d'en construire.

Interface et implémentation dans le même fichier La deuxième méthode disponible pour écrire un module `Caml` permet d'écrire son interface et son implémentation dans un même fichier. Pour l'illustrer, nous allons développer un petit module permettant de construire des ensembles. Conceptuellement, un ensemble est une structure de données très similaire à une liste, la seule différence étant qu'une liste véhicule une notion d'ordre, tandis que cette notion n'existe pas avec les ensembles. Un ensemble permet donc de regrouper des objets d'un même type sans les ordonner, ce qui correspond bien à la définition mathématique d'un ensemble. Pour pouvoir écrire un module sur les ensembles, nous devons d'abord réfléchir aux fonctions que nous voulons proposer pour manipuler les ensembles (ce qui nous permettra d'écrire l'interface du module), puis nous devons définir les fonctions de l'interface, ce qui se fera, comme nous l'avons vu, dans l'implémentation. Pour notre exemple, nous allons nous contenter d'un module définissant un type polymorphe `'a t` pour les ensembles d'éléments de type `'a`, d'une valeur `empty` de type `'a t` représentant l'ensemble vide, d'une fonction `add` de type `'a -> 'a t -> 'a t` permettant d'ajouter un élément à un ensemble et d'une fonction `mem` de type `'a -> 'a t -> bool` testant l'appartenance d'un élément à un ensemble. Un dernier problème reste à résoudre avant de pouvoir écrire l'interface du module : comment convient-il d'exporter le type représentant les ensembles ? Nous avons vu précédemment trois

façons de rendre un type visible à l'extérieur du module où il est défini. Laquelle de ces trois méthodes est la plus adaptée ici ? Dans la mesure où un ensemble peut être représenté de plusieurs façons en machine, il paraît raisonnable de cacher la représentation concrète, pour que l'on ne soit pas tenté, lors de l'utilisation du module, de faire des hypothèses sur la représentation concrète des ensembles, rendant ainsi plus difficiles des évolutions ultérieures de cette représentation. Nous choisissons donc de n'exporter que le nom du type. Voici comment peut alors être écrite l'interface (la signature) du module que nous souhaitons écrire :

```
module type SET =
sig
  type 'a t
  val empty : 'a t
  val add : 'a -> 'a t -> 'a t
  val mem : 'a -> 'a t -> bool
end
```

Quant à notre implémentation, elle sera des plus simples puisque donnée à des fins pédagogiques uniquement. Nous choisissons de représenter les ensembles par des listes. Bien sûr, l'ensemble vide est alors représenté par la liste vide, l'ajout d'un élément à un ensemble consiste à ajouter un élément en tête d'une liste [3], et le test d'appartenance peut être effectué en utilisant la fonction `mem` fournie par le module `List` de la bibliothèque standard de `Caml`. Nous obtenons donc l'implémentation suivante :

```
module Set : SET =
struct
  type 'a t = 'a list
  let empty = []
  let add x s = x::s
  let mem = List.mem
end
```

La première ligne de cette définition d'implémentation assure le lien avec l'interface. Elle demande au compilateur de vérifier, une fois l'implémentation construite, que tous les objets qui y sont définis ont des types compatibles avec ceux de même nom déclarés par l'interface `SET`. Il est aussi vérifié que tous les objets déclarés par l'interface ont bien été définis par l'implémentation. Remarquons que, lorsque l'interface et l'implémentation sont définies dans des fichiers séparés, comme c'était le cas au paragraphe précédent, le lien entre les deux composants du module est effectué grâce aux noms des fichiers, ces noms devant différer seulement par l'extension `.mli` pour l'interface et `.ml` pour l'implémentation.

Pour conclure cette partie sur l'écriture des modules, signalons que les deux mécanismes que nous venons de présenter peuvent être combinés et qu'un module peut en contenir d'autres. Ainsi, un fichier `.ml` peut définir des modules en utilisant la notation que nous venons de voir et ces modules peuvent ou non être déclarés dans le fichier `.mli` et peuvent éventuellement contenir d'autres modules voire des interfaces.

[3] Pour simplifier, on ne cherche pas à garantir que chaque élément est présent au plus une fois dans les ensembles construits.

Accès aux éléments définis par les modules

Notre implémentation des ensembles a donné un premier exemple d'appel de fonction appartenant à un module. Nous avons en effet appelé la fonction `mem` du module `List`, et cela en utilisant la notation pointée `List.mem`. De la même façon, il est possible de connaître le type de la fonction `add` du module `Set` que nous venons de définir :

```
# Set.add;;
- : 'a -> 'a Set.t -> 'a Set.t = <fun>
```

Une autre possibilité est d'utiliser la directive `open` suivie d'un nom de module. Cette directive permet de rendre tous les objets du module accessibles sans qu'il soit nécessaire de les faire précéder du nom du module. Par exemple :

```
# open Set;;
# add;;
- : 'a -> 'a Set.t -> 'a Set.t = <fun>
```

Bien qu'il puisse paraître séduisant de pouvoir accéder aux éléments d'un module sans avoir à les qualifier par le nom du module, abuser de cette possibilité peut conduire à des difficultés, car si un module contient le nom d'un symbole déjà défini dans l'espace courant, seule la nouvelle définition associée au nom (celle exportée par le module) est accessible, l'ancienne ne l'étant plus. D'ailleurs, il est courant pour un module définissant une structure de donnée d'utiliser `t` comme nom de type pour cette structure, comme nous l'avons fait pour les ensembles. Dans ce cas, c'est le nom du module plutôt que celui du type qui indique quelle structure on manipule, et on n'a pas vraiment intérêt à utiliser `open`. Dans toute la suite de ce chapitre, nous nous conformerons à cette convention consistant à utiliser `t` comme nom de type pour les structures de données. Nous utiliserons alors toujours la notation pointée incluant le nom de module, ce qui permettra de savoir sans ambiguïté de quel type il est question.

Nous pouvons ici prolonger les remarques faites à la fin de la section consacrée aux types. La souplesse du système de modules de `Caml` est bien plus grande que celle offerte par d'autres langages, et notamment par `Prolog`. Par exemple, les trois méthodes disponibles pour exporter des types ne sont en général pas toutes supportées par d'autres langages, ce qui donne à `Caml` un avantage non négligeable. Nous reviendrons et amplifierons ces remarques à la fin de la section suivante consacrée aux foncteurs, ces derniers étant une autre caractéristiques des plus intéressantes de `Caml`.

2.1.4 Foncteurs ou modules paramétrés

Supposons que le module `Set` définit précédemment est accessible. Nous pouvons alors avoir avec l'interpréteur de `Caml` le dialogue suivant :

```
# let f (x : int) = x;;
# let s = Set.add f Set.empty;;
val s : (int -> int) Set.t = <abstr>
# Set.mem f s;;
- : bool = true
# let g (x : int) = x;;
```

```
val g : int -> int = <fun>
# Set.mem g s;;
Exception: Invalid_argument "equal: functional value".
```

Cette session commence par définir la fonction `f` qui est la fonction identité sur les entiers [4]. Puis, on crée un ensemble `s` de fonctions des entiers vers les entiers ne contenant que `f`. On teste ensuite l'appartenance de `f` à `s` et, conformément à notre intuition, ce test renvoie `true`. Enfin, on déclare la fonction `g` de la même façon qu'on avait déclaré `f` et l'on teste l'appartenance de `g` à `s`. Or, alors qu'on pourrait s'attendre à ce que ce second test renvoie à nouveau `true`, il donne lieu à une erreur entraînant l'affichage du message

```
Exception: Invalid_argument "equal: functional value".
```

Au premier abord, ce résultat pourrait paraître surprenant, voire incompréhensible. Pourtant, à y regarder de plus près, on peut se demander quel est le sens d'un test d'égalité entre fonctions ? Dans le premier cas, `f` est comparée avec `f`, et là il s'agit, quelle que soit la façon dont `Caml` représente les fonctions, de comparer un objet avec lui-même. Cette comparaison ne peut qu'être vraie, quel que soit le sens qu'on donne à « être égales » pour des fonctions. Lorsqu'on teste l'appartenance de `g` à `s` en revanche, la question est plus délicate, car `f` et `g` ont des représentations distinctes en mémoire. De ce point de vue, ce sont donc des fonctions différentes. Mais, comme elles calculent la même chose et le font de la même façon, on pourrait aussi les considérer comme égales.

On le voit, le véritable problème semble bien être le fait que l'on puisse construire des ensembles d'éléments appartenant à des types pour lesquels la notion d'égalité est mal définie. Pour que l'on puisse construire des ensembles de façon satisfaisante, il nous faudrait soit restreindre le type des éléments autorisés de sorte que l'égalité ait toujours un sens précis, soit faire en sorte que l'utilisateur désireux de construire des ensembles pour des valeurs d'un type donné doive préciser quelle égalité il souhaite utiliser pour comparer les éléments. Une autre manière d'expliquer cela consiste à dire que le polymorphisme est ici trop générique, dans le sens où il permet de construire des ensembles à partir de n'importe quel type, alors que ce que l'on souhaite, c'est construire des ensembles à partir de types pour lesquels on dispose d'une notion d'égalité satisfaisante.

Les foncteurs, ou modules paramétrés, représentent l'un des moyens fournis par `OCaml` pour résoudre ce problème. Un module paramétré `M` est un module prenant un autre module `P` en paramètre et construisant ses propres types (resp. valeurs) à partir des types (reps. valeurs) définis par `P`. Pour bien comprendre ce qu'est un foncteur, il peut être utile de l'envisager comme étant une fonction sur les modules. En effet, de la même façon qu'une fonction prend des valeurs en entrée et retourne des valeurs en sortie, un foncteur prend des modules en entrée et retourne un module en sortie. L'appellation *type d'un module* que nous avions mentionnée comme synonyme de *interface* prend ici tout son sens, puisque pour créer un foncteur il est nécessaire de lui indiquer à quelle spécification se conforment ses paramètres, ce qui est analogue à donner le type des arguments d'une fonction.

Dans le cas des ensembles qui nous intéresse ici, il s'agit donc de définir dans un premier temps un type de module `EQ` déclarant un type `t` et une fonction `eq : t -> t -> bool`

[4] Sans la contrainte `x :int`, la fonction `f` serait polymorphe de type `'a -> 'a`.

testant l'égalité entre éléments de type `t`. Un module définissant des ensembles pourra alors être défini comme un foncteur prenant en paramètre un module de type `EQ` (c'est-à-dire compatible avec l'interface `EQ`) et construisant une implémentation conforme à l'interface `SET` que nous avons donnée précédemment.

Voici donc l'interface `EQ` :

```
module type EQ =
sig
  type t
  val eq : t -> t -> bool
end
```

Et voici l'implémentation des ensembles sous la forme d'un foncteur, construite à partir de celle vue précédemment :

```
module Set (M : EQ) : SET =
struct
  type t = M.t list
  let empty = []
  let add x s = x::s
  let rec mem x0 = function
    | [] -> false
    | x::xs -> (M.eq x0 x) || (mem x0 xs)
end
```

Deux changements peuvent être constatés par rapport à la version précédente. Le premier concerne la définition du type des ensembles : le type polymorphe a été remplacé par une liste d'éléments du type `M.t`. Le deuxième changement concerne la fonction `mem` que nous avons dû écrire, parce que celle fournie par la bibliothèque standard sur les listes ne permet pas de spécifier quelle égalité on souhaite utiliser pour comparer deux éléments.

La bibliothèque standard de `Caml` fournit une implémentation des ensembles à base de foncteurs, implémentation très proche de celle que nous venons de présenter. En effet, la seule différence notable entre les définitions que nous venons de voir et celles de la bibliothèque standard de `Caml` est qu'à la place de notre signature `EQ` la bibliothèque standard utilise une signature `OrderedType` qui demande la définition d'un ordre total sur les éléments du type considéré. Le fait de disposer d'un ordre total permet de construire des ensembles en utilisant une structure plus efficace que les listes et où les éléments apparaissent triés, ce qui permet de décider l'appartenance d'un élément à un ensemble en un temps logarithmique en le cardinal de l'ensemble, contre un temps linéaire pour des ensembles construits à l'aide de listes. En outre, le foncteur que nous avions appelé `Set` s'appelle `Make` dans la bibliothèque standard de `Caml` et est défini dans le module `Set`, tout comme la signature `OrderedType` de son argument et la signature `S` du module qu'il permet de créer à partir de cet argument.

Dans toute la suite de ce chapitre, c'est l'implémentation des ensembles fournie par la bibliothèque standard de `Caml` que nous allons utiliser. Nous allons en particulier souvent utiliser des ensembles de chaînes de caractères. Ces ensembles sont construits à partir d'un module `OrderedString` se conformant à la signature `OrderedType`, manipulant des chaînes de caractères et utilisant pour les comparer l'ordre lexicographique :

```
module StringSet : Set.S = Set.Make(OrderedString)
```

Donc, `StringSet.t` désignera dans toute la suite de ce chapitre le type des ensembles de chaînes de caractères.

Donnons pour terminer un autre exemple de structure de données dont l'implantation dans la bibliothèque standard de `Caml` est faite grâce à des foncteurs : les dictionnaires ou listes associatives. Il s'agit d'une structure permettant d'associer des *valeurs* à des *clés*. Un dictionnaire peut ainsi être utilisé pour associer à des mots leur représentation par un λ-terme. Dans ce cas, on dit que les mots sont les clés du dictionnaire, tandis que les λ-termes sont les valeurs associées. Pour qu'une telle structure puisse être utilisée, il faut non seulement pouvoir y ajouter des couples (clé, valeur), mais aussi pouvoir, étant donné une clé, retrouver la valeur qui lui est associée. Ceci nécessite d'être en mesure de comparer une clé donnée avec toutes celles présentes dans le dictionnaire, ce qui pose exactement les mêmes problèmes que ceux soulevés par le test d'appartenance d'un élément à un ensemble. Donc, pour que des valeurs puissent être utilisées comme clés d'un dictionnaire, il faut pouvoir les comparer et, si l'on souhaite de plus être en mesure de stocker les dictionnaires de façon efficace et retrouver la valeur associée à une clé efficacement, il est nécessaire de disposer d'un ordre total sur les données que l'on souhaite utiliser comme clés. Il n'est donc pas étonnant que, tout comme les ensembles, les dictionnaires soient construits par la bibliothèque standard de `Caml` à l'aide de foncteurs. Plus précisément, le module `Map` fourni par la bibliothèque standard de `Caml` exporte, tout comme le module `Set`, une signature `OrderedType` (qui est identique à celle exportée par le module `Set`), une signature `S` spécifiant un type pour les dictionnaires et des fonctions permettant de les manipuler, et un foncteur `Make` prenant en argument un module de type `Map.OrderedType` et retournant un module de type `Map.S`. Enfin, comme les dictionnaires ne requièrent aucune opération sur les valeurs associées aux clés, il n'est pas nécessaire d'en restreindre le type. Par conséquent, les dictionnaires peuvent être représentés par un type polymorphe à un paramètre, ce paramètre correspondant au type des valeurs associées aux clés.

Dans la suite de ce chapitre, nous utiliserons souvent des dictionnaires ayant pour clés des chaînes de caractères. Ces dictionnaires sont construits à partir du module `OrderedString` introduit précédemment et du foncteur `Map.Make` exactement de la même façon que les ensembles de chaînes de caractères :

```
module StringMap : Map.S = Map.Make(OrderedString)
```

Grâce à cette déclaration, nous disposons d'un type polymorphe `'a StringMap.t` représentant des dictionnaires avec comme clés des chaînes de caractères et comme valeurs associées des objets de type `'a`. Si nous voulions définir un lexique comme un dictionnaire associant à un mot un terme dont le type est `Term.t`, il nous suffirait d'écrire :

```
type lexicon = Term.t StringMap.t
```

En conclusion, nous pouvons, comme nous l'avons annoncé précédemment, amplifier les remarques qui ont été faites au sujet des modules. Nous avions en effet signalé que la flexibilité du système de modules de `Caml` était une spécificité du langage, et que les modules disponibles dans la plupart des autres langages n'étaient pas aussi flexibles que ceux fournis par `Caml`. Or, cette différence signalée pour les modules devient plus grande encore lorsqu'il est question de

foncteurs, puisque cette notion est totalement absente de la plupart des autres langages de programmation. Or, comme nous ne cesserons de le montrer à la section suivante, les foncteurs ont été utilisés en permanence dans l'implantation de `Nessie`, et celle-ci n'aurait pu être réalisée avec autant de facilité sans un tel mécanisme. On le voit, le choix du langage `OCaml` est donc des plus fondés pour le développement d'un programme tel que `Nessie`, développement que nous allons présenter maintenant.

2.2 Description de `Nessie`

Cette section présente `Nessie`, l'outil que nous avons développé pour faciliter la construction compositionnelle de représentations sémantiques dans TYn. Comme nous l'avions indiqué au début de ce chapitre, les représentations sémantiques sont construites à partir d'un lexique et d'un arbre syntaxique. Le lexique contient les représentations sémantiques des mots qui sont elles aussi des termes de TYn, tandis que l'arbre syntaxique spécifie comment les représentations fournies par le lexique doivent être combinées pour obtenir la représentation de syntagmes plus complexes. Comme nous le verrons lorsque nous présenterons les arbres syntaxiques, chacun de leurs nœuds peut contenir une règle spécifiant comment les représentations sémantiques des sous-arbres doivent être combinées pour obtenir celle de l'arbre. Le langage utilisé pour écrire ces règles de combinaison de représentations sémantiques n'est autre que TYn lui-même. Ainsi, comme nous le verrons par la suite, une règle de combinaison de sous-arbres n'est rien d'autre qu'un terme typé.

Dans la mesure où les termes de TYn apparaissent à la fois dans les lexiques où ils spécifient la sémantique des mots et dans les arbres syntaxiques où ils spécifient les règles de combinaison de sous-arbres, nous allons commencer par discuter en section 2.2.1 de leur représentation et de la façon dont les opérations les concernant ont été implantées. Dans les sections 2.2.2 et 2.2.3, nous présenterons successivement les lexiques et les arbres syntaxiques. Nous donnons ensuite l'algorithme utilisé pour construire une représentation sémantique à partir d'un arbre et d'un lexique (section 2.2.4) et nous terminons en décrivant brièvement dans la section 2.2.5 le fonctionnement de `Nessie` d'un point de vue utilisateur.

2.2.1 Termes

Nous commençons par expliquer comment nous représentons les types de TYn puis nous discutons de leur représentation interne et du calcul de leur type, avant d'évoquer leur syntaxe concrète.

Types

Nous avons vu à la section 1.2.2 que l'ensemble $\mathcal{T}$ des types acceptables dans une version donnée de TYn est défini par la donnée de n types de base (auxqules s'ajoute le type t). Cet ensemble est le plus petit ensemble contenant ces types et tel que $\forall \tau_1, \tau_2 \in \mathcal{T}, \tau_1 \rightarrow \tau_2 \in \mathcal{T}$.

Un tel système de type peut être décrit en `OCaml` de la façon suivante :

```
type typeSystem = StringSet.t
```

```
type t = private
  | Atomic of string
  | Arrow of t * t
val makeAtomicType : typeSystem -> string -> t
val makeArrowType : typeSystem -> t -> t
```

La première ligne indique qu'un système de type est un ensemble de chaînes de caractères, tandis que le type `t` définit un sur-ensemble des types acceptables pour un système de type donné. En effet, les types atomiques sont des chaînes de caractères mais rien ne garantit qu'ils appartiennent au système de type considéré. C'est pourquoi, nous déclarons ce type `private`, ce qui contraint la construction de types à l'extérieur du module à utiliser les fonctions `makeAtomicType` et `makeArrowType` qui peuvent vérifier que leurs arguments sont des types valides dans un système de type donné. Comme nous l'avions vu à la section précédente, le mot réservé `private` permet de garantir que toutes les valeurs de type `t` seront correctes tout en autorisant le filtrage de motif sur les types à l'extérieur du module.

Bien que cette première implémentation soit fidèle à la définition de TYn qui a été donnée à la section 1.2.2, l'utilisation des premières versions de `Nessie` a permis de constater rapidement qu'une telle définition s'avérait limitée. En effet, dès que l'on manipule des types complexes, on voudrait pouvoir les écrire sous une forme abrégée, par exemple `pred` au lieu de `e -> t`, `bpred` au lieu de `e -> e -> t`. Pour rendre cela possible, nous avons quelque peu étendu la définition précédente de `typeSystem` pour obtenir celle qui suit :

```
type alias = {
  definition : t;
  unfolded : t
}

type typeSystem = private {
  basicTypes : StringSet.t;
  aliases : alias StringMap.t
}

val ty0 : typeSystem
val addBasicType : string -> typeSystem -> typeSystem
val addAlias : string -> t -> typeSystem -> typeSystem
```

Chaque alias contient l'expression de type utilisée pour le définir, ainsi qu'une version « dépliée » où n'apparaissent plus que des types atomiques. Cette seconde version est calculée par la fonction `addAlias` dont les arguments sont un nom d'alias, le type non étendu auquel il correspond et le système de type auquel il convient d'ajouter cet alias. Ainsi, si l'on ajoute le type `e` puis l'alias `pred = e -> t` au système `ty0` [5], puis l'alias `bpred = e -> pred` au système de type ainsi obtenu, l'alias construit pour `bpred` aura pour définition `e -> pred` et pour valeur dépliée `e -> e -> t`.

Bien sûr, en présence d'alias il ne suffit plus de comparer deux types syntaxiquement pour décider leur égalité. Pour le moment, nous la décidons en comparant leurs formes dépliées, ce qui

[5] Le système `ty0` ne contient que le type de base `t` et aucun alias.

n'est pas nécessairement une bonne stratégie. Une autre approche pourrait être de comparer les versions non dépliées et de ne comparer les formes étendues que si cette première comparaison signale que les deux types sont différents.

Enfin, signalons que l'ajout d'alias n'augmente en rien le pouvoir d'expressivité des types. Par conséquent, notre implantation de TYn reste pour le moment tout à fait conforme à la définition que nous en avons donnée à la section 1.2.2.

Termes et typage

Nous avons présenté les lexiques comme des structures permettant d'associer à chaque mot sa représentation sémantique. Cependant, comme nous l'avons vu au chapitre précédent, il est possible que la représentation d'un mot utilisée lors de la construction sémantique ne soit pas exactement celle donnée par le lexique, mais une expression qui lui est α-équivalente, c'est-à-dire équivalente aux noms des variables liées près. Nous avions expliqué que, pour manipuler des représentations sémantiques spécifiées à l'aide du λ-calcul, il est important non seulement de traiter de la même façon des représentations α-équivalentes, mais aussi de savoir renommer les variables liées d'une expression pour éviter le phénomène de capture des variables. Les structures de données utilisées pour représenter les termes doivent donc se conformer à ces spécifications et permettre la manipulation de termes modulo α-équivalence. L'une des techniques les plus connues pour résoudre ce problème consiste à représenter les variables non pas par leur nom comme on le fait habituellement, mais par des nombres que l'on appelle indices de De Bruijn, nom de leur inventeur qui les introduisit dans (de Bruijn, 1972). Plus précisément, chaque occurrence d'une variable liée est remplacée par un nombre qui indique le nombre de lieurs qui la sépare de celui qui la lie. Par exemple, le terme $\lambda x.x$ serait représenté par $\lambda 1$ en notation avec indices de De Bruijn, tout comme le terme $\lambda y.y$. De la même façon, les termes $\lambda xyz.(xy)z$ et $\lambda uvw.(uv)w$, qui sont α-équivalents l'un à l'autre, ont avec la notation de De Bruijn une même représentation, à savoir le terme $\lambda\lambda\lambda(32)1$. On le voit, la notation introduite par De Bruijn est telle que tous les termes congruents modulo α-équivalence ont la même représentation, ce qui permet de ramener le test d'α-équivalence à un test d'égalité syntaxique et supprime le besoin de renommer les variables. Par ailleurs, l'implantation de substitutions, nécessaire à celle de la β-réduction, peut se faire relativement simplement lorsque les variables sont représentées par les indices de De Bruijn. En effet, les substitutions se ramènent essentiellement à des opérations arithmétiques simples.

Cependant, le choix d'une telle représentation nécessite un certain nombre d'opérations de conversion entre les représentations à base d'indices de De Bruijn et celles utilisant des noms de variables, ces dernières étant plus faciles à lire et à écrire. Ces opérations pouvant se révéler délicates à mettre en œuvre, des techniques alternatives à l'utilisation des indices de De Bruijn ont été proposées. Parmi les outils développés pour le langage `OCaml`, nous pouvons citer `Fresh OCaml` (Shinwell et Pitts, 2005) et `alphaCaml` (Pottier, 2005). Ces deux outils permettent de définir des structures récursives avec variables liées et font en sorte que, lorsqu'on accède à ces structures, les occurrences des variables liées soient renommées de façon à éviter les collisions entre noms de variables.

D'un point de vue technique, les solutions proposées par `Fresh OCaml` et `alphaCaml` sont relativement proches. L'une des différences notables entre ces deux outils est que `Fresh OCaml`

est une version modifiée de OCaml, tandis que alphaCaml consiste en un pré-processeur et une bibliothèque qui fonctionnent de concert avec la distribution standard de OCaml. Cette différence nous a conduit à préférer alphaCaml à Fresh OCaml. C'est donc ce système que nous utiliserons pour créer une structure de termes compatible avec la relation d'α-équivalence.

Une deuxième question qu'il est nécessaire de se poser est celle de la représentation précise des termes et des opérations dont on souhaite disposer pour les manipuler. Nous avons certes vu une définition inductive de TYn à la section 1.2.2, définition qui ne demande pas mieux que d'être traduite en OCaml. Cependant, nous pensons qu'une traduction aussi directe de la définition de la section 1.2.2 aurait pour conséquence de limiter la capacité de Nessie à évoluer pour éventuellement utiliser un autre système de type que TYn. Nous avons donc choisi une représentation de termes qui n'est pas une traduction directe des définitions de la section 1.2.2, mais qui se prête mieux, à notre avis, à de futures évolutions. C'est cette implantation que nous allons présenter maintenant. Pour ce faire, nous allons introduire les objets dont nous avons eu besoin, en commençant par définir une notion de signature proche de celle utilisée en réécriture. Nous montrerons ensuite comment il est possible de construire une signature par réunion de deux signatures existantes, puis nous passerons à la construction des termes proprement dits, cette construction consistant en une interface spécifiant les opérations disponibles sur les termes, et en un foncteur construisant une implémentation conforme à cette interface à partir d'une signature.

Lorsque nous avons défini TYn, nous avons donné des règles de typage associées à chaque opérateur. Pour les connecteurs logiques, par exemple, nous avons donné une règle indiquant que si leurs arguments sont de type t, alors leur résultat est aussi de type t et que cette combinaison de types est la seule possible. Cependant, pour d'autres symboles, les choses peuvent être légèrement plus complexes. Ainsi, pour le symbole =, on aimerait qu'il puisse prendre deux arguments de n'importe quel type et renvoyer une valeur de type t. Pour le dire en Caml, nous aimerions que l'égalité soit de type 'a -> 'a -> t, c'est-à-dire que l'égalité soit polymorphe, ou qu'il y ait autant de symboles d'égalité qu'il y a de types. Et ceci n'est pas vrai seulement pour l'égalité : les quantificateurs existentiels et universels ont eux aussi besoin d'être polymorphes, de type ('a -> t) -> t. Nous avons donc besoin de pouvoir associer aux opérateurs qui nous intéressent des règles de typage arbitraires, ce qui nous conduit à introduire les définitions suivantes :

```
type 'a symbol = {
  typingRule : 'a -> Type.t;
  representations : string list
}
type binder = (Type.t * Type.t) symbol
type unaryOperator = Type.t symbol
type binaryOperator = (Type.t * Type.t) symbol
```

Le premier type, 'a symbol, est un type polymorphe qui définit une structure regroupant toutes les informations relatives à un symbole. Le premier champ de cette structure est une fonction qui prend un argument dont le type n'est pas spécifié et renvoie un type. Comme son nom l'indique, cette fonction n'est autre que la règle de typage associée au symbole. Le type de son argument n'est pas spécifié car il dépend du type de symbole, comme le montrent les définitions suivantes et comme nous allons l'expliquer dans un instant. Le deuxième champ est une

liste de représentations possibles pour ce symbole, ce qui permet d'utiliser plusieurs notations pour un même symbole dans la syntaxe concrète des termes. Comme nous avons commencé à l'expliquer, les trois types suivants définissent trois familles de symboles particulières, à savoir des lieurs, des opérateurs unaires et des opérateurs binaires. Par exemple, un opérateur binaire est défini comme un symbole où le type `'a` est instancié par `Type.t * Type.t`. Ceci signifie qu'un opérateur binaire est vu comme un symbole dont la règle de typage prend en paramètre un couple de types (les types des opérandes de cet opérateur) et renvoie le type du résultat de l'application de l'opérateur à ses deux opérandes. La définition des opérateurs unaires est encore plus simple, puisqu'ils n'ont qu'une seule opérande et que par conséquent la règle de typage qui leur est associée n'a besoin que d'un argument de type `Type.t`. Enfin, la règle de typage pour les lieurs prend elle aussi deux arguments : l'un pour le type de la variable qu'elle lie, l'autre pour le type du terme dans lequel cette variable est liée.

Le concept de signature que nous utilisons s'appuie sur ces trois types instances du type `'a symbol`. En effet, pour nous, une signature sera une famille de trois ensembles d'opérateurs : des lieurs, des opérateurs unaires et des opérateurs binaires. Cette spécification est traduite en `OCaml` par le type de module suivant :

```
module type SIGNATURE = sig
  val binders : binder StringMap.t
  val unaryOperators : unaryOperator StringMap.t
  val binaryOperators : binaryOperator StringMap.t
end
```

Comme on peut le constater, la définition que nous venons de donner utilise un dictionnaire d'opérateurs unaires, un dictionnaire d'opérateurs binaires et un dictionnaire de lieurs, alors que nous avions annoncé une définition à l'aide d'ensembles. En réalité, ensembles et dictionnaires ne sont pas très différents et il aurait été possible d'utiliser des ensembles. Les clés utilisées pour les dictionnaires des signatures sont des noms utilisés par `Nessie` pour représenter les symboles en interne. Ces noms ne sont donc pas visibles de l'utilisateur.

Pour illustrer la spécification des signatures que nous venons de donner, voici deux exemples de signatures qui ont été définies et seront utilisées par la suite pour construire TYn. La première signature définie est celle du λ-calcul typé. Elle comporte un lieur λ et un opérateur binaire, l'application. La seconde signature décrit la logique du premier ordre, également dans un contexte typé. Elle définit pour sa part deux lieurs, à savoir le quantificateur existentiel et le quantificateur universel, un opérateur unaire (la négation logique) et les opérateurs binaires habituels : conjonction et disjonction, implication, équivalence et égalité. Voici par exemple comment est définie la signature du λ-calcul :

```
module LambdaCalculus : SIGNATURE = struct
  let typeLambda (varType, termType) =
    Type.arrow varType termType
  let typeApp (typeFunctor, typeArgument) =
    match typeFunctor with
      | (Type.Arrow t1,t2) ->
        if t1=typeArgument
        then t2
        else TypingError
```

```
      | _ -> TypingError
  let binders =
    StringMap.add
      "lam" {
        typingRule = typeLambda;
        representations = ["lam"]
      }
      StringMap.empty
  let unaryOperators = StringMap.empty
  let binaryOperators =
    StringMap.add
      "app" {
        typingRule = typeApp;
        representations = ["@";"app"]
      }
      StringMap.empty
end
```

Le module commence par définir les deux fonctions qui seront utilisées comme règles de typage pour l'abstraction et l'application. La règle pour l'abstraction (fonction `typeLam`) construit un type flèche à partir du type de la variable et du type de l'argument. La fonction `typeApp` qui définit la règle de typage associée avec l'application vérifie que le type du foncteur est de la forme $\alpha \to \beta$ et que l'argument est de type α, auquel cas elle renvoie β. Dans tous les autres cas, une erreur de typage est reportée. La suite du module définit les trois dictionnaires qu'un module doit définir pour se conformer à la spécification des signatures qui a été introduite précédemment. Nous ne donnons pas ici la définition de la signature pour la logique du premier ordre, car elle est similaire à celle que l'on vient de voir. Disons seulement que cette seconde signature a été définie dans un module `FirstOrderLogic` qui, tout comme le module `LambdaCalculus` qui vient d'être introduit, est conforme à la spécification `SIGNATURE`. Précisons simplement le fonctionnement de la règle de typage associée au quantificateur universel et au quantificateur existentiel. Cette règle n'impose aucune contrainte sur le type des variables quantifiées. Elle demande seulement que le terme apparaissant dans sa portée soit de type `t`.

Une fois que l'on dispose de plusieurs signatures, on peut vouloir en construire de nouvelles à partir de celles déjà construites. L'opération qui semble alors la plus naturelle est la réunion de signatures, qui permet, étant données deux signatures, d'en construire une contenant tous les symboles définis par les signatures préexistantes. Nous avons donc défini la réunion de deux signatures à l'aide d'un foncteur prenant en argument deux modules se conformant à la spécification `SIGNATURE` et construisant un nouveau module conforme à cette spécification. Voici la définition de ce foncteur :

```
module Union (S1 : SIGNATURE) (S2 : SIGNATURE) : SIGNATURE =
struct
  let merge d1 d2 =
    let intersection =
      StringSet.inter
        (StringMap.domain d1)
        (StringMap.domain d2)
```

```
    in
    if StringSet.is_empty intersection
    then StringMap.union d1 d2
    else MergeError
  let binders =
    merge S1.binders S2.binders
  let unaryOperators =
    merge S1.unaryOperators S2.unaryOperators
  let binaryOperators =
    merge S1.binaryOperators S2.binaryOperators
end
```

Le module commence par définir une fonction privée `merge` qui retourne la réunion de deux ensembles de symboles, à condition que ces deux ensembles soient disjoints. Ceci permet d'éviter de réunir des signatures qui définiraient toutes deux un même symbole, l'opération d'union n'ayant alors pas vraiment de sens. Une fois cette fonction définie, la construction de la signature union est facile. Et, grâce à ce module, nous pouvons construire la signature `LCFOL` qui réunit les deux signatures que nous avons introduites précédemment :

```
module LCFOL : SIGNATURE =
  Union (LambdaCalculus) (FirstOrderLogic)
```

Nous avons ainsi construit de façon élégante une signature permettant de combiner logique du premier ordre et λ-calcul, ce qui peut être vu comme un premier pas vers la construction d'une implémentation de TYn. Pour compléter cette construction, il nous reste à présenter la spécification des termes que nous allons utiliser et à montrer comment construire un module conforme à cette spécification à partir d'une signature. Voici un extrait de la spécification des termes :

```
module type TERM =
struct
  type t
  type environment
  val empty : environment
  val addVariable :
    string -> Type.t -> environment -> environment
  val addConstant :
    string -> Type.t -> t option -> environment -> environment
  val getVariableType :
    environemnt -> string -> Type.t
  val getConstanttType :
    environemnt -> string -> Type.t
  val typeof :
    environment -> t -> Type.t
  val alphaEquivalent :
    t -> t -> bool
  val betaReduce : t -> t
  val constSubst : string -> string -> t -> t
  val unfoldConstants :
    environment -> StringSet.t -> t -> t
end
```

Nous avons signalé que la spécification que nous venons de donner n'est qu'un extrait de celle réellement utilisée, parce que la spécification réelle comporte des fonctions de conversion entre termes et chaînes de caractères et d'autres fonctions utilitaires que nous n'avons pas jugé utile de mentionner ici. Pour ce qui est de la spécification ci-dessus, elle commence par déclarer un type `t` correspondant aux termes. Puis, nous déclarons le type `environment` décrivant des environnements de typage. Ces structures de donnée sont utilisées lors du calcul des types des termes. Les environnements que nous utilisons ici sont cependant un peu différents de ceux rencontrés habituellement en λ-calcul. En effet, ceux utilisés habituellement ne font qu'associer des types à des variables, tandis que ceux que nous utilisons ici proposent deux fonctionnalités supplémentaires. D'une part, ils permettent de garder trace des constantes et de leur type, en plus des variables. D'autre part, ils permettent de *définir* des constantes en leur associant un λ-terme par lequel elles peuvent être remplacées. La spécification précédente introduit 4 fonctions de manipulation d'environnements : `addVariable` (resp. `addConstant`) qui permettent d'ajouter une variable (resp. une constante) à un environnement, et `getVariableType` (resp. `getConstantType`) qui permettent de connaître le type d'une variable (resp. d'une constante). On peut noter la légère différence entre les déclarations de `addVariable` et `addConstant`. En effet, cette dernière fonction prend un paramètre supplémentaire qui est un terme optionnel. Si ce paramètre vaut `None`, la constante n'est que déclarée et aucune valeur ne lui est associée, tandis que s'il vaut `Some t`, alors la constante est définie et le terme `t` lui est associé. La fonction `typeof` permet de calculer le type d'un terme ; nous verrons un peu plus loin comment elle a été implémentée et reviendrons sur l'algorithme choisi au chapitre 8.

Pour en revenir aux termes, il convient de remarquer que la vérification du type d'un terme est découplée de sa construction. Pour le dire autrement, disposer d'une valeur de type `t` ne garantit en rien que le terme qu'elle représente est bien typé. Il est nécessaire de séparer la vérification du type d'un terme de sa construction car les informations nécessaires à la vérification de type ne sont pas toujours disponibles au moment de la construction du terme.

Les deux fonctions suivantes de la spécification, `alphaEquivalent` et `betaReduce`, nécessitent à notre avis moins de commentaires. La première permet de décider si deux termes sont α-équivalents, tandis que la deuxième permet de β-réduire un terme jusqu'à en obtenir une forme normale. La fonction `constSubst` permet de remplacer une constante par une autre dans un terme. Nous utiliserons cette fonction lorsque nous introduirons les arguments d'occurrences, section 2.2.4. Enfin, la fonction `unfoldConstants` permet de remplacer des constantes par leurs définitions. Comme nous le verrons lorsque nous présenterons les lexiques, cette fonction est utilisée pour implanter un mécanisme de macros qui facilite grandement l'écriture de lexiques en améliorant leur factorisation et en fournissant un niveau d'abstraction qui se révélera des plus utiles pour construire des représentations sémantiques différentes à partir d'un même arbre syntaxique, comme nous le ferons au chapitre 6.

Il nous reste à dire quelques mots du foncteur `Make` utilisé pour construire un module respectant la spécification `TERM` à partir d'un module `S` conforme à la spécification `SIGNATURE`. En réalité, ce module est surtout une interface entre `Nessie`, d'une part et, d'autre part, un module générique de gestion de termes construit à l'aide de `alphaCaml`. À titre d'information, voici comment est décrit le type des termes dans le langage de spécification de `alphaCaml` qui, comme on pourra le constater, est très proche du langage `OCaml` tel qu'il a été introduit à la section précédente :

```
sort variable
type term =
  | Cst of [string]
  | Var of atom variable
  | Bin of [string] * <binding>
  | Uop of [string] * term
  | Bop of [string] * term * term

type binding binds variable =
  atom variable * [Type.t] * inner term
```

La première ligne de cette spécification exprime que les objets de type `variable` peuvent apparaître libres ou liés dans les types dont les déclarations suivent. Le type `term` est ensuite défini récursivement, à l'aide du type auxiliaire `binding`. Pour lire ces définitions, il est utile de savoir que les types apparaissant entre crochets `[]` désignent des types `Caml` traditionnels, tandis que les types apparaissant entre chevrons <> correspondent à des types définis ultérieurement dans la spécification `alphaCaml`, comme l'illustre l'exemple de `<binding>`. D'après la première définition de type, un terme peut donc être une constante, une variable, un lieur, un opérateur unaire ou un opérateur binaire. Les constantes sont représentées par des chaînes de caractères traditionnelles, tandis que les variables sont des « atomes » gérés par `alphaCaml`. Les opérateurs unaires et binaires prennent en premier argument le nom de l'opérateur à construire. Les noms qui seront utilisés comme premier argument des constructeurs `Uop` et `Bop` sont les représentations internes des opérateurs que nous avons mentionnés précédemment lorsque nous avons introduit les signatures. Il en va d'ailleurs de même pour le constructeur `Bin` représentant les lieurs : son premier argument est le nom du lieur qu'il représente. Le second, de type `binding` est un triplet comportant une variable, le type de celle-ci et un terme. Le mot réservé `inner` spécifie que l'on est en train de définir un type avec variables liées.

À partir de cette spécification, `alphaCaml` est en mesure de produire un module de gestion de termes modulo renommage des variables. C'est ce module, appelé `AlphaTerm`, que nous utilisons continuellement dans notre foncteur `Make` construisant des termes à partir de signatures. Le module produit par `alphaCaml` à partir de la spécification précédente définit en fait deux représentations des termes. Dans l'une, dite « crue », les variables sont représentées par des chaînes de caractères, ce qui permet de les manipuler comme on le souhaite mais n'offre aucune garantie quant aux conflits de noms. Dans l'autre, dite « interne », les variables sont vues comme des données d'un type abstrait que seul `alphaCaml` peut manipuler, ce qui garantit qu'il n'y aura pas de collisions entre variables. Dans notre foncteur, nous utilisons la forme interne des termes et ne passons à la forme « crue » que pour leur affichage. Comme ce foncteur est d'une taille relativement conséquente, nous n'allons pas présenter ici son contenu dans son intégralité. Nous nous contenterons de donner le code de la fonction `typeof`, d'une part parce que ce code montre bien comment le foncteur utilise les outils fournis par le module `AlphaTerm`, d'autre part parce que nous aurons à revenir ultérieurement sur l'algorithme de calcul des types mis en œuvre. Voici donc un extrait du foncteur `Make` construisant une implémentation des termes à partir d'un module conforme à la spécification `SIGNATURE` :

```
module Make (S : SIGNATURE) : TERM =
struct
```

```
  type t = AlphaTerm.term
  let rec typeof env = function
    | AlphaTerm.Cst c -> getConstantType env c
    | AlphaTerm.Var v -> getVariableType env v
    | AlphaTerm.Bin (binder, (var, varType, trm)) ->
      let newEnv = addVariable var varType env in
      let trmType = typeof newEnv trm in
      let typingRule =
        (StringMap.find binder S.binders).typingRule
      in typingRule (varType, trmType)
    | AlphaTerm.Uop (uop, trm) ->
      let trmType = typeof env trm in
      let typingRule =
        (StringMap.find uop S.unaryOperators).typingRule
      in typingRule trmType
    | AlphaTerm.Bop (bop, trm1, trm2) ->
      let trm1Type = typeof env trm1 in
      let trm2Type = typeof env trm2 in
      let typingRule =
        (StringMap.find bop S.binaryOperators).typingRule
      in typingRule (trm1Type, trm2Type)
end
```

On le voit, l'algorithme de typage est défini inductivement. Le type des variables et des constantes est extrait de l'environnement, tandis que pour les lieurs, opérateurs unaires et binaires, le type est obtenu à partir des types des sous-termes et des règles de typage associées aux symboles considérés par la signature passée en paramètre au module `Make`.

Nous obtenons finalement notre implémentation de TYn par application du foncteur `Make` à la signature `LCFOL` définie précédemment :

```
module TYn = Make (LCFOL)
```

Syntaxe concrète

Jusqu'ici, nous ne nous sommes préoccupés que de la représentation interne des termes. Cependant, comme on souhaite pouvoir utiliser des termes, tant dans les lexiques que dans les arbres, il est nécessaire d'adopter une syntaxe concrète permettant l'écriture de termes sous la forme de chaînes de caractères. Idéalement, la représentation des termes sous forme de chaînes de caractères devrait être à la fois facile à lire et à écrire pour l'utilisateur, et facile à analyser pour le système. Dans la pratique, ces contraintes peuvent s'avérer difficiles à concilier, dans la mesure où le souci de faciliter la lecture et l'écriture par un humain suggère l'utilisation d'une syntaxe aussi légère que possible, par exemple en proposant des notations minimisant le nombre de parenthèses à utiliser, tandis que la recherche d'un langage facile à analyser par un programme conduirait plutôt à un alourdissement de la syntaxe, avec beaucoup de délimiteurs pour lever toute ambiguïté.

Remarquons aussi que les termes que nous voulons écrire sont assez différents des expressions rencontrées dans les langages de programmation traditionnels. En effet, aux constructions

utilisant des opérateurs s'ajoutent dans notre cas celles utilisant des lieurs, qui n'apparaissent pas en général dans les langages traditionnels, exception faite des langages fonctionnels. Nous ne pouvons donc pas vraiment nous inspirer des langages de programmation quant à la syntaxe à adopter pour représenter les termes. On pourrait s'inspirer des logiciels d'aide à la preuve en logique d'ordre supérieure, puisque ceux-ci utilisent la λ-abstraction, la quantification universelle et la quantification existentielle en plus des opérateurs. Cependant, les analyseurs syntaxiques utilisés dans ce type d'application sont complexes, dans le sens où les langages qu'ils reconnaissent n'appartiennent pas à la classe des langages LALR. En particulier, les analyseurs pour de tels langages peuvent être modifiés pendant l'exécution, ce qui permet de les enrichir à l'exécution. Une telle flexibilité est non seulement difficile à mettre en œuvre, mais aussi, nous semble-t-il, relativement peu utile dans le cas de `Nessie`.

Nous avons donc défini une notation pour les termes correspondant à nos besoins. Plus précisément, nous n'avons pas cherché une notation permettant d'exprimer tous les termes reconnus par tous les modules se conformant à la spécification `TERM`. Au lieu de cela, nous avons préféré développer un analyseur syntaxique permettant de construire uniquement des termes de TYn, c'est-à-dire des termes utilisables par le module `TYn` vu précédemment.

Durant ce développement, notre principale difficulté a été une certaine hésitation, entre d'une part une syntaxe s'inspirant du λ-calcul, et, d'autre part, une syntaxe proche de la logique du premier ordre, puisque c'est souvent ce langage logique qui est utilisé pour construire les représentations finales, le λ-calcul n'étant qu'un langage permettant de spécifier comment combiner des représentations. Concrètement, si nous prenons l'exemple du prédicat binaire `love` prenant en argument deux entités `x` et `y` et renvoyant vrai si `x` aime `y`, notre question était de savoir s'il valait mieux écrire `love(x,y)` ou `((love x) y)`. En effet, la première expression est plus naturelle du point de vue de la logique du premier ordre, tandis que la seconde (ou sa forme abrégée `(love x y)`) a plus de sens en λ-calcul. La forme `(love x y)` pourrait convenir, mais elle est plus difficile à obtenir à l'affichage, car il convient de bien placer les parenthèses, problème que ne pose aucune des deux autres formes.

Il pourrait sembler, au premier abord, que ces questions ne relèvent que de choix purement techniques et sans grande importance. En réalité, elles ne sont que la manifestation d'un phénomène plus profond qui nous est apparu en cours d'implémentation. En effet, les questionnements que nous venons d'esquisser proviennent du fait que des termes tels que `love(x,y)` et `((love x) y)` utilisent non pas une seule et même notion d'application, mais bien deux opérations d'application distinctes. Dans le cas de `love(x,y)`, l'application utilisée est une opération algébrique visant à appliquer une fonction à un ensemble d'arguments. La « fonction » dont il est ici question ne peut être qu'un symbole, et non un terme. Autrement dit, il y a une contrainte d'ordre syntaxique sur l'ensemble des fonctions possibles. En outre, l'arité de cette opération d'application dépend du symbole considéré et, étant donné un symbole, ce qui est attendu est que toutes les occurrences de ce symbole apparaissent appliquées au même nombre d'arguments. D'un autre côté, les applications apparaissant dans le terme `((love x) y)` sont des applications du λ-calcul. Or cette opération est une opération algébrique d'arité 2 par laquelle une fonction est appliquée à un argument. Contrairement à l'application vue précédemment, celle du λ-calcul n'impose aucune restriction syntaxique sur la fonction, ni sur l'argument (hormis des contraintes de typage si l'on se place dans le cadre du λ-calcul typé).

Pour que la spécification des termes puisse se faire de façon intuitive, il nous a semblé im-

portant de supporter à la fois ces deux opérations, dans la mesure du possible. Ainsi, si une constante `love` de type `e -> e -> t` est déclarée, et si `x` et `y` sont deux variables de type `e`, alors la syntaxe concrète que nous utilisons pour les termes permet d'écrire, de façon équivalente, `love(x,y)` et `((love @ x) @ y)`, où `@` est le symbole que nous utilisons pour représenter, de façon infixe, l'opération d'application du λ-calcul. De la même façon, nous permettons de choisir, lors de l'affichage des termes, laquelle de ces deux notations est utilisée.

Cette façon de procéder oblige à avoir une syntaxe avec beaucoup de parenthèses, ce qui tend à rendre l'écriture de termes parfois fastidieuse. Avec le recul, il nous semble donc qu'il aurait peut-être été plus pertinent de ne supporter qu'une seule opération, à savoir l'application du λ-calcul, ce qui aurait donné lieu à une syntaxe plus uniforme. Cependant, même avec une telle syntaxe, il reste des opérateurs qu'il est souhaitable, à notre avis, de représenter de façon infixe, comme par exemple les connecteurs logiques. En effet, bien que ces opérateurs puissent être vus comme des constantes du λ-calcul, leur notation infixe est plus intuitive. De même, les opérateurs unaires tels que la négation, tout comme les lieurs, restent difficiles à analyser si l'on souhaite pouvoir les écrire de façon un tant soi peu intuitive.

À titre indicatif, voici quelques termes reconnus par `Nessie` et qui devraient permettre au lecteur de se familiariser suffisamment avec sa syntaxe concrète pour pouvoir lire sans difficultés les exemples qui apparaîtront dans la suite de cette thèse. Nous supposons que nous travaillons dans TY1 avec pour seul type autre que `t` le type `e` des entités. Nous avons alors :

- `lam x, y : e. ( love(x,y) );`
- `lam P, Q : e->t. exists x : e. ( P(x) => Q(x) );`
- `lam P, Q : e->t. forall x : e. ( P(x) && Q(x) ).`

2.2.2 Lexiques

Comme nous l'avons vu au début de cette section, les lexiques ont pour fonction principale d'associer à des mots leur représentation sémantique, celle-ci étant décrite par un λ-terme typé. Or, il est possible que les représentations sémantiques que l'on souhaite associer aux mots utilisent des types ou des constantes propres au formalisme sémantique particulier auquel on s'intéresse.

Un lexique doit donc permettre non seulement de définir des associations entre des mots et leurs représentations sémantiques, mais aussi de déclarer les types et constantes que l'on souhaite utiliser dans ces représentations. En outre, comme les représentations sémantiques que l'on souhaite associer aux mots sont, en toute généralité, des termes, il est clair que le type des lexiques doit, d'une façon ou d'une autre, dépendre du type des termes qu'ils contiennent.

Nous allons commencer par écrire une première spécification des lexiques, abstraite et de haut niveau, qui permet de donner une idée de leur utilisation. Comme nous l'avons fait pour les termes, nous donnerons à la spécification des lexiques la forme d'un type de module. Une fois cette spécification donnée, nous réfléchirons à la représentation interne concrète des lexiques, représentation que nous construirons progressivement.

Nous l'avons déjà dit, on peut schématiquement envisager un lexique comme étant un dictionnaire associant à chaque mot sa représentation sémantique. Bien que ce point de vue soit une approximation assez grossière de l'implantation réelle à laquelle nous arriverons en fin de section, un tel point de vue suffit pour écrire une première spécification des lexiques, que voici :

```
module type LEXICON =
sig
  type term
  type environment
  type t
  val empty : t
  val addBasicType : string -> t -> t
  val addAliasType : string -> Type.t -> t -> t
  val addConstant : string -> Type.t -> term option -> t -> t
  val addWord : string -> term -> t -> t
  val find : string -> t -> term
  val environment_of_lexicon : t -> environment
end
```

Cette spécification commence par déclarer le type des termes qui seront stockés dans le lexique. On pourrait se demander pourquoi nous n'utilisons pas le type `TYn.t` défini précédemment. La raison pour un tel choix est que nous ne souhaitons pas imposer ce type dans la spécification des lexiques, ce qui permet de construire des lexiques s'appuyant sur des structures de termes autres que celles que nous avons définies. Il va cependant de soi que, dans le foncteur que nous allons présenter pour implanter les lexiques, c'est bien le type des termes construits précédemment que nous utiliserons. En outre, ces remarques s'appliquent au type `environment` dont la déclaration suit celle de `term`.

La spécification se poursuit en déclarant un type `t` pour les lexiques, ainsi qu'un lexique vide. Viennent ensuite les fonctions sur les lexiques, que nous pouvons répartir en deux groupes : celles permettant d'enrichir un lexique, et celles permettant de le consulter (pour l'instant cette seconde catégorie ne contient que les fonctions `find` et `environment_of_lexicon`, toutes les autres fonctions appartenant à la première catégorie).

La suite de cette section consiste à implanter un foncteur qui, étant donné un module `Term` conforme à la spécification `TERM`, construit un lexique associant à chaque mot une représentation de type `Term.t`. Au cours de l'implantation de ce foncteur, nous serons amenés à raffiner quelque peu notre définition des lexiques et ces raffinements induiront quelques changements dans la spécification donnée précédemment.

Puisque nous envisageons les lexiques comme des structures associant à chaque mot sa représentation sémantique, une première idée est de les représenter en utilisant un environnement pour stocker les types et constantes qui ont été déclarées, ainsi qu'un dictionnaire ayant pour clés des lexèmes et pour valeurs associées leurs représentations sémantiques. Nous obtenons donc une première implémentation du foncteur de construction de lexiques dont voici un extrait :

```
module Make (Term : TERM) : LEXICON =
struct
  type term = Term.t
  type environment = Term.environement
  type t = {
    environment : environment;
    entries : term StringMap.t
  }
end
```

Cette façon d'envisager les lexiques, bien qu'elle donne une bonne vision intuitive de leur rôle, n'en demeure pas moins naïve. En effet, elle ne tient pas compte de certaines données linguistiques qui, si elles étaient prises en compte, pourraient rendre les lexiques plus concis et leur développement plus aisé et plus efficace.

La première donnée linguistique dont il nous paraît souhaitable de tenir compte est que la sémantique d'un mot peut être obtenue à partir d'une sémantique « noyau » que l'on souhaite modifier en fonction de la forme particulière du mot que l'on rencontre. Ainsi, dans beaucoup de théories sémantiques, la sémantique d'un verbe conjugué est construite à partir d'une représentation sémantique du verbe à l'infinitif, celle-ci étant modifiée par d'autres formules représentant le temps et la personne auxquels le verbe est conjugué [6]. Or, si l'on utilise des lexèmes comme clés du lexique, il n'est pas possible de tenir compte de cette façon de représenter la sémantique des verbes. On est en effet obligé de définir une entrée par forme conjuguée du verbe, ces entrées étant toutes différentes et ne faisant pas apparaître le procédé de construction sémantique que nous venons de décrire.

Il ne semble donc pas très efficace d'utiliser les formes fléchies (lexèmes) comme clés dans un lexique comme celui que nous voulons construire. Une solution qui semble plus prometteuse consiste à utiliser comme clé les lemmes, c'est-à-dire des formes non fléchies des lexèmes. Pour le cas des verbes que nous venons de citer, les formes fléchies correspondent aux verbes conjugués, et les lemmes aux verbes à l'infinitif. L'utilisation des lemmes comme clés du lexique sémantique ne nécessite aucun changement concernant le type du lexique introduit précédemment, les dictionnaires avec pour clés des chaînes de caractères et pour valeurs associées des termes étant toujours utilisables.

Nous venons ainsi de prendre en compte une première forme de régularité des langues qui permet une première simplification des lexiques.

Cependant, la forme de régularité des langues dont nous venons de faire état n'est pas la seule à être intéressante dans le contexte du développement de lexiques sémantiques qui est le nôtre. Nous pouvons aussi remarquer que certains groupes de mots ont des sémantiques non pas identiques, mais similaires. Il en est ainsi dans beaucoup de théories, par exemple, pour les verbes transitifs : ils sont tous représentés par une formule logique dans laquelle apparaît un prédicat binaire correspondant au verbe considéré, et dont les arguments représentent le sujet et l'objet. Le prédicat est le seul élément qui varie d'une représentation sémantique à l'autre. Un autre exemple que nous pouvons citer est celui des noms propres. Avec un lexique basé sur un dictionnaire tel que celui dont nous disposons pour le moment, nous serions obligés, pour définir la sémantique de Marie, Jean et Pierre, d'ajouter au lexique trois entrées : une spécifiant que la sémantique de Marie est $\lambda P : e \rightarrow t.(P\text{Marie })$, une autre indiquant que la sémantique de Jean est $\lambda P : e \rightarrow t.(P\text{Jean})$ et une troisième spécifiant que la sémantique de Pierre est $\lambda P : e \rightarrow t.(P\text{Pierre})$. Au lieu de cela, on préférerait disposer d'une notion de famille de mots ayant tous des représentations similaires, familles qu'il serait ensuite possible d'étendre tout à loisir. On pourrait par exemple déclarer une famille pour les noms propres et indiquer que tous les mots appartenant à cette famille ont pour représentation $\lambda P : e \rightarrow t.(P_)$, où _ est une sorte de variable à remplacer par le nom propre dont on souhaite obtenir la représentation. Il suffirait ensuite de préciser que Marie, Jean et Pierre appartiennent à cette famille et de modifier la

[6] Nous en verrons un exemple avec les verbes polonais au chapitre 4.

fonction de recherche de représentations sémantiques pour qu'elle construise une représentation. Pour parvenir à ce résultat, il est nécessaire de changer la représentation des lexiques, celle basée uniquement sur des dictionnaires n'étant pas suffisamment flexible. Nous commençons par introduire un type permettant de rassembler toutes les données relatives à chaque famille, à savoir l'ensemble de lemmes appartenant à cette famille, un terme spécifiant comment construire les représentations sémantiques pour les lemmes de cette famille, et enfin un type, celui des constantes logiques associées aux lemmes. Un lexique peut alors être envisagé comme un dictionnaire dont les clés sont les noms des familles et dont les valeurs associées sont les familles telles que nous venons de les décrire. Nous obtenons ainsi les définitions suivantes :

```
module Make (Term : TERM) : LEXICON =
struct
  type term = Term.t
  type family = {
    lemmaType : Type.t;
    pattern : Term.t;
    lemmas : StringSet.t
  }
  type t = {
    environment : environment
    entries : family StringMap.t
  }
end
```

Bien sûr, la spécification `LEXICON` doit elle aussi être modifiée. Plus précisément, la fonction `addWord` permettant d'ajouter un mot à un lexique n'est plus adaptée à notre nouvelle façon d'envisager ces derniers. Nous devons la remplacer par des fonctions `addFamily` permettant d'ajouter une famille à un lexique et `addLemma` permettant l'ajout d'un lemme à une famille donnée d'un lexique. Voici les spécifications de ces deux fonctions :

```
val addFamily : string -> Type.t -> Term.t -> t -> t
val addLemma : string -> string -> t -> t
```

La recherche de la représentation sémantique associée à un lemme ne peut plus se faire par simple consultation du lexique. En effet, le terme stocké dans le champ `pattern` est un terme générique donnant un modèle (patron) à partir duquel peuvent être construites toutes les représentations des lemmes d'une même famille. Par exemple, le terme stocké dans le champ `pattern` de la famille des noms propres vue précédemment pourrait être

```
lam P:e->t. ( P(_))
```

À partir de ce terme, la représentation du nom propre `Mary` peut être obtenue en remplaçant le symbole _ par `Mary`. Il ne suffit donc plus de consulter le lexique pour obtenir une représentation sémantique, il faut en outre modifier le terme obtenu pour une famille de sorte à l'adapter au lemme particulier dont on cherche une représentation. Nous appellerons ce processus de calcul d'une représentation une *instanciation*, parce que le terme représentant la sémantique pour une famille est en quelque sorte instancié pour obtenir la représentation d'un lemme particulier. En

outre, il arrive parfois qu'un lemme appartienne à plusieurs familles. C'est par exemple le cas du mot français « ferme » qui est un substantif, un adjectif, *etc.* Il n'est donc pas suffisant de donner un lemme à la fonction d'instanciation pour en obtenir une représentation sémantique. Il faut en plus spécifier à quelle famille il appartient, pour lever toute ambiguïté. Nous allons donc définir un type `entry` regroupant lemme et famille et ajouter cette définition tant à la spécification `LEXICON` qu'au foncteur `Make` :

```
type entry = {
  lemma : sring;
  family : string
}
```

C'est à partir d'une valeur de ce type que la fonction d'instanciation pourra calculer la représentation sémantique pour une entrée de lexique. Nous remplaçons donc la fonction `find` de la spécification `LEXICON` par la fonction `instanciate` suivante :

```
val instanciate : t -> entry -> term
```

Le champ `lemmaType`, quant à lui, donne le type des constantes logiques représentant les lemmes et qui devront être ajoutées à l'environnement de typage pour que le type d'une représentation sémantique puisse être calculé. Il est important de comprendre que le type contenu dans ce champ est celui des *constantes logiques* introduites par les lemmes et non celui des termes construits pour les représenter. Ainsi, le type que doit contenir le champ `lemmaType` de la famille des noms propres est `e`, qui est le type des constantes `jean`, `marie` et `pierre`, et non pas `(e->t)->t` qui est le type du terme donnant la sémantique d'un nom propre.

Nous en sommes donc à un modèle de lexique consistant en des déclarations de types et constantes et en la définition de familles et de lemmes, les définitions des lemmes indiquant à quelle famille ils appartiennent et les définitions de familles contenant une représentation sémantique générique à partir de laquelle celle des lemmes peut être obtenue en remplaçant la variable _ par le lemme considéré. Ce modèle permet certes de partager au mieux les représentations sémantiques, mais il manque pourtant encore de flexibilité.

En effet, il existe des familles dont les lemmes n'ont pas des représentations sémantiques aussi proches que celles des noms propres vues précédemment. Les déterminants, par exemple, constituent une famille dont les lemmes (déterminant universel, indéfini, défini, *etc.*) ont des représentations sémantiques très différentes les unes des autres.

Pour que cette donnée linguistique puisse être prise en compte, il convient donc d'enrichir quelque peu le modèle de lexique vu précédemment, en y ajoutant la possibilité de définir des familles et des lemmes de sorte que l'essentiel de l'information sémantique puisse également être stockée dans les définitions des lemmes, et plus seulement dans les définitions de familles. Cet enrichissement nous conduit donc à envisager des lexiques pouvant contenir deux types de définitions de familles et de lemmes :

– Des définitions de familles contenant l'information sémantique sous la forme de patrons, les représentations sémantiques des lemmes appartenant à ces familles étant obtenues par instanciation des dits patrons (ceci correspond aux familles dont nous disposons pour le moment) ;

– Des définitions de familles ne contenant pas d'information sémantique, celle-ci étant portée par les définitions des lemmes appartenant à ces familles (par exemple les déterminants).

Nous avons donc mis en évidence deux types de familles : celles pour lesquelles l'essentiel de l'information sémantique est commun à la famille, et celles pour lesquelles ce n'est pas le cas. Cette façon de caractériser les familles, qui peut paraître quelque peu abstraite, n'est cependant pas la seule qui permette de distinguer les premières familles des secondes. En effet, on peut trouver entre les deux sortes de familles une différence d'une nature bien plus linguistique que celle, purement formelle, que nous avons dégagée précédemment : si les premières familles sont susceptibles d'être enrichies par l'ajout de mots (lemmes) nouveaux, ce n'est pas le cas (ou de façon très marginale) pour les secondes. Ainsi, l'adaptation des langues aux évolutions constantes des besoins expressifs des sociétés qui les utilisent conduit-elle naturellement à la création de nouveaux verbes, noms communs ou adjectifs, tandis que l'apparition de nouveaux déterminants est extrêmement rare. Le fait que des familles telles que celles des verbes et des noms communs puissent être étendues de façon naturelle a conduit les linguistes à baptiser ces familles des familles ouvertes, tandis que des familles telles que celle des déterminants, peu susceptibles d'évoluer sont appelées des familles fermées. C'est cette terminologie que nous utiliserons par la suite, en l'étendant aux lemmes, ceux appartenant à une famille ouverte étant appelés lemmes ouverts, tandis que ceux appartenant à une famille fermée seront appelés lemmes fermés.

Bien sûr, il nous faut maintenant adapter à la fois notre spécification abstraite des lexiques et leur implantation concrète, de sorte que ceux-ci puissent contenir à la fois des familles ouvertes et des familles fermées. Pour ce qui est de la spécification abstraite, nous allons renommer les fonctions `addFamily` et `addLemma` en `addOpenFamily` et `addOpenLemma` et y ajouter les fonctions `addClosedFamily` et `addClosedLemma`. En outre, nous allons utiliser dans les prototypes de ces fonctions le type `entry` que nous avons introduit précédemment pour regrouper un nom de lemme et un nom de famille. Nous obtenons ainsi les 4 spécifications suivantes :

```
val addClosedFamily : string -> t -> t
val addOpenFamily : string -> Type.t -> term -> t -> t
val addClosedLemma : entry -> term -> t -> t
val addOpenLemma : entry -> t -> t
```

Les lexiques peuvent alors être représentés à l'aide des structures de données suivantes, qui sont très proches de celles effectivement utilisées par `Nessie` (l'élément manquant concerne les constantes d'occurrence, que nous introduirons en section 2.2.4) :

```
type openFamily = {
  lemmaType : Type.t;
  pattern : Term.t;
  lemmas : StringSet.t
}

type closedFamily = term StringMap.t

type family =
  | Open of openFamily
  | Closed of closedFamily
```

```
type t = {
  environment : Term.environment
  entries : family StringMap.t
}
```

Comme on peut le constater, les lexiques sont toujours représentés à l'aide d'une structure contenant un environnement et un dictionnaire associant à chaque nom de famille les données la concernant. En revanche, la description d'une famille a évolué. Une famille peut maintenant être ouverte ou fermée. La définition des familles ouvertes de cette version du lexique correspond à la définition des familles du lexique précédent. Quant aux familles fermées, elles contiennent un dictionnaire associant à chaque lemme de la famille sa représentation, conformément à la première intuition que nous avions quant aux lexiques. Pour compléter notre définition, voyons comment fonctionne la fonction d'instanciation qui calcule une représentation sémantique à partir d'une entrée contenant un lemme et une famille :

```
let instanciate lex e =
  match StringMap.find e.family lex.entries with
    | Open f ->
      Term.constSubst "_" e.lemma f.pattern
    | Closed f ->
      StringMap.find e.lemma f
```

La fonction commence par chercher la famille contenue dans l'entrée `e` dans le lexique `lex`. Si la famille s'y trouve, le calcul se poursuit par un filtrage de motif :

- Si la famille est ouverte, alors le terme qu'il convient de renvoyer est celui stocké dans son champ `pattern`, où la constante _ a été remplacée par le lemme contenu dans l'entrée `e` ;
- Si la famille est fermée, il suffit de renvoyer le terme associé par cette famille au lemme contenu dans `e`.

Nous terminons cette présentation des lexiques de `Nessie` en donnant un exemple illustrant leur syntaxe concrète. Le lexique que voici déclare un type atomique `e`, un alias `pred`, une famille fermée `det` dont le seul lemme est le déterminant indéfini, et deux familles ouvertes `iv` (verbes intransitifs) et `noun` (substantifs) :

```
type e;
type pred = e->t;
family det;
lemma indef {
  family = det;
  term = lam P, Q : pred. exists x:e. ( P(x) => Q(x) )
};

family iv {
  type = pred;
  pattern = lam x:e. (  _(x) )
};

lemma walk,dance : iv;
```

```
family noun {
  type = pred;
  pattern = lam x:e. (  _(x) )
};

lemma man, woman : noun;
```

Ce lexique devrait être assez aisé à lire compte tenu de ce qui précède. Remarquons qu'il ne définit aucune constante, mais nous aurons l'occasion de montrer des lexiques plus conséquents par la suite, notamment aux chapitres 3, 4 et 6.

2.2.3 Arbres

À la sous-section précédente, nous avons vu comment définir un lexique permettant d'associer à chaque mot sa représentation sémantique. Dans cette section, nous allons présenter un langage permettant de spécifier comment combiner ces représentations sémantiques pour obtenir celles de syntagmes plus complexes et, finalement, la représentation sémantique de phrases et de discours.

Comme nous voulons que la combinaison des représentations des mots soit guidée par la structure syntaxique de la phrase, nous supposons que l'étape d'analyse syntaxique peut rendre compte de cette structure et demandons qu'elle le fasse en utilisant un arbre. En première approximation, disons que les feuilles de cet arbre correspondent essentiellement aux mots de la phrase et que les nœuds de l'arbre représentent les syntagmes plus complexes, leurs sous-arbres correspondant aux syntagmes plus simples à partir desquels les syntagmes plus complexes sont construits. Intuitivement, les arbres qui sont envisagés ici sont proches des arbres de dérivation qui apparaissent notamment lorsque l'on prouve qu'une certaine chaîne appartient au langage reconnu par une grammaire hors contexte. Il ne faut cependant pas prendre cette intuition trop au sérieux car, comme nous allons le voir, il n'est pas nécessaire, pour qu'un arbre puisse être utilisé par `Nessie`, qu'il rende compte aussi fidèlement de la syntaxe du texte dont on cherche à construire le sens.

Par ailleurs, cette structure d'arbre ne suffit pas, à elle seule, à spécifier comment les représentations des sous-syntagmes d'un syntagme doivent être combinées pour obtenir la sémantique de ce dernier. Il convient donc, d'une façon ou d'une autre, de spécifier pour chaque nœud de l'arbre comment sa représentation sémantique peut être obtenue à partir de celles de ses fils. Le langage que nous utilisons pour indiquer les règles de combinaison à utiliser dans les nœuds n'est autre que le λ-calcul simplement typé lui-même. Plus précisément, nous allons demander que chaque nœud de l'arbre fourni en entrée à `Nessie` contienne un λ-terme qui est à même de construire la représentation sémantique du nœud complet lorsqu'il reçoit comme arguments les représentations des sous-arbres de ce nœud. C'est pour cette raison que l'implantation des arbres, tout comme celle des lexiques, dépend d'une implantation des termes. Fidèles à la démarche que nous avions adoptée pour les lexiques, nous allons donc donner à la fois une spécification abstraite des arbres et un foncteur construisant une implantation de cette spécification à partir d'une implantation des termes. Cependant, contrairement aux lexiques, les arbres sont implantés de sorte que la spécification exporte le type des arbres, ce qui ne permet pas plusieurs implantations

de ce type. Il nous a en effet semblé préférable que la définition du type des arbres soit visible de l'extérieur, plutôt que cachée. Voici donc la spécification des arbres ainsi que le foncteur l'implantant à partir d'un module sur les termes :

```
module type TREE = sig
  type term
  type t =
    | Leaf of entry
    | Unary of string * t
    | Binary of string * t * t
    | Nary of string * term * t list
end

module Make (Term : TERM) : TREE = struct
  type term = Term.t
  type t =
    | Leaf of entry
    | Unary of string * t
    | Binary of string * t * t
    | Nary of string * term * t list
end
```

Les extraits de code que nous venons de montrer appellent plusieurs explications. En premier lieu, rappelons que le type `entry` est celui qui a été défini à la sous-section précédente et qui regroupe un nom de lemme et un nom de famille. Dans le code réel de `Nessie`, ce type n'est pas défini dans le module de lexiques, mais dans un module distinct. C'est pour cette raison que l'implantation des arbres ne dépend pas de celle des lexiques. En second lieu, remarquons que tous les constructeurs utilisés dans le code ci-dessus hormis ceux pour les feuilles prennent comme premier argument une chaîne de caractère. Cette chaîne de caractère n'est pas utilisée pour la construction de représentations sémantiques. Elle n'a qu'un but informatif et l'utilisateur est libre de s'en servir comme il l'entend. Pour notre part, nous utilisons ce champ principalement pour y stocker le nom de la catégorie syntaxique du nœud, comme on pourra le voir dans les exemples qui vont suivre.

Enfin, nous avions annoncé des arbres dont chaque nœud devrait contenir un terme indiquant comment combiner les représentations sémantiques des sous-arbres pour construire celle de l'arbre tout entier. Pourtant, dans le code ci-dessus, les constructeurs `Unary` et `Binary` ne prennent pas de termes en arguments. Ceci est lié au fait que nous avons choisi de permettre la construction d'arbres où les règles de construction des représentations sémantiques sont implicites. C'est le cas pour les arbres unaires et binaires qui apparaissent ci-dessus. Ainsi, la représentation d'un arbre unaire est celle de son seul sous-arbre. Ceci pourrait être exprimé à l'aide d'un arbre n-aire à un seul fils et contenant un λ-terme décrivant la fonction identité. De la même façon, la règle de construction de représentations sémantiques implicitement associée aux arbres binaires applique la représentation du sous-arbre gauche à celle du sous-arbre droit. Ceci est équivalent à un arbre n-aire contenant deux sous-arbres et dont le terme est une fonction à deux arguments qui applique son premier argument à son second argument.

On le voit, les arbres unaires et binaires n'augmentent en rien l'expressivité du langage d'arbres, puisque tout ce qu'ils expriment pourrait être exprimé en termes d'arbres n-aires. Nous

les utilisons cependant, d'abord pour une raison historique – ces arbres étaient les seuls arbres utilisés par les premières versions de `Nessie`–, ensuite parce que, dans la pratique, ils suffisent bien souvent, dans la mesure où il est fréquent qu'un syntagme soit construit à partir d'un ou de deux sous-syntagmes, sa représentation sémantique pouvant quant à elle être construite simplement par application d'une représentation sémantique à une autre, l'ordre des syntagmes dans le texte n'indiquant pas nécessairement quelle représentation utiliser en tant que foncteur et laquelle passer en argument. Ceci signifie que l'arbre utilisé pour calculer la représentation sémantique d'un texte ne reflète pas nécessairement de façon précise sa structure syntaxique. En particulier, il n'est pas nécessaire que l'ordre des nœuds reflète l'ordre des syntagmes qu'ils représentent. Toute liberté est donc laissée à l'étape d'analyse syntaxique quant à l'arbre qu'elle produit.

Ceci constitue une première nuance à apporter à ce qui a été affirmé au début de cette section, à savoir que l'arbre donné en entrée à `Nessie` est un arbre syntaxique du texte d'entrée. Une deuxième nuance à apporter concerne les feuilles de cet arbre. Au début de cette sous-section, nous avions affirmé que, en première approximation, on pouvait considérer que chaque feuille de l'arbre correspond à un mot. Ceci n'est qu'une approximation, dans le sens où rien n'empêche d'introduire dans l'arbre des feuilles qui ne correspondent à aucun mot. Par exemple, nous avons mentionné à la section précédente le fait que la sémantique des verbes conjugués peut être obtenue en appliquant une fonction conjuguant le verbe à une représentation sémantique du verbe à l'infinitif. Ceci peut être modélisé en définissant un ensemble de fonctions prenant en entrée une représentation sémantique à l'infinitif et renvoyant une représentation conjuguée. L'arbre construit pour un verbe conjugué peut alors être non une feuille, mais plutôt un arbre binaire dont la première feuille contient la fonction de conjugaison et dont la deuxième feuille contient le verbe à l'infinitif. Un exemple de tels arbres sera présenté en détail au chapitre 4.

Pour en revenir à l'expressivité du langage d'arbres, nous avons fait remarquer précédemment que les arbres unaires et binaires pouvaient être exprimés à l'aide d'arbres n-aires et que, par conséquent, ils n'ajoutaient rien à l'expressivité du langage. Autrement dit, nous sommes en train d'affirmer que les arbres n-aires sont plus expressifs que les arbres unaires et binaires, résultat qui n'est pas très surprenant. On est alors naturellement conduit à se demander si les arbres n-aires sont *strictement* plus expressifs que les arbres unaires et binaires, ce qui revient à se demander s'il est possible de simuler des arbres n-aires à l'aide d'arbres unaires et binaires. Comme nous le verrons au chapitre 7, sous certaines conditions on peut répondre par l'affirmative à cette question et montrer que les arbres n-aires peuvent être simulés à l'aide d'arbres binaires. Le fait de disposer de plusieurs types d'arbres n'est donc pas motivé par la recherche d'expressivité du langage, mais plutôt par le souci d'offrir un langage aussi commode et facile d'utilisation que possible en pratique.

Un autre questionnement en relation avec l'expressivité du langage d'arbres concerne les λ-termes autorisés à apparaître dans les nœuds des arbres n-aires. Jusqu'à présent, nous avons été assez flous à ce sujet, laissant entendre que n'importe quel terme pouvait apparaître dans un nœud n-aire. La preuve d'équivalence entre `Nessie` et une grammaire catégorielle abstraite d'ordre 2, que nous donnerons au chapitre 7 nous a cependant amenés à nuancer quelque peu ce point de vue. En effet, comme nous le verrons au chapitre 7, si n'importe quel terme peut apparaître dans un nœud n-aire, alors cela signifie que la construction sémantique peut faire appel à des règles n'apparaissant pas dans le lexique, ce qui conduit à perdre l'équivalence

avec une grammaire catégorielle abstraite d'ordre 2. En revanche, si les seuls termes autorisés à apparaître dans les nœuds n-aires sont des constantes déclarées ou définies dans le lexique, alors Nessie est bien équivalent à une grammaire catégorielle abstraite d'ordre 2.

Nous avons néanmoins fait le choix de continuer à accepter – nous sommes tentés d'écrire « de tolérer »– des termes arbitraires dans les nœuds n-aires, principalement pour des raisons de compatibilité ascendante. Il convient cependant de remarquer que, même en pratique, l'utilisation de termes arbitraires dans les nœuds n-aires n'est pas recommandée. En effet, comme nous le verrons au chapitre 6, une telle pratique conduit à produire des arbres qui ne pourront être utilisés qu'avec certains lexiques, alors que si l'on se limite à des constantes, il devient possible d'utiliser un même arbre avec des lexiques différents, chacun d'eux ayant alors toute liberté pour associer à ces constantes la sémantique appropriée.

Pour clore cette sous-section consacrée aux arbres, il nous reste à dire un mot de leur syntaxe concrète. Comme Nessie a été principalement utilisé en conjonction avec des DCGs, nous avons fait le choix de représenter les arbres par des termes Prolog qui peuvent être construits facilement par les DCGs lors de l'analyse syntaxique des textes dont on cherche la représentation sémantique. Ainsi, une feuille étiquetée par un lemme `l` et une famille `f` sera représentée par le terme Prolog `leaf(l,f)`. Par exemple, une feuille représentant le nom propre *John* pourrait s'écrire `leaf(john,pn)`, tandis qu'une feuille représentant le verbe intransitif *walk* pourrait s'écrire `leaf(walk,iv)`. De la même façon :

- `unary(c,t)` représente un arbre unaire de catégorie syntaxique `c` ayant comme unique sous-arbre `t` ;
- `binary(c,t1,t2)` représente un arbre binaire de catégorie syntaxique `c` ayant pour sous-arbre gauche `t1` et pour sous-arbre droit `t2` ;
- `nary(c,m,t1,...,tn)` représente un arbre n-aire de catégorie syntaxique `c` dont la représentation sémantique est construite en appliquant `m` aux représentations sémantiques des n sous-arbres `t1`, ..., `tn`.

Ainsi, si l'on souhaite calculer à l'aide de Nessie le sens de la phrase « A man walks. », on peut le faire à l'aide de l'arbre suivant :

```
binary(s,
  binary(np,leaf(indef,det),leaf(man,noun)),
  unary(vp,leaf(walk,iv))
)
```

Cet arbre met bien en évidence la structure syntaxique de la phrase, constituée d'un groupe nominal suivi d'un groupe verbal. Le groupe nominal consiste en un déterminant indéfini suivi d'un substantif, tandis que le groupe verbal consiste en un verbe intransitif. Remarquons que cet arbre ne contient pas d'arbres n-aires. Des exemples utilisant de tels arbres seront donnés au chapitre 3.

2.2.4 Algorithme de construction sémantique

Dans cette section, nous allons présenter en détail l'algorithme utilisé pour construire des représentations sémantiques et sur lequel la section précédente a pu donner quelques intuitions. En réalité, il s'agit ici de calculer deux objets : d'une part la représentation sémantique associée

à un arbre et, d'autre part, un environnement de typage dans lequel le type de cette représentation pourra être calculé. Nous allons commencer par présenter une première version simple de l'algorithme de construction sémantique. Nous montrerons ensuite en quoi cet algorithme est limité, ce qui nous conduira à proposer un second algorithme de construction sémantique qui nécessite quelques modifications des lexiques et des arbres introduits précédemment.

Avant de présenter les algorithmes, signalons que tous deux seront mis en œuvre au sein d'un foncteur paramétré par 3 modules : un module `Term` implantant les termes, un module `Lexicon` implantant les lexiques et un module `Tree` implantant les arbres.

Une première version de l'algorithme de construction sémantique

Pour cette première version de l'algorithme de construction sémantique, nous n'avons à produire que la représentation sémantique associée à l'arbre donné en entrée. En effet, l'environnement de typage permettant de typer cette représentation nous est donné par la fonction `environment_of_lexicon` du module `Lexicon` qui construit un environnement contenant tous les types et toutes les constantes logiques déclarées dans un lexique, que celles-ci aient été déclarées directement ou qu'elles proviennent de déclarations de lemmes. Voici donc le code de la fonction `compute` permettant de construire des représentations sémantiques :

```
let rec compute lexicon tree = function
  | Leaf entry ->
    Lexicon.instanciate entry
  | UnaryNode (syntacticCategory,subtree) ->
    compute lexicon subtree
  | BinaryNode (syntacticCategory,subtree1,subtree2) ->
    let t1 = compute lex subtree1 in
    let t2 = compute lex subtree2 in
    makeApplication t1 t2
  | NaryNode (syntacticTree, term, subtrees) ->
    begin
      match subtrees with
        | [] -> term
        | subtree1::lst ->
          let (var, term') =
            removeFirstAbstraction term
          in let term'' =
            term'[var:=(compute lex subtree1)]
          in compute
            lex
            (NaryNode syntacticCategory, term'', lst)
    end
```

Comme on peut le constater, la fonction `compute` est définie récursivement sur la structure de l'arbre. Les premiers cas de cette définition portant sur les feuilles, les arbres unaires et les arbres binaires ne posent à notre avis aucun problème de lecture. Ils sont une implantation directe des intuitions déjà mentionnées dans la section précédente. Nous supposons simplement que nous disposons d'une fonction `makeApplication` qui, étant donné deux termes `t1` et `t2` construit

le λ-terme `t1 t2`. La dernière clause de la définition de `compute`, en revanche, nécessite de plus amples explications. En effet, nous avions laissé entendre que la représentation sémantique d'un arbre n-aire serait obtenue en appliquant le terme contenu dans le nœud aux représentations sémantiques des n sous-arbres, et pourtant, à première vue, ce n'est pas ce que fait `compute`. Analysons cependant le code qui vient d'être donné pour les arbres n-aires. Comme le terme qu'ils contiennent doit recevoir n arguments, nous pouvons raisonnablement supposer que ce terme commence par n abstractions, et qu'il est donc de la forme $\lambda X_1.\lambda X_2.\ldots.\lambda X_n.M$. Pour en revenir au code, il commence par tester si la liste de sous-arbres est vide, autrement dit si l'arbre n-aire ne contient aucun sous-arbre. Si c'est le cas, la valeur qui est renvoyée est celle du terme qu'il contient, ce qui semble être un choix plutôt raisonnable, puisque c'est la seule valeur dont nous disposons. Si la liste de sous-arbres n'est pas vide, alors elle contient un premier sous-arbre `subtree1`, suivi d'une liste de sous-arbres `lst`. Dans ce cas, on commence par extraire la première abstraction du terme contenu dans le sous-arbre. C'est ce que fait la fonction `removeFirstAbstraction`. Après l'appel de cette fonction et compte tenu de nos conventions, `var` va contenir X_1 et `term'` va contenir le terme d'origine privé de l'abstraction la plus externe, c'est-à-dire $\lambda X_2.\ldots.\lambda X_n.M$. Puis nous calculons `term''`, qui n'est autre que le terme précédent dans lequel toutes les occurrences de `var` (donc de X_1) sont remplacées par la représentation sémantique de `subtree1`. Le terme `term''` est donc un terme commençant par $n-1$ abstractions et prêt à recevoir en argument les $n-1$ sous-arbres de la liste `lst`. C'est pourquoi le calcul se termine en rappelant `compute` sur un arbre n-aire dont le terme est celui qui vient d'être calculé et dont les sous-arbres sont ceux de la liste `lst`.

En d'autres termes, l'algorithme que nous venons de présenter ne construit pas explicitement l'application du terme contenu dans le nœud aux représentations sémantiques des sous-arbres. Cette construction aurait certes donné lieu à un code plus simple que celui que nous venons de présenter, mais elle aurait aussi conduit à une perte d'efficacité. En effet, si l'on se contente de construire un terme en appliquant le terme contenu dans un nœud n-aire à ses n arguments, le résultat ainsi obtenu est un λ-terme qui comporte n β-radicaux qui ont été introduits par l'algorithme de construction sémantique et qu'il faudra par la suite réduire. La version de l'algorithme de construction sémantique que nous venons de proposer, quant à elle, produit pour les arbres n-aires des termes dans lesquels les β-radicaux qui auraient été introduits par une construction d'applications naïve sont déjà réduits, ce qui conduit au final à des termes avec moins de β-radicaux et donc plus proches de leur forme normale en termes du nombre de réductions restant à effectuer.

Cette façon de construire la représentation des arbres n-aires en procédant tout de suite aux β-réductions offre un autre avantage par rapport à la construction d'applications sans β-réduction. Pour comprendre cet avantage, revenons un instant sur les explications que nous avons données précédemment. Nous avons vu comment la première abstraction λX_1 du terme présent dans un nœud n-aire est isolée et comment la variable X_1 sur laquelle a lieu cette abstraction est remplacée par la représentation sémantique du premier sous-arbre. Donc, X_1 n'apparaîtra pas dans la représentation sémantique renvoyée par `compute` et il en est de même pour toutes les variables $X_2, \ldots, X_n$ utilisées dans les abstractions suivantes du terme. Donc, si l'on souhaite connaître le type de la représentation sémantique ainsi construite, on n'a pas besoin de connaître celui des variables $X_1, \ldots, X_n$, puisqu'aucune de ces variables ne figure dans la représentation produite. En revanche, si nous avions produit un terme de façon naïve, sans procéder d'emblée aux β-

réductions correspondant aux applications, les variables $X_1, \ldots, X_n$ auraient été présentes dans le terme renvoyé par `compute` et il n'aurait donc pas été possible de connaître le type de ce terme sans connaître celui des variables $X_1, \ldots, X_n$.

On l'aura compris, cette façon de construire les représentations sémantiques des arbres n-aires nous autorise à ne pas spécifier les types des variables apparaissant dans les termes contenus dans les nœuds n-aires et qui sont appelées à être remplacées par des représentations sémantiques de sous-arbres, dont le type est connu. Ceci est un gain important, car, bien souvent, les types de ces variables (c'est-à-dire les types des représentations sémantiques des sous-arbres) sont difficiles à trouver et à écrire, et en outre leur recherche ne présente, à notre avis, que peu d'intérêt.

Un deuxième algorithme de construction sémantique

L'algorithme que nous avons présenté précédemment permet de construire la représentation sémantique de textes complexes à partir de leur arbre d'analyse et d'un lexique associant une représentation sémantique aux mots. En particulier, si un texte comporte deux occurrences d'un même mot, les variables liées apparaissant dans la représentation sémantique de la seconde occurrence seront renommées, de sorte que la représentation de cette seconde occurrence sera α-équivalente à la représentation de la première, mais non identique. Nous pourrions donc nous contenter de cet algorithme, qui fonctionne pour toutes les théories sémantiques associant à différentes occurrences du même mot des représentations identiques ou α-équivalentes entre elles. Cependant, il s'avère que toutes les théories sémantiques n'ont pas cette propriété. En effet, certaines d'entre elles associent à deux occurrences distinctes d'un même symbole des représentations qui ne sont pas α-équivalentes. Dans l'encodage des DRS proposé par Reinhard Muskens dans (Muskens, 1996c), par exemple, les entités introduites par les déterminants sont représentées non pas par des variables liées, comme nous l'avons vu jusqu'à présent, mais par des constantes, une constante différente étant utilisée pour chaque occurrence d'un déterminant [7]. Ainsi, dans le texte « A man adores a woman », la première occurrence du déterminant indéfini pourrait être représentée dans la proposition de Muskens par le terme $\lambda P' P : \pi \rightarrow s \rightarrow s \rightarrow t.[u_1|]; [P'(u_1)]; [P(u_1)]$, tandis que la seconde occurrence pourrait être représentée par le terme $\lambda P' P : \pi \rightarrow s \rightarrow s \rightarrow t.[u_2|]; [P'(u_2)]; [P(u_2)]$, u_1 et u_2 étant des *constantes* supposées *distinctes* du langage logique. La signification précise de ces représentations sera donnée au chapitre 5. Pour l'instant, il suffit de remarquer que ces deux représentations, bien que représentant toutes deux un même lemme d'une même famille (le déterminant indéfini) ne sont pas α-équivalentes.

Or, en l'état actuel des choses, `Nessie` ne permet pas de construire des représentations non α-équivalentes pour deux occurrences distinctes d'un même lexème. L'algorithme de construction sémantique vu précédemment le prouve : si un même lexème apparaît plusieurs fois dans un texte, le lexique sera consulté pour chaque occurrence du lexème et, comme cette consultation se fera toujours avec le même lemme et la même famille, on aboutira toujours, nécessairement, à la même représentation sémantique.

[7] L'encodage de la DRT dans TYn proposé par Muskens sera décrit en détail au chapitre 5, tandis que son implantation à l'aide de `Nessie` sera présentée au chapitre 6, section 6.2. En outre, le travail présenté dans cette sous-section a également été présenté dans (Blackburn et Hinderer, 2007b).

Nous devons donc trouver un moyen permettant à Nessie de construire, au besoin, des représentations non α-équivalentes pour différentes occurrences d'un même symbole. Cela suppose à la fois de pouvoir distinguer deux occurrences d'un même symbole l'une de l'autre dans un arbre syntaxique, et de pouvoir utiliser les informations propres à chaque occurrence pour construire une représentation qui lui est propre.

Pour résoudre ce problème, nous allons introduire ce que nous appellerons des *arguments d'occurrences*. Il s'agit de paramètres additionnels qui peuvent être inclus dans les feuilles de l'arbre syntaxique et qui seront traités comme des constantes, ces constantes pouvant apparaître dans le terme construit par instanciation d'une feuille. Ainsi, dans l'exemple précédent, la première occurrence du déterminant indéfini pourrait être représentée par la feuille `leaf(indef,det,u1)`, tandis que la deuxième occurrence pourrait être représentée par la feuille `leaf(indef,det,u2)`. Bien sûr, nous devons aussi augmenter la syntaxe des termes, de sorte que les représentations sémantiques construites dans les lexiques puissent faire référence aux arguments d'occurrence et ainsi construire des représentations prenant ces arguments d'occurrence en compte. L'extension que nous proposons consiste à autoriser des termes de la forme `$i`, où `i` est un numéro d'argument. Avec une telle notation, nous pouvons stocker dans le lexique une représentation des déterminants qui donnera lieu, pour chaque instanciation, à une représentation sémantique différente. Une telle entrée pourrait être définie de la façon suivante :

```
family det;
lemma indef {
  family = det;
  term = lam P', P : pi -> drs.
    ( [$1|];P'($1);P($1) )
};
```

Voici deux feuilles accompagnées par les représentations qui leurs seraient associées par Nessie à partir de l'entrée de lexique précédente :

- `leaf(indef,det,u1)` : $\lambda P' P : \pi \rightarrow s \rightarrow s \rightarrow t.[u_1|]; P'(u_1); P(u_1)$;
- `leaf(indef,det,u2)` : $\lambda P' P : \pi \rightarrow s \rightarrow s \rightarrow t.[u_2|]; P'(u_2); P(u_2)$.

Pour bien comprendre comment fonctionnent ces arguments d'occurrence, il peut être bénéfique de les appréhender comme les arguments d'une fonction. Les arguments de la forme `$i` sont comparables aux paramètres formels d'une fonction, c'est-à-dire à ceux apparaissant dans sa définition. Les arguments comme `u1` et `u2` peuvent quant à eux être vus comme des paramètres effectifs, c'est-à-dire ceux passés à la fonction pendant « l'exécution » qui n'est autre que le processus d'instanciation dans ce cas.

Il reste à voir comment spécifier le type des arguments d'occurrence et comment garder la trace de ce type. En ce qui concerne la spécification des types des arguments d'occurrence, la métaphore les comparant à des arguments de fonctions traditionnelles continue de s'appliquer et nous incite à envisager l'enrichissement des lexiques de sorte à pouvoir spécifier le type de chaque argument d'occurrence. Cependant, dans la mesure où nous nous trouvons dans le cadre du λ-calcul, il convient de remarquer que ces arguments ne sont pas des variables, mais bien des constantes. Nous allons donc procéder à un dernier enrichissement de la représentation des lexiques pour leur permettre de gérer les arguments d'occurrences. Il paraît naturel de partager les arguments d'occurrence de la même façon que les représentations sémantiques. Plus

précisément, cela signifie que pour une famille ouverte, où toute l'information sémantique est centralisée dans la famille, les types des arguments d'occurrence doivent être déclarés dans la déclaration de la famille, et non dans celle des lemmes. Dans les familles fermées en revanche, où ce sont les définitions de lemmes qui contiennent l'essentiel de l'information sémantique, nous allons permettre à chaque lemme de déclarer ou non des arguments d'occurrence. Compte tenu de ces précisions, voici la structure de lexique que nous obtenons :

```
type openFamily = {
  constants : (string * Type.t) list;
  lemmaType : Type.t;
  pattern : Term.t;
  lemmas : StringSet.t
}

type closedLemma = {
  constants : (string * Type.t) list;
  term : term
}

type closedFamily = closedLemma StringMap.t

type family =
  | Open of openFamily
  | Closed of closedFamily

type t = {
  environment : Term.environment
  entries : family StringMap.t
}
```

Cette dernière extension consiste essentiellement à ajouter un champ `constants` aux familles ouvertes et aux lemmes fermés. Ce champ est une liste de noms de constantes accompagnés de leurs types. Les noms des constantes peuvent *a priori* prendre des valeurs arbitraires, mais seuls des noms de la forme `$i` ont réellement du sens. En effet, ces noms seront les seuls à être remplacés lors de chaque instanciation par les arguments d'occurrences apparaissant dans l'arbre syntaxique, ce qui permet effectivement de distinguer des occurrences. Si un autre nom était utilisé, la même constante serait ajoutée à l'environnement à chaque fois qu'une occurrence du lemme concerné serait rencontrée, ce qui n'a pas grand intérêt.

Il nous reste à voir comment garder trace des noms et types des constantes ajoutées lors des diverses instanciations. Pour cette dernière tâche, ce sont bien sûr les environnements de typage qui vont nous être utiles. Cependant, pour le moment l'algorithme de construction sémantique dont nous disposons ne permet pas l'ajout de nouvelles constantes à l'environnement de typage pendant le parcours de l'arbre d'analyse. Et c'est précisément en cela que notre premier algorithme de construction sémantique est insuffisant, c'est donc cela qu'il convient de modifier. En d'autres termes, nous allons modifier l'algorithme de construction sémantique de sorte que non seulement de nouvelles constantes puissent être ajoutées à l'environnement de typage pendant le parcours e l'arbre syntaxique, mais que, de surcroît, ces constantes puissent être réutilisées

plus loin dans l'arbre. La représentation sémantique et l'environnement à utiliser pour la typer ne sont donc plus produits séparément comme c'était le cas dans notre première approche, mais conjointement. En outre, pour que l'environnement puisse être modifié pendant le parcours de l'arbre d'analyse, il est aussi nécessaire de le fournir en entrée à l'algorithme de construction sémantique.

Concrètement, il y a deux modifications à apporter à l'algorithme de construction sémantique vu précédemment. Dans un premier temps, il nous faut modifier la fonction `instanciate` introduite en section 2.2.2 pour qu'elle prenne un environnement en entrée et le mette à jour en y ajoutant les éventuelles déclarations de constantes correspondant aux arguments d'occurrence. Dans un deuxième temps, nous aurons à modifier la fonction `compute` pour qu'elle puisse prendre un environnement en entrée et propager les environnements mis à jour dans les appels récursifs.

Nous remplaçons la spécification de `instanciate` par la spécification suivante :

```
val instanciate : environment -> entry -> Term.t * environment
```

Et voici comment nous adaptons le code original de `instanciate` à cette nouvelle spécification. Notons que, compte tenu de la complexité du code réel, nous ne donnons ici qu'une version en pseudo-code `Caml` de cette fonction :

```
let aux lex env ent = match StringMap.find e.family lex.entries with
  | Open f ->
    ( f.constants,
      Term.constSubst "_" e.lemma f.pattern,
      Term.addConstant e.lemma env
    )
  | Closed lemmas ->
    let l = StringMap.find e.lemma lemmas in
    ( l.constants,
      l.term
      env
    )

let instanciate lex env ent =
  let (declaredConstants, semantics, newEnv) = aux lex env ent in
  let usedConstants = Term.collectConstants semantics in
  let sigma = expandConstants termConstants in
  let finalEnv = addConstants (sigma declaredConstnats) newEnv in
  let finalSemancics = sigma semantics in
  (finalSemantics, finalEnv)
```

Pour bien comprendre le fonctionnement de cette version modifiée de `instanciate`, nous allons montrer ce que fait cette fonction sur l'exemple des déterminants que nous avions introduit précédemment. Voici une déclaration plausible des déterminants dans les lexiques de `Nessie` :

```
family det;
lemma indef {
  family = det;
```

```
  const $1 : pi;
  term = lam P', P : pi -> drs.
    ( [$1|];P'($1);P($1) )
};
```

Notons que la seule modification entre l'entrée de lexique que nous avions donnée précédemment et celle-ci est l'ajout dans cette dernière de la ligne `const $1 : pi;` qui demande qu'à chaque instanciation le premier argument d'occurrence de la feuille instanciée soit ajouté à l'environnement en tant que constante de type π. Nous allons voir comment se comporte la fonction `instanciate` sur les feuilles `leaf(indef,det,u1)` et `leaf(indef,det,u2)`.

On commence par appeler la fonction `aux` qui effectue la consultation du lexique proprement dite. Cette fonction a pour résultat un triplet comportant l'ensemble des constantes à déclarer pour le lemme considéré, sa représentation sémantique et un environnement éventuellement mis à jour. Si le lemme à instancier appartient à une famille fermée, l'environnement renvoyé est l'environnement initial. En revanche, si le lemme appartient à une famille ouverte, l'environnement est mis à jour par l'ajout d'une constante correspondant à ce lemme. Nous reviendrons sur ce point un peu plus tard.

Pour les déterminants, par exemple, l'appel à `aux` renvoie les résultats suivants :

1. Liste de constantes à déclarer : `[($1,pi)]`;
2. Représentation sémantique :
   `lam P', P : pi -> drs. ( [$1|];P'($1);P($1) );`
3. Environnement : l'environnement initial puisque les déterminants ont été définis à l'aide d'une famille fermée.

On calcule ensuite l'ensemble des constantes présentes dans la représentation sémantique, ensemble qui se réduit dans le cas des déterminants à la seule constante `$1`.

Le calcul se poursuit par la construction d'une substitution qui associe aux noms de la forme `$i` les noms des arguments d'occurrence effectifs. La substitution qui est calculée pour la feuille `leaf(indef,u1)` est une substitution qui à `$1` associe `u1`. De même, dans le cas de la feuille `leaf(indef,det,u2)`, la substitution calculée est celle qui à `$1` associe `u2`. Comme on peut le constater, c'est cette étape qui est la clé de voûte de cette version révisée de l'algorithme de construction sémantique. Le calcul se poursuit en appliquant cette substitution à la liste des constantes à déclarer telle qu'elle a été retournée par `aux`. Dans le cas de la première nous obtenons la liste de constantes `[(u1,pi)]`, tandis que pour la deuxième feuille on obtient `[(u2,pi)]`. Ce sont ces listes qui sont utilisées pour mettre à jour l'environnement initial et obtenir celui qui est finalement retourné. Enfin, on applique la substitution précédente à la représentation sémantique retournée par `aux`, ce qui donne dans le cas de la première feuille le terme `lam P', P : pi -> drs. ( [u1|];P'(u1);P(u1) )` et dans le cas de la seconde feuille le terme `lam P', P : pi -> drs. ( [u2|];P'(u2);P(u2) )`, termes qui sont renvoyés par la fonction, accompagnés des environnements mis à jour.

Remarquons qu'il n'est pas nécessaire qu'une feuille déclare les arguments d'occurrences utilisés par sa représentation sémantique. Il est en effet possible que ceux-ci aient été ajoutés à l'environnement par une feuille rencontrée auparavant dans l'arbre d'analyse syntaxique. Comme nous le verrons aux chapitres 5 et 6, ceci est par exemple le cas des pronoms dans l'encodage

de la DRT proposé par Muskens. En effet, la représentation sémantique des pronoms dans cet encodage utilise les constantes ajoutées à l'environnement par les entités auxquelles les pronoms considérés font référence. Comme nous le verrons, notre algorithme d'instanciation fonctionne également dans cette situation.

Il nous reste à présenter la spécification modifiée de `compute` ainsi que sa nouvelle version prenant en compte des environnements susceptibles d'être enrichis en cours de construction sémantique. La spécification que nous utilisons est la suivante :

```
val compute :
  Lexicon.t -> Term.environment -> Tree.t ->
  Term.t * Term.environment
```

Quant à l'algorithme lui-même, il s'obtient en modifiant légèrement le précédent, de sorte à récupérer les environnements renvoyés par les appels récursifs de `compute` et à les passer aux appels suivants. Voici cet algorithme modifié :

```
let rec compute lexicon env tree = match tree with
  | Leaf entry -> Lexicon.instanciate entry env
  | UnaryNode (syntacticCategory,subtree) ->
    compute lexicon env subtree
  | BinaryNode (syntacticCategory,subtree1,subtree2) ->
    let (t1,newEnv1) = compute lex subtree1 in
    let (t2,newEnv2) = compute lex newEnv1 subtree2 in
    (makeApplication t1 t2, newEnv2)
  | NaryNode (syntacticTree, term, subtrees) ->
    begin
      match subtrees with
        | [] -> (term,env)
        | subtree1::lst ->
          let (var, term') = removeFirstAbstraction term in
          let (term1,env1) = compute lex env subree1 in
          let term'' = term'[var:=term1] in
          compute
            lex
            env1
            (NaryNode syntacticCategory, term'', lst)
    end
```

Comme on peut le constater, cette fonction est très proche de la précédente et, conformément à ce que nous avions annoncé, le seul changement concerne la propagation des environnements.

Nous avons donc obtenu un algorithme de construction sémantique capable de prendre en compte des environnements qui s'enrichissent au fur et à mesure que l'arbre d'analyse est parcouru et qui ne dépendent donc plus seulement du lexique, comme cela était le cas dans la précédente version de l'algorithme. Ceci permet d'associer à différentes occurrences d'un même lemme des représentations non α-équivalentes (à condition que l'arbre d'analyse contienne des arguments d'occurrences bien choisis), ce que ne permettait pas la première version de l'algorithme. En outre, cette seconde version de l'algorithme de construction sémantique a, comme nous avons commencé à le laisser entendre, un autre avantage sur la première. Nous avons

en effet indiqué que la fonction `instanciate` ajoutait à l'environnement les constantes associées aux lemmes ouverts instanciés. Or, quand nous avons présenté le premier algorithme de construction sémantique, nous avons indiqué que l'environnement de typage était calculé une fois pour toutes en fonction du lexique. L'environnement associé au lexique contenait alors des constantes pour *tous* les lemmes ouverts définis par le lexique, que ceux-ci apparaissent dans un arbre syntaxique ou non. Avec ce deuxième algorithme de construction sémantique qui ajoute les constantes correspondant aux lemmes ouverts à l'environnement au fur et à mesure que ceux-ci sont rencontrés pendant le parcours d'un arbre syntaxique donné, il n'est plus nécessaire de les inclure dans l'environnement initialement associé au lexique.

Dans le cas d'un lexique de taille conséquente, ce changement constitue une amélioration non négligeable, puisque les environnements qui sont construits pour un arbre syntaxique donné ne contiennent que les symboles apparaissant effectivement dans la représentation sémantique produite, alors qu'avec la première version de l'algorithme les environnements produits avaient une taille linéaire en le nombre de lemmes déclarés dans le lexique.

2.2.5 Fonctionnement

Jusqu'ici, nous avons présenté les algorithmes constituant le cœur de `Nessie` en adoptant pour ce faire un point de vue plutôt technique. Dans cette dernière section, nous aimerions donner quelques informations concrètes sur la façon d'utiliser `Nessie`.

Initialement envisagé comme une bibliothèque pouvant être utilisée au sein de projets plus ambitieux, `Nessie` a évolué pour devenir un programme à part entière. Cette évolution permet une utilisation des fonctionnalités de `Nessie` depuis des programmes écrits en n'importe quel langage et pas seulement en `OCaml`. Cependant, les fonctionnalités de `Nessie` étant toujours réunies dans une bibliothèque, rien n'empêche de réutiliser celle-ci depuis d'autres programmes `Caml` si on le souhaite.

Le programme `Nessie` peut être utilisé soit pour vérifier la bonne formation d'un lexique, soit pour calculer la représentation sémantique associée à un arbre syntaxique donné, ce qui est son utilisation principale. La vérification d'un lexique consiste à vérifier sa syntaxe et aussi que toutes les représentations sémantiques qui y sont définies peuvent être typées. Par exemple, si nous plaçons dans un fichier `demo.lex` le petit lexique que nous avions donné à titre d'exemple à la fin de la section 2.2.2, nous pouvons en vérifier la bonne formation grâce à la commande :

```
$ nessie check demo.lex
```

qui produit le résultat suivant :

```
Family det
  indef : lam P, lam Q, some x, ((P app x) imp (Q app x))
Family iv: dance walk
Family noun: man woman
```

Une fois qu'un lexique a pu être vérifié, nous pouvons l'utiliser pour calculer les représentations sémantiques associées à des arbres syntaxiques. Ceux-ci sont fournis sur l'entrée standard de `Nessie`. Nous pouvons par exemple calculer la représentation sémantique de l'arbre donné en section 2.2.3 :

```
$ nessie compute demo.lex
binary(s,
  binary(np,leaf(indef,det),leaf(man,noun)),
  unary(vp,leaf(walk,iv))
)
```

Lorsque Nessie est ainsi appelé et reçoit ces données, il commence par calculer la représentation sémantique et l'environnement de typage associés à l'arbre en utilisant pour cela l'algorithme qui a été décrit précédemment. Une fois cette représentation calculée, la première opération effectuée par Nessie est une vérification de son type. Une erreur de typage entraînant l'arrêt immédiat du programme. Si la vérification de type réussit, les constantes qui ont été définies dans le lexique à l'aide de l'opérateur := sont remplacées par leur définition si cela a été demandé (un exemple utilisant cette fonctionnalité est fourni au chapitre 6, section 2). Puis la représentation sémantique ainsi obtenue est normalisée par β-réduction, et c'est cette valeur réduite qui est affichée. Pour l'exemple précédent, nous obtenons comme résultat :

```
some x, ((man app x) imp (walk app x))
```

Ici la représentation sémantique est affichée dans une forme qui se veut lisible par un utilisateur humain. Il est également possible de l'afficher de sorte qu'elle soit lisible par Prolog en procédant comme suit :

```
$ nessie compute demo.lex -o prolog
binary(s,
  binary(np,leaf(indef,det),leaf(man,noun)),
  unary(vp,leaf(walk,iv))
)
some(X,imp(man(X),walk(X))).
```

Il est également possible d'afficher des informations supplémentaires telles que le type de la représentation sémantique, ainsi que la représentation à divers stades du calcul (représentation « brute » telle qu'elle est construite par l'algorithme de la section précédente, représentation après dépliage des constantes et avant β-réduction). Nous aurons l'occasion d'exploiter cette dernière fonctionnalité au chapitre suivant, lorsque nous voudrons comparer les résultats produits par Nessie à ceux produis par Curt, qui a été brièvement introduit au chapitre précédent.

2.2.6 Conclusion

Dans ce chapitre, nous avons présenté Nessie, un outil permettant de construire automatiquement la représentation sémantique de textes à partir des représentations des mots et d'arbres permettant de les combiner. Comme nous l'avons vu à la section 1.1, l'idée d'utiliser des arbres abstraits pour guider la construction sémantique est due à Montague et n'est donc pas nouvelle. Or, les formalismes habituels (et en tout cas ceux qui utilisent des constituants) font partie généralement de la classe des langages faiblement sensibles au contexte (middly context-sensitive languages), ce qui signifie en particulier qu'ils on t une structure de dérivation qui est un arbre de grammaire hors-contexte. C'est donc une structure privilégiée pour la construction sémantique. Cependant, si cette idée structure semble largement admise par la communauté, la façon précise

dont il convient de combiner les représentations sémantiques des sous-arbres pour obtenir celle d'un arbre complet n'a pas, quant à elle, fait l'objet d'un consensus jusqu'à présent. Comme cela est expliqué dans (Heim et Kratzer, 1998), deux approches ont été proposées pour résoudre ce problème. La première consiste à associer à chaque nœud de l'arbre abstrait une règle spécifiant comment combiner les représentations de ses sous-arbres. La seconde approche préconise quant à elle que ce soient les types qui guident la combinaison des représentations sémantiques, avec pour avantage qu'il n'est plus nécessaire d'associer à chaque nœud une règle de construction explicite. Pour notre part, nous trouvons ces deux approches séduisantes et, surtout, nous ne les concevons pas comme antinomiques. C'est ainsi que, dans `Nessie`, nous avons souhaité concilier les deux approches. La première est mise en œuvre grâce à l'utilisation d'arbres n-aires auxquels sont associés des λ-termes spécifiant comment combiner les représentations sémantiques des sous-arbres. La seconde est partiellement mise en œuvre par l'utilisation des arbres binaires. En effet, lorsque ceux-ci sont utilisés, il n'est pas nécessaire de fournir une règle de combinaison explicite. En outre, nous aimerions faire remarquer qu'il serait facile, compte tenu de l'architecture qui est celle de `Nessie`, d'ajouter d'autres règles permettant, à partir des types, d'inférer comment une représentation sémantique doit être construite.

Une telle extension de `Nessie` n'est cependant pas urgente. En effet, comme nous allons le montrer dans les chapitres suivants, `Nessie`, dans son état actuel, peut déjà être utilisé en sémantique computationnelle. Ainsi, nous commencerons par en montrer le bon fonctionnement dans le chapitre suivant, avant de l'utiliser dans l'étude de phénomènes sémantiques plus complexes, à savoir la construction de représentations temporelles pour les verbes du polonais et la construction compositionnelle de la sémantique de discours comportant plusieurs phrases, et cela en utilisant deux formalismes bien distincts.

Chapitre 3

Intégration de Nessie à Curt

Au début du chapitre précédent, nous avons expliqué que `Curt` avait été conçu surtout à des fins pédagogiques et ne nous semblait pas très bien adapté à une utilisation en vue d'explorer la construction sémantique pour différentes théories. Nous avions cependant annoncé, en section 1.3, que nous n'abandonnerions pas pour autant `Curt` dans cette thèse et que, au contraire, une fois en possession d'un outil capable de construire des représentations sémantiques dans TYn à partir de lexiques et d'arbres, nous ferions en sorte d'intégrer cet outil à l'architecture de `Curt`, de sorte à pouvoir profiter de sa DCG et de sa composante inférentielle. `Nessie` ayant été présenté au chapitre précédent, il est donc temps de voir comment il peut être intégré à l'architecture de `Curt`, et c'est à cette tâche que nous allons nous consacrer maintenant.

L'objectif que nous poursuivons dans ce chapitre est double. Premièrement, nous souhaitons montrer que `Nessie` fonctionne bien, et c'est cette raison qui nous pousse à l'intégrer à `Curt`. Notre démarche pour atteindre ce premier objectif va en effet consister à montrer que les représentations sémantiques construites par `Nessie` sont les mêmes que celles construites par `Curt`, ce qui nous permettra de nous convaincre que `Nessie` fonctionne de façon satisfaisante. Deuxièmement, nous cherchons dans ce chapitre à poser des bases qui nous seront utiles dans la suite de cette thèse, et cela de deux façons. D'une part, nous voudrions, comme nous l'avons annoncé précédemment, être capables de construire pour une phrase reconnue par une grammaire des représentations sémantiques dans plusieurs formalismes différents, comme par exemple la DRT compositionnelle de (Muskens, 1996c) et le traitement compositionnel de la dynamicité de (de Groote, 2006). L'intégration de `Nessie` à `Curt` est un pré-requis indispensable à de telles expérimentations. Une fois celle-ci réalisée, il devrait en effet suffire d'écrire un lexique `Nessie` pour chaque théorie sémantique considérée, la construction sémantique pouvant alors être faite automatiquement (nous verrons au chapitre 6 qu'en réalité, écrire un lexique `Nessie` n'est pas suffisant, mais cela ne change rien au fait que le travail d'intégration de `Nessie` à l'architecture de `Curt` mené à bien dans ce chapitre est nécessaire). Les travaux de ce chapitre servent donc de base aux implantations de (Muskens, 1996c) et (de Groote, 2006) qui seront réalisées au chapitre 6. D'autre part, ce chapitre a une portée méthodologique : il montre comment modifier une grammaire existante de sorte qu'elle produise des arbres d'analyse utilisables par `Nessie`. Les enseignements tirés de ce travail d'adaptation faciliteront grandement le développement, au chapitre suivant, d'une DCG reconnaissant des phrases polonaises simples, ce développement permettant quant à lui une première étude des conséquences du passage à la logique d'ordre supérieur sur l'inférence.

Nous commençons par discuter, dans la section 3.1, des différentes possibilités disponibles pour tester `Nessie`. La section 3.2 détaille la mise en œuvre de la procédure de test choisie sur

les cas les plus simples. Nous nous intéressons ensuite à deux phénomènes sémantiques particuliers : les coordinations (section 3.3) et les questions 3.4, l'étude des coordinations conduisant à une première confrontation avec les limites expressives de la famille de logiques TYn.

3.1 Procédure de test

Nous l'avons vu au chapitre 2, `Nessie` est un programme complexe. Il est composé de plusieurs modules et fait intervenir des structures de données complexes. En outre, comme nous l'avons montré, le processus permettant de construire une représentation sémantique β-réduite à partir d'un lexique et d'un arbre est long et fait intervenir plusieurs étapes de calcul, dont certaines relativement raffinées.

Il paraît donc légitime, une fois un tel outil écrit et avant son utilisation dans des applications réalistes, de s'assurer que le fonctionnement de l'outil correspond bien à sa spécification telle que donnée au chapitre précédent. Ceci pose une question méthodologique : comment pouvons-nous être sûr que `Nessie` fait bien ce qui est attendu et qu'il peut être utilisé en toute confiance en construction sémantique ?

Deux pistes sont communément proposées pour répondre à cette question. Une première piste consiste à prouver la correction de `Nessie` à l'aide de logiciels d'aide à la preuve. Il s'agit dans ce cas de spécifier formellement chaque étape de calcul, de définir les fonctions réalisant le calcul et de prouver sur machine que les fonctions ainsi définies vérifient les propriétés attendues. Un outil d'extraction permet ensuite d'obtenir à partir des définitions de fonctions et des preuves associées du code certifié correct [1]. Une seconde possibilité consiste à vérifier les résultats produits par un programme. La manière de procéder à cette vérification dépend alors du programme que l'on étudie. Si le programme résout des équations, on peut par exemple vérifier que les solutions qu'il produit sont bien des solutions de l'équation initiale. Il est aussi possible de comparer les résultats fournis par le programme à tester à ceux fournis par d'autres programmes semblables existant depuis longtemps et dont on peut se convaincre qu'ils produisent les résultats attendus.

Bien sûr, la première méthode est plus puissante, dans la mesure où ce qui est prouvé est la correction de l'algorithme. Une fois la preuve faite, il n'est plus besoin de procéder à aucune vérification supplémentaire des résultats : ils sont corrects parce que l'algorithme qui les produit l'est. Cependant, étant donné les outils disponibles pour développer des programmes certifiés, prouver la correction d'un programme tel que `Nessie` serait une tâche extrêmement complexe compte tenu de sa taille.

Donc, bien que le travail de certification d'algorithmes soit en lui-même enrichissant et instructif (la formalisation oblige à comprendre les algorithmes et leurs preuves de correction avec une granularité bien plus fine que celle avec laquelle peuvent se faire les preuves sur le papier), nous y avons renoncé d'une part à cause de sa longueur, d'autre part parce qu'il nous aurait conduit, nous semble-t-il, bien loin des préoccupations linguistiques qui sont les nôtres dans cette thèse.

On l'aura compris, c'est donc la deuxième méthode, moins générique mais plus facile à mettre en œuvre sur un projet de ce type, que nous allons utiliser. Plus précisément, les tests vont

[1] Cette approche est explorée et mise en œuvre dans (Letouzey, 2004).

consister à comparer les résultats produits par Nessie à ceux d'un programme existant et plus éprouvé que lui, en l'occurrence Curt, qui a été introduit en section 1.3.

Le test que nous envisageons de mettre en place consiste à vérifier que, étant donné une phrase reconnue par Curt, il est possible d'en construire une représentation sémantique à l'aide de Nessie. En outre, il s'agit de vérifier que cette représentation sémantique produite par Nessie est équivalente à celle construite par Curt. Reproduire ce test pour un nombre de phrases relativement important est, à notre avis, une bonne façon de vérifier que Nessie fonctionne bien. Certes, cela ne suffit pas à prouver l'absence d'erreurs de conception ou de programmation, mais le test n'en demeure pas moins intéressant à notre avis.

La grammaire de Curt permet de reconnaître des phrases déclaratives simples ainsi que des questions. En outre, phrases déclaratives et questions peuvent inclure des coordinations disjonctives et conjonctives. Dans la section 3.2, nous montrerons comment mettre en place le test évoqué ci-dessus. Nous ne nous intéresserons dans cette section qu'aux phrases déclaratives sans coordination. Les phrases avec coordinations seront quant à elles l'objet d'un traitement séparé, tout comme les questions, ces deux classes de phrases conduisant à des discussions qui nous paraissent importantes.

3.2 Phrases simples

Nous allons montrer ici comment construire à l'aide de Nessie une représentation sémantique pour une phrase simple reconnue par Curt, et comment vérifier que ces représentations sont « les mêmes » que celles produites par Curt, « les mêmes » étant à comprendre dans un sens défini ultérieurement. Pour ce faire, nous commençons par expliquer comment mettre au point un lexique utilisable par Nessie, puis nous présentons la construction des arbres syntaxiques nécessaires au calcul sémantique, ce qui nous conduira à expliquer quelles ont été les modifications à apporter à Curt en vue des tests qui nous intéressent. Nous précisons ensuite comment comparer deux représentations sémantiques et terminons cette section consacrée aux phrases simples en donnant une spécification des tests plus précise que le schéma présenté précédemment ainsi que leurs résultats.

3.2.1 Conception d'un lexique pour Nessie

Comme Nessie ne manipule que des termes de TYn et que ceux manipulés par Curt sont non typés, il est nécessaire, pour pouvoir construire avec Nessie des représentations sémantiques des phrases engendrées par la grammaire de Curt, de choisir une version de TYn suffisamment riche pour pouvoir contenir tous les termes utilisés par Curt. Dans la mesure où les phrases qui nous intéressent ont pour objet d'énoncer des propositions concernant des individus, TY1 paraît être un choix relativement naturel, et c'est celui que nous avons fait. Du point de vue de Nessie, les termes qui nous intéressent seront donc vus comme des termes de TY1, le seul type de base autre que le type t des valeurs de vérité étant le type e introduit par Montague et traditionnellement utilisé pour représenter les entités ou individus.

Outre la déclaration du type e, le lexique dont Nessie a besoin doit contenir les noms des familles syntaxiques et des lemmes qui en dépendent, ainsi que leurs représentations sémantiques.

Or, ces informations sont déjà présentes dans `Curt`. En effet, `Curt` dispose d'un lexique syntaxique et de plusieurs lexiques sémantiques. Le lexique syntaxique recense les différentes familles et lemmes manipulés par la grammaire, tandis que les lexiques sémantiques proposent des représentations sémantiques des mots du lexique dans différents formalisme.

On pourrait donc souhaiter construire le lexique de `Nessie` à partir du lexique syntaxique de `Curt` et de l'un de ses lexiques sémantiques, par exemple `semLexLambda.pl`. Il convient cependant de remarquer que les termes que `Curt` associe aux mots du lexique ne sont pas typés. Or, `Nessie` ne peut manipuler que des termes typés. Donc, si nous voulions pouvoir générer automatiquement un lexique pour `Nessie` à partir des informations contenues dans `Curt`, il faudrait trouver un moyen d'annoter les termes de `Curt` par des types.

Nous pensons que cette approche ne peut être mise en œuvre simplement. D'ailleurs, compte tenu de la taille somme toute modeste des lexiques de `Curt`, écrire un lexique équivalent pour `Nessie` à la main ne paraît pas irréaliste. Nous avons donc opté pour cette solution qui a, comme nous le verrons par la suite, une conséquence importante sur la façon dont les tests que nous essayons de mettre en place doivent être menés à bien.

Le lexique que nous avons écrit pour `Nessie` commence par déclarer une constante de négation logique, le type e et quelques abréviations de types utiles :

```
const not : t -> t;
type e;

type pred = e -> t;
type bpred = e -> e -> t;
type np = pred -> t;
```

Le reste du lexique consiste en la déclaration de familles ouvertes et fermées et des lemmes qui leur appartiennent. Les paragraphes qui suivent donnent une brève description de ces familles.

Familles ouvertes Les familles ouvertes disponibles sont les suivantes : noms propres, noms communs, verbes intransitifs, verbes transitifs, prépositions et adjectifs. Les déclarations de ces familles ouvertes sont relativement semblables entre elles, c'est pourquoi nous n'en présentons que deux ici : l'une simple (celle des noms propres), l'autre plus complexe (celle des verbes transitifs). Les déclarations de ces familles se présentent donc comme suit :

```
family pn {
  type = e;
  pattern = lam P : pred. ( P(_) )
};

lemma butch, esmeralda, honey_bunny, jimmy : pn;
lemma jody, jules, lance, marsellus, marvin, mia : pn;
lemma pumpkin, thewolf, vincent, yolanda : pn;

family tv {
  type = bpred;
  pattern =
```

```
    lam K : pred -> t. lam y :e. (K @ lam x:e. (_(y,x)))
};

lemma clean, drink, date, discard, eat, enjoy, hate : tv;
lemma have, kill, know, like, love, pickup, shoot : tv;
```

Familles fermées Comme nous l'avions expliqué au chapitre précédent, les familles fermées sont celles dont les lemmes ne partagent pas une même représentation sémantique dérivée d'un patron commun à toute la famille. Les lemmes appartenant à des familles fermées ont des représentations sémantiques distinctes, et ces familles ne permettent donc quasiment aucun partage de l'information sémantique. À titre d'exemple, voici comment a été déclarée la famille des déterminants, qui comporte deux lemmes : un pour le déterminant universel apparaissant par exemple dans « every man walks », l'autre étant le déterminant indéfini apparaissant dans « a man walks ». La déclaration de la famille et des deux lemmes se présente comme suit :

```
family det;

lemma uni {
  family = det;
  term = lam p, q : pred. forall x : e. ( p(x) => q(x) )
};

lemma indef {
  family = det;
  term = lam p, q : pred. exists x : e. ( p(x) && q(x) )
};
```

Les déclarations des autres familles fermées sont similaires à celles qui viennent d'être présentées. En voici une description :

Copules Cette famille comporte deux lemmes `pos` et `neg` et permet de reconnaître des phrases comme « Mia is a woman » et « Mia is not a plant » ;

Pronom relatif cette famille `relpro` n'a qu'un seul lemme, `relpro`. Il s'agit de reconnaître des constructions comme « A woman who smokes collapses » ;

Verbe auxiliaire Il y a ici deux lemmes, comme pour les copules, pour reconnaître des phrases telles que « Vincent does love Mia » et « Vincent does not love Mia » ;

3.2.2 Construction d'arbres syntaxiques

Nous l'avons dit, l'autre entrée nécessaire à `Nessie` pour le calcul de représentations sémantiques consiste en des arbres représentant la structure des phrases à traiter. Ces arbres doivent être fournis par l'analyseur syntaxique, puisque lui seul connaît la structure de la phrase à analyser. Or, dans sa version originale, `Curt` n'est pas en mesure de construire de tels arbres. La seule valeur à laquelle on a accès après la phase d'analyse syntaxique est la représentation sémantique β-réduite de la phrase qui vient d'être analysée. Une modification de la DCG de `Curt` est donc

nécessaire, de sorte que celle-ci fournisse aussi, à l'issu de l'analyse syntaxique, un arbre de la phrase analysée qui pourra ensuite être passé à `Nessie`.

Dans la DCG originale de `Curt`, les fonctionnalités de `Prolog` permettant de transporter des attributs pendant l'analyse syntaxique étaient déjà utilisées. Ainsi, tous les prédicats définissant des non-terminaux prennent en argument une liste de paires `attribut:valeur` permettant de calculer et de passer d'un non-terminal à l'autre différentes valeurs utiles soit à la construction sémantique, soit à la reconnaissance du langage engendré. Considérons par exemple la règle suivante présente dans la DCG d'origine :

```
t([sem:T])-->
   s([coord:no,sem:S]),
   {combine(t:T,[s:S])}.
```

Cette règle spécifie qu'un texte `t` peut prendre la forme d'une phrase `s` ne comportant pas de coordination. L'appel `{combine(t:T,[s:S])}` spécifie comment, à partir de la représentation sémantique d'une phrase sans coordinations, obtenir celle d'un texte. La définition de ce prédicat dépend de la sémantique utilisée. Par exemple, dans le fichier `semRulesLambda.pl` définissant une sémantique à base de λ-termes, nous trouvons la règle suivante définissant le comportement de `combine` dans le cas qui nous intéresse :

```
combine(t:Converted,[s:Sem]):-
   betaConvert(Sem,Converted).
```

On le voit, la représentation sémantique du texte est obtenue à partir de celle de la phrase par β-réduction. Cet exemple, auquel nous reviendrons plus tard, illustre donc bien le fait que les attributs de la DCG peuvent être utilisés pour calculer des valeurs pendant l'analyse syntaxique.

Pour illustrer la seconde utilisation des attributs (celle permettant de contrôler le langage reconnu), considérons la règle suivante, elle aussi présente dans la DCG originale de `Curt` :

```
s([coord:no,sem:Sem])-->
   np([coord:_,num:Num,gap:[],sem:NP]),
   vp([coord:_,inf:fin,num:Num,gap:[],sem:VP]),
   {combine(s:Sem,[np:NP,vp:VP])}.
```

Comme dans l'exemple précédent, cette règle fait intervenir des attributs utilisés pour mener à bien un calcul pendant la construction sémantique. On trouve aussi dans les listes apparaissant à la deuxième et à la troisième ligne une paire `num:Num` qui n'apparaît pas dans la première ligne. Ceci signifie que cet attribut, bien qu'il soit utilisé pendant l'analyse de la phrase, ne sera pas transmis au non-terminal `s`. Cet attribut n'est en fait utilisé que pour garantir que le groupe nominal et le groupe verbal s'accordent en nombre, c'est pour cela que nous disons qu'il est utilisé pour contrôler le langage reconnu plutôt que pour calculer une valeur.

En résumé, les attributs sont déjà utilisés de manière intensive par la DCG d'origine. C'est pourquoi il a paru naturel de les utiliser aussi pour construire l'arbre d'analyse dont `Nessie` a besoin. Nous avons donc été amenés à modifier la DCG de `Curt` en ajoutant systématiquement aux listes d'attributs passées entre non-terminaux une paire commençant par `ast:`. À titre d'exemple, voici la règle de définition des phrases comme groupe nominal suivi d'un groupe verbal que nous avions vue précédemment, après modification :

```
s([ast:binary(s,AstNP,AstVP),coord:no,sem:Sem])-->
   np([ast:AstNP,coord:_,num:Num,gap:[],sem:NP]),
   vp([ast:AstVP,coord:_,inf:fin,num:Num,gap:[],sem:VP]),
   {combine(s:Sem,[np:NP,vp:VP])}.
```

La seule différence entre cette version de la règle et la précédente est l'ajout des paires `ast:...`. Intuitivement, ce qu'il se passe est clair : pour analyser une phrase, il faut commencer par reconnaître un groupe nominal. Pendant la reconnaissance de celui-ci, son arbre d'analyse est construit et stocké dans la variable `AstNP`. Ensuite, il faut reconnaître un groupe verbal, dont l'arbre est stocké dans la variable `AstVP`. On a alors reconnu une phrase dont l'arbre d'analyse est un arbre binaire avec comme étiquette `s`, comme premier sous-arbre celui du groupe nominal et comme second sous-arbre celui du groupe verbal.

De façon générale, les modifications que nous avons apporté à la grammaire d'origine pour pouvoir disposer d'un arbre syntaxique sont de deux types : celles portant sur les règles non terminales dont nous venons de donner un exemple, et celles concernant les règles lexicales qui permettent de reconnaître les terminaux définis par le lexique `englishLexicon.pl`.

Concernant les règles non terminales, il est important de comprendre que l'arbre que nous construisons s'inspire fidèlement des définitions du prédicat `combine` contenues dans le fichier `semRulesLambda.pl`. En effet, l'arbre n'est qu'une manière de spécifier comment les représentations des sous-arbres doivent être combinées pour obtenir celle de l'arbre complet, ce que fait précisément `combine`. Chaque règle définissant `combine` a donc un équivalent en terme de construction d'arbre syntaxique. Par exemple, la règle de DCG donnée précédemment pour la construction de l'arbre syntaxique d'une phrase à partir d'un groupe nominal et d'un groupe verbal peut et même doit être rapprochée de la clause suivante de `semRulesLambda.pl` :

```
combine(s:app(A,B),[np:A,vp:B]).
```

En effet, ce que dit cette règle est que la représentation sémantique de la phrase est obtenue en appliquant la représentation sémantique du groupe nominal à celle du groupe verbal. Or, compte tenu du fait que, dans `Nessie`, un arbre `binary(s,T1,T2)` donnera lieu à la représentation sémantique `(T1 T2)`, la règle de combinaison et l'arbre conduiront bien, au final, au même résultat.

De la même façon, considérons la règle de DCG initialement présente dans `Curt` et stipulant qu'un nom peut prendre la forme d'un nom suivi d'un modificateur de nom (c'est-à-dire, pour la grammaire qui nous intéresse, une phrase prépositionnelle ou une clause relative). La règle initiale se présente comme suit :

```
n([coord:no,sem:Sem])-->
   noun([sem:N]),
   nmod([sem:PP]),
   {combine(n:Sem,[noun:N,nmod:PP])}.
```

La règle construisant la sémantique de `n` à partir de celle de `noun` et `nmod`, quant à elle, se présente comme suit :

```
combine(n:app(B,A),[noun:A,nmod:B]).
```

Comme on peut le constater, ici le sens de l'application est inversé par rapport à la règle construisant une phrase à partir d'un groupe nominal et d'un groupe verbal. En effet, dans cette autre règle, le syntagme de gauche (groupe nominal) était appliqué au syntagme de droite (groupe verbal), alors qu'ici c'est le contraire qui se produit : le syntagme de droite (modificateur du nom) est appliqué au syntagme de gauche (nom qu'il modifie).

L'arbre que nous construisons dans ce cas doit bien sûr tenir compte de cette inversion. voici comment se présente la règle de DCG modifiée par nos soins :

```
n([ast:binary(n,AstNmod,AstNoun),coord:no,sem:Sem])-->
   noun([ast:AstNoun,sem:N]),
   nmod([ast:AstNmod,sem:PP]),
   {combine(n:Sem,[noun:N,nmod:PP])}.
```

Comme en témoigne la paire `ast:binary(n,AstNmod,AstNoun)` qui apparaît dans la première ligne de cette règle, l'arbre du modificateur de nom est passé comme second argument de `binary` et celui du nom comme troisième argument, ce qui va bien aboutir, dans `Nessie`, à une représentation sémantique où le modificateur de nom sera utilisé comme foncteur et le nom comme son argument, ce qui correspond bien au comportement spécifié dans la clause correspondante de `combine`.

Le parallélisme entre la définition de `combine` et les arbres à construire apparaît de façon plus évidente encore pour les règles donnant lieu à des arbres n-aires. En effet, pour ces règles, l'arbre n-aire qui est construit reçoit comme premier argument un terme spécifiant comment combiner les sous-arbres, et ce terme est α-équivalent au terme donné dans le prédicat `combine`. À titre d'exemple, voici la règle originalement utilisée pour construire des phrases « inversées » telles que « does mia love » [2] :

```
sinv([gap:G,sem:S])-->
   av([inf:fin,num:Num,sem:Sem]),
   np([coord:_,num:Num,gap:[],sem:NP]),
   vp([coord:_,inf:inf,num:Num,gap:G,sem:VP]),
   {combine(sinv:S,[av:Sem,np:NP,vp:VP])}.
```

La règle de construction sémantique présente dans `Curt` est la suivante :

```
combine(sinv:app(B,app(A,C)),[av:A,np:B,vp:C]).
```

Et voici la règle modifiée pour construire un arbre syntaxique :

```
sinv(
  [ast:nary(sinv,'lam A, B, C. (B(A(C)))',
  AstAV,AstNP,AstVP),gap:G,sem:S])-->
   av([ast:AstAV,inf:fin,num:Num,sem:Sem]),
```

[2] Ces constructions sont utilisées dans les questions. Nous avons certes dit que nous ne traiterons pas les questions à cet endroit, mais nous utilisons cet exemple parce que c'est le seul cas (avec les coordinations) de règles engendrant des arbres avec plus de deux sous-arbres dans la grammaire. Il est toutefois facile de trouver un exemple de règle avec trois non-terminaux en partie droite : les verbes ditransitifs. Nous avons préféré utiliser l'exemple des phrases avec inversion (déjà reconnues par `Curt`) plutôt que d'étendre la grammaire seulement pour cet exemple.

```
   np([ast:AstNP,coord:_,num:Num,gap:[],sem:NP]),
   vp([ast:AstVP,coord:_,inf:inf,num:Num,gap:G,sem:VP]),
   {combine(sinv:S,[av:Sem,np:NP,vp:VP])}.
```

Pour terminer cette présentation de la construction d'arbres pendant l'analyse syntaxique, il nous reste à montrer comment les règles interagissant avec le lexique doivent être modifiées. Comme le lecteur l'aura peut-être deviné, les arbres créés par ces règles « lexicales » sont des feuilles. Le choix des familles à utiliser pour les étiqueter est relativement simple. Celui des lemmes, en revanche, demande un petit peu plus d'attention. En effet, il s'agit de choisir un lemme de sorte qu'il soit le même pour toutes les formes syntaxiques possibles. Pour les familles ouvertes, le lemme utilisé pour étiqueter les feuilles est le symbole apparaissant dans leur représentation sémantique. Ainsi, la règle de recherche dans le lexique des verbes intransitifs, par exemple, se présente comme suit :

```
iv([ast:leaf(Sym,iv),inf:Inf,num:Num,sem:Sem])-->
   {lexEntry(iv,[symbol:Sym,syntax:Word,inf:Inf,num:Num])},
   Word,
   {semLex(iv,[symbol:Sym,sem:Sem])}.
```

(la seule modification par rapport à la règle originale étant l'ajout de l'attribut `ast:leaf(Sym,iv),inf:Inf,` dans la première ligne de la règle). Les déclarations des autres familles ouvertes sont très similaires à celle-ci. Pour ce qui est des familles fermées, le choix des lemmes à utiliser est différent pour chaque famille. Pour la famille des pronoms relatifs, qui ne comporte en fait qu'un seul lemme, la feuille correspondante est directement placée dans la liste des attributs de la DCG. On obtient ainsi la règle :

```
relpro([ast:leaf(relpro,relpro),sem:Sem])-->
   {lexEntry(relpro,[syntax:Word])},
   Word,
   {semLex(relpro,[sem:Sem])}.
```

Pour les déterminants, le lemme utilisé est le type du déterminant, à savoir « uni » pour le déterminant universel et « indef » pour l'indéfini. Enfin, la famille des copules et la famille pour le verbe auxiliaire comportent chacune deux lemmes « pos » et « neg » correspondant aux deux polarités possibles pour les membres de ces familles.

3.2.3 Equivalence de représentations sémantiques

Nous l'avons dit, nous voulons construire des représentations sémantiques avec `Nessie` et vérifier que ce sont les mêmes que celles produites par `Curt`. Mais que signifie précisément « les mêmes » ? Faut-il que les deux représentations soient égales syntaxiquement, ou une notion d'équivalence plus complexe doit-elle être utilisée ?

Comme la grammaire de `Curt` peut reconnaître des déterminants, et comme ceux-ci sont représentés à l'aide de variables liées, il semble assez évident que l'égalité syntaxique est une relation trop restrictive. En effet, la représentation calculée par `Curt` pour la phrase « Every woman dances » est le terme

```
all(A, imp(woman(A), dance(A)))
```

tandis que la représentation produite par `Nessie` est :

```
all(X,imp(woman(X),dance(X))).
```

Bien sûr, nous pourrions essayer de faire en sorte que les noms des variables utilisés par `Nessie` coïncident avec ceux utilisés par `Curt`, mais cette stratégie a plusieurs inconvénients. Le premier d'entre eux est qu'elle est difficile à mettre en œuvre. En effet, il ne suffit pas de changer le nom de la variable dans la représentation du déterminant universel contenue dans le lexique. Il faut aussi savoir comment les variables seront nommées si ce déterminant apparaît plusieurs fois dans un texte, et faire concorder alors la stratégie de choix de noms utilisée par `Nessie` avec celle utilisée par `Curt`. Un second inconvénient, plus fondamental que le premier, est que l'idée d'essayer de « forcer » les représentations sémantiques à être syntaxiquement égales ne correspond pas bien à notre intuition du sens de ces représentations sémantiques. En effet, ce que nous voulons réellement exprimer, ce n'est pas que les représentations sémantiques à comparer doivent être syntaxiquement égales, mais bien qu'elles doivent être égales modulo les noms donnés aux variables, autrement dit que ces noms n'ont pas d'importance. Cette notion d'égalité « aux noms de variables près » est bien connue en λ-calcul sous le nom d'α-équivalence. Deux λ-termes sont dits α-équivalents précisément s'ils sont égaux modulo les noms des variables liées. Ainsi, les termes $\lambda x.x$ et $\lambda y.y$ sont α-équivalents et représentent tous deux la fonction identité.

Dire que deux représentations sémantiques sont « les mêmes » signifie donc dire qu'elles sont α-équivalentes.

3.2.4 Mise en place du test

Dans les sous-sections précédentes, nous avons présenté un lexique permettant à `Nessie` de calculer des représentations sémantiques pour les phrases reconnues par `Curt`, puis nous avons montré comment modifier la DCG de `Curt` pour pouvoir disposer, en sortie de l'analyse syntaxique, d'un arbre utilisable par `Nessie`. Voici par exemple comment récupérer l'arbre d'analyse de la phrase « Vincent walks » :

```
?- t([ast:Ast|_],[vincent,walks],[]).
Ast =
unary(t,
  binary(s,
    unary(np, leaf(vincent, pn)),
    unary(vp, leaf(walk, iv))
  )
)
Yes
```

Nous allons voir maintenant comment passer cet arbre à `Nessie`, comment récupérer la représentation sémantique produite et comment la comparer à celle produite par `Curt`.

Mais d'abord, revenons un instant au lexique que nous avons mis au point pour `Nessie`. Comme nous l'avions expliqué alors, ce lexique n'a pu être produit automatiquement à partir des ressources présentes dans `Curt`, et nous avons donc dû l'écrire manuellement. Or le fait que le lexique ait été produit manuellement pose un problème. Si l'on se contente de comparer les

représentations β-réduites calculées par `Nessie` et `Curt`, que devrons-nous conclure s'il s'avère que, pour une phrase donnée, elles ne sont pas α-équivalentes ? Leur différence sera-t-elle due à une erreur dans le lexique, ou à un problème pendant la β-réduction dans `Nessie` (ou `Curt`) ?

Ces interrogations montrent à notre avis que la comparaison des représentations sémantiques β-réduites produites par `Curt` et `Nessie` n'est pas suffisante pour localiser un éventuel problème. La solution que nous proposons pour remédier à ce problème consiste à faire deux comparaisons : celle des représentations sémantiques non β-réduites, puis celle des représentations sémantiques réduites. Ainsi, il est possible de savoir si une erreur provient du lexique ou de l'arbre d'une part, ou de l'algorithme de β-réduction, d'autre part. En effet, la comparaison des valeurs sémantiques non réduites permet de vérifier que `Nessie` construit correctement une première représentation sémantique à partir des données qu'il reçoit en entrée. De plus, si les valeurs non β-réduites sont équivalentes et que les représentations β-réduites ne le sont pas, alors l'erreur se trouvera manifestement dans l'algorithme de β-réduction.

Ces remarques nous permettent de spécifier comment doit se dérouler un test d'une façon un peu plus précise que ce que nous avons fait jusqu'ici. Voici donc la procédure de test proposée. Etant donnée une phrase, on l'analyse avec `Curt`. En sortie de l'analyse syntaxique, on obtient une représentation sémantique non β-réduite C_1 et un arbre d'analyse T. Dans `Curt`, on réduit C_1 vers C_2. Puis on passe l'arbre T ainsi que le lexique développé précédemment à `Nessie`, en lui demandant de générer à la fois une représentation non β-réduite N_1 et une représentation β-réduite N_2. Enfin, on vérifie l'α-équivalence de C_1 et N_1 d'une part, et de C_2 et N_2 d'autre part.

Il reste cependant une modification à effectuer dans la grammaire de `Curt` avant de pouvoir mettre en œuvre cette procédure de test. Nous avons en effet supposé que la phase d'analyse syntaxique fournissait une représentation non β-réduite. Or, comme nous l'avons vu précédemment, ce n'est pas ce que fait la version originale de `Curt`, puisque les représentations sémantiques sont réduites par le prédicat `combine`. Il faut donc retirer cette étape de réduction de la construction sémantique, de sorte à pouvoir récupérer des représentations sémantiques non réduites que l'on pourra réduire ultérieurement. Une fois cette modification faite, la grammaire est prête à être utilisée par notre procédure de test, dont nous allons maintenant décrire la mise en œuvre.

Nous commençons par écrire un petit programme `Prolog termae.pl` qui lit des termes dans les fichiers dont les noms lui sont donnés en entrée et vérifie leur α-équivalence. Notons que l'essentiel du code nécessaire à cette vérification était déjà présent dans `Curt`.

Puis, nous créons un programme que nous appelons `Scriptable Curt` et qui s'inspire du fichier `lambda.pl` fourni par `Curt`. Ce programme `Prolog` lit une phrase sur son entrée standard, l'analyse puis écrit son arbre d'analyse, sa représentation non β-réduite et sa représentation β-réduite dans trois fichiers distincts.

Enfin, nous utilisons un script qui lit des phrases depuis un fichier, les transmet à `Scriptable Curt` puis vérifie les deux α-équivalences des formes non réduites et réduites.

3.2.5 Résultats

La version 1.3 de `Curt` avec laquelle nous travaillons définit un ensemble de phrases reconnues par `Curt` et qui peuvent être utilisées pour des tests. L'ensemble initial comporte 45 phrases. Sur ces 45 phrases, 5 sont des questions et 16 sont des phrases déclaratives avec coor-

dinations. Comme nous l'avions indiqué, la construction sémantique de ces phrases (questions et phrases déclaratives avec coordination) fera l'objet des deux sections suivantes. Il nous reste donc un ensemble de 24 phrases déclaratives sans coordinations, auquel nous en avons ajouté 3 incluant des adjectifs, cette construction n'étant utilisée par aucune phrase de test de `Curt`, bien que reconnue par sa grammaire. Nous obtenons donc un corpus de 27 phrases sur lequel nous pouvons lancer le test mis au point précédemment. Les résultats affichés par le programme de test se présentent comme suit :

```
Processing sentence 1: a man walks
Testing equivalence before reduction... Ok
Testing equivalence after reduction... Ok
Processing sentence 2: mia dances
Testing equivalence before reduction... Ok
Testing equivalence after reduction... Ok
...
Processing sentence 27: every tall man is not a small plant
Testing equivalence before reduction... Ok
Testing equivalence after reduction... Ok
```

Les deux tests d'α-équivalence sont réussis pour chacune de ces 27 phrases. Ceci est, nous semble-t-il, un indicateur significatif du bon fonctionnement de `Nessie`. Plus précisément, le fait que les représentations sémantiques non réduites produites par `Nessie` sont bien α-équivalentes à celles produites par `Curt` montre le bon fonctionnement de l'algorithme d'instanciation (qui construit un terme à partir d'une feuille par consultation du lexique). En outre, le fait que les représentations sémantiques produites par `Nessie` et `Curt` demeurent équivalentes après β-réduction atteste du bon fonctionnement de l'algorithme de β-réduction de `Nessie`. À ce sujet, notons que tous ces tests, dont le but était de vérifier le bon fonctionnement de `Nessie`, ont permis de mettre en évidence une erreur de programmation de l'algorithme de β-réduction non pas de `Nessie`, mais de `Curt`. Ce dernier a donc lui aussi pu profiter de cette batterie de tests.

Deux remarques méritent d'être faites en conclusion de cette première série de tests.

D'une part, il y a un autre indice important qui nous permet de penser que `Nessie` fonctionne bien : le typage. En effet, aucune erreur de typage ne se produit lors des tests. Or une erreur (par exemple dans l'élaboration du lexique) aurait de grandes chances de produire une erreur de typage. L'absence d'erreur de typage nous semble donc une garantie relativement sûre que les choses se passent effectivement comme nous le souhaitons.

D'autre part, il est important de remarquer qu'une partie de `Nessie` n'est que peu utilisée (et donc ne saurait être complètement validée) par les phrases testées jusqu'ici. En effet, la plupart des règles de grammaire utilisées pour engendrer les phrases considérées ici produisent des arbres qui n'utilisent pas de λ-termes pour spécifier comment combiner les représentations sémantiques des sous-arbres. Les seules règles incluant explicitement des λ-termes dans les arbres sont celles reconnaissant des structures comme « if Sentence then Sentence » ou « either Sentence or Sentence ». Mais, dans ces cas, les arbres concernés n'ont qu'un ou deux sous-arbres. On n'a donc pas eu affaire, pour l'instant, à des arbres n-aires avec combinateur et n sous-arbres, et donc la construction sémantique pour ce type de nœuds n'a pas encore été pleinement

testée. Elle le sera lorsque nous étendrons les tests aux phrases avec coordinations, ce que nous allons faire dans la section suivante.

3.3 Coordinations

En plus des phrases déclaratives simples examinées jusqu'ici, la grammaire de `Curt` est suffisamment riche pour pouvoir reconnaître des phrases comportant des coordinations. Ainsi, les phrases suivantes sont reconnues :

1. Vincent or Mia dances ;
2. Every man or woman dances ;
3. Mia dances or smokes.

Tous ces exemples utilisent la disjonction, mais des exemples similaires utilisant la conjonction sont également reconnus. Par ailleurs, il est important de remarquer que chacun de ces exemples fait une utilisation différente de la coordination. Plus précisément, dans chacun des exemples, ce sont des constituants syntaxiques différents qui sont coordonnés. Dans la phrase 1 la conjonction coordonne des groupes nominaux, tandis qu'elle coordonne des noms dans la phrase 2 et des groupes verbaux dans la phrase 3.

Dans `Curt`, le support des coordinations est entièrement lexicalisé. En effet, dans le fichier `semLexLambda.pl` de `Curt`, nous pouvons trouver les représentations des coordinations. Elles sont définies comme suit :

```
semLex(coord,M):-
   M = [type:conj,
        sem:lam(X,lam(Y,lam(P,and(app(X,P),app(Y,P))))))];
   M = [type:disj,
        sem:lam(X,lam(Y,lam(P,or(app(X,P),app(Y,P))))))].
```

En d'autres termes, la coordination conjonctive « and » est représentée par le terme $\lambda XYP.(XP) \wedge (YP)$, tandis que la coordination disjonctive « or » est représentée par le λ-terme $\lambda XYP.(XP) \vee (YP)$. Nous aimerions donc ajouter au lexique de `Nessie` une famille fermée `coord` comportant deux lemmes `conj` et `disj` dont les représentations sémantiques sont celles que nous venons de donner. Cependant, pour y parvenir, nous devons faire en sorte que les termes représentant les coordinations puissent être typés, sans quoi ils seront refusés par `Nessie`. Si l'on note $\tau(V)$ le type d'une variable V, alors le type des deux termes représentant les coordinations peut s'écrire comme $\tau(X) \rightarrow \tau(Y) \rightarrow \tau(P) \rightarrow t$, où t représente comme nous l'avons déjà vu le type des valeurs booléennes. Mais que valent $\tau(X)$, $\tau(Y)$ et $\tau(P)$? Un examen plus attentif des définitions des coordinations montre que, dans les deux cas, X et Y sont des fonctions prenant P en argument et renvoyant des valeurs de type t, ce qui peut se traduire par les équations de types suivantes :
$\tau(X) = \tau(P) \rightarrow t$ et $\tau(Y) = \tau(P) \rightarrow t$. En posant $\tau(P) = \alpha$, nous pouvons donc réécrire le type des termes représentant les coordinations de la façon suivante :
$(\alpha \rightarrow t) \rightarrow (\alpha \rightarrow t) \rightarrow \alpha \rightarrow t$. Il reste donc, pour pouvoir affecter un type aux coordinations, à trouver la valeur de α. Pour y parvenir, nous pouvons chercher où la coordination est employée

dans la grammaire de `Curt` et voir quel est le type des constituants coordonnés. Or, comme nous l'avons vu au début de cette section, les coordinations peuvent coordonner des noms, des groupes nominaux et des groupes verbaux. Cependant, les types sémantiques associés à ces différents constituants sont différents. En effet, les noms ont pour type $e \rightarrow t$, tandis que les noms propres et groupes verbaux ont pour type $(e \rightarrow t) \rightarrow t$. Nous remarquons donc que, dans le cas des noms communs, la coordination est utilisée avec $\alpha = e$, alors que pour une utilisation de la coordination entre groupes nominaux ou groupes verbaux nous avons $\alpha = e \rightarrow t$. Ceci signifie que la coordination telle qu'elle est définie dans `Curt` est *polymorphe*, c'est-à-dire que ses arguments peuvent être de plusieurs types, pourvu que ces types satisfassent les équations de typage données précédemment. Cependant, le système de typage TYn sur lequel `Nessie` est basé ne permet pas l'expression de tels types polymorphes. Donc, le traitement des coordinations par `Curt` ne peut être repris tel quel dans `Nessie`. Il faut trouver une solution ne faisant pas appel à du polymorphisme.

La solution que nous proposons consiste à utiliser plusieurs coordinations différentes, à savoir une par constituants syntaxiques que l'on souhaite coordonner. Ici, comme nous l'avons vu, la grammaire de `Nessie` ne permet des coordinations qu'entre noms, groupes verbaux et groupes nominaux. Nous définissons donc trois familles fermées (`coordn`, `coordnp` et `coordvp`) comportant chacune deux lemmes `conj` et `disj`. Voici les définitions qui sont ajoutées au lexique de `Nessie` :

```
family coordn;

lemma conj {
  family = coordn;
  term = lam n1, n2 : pred. lam x : e. ( n1(x) && n2(x) )
};

lemma disj {
  family = coordn;
  term = lam n1, n2 : pred. lam x : e. ( n1(x) || n2(x) )
};

family coordnp;

lemma conjnp {
  family = coordnp;
  term = lam np1, np2 : np. lam p : pred. ( np1(p) && np2(p) )
};

lemma disjnp {
  family = coordnp;
  term = lam np1, np2 : np. lam p : pred. ( np1(p) || np2(p) )
};

family coordvp;

lemma conj {
```

```
  family = coordvp;
  term = lam vp1, vp2 : pred. lam x : e. ( vp1(x) && vp2(x) )
};

lemma disj {
  family = coordvp;
  term = lam vp1, vp2 : pred. lam x : e. ( vp1(x) || vp2(x) )
};
```

Il reste à voir comment construire les arbres syntaxiques correspondant à ces constructions. Il y a ici une difficulté qui est que, au moment où la feuille correspondant à la conjonction de coordination est parcourue, on ne sait pas quels sont les constituants syntaxiques qu'elle coordonne. Il n'est donc pas possible de générer la bonne feuille. La solution que nous proposons consiste à utiliser une feuille générique représentant une conjonction de coordination sans spécifier laquelle, feuille qui sera remplacée, dans la règle de DCG coordonnant deux constituants, par la véritable conjonction, celle qui coordonne les deux constituants syntaxiques dont on dispose. La règle assurant l'interface avec le lexique pour y rechercher des conjonctions de coordination s'écrit donc comme suit :

```
coord([ast:leaf(Type,coord),type:Type,sem:Sem])-->
   {lexEntry(coord,[syntax:Word,type:Type])},
   Word,
   {semLex(coord,[type:Type,sem:Sem])}.
```

La variable `Type` est instanciée dans le lexique syntaxique et peut prendre les deux valeurs `conj` et `disj`. Les deux feuilles qui peuvent être générées par cette règle sont donc `leaf(conj, coord)` et `leaf(disj, coord)`.

Comme nous l'avons dit, les règles coordonnant plusieurs constituants syntaxiques doivent « modifier » cette feuille en spécifiant la famille adéquate pour la coordination. Voici par exemple comment est écrite la règle coordonnant deux noms :

```
n([
   ast:nary(n,'lam A, B, C. (B(A,C))',AstN1,AstCoord,AstN2),
   coord:yes,sem:N
  ])-->
   n([ast:AstN1,coord:no,sem:N1]),
   coord([ast:_,type:Type,sem:C]),
   n([ast:AstN2,coord:_,sem:N2]),
   {AstCoord=leaf(Type,coordn)},
   {combine(n:N,[n:N1,coord:C,n:N2])}.
```

Pour comprendre ce qu'il se passe au niveau des arbres, supposons que l'on souhaite calculer l'arbre d'analyse pour le nom « man or woman ». Le tableau suivant donne des arbres possibles pour les noms « man » et « woman », ainsi que pour la conjonction disjonctive « or ».

Lemme	Arbre
man	`leaf(man,noun)`
woman	`leaf(woman,noun)`
or	`leaf(disj,coord)`

À partir de ces trois arbres, la règle que nous venons de montrer produira l'arbre suivant :

```
nary(
     n,
     'lam A, B, C. (B(A,C))',
     leaf(man,noun),
     leaf(disj,coordn),
     leaf(woman,noun)
).
```

Le λ-terme qui apparaît dans cet arbre spécifie que sa représentation sémantique est obtenue en appliquant celle du deuxième sous-arbre à celle du premier et à celle du troisième, c'est-à-dire en appliquant le sous-arbre produit par la coordination aux sous arbres correspondant aux deux noms à coordonner, ce qui est bien ce qui est fait dans `Curt`. Les règles permettant de coordonner les groupes nominaux et verbaux fonctionnent exactement de la même façon. Les λ-termes sont les mêmes, la seule chose qui change est la feuille choisie pour représenter la conjonction de coordination.

Avant d'en revenir à notre suite de tests, une remarque sur le polymorphisme nous paraît opportune. Il est clair que si le système de type était polymorphe, une seule coordination aurait suffit, comme dans `Nessie`. Nous aurions alors pu, comme nous l'envisagions au début de cette sous-section, n'ajouter qu'une famille à deux lemmes au lexique de `Nessie`. Avec les notations `OCaml` introduites au chapitre précédent pour représenter les types polymorphes, la déclaration de la famille et des lemmes aurait pu s'écrire ainsi :

```
family coord;

lemma conj {
  family = coord;
  term = lam X, Y : ('a -> t). lam P : 'a. ( X(P) && Y(P) )
};

lemma disj {
  family = coord;
  term = lam X, Y : ('a -> t). lam P : 'a. ( X(P) || Y(P) )
};
```

La règle de DCG avec construction d'arbre syntaxique pour la coordination de noms, que nous avions présentée précédemment, pourrait alors s'écrire, un peu plus simplement :

```
n([
   ast:nary(
            n,
            'lam A, B, C. (B(A,C))',
            AstN1,
            AstCoord,
            AstN2),
   coord:yes,
   sem:N
  ])-->
   n([ast:AstN1,coord:no,sem:N1]),
```

```
coord([ast:AstCoord,type:Type,sem:C]),
n([ast:AstN2,coord:_,sem:N2]),
{combine(n:N,[n:N1,coord:C,n:N2])}.
```

et l'on n'aurait donc plus besoin d'« oublier » la valeur de la feuille correspondant à la coordination, comme on l'a fait à l'aide de la variable anonyme _ pour en mettre une autre, *ad hoc*, à la place. Cette solution paraît certes plus élégante, mais elle pose, en même temps, la question de l'expressivité du système de types ainsi obtenu. Nous reviendrons à cette question et donnerons quelques pistes de réflexion et éléments de réponse dans la conclusion de cette thèse, chapitre 8.

Nous pouvons enfin en revenir aux tests de validation de `Nessie`. Comme nous l'avions expliqué à la fin de la section précédente, 16 phrases, parmi toutes celles fournies avec `Curt`, comportent des coordinations. Dans notre première série de tests, nous n'avions pas considéré ces 16 phrases. Mais, maintenant que, d'une part, le lexique de `Nessie` a été étendu et que, d'autre part, la DCG de `Curt` a été modifiée de sorte à pouvoir produire des arbres aussi pour les coordinations, nous pouvons réintégrer ces 16 phrases avec coordination à l'ensemble de phrases que nous utilisons pour valider `Nessie`. Comme pour les 27 phrases considérées précédemment, nous constatons que, pour ces 16 nouvelles phrases, les deux formes sémantiques produites par `Curt` et `Nessie` sont bien α-équivalentes, à la fois avant et après réduction. Cette nouvelle série de résultats positifs constitue une preuve supplémentaire du bon fonctionnement de `Nessie`. En outre, nous avions signalé que la première série de phrases ne permettait pas de tester complètement le traitement par `Nessie` des arbres n-aires, les arbres générés par cette première série de phrases ne comportant en fait qu'un nœud et un terme modifiant sa représentation. Il en va autrement avec les coordinations, puisque, comme nous avons pu le constater par exemple avec la règle cordonnant les noms, les arbres générés comportent 3 sous-arbres. C'est cette constatation qui nous conduit à penser que cette deuxième série de tests complète la validation, ébauchée par la première, de la correction de la construction sémantique pour les arbres n-aires.

3.4 Questions

Voici des exemples de questions pouvant être reconnues par la grammaire de `Curt` :

1. who dances
2. which robber dies
3. who does mia love

Ces exemples illustrent les trois formes syntaxiques de questions reconnues par `Curt` : groupe nominal interrogatif (noté whnp dans la DCG) suivi d'un groupe verbal (questions 1 et 2) ou d'une phrase inversée (question 3). Les groupes nominaux interrogatifs peuvent se présenter soit sous la forme d'un pronom interrogatif (phrase 1), soit sous la forme d'un déterminant interrogatif suivi d'un nom (question 2). Quant aux phrases inversées, il s'agit d'un verbe auxiliaire suivi d'un groupe nominal et d'un verbe). Notons qu'ici seul le verbe auxiliaire « does » (dans sa forme positive) est autorisé, la forme négative étant plus difficile à reconnaître compte tenu du fait que le « not » vient après le groupe nominal. Pour ce qui est des représentations sémantiques, voici celles construites par `Curt` pour les trois exemples précédents :

1. `que(A, person(A), dance(A))`
2. `que(A, robber(A), die(A))`
3. `que(A, person(A), love(mia, A))`

Nous pouvons dès à présent faire deux remarques concernant ces représentations. La première est qu'elles utilisent une constante `que` qui n'avait pas été rencontrée jusqu'à présent. Pour que `Nessie` soit en mesure de construire ce type de représentation, il est nécessaire de déclarer cette nouvelle constante et, par conséquent, de lui donner un type approprié. La seconde remarque est que la constante `que` semble ici être utilisée comme un lieur. En effet, son premier argument est une variable qui apparaît ensuite libre dans les arguments 2 et 3 de `que`. Or `Nessie` ne permet pas la déclaration de nouveaux lieurs, et encore moins de lieurs liant une variable dans deux termes, comme semble le faire `que` ici.

Donc, si nous voulons que `Nessie` soit en mesure de construire des représentations sémantiques pour les questions, nous avons à résoudre plusieurs problèmes : notamment le choix d'un type pour `que` et la construction effective de termes de la forme `que(variable,arg1,arg2)`.

Nous allons en réalité proposer deux solutions aux problèmes que nous venons de poser. Dans un premier temps, nous voudrions montrer une solution permettant à `Nessie` de construire des représentations sémantiques des questions α-équivalentes à celles construites par `Curt`, dans le même esprit que ce que nous avons fait jusqu'ici pour les phrases simples et celles comportant des coordinations. Dans un second temps, nous voudrions explorer plus avant la piste évoquée précédemment et selon laquelle `que` peut être vu comme un lieur. Ceci nous amènera à proposer une représentation alternative de la sémantique des questions, non α-équivalente à la représentation utilisée par `Curt` mais, à notre avis, plus pertinente d'un point de vue sémantique, surtout dans le contexte du λ-calcul typé qui est le nôtre.

3.4.1 Construction de représentations α-équivalentes à celles de `Curt`

Comme nous l'avons indiqué précédemment, les lexiques de `Nessie` ne permettent pas de déclarer de lieurs ou de quantificateurs. Le symbole `que` utilisé par `Curt` ne peut donc, du point de vue de `Nessie`, qu'être envisagé comme étant une constante. En outre, si nous souhaitons que les termes construits par `Nessie` soient α-équivalents à ceux produits par `Curt`, cette constante doit, comme le suggèrent les représentations sémantiques précédentes, prendre trois arguments : un premier de type e correspondant à la variable, puis deux de type t correspondant à l'application de prédicats à ladite variable. Il reste à choisir le type renvoyé par la constante `que` que nous envisageons d'ajouter au lexique de `Nessie`. Pour trouver ce type, nous pouvons examiner quelles entrées du lexique de `Curt` utilisent cette constante et voir si ces entrées ne seraient pas semblables à des entrées auxquelles un type a déjà été affecté. On remarque alors que, dans le lexique de `Curt`, la constante `que` est utilisée, par exemple, par l'entrée `which`. En effet, nous trouvons dans le lexique de `Curt` la déclaration suivante :

```
semLex(det,M):-
   M = [type:wh,
        sem:lam(P,lam(Q,que(X,app(P,X),app(Q,X))))].
```

Nous pouvons constater d'après cette entrée que `which` est un déterminant, ce que confirme le rapprochement de deux phrases comme « every man walks » et « which man walks ».

Donc, nous pouvons à la fois construire une représentation de `which` qui utilise la constante `que` et, en même temps, par analogie avec les autres déterminants, nous découvrirons ainsi quel type de retour donner à `que`.

De cette recherche, il ressort que, pour que la représentation du déterminant `which` soit cohérente avec celle des autres déterminants du point de vue des types, il est nécessaire que la constante `que` renvoie des valeurs de type t. Nous pouvons donc ajouter au lexique de `Nessie` la déclaration suivante :

```
const que : e -> t -> t -> t;
```

Il reste à voir comment construire, en utilisant cette constante, une représentation sémantique de `which`. Plus précisément, quels arguments faut-il lui passer ? Dans `Curt`, le premier argument est une variable X qui est ensuite appliquée dans les arguments 2 et 3 à deux prédicats. Comme nous l'avions expliqué précédemment, cette variable peut être vue comme une variable libre. Cependant, dans `Nessie`, il n'est pas possible d'utiliser de telles variables, car `Nessie` ne peut manipuler que des termes clos. Pour résoudre ce problème, nous proposons d'ajouter au lexique de `Nessie` une nouvelle constante, `QUESTION`, qui sera utilisée dans la construction `que(...)` en lieu et place des variables libres. Notons que cette idée paraît raisonnable, dans la mesure où constantes et variables libres ne sont, au bout du compte, que deux aspects d'une même notion, deux termes qui correspondent plus à des points de vue différents sur une même chose qu'à deux réalités différentes. Notons en outre que, compte tenu du type de questions reconnues par `Curt`, cette constante est légitime d'un point de vue sémantique. En effet, les questions considérées ici visent toutes à trouver quels individus vérifient certaines propriétés. Les questions peuvent donc être comprises comme des équations dont les inconnues sont des individus, inconnues représentées, précisément, par la constante `QUESTION`. Nous ajoutons donc au lexique de `Nessie` la déclaration suivante :

```
const QUESTION : e;
```

Et nous avons maintenant tout ce dont nous avons besoin pour ajouter à la famille des déterminants le lemme correspondant au déterminant interrogatif :

```
lemma wh {
  family = det;
  term = lam P, Q : pred. ( que(QUESTION,P(QUESTION),Q(QUESTION)) )
};
```

Nous pouvons également, maintenant, définir la représentation sémantique des pronoms interrogatifs. Nous ajoutons pour les définir la famille ouverte qnp à laquelle appartiennent les deux lemmes `person` (correspondant au pronom `who`) et `thing` (correspondant au pronom `what`). Les définitions se présentent comme suit :

```
family qnp {
  type = pred;
  pattern = lam Q : pred. (que(QUESTION,_(QUESTION),Q(QUESTION)))
};

lemma person, thing : qnp;
```

Aux lemmes `person` et `thing` correspondent donc deux constantes homonymes de type `pred` qui traduisent le fait qu'être une chose ou une personne est une propriété des entités. On comprend bien les représentations sémantiques construites, qui reproduisent d'ailleurs fidèlement celles utilisées par `Curt`. Elles traduisent le fait que l'entité cherchée doit vérifier le prédicat donné en argument et être une chose ou une personne, selon le prédicat utilisé.

Il nous reste à voir comment construire les arbres syntaxiques qui permettront à `Nessie` de calculer la représentation sémantique des questions. Pour les règles reconnaissant des questions consistant en un groupe nominal interrogatif suivi d'un groupe verbal, la construction d'arbres ne présente pas de difficultés particulières et utilise les techniques déjà présentées pour les phrases simples. En revanche, les choses se passent différemment pour les questions où le groupe nominal interrogatif est suivi d'une phrase inversée. On se trouve alors dans le cas des dépendances non bornées, bien connu des linguistes. En bref, une question de la forme « who does Mia love » peut être vue comme « Mia does love who ». Ceci signifie que la représentation sémantique du groupe nominal interrogatif doit être transportée jusqu'à l'intérieur du groupe verbal (qui utilise nécessairement un verbe transitif) dont elle devient l'objet. Ce problème linguistique est traité en `Prolog` à l'aide d'une technique appelée « gap threading » et qui est décrite dans FIXME ?.

Ici, nous devons modifier la liste passée dans la paire `gap:[]` de sorte qu'elle transporte non seulement une représentation sémantique, mais aussi l'arbre syntaxique qui lui correspond. On aboutit ainsi à des règles de grammaire de la forme :

```
q([ast:unary(q,AstSINV),sem:Sem])-->
   whnp([ast:A,num:_,sem:NP]),
   sinv([ast:AstSINV,gap:[ast:A,np:NP],sem:S]),
   {combine(q:Sem,[sinv:S])}.

sinv([
      ast:nary(
                sinv,
                'lam A, B, C. (B(A(C)))',
                AstAV,
                AstNP,
                AstVP),
      gap:G,
      sem:S
     ])-->
   av([ast:AstAV,inf:fin,num:Num,sem:Sem]),
   np([ast:AstNP,coord:_,num:Num,gap:[],sem:NP]),
   vp([ast:AstVP,coord:_,inf:inf,num:Num,gap:G,sem:VP]),
   {combine(sinv:S,[av:Sem,np:NP,vp:VP])}.

vp([
    ast:binary(vp,AstTV,AstNP),
    coord:no,
    inf:I,
    num:Num,gap:G,sem:VP
   ])-->
   tv([ast:AstTV,inf:I,num:Num,sem:TV]),
```

```
   np([ast:AstNP,coord:_,num:_,gap:G,sem:NP]),
   {combine(vp:VP,[tv:TV,np:NP])}.

np([ast:A,coord:no,num:sg,gap:[ast:A,np:NP],sem:NP])--> [].
```

La première règle est celle reconnaissant une question consistant en un groupe nominal interrogatif suivi d'une phrase inversée. Si l'on ne s'intéresse qu'à la tête de cette règle, on pourrait penser que l'arbre construit pour la question ne fait intervenir que celui de la phrase inversée, celui du groupe nominal étant en quelque sorte « oublié ». En réalité, les choses ne se passent pas ainsi. En effet, la représentation sémantique du groupe nominal interrogatif, ainsi que son arbre syntaxique sont récupérés et passés à la règle reconnaissant les phrases inversées dans la liste associée à `gap:`. Puis, comme le montrent la deuxième et la troisième règle (celles définissant sinv et vp), la liste d'attributs associée à `gap:` est passée telle quelle d'abord au groupe verbal, puis, si celui-ci commence effectivement par un verbe transitif, à la règle reconnaissant le groupe nominal qui doit normalement compléter les groupes verbaux commençant par un verbe transitif. Enfin, si rien n'apparaît après le verbe, la quatrième règle qui reconnaît la chaîne vide peut s'appliquer. Cette règle récupère l'arbre syntaxique et la représentation sémantique qu'elle reçoit dans la liste associée à `gap:` et les renvoie comme représentation sémantique et arbre syntaxique du groupe nominal suivant le verbe. Notons enfin que les règles 1 et 4 sont les seules à utiliser l'attribut `gap`. Dans toutes les autres règles où il apparaît, il est associé à la liste vide et ne transporte donc aucune information. En outre, comme nous l'avons laissé entendre précédemment, le gap threading était déjà implanté dans la version originale de `Curt`. Notre seul travail ici consiste à l'avoir modifié de sorte qu'il ne transporte pas seulement des représentations sémantiques, mais aussi les arbres syntaxiques correspondant.

Une fois ces extensions apportées au lexique de `Nessie` et à la grammaire de `Curt`, nous pouvons tester la construction de représentations sémantiques par `Nessie` pour les questions, et comparer les représentations obtenues à celles produites par `Curt` selon la procédure maintenant familière. Pour ce test, nous utilisons les 5 questions présentes dans la suite de tests de `Curt`, auxquelles nous en ajoutons 4 nouvelles. Pour ce petit corpus, nous obtenons bien l'équivalence entre les représentations sémantiques produites par `Curt` et celles produites par `Nessie`, tant avant qu'après la β-réduction. En outre, le fait que `Nessie` réduise bien les représentations sémantiques prouve que leur type est correct, ce qui est à notre avis un autre résultat intéressant.

Ce qu'il nous paraît important de retenir de cette première solution aux problèmes posés par les questions, c'est que, bien que `Nessie` ait été conçu pour manipuler des représentations sémantiques closes, il est toujours possible, à l'aide de constantes, de prendre en compte des formalismes utilisant des variables libres. Comme nous le verrons aux chapitres 5 et 6, cette possibilité sera abondamment exploitée pour construire des DRSs en utilisant leur encodage dans le λ-calcul proposé par Reinhard Muskens.

3.4.2 Vers des représentations plus pertinentes

Nous concluons ce premier exemple d'utilisation de `Nessie` en proposant une manière alternative d'envisager la sémantique des questions. `Nessie` ayant déjà été relativement bien testé dans ce chapitre, nous ne chercherons pas ici à comparer les représentations sémantiques produites à d'autres. Nous nous contenterons de proposer une petite modification de la représenta-

tion des questions, et de montrer que `Nessie` est toujours en mesure de construire des représentations sémantiques bien typées.

Nous avons fait remarquer dans l'introduction de cette partie sur les questions que le symbole `que` utilisé par `Curt` pour représenter les questions ressemblait beaucoup à un lieur, dans la mesure où la variable qu'il prend comme premier argument apparaît libre dans son second et son troisième argument. Nous avons aussi fait remarquer qu'il n'était pas possible de déclarer de nouveaux lieurs dans `Nessie` et c'est pourquoi nous avons décidé de voir ce symbole comme une constante. Il reste cependant une possibilité que nous n'avons pas encore évoquée : celle que `que` puisse en réalité être représenté par un lieur qui existe déjà dans `Nessie`. Au premier abord, cette hypothèse peut paraître peu vraisemblable, dans la mesure où `Nessie` n'a pas de lieur liant une variable dans deux termes à la fois.

Revenons pourtant aux représentations sémantiques que nous avions montrées précédemment :

1. `que(A, person(A), dance(A))`
2. `que(A, robber(A), die(A))`
3. `que(A, person(A), love(mia, A))`

Intuitivement, il est clair que l'individu que l'on cherche dans la première représentation sémantique est un individu qui est à la fois une personne *et* qui danse. De la même façon, l'intuition sous-jacente à la seconde (resp. la troisième) représentation est que l'individu cherché est à la fois un voleur *et* meurt (resp. est à la fois une personne *et* aimé par Mia). On le voit, le lien entre le second et le troisième argument de `que` est de nature conjonctive. Donc, on pourrait tout aussi bien réécrire les représentations précédentes à l'aide d'un symbole de conjonction, ce qui donnerait :

1. `que(A, and( person(A), dance(A)))`
2. `que(A, and( robber(A), die(A)))`
3. `que(A, and(person(A), love(mia, A)))`

Il pourrait sembler que nous n'avons pas changé grand chose, et pourtant... Nous avons obtenu des représentations sémantiques dans lesquelles un « lieur », `que`, lie une variable dans *un* terme, et non plus *deux*. En outre, il semble clair que ces représentations sémantiques spécifient « l'ensemble des individus vérifiant les deux propriétés apparaissant dans la conjonction ». C'est pourquoi, il semble assez naturel de considérer que `que` est effectivement un lieur, qui n'est autre que le lieur λ, évidemment présent dans `Nessie`. Nous pouvons donc donner une nouvelle formulation des représentations sémantiques vues précédemment, cette fois sans le `que` :

1. `lam(A, and( person(A), dance(A)))`
2. `lam(A, and( robber(A), die(A)))`
3. `lam(A, and(person(A), love(mia, A)))`

Avec cette représentation, il devient clair que le type des valeurs associé aux questions n'est plus t comme pour les phrases, mais plutôt $e \rightarrow t$ qui est, nous semble-t-il, bien plus légitime, dans le sens où il ne nous paraît pas très naturel qu'une question et une phrase déclarative aient le même type.

Il nous reste à montrer comment on peut modifier le lexique de `Nessie` pour construire ces nouvelles représentations sémantiques. Disons-le tout de suite, il ne sera pas possible, sans bouleverser la grammaire, d'obtenir exactement des représentations de type $e \rightarrow t$. Nous allons donc nous contenter de nous approcher de ce résultat et expliquer comment l'atteindre à partir des représentations sémantiques obtenues.

On remarque tout d'abord qu'il suffit, pour construire une question, de connaître les deux prédicats qui la constituent.

Nous modifions donc le type de la constante `que` pour la transformer en un « constructeur de question » :

```
const que : pred -> pred -> t;
```

C'est dans la déclaration de cette constante que nous n'allons pas jusqu'au bout de notre idée. En effet, nous devrions déclarer :

```
const que : pred -> pred -> predt;
```

mais avec cette seconde déclaration il devient impossible de construire des questions à partir de pronoms interrogatifs suivis de phrases inversées, pour des questions de typage.

Nous modifions ensuite le lexique précédent en déclarant :

```
lemma wh {
  family = det;
  term = lam P, Q : pred. ( que(P,Q) )
};

family qnp {
  type = pred;
  pattern = lam Q : pred. ( que(_,Q) )
};

lemma person, thing : qnp;
```

Avec ces quelques modifications, nous pouvons calculer de nouvelles représentations sémantiques pour les questions considérées précédemment. Les résultats que nous obtenons sont les suivants :

1. que(person,lam(X,dance(X)))
2. que(lam(X,robber(X)),lam(X,die(X)))
3. que(person,lam(X,love(mia,X)))

Certes, comme nous l'avons expliqué ces termes sont vus par `Nessie` comme étant de type t. Mais rien n'empêche, une fois le résultat final obtenu, de les réécrire. La règle de réécriture à appliquer est simple :

$$\text{que}(P, Q) \rightarrow \lambda x : e.(Px) \wedge (Qx)$$

Une telle règle de réécriture ne pourrait cependant être mise en œuvre au sein même de `Nessie`. On devrait donc, si l'on souhaitait l'utiliser, recourir à un outil externe ou implanter cette règle par nous-mêmes. Nous n'allons pas le faire dans ce chapitre, mais nous verrons

un exemple illustrant notre propos au chapitre 6 lorsque nous voudrons simplifier les DRSs obtenues grâce à leur encodage dans la théorie des types tel qu'il a été proposé dans (Muskens, 1996c).

Remarquons pour terminer cette section que le fait de représenter les questions par des termes de type $e \rightarrow t$, s'il peut certes être vu comme une amélioration par rapport à la représentation des questions par des termes de type t, n'en demeure pas moins une solution relativement limitée. Par exemple, une telle représentation n'est pas adéquate pour des questions aussi simples que « Does Vincent love Mia ? ». Pour un traitement plus élaboré de la sémantique des questions, on pourra par exemple se reporter à (Groenendijk, 2003), où la sémantique des questions est représentée à l'aide de partitions d'ensembles de mondes possibles.

3.5 Conclusion

Ce chapitre nous a permis de montrer que `Nessie` fonctionne conformément à la spécification qui en a été donnée au chapitre précédent. Nous parvenons en effet à retrouver les résultats déjà obtenus avec `Curt`, mais nous le faisons en appliquant des techniques plus générales, et cela à plusieurs niveaux. Au niveau logique, nous sommes passés de la logique du premier ordre au λ-calcul simplement typé, ce qui nous permet de mieux rendre compte du contenu sémantique des représentations manipulées et offre plus de garanties, dans la mesure où le type des représentations est vérifié, ce qui n'était pas le cas avec `Curt`. Au niveau de l'interface syntaxe-sémantique, celle que nous proposons est claire, reposant sur des arbres abstraits qui ne dépendent pas d'un langage de programmation particulier et ne sont pas davantage spécifiques à un analyseur ou à un formalisme syntaxique donné. Après avoir montré le bon fonctionnement de cette approche, nous allons nous intéresser, dans le chapitre suivant, à l'étude de son adéquation avec les besoins réels en sémantique computationnelle. Notre démarche pour mener à bien cette étude va consister à appliquer notre outil à un phénomène sémantique précis, à savoir la représentation du temps et de l'aspect des verbes du polonais.

Chapitre 4

Construction de représentations temporelles pour le polonais

Au chapitre 2, nous avions introduit `Nessie`, un outil capable de construire des représentations sémantiques sous la forme de termes de TYn à partir de lexiques et d'arbres. Puis, au chapitre précédent, nous avons montré que cet outil pouvait être intégré au système `Curt` de (Blackburn et Bos, 2005b). Ce travail d'intégration nous a permis non seulement de vérifier le bon fonctionnement de `Nessie`, en comparant les représentations qu'il permet de construire à celles construites par `Curt`, mais aussi de voir comment il était possible d'adapter une grammaire existante, de sorte qu'elle puisse être utilisée avec `Nessie` pour construire la représentation sémantique des phrases qu'elle engendre. Nous allons à présent mettre à profit les enseignements tirés du chapitre précédent pour étudier l'impact du passage de la logique du premier ordre à la logique d'ordre supérieur avec types sur les tâches d'inférence. Cette question étant vaste, nous n'essaierons pas ici d'y apporter une réponse complète. Notre démarche consiste plutôt à considérer un exemple de tâche nécessitant le recours à des outils d'inférence et à nous demander quels enseignements il est possible de tirer du traitement de cet exemple.

La tâche d'inférence que nous avons choisie de traiter ici est la construction de modèles (au sens de la logique du premier ordre) restituant le contenu temporel et aspectuel de phrases polonaises simples. Le choix de la langue polonaise s'explique par le fait que, dans cette langue, les verbes ont un contenu aspectuel à la fois riche et fortement lexicalisé, ce qui signifie que des phrases syntaxiquement simples peuvent exprimer une sémantique complexe. En d'autres termes, nous avons choisi la langue polonaise parce qu'elle offre la possibilité d'étudier des phénomènes sémantiquement riches sans avoir pour cela à recourir à une grammaire complexe. Notre travail s'appuie en premier lieu sur (Młynarczyk, 2004), qui propose une classification des verbes polonais en cinq classes. Bien que cette classification ait été établie à partir de critères et tests morphologiques, il s'avère que chacune des classes verbales ainsi dégagées a une sémantique temporelle et aspectuelle précise, distincte de celles des autres classes. Par la suite, Aalstein (Młynarczyk) et Blackburn ont proposé dans (Alstein et Blackburn, 2007), un article non publié à ce jour, d'exprimer la sémantique des classes verbales à l'aide d'événements, ces entités abstraites qui ont été introduites à la section 1.2.1. Cependant, cette proposition a été faite dans le cadre du λ-calcul non typé. En outre, aucune description formelle de l'ontologie utilisée pour rendre compte des événements n'est fournie, les auteurs se contentant de donner quelques intuitions quant aux propriétés qu'elle pourrait avoir.

Notre premier travail dans ce chapitre consistera donc à proposer une sémantique dans TYn pour une petite grammaire du polonais, ce qui implique, comme nous l'avons vu en section 1.2.1,

non seulement d'associer aux mots du lexique des représentations sémantiques dans TYn, mais aussi de formaliser les propriétés de l'ontologie utilisée. Une fois ces propriétés formalisées, nous les utiliserons pour envisager la construction de modèles proprement dite. Bien que le module d'inférence défini pour `Curt` dans (Blackburn et Bos, 2005b) utilise à la fois la preuve de théorèmes et la construction de modèles, nous avons choisi ici de nous intéresser plus spécifiquement à la construction de modèles, car elle pose des problèmes qui n'apparaissent pas lors de la preuve de théorèmes. Cette dernière n'a en effet besoin que des axiomes régulant l'ontologie considérée, tandis que pour la construction de modèles, il faut tenir compte du fait que les programmes qui la mettent en œuvre construisent des modèles *minimaux*, qui ne sont pas nécessairement les seuls modèles intéressants dans le contexte de la sémantique computationnelle.

Comme nous l'avons laissé entendre, la construction des modèles sera implantée dans `Curt`, qui reste responsable de l'analyse syntaxique et de l'inférence, tandis que le calcul sémantique est externalisé et confié à `Nessie`.

Nous commençons par énoncer (section 4.1) quelques propriétés de la langue polonaise, concernant principalement les verbes et les groupes nominaux. Puis, dans la section 4.2, nous introduisons d'abord la classification des verbes polonais établie dans (Młynarczyk, 2004), puis la proposition de donner aux verbes une sémantique à base d'événements, développée dans (Alstein et Blackburn, 2007) et utilisant le λ-calcul non typé. Dans la section 4.3, nous donnons une sémantique dans TYn pour le fragment de polonais considéré. Comme nous le verrons, il s'agira surtout de trouver une représentation convenable pour les verbes. Celle à laquelle nous aboutirons nous permettra de réutiliser, pour les autres catégories syntaxiques, les représentations qui leurs sont classiquement associées dans TY1. Cette section décrit également l'implantation de cette méthode à l'aide d'une DCG et de `Nessie`. Nous poursuivons (section 4.4) par l'axiomatisation des événements. Après l'avoir présentée, nous montrons comment elle permet de construire un premier modèle minimal pour les représentations sémantiques considérées ; nous verrons aussi que les annotations en types présentes dans le lexique sémantique de `Nessie` peuvent être utilisées pour générer automatiquement une théorie du premier ordre axiomatisant les contraintes exprimées par les types. Enfin, dans la section 4.5, nous donnons un algorithme qui, étant donné un modèle minimal tel que celui construit précédemment, est capable, en le « perturbant », de construire *tous* les modèles pertinents de la représentation sémantique considérée, du point de vue du temps et de l'aspect.

4.1 Quelques pré-requis sur le polonais

Nous commençons par introduire la notion d'aspect, puis expliquons comment elle est exprimée en polonais. Nous évoquons ensuite le temps, tant dans sa dimension morpho-syntaxique que dans sa dimension sémantique. Nous terminons cette petite introduction au polonais par une remarque sur les groupes nominaux.

4.1.1 La notion d'aspect

Il nous semble que l'aspect est, en général, une notion relativement floue et dont la définition a pu évoluer au cours du temps. En résumé, nous dirons que l'aspect est le point de vue avec

lequel on s'intéresse à une action. Ce point de vue peut être soit ponctuel, soit considérer l'action comme prenant du temps. Dans le cas où le point de vue est ponctuel, on parle d'aspect *perfectif*. Dans le cas où c'est l'action dans sa durée qui est considérée, on parle de point de vue *imperfectif*. Ainsi, voici la définition donnée par Saussure des points de vue imperfectifs et perfectifs : « Le perfectif représente l'action dans sa totalité, comme un point, en dehors de tout devenir ; l'imperfectif la montre en train de se faire, et sur la ligne du temps » (Saussure, Linguistique Générale, 1916, p. 162).

4.1.2 Expression de l'aspect et du temps en polonais

En polonais, les verbes à l'infinitif existent tous sous au moins deux formes : l'une imperfective et l'autre perfective. Le passage d'une forme « racine » à la forme dérivée (ou aux formes dérivées) se fait grâce à des règles morphologiques précises. Par exemple, le verbe français « marcher » existe en polonais sous une forme racine imperfective « spacerować » et sous une forme perfective dérivée de la première par l'ajout du préfixe « po- », à savoir « pospacerować ». Pour d'autres verbes, c'est la forme perfective qui est la forme racine et la forme imperfective qui est obtenue par dérivation. C'est par exemple le cas pour le verbe « acheter » dont la racine perfective est « kupić » et à partir de laquelle est construite la forme dérivée imperfective « kupować ». Il convient également de remarquer que toutes les formes perfectives n'ont pas la même sémantique, et qu'une même racine imperfective peut donner lieu à plusieurs formes perfectives dérivées. Pour illustrer les différentes sémantiques des formes perfectives, nous pouvons remarquer que, si la plupart d'entre elles se réfèrent à la fin d'une action, à l'action accomplie, certaines formes perfectives ont une lecture *inchoative*, ce qui signifie qu'elles se réfèrent au moment à partir duquel une action commence à avoir lieu, ou à partir duquel il est vrai qu'une propriété est satisfaite. Par exemple, les deux formes possibles du verbe « aimer » sont « kochać » (racine imperfective) et « pokochać » (forme dérivée perfective), ce dernier verbe pouvant se traduire par « avoir commencé à aimer » ou, peut-être plus naturellement par « être tombé amoureux de ».

Quant au fait qu'une racine imperfective puisse donner lieu à plusieurs formes perfectives, nous pouvons l'illustrer en citant les différentes formes disponibles en polonais pour les verbes « écrire » et « toquer ». La forme racine de « écrire » est « pisać » (imperfectif) à partir de laquelle sont construites deux formes perfectives, « napisać » et « popisać ». La différence sémantique entre ces deux derniers verbes porte sur la manière dont l'écriture s'est arrêtée. Dans le cas de « napisać », on sait que non seulement l'écriture s'est arrêtée, mais qu'en plus ce qui était en train d'être écrit l'a été complètement. Par la suite, nous dirons qu'il y a eu *culmination* pour indiquer le fait qu'une action s'est arrêtée « naturellement », parce qu'elle avait atteint son terme, produit le résultat qu'elle devait produire. La variante « popisać », quant à elle, dit exactement le contraire, à savoir que, bien que l'écriture se soit arrêtée, elle n'a pas été jusqu'à son terme naturel, ayant été interrompue prématurément et donc avant que le résultat attendu ait pu être produit. Ce verbe indique donc que ce qui était en train d'être écrit ne l'a pas été complètement.

Quant au verbe « toquer », sa racine imperfective est « pukać » qui signifie « être en train de toquer ». À partir de cette racine sont construites trois formes perfectives dérivées : « zapukać », « popukać » et « puknąć ». La première forme signifie simplement « avoir toqué ». La seconde peut se traduire par « avoir toqué pendant un certain temps », tandis que la troisième a une lecture sémelfactive, c'est-à-dire qu'elle signifie « avoir toqué une seule fois ».

Enfin, comme nous l'avons laissé entendre, toutes les formes verbales que nous venons d'examiner, qu'elles soient racine ou dérivées, perfectives ou imperfectives, sont des formes à l'infinitif. Chacune de ces formes verbales peut donc être conjuguée, et c'est la conjugaison qui permet d'ajouter une information temporelle à l'information aspectuelle dont le verbe à l'infinitif est déjà porteur. Au niveau morphologique, il y a deux temps disponibles en polonais : le passé et le présent. Cependant, le temps sémantique d'un verbe conjugué dépend à la fois du temps morphologique *et* de l'aspect du verbe conjugué. En effet, si un verbe perfectif est conjugué au présent, il doit être interprété comme un futur, le temps de l'action se situant après le temps de l'énonciation. Pour reprendre l'exemple du verbe « pospacerować » (forme perfective de « spacerować », se promener », la phrase « Piotr pospaceruje » dans laquelle « pospacerować » est conjugué au présent est à interpréter à l'aide d'un futur et peut donc se traduire par « Pierre se sera promené ». Les autres interactions entre temps morphologique et aspect s'interprètent d'une manière plus intuitive. Le présent morphologique d'un verbe imperfectif signifie que l'action décrite est en train d'avoir lieu au moment de l'énonciation. Le passé morphologique, quant à lui, interagit de manière « monotone » avec l'aspect. En d'autres termes, le passé d'un verbe imperfectif traduit une action en cours dans le passé et le passé d'un verbe perfectif traduit une action qui s'est terminée dans le passé. En résumé, l'aspect « morphologique » d'un verbe ne se traduit pas seulement par l'aspect sémantique, il a aussi des conséquences sur le temps sémantique du verbe. L'aspect morphologique est donc porteur de deux informations : l'une ayant trait à l'aspect sémantique, l'autre étant utilisée, en combinaison avec le temps morphologique du verbe, pour déterminer son temps sémantique. Cette remarque pourrait sembler anodine, mais son importance devrait apparaître lorsque nous présenterons les diverses approches possibles pour la construction compositionnelle de représentations sémantiques.

4.1.3 Groupes nominaux

La dernière précision d'ordre général que nous aimerions apporter sur le polonais concerne les groupes nominaux. Plus précisément, il faut savoir qu'en polonais, comme dans les autres langues slaves, un groupe nominal peut être construit à partir d'un nom uniquement, sans qu'on ait besoin de l'accompagner d'un article. Ainsi, le nom « list » (lettre) constitue à lui seul un groupe nominal. En l'absence d'article, l'interprétation de tels groupes nominaux est source d'ambiguïté, puisqu'on ne sait pas si le nom est à interpréter de façon définie ou indéfinie. La « bonne » lecture dépend du contexte et les locuteurs slaves parviennent à désambiguïser ce type de construction sans difficulté. Pour ce qui est des outils de traitement automatique des langues, un outil réaliste de construction sémantique pour une langue slave se doit de générer les deux interprétations possibles de chaque groupe nominal de cette forme. Nous reviendrons sur ce problème lorsque nous présenterons notre implantation de la construction sémantique pour un fragment de polonais.

4.2 La proposition de Aalstein (Młynarczyk)

Dans (Młynarczyk, 2004), Aalstein propose une classification des verbes polonais qui généralise et systématise les remarques faites à la section précédente. Les critères permettant d'établir

cette classification sont essentiellement d'ordre morphologique. Pour chaque verbe, on détermine si sa racine est imperfective ou perfective et quelles sont les règles qui, étant donné cette racine, peuvent être utilisées pour construire des formes dérivées. Cette méthode aboutit à une répartition des verbes en cinq classes. Dans l'une d'entre elles (la classe 5), chaque verbe racine est une forme perfective et ne donne lieu qu'à une seule forme imperfective. Dans les 4 autres classes, les verbes racines sont des formes imperfectives dont le nombre de formes dérivées perfectives dépend de la classe.

Le principal intérêt de cette classification est que, bien qu'elle ait été établie à partir de critères morphologiques, chacune des classes verbales ainsi dégagée a des propriétés sémantiques caractéristiques. Les verbes de la classe 2, par exemple, sont tous des processus (non instantanés) qui peuvent s'arrêter à tout moment et auxquels aucune notion de résultat ou d'accomplissement n'est associée. C'est pourquoi ils n'ont qu'une forme perfective dérivée traduisant la fin du processus. Le verbe « spacerować » (être en train de marcher) que nous avions introduit précédemment est un membre de cette classe. Les verbes de la classe 3, quant à eux, sont appelés des *processus culminants*. Cette appellation provient du fait que, à l'image du verbe écrire (« pisać ») cité précédemment, une notion de culmination est associée à chacun d'eux. C'est pourquoi tous les verbes de cette classe ont deux formes perfectives, l'une traduisant la fin du processus avec culmination, l'autre signifiant la fin du processus sans culmination. Pour finir, citons le cas de la classe 1 qui est un peu plus complexe. En effet, cette classe regroupe deux types de verbes : les verbes d'état tels que « kochać » (aimer) vu précédemment, et des verbes de « transition graduelle » tels que « madrzec » (devenir sage). Les verbes d'état n'ont qu'une seule forme perfective dont la lecture est, comme nous l'avions signalé, inchoative. La forme perfective des transitions graduelles, en revanche, est plus traditionnelle : elle marque la fin de la transition graduelle, un état où cette transition est complète. En d'autres termes, nous pourrions dire que la forme perfective des transitions graduelles décrit le premier instant du temps à partir duquel la propriété vers laquelle on tend pendant la transition est vraie. Ainsi « zmadrzec », forme perfective de « madrzec ») (devenir sage) peut se traduire par « avoir fini de devenir sage » ou plus simplement « être devenu sage ». Notons que cette interprétation permet de mieux comprendre à la fois pourquoi les transitions graduelles et les verbes d'état sont regroupés au sein d'une même classe, et pourquoi la lecture des formes perfectives des transitions graduelles est *terminative*, tandis que celle des formes perfectives des verbes d'état est inchoative.

Nous ne poursuivrons pas plus avant cette description des propriétés sémantiques des classes verbales dégagées par Aalstein, laissant au lecteur désireux d'en apprendre d'avantage le soin de se reporter à (Młynarczyk, 2004). Pour notre part, nous allons décrire dans la fin de cette section un travail présenté dans (Alstein et Blackburn, 2007) et qui prolonge la thèse de Młynarczyk (Aalstein).

Dans (Alstein et Blackburn, 2007), les idées de (Młynarczyk, 2004) sont développées en utilisant les événements, ces entités abstraites que nous avions introduites à la section 1.2.1. Rappellons que les événements sont des objets formels utilisés pour rendre compte des actions décrites par les verbes. Chaque occurrence d'un verbe introduit un événement, qui est ensuite utilisé pour garder trace de toutes les informations sémantiques se rapportant à l'action décrite par cette occurrence du verbe. Par exemple, si l'unique occurrence du verbe « eat » dans la phrase « Vincent eats a burger. » introduit l'événement e, alors le fait que l'entité qui mange e est Vincent peut être représenté formellement par le littéral agent(e, vincent, tandis que le fait que l'entité mangée par

Vincent est un burger peut être représenté par la formule patient(e, x), où x est une variable liée par un quantificateur existentiel et telle que burger(x) est vrai. Une représentation du sens de la phrase précédente pourrait donc être :

$$\exists e.\exists x.\text{burger}(x) \wedge \text{eat}(e) \wedge \text{agent}(e, \text{Vincent}) \wedge \text{patient}(e, x)\,.$$

Comme on peut le constater, l'opération principale utilisée pour rendre compte du sens est la conjonction.

Les représentations des verbes construites à l'aide d'événements, telles que celle que nous venons d'introduire, rendent l'ajout d'information concernant une action particulièrement aisé. En effet, pour chaque modificateur ajoutant une nouvelle information, il suffit d'ajouter un littéral encodant cette information à la conjonction préexistante. L'encodage de l'information peut se faire à l'aide d'un prédicat binaire, comme c'est le cas pour l'agent et le patient, mais aussi par le biais d'un prédicat d'arité différente. Les adverbes peuvent par exemple être représentés par des prédicats unaires sur les événements.

Pour rendre compte du contenu temporel et aspectuel, (Alstein et Blackburn, 2007) propose d'ajouter aux événements une machinerie formelle permettant de faire référence au temps. On a besoin de pouvoir localiser les événements dans le temps (c'est-à-dire d'exprimer qu'ils ont lieu dans le passé, le présent ou le futur), et, pour les événements qui ont une durée, de parler du moment où ils commencent, des moments pendant lesquels ils se poursuivent et du moment où ils se terminent. Pour ce faire, (Alstein et Blackburn, 2007) utilise les constantes et prédicats suivants, qui avaient été introduits dans (Blackburn et al., 1993) :

- now est une constante représentant un instant particulier dans le temps, à savoir celui où le discours est énoncé, ce qui revient à dire que cette constante est utilisée pour représenter l'instant présent ;
- lt est une relation binaire sur les instants du temps. L'expression $\text{l}t(t_1, t_2)$ signifie que t_1 est antérieur à t_2. Dans la suite, nous écrirons souvent $t_1 < t_2$ plutôt que $\text{l}t(t_1, t_2)$. Ainsi, $t < \text{n}ow$ signifie que l'instant t appartient au passé, tandis que $\text{n}ow < t$ signifie que t appartient au futur. Notons enfin que l'égalité entre instants est autorisée, en vertu de quoi l'expression $t = \text{n}ow$ spécifie que t est l'instant présent ;
- i$nception(e, t)$ signifie que l'événement e commence à avoir lieu à l'instant t ;
- $conc(e, t)$ signifie que l'événement e se termine à l'instant t ;
- i$nduration(e, t)$ signifie que l'événement e est en train de se produire à l'instant t ;
- $culminates(e)$ spécifie que l'événement e culmine.

Cette ontologie permet à Aalstein et Blackburn de proposer des représentations des verbes polonais sous la forme de λ-termes « à la Montague », bien que *non typés*. Pour ce faire, les auteurs remarquent que les formes imperfectives peuvent se traduire en exprimant que l'événement qu'elles dénotent est en cours, tandis que les formes perfectives (celles à lecture inchoative exceptées) stipulent que l'événement auxquelles elles font référence se termine au moment indiqué par l'énoncer. En s'appuyant sur cette hypothèse, les auteurs proposent la représentation suivante pour le verbe « spacerować » (être en train de marcher) :

$$\lambda y.\lambda t.\lambda e.(\text{spacerowac}(e) \wedge \text{agent}(e, y) \wedge \text{induration}(e, t))\,.$$

Mis à part l'ajout de deux abstractions sur les variables t et e (correspondant respectivement à une variable sur les instants et à une variable sur les événements), cette représentation est en

tout point semblable à la représentation habituellement affectée aux verbes intransitifs dans les systèmes « à la Montague ».

Pour construire la représentation d'une phrase telle que « Piotr spacerowal » (Pierre était en train de marcher), les auteurs affectent à Piotr sa représentation usuelle $\langle \text{Piotr} \rangle = \lambda u.uP$ et introduisent un opérateur encodant la sémantique du passé :

$$\mathbf{PAST} = \lambda\phi.\exists t.\exists e.(t < \text{now} \ \wedge (\phi te)) \, .$$

À partir de ces représentations, la sémantique de la phrase précédente est construite en deux étapes. Dans un premier temps, conformément aux habitudes montagoviennes, on commence par appliquer la représentation du groupe nominal (Piotr) à celle du groupe verbal (spacerowal). Ce faisant, on obtient :

$$\begin{aligned}
\langle \text{Piotr spacerowac} \rangle &= (\langle \text{Piotr} \rangle \langle \text{spacerowac} \rangle) \\
&= (\lambda u.uP)\langle \text{spacerowac} \rangle \\
&\rightarrow_\beta \langle \text{spacerowac} \rangle P \\
&= \lambda y.\lambda t.\lambda e.(\text{spacerowac}(e) \wedge \text{agent}(e, y) \wedge \text{induration}(e, t))P \\
&\rightarrow_\beta \lambda t.\lambda e.(\text{spacerowac}(e) \wedge \text{agent}(e, P) \wedge \text{induration}(e, t))
\end{aligned}$$

Dans un second temps, le terme que nous venons d'obtenir est passé en argument à l'opérateur encodant la sémantique du passé et dont nous avons donné la représentation sémantique ci-dessus. Après β-réduction de cette dernière expression, on obtient la représentation sémantique de la phrase :

$$\exists t \exists e(t < \text{now} \ \wedge \text{spacerowac}(e) \wedge \text{agent}(e, P) \wedge \text{induration}(e, t)) \, .$$

Cet exemple illustre la méthode proposée par Aalstein et Blackburn pour construire compositionnellement des représentations sémantiques à base d'événements pour les classes verbales dégagées dans (Młynarczyk, 2004). Schématiquement, cette méthode procède en deux temps. D'abord, on construit la sémantique d'une phrase au sein de laquelle le verbe n'est pas conjugué. La représentation d'une telle phrase est une fonction prenant en argument un événement et un instant du temps et renvoyant une proposition. Dans un deuxième temps, la sémantique de la phrase d'origine (où le verbe était conjugué) est obtenue en appliquant un opérateur encodant le temps sémantique du verbe à la représentation sémantique précédente. Nous avons montré à quoi ressemble un tel opérateur, à savoir le terme **PAST**. À cet opérateur, il convient d'en ajouter deux autres pour que le système de construction sémantique soit complet, l'un pour le présent, l'autre pour le futur. Or, nous avions vu précédemment que le présent sémantique s'exprimait en conjuguant au présent (« morphologique ») une forme verbale imperfective, et que le futur sémantique était obtenu par une conjugaison au présent « morphologique » d'un verbe perfectif. C'est pourquoi Aalstein et Blackburn ajoutent à l'opérateur précédent les deux opérateurs suivants :

$$\mathbf{PRES}_{\text{impf}} = \lambda\phi.\exists t.\exists e.(t = \text{now} \ \wedge (\phi te)) \, ,$$

$$\mathbf{PRES}_{\text{perf}} = \lambda\phi.\exists t.\exists e.(\text{now} \ < t \wedge (\phi te)) \, .$$

Pour terminer cette présentation de la proposition de Aalstein et Blackburn concernant la construction sémantique, il nous paraît important de signaler que cette approche est non déterministe. En effet, comme nous l'avions remarqué précédemment, un groupe nominal peut être constitué d'un nom seul, c'est-à-dire non accompagné d'un déterminant. Nous avions expliqué que, dans ce cas, il y avait deux lectures possibles pour le groupe nominal considéré : l'une définie, l'autre indéfinie. Aalstein et Blackburn rendent compte de cette réalité linguistique en fournissant les deux règles suivantes permettant de construire la sémantique d'un groupe nominal à partir de celle du nom qui en est le seul constituant : $\langle \text{NP} \rangle = \langle \text{CI} \rangle \langle \text{N} \rangle$ et $\langle \text{NP} \rangle = \langle \text{CD} \rangle \langle \text{N} \rangle$, où CI représente l'indéfini contextuel et CD le défini contextuel. L'indéfini contextuel (c'est-à-dire le déterminant indéfini) a sa sémantique habituelle, à savoir

$$\langle \text{CI} \rangle = \lambda P.\lambda Q.\exists x.(Px \wedge Qx)\,.$$

Quant à CD, le défini contextuel ou article défini, les auteurs lui associent le terme suivant :

$$\langle \text{CD} \rangle = \lambda P.\lambda Q.\exists y.(\forall x(Px \leftrightarrow x = y) \wedge Qy)\,.$$

La variable P correspond à la représentation du nom auquel le déterminant défini est adjoint. La variable Q est la propriété que les groupes nominaux prennent habituellement en argument dans les représentations « à la Montague ». Le terme précédent affirme qu'il existe un unique y ayant la propriété P, et que y a aussi la propriété Q. Ceci est, nous semble-t-il, une traduction fidèle de l'intuition que l'on peut avoir sur les déterminants définis.

Considérons par exemple la phrase « Piotr pisal list » (Pierre était en train d'écrire une/la lettre). Dans cette phrase, « list » est à la fois un substantif et un groupe nominal, auquel Aalstein et Blackburn proposent d'associer deux lectures : l'une indéfinie, l'autre définie. Par conséquent, il y aura aussi deux lectures possibles pour la phrase tout entière. En des termes plus abstraits, cela signifie que, dans l'approche de Aalstein et Blackburn, la construction sémantique n'est pas une opération fonctionnelle (*i.e.* qui à tout texte associe une unique représentation), mais une opération relationnelle qui, à un texte donné peut associer plusieurs représentations sémantiques.

Il convient enfin de remarquer que la discussion de la construction sémantique et des représentations à base d'événements qui est menée dans (Alstein et Blackburn, 2007) et que nous venons de présenter n'est pas très détaillée. En particulier, les auteurs présentent leurs idées dans le cadre du λ-calcul *non typé* et ils ne font que donner une intuition des propriétés que devraient avoir les événements qu'ils utilisent, ces propriétés n'étant donc pas exposées de façon formalisée. Dans le reste de ce chapitre, nous allons nous attacher à développer ces idées dans TYn, ce qui, comme nous l'avons laissé entendre en section 1.2.2, se résume à deux tâches principales. Dans un premier temps, il nous faut mettre au point des entrées lexicales pour le polonais et montrer comment celles-ci peuvent être utilisées avec `Nessie`. Puis, dans un deuxième temps, nous serons amenés à axiomatiser la structure d'événements sous-jacente à la proposition de (Alstein et Blackburn, 2007). Nous nous consacrons dès à présent à la première de ces tâches.

4.3 Sémantique des verbes polonais dans TYn

Nous allons maintenant expliquer comment nous proposons d'automatiser la construction de représentations sémantiques pour le fragment de polonais considéré par Aalstein et Blackburn, à

l'aide d'une DCG et de `Nessie`. Nous commençons par introduire le typage et les modifications qu'il nous conduit à apporter à la représentation des verbes. Puis, nous expliquons ce qui nous a conduit à passer de TY3 à TY4. Enfin, nous décrivons l'implantation que nous proposons pour la construction sémantique.

4.3.1 Typage et représentation sémantiques des verbes

Notre approche, bien qu'elle s'inspire largement de celle qui vient d'être présentée, s'en démarque cependant sur plusieurs points. En premier lieu, remarquons que la méthode de construction sémantique que nous venons de présenter utilise le λ-calcul non typé. Pour notre part nous allons, comme nous l'avons fait au chapitre précédent, utiliser le λ-calcul typé. Plus précisément, la version de TYn qui va nous intéresser ici est TY3. En plus du type t des valeurs booléennes, toujours présent, les trois autres types de base que nous introduisons sont $entity$ représentant les entités, $time$ représentant les instants du temps et $event$ représentant les événements. Le type $entity$ est à considérer comme un équivalent du type e que nous avions utilisé au chapitre précédent.

Ce passage du λ-calcul non typé au λ-calcul typé, bien qu'important d'un point de vue conceptuel, est trivial d'un point de vue pratique, puisqu'il consiste simplement à affecter un type aux symboles constants constituant l'ontologie et aux variables apparaissant dans les représentations sémantiques. Pour ce qui est de l'ontologie, nous avions introduit la constante now pour représenter le présent. Il paraît donc assez naturel de la considérer comme un objet de type time. De la même façon et compte tenu de la signification que nous avons proposée précédemment pour les symboles de l'ontologie, on obtient les jugements de typage suivants :

- `lt : time -> time -> t;`
- `inception, induration, conc : event -> time -> t;`
- `culminates : event -> t;`
- `agent, patient : event -> entity -> t.`

Ces affectations de types aux symboles de l'ontologie permettent d'associer un type aux termes qui ont été proposés pour représenter les symboles lexicaux. Considérons par exemple le cas du verbe « spacerować » (être en train de marcher). Comme nous l'avions vu précédemment, la représentation proposée par Aalstein et Blackburn pour ce verbe est :

$$\lambda y.\lambda t.\lambda e.(\text{spacerowac}(e) \wedge \text{agent}(e, y) \wedge \text{induration}(e, t))$$

Voyons quels types nous pouvons associer aux variables. La variable y apparaît comme second argument de agent, donc son type ne peut être que entity. Des arguments similaires permettent de déduire que e est de type event et que t est de type time, ce qui correspond bien à ce que nous avions remarqué lorsque nous avons introduit cette représentation, le typage ne faisant que rendre explicite une information qui est déjà présente de manière implicite dans les termes non typés. Nous pouvons maintenant écrire le type du terme représentant le verbe « spacerować » :

$$\text{entity} \ \rightarrow \text{time} \ \rightarrow \text{event} \ \rightarrow t\,.$$

Or, nous avions vu précédemment que la représentation du groupe nominal « Piotr » était donnée par

$$\langle \text{Piotr} \rangle = \lambda u.uP$$

et que, pour obtenir la sémantique de la phrase « Piotr spaceruje », il fallait commencer par appliquer la représentation du verbe à cette dernière représentation. Nous pouvons en déduire que le type de la variable u doit être égal à celui du groupe verbal, ce qui nous donne le type du terme représentant Piotr :

$$(\text{entity} \rightarrow \text{time} \rightarrow \text{event} \rightarrow t) \rightarrow t\,.$$

On remarque que, bien que les objets de type time et event concernent *a priori* seulement le groupe verbal, leur introduction nécessite aussi une modification des types sémantiques associés à d'autres catégories syntaxiques, comme les noms propres. De la même façon, dans le cas d'un groupe nominal constitué d'un nom seul, il faudrait modifier la représentation du nom ou du déterminant « contextuel » de sorte que le type résultant soit le même que celui que nous venons de présenter. Donc, si l'on souhaitait ajouter les notions d'événements et de temps à des représentations sémantiques préexistantes comme celles construites au chapitre précédent, le faire en utilisant la représentation des verbes proposée par Aalstein et Blackburn nécessiterait des changements dans les types (et donc dans les termes) représentant des objets autres que les verbes, tels que par exemple les noms propres, comme nous l'avons expliqué plus haut. Avec l'approche de Aalstein et Blackburn, le passage d'une représentation sémantique dans TY1 à une représentation dans TY3 ne peut donc pas se faire en se contentant de ne modifier que quelques représentations sémantiques. Ce passage nécessite, au contraire, la modification de beaucoup de représentations sémantiques. En cela, cette représentation est, pensons-nous, hautement non-locale.

Pour en revenir à notre exemple, l'application $(\langle\text{Piotr}\rangle\langle\text{spacerowac}\rangle)$ produit un terme de type time $\rightarrow$ event $\rightarrow t$, c'est-à-dire, comme nous l'avions indiqué précédemment, une fonction prenant en argument un instant du temps et un événement et renvoyant une proposition. C'est une telle fonction que les opérateurs de temps $\mathbf{PAST}$, $\mathbf{PRES}_{\text{impf}}$ et $\mathbf{PRES}_{\text{perf}}$ prennent en premier argument et à partir de laquelle ils construisent une proposition, en quantifiant sur le temps et l'événement et en passant les variables ainsi liées à la fonction qu'ils reçoivent.

Le fait de passer ainsi en revue les étapes permettant de construire la représentation de la phrase précédente, mais cette fois dans un contexte avec types, met en lumière un fait qui, sans les types, serait peut-être passé inaperçu, ou, du moins, dont les conséquences ne seraient pas apparues de manière aussi claire. Nous voulons parler du fait que, dans la proposition de Aalstein et Blackburn, le traitement de la sémantique des verbes est non local, mais cette fois du point de vue de l'arbre syntaxique. En effet, dans cette approche, seule l'information portée par la forme infinitive du verbe est encodée directement dans le λ-terme qui le représente. Nous pouvons même ajouter que l'information d'aspect n'est pas entièrement prise en compte par le terme représentant le verbe. Plus précisément, nous avions vu que l'aspect morphologique se traduit par un aspect sémantique mais qu'il a aussi un impact sur le temps sémantique. Or, les λ-termes proposés par Aalstein et Blackburn pour représenter les verbes ne tiennent compte que de l'aspect sémantique dérivé de l'aspect morphologique. D'une façon ou d'une autre, l'information d'aspect doit « remonter » dans l'arbre syntaxique, jusuqu'au moment où elle pourra être utilisée, en conjonction avec le temps du verbe (qui doit « remonter l'arbre syntaxique » lui aussi), pour choisir le bon opérateur temporel parmi les trois qui ont été introduits. En d'autres termes, lorsqu'un verbe est rencontré, la représentation sémantique qui lui est assignée par Aal-

stein et Blackburn ne rend pas compte de toute l'information sémantique qu'il contient. Il est nécessaire de garder la trace à la fois de son temps et de son aspect, jusqu'au moment où ces deux informations pourront être utilisées pour choisir un opérateur qui sera appliqué à la représentation sémantique déjà construite pour la phrase. Ceci nous conduit à penser que la méthode de construction sémantique proposée par Aalstein et Blackburn n'est pas complètement compositionnelle. En effet, une approche compositionnelle implique, nous semble-t-il, que toute l'information sémantique portée par un lexème soit restituée par le λ-terme qui le représente, ce qui n'est pas le cas pour les verbes conjugués dans l'approche de Aalstein et Blackburn.

Nous allons donc proposer une autre représentation sémantique pour les verbes conjugués, représentation qui sera à la fois complète (dans le sens où toute l'information sémantique portée par le verbe conjugué sera restituée par sa représentation) et locale, dans le sens où cette représentation permettra d'incorporer les notions de temps et d'événements à un système de construction sémantique préexistant tel que celui présenté au chapitre précédent, sans avoir pour cela à modifier toutes les représentations sémantiques existantes. En réalité, notre proposition consiste en une légère modification de celle qui a été présentée jusqu'ici. Dans cette dernière, les opérateurs encodant le temps sémantique sont envisagés comme des fonctions prenant en argument la représentation d'une phrase avec verbe à l'infinitif et renvoyant une représentation qui est un enrichissement de celle de la phrase par les informations temporelles. De notre côté, nous proposons de garder la notion d'opérateurs encodant la sémantique du temps mais, contrairement à ce que proposent Aalstein et Blackburn, nous proposons que ces opérateurs s'appliquent directement à la représentation du verbe, et non à celle de la phrase tout entière. Les opérateurs de temps que nous proposons d'utiliser sont donc des fonctions prenant en argument un terme représentant un verbe à l'infinitif et renvoyant une représentation complète d'un point de vu temporel et aspectuel. Pour en revenir au verbe « spacerować », nous avions vu que le type du terme le représentant était :

$$\text{entity} \ \rightarrow \text{time} \ \rightarrow \text{event} \ \rightarrow t\,.$$

C'est donc de ce type que doit être l'argument des opérateurs encodant la sémantique du temps. Quant au type de leur résultat, nous souhaitons qu'il ne comporte ni temps ni événement, et qu'il soit analogue à celui donné aux verbes dans TY1 au chapitre précédent. Nous cherchons donc à construire un objet de type entity $\rightarrow t$. Un λ-terme possible pour transformer la représentation de l'infinitif « spacerować » en une représentation du verbe conjugué au passé est :

$$\lambda V.\lambda x.\exists t.\exists e.(t < \text{now}) \wedge (Vtex)$$

où V est la variable qui sera remplacée par la représentation du verbe à l'infinitif. Comme on peut le constater, ce λ-terme ne porte aucune information spécifique au verbe « spacerować ». Par conséquent, il peut être utilisé pour d'autres verbes. Plus précisément, ce λ-terme convient à tous les verbes intransitifs, c'est pourquoi nous le noterons $\mathbf{PASTIV}$ par la suite. En effet, les verbes transitifs ayant une représentation d'un autre type, le même opérateur $\mathbf{PASTIV}$ ne saurait être utilisé pour passer d'une forme infinitive d'un verbe transitif à sa forme conjuguée. Nous avons donc besoin d'un autre opérateur pour manipuler les verbes transitifs. Pour le trouver, nous partons de la représentation proposée par Aalstein et Blackburn pour le verbe « pisać » (être en train d'écrire), qui est bien un verbe transitif. La sémantique proposée par les auteurs pour ce

verbe est :

$$\lambda w.\lambda y.(w\lambda z.\lambda t.\lambda e.(\text{pisac}(e) \wedge \text{agent}(e, y) \wedge \text{patient}(e, z) \wedge \text{induration}(e, t)))\,.$$

Les types des variables y, z, t et e sont faciles à trouver. Les deux premières sont de type entity, la troisième de type event et la quatrième de type time. Il reste le cas de w. Nous constatons que w est appliqué à un terme de type entity $\rightarrow$ time $\rightarrow$ event $\rightarrow t$, et que l'application de w à ce terme doit retourner une valeur de type t. On en déduit que w est de type

$$(\text{entity} \rightarrow \text{time} \rightarrow \text{event} \rightarrow t) \rightarrow t$$

ce qui donne le type suivant pour la représentation des verbes transitifs :

$$((\text{entity} \rightarrow \text{time} \rightarrow \text{event} \rightarrow t) \rightarrow t) \rightarrow \text{entity} \rightarrow t\,.$$

Nous remarquons que pour les verbes transitifs, les arguments de type time et event sont fournis *après* les arguments de type entity. Ceci est assez naturel, puisque dans l'approche de Aalstein et Blackburn les phrases sont vues (dans un premier temps) comme des fonctions attendant temps et événement en argument pour fournir une proposition. Cependant, puisque nous cherchons à écrire un opérateur transformant la représentation infinitive du verbe transitif en une représentation conjuguée qui n'attend plus ni temps ni événement en arguments, nous ne pourrons pas utiliser les représentations proposées pour les verbes transitifs. Nous allons donc les modifier légèrement, de sorte à pouvoir, ensuite, écrire des opérateurs à même de les transformer. Nous proposons donc la sémantique suivante pour le verbe « pisać » :

$$\lambda t.\lambda e.\lambda K.\lambda x.(K\lambda y.\text{pisac}(e) \wedge \text{agent}(e, x) \wedge \text{patient}(e, y) \wedge \text{induration}(e, t))\,.$$

Cette représentation a été obtenue très simplement, en partant de celle utilisée pour les verbes transitifs au chapitre précédent. Les seules transformations qui ont été nécessaires ont été l'ajout en tête de deux abstractions sur le temps et l'événement, et, dans la conjonction, l'adaptation à la nouvelle ontologie. Nous pouvons maintenant donner un opérateur permettant de mettre un verbe transitif au passé :

$$\mathbf{PASTTV} = \lambda V.\lambda u.\lambda y.\exists t.\exists e.(t < \text{now}) \wedge (Vteuy)\,.$$

En ajoutant 4 autres opérateurs aux 2 déjà définis (pour le présent des verbes imperfectifs intransitifs et transitifs, et pour leurs équivalents perfectifs), on obtient un système où la sémantique d'un verbe conjugué est obtenue de façon systématique à partir de la sémantique de sa forme à l'infinitif et d'un opérateur encodant le temps sémantique. La représentation sémantique ainsi construite ne fait plus intervenir d'abstractions ni sur le temps, ni sur les événements, ce qui en facilite l'intégration à un système de construction sémantique préexistant, puisque les représentations sémantiques des autres catégories syntaxiques peuvent être réutilisées telles qu'elles ont été définies au chapitre précédent. Certes, il a été nécessaire pour obtenir ce résultat de multiplier par 2 le nombre d'opérateurs, et il faudrait en ajouter encore si l'on souhaitait, par exemple, traiter les verbes ditransitifs. Nous pensons toutefois que cet inconvénient est mineur, compte tenu des avantages de la nouvelle approche de la construction sémantique que nous venons de présenter. En effet, en plus du fait que les représentations sémantiques ainsi construites pour les

verbes sont plus faciles à intégrer à un système sémantique préexistant que celles construites par Aalstein et Blackburn, il faut noter qu'elles corrigent un autre problème : toute l'information temporelle et aspectuelle contenue dans les verbes est exploitée localement. Avec ces nouvelles représentations, il n'est plus nécessaire de faire remonter l'information sémantique le long de l'arbre syntaxique pour l'exploiter plus tard. À ce titre, nous avons gagné aussi en compositionnalité.

Pour clore cette discussion théorique sur le calcul compositionnel de représentations sémantiques à base d'événements, rappelons que, lorsque nous avions introduit les événements, nous avions signalé que l'un de leurs avantages était qu'ils permettaient de stocker facilement autant d'information que nécessaire sur une action. Nous avions notamment donné l'exemple des adverbes dont la contribution sémantique peut être envisagée comme un littéral à ajouter à une conjonction de propriétés d'événements. On pourrait alors remarquer qu'avec des représentations sémantiques telles que celles que nous venons d'introduire et qui « cachent » les événements en limitant leur accès au seul verbe, il est difficile pour d'autres composants d'y faire référence. On pourrait même ajouter que, de ce point de vue, les représentations sémantiques proposées par Aalstein et Blackburn sont plus intéressantes, puisque les événements sont intégrés plus profondément aux représentations sémantiques. Au chapitre suivant, nous introduirons une méthode générique, basée sur les continuations, pour passer de l'information sémantique d'une représentation à l'autre. Bien que nous ne le fassions pas dans cette thèse, on pourrait envisager d'utiliser cette méthode à base de continuations pour rendre accessibles les événements depuis d'autres représentations sémantiques que celles qui les introduisent et résoudre ainsi le problème que nous venons d'évoquer.

4.3.2 Passage de TY3 à TY4

La sémantique de certaines classes de verbes telles que les culminations (dont le verbe « pisać » (être en train d'écrire) fait partie) est exprimée à l'aide de deux événements. Le premier, présent pour tous les verbes, est un événement non instantané qui correspond au processus décrit par le verbe, par exemple au processus d'écriture. Le second, dont on dit qu'il est la *culmination* du premier, n'est introduit que lorsque le processus décrit par le verbe se va jusqu'à son terme naturel (culmine). Ce second événement, qui est instantané, peut être compris comme un marqueur de la complétion de l'action décrite par le verbe.

Bien sûr, il existe entre les deux événements des liens forts : ils ont le même agent et le même patient, et surtout, ils dérivent du même verbe. Ainsi, si la sémantique du verbe « napisać » (avoir complètement terminé d'écrire) est restituée par les deux événements E_1 et E_2, on s'attend à ce que les littéraux p$isac(E_1)$ et p$isac(E_2)$ soient vrais.

L'une des possibilités dont on dispose pour parvenir à ce résultat est de modifier les représentations sémantiques des verbes tels que « napisać » de sorte qu'elles introduisent deux événements au lieu d'un. Il serait alors facile de faire en sorte que le prédicat pisac soit vrai pour les deux événements.

Cependant, cette solution ne nous paraît pas satisfaisante. En effet, elle rend plus complexes des représentations sémantiques qui, à notre avis, le sont déjà et nécessite en outre la modification de chaque représentation sémantique.

Au lieu de procéder ainsi, nous proposons d'ajouter un type correspondant aux verbes à notre

logique, passant ainsi de TY3 à TY4. Ce type, que nous appellerons kind, permet de réifier le type des événements. Nous nous donnons en outre une relation binaire notée ek et qui permet de lier un événement à sa sorte. En d'autres termes, le littéral $ek(E, K)$ permet d'exprimer que l'événement E est de sorte K.

Pour parachever la transition de TY3 vers TY4, nous modifions le type des constantes représentant les verbes, ainsi que les représentations de ces derniers. Auparavant, les verbes étaient représentés par des prédicats sur les événements. Nous les représentons maintenant par des constantes de type kind et modifions leur représentation de sorte que le littéral verb(E) soit remplacé par le littéral ek$(E,$ verb$)$. Ainsi, la nouvelle représentation du verbe « spacerować » s'écrit :

$$\lambda y \lambda t \lambda e(\text{ek}(e, \text{spacerowac}) \wedge \text{agent}(e, y) \wedge \text{induration}(e, t)) .$$

Grâce à cette réification de la sorte des événements, il devient possible de retrouver, à partir d'un événement, le verbe dont il provient. Ceci nous permet par exemple d'écrire des propriétés comme :

> Si e_2 est la culmination de e_1 (ce que nous noterons par la suite culm(e_1, e_2)), alors il existe k tel que $ek(e_1, k$ et $ek(e_2, k)$.

Bien sûr, il serait impossible d'énoncer de telles propriétés si nous ne disposions pas du type kind et de la relation ek.

4.3.3 Implantation

Les outils utilisés ici sont essentiellement les mêmes que ceux vus au chapitre précédent, à savoir une DCG pour l'analyse syntaxique et `Nessie` pour le calcul de la représentation sémantique, à partir d'un lexique et d'un arbre syntaxique. Nous n'allons donc pas donner une présentation approfondie de la façon dont les représentations sémantiques sont construites, préférant nous concentrer sur quelques détails intéressants et, à notre avis, suffisants pour donner une idée du travail réalisé.

Le lexique de `Nessie` commence par déclarer tous les types et toutes les constantes nécessaires. Voici quelques déclarations :

```
type entity;
type event;
type time;
type kind;
const now : time;
const lt : time -> time -> t;
const agent, patient : event -> entity -> t;
const inception, induration, conc : event -> time -> t;
const ek : event -> kind -> t;
const culminates : event -> t;
const culm : event -> event -> t;
```

Le lexique se poursuit par la déclaration d'une famille ouverte `op` contenant tous les opérateurs temporels que nous venons d'introduire. Voici la déclaration de la famille et de ses différents lemmes :

```
family op; # Operators for tense and aspect

lemma presimpiv {
  family = op;
  term =
  lam verb : (time -> event -> entity -> t).
  lam x : entity.
  exists t : time. exists e : event.
  ( t=now && verb(t,e, x) )
};

lemma presperfiv {
  family = op;
  term =
  lam verb : (time -> event -> entity -> t).
  lam x : entity.
  exists t : time. exists e : event.
  ( lt(now,t) && verb(t, e, x) )
};

lemma pastiv {
  family = op;
  term =
  lam verb : (time -> event -> entity -> t).
  lam x : entity.
  exists t : time. exists e : event.
  ( lt(t,now) && verb(t,e,x) )
};

lemma presimptv {
  family = op;
  term =
  lam verb : time -> event -> ((entity -> t) -> t) -> entity -> t.
  lam u : ((entity -> t) -> t). lam y : entity.
  exists t : time. exists e : event.
  ( t=now && verb(t, e, u, y) )
};

lemma presperftv {
  family = op;
  term =
  lam verb : time -> event -> ((entity -> t) -> t) -> entity -> t.
  lam u : ((entity -> t) -> t). lam y : entity.
  exists t : time. exists e : event.
  ( lt(now,t) && verb(t, e, u, y) )
};

lemma pasttv {
  family = op;
```

```
  term =
  lam verb : time -> event -> ((entity -> t) -> t) -> entity -> t.
  lam u : ((entity -> t) -> t). lam y : entity.
  exists t : time. exists e : event.
  ( lt(t,now) && verb(t, e, u, y) )
};
```

Remarquons que les représentations associées aux lemmes de cette famille ont des types différents, ce qui constitue une nouveauté par rapport aux familles fermées que nous avons vues jusqu'ici et dont tous les lemmes étaient représentés par des termes d'un même type.

Pour compléter la définition des représentations sémantiques des verbes, nous commençons par déclarer une constante par verbe, puis nous définissons toutes les formes infinitives possibles pour chaque verbe. Nous donnons ici les définitions des verbes construits à partir de « kochać » (être en train d'aimer), « spacerować » (être en train de se promener) et « pisać » (être en train d'écrire) :

```
const pisac, spacerowac, kochac : kind;
family iv;
family tv;
lemma pisac { # to write, imperfective
  family = tv;
  term =
  lam t : time. lam e : event.
  lam K:(entity->t)->t . lam x:entity .
  (K @ lam y:entity .
    ( ek(e,pisac) &&
      agent(e,x) &&
      patient(e,y) &&
      induration(e,t)
    )
  )
};

lemma napisac { # to write, perfective
  family = tv;
  term =
  lam t : time. lam e : event.
  lam K:(entity->t)->t . lam x:entity .
  (K @ lam y:entity .
    ( ek(e,pisac) &&
      culminated(e) &&
      agent(e,x) &&
      patient(e,y) &&
      conc(e,t)
    )
  )
};

lemma popisac { # to write, perfective
```

```
  family = tv;
  term =
  lam t : time. lam e : event.
  lam K:(entity->t)->t . lam x:entity .
  (K @ lam y:entity .
    ( ek(e,pisac) &&
      (! (culminated(e)) ) &&
      agent(e,x) &&
      patient(e,y) &&
      conc(e,t)
    )
  )
};

lemma spacerowac { # to walk, imperfective
  family = iv;
  term = lam t : time. lam e : event.
  lam y : entity.
  ( ek(e,spacerowac) &&
    agent(e,y) &&
    induration(e,t)
  )
};

lemma pospacerowac { # to walk, perfective
  family = iv;
  term =
  lam t : time. lam e : event. lam y : entity.
  ( ek(e,spacerowac) &&
    agent(e,y) &&
    conc(e,t)
  )
};

lemma kochac { # to love, imperfective
  family = tv;
  term =
  lam t : time. lam e : event.
  lam K:(entity->t)->t . lam x:entity .
  (K @ lam y:entity .
    ( ek(e,kochac) &&
      agent(e,x) &&
      patient(e,y) &&
      induration(e,t)
    )
  )
};

lemma pokochac { # to love, perfective
```

```
  family = tv;
  term =
  lam t : time. lam e : event.
  lam K:(entity->t)->t . lam x:entity .
  (K @ lam y:entity .
    ( ek(e,kochac) &&
      agent(e,x) &&
      patient(e,y) &&
      inception(e,t)
    )
  )
};
```

Comme le montrent ces définitions, toutes les formes d'un même verbe utilisent la même constante (pisac, spacerowac ou kochac). On remarque aussi une différence avec le traitement des verbes anglais vu au chapitre précédent : ici les familles des verbes intransitifs et transitifs sont fermées, alors qu'elles étaient ouvertes à la section précédente. Le choix de distinguer ces deux familles, plutôt que, par exemple, de définir une famille par classe de verbe est lié au typage : du point de vue des types, deux verbes transitifs de classes différentes d'après la classification de Aalstein et Blackburn sont plus proches l'un de l'autre qu'un verbe transitif et un verbe intransitif appartenant à une même classe.

En ce qui concerne les autres familles et lemmes, le lexique que nous utilisons ici est en tout point similaire à celui que nous avons utilisé au chapitre précédent pour l'anglais. Il en va de même pour la DCG utilisée pour l'analyse syntaxique. Tout comme celle vue au chapitre précédent, celle que nous utilisons pour le polonais comporte, en plus des traits syntaxiques, un attribut `ast:` qui est utilisé pour construire un arbre pendant l'analyse de la phrase. C'est cet arbre qui sera transmis à `Nessie`.

Du point de vue de la DCG, la seule nouveauté par rapport à celle du chapitre précédent concerne le traitement des verbes. Il convient en effet de construire une représentation sémantique utilisant la sémantique de la forme infinitive et l'opérateur temporel, problème qui ne se posait pas précédemment. Pour ce faire, nous commençons par définir une règle choisissant le bon opérateur temporel, en fonction des informations morphologiques portées par le verbe et qui ont été extraites du lexique pendant l'analyse syntaxique. Voici à quoi ressemblent ces règles :

```
tenseAspect(past, _, iv, pastiv).
tenseAspect(past, _, tv, pasttv).
tenseAspect(pres, imp, iv, presimpiv).
tenseAspect(pres, imp, tv, presimptv).
tenseAspect(pres, perf, iv, presperfiv).
tenseAspect(pres, perf, tv, presperftv).
```

Puis, les règles de DCG reconnaissant un verbe et construisant son arbre syntaxique associé sont les suivantes :

```
vp([ast:Ast,gender:Gender]) -->
  iv([ast:AstIV,aspect:Aspect,tense:Tense,gender:Gender]),
  {tenseAspect(Tense,Aspect,iv,TAOperator)},
```

```
  {Ast=binary(vp,leaf(TAOperator,op),AstIV)}.

vp([ast:Ast,gender:Gender]) -->
  tv([ast:AstTV,aspect:Aspect,tense:Tense,gender:Gender]),
  np([ast:AstNP,case:acc,gender:_,rc:_]),
  {tenseAspect(Tense, Aspect, tv, TAOperator)},
  {Ast=binary(vp,binary(vp,leaf(TAOperator, op), AstTV),AstNP)}.
```

Ces deux règles on un fonctionnement similaire. Elles commencent par reconnaître un verbe, dont elles récupèrent l'arbre syntaxique, l'aspect morphologique et le temps morphologique. Ensuite, la seconde règle reconnaît le reste de la construction syntaxique, c'est-à-dire qu'elle attend un groupe nominal. La première règle n'a rien à faire à ce stade, puisque le verbe intransitif constitue à lui seul le groupe verbal. Puis, dans les deux règles, est déterminé l'opérateur à utiliser pour obtenir la représentation du verbe conjugué, à l'aide du prédicat `tenseAspect`. Enfin, les deux règles construisent l'arbre syntaxique correspondant au verbe conjugué. Pour mieux comprendre ce qu'il se passe, considérons l'exemple du verbe « pospaceruje » dans la phrase « Piotr pospaceruje » (Piotr se sera promené). Ce verbe est intransitif, c'est donc la première règle qui sera utilisée pour le reconnaître. En outre, comme il s'agit d'une forme perfective conjuguée au présent, l'entrée lexicale correspondante se présente comme suit :

```
lexEntry(iv,
  [ lemma:pospacerowac,
    word:[pospaceruje],
    aspect:perf,
    tense:pres,gender:_]).
```

Donc, l'appel

```
iv([ast:AstIV,aspect:Aspect,tense:Tense,gender:Gender])
```

va unifier `AstIV` avec `leaf(pospacerowac, iv)` (terme fourni par le lexique syntaxique), `Aspect` avec `perf` et `Tense` avec `pres`.

Le prédicat `tenseAspect` sera donc appelé avec comme premiers arguments `pres`, `perf` et `iv`, aboutissant à la sélection de l'opérateur `presperfiv`. Enfin, l'arbre syntaxique complet du verbe est construit, à savoir dans notre cas
`binary(vp,leaf(presperfiv, op), leaf(pospacerowac,iv)).`

L'analyse de la phrase « Piotr spaceruje » produit donc l'arbre suivant :

```
binary(s,
       unary(np, leaf(piotr, pn)),
       binary(vp,
              leaf(presperfiv, op),
              leaf(pospacerowac, iv)
       )
).
```

Lors du calcul sémantique par `Nessie`, la feuille
`leaf(presperfiv, op)` est instanciée par la représentation de l'opérateur encodant le futur

sémantique, tandis que la feuille `leaf(pospacerowac, iv)` est instanciée par la sémantique de la forme infinitive perfective de « pospacerować ». Or, comme la règle implicitement associée aux arbres binaires par `Nessie` est l'application de la représentation du sous-arbre gauche à celle du sous-arbre droit, l'arbre binaire contenant les deux feuilles précédentes permet bien de construire la sémantique du verbe, conformément à la description précédente. Compte tenu de la capacité de `Nessie` à produire des représentations en logique du premier ordre avec sortes, la représentation sémantique que nous obtenons finalement pour la phrase précédente est :

```
some(A, and(time(A), some(B, and(event(B),
and(lt(now, A), and(and(ek(B,spacerowac),
agent(B, piotr)), conc(B, A)))))))
```

Pour être tout à fait complets, nous terminons en signalant que nous avons simplifié le traitement des groupes nominaux constitués d'un nom sans déterminant. En effet, comme nous l'avions indiqué précédemment, ces constructions peuvent donner lieu à deux lectures, une définie et une indéfinie, la bonne lecture dépendant du contexte. Pour notre part, nous choisissons par défaut la lecture indéfinie, parce que les représentations sémantiques sont plus simples. En outre, comme le phénomène qui nous intéresse (à savoir le temps et l'aspect des verbes) n'a pas de relation directe avec le caractère défini ou non des groupes nominaux, cette simplification ne pose aucun problème.

4.4 Axiomatisation des événements

Comme nous l'avons remarqué en section 1.2.2, l'utilisation d'une entité abstraite ne peut se réduire au seul enrichissement du langage logique ; il faut aussi enrichir la théorie logique sous-jacente, sans quoi les symboles ajoutés seraient totalement dépourvus de signification. Aussi, bien que Aalstein et Blackburn aient donné quelques intuitions à propos de ce qu'est leur ontologie, ils n'ont pas formalisé leurs idées. Le premier objectif de cette section est de procéder à cette formalisation.

Le second objectif de cette section est de montrer que les représentations produites par `Nessie` peuvent être utilisées directement par le module d'inférence de `Curt`, et en particulier par la partie de ce module responsable de la construction de modèles. Rappelons que, comme nous l'avons vu en section 1.3, (Blackburn et Bos, 2005b) définit dans `Curt` une architecture pour l'inférence qui utilise conjointement la preuve automatique de théorèmes et la construction de modèles. Nous allons pour notre part nous concentrer sur la construction de modèles, parce qu'elle pose un problème intéressant qui n'apparaît pas dans le contexte de la preuve de théorèmes. Comme nous le verrons, alors qu'un prouveur de théorèmes n'a besoin que d'axiomes, la construction de modèles nécessite plus d'informations. Les constructeurs de modèles sont en effet conçus pour construire des modèles *minimaux*. Cependant, comme nous aurons l'occasion de le constater, les modèles minimaux perdent souvent de l'information. Nous montrerons comment calculer les modèles non minimaux pour les représentations temporelles que nous étudions ici.

4.4.1 Présentation des axiomes

Pour bien comprendre à quel point les axiomes mentionnés précédemment sont indispensables, nous pouvons utiliser la version de `Curt` à laquelle le travail de la section précédente permet d'aboutir. On obtient alors le dialogue suivant :

```
> piotr pospaceruje

Message (consistency checking): mace found a result.
Message (informativeness checking): mace found a result.
```

Ce dialogue nous apprend que la représentation sémantique de « Piotr pospaceruje » a été construite avec succès, qu'elle est consistante (c'est-à-dire non contradictoire) et informative (c'est-à-dire que ce n'est pas une tautologie). En particulier, `Curt` nous informe qu'un modèle a pu être construit pour cette représentation. Il est alors possible de demander à `Curt` quel modèle a été construit :

```
> models
D=[d1]
  f(0, c2, d1)
  f(1, time, [d1])
  f(0, c1, d1)
  f(1, event, [d1])
  f(0, now, d1)
  f(2, lt, [ (d1, d1)])
  f(0, spacerowac, d1)
  f(2, ek, [ (d1, d1)])
  f(0, piotr, d1)
  f(2, agent, [ (d1, d1)])
  f(2, conc, [ (d1, d1)])
```

Bien qu'il soit incontestable que ce modèle satisfait effectivement la représentation sémantique de la phrase initiale, ce n'est pas vraiment celui auquel on peut s'attendre. En effet, on aurait plutôt attendu un modèle où chaque objet introduit par la phrase initiale (Piotr, le temps où sa marche sera terminée, l'événement de marcher, *etc.*) serait interprété par un élément distinct du modèle. Au lieu de cela, on obtient un modèle où tous les objets sont interprétés par le même élément, quelque soit leur type.

Ce résultat, qui pourrait sembler déroutant à première vue, s'explique en fait assez facilement. En effet, les constructeurs de modèles tels que `Mace` et `Paradox` cherchent à construire des modèles *minimaux*, c'est-à-dire des modèles dont le domaine comporte le moins d'éléments possibles (parce que ces modèles sont les plus faciles à construire et que, moins il y a d'éléments dans le domaine, plus l'espace de recherche est petit). Donc, si la formule logique dont on cherche un modèle ne contient pas de contrainte explicite forçant à distinguer deux éléments l'un de l'autre, le constructeur de modèles ne les distinguera pas. Il est donc nécessaire de trouver un moyen de forcer le constructeur de modèles à construire des modèles plus réalistes, en le forçant à distinguer les éléments les uns des autres.

Déjà confrontés à ce problème dans leur ouvrage, Blackburn et Bos l'avaient résolu, comme nous l'avons mentionné à la section 1.3, en ajoutant à la formule passée aux constructeurs de

modèles (et aux prouveurs de théorèmes) une série d'axiomes formalisant les contraintes que l'on souhaite imposer. Dans la suite de cette section, nous allons faire de même. Nous serons cependant amenés à raffiner l'approche proposée par Blackburn et Bos, et à la systématiser, pour tenir compte du fait que le λ-calcul que nous utilisons est typé, ce qui n'était pas le cas dans le travail de Blackburn et Bos. Nous commençons par présenter les axiomes dont nous aurons besoin, avant de voir comment les passer aux outils d'inférence.

Nous pouvons distinguer deux groupes d'axiomes : l'un réunissant tous les axiomes en relation avec le typage, l'autre contenant des axiomes formalisant la structure du temps et les relations entre temps et événements. Par « axiomes en relation avec le typage », nous entendons des axiomes qui traduisent en logique du premier ordre des contraintes qui sont implicitement présentes lorsqu'on utilise le typage et qui n'ont donc pas besoin d'être axiomatisées tant que l'on s'intéresse à la logique d'ordre supérieur. Par exemple, il est assez naturel que des objets de types différents ne puissent être égaux, mais lorsqu'on passe en logique du premier ordre, ceci doit être explicité. De même, dans TYn, tout objet a un type et il faut donc interdire à un objet de n'avoir aucun type. En admettant que les types soient représentés par des prédicats unaires, nous pouvons résumer ces deux derniers points en disant que l'une des contraintes imposée implicitement par les types est que ceux-ci constituent une *partition* du domaine d'interprétation. Ceci se traduit par des axiomes comme :

not_event_entity	$\forall A.\neg(\text{event}(A) \wedge \text{entity}(A))$
not_entity_time	$\forall A.\neg(\text{entity}(A) \wedge \text{time}(A))$
partition	$\forall A.(\text{entity}(A) \vee \text{event}(A) \vee \text{time}(A))$

Notons tout d'abord que, dans ce tableau, nous avons utilisé des noms abrégés pour les axiomes, pour des raisons de mise en page. Nous ferons de même pour tous les axiomes à venir. Dans le code qui sera présenté, en revanche, nous avons fait le choix de conserver les noms complets, ceux-ci nous paraissant plus lisibles.

Remarquons ensuite que cette axiomatisation du partitionnement en types ne tient pas compte du type t des valeurs de vérité. Ceci provient du fait que les modèles du premier ordre ne cherchent généralement pas à interpréter ce type ou ses valeurs.

Une autre série d'axiomes liés au typage a pour mission de traduire en logique du premier ordre toutes les informations de typage des constantes et des symboles de prédicats. Nous avions par exemple vu que la constante now est un objet de type time, ce qui peut s'axiomatiser ainsi :

now_type	$\text{time}(\text{now})$

De la même façon, nous avons vu que, compte tenu de la sémantique du prédicat binaire agent, il est nécessaire que son premier argument soit de type event et que son deuxième argument soit de type entity. Pour traduire cela en logique du premier ordre, nous utilisons l'axiome suivant :

agent_type	$\forall A, B.(\text{agent}(A, B) \rightarrow \text{event}(A) \wedge \text{entity}(B))$

Un axiome similaire peut être donné pour chaque symbole de l'ontologie. Avant de passer à la description du second groupe d'axiomes formalisant la structure du temps et des événements, remarquons que tous les axiomes que nous avons vus jusqu'à présent peuvent être produits à partir

des seules informations de typage. Plus précisément, nous pouvons remarquer que l'information contenue dans un lexique chargé par `Nessie` est suffisante pour produire *automatiquement* tous les axiomes formalisant les contraintes liées au typage, que ce soit ceux partitionnant le domaine ou ceux issus des constantes et prédicats.

Pour mener à bien cette tâche à la fois fastidieuse et peu intéressante, nous avons développé l'outil `TheoGen` qui, étant donné un lexique de `Nessie` est capable de produire une théorie du premier ordre dans un format lisible par `Curt`. Etant donné un lexique contenant des déclarations de types, constantes, familles et lemmes, `TheoGen` procède en deux étapes. Dans un premier temps, tous les types atomiques (donc ceux qui ne sont pas des alias) sont extraits du lexique. Pour toute paire de types atomiques (τ_1, τ_2) telle que $\tau_1 \neq \tau_2$, $\tau_1 \neq t$ et $\tau_2 \neq t$, on génère un axiome interdisant à un élément du domaine du modèle de vérifier à la fois les prédicats τ_1 et τ_2. De plus, si l'ensemble des types atomiques déclarés par le lexique (donc à l'exclusion de t) est $\mathcal{T}$, on produit un axiome stipulant que tout élément du modèle doit vérifier au moins l'un des prédicats de $\mathcal{T}$.

Dans un deuxième temps, on calcule l'ensemble des constantes définies par le lexique : celles déclarées explicitement au moyen d'une directive `const` et celles introduites implicitement par les déclarations de lemmes appartenant à des familles ouvertes. Notons par ailleurs que nous excluons toutes les constantes définies à l'aide de l'opérateur := puisque celles-ci sont appelées à être remplacées par leur définition dans la représentation finale et ne seront donc pas visibles par les outils d'inférence, ce qui signifie qu'il n'est pas nécessaire de générer des axiomes les concernant. Comme nous l'avions indiqué précédemment, les constantes de type t ne sont pas prises en compte non plus par cette étape de génération d'axiomes. Chacune des constantes c de l'ensemble obtenu donne lieu à au plus un axiome qui est calculé à partir de son type τ. Trois cas peuvent se présenter :

1. Si τ est un type atomique (dont on sait déjà qu'il est différent de t), alors on génère l'axiome $\tau(c)$. C'est ce cas qui a donné lieu à l'axiome `now_type` vu précédemment ;
2. Sinon, τ est de la forme $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_r$, c'est-à-dire que c est une fonction à n arguments de types $\tau_1, \ldots, \tau_n$ et dont le résultat est de type τ_r. Il faut alors distinguer deux sous-cas :
   a) Si $\tau_r = t$, alors c est un prédicat (une relation) et les seules contraintes de typage dont on a besoin portent sur les arguments de ce prédicat, le constructeur de modèle garantissant de toute façon un résultat de type t. L'axiome produit dans ce cas est de la forme

   $$\forall X_1, \ldots, X_n . c(X_1, \ldots, X_n) \rightarrow (\tau_1(X_1) \wedge \ldots \wedge \tau_n(X_n)) .$$

   C'est ce cas qui a donné lieu à l'axiome `agent_type` vu précédemment ;
   b) Si $\tau_r \neq t$, alors nous avons à faire à une véritable fonction, au sens des langages du premier ordre. Dans ce cas, on aimerait générer non seulement des contraintes sur les arguments de c comme nous l'avons fait dans le sous-cas précédent, mais aussi une contrainte sur le résultat de l'application de la fonction à ses arguments. L'axiome à produire dans ce cas serait donc de la forme :

   $$\forall X_1, \ldots, X_n . c(X_1, \ldots, X_n) \rightarrow (\tau_1(X_1) \wedge \ldots \wedge \tau_n(X_n) \wedge \tau_r(C(X_1, \ldots, X_n))) .$$

> Cependant, cet axiome n'en est pas vraiment un, puisque l'antécédent de l'implication n'a pas de sens. En fait, on ne peut pas, dans ce cas, produire d'axiome reflétant les contraintes imposées par la fonction. Le seul moyen de contourner cet obstacle serait, à notre avis, de générer une théorie non pas seulement en fonction d'un lexique, mais en fonction d'un lexique *et* de la représentation sémantique considérée. Dans une telle situation, il serait possible de parcourir la formule à la recherche d'occurrences de la fonction et de générer un axiome adapté à la relation dont l'un des arguments utilise la fonction dans l'un de ses sous-termes. Pour notre part, nous ne produisons aucun axiome dans ce cas, ce qui n'est pas gênant dans la mesure où `Curt` ne peut de toute façon pas gérer les symboles de fonction. Nous aurons cependant l'occasion de revenir sur ces symboles par la suite.

L'outil que nous venons de présenter permet donc de générer automatiquement toute une série d'axiomes contenus « implicitement » dans le lexique que nous avons présenté précédemment. Pour ne donner qu'un seul exemple, citons celui du verbe « pospacerować » que le lecteur devrait commencer à connaître. Dans le lexique sémantique de `Nessie`, ce verbe est représenté par une constante de type kind et par deux lemmes fermés appartenant à la famille des verbes intransitifs. Les lemmes ne donnent lieu à aucun axiome, seule la constante est prise en compte. L'axiome qui est généré pour cette constante est :

$$\text{spacerowac_type} \qquad \text{kind}(\text{spacerowac})$$

Un outil tel que celui qui vient d'être présenté évite à l'utilisateur d'avoir à écrire par lui-même des axiomes qui peuvent être générés automatiquement, dont l'écriture serait fastidieuse et – nous avons eu l'occasion d'en faire l'expérience – source de beaucoup d'erreurs. En libérant l'utilisateur de cette tâche peu intéressante, il lui permet de se consacrer pleinement à une réflexion bien plus intéressante et qui ne saurait être automatisée, à savoir celle sur les propriétés que doivent avoir les structures mises en jeu. Dans l'exemple qui nous intéresse ici, les réflexions doivent clairement porter sur la structure que l'on souhaite donner au temps et aux événements.

Pour ce qui est du temps, nous nous contenterons pour le moment d'une axiomatisation simple. Nous demandons que la relation entre instants du temps soit irréflexive, transitive et totale. D'autres axiomes pourraient être ajoutés, comme par exemple le fait que le temps est illimité à gauche et à droite et dense, mais nous n'aurons pas besoin de ces axiomes dans un premier temps. Nous y reviendrons cependant par la suite, car ils ont, comme nous le verrons, des propriétés intéressantes.

Il nous reste à axiomatiser la structure des événements. Les axiomes se rapportant aux événements et décrivant leurs relations avec le temps peuvent être classés en trois catégories : des axiomes généraux, des axiomes concernant les événements instantanés, et des axiomes spécifiques à chacune des 5 classes verbales introduites précédemment.

Axiomes généraux

Les axiomes décrits ici formalisent les rapports entre événements et temps. Afin de se faire une idée intuitive de ce que ces axiomes expriment, le lecteur peut s'imaginer le temps comme étant une barre horizontale à laquelle les événements seraient accrochés, tels des ballons, par des fils. Voici les axiomes que nous utilisons dans cette catégorie :

event_has_inception	$\forall A.(\text{event}(A) \rightarrow \exists B.\text{inception}(A, B))$
inception_unique	$\forall ABC.((\text{inception}(A, B) \wedge \text{inception}(A, C)) \rightarrow B = C)$
event_has_conc	$\forall A.(\text{event}(A) \rightarrow \exists B.\text{conc}(A, B))$
conc_unique	$\forall ABC.((\text{conc}(A, B) \wedge \text{conc}(A, C)) \rightarrow B = C)$
agent_unique	$\forall ABC.((\text{agent}(A, B) \wedge \text{agent}(A, C)) \rightarrow B = C)$
patient_unique	$\forall ABC.((\text{patient}(A, B) \wedge \text{patient}(A, C)) \rightarrow B = C)$
inception_not_after_conc	$\forall ABC.((\text{inception}(A, B) \wedge \text{conc}(A, C)) \rightarrow \neg C < B)$
duration_before_conc	$\forall ABC.((\text{induration}(A, B) \wedge \text{conc}(A, C)) \rightarrow B < C)$
duration_after_inception	$\forall ABC.((\text{induration}(A, B) \wedge \text{inception}(A, C)) \rightarrow C < B)$
not_incep_and_dur	$\forall AB.\neg(\text{inception}(A, B) \wedge \text{induration}(A, B))$
not_induration_and_conc	$\forall AB.\neg(\text{induration}(A, B) \wedge \text{conc}(A, B))$
dur_until_conc	$\forall ABCD.((\text{induration}(A, B) \wedge \text{conc}(A, C)$ $\wedge B < D \wedge D < C) \rightarrow \text{induration}(A, D))$
dur_since_incep	$\forall ABCD.((\text{induration}(A, C) \wedge \text{inception}(A, B)$ $\wedge B < D \wedge D < C)) \rightarrow \text{induration}(A, D))$

Nous aimerions faire remarquer que, parmi les axiomes que nous venons de présenter, certains pourraient être évités sans affecter en rien la théorie obtenue. Nous voulons par exemple parler des quatre premiers axiomes :
event_has_inception, inception_unique, event_has_conc et conc_unuique. Ce que ces axiomes stipulent, c'est que tout événement a une inception et une conclusion [1], et que ces instants sont uniques. Ces axiomes perdraient à la fois leur sens et leur utilité si inception et conc étaient envisagées comme des fonctions prenant en argument un événement et renvoyant un instant du temps. Le λ-calcul simplement typé permet de représenter sans aucune difficulté de telles fonctions, et donc ce n'est pas `Nessie` qui nous empêcherait de procéder à un tel changement. Ce qui nous en empêche, c'est d'une part `Curt` qui n'a pas été conçu pour manipuler des symboles de fonction et, d'autre part, les constructeurs de modèles auxquels les fonctions posent quelques difficultés. Ces restrictions constituent à notre avis une raison profonde de ne pas s'en tenir à `Curt` pour les expérimentations futures et pour travailler autant que possible en logique d'ordre supérieure, si des outils d'inférence pour de telles logiques sont développés.

Evénements instantanés

La formalisation des culminations que nous allons présenter par la suite s'appuie sur le concept d'événement instantané qu'il convient donc d'axiomatiser également. Cette notion peut être axiomatisée de plusieurs façons : instant de l'inception égal à l'instant de la conclusion, événement sans durée, *etc.*Dans le contexte de la construction de modèles qui est le nôtre, nous utilisons l'axiomatisation suivante, où ie signifie instantaneous_event :

[1] On pourrait tout à fait concevoir une théorie des événements où cela n'est pas exigé. Il existe en effet des événements qui n'ont pas d'inception ou de conclusion « naturelle », comme par exemple le fait d'arriver.

ie_def_1	$\forall A.(\text{instantaneous}(A) \rightarrow \exists B.(\text{inception}(A,B) \wedge \text{conc}(A,B)))$
ie_def_2	$\forall AB.(\text{event}(A) \rightarrow ((\text{inception}(A,B) \wedge \text{conc}(A,B)) \rightarrow \text{instantaneous}(A)))$

Le premier de ces axiomes exprime qu'un événement est instantané s'il existe un instant qui est à la fois son inception et sa conclusion. Le second axiome est la réciproque du précédent. Ce second axiome pourrait paraître anodin, voire inutile, mais il n'est ni l'un ni l'autre. En effet, il devient utile pour construire des événements non instantanés. Pour de tels événements, on a réellement besoin d'une définition à l'aide d'une équivalence ; une implication seule ne suffirait pas.

Evénements spécifiques à une classe de verbes

À tous les axiomes généraux présentés jusqu'ici, il convient d'ajouter des axiomes concernant plus spécifiquement certaines classes de verbes. Nous ne présenterons ici que ceux ayant trait aux culminations, l'une des classes que nous avons le plus étudiée.

Nous avons introduit en section 4.3.2 la relation binaire $culm(E_1, E_2)$ qui signifie que E_2 est la culmination de E_1. Nous avons de plus expliqué comment nous pouvons, à l'aide d'un ou de deux événements, rendre compte de la sémantique des différentes formes du verbe « pisać » (être en train d'écrire). Cependant, les représentations qui apparaissent dans le lexique de `Nessie` ne font jamais référence *directement* à la relation culm. Elles se content d'utiliser le prédicat unaire culminates, pour exprimer que l'événement d'écriture, non instantané, a ou n'a pas culminé. Les axiomes qui suivent régissent les relations existant entre la relation binaire et le prédicat unaire.

culm_unique	$\forall ABC.((\text{culm}(A,B) \wedge \text{culm}(A,C)) \rightarrow B = C)$
culm_inj	$\forall ABC.((\text{culm}(A,C) \wedge \text{culm}(B,C)) \rightarrow A = B)$
culm_no_fix	$\forall A.\neg\text{culm}(A,A)$
culm_antisym	$\forall AB.(\text{culm}(A,B) \rightarrow \neg\text{culm}(B,A))$
culm_agent	$\forall ABC.((\text{culm}(A,B)_et\text{agent}(A,C)) \rightarrow \text{agent}(B,C))$
culm_patient	$\forall ABC.((\text{culm}(A,B) \wedge \text{patient}(A,C)) \rightarrow \text{patient}(B,C))$
culm_kind	$\forall ABC.((\text{culm}(A,B) \wedge \text{ek}(A,C)) \rightarrow \text{ek}(B,C))$
culm_inception	$\forall ABC.((\text{culm}(A,B) \wedge \text{conc}(A,C)) \rightarrow \text{inception}(B,C))$
culm_imp_inst	$\forall AB.(\text{culm}(A,B) \rightarrow \text{instantaneous}(B))$
culminated_def	$\forall A.(\text{culminated}(A) \rightarrow \exists B.\text{culm}(A,B))$
culminated_non_inst	$\forall A.(\text{culminated}(A) \rightarrow \neg\text{instantaneous}(A))$

Bien sûr, d'autres formalisations de cette théorie, peut-être plus simples, sont également possibles. Notons que, contrairement à d'autres symboles, la relation binaire culm ne peut être remplacée par une fonction. En effet, s'il est vrai que la culmination d'un événement est unique, il n'est pas vrai que tous les événements ont une culmination. Donc, si nous voulions faire de culm une fonction, il faudrait que cette fonction soit partielle, ce qui ne peut s'exprimer aisément dans le cadre du λ-calcul simplement typé.

Pour terminer cette présentation de quelques axiomes spécifiques à chaque classe de verbe, signalons que nous avons introduit un axiome spécifique au verbe « spacerować » et selon lequel

tous les événements introduits par ce verbe sont non instantanés. Là encore, c'est le type kind qui rend possible l'expression d'un tel axiome. Un axiome similaire devrait être ajouté pour chaque verbe de la classe 2 ajouté à la grammaire. Ceci pourrait être évité en procédant à une réification des classes des verbes, permettant d'écrire des axiomes portant sur tous les verbes d'une même classe. Nous n'avons cependant pas procédé à une telle réification, notre but n'étant pas la construction sémantique pour un grand nombre de verbes.

Le modèle

Si nous réessayons l'exemple que nous avions essayé au début de cette section, mais cette fois avec tous les axiomes que nous venons de présenter, voici ce que nous obtenons :

```
> piotr pospaceruje

Message (consistency checking): mace found a result.
Message (informativeness checking): mace found a result.
> models
1 D=[d2, d3, d4, d5, d1]
  f(1, time, [d2, d1])
  f(0, spacerowac, d3)
  f(1, process, [d3])
  f(0, piotr, d4)
  f(0, now, d1)
  f(2, lt, [ (d1, d2)])
  f(1, kind, [d3])
  f(1, instantaneous, [])
  f(2, inception, [ (d5, d1)])
  f(1, event, [d5])
  f(1, entity, [d4])
  f(2, ek, [ (d5, d3)])
  f(2, conc, [ (d5, d2)])
  f(2, agent, [ (d5, d4)])
```

On le voit, ce modèle est beaucoup plus en accord avec notre intuition de ce que devrait être un modèle pour la phrase considérée.

4.4.2 Raffinement de la sélection des axiomes

Nous sommes parvenus à construire un modèle satisfaisant pour la phrase « Piotr pospaceruje ». De la même façon, voyons quel modèle `Curt` est à même de proposer pour la phrase « Piotr pokochal Aline » (Piotr a commencé à aimer Alina). Après avoir ajouté Alina au lexique syntaxique accompagnant la DCG et au lexique sémantique de `Curt`, nous regénérons à l'aide de `TheoGen` une base d'axiomes qui ne diffère de la précédente que par l'ajout d'un axiome, à savoir celui spécifiant qu'Alina (comme Piotr) est une entité. Nous pouvons alors demander à `Curt` de construire un modèle pour la phrase précédente :

```
> piotr pokochal aline
Message (consistency checking): paradox found a result.
```

```
Message (informativeness checking): paradox found a result.
Curt: OK.
> models
1 D=[d2, d3, d4, d5, d1]
  f(1, time, [d2, d1])
  f(1, state, [d5, d4, d3, d2, d1])
  f(0, piotr, d4)
  f(2, patient, [ (d5, d4)])
  f(0, now, d1)
  f(2, lt, [ (d2, d1)])
  f(0, kochac, d3)
  f(1, kind, [d3])
  f(1, instantaneous, [])
  f(2, inception, [ (d5, d2)])
  f(1, event, [d5])
  f(1, entity, [d4])
  f(2, ek, [ (d5, d3)])
  f(2, conc, [ (d5, d1)])
  f(0, alina, d4)
  f(2, agent, [ (d5, d4)])
```

Si nous observons attentivement ce modèle, nous constatons que Piotr et Alina ont été interprétés par le même élément, à savoir `d4`. Ce résultat n'est pas vraiment conforme à notre intuition, selon laquelle Piotr et Alina devraient être des individus distincts. Mais, pour insatisfaisante qu'elle soit, la réponse de `Curt` n'a rien de surprenant : il s'agit encore une fois d'une conséquence du fait que les constructeurs de modèles cherchent à construire des modèles *minimaux*. En effet, puisque rien n'exige qu'Alina et Piotr soient des individus distincts, les constructeurs de modèles ne les distinguent pas.

Nous l'avons vu, la résolution de ce problème est simple. Il suffit d'ajouter un axiome spécifiant que Piotr n'est pas égal à Alina. Le modèle de la phrase précédente obtenu après ajout de cet axiome est satisfaisant. Cependant, si nous redemandons à `Curt` un modèle pour « Piotr pospaceruje » avec une base contenant l'axiome selon lequel Piotr est différent d'Alina, nous obtenons le modèle suivant :

```
D=[d5, d3, d4, d2, d6, d1]
  f(1, time, [d5, d3])
  f(0, spacerowac, d4)
  f(1, process, [d4])
  f(0, piotr, d1)
  f(0, now, d3)
  f(2, lt, [ (d3, d5)])
  f(1, kind, [d4])
  f(1, instantaneous, [])
  f(2, inception, [ (d6, d3)])
  f(1, event, [d6])
  f(1, entity, [d2, d1])
  f(2, ek, [ (d6, d4)])
  f(2, conc, [ (d6, d5)])
  f(0, alina, d2)
```

```
f(2, agent, [ (d6, d1)])
```

On constate qu'Alina, bien qu'elle ne soit en aucun cas mentionnée dans la phrase initiale, apparaît dans le modèle produit par `Curt`. Reconnaissons-le, ce résultat a de quoi surprendre ! Pour le comprendre, il nous faut revenir au stockage des axiomes dans `Curt` et à la façon dont ceux qui vont être passés aux outils d'inférence sont sélectionnés.

Nous l'avons dit à la section 1.3, les axiomes utilisés par `Curt` sont stockés dans trois bases de connaissances `Prolog` distinctes : une base de connaissances lexicales, une base de connaissances liées à la situation à laquelle se rapportent les phrases engendrées par la grammaire, et une base de connaissances générales du monde. Chacune de ces bases de connaissances stocke les axiomes à l'aide d'un prédicat qui lui est propre, les trois prédicats ayant cependant la même arité. À titre d'exemple, voici les premiers axiomes issus de ces trois bases de connaissances présentes dans la version originale de `Curt` :

```
lexicalKnowledge(event,1,Axiom):-
   Axiom = all(A,imp(event(A),thing(A))).
situationalKnowledge(car,1,Axiom):-
   Axiom = some(X,some(Y,and(car(X),and(car(Y),not(eq(X,Y)))))).
worldKnowledge(have,2,Axiom):-
   Axiom = and(not(some(X,have(X,X))),all(X,all(Y,
               imp(some(Z,and(object(X),and(object(Y),and(object(Z),
                   and(have(X,Z),have(Y,Z))))),eq(X,Y))))).
```

Les prédicats `lexicalKnowledge/3`, `situationalKnowledge/3` et `worldKnowledge/3` associent à chaque axiome de la base un symbole et une arité qui permettent, comme nous allons le voir, de décider si l'axiome doit être passé ou non aux outils d'inférence.

Lorsque le moment est venu de choisir, en fonction de la représentation sémantique considérée, quels axiomes il convient de passer aux outils d'inférence, les trois bases de connaissance sont consultées. Etant donnée une représentation sémantique R, voici comment fonctionne l'algorithme de sélection d'axiomes.

1. On établit une liste des symboles présents dans R et de leurs arités ;
2. La liste précédente est utilisée pour extraire de la base de connaissances une liste d'axiomes pertinents. Un axiome est pertinent si le symbole et l'arité qui lui sont associés appartiennent à la liste construite à l'étape 1 ;
3. On remplace R par la conjonction de R et de tous les axiomes de la liste calculée à l'étape précédente ;
4. On calcule la liste des symboles et arités présents dans la nouvelle valeur de R ;
5. Si la liste de symboles et arités est la même que celle obtenue à l'étape 1, on retourne la nouvelle valeur de R qui n'est autre que la valeur initiale de R à laquelle ont été ajoutés tous les axiomes jugés pertinents ;
6. Sinon, on retourne à l'étape 2.

En résumé, les axiomes qui seront finalement passés aux outils d'inférence sont ceux qui sont associés à un symbole présent dans la représentation sémantique initiale ou dans un autre axiome dont un symbole est présent dans la représentation sémantique initiale, *etc.*

Ainsi, dans l'exemple précédent, l'apparition d'Alina dans le modèle d'une phrase où elle ne figure pas paraît moins surprenante. Une hypothèse raisonnable pour l'expliquer est que l'axiome spécifiant que Piotr est différent d'Alina est associé au symbole Piotr d'arité 1. Alors, comme Piotr apparaît dans la représentation sémantique initiale, cet axiome est sélectionné, conduisant à l'introduction de la constante Alina, pour laquelle le constructeur de modèles est forcé de trouver une interprétation. Bien sûr, nous devons trouver une solution à ce problème. Cependant, avant d'en proposer une, remarquons que ce problème fait aussi apparaître un besoin : celui de pouvoir connaître avec précision les axiomes utilisés par les outils d'inférence. En effet, si nous pouvons savoir quels axiomes sont effectivement utilisés pendant les tâches d'inférence, cela sera une aide précieuse pour comprendre les résultats fournis par ces tâches et qui, sans ces indications, resteraient la plupart du temps assez mystérieux.

Bien qu'il suffise, pour connaître les axiomes utilisés lors de l'inférence, d'examiner la formule produite par l'algorithme de sélection d'axiomes vu précédemment, cette façon de procéder n'est pas très commode car ces formules sont longues et peu lisibles. Nous allons donc associer à chaque axiome un nom qui pourra être utilisé à des fins d'affichage. En outre, pour faciliter le traitement des bases de connaissances d'axiomes et leur extension future, nous proposons trois changements par rapport au stockage de ces bases proposé par Blackburn et Bos. D'abord, il convient de remarquer que les symboles sont toujours utilisés avec la même arité. En d'autres termes, ceci revient à dire que `Curt` n'utilise pas de symbole variadique. Par conséquent, si deux axiomes sont associés à un même symbole, l'arité sera la même dans les deux cas. Donc, réécrire l'arité du symbole dans chaque axiome qui lui est associé ne semble pas très judicieux. C'est pourquoi, nous introduisons le prédicat `arity/2` qui à chaque symbole associe son arité. Les définitions d'axiomes n'ont donc plus besoin de mentionner l'arité, celle-ci pouvant être obtenue à partir du symbole et du nouveau prédicat. Ensuite, au lieu de stocker les axiomes dans des prédicats à 3 arguments, nous allons utiliser des prédicats à un argument, cet argument unique étant une liste de paires `atribut:valeur`. Cette façon de procéder est inspirée de celle utilisée dans les DCG pour associer plusieurs traits à un non-terminal. Enfin, plutôt que d'avoir un prédicat distinct pour chaque base de connaissance, nous proposons d'utiliser pour toutes les bases le prédicat `axiom/1`, la distinction entre bases de connaissances n'étant pas, en définitive, d'une importance capitale. Compte tenu de ces changements, voici comment s'écrivent les axiomes que nous avons vu précédemment :

```
arity(event,1).
arity(car,1).
arity(have,2).

axiom([symbol:event,formula:Axiom]):-
   Axiom = all(A,imp(event(A),thing(A))).
axiom([symbol:car,formula:Axiom]):-
   Axiom = some(X,some(Y,and(car(X),and(car(Y),not(eq(X,Y)))))).
axiom([symbol:have,formula:Axiom]):-
   Axiom = and(not(some(X,have(X,X))),all(X,all(Y,
```

```
imp(some(Z,
  and(object(X),and(object(Y),and(object(Z),
    and(have(X,Z),have(Y,Z)))))),eq(X,Y))))).
```

Et il devient alors aisé d'ajouter à chaque axiome un nom : il suffit pour cela d'ajouter à chaque lise une paire `name:nom`. C'est donc de cette façon que sont représentés les axiomes que nous avons présenté précédemment. À titre d'exemple, voici comment est représenté l'axiome spécifiant que Piotr est différent d'Alina :

```
axiom([
  name:piotr_neq_alina,
  symbol:piotr,
  formula:not(eq(piotr, alina))
]).
```

En apportant à l'algorithme de sélection d'axiomes les modifications imposées par ces changements, et en y ajoutant quelques lignes d'affichage, nous sommes en mesure de savoir, pour chaque représentation sémantique, quelle partie de la théorie présentée précédemment est effectivement passée aux outils d'inférence pour analyser cette formule. Par exemple, si nous reprenons l'exemple de la phrase « Piotr pospaceruje », voici ce que nous obtenons :

```
Axioms given to model builders: not_event_entity not_entity_time
not_time_event not_event_kind not_entity_kind not_time_kind
process_type now_type lt_type agent_type conc_type
inception_type ek_type lt_irreflexive lt_transitive lt_total
agent_unique event_has_inception inception_unique event_has_conc
conc_unique inception_not_after_conc instantaneous_definition_1
instantaneous_definition_2 processes_axiom spacerowac_type
piotr_type alina_type piotr_neq_alina spacerowac_process
```

Ces résultats confirment notre hypothèse : l'axiome piotr_neq_alina est passé aux outils d'inférence et c'est donc à lui que nous devons la présence d'Alina dans le modèle produit. Bien sûr, modifier le symbole associé à cet axiome en remplaçant piotr par alina ne ferait que déplacer le problème, conduisant à l'apparition de piotr dans le modèle d'une phrase où seule Alina est présente. Il faut donc procéder autrement et raffiner quelque peu l'algorithme de sélection d'axiomes implanté dans `Curt`.

La solution que nous proposons consiste à associer à chaque axiome non pas un mais plusieurs symboles, l'axiome étant alors sélectionné si et seulement si tous les symboles qui lui sont associés sont présents dans la représentation sémantique initiale. Les changements que nous avons apporté à la représentation des axiomes facilitent grandement une telle évolution. Il suffit en effet de remplacer l'attribut `symbol` de la liste des propriétés des axiomes par un attribut `symbols` dont la valeur est une liste de symboles au lieu d'être un symbole seul. Avec cette modification, l'axiome distinguant Piotr d'Alina s'écrit :

```
axiom([
  name:piotr_neq_alina,
  symbols:[piotr,alina],
  formula:not(eq(piotr, alina))
]).
```

Cette représentation est beaucoup plus satisfaisante, parce que plus « symétrique ».

De façon générale, le fait que certains axiomes soient passés aux outils d'inférence bien qu'ils ne soient pas réellement utiles pour raisonner sur la représentation sémantique considérée peut avoir des conséquences moins visibles que d'ajouter des éléments « indésirables » aux modèles comme nous venons de le voir. L'une des conséquences de la présence d'axiomes inutiles est – c'est presque une évidence – de faire augmenter inutilement la taille des formules que doivent manipuler les outils d'inférence. Or, compte tenu de la complexité des tâches telles que la construction de modèles et la recherche automatique de preuves, l'augmentation de la taille des formules conduit rapidement à des pertes d'efficacité notables, comme nous avons pu le constater au cours de nos expérimentations avec différentes théories. Ce sont ces constats qui nous conduisent à penser que l'amélioration que nous venons de présenter est des plus utiles.

Pour terminer cette section dédiée au choix des axiomes, nous voudrions faire part d'un autre enseignement que nous avons pu tirer de nos expérimentations. Lorsque nous avons introduit notre axiomatisation du temps, nous avons supposé que la relation d'ordre sur les instants était irréflexive, transitive et totale. Nous avions alors signalé que d'autres hypothèses pouvaient être faites, comme par exemple la densité, qui affirme qu'entre deux instants il y a toujours un troisième instant. Cet axiome peut être utile par exemple pour prouver qu'un événement qui se termine à l'instant t dans le futur peut également commencer dans le futur. Ceci n'est en effet possible que si l'on peut construire un instant se situant *après* now et avant t, ce que permet précisément la propriété de densité.

Cet axiome doit cependant être manipulé avec précaution. En effet, il exige que le temps soit un ensemble infini. Par conséquent, si cet axiome est transmis aux constructeurs de modèles, ceux-ci échoueront, car ils ne peuvent construire que des modèles finis dont aucun ne sera à même de satisfaire cet axiome de densité. Pour résoudre ce problème, nous ajoutons à la liste de propriétés des axiomes les attributs `tp` (theorem prover) et `mb` (model builder) qui peuvent prendre chacun les valeurs `yes` et `no`. Ces attributs permettent de spécifier, pour chaque axiome, s'il peut être passé aux prouveurs de théorèmes et aux constructeurs de modèles. Presque tous les axiomes peuvent être passés aux deux types d'outils et ont donc leurs attributs `tp` et `mb` à `yes`. Seuls les axiomes conduisant à une modélisation du temps par des ensembles infinis ne sont passés qu'aux prouveurs de théorèmes. Donner une valeur `no` aux deux attributs revient à rendre l'axiome inutilisable tant par les prouveurs de théorèmes que par les constructeurs de modèles, ce qui revient à le désactiver, c'est-à-dire, en quelque sorte, à le mettre en commentaire. Enfin, nous n'avons pas trouvé jusqu'ici d'exemple d'axiome qu'il serait nécessaire de passer aux constructeurs de modèles mais pas aux prouveurs de théorèmes, ce qui ne veut évidemment pas dire que de tels exemples ne verront pas le jour lors d'expérimentations ultérieures.

4.5 Génération de modèles non minimaux

Dans cette section, nous allons présenter un algorithme qui, à partir d'une représentation sémantique, d'une théorie du temps et des événements et d'un modèle de cette théorie satisfaisant la représentation sémantique est capable de construire d'autres modèles de la théorie satisfaisant la représentation sémantique initiale.

Dans la section précédente, nous sommes parvenus à construire le modèle suivant pour la

phrase « Piotr pospaceruje » :

```
1 D=[d2, d3, d4, d5, d1]
  f(1, time, [d2, d1])
  f(0, spacerowac, d3)
  f(1, process, [d3])
  f(0, piotr, d4)
  f(0, now, d1)
  f(2, lt, [ (d1, d2)])
  f(1, kind, [d3])
  f(1, instantaneous, [])
  f(2, inception, [ (d5, d1)])
  f(1, event, [d5])
  f(1, entity, [d4])
  f(2, ek, [ (d5, d3)])
  f(2, conc, [ (d5, d2)])
  f(2, agent, [ (d5, d4)])
```

Ce modèle décrit une situation dans laquelle la marche de Pierre commence à l'instant présent et se termine dans le futur. Cependant, cette situation n'est pas la seule situation décrite par la phrase qu'elle est sensée modéliser. En effet, si la phrase affirme bien que la promenade de Pierre se termine dans le futur, rien n'est affirmé quant au moment de son commencement. Certes, il se peut que Pierre commence à se promener à l'instant présent, mais il se peut aussi que sa promenade ait commencé il y a fort longtemps, ou qu'elle ne commence que plus tard. En résumé, bien que les modèles fournis par les constructeurs de modèles décrivent des situations réalistes, les situations qu'ils décrivent ne sont pas les seules situations réalistes possibles compte tenu des phrases qui ont été données.

Cependant, les constructeurs de modèles ne sont pas à même de trouver les autres modèles susceptibles de nous intéresser. En effet, ces autres modèles ne sont pas minimaux et échappent donc au champ de recherche des constructeurs de modèles. Dans l'exemple que nous avons donné précédemment, l'élément `d1` du modèle produit a été identifié à la fois avec `now` *et* avec l'inception de l'événement de marche. Ceci conduit à un modèle parfaitement légitime — mais la stratégie consistant à identifier des points lorsque cela est possible élimine les deux autres possibilités sémantiques que nous avions mentionnées. Les autres modèles ne sont pas minimaux parce qu'ils n'identifient *pas* le temps de l'énoncer avec le temps de l'inception. Or, il se pourrait très bien que l'un de ces modèles soit celui dont on a besoin pour traiter la suite du discours.

Nous devons donc trouver un moyen de produire les autres modèles intéressants, et cette section présente un algorithme qui renvoie une liste de *toutes* les situations réalistes, pour autant que le temps et l'aspect soient concernés [2]. L'entrée de cet algorithme est un modèle tel que celui que nous venons de présenter. Les modèles qu'il renvoie peuvent être vus comme des perturbations du modèle initial. Les modèles sont produits en deux étapes. D'abord, une étape de génération produit une liste de modèles possibles, de candidats. Puis, on procède à un filtrage permettant de ne garder que les modèles satisfaisant effectivement la représentation sémantique et la théorie utilisée. La seconde étape se réduit à la vérification de modèles du premier ordre

[2] Les travaux présentés dans cette section ont été publiés dans (Blackburn et Hinderer, 2007a)

telle qu'elle a été décrite, par exemple, dans (Blackburn et Bos, 2005a). Nous ne décrirons donc ici que la première étape, à savoir l'étape de génération.

Nos entrées pour cette étape sont une phrase S, sa représentation sémantique R en logique du premier ordre et une théorie T du temps et des événements telle que celle décrite à la section précédente. La formule R est supposée close et consistante avec la théorie T. Par conséquent, il existe un modèle M_0 de T, R. Notre but est de construire, en partant de M_0, l'ensemble $\mathcal{M}_f$ de tous les modèles « perturbations minimales » non-isomorphes de T, R. Nous construisons d'abord un ensemble $\mathcal{M}_i$ de modèles candidats (ceci correspond à la première étape, celle que nous allons décrire de façon plus précise ici). Tous les modèles produits peuvent être vus comme des perturbations du modèle initial M_0. La partie de M_0 qui ne concerne pas directement le temps et les événements sera la même pour tous les modèles. Les variations entre les modèles n'affectent que les points dénotant des instants du temps et les relations impliquant ces points. Plus précisément, la partie constante des modèles finaux (que nous appellerons le *cœur* dans le reste de cette section) est obtenue en enlevant l'information concernant le temps dans M_0. Par exemple, si M_0 est le modèle donné précédemment, alors son cœur est :

```
D=[d1,d2,d3]
f(0, piotr, d1)          f(1, entity, [d1])
f(0, spacerowac, d2)     f(1, event, [d3])
f(2, agent, [(d3,d1)])   f(1, kind, [d2])
f(2, ek, [(d3,d2)])      f(1, instantaneous, [])
                         f(1, process, [d2])
```

À partir du modèle cœur, nous construisons un autre modèle intermédiaire dans lequel tous les instants significatifs du temps sont représentés par des points distincts. Par « instants significatifs » nous entendons les moments où il se passe quelque chose, du point de vue de la représentation sémantique considérée. En premier lieu, nous ajoutons un point interprétant la constante now. Puis, nous parcourons tous les événements présents dans le cœur et, pour tout événement e nous procédons comme suit :

1. Si e est instantané, un point d_k est ajouté et le couple (e, d_k) est ajouté aux relations binaires inception et conc ;
2. Si e est non instantané, nous examinons les relations inception, induration et conc du modèle M_0. Pour chacune de ces relations binaires R dans laquelle e est impliqué, nous ajoutons un nouveau point d_i et étendons la relation R du modèle en cours de construction par le couple (e, d_i).

Remarquons que cette façon de procéder n'est pas indépendante de la théorie utilisée. Par exemple, la partie de l'algorithme traitant les événements instantanés ne fonctionnerait plus si ces derniers étaient modélisés par des événements sans durée. Cependant, l'algorithme serait assez facile à adapter à de tels changements dans la théorie.

L'application de cet algorithme au cœur vu précédemment donne le modèle intermédiaire suivant :

```
D=[d1,d2,d3,d4,d5,d6]
f(0, piotr, d1)          f(1, entity, [d1])
```

```
f(0, spacerowac, d2)    f(1, event, [d3])
f(0, now, d4)           f(1, instantaneous, [])
f(2, ek, [(d3,d2)])     f(1, kind, [d2])
f(2, conc, [(d3,d6)])   f(1, process, [d2])
f(2, agent, [(d3,d1)]) f(1, time, [d4,d5,d6])
f(2, inception, [(d3,d5)])
```

Le modèle obtenu après cette étape d'extension est presque complet. La seule chose qui manque est la relation $<$ spécifiant un ordre sur les instants du temps. Pour compléter le modèle, nous générons tous les ordres possibles sur les instants du temps, sans nous préoccuper de la cohérence des ordres ainsi produits. Nous appelons ces ordres des *successions*. Puis, pour chaque succession, nous construisons le modèle associé.

Le nombre de successions possibles croît exponentiellement avec le nombre d'instants considérés : 2 instants x et y donnent lieu à 3 successions ($x < y, x = y, y < x$), 3 instants engendrent 13 successions, 4 instants donnent 75 successions, *etc.*

Avant qu'une succession soit utilisée pour compléter un modèle, elle est simplifiée. La simplification consiste à remplacer tous les points dénotant un même instant du temps par un seul point. Par exemple, la succession $d_i = d_j$ serait remplacée par un seul élément d_k et une substitution serait produite pour renommer d_i et d_j en d_k, ce qui revient à donner à d_k à la fois toutes les propriétés de d_i et toutes celles de d_j. Bien sûr, cette substitution doit être appliquée au modèle intermédiaire de sorte que les fusions d'éléments soient prises en compte.

La simplification d'une succession produit une liste d'instants ainsi qu'une substitution à appliquer au modèle intermédiaire. L'ordre des éléments dans la liste traduit leur ordre chronologique. Le modèle final correspondant à une succession donnée est donc obtenu à partir du modèle intermédiaire en deux étapes :

1. Application au modèle intermédiaire de la substitution renvoyée par la simplification de la succession ;
2. Si $x_1, \ldots, x_n$ est la liste des instants renvoyée par la simplification de la succession, tout couple (x_i, x_j) tel que $1 < i < j < n$ est ajouté à la relation lt. Ceci permet de s'assurer que les propriétés de $<$ telles que la transitivité, l'irréflexivité et le fait d'être un ordre total sont respectées dans le modèle produit.

Ceci marque la fin de l'étape de génération de modèles que nous avons mentionnée précédemment.

Comme le modèle intermédiaire que nous avons montré précédemment comporte 3 instants, il donne lieu à 13 successions et donc à autant de modèles possibles en sortie de l'étape de génération. Ces 13 modèles sont testés (en utilisant un vérificateur de modèles du premier ordre) pour déterminer lesquels satisfont réellement à la fois la représentation sémantique R et la théorie T. Finalement, trois modèles sont conservés. Le premier d'entre eux est le modèle initial M_0. Le second se présente comme suit :

```
D=[d1,d2,d3,d4,d5,d6]
f(0, piotr, d1)         f(1, entity, [d1])
f(0, spacerowac, d2)    f(1, event, [d3])
f(0, now, d4)           f(1, instantaneous, [])
```

```
f(2, ek, [(d3,d2)])     f(1, kind, [d2])
f(2, agent, [(d3,d1)]) f(1, process, [d2])
f(2, conc, [(d3,d6)])  f(1, time, [d4,d5,d6])
f(2, inception, [(d3,d5)])
f(2, lt, [(d5,d4),(d5,d6),(d4,d6)])
```

Comme prévu, ce modèle correspond à une situation où la promenade de Pierre commence dans le passé. Le troisième modèle diffère du second seulement pour ce qui concerne l'ordre des instants du temps. Cette information se présente comme suit :

```
f(2, lt, [(d4,d5),(d4,d6),(d5,d6)])
```

Dans ce modèle, la promenade de Pierre commence dans le futur.

De la même façon, l'algorithme trouve les trois modèles possibles pour la phrase « Piotr pokochal Aline » (Piotr a commencé à aimer Alina) : l'un où l'événement d'aimer se termine dans le passé, un autre où il se termine au présent, et un troisième où il se termine dans le futur. Quant aux phrases « Piotr napisal list », et « Piotr popisal list » (Piotr a complètement / n'a pas complètement fini d'écrire une lettre), chacune d'elle n'a qu'un modèle. Le constructeur de modèles que nous utilisons trouve ce modèle, et notre algorithme de génération de modèles conclut avec justesse que ce modèle ne peut être perturbé.

Pour terminer, nous aimerions revenir sur un point que nous n'avons qu'évoqué au début de cette section et donner à ce sujet quelques détails supplémentaires. Nous avons signalé que parmi les modèles non minimaux que les constructeurs de modèles ne sont donc pas à même de fournir pourrait se trouver celui qui convient pour analyser la suite du discours. Cependant, nous n'avons pas expliqué comment un modèle pourrait être pris en compte pour analyser la suite d'un discours. La réponse à cette question, que nous n'avons pas implantée, consiste à construire, à partir du modèle choisi, une formule logique qui n'est satisfaite que dans ce modèle. Pour les modèles du premier ordre, une telle formule existe toujours et s'appelle le *diagramme* d'un modèle. Ainsi, de la même manière qu'il était possible, dans `Curt`, de choisir une lecture donnée pour une phrase qui en possède plusieurs, nous pouvons imaginer de modifier `Curt` de sorte qu'il soit possible de choisir, parmi les modèles fournis par l'algorithme précédent, celui que l'on souhaite utiliser pour analyser la suie du discours. `Curt` devrait alors construire un diagramme pour ce modèle et le passer aux outils d'inférence, au même titre que la théorie et les représentations sémantiques des phrases futures. Il est aussi possible que `Curt` génère de lui-même, pour chaque modèle, le diagramme qui lui correspond, ces diagrammes pouvant alors être envisagés comme autant de lectures possibles de la phrase initiale, au même titre que les différentes lectures résultant des ambiguïtés de portée.

Chapitre 5

Des formalismes pour calculer la sémantique des Discours

Au chapitre précédent, nous avons cherché à savoir quels pouvaient être les bénéfices du passage de la logique du premier ordre et du λ-calcul non typé à une logique telle que TYn pour la construction sémantique et l'inférence. Or, nous avons pu commencer à nous en apercevoir, ces bénéfices sont réels. Du point de vue de la construction sémantique, le recours à TYn incite à associer à chaque élément du lexique toute l'information portée par cet élément, plutôt que de laisser une partie de l'information remonter le long de l'arbre syntaxique pour n'être intégrée qu'ultérieurement à la représentation sémantique. Du point de vue de l'inférence, l'utilisation de représentations typées rend possible la génération automatique de théories du premier ordre qui facilitent sensiblement la construction de modèles et la preuve de propriétés concernant les représentations sémantiques.

Ces premiers résultats sont encourageants. Ils montrent que TYn, qui avait jusqu'ici surtout été utilisé en sémantique formelle, a aussi un intérêt réel en sémantique computationnelle. Il apparaît alors souhaitable de poursuivre les expérimentations entreprises au chapitre précédent et d'étudier d'autres phénomènes sémantiques, l'idée étant que de telles expérimentations pourraient apporter d'autres éléments de réponse à la question posée de la pertinence de TYn en sémantique computationnelle. Cependant, la plupart des phénomènes qui pourraient être étudiés (résolution d'anaphores, calcul de présuppositions…) ne peuvent l'être réellement que dans le cadre de discours constitués de plusieurs phrases, et non dans celui de phrases isolées. En d'autres termes, nous avons pu au chapitre précédent profiter des propriétés de la langue polonaise pour entreprendre une expérimentation en construction sémantique en n'utilisant que des phrases isolées, mais il nous semble évident qu'on ne pourra pas pousser beaucoup plus loin les expérimentations avec TYn dans un cadre aussi restrictif que celui des phrases isolées qui a été le nôtre jusqu'à présent.

Pour résumer, on peut dire que si l'on souhaite poursuivre les expérimentations en construction sémantique, il est nécessaire de se poser au préalable la question de la représentation de la sémantique de discours pouvant être constitués de plusieurs phrases. C'est donc à cette question que nous allons commencer à nous intéresser dans ce chapitre. Pour l'aborder, nous allons utiliser exactement la même démarche que nous avions déjà utilisée, au début de cette thèse, pour choisir un formalisme logique de spécification des représentations sémantiques. Nous allons en effet une nouvelle fois nous tourner vers la sémantique formelle et nous demander si certains des outils qu'elle met à notre disposition ne pourraient pas être utilisés avec profit en sémantique computationnelle. Plus précisément, nous allons présenter dans ce chapitre deux outils proposés

dans le cadre de la sémantique formelle, basés sur la théorie des types et permettant de calculer de façon compositionnelle la sémantique de discours constitués de plusieurs phrases. Ces outils seront ensuite mis en œuvre, au chapitre suivant.

Le premier outil est la DRT compositionnelle. Il est dû à Reinhard Muskens et a été introduit dans (Muskens, 1996c). Comme nous allons le voir en section 5.1, la proposition de (Muskens, 1996c) se donne pour objectif de traduire aussi fidèlement que possible les DRSs qui ont été brièvement introduites en section 1.2.3 par des λ-termes de TYn. Ainsi que nous le verrons, la structure des boîtes est traduite à l'aide de macros qui peuvent soit être conservées dans leur forme abrégée si l'on souhaite garder la structure de boîtes, soit être étendues si l'on souhaite obtenir un véritable terme de TYn. Lors de cette présentation, nous insisterons sur deux propriétés de la proposition de (Muskens, 1996c) qui nous paraissent importantes. Premièrement, elle n'est pas complètement compositionnelle, dans la mesure où, comme nous le verrons, les occurrences de certains éléments du lexique n'ont pas des représentations α-équivalentes. Deuxièmement, nous montrerons, en utilisant la réécriture d'ordre supérieur, que la construction de représentations reflétant la structure des boîtes ne peut se faire qu'en adjoignant à la β-réduction une autre règle de réécriture, ce qui signifie que l'on sort alors de TYn. En somme, nous montrerons qu'avec cette proposition, on est forcé de choisir entre, d'une part, la construction de termes reflétant la structure des boîtes mais obtenus au prix d'un détour hors de TYn, et, d'autre part, une construction faite entièrement dans TYn mais qui ne peut produire que des termes où la structure des boîtes n'apparaît plus. Le deuxième outil, appelé traitement compositionnel de la dynamicité, sera présenté en section 5.2. Il est dû à Philippe de Groote et a été introduit dans (de Groote, 2006). L'objectif visé par cette seconde approche est de rendre compte de la dynamicité de façon complètement compositionnelle, sans toutefois recourir à la DRT. Comme nous le verrons, cette seconde approche, qui fait des contextes (ensembles d'individus accessibles) des citoyens de première classe et utilise des continuations, est plus fortement compositionnelle que la première, dans la mesure où elle associe à toutes les occurrences d'un élément lexical des représentations α-équivalentes. Nous verrons aussi que cette approche est plus générale que la DRT, dans le sens où elle permet de modéliser tant ses contraintes d'accessibilité que d'autres contraintes.

5.1 La DRT compositionnelle

Les premières méthodes qui ont été proposées pour calculer des DRSs procédaient en partant d'une DRS vide à laquelle étaient ajoutées les contributions des phrases successives, au fur et à mesure que celles-ci étaient analysées. Plus précisément, il s'agissait de placer l'arbre syntaxique d'une phrase dans une DRS contenant les informations précédemment analysées, et de réduire cet arbre syntaxique jusqu'à ce qu'il ait été complètement incorporé à la DRS préexistante.

Cette façon de construire les DRSs peut être qualifiée à la fois d'incrémentale et de descendante. On la dit incrémentale parce qu'une DRS initialement vide est enrichie au fur et à mesure du parcours du texte d'entrée, et descendante parce que les règles de réduction des arbres syntaxiques sont utilisées en commençant par celle qui s'applique sur le nœud le plus proche de sa racine, celles concernant les feuilles s'appliquant en dernier.

Ainsi, nous constatons que les méthodes originelles de construction des DRSs ne respectaient pas vraiment le principe de compositionnalité. On pouvait alors se demander s'il était possible

de construire des DRSs de façon compositionnelle, ou si le langage des DRSs était intrinsèquement non compositionnel. C'est à cette question que Muskens répond en introduisant la DRT compositionnelle. Pour lui, le manque de compositionnalité est un problème des méthodes de construction de DRSs plus que du langage des DRSs lui-même. C'est cette méthode de construction compositionnelle des DRSs que nous allons présenter ici. Nous commencerons par présenter deux extensions du langage des DRSs proposées par Muskens et qui permettent d'en faciliter la construction compositionnelle. Puis, puisque cette méthode vise, comme nous l'avons dit, à construire les dénotations de DRSs plutôt que des DRSs en tant qu'objets syntaxiques, nous poursuivrons en montrant comment le λ-calcul permet d'exprimer la dénotation de formules du premier ordre, ce qui nous permettra d'introduire dans un cadre familier les concepts clés qui nous serviront à la présentation de la DRT compositionnelle par la suite.

5.1.1 Extensions du langage des DRSs

Nous allons présenter ici les deux extensions du langage des DRSs proposées par Muskens, à savoir l'ajout d'un opérateur de fusion de DRS et la possibilité d'utiliser des référents de discours constants au lieu de ne pouvoir utiliser que des variables.

Opérateur de fusion de DRSs

Nous l'avons dit, les méthodes originelles de construction de DRSs fonctionnent de façon incrémentale. On part d'une DRS vide à laquelle référents de discours et conditions sont ajoutés au fur et à mesure que le texte d'entrée est parcouru. Avec une telle méthode de construction, il n'y a donc jamais besoin d'associer une DRS à une phrase, ni de fusionner deux DRSs existantes. Il en va cependant autrement dès que l'on souhaite rendre la construction de DRSs compositionnelle. En effet, il paraît alors naturel d'associer à chaque phrase une DRS distincte puis, pour calculer la DRS associée à un discours comportant plusieurs phrases, de fusionner les DRSs qui ont été préalablement associées à chacune des phrases.

C'est pourquoi Muskens ajoute au langage de la DRT qui a été défini à la section 1.2.3 l'opérateur de fusion de DRSs représenté par un point-virgule. Ainsi, si un discours est composé des deux phrases S_1 et S_2, et si l'on peut associer à la première phrase la DRS K_1 et à la seconde phrase la DRS K_2, alors la DRS associée au discours complet est notée $K_1; K_2$.

Comme nous le verrons à la section 5.1.3, d'un point de vu sémantique l'opérateur de fusion de DRSs est en essence une conjonction. Ceci explique que, comme on le verra à la section 5.1.4, bien que la motivation qui a été donnée pour introduire cet opérateur soit la fusion de DRSs pour les phrases, on l'utilisera aussi dans d'autres contextes. Nous pouvons même dire que cet opérateur constitue en quelque sorte la clé de voûte de l'approche proposée dans (Muskens, 1996c). En effet, comme on va le voir, la représentation proposée pour les éléments lexicaux suit à peu près toujours le même principe : à partir d'un certain nombre de paramètres (dont le nombre et le type dépendent de la famille syntaxique), on construit une DRS « élémentaire » représentant l'élément du lexique. La DRS complète peut alors être obtenue en fusionnant toutes les DRSs élémentaires. C'est ce procédé de construction qui explique pourquoi l'opérateur de fusion que nous venons d'introduire joue un rôle crucial.

Enfin, signalons une propriété de l'opérateur de fusion que nous utiliserons abondamment par

la suite. Soient les deux DRSs $[x_1, \ldots, x_n|\gamma_1, \ldots, \gamma_m]$ et $[y_1, \ldots, y_p|\delta_1, \ldots, \delta_q]$. Si les référents de discours $y_1, \ldots, y_p$ de la seconde DRS n'apparaissent pas dans les conditions $\gamma_1, \ldots, \gamma_m$ de la première, alors les deux DRSs suivantes ont la même sémantique, c'est-à-dire qu'elles désignent la même relation entre enchâssements :

$$[x_1, \ldots, x_n|\gamma_1, \ldots, \gamma_m]; [y_1, \ldots, y_p|\delta_1, \ldots, \delta_q]$$

et

$$[x_1, \ldots, x_n, y_1, \ldots, y_p|\gamma_1, \ldots, \gamma_m, \delta_1, \ldots, \delta_q]$$

Par exemple, si la phrase « A man walks » est traduite par la DRS

$$[x_1|\text{man}(x_1), \text{walk}(x_1)]$$

et la phrase « A woman runs » est traduite par

$$[x_2|\text{woman}(x_2), \text{run}(x_2)]$$

alors le discours « A man walks. A woman runs. » peut se traduire par la DRS

$$[x_1|\text{man}(x_1), \text{walk}(x_1)]; [x_2|\text{woman}(x_2), \text{run}(x_2)]$$

Or d'après le lemme de fusion énoncé précédemment, cette dernière DRS dénote la même relation que la DRS réduite suivante :

$$[x_1, x_2|\text{man}(x_1), \text{walk}(x_1), \text{woman}(x_2), \text{run}(x_2)]$$

Référents de discours constants

Dans la formulation originale de la DRS, seules les variables pouvaient être utilisées comme référents de discours. La représentation d'un nom propre introduisait alors un nouveau référent de discours (donc une variable), et une condition rendant cette variable égale à la constante correspondant au nom propre introduit. Par exemple, la phrase « Mary walks » serait représentée par la DRS suivante :

$$[x|x = \text{Mary}, \text{walk}(x)]$$

Muskens propose de remplacer cette approche par une autre dans laquelle les constantes, comme celles représentant les noms propres, sont autorisées à apparaître en tant que référents de discours. Avec cette possibilité, la phrase précédente pourrait être représentée par la DRS suivante, plus simple que la précédente :

$$[\text{Mary}|\text{walk}(\text{Mary})]$$

Comme nous le verrons par la suite, la possibilité d'utiliser des constantes comme référents de discours ne sera pas exploitée seulement pour les noms propres. En effet, dans la proposition de (Muskens, 1996c), les représentations des déterminants et des pronoms utilisent eux aussi des constantes. Les constantes introduites par les déterminants permettent d'introduire les individus

sur lesquels porte la détermination, ces constantes pouvant par la suite être réutilisées par les pronoms.

Remarquons enfin que les constantes ont une portée *globale*. En d'autres termes, une fois introduite, une constante peut être réutilisée n'importe où dans une DRS. En particulier, une constante introduite par un déterminant pourra, comme nous le verrons, être réutilisée par la représentation d'un pronom même si les contraintes d'accessibilité font que cette constante n'est pas un antécédent valide pour ce pronom.

5.1.2 Dénotation des formules du premier ordre

Soit $\mathcal{C}$ un ensemble de constantes, $\mathcal{R}$ un ensemble de relations et $\mathcal{V}$ un ensemble infini dénombrable de variables. Soit de plus $M = (D, F)$ un modèle du premier ordre dont D est le domaine et F la fonction d'interprétation permettant d'interpréter à la fois les constantes et les relations. Nous allons ici montrer comment la satisfaction de formules du premier ordre construites à l'aide des constantes de $\mathcal{C}$, des variables de $\mathcal{V}$ et des relations de $\mathcal{R}$ peut être exprimée dans TYn. Pour ce faire, nous commençons par exprimer dans ce langage l'interprétation des termes du premier ordre et de la propriété pour une valuation de différer en au plus un point d'une autre valuation.

Interprétation des termes Pour pouvoir parler de l'interprétation d'un terme, il convient de bien distinguer les termes en tant qu'objets syntaxiques d'une part, et les éléments de D par lesquels on les interprète, d'autre part. Le langage TYn nous permet d'expliciter cette distinction de façon simple, à savoir en considérant que les termes en tant qu'objets syntaxiques sont d'un type, tandis que les éléments du modèle par lesquels ils sont interprétés sont d'un autre type, différent du premier. Nous noterons ici π le type des termes en tant qu'objets syntaxiques, et e le type des éléments du modèle, c'est-à-dire des éléments de D. Comme on le verra plus tard, ces notations sont celles utilisées dans (Muskens, 1996c) et c'est pour cette raison que nous y recourons dès maintenant. Les raisons justifiant de tels choix devraient devenir plus faciles à saisir lorsque nous présenterons la DRT compositionnelle proprement dite.

Pour en revenir aux termes, ceux que nous considérons ici ne peuvent être que des constantes ou des variables. Le vocabulaire Σ ne comporte en effet aucun symbole de fonction, et c'est pour cette raison que l'ensemble des termes considéré se réduit ici à $\mathcal{C} \cup \mathcal{V}$. Il faut donc, pour pouvoir interpréter les termes, être en mesure d'associer un élément de D à la fois aux constantes et aux variables. Comme on le fait d'habitude, nous allons interpréter les variables à l'aide de valuations, qui ne sont rien d'autre que des fonctions des variables vers les éléments de D. Nous noterons s le type des valuations. Nous verrons par la suite que l'on pourrait se passer de ce type, mais pour l'instant nous le conservons, toujours par souci de se conformer aux choix de (Muskens, 1996c).

On peut alors envisager la fonction d'interprétation d'un terme comme étant une fonction prenant en argument le terme à interpréter et la valuation utilisée pour interpréter ce terme s'il s'agit d'une variable. Nous noterons V cette fonction d'interprétation qui est donc de type $\pi \rightarrow s \rightarrow e$. Ainsi, l'interprétation du terme τ à l'aide de la valuation g peut être représentée par le λ-terme $(V\tau g)$, où τ est un objet de type π, g est de type s et $(V\tau g)$ est de type e.

Différence de valuations Étant données deux valuations g et g' et une variable x, nous cherchons ici à exprimer la propriété affirmant que g et g' associent les mêmes éléments de D à toutes les variables différentes de x, ce qui s'écrit plus formellement :

$$\forall y.y \neq x \rightarrow g(y) = g'(y)$$

où y est une variable. Le λ-terme de TYn dont nous avons besoin pour exprimer une telle propriété devrait prendre en paramètres une variable et deux valuations et retourner une proposition, c'est-à-dire un objet de type t. Cependant, compte tenu des types que nous avons introduits précédemment, il paraît malaisé d'attribuer un type correct à ce λ-terme, puisque nous ne disposons pas de type pour les variables.

Néanmoins, si nous appelons t la variable x vue comme un terme, nous pouvons remarquer que la proposition précédente est logiquement équivalente à la suivante :

$$\forall u.u \neq t \rightarrow (Vug) = (Vug')$$

où u ne désigne plus une variable, mais un terme, c'est-à-dire un objet de type π. Puisque ces deux formulations sont logiquement équivalentes, nous pouvons encoder celle de notre choix dans TYn. Et, comme on peut s'y attendre, c'est la deuxième que nous allons choisir, puisqu'elle s'exprime très facilement à l'aide des types dont nous disposons déjà.

La propriété pour deux valuations g et g' de coïncider partout sauf éventuellement en u, qui sera souvent notée $g[u]g'$ par la suite, peut donc s'exprimer dans TYn à l'aide du terme suivant :

$$\forall v : \pi.v \neq u \rightarrow Vvg = Vfg'$$

Par la suite, la notation $g[u]g'$ sera utilisée comme une abréviation pour ce terme. Cette notation sera d'ailleurs étendue pour exprimer que deux valuations diffèrent sur au plus n variables, c'est-à-dire que l'on notera $g[u_1, \ldots, u_n]g'$ le terme suivant :

$$\forall v : \pi.(v \neq u_1 \wedge \ldots \wedge v \neq u_n) \rightarrow Vvg = Vfg'$$

Dénotations des formules du premier ordre Comme nous l'enseigne Tarski, une formule du premier ordre peut être vue comme une description concise d'un ensemble de situations qui la rendent vraie. Par « situations », on entend généralement à la fois des modèles et des valuations dans ces modèles qui satisfont la formule considérée. Cependant, si le modèle est fixé une fois pour toute, alors la « situation » se résume à une valuation et une formule peut alors être vue comme spécifiant l'ensemble des valuations la rendant vraie dans ce modèle. C'est cet ensemble que nous allons construire, et plus précisément nous allons construire sa fonction caractéristique. Pour résumer, nous allons représenter les formules par des λ-termes de type $s \rightarrow t$, le terme représentant une formule ϕ étant noté $T(\phi)$. Pour toute formule ϕ, $T(\phi)$ est une fonction prenant une valuation (donc un terme de type s) en argument et renvoyant vrai si la valuation passée en argument satisfait ϕ.

Soit par exemple R une relation d'arité n. L'interprétation de cette relation dans le modèle M est notée $F(R)$ et peut être vue comme un terme de type T_n, où la suite (T_n) est définie de la façon suivante : $T_0 = t$ et $T_{n+1} = e \rightarrow T_n$. Par exemple, si $n = 1$, R est une relation unaire et

par conséquent $F(R)$ est un terme de type $e \rightarrow t$. De même, si $n = 2$, R est une relation binaire et $F(R)$ est un terme de type $e \rightarrow (e \rightarrow t)$ que l'on peut simplifier en $e \rightarrow e \rightarrow t$.

Revenons au cas général des relations n-aires et donnons-nous $\tau_1, \ldots, \tau_n$ n termes. Alors, si $\phi = R(\tau_1, \ldots, \tau_n)$, le λ-terme $T(\phi)$ caractérisant l'ensemble des valuations pour lesquelles ϕ est vraie est défini de la façon suivante :

$$T(R(\tau_1, \ldots, \tau_n)) = \lambda f : s.((FR)(V\tau_1 f) \ldots (V\tau_n f))$$

La dénotation des autres formules du premier ordre est définie récursivement de la façon suivante :

$$T(\neg\phi) = \lambda f : s.\neg T(\phi)(f)$$
$$T(\phi \wedge \psi) = \lambda f : s.T(\phi)(f) \wedge T(\psi)(f)$$
$$T(\phi \vee \psi) = \lambda f : s.T(\phi)(f) \vee T(\psi)(f)$$
$$T(\phi \rightarrow \psi) = \lambda f : s.T(\phi)(f) \rightarrow T(\psi)(f)$$
$$T(\forall x.\phi) = \lambda f : s.\forall g : s.(f[x]g \rightarrow T(\phi)(g))$$
$$T(\exists x.\phi) = \lambda f : s.\exists g : s.(f[x]g \wedge T(\phi)(g))$$

Ces définitions se lisent presque toutes simplement. Les seules qui peuvent présenter quelques difficultés sont celles proposées pour les dénotations des quantificateurs. Par exemple, la dénotation du quantificateur universel se lit de la façon suivante : la formule $\forall x.\phi$ est vraie pour la valuation f si et seulement si pour toute valuation g qui diffère de f au plus en x, g rend ϕ vraie. Une lecture semblable peut être donnée pour la dénotation du quantificateur existentiel.

Pour résumer, le λ-calcul typé et plus précisément TYn permet d'écrire dans un langage formel la définition de la sémantique des formules du premier ordre, qu'on ne peut habituellement donner que dans un langage informel.

5.1.3 Dénotation des DRSs

Nous avons donné en section 1.2.3 une définition du langage de la DRT, et c'est cette définition qui a été étendue au début de cette section par l'ajout d'un opérateur de fusion de boîtes et la possibilité d'utiliser des constantes (et non plus uniquement des variables) en tant que référents de discours. Une fois que la syntaxe d'un tel langage a été définie, il est possible de lui donner une sémantique dans un modèle du premier ordre, exactement de la même façon que cela a été fait pour les formules de la logique du premier ordre. Pour la DRT, il existe même deux façons de l'inerpréter dans un modèle. L'une des interprétations se fait à l'aide de fonctions partielles, tandis que l'autre, celle qui va nous intéresser ici et que l'on qualifie souvent de *sémantique dynamique*, interprète chaque boîte comme étant une relation entre deux fonctions totales, chacune de ces fonctions associant à tout référent de discours un élément du domaine d'interprétation du modèle.

C'est cette dernière définition qui est utilisée dans (Muskens, 1996c) pour construire la DRT compositionnelle. En fait, celle-ci est construite en exprimant la sémantique relationnelle des DRSs dans TYn, *exactement* comme nous venons de le faire pour la sémantique de Tarski de la logique du premier ordre. De ce fait, le système de type utilisé par Muskens est précisément celui

qui a été introduit, c'est-à-dire une version de TY3 dans laquelle les types autres que t sont e, π et s. La seule chose qu'il nous reste à faire, c'est de reprendre les explications que nous avons données pour les adapter au cas particulier de la DRT. Par exemple, nous avons indiqué précédemment que le type π représentait les termes du premier ordre en tant qu'objets syntaxiques, ces termes pouvant prendre la forme d'une constante ou d'une variable. Dans le contexte de la DRT telle qu'elle se présente après enrichissements par l'ajout de l'opérateur ; et l'acceptation de référents de discours constants, ce que nous appelions des « termes » correspond aux référents de discours. Pour distinguer les référents de discours constants de ceux qui sont des variables, (Muskens, 1996c) appelle les premiers des référents de discours *spécifiques*, tandis que les seconds sont dits *non spécifiques*. Une autre dénomination utilisée dans (Muskens, 1996c) pour parler des référents de discours est celle de « registres ». Il s'agit de souligner le fait que les référents de discours peuvent également être vus comme des objets pouvant contenir des entités, le contenu d'un registre non spécifique pouvant varier au fur et à mesure de la construction de la DRS. Dans la mesure où la sémantique dynamique envisage les DRSs comme des programmes, on comprend bien cette idée de registres dont le contenu peut varier en fonction de l'état d'exécution, les états n'étant alors rien d'autre que les enchâssements, ces fonctions qui associent à chaque registre son contenu à un moment donné de « l'exécution du programme ».

C'est pour rappeler cette intuition que le type des enchâssements est noté s (pour « state »). De même, les registres sont vus comme des « pigeon-holes » (c'est-à-dire des nids de pigeons) contenant quelque chose (comme nous l'avons dit, une entité), d'où le choix de la lettre π pour représenter leur type.

Voici finalement la traduction des DRSs en λ-termes de TY3. Pour donner cette traduction, nous considérons chaque clause de la définition des DRSs étendue par Muskens. Nous rappelons sa syntaxe, sa sémantique et donnons le λ-terme représentant cette sémantique, tout comme nous l'avons fait précédemment pour la sémantique de Tarski des formules du premier ordre. Dans cette définition, nous supposons que $u_1, \ldots, u_n$ sont des référents de discours (spécifiques ou non spécifiques), R est une relation d'arité n, $\gamma_1, \ldots, \gamma_m$ des conditions, K_1, K_2 et K des DRSs ou boîtes. Alors, nous avons les deux traductions mutuellement récursives pour les DRSs et les conditions.

D'après ces définitions, nous pouvons constater que les conditions qui apparaissent dans les boîtes sont représentées par des termes de type $s \to t$, tout comme l'avaient été les formules du premier ordre dans la sous-section précédente. Les boîtes, quant à elles, sont représentées par des termes de type $s \to s \to t$, ce qui rappelle bien l'interprétation que leur donne la sémantique dynamique par des relations binaires entre enchâssements.

Une fois que cette traduction a été donnée, les termes apparaissant dans la première colonne peuvent réellement être considérés comme les abréviations des λ-termes apparaissant dans la troisième colonne. Dans la suite de ce chapitre, nous ferons référence aux termes de la première colonne en parlant de *macros*, ceux de la troisième colonne pouvant alors être considérés comme les *définitions* de ces macros.

Dans (Muskens, 1996c), les macros et leurs définitions sont considérées comme équivalentes. Cette proposition utilise tantôt des termes comportant des macros, tantôt des termes où les macros ont été remplacées par leur définition, selon ce qui est le plus commode. Nous reviendrons sur ce point un peu plus tard, et montrerons qu'en réalité, les choses ne sont pas tout à fait aussi simples et que, d'un point de vue computationnel, les termes avec macros ne sont pas équivalents

DRS	Sémantique dynamique	Traduction dans TYn
$[u_1, \ldots, u_n \vert \gamma_1, \ldots, \gamma_m]$	$\{(i,j) \vert i[u_1, \ldots, u_n]j \wedge \gamma_1(j) \wedge \ldots \wedge \gamma_m(j)\}$	$\lambda i, j.i[u_1, \ldots,_n]j \wedge \gamma_1(j) \wedge \ldots \wedge \gamma_m(j)$
$K_1; K_2$	$\{(i,j) \vert i[u_1, \ldots, u_n]j \wedge \gamma_1(j) \wedge \ldots \wedge \gamma_m(j)\}$	$\lambda i, j.i[u_1, \ldots,_n]j \wedge \gamma_1(j) \wedge \ldots \wedge \gamma_m(j)$

Conditions	Sémantique dynamique	Traduction dans TYn
$R(u_1, \ldots, u_n)$	$\{i \vert (V(u_1, i), \ldots, V(u_n, i)) \in F(R)\}$	$\lambda i.R(V(u_1, i), \ldots, V(u_n, i))$
$K_1 \vee K_2$	$\{i \vert \exists j.iK_1j \vee \exists j \vert iK_2j\}$	$\lambda i.\exists j.iK_1j \vee \exists j \vert iK_2j$
$K_1 \rightarrow K_2$	$\{i \vert \forall j(i,j) \in K_1 \rightarrow \exists k.(j,k) \in K_2\}$	$\lambda i.\forall j.K_1(i,j) \rightarrow \exists k.K_2(j,k)$
$\neg K$	$\{i \vert \neg \exists j \vert (i,j) \in K\}$	$\lambda i.\neg \exists j.K(i,j)$

TAB. 5.1: Encodage de la DRT dans TYn.

à ceux où les macros ont été remplacées par leur définition.

5.1.4 Calculer des DRS compositionnellement

Dans cette partie, nous allons montrer comment calculer de façon compositionnelle des interprétations de boîtes telles qu'elles ont été présentées précédemment. La proposition de (Muskens, 1996c) ne permet pas d'associer une représentation à n'importe quel texte, mais seulement à des textes préalablement annotés de sorte à rendre compte des relations anaphoriques. Plus précisément, il est supposé que les textes d'entrée vérifient la convention de Barwise, et c'est donc par la présentation de cette convention, introduite pour la première fois dans (Barwise, 1987), que nous allons commencer. Nous poursuivrons en donnant les représentations que (Muskens, 1996c) propose d'associer aux éléments du lexique, puis nous donnons les règles utilisées pour combiner ces représentations. Enfin, nous verrons quelques exemples illustrant la construction compositionnelle de DRSs suivant cette méthode.

Convention de Barwise

Nous l'avons dit, (Muskens, 1996c) fait l'hypothèse que les relations anaphoriques sont explicitées dans les textes considérés. Il est supposé qu'elles seront restituées dans les textes d'entrée en annotant d'une part les groupes nominaux introduisant des entités auxquelles on peut faire référence plus tard, d'autre part les groupes nominaux constitués de pronoms et qui se réfèrent à des entités introduites précédemment. Chaque groupe nominal introduisant une entité est annoté avec un nombre écrit en exposant et chaque groupe nominal faisant référence à cette entité est annoté avec ce même nombre écrit en indice. Cette convention pour représenter les liens anaphoriques est due à Barwise. Voici un exemple de texte annoté en utilisant cette convention :

$$\text{A}^1 \text{ man owns a}^2 \text{ donkey. He}_1 \text{ beats it}_2.$$

Grâce à la convention de Barwise, les liens anaphoriques apparaissent clairement dans ce texte : le pronom « he » marqué par le 1 en indice fait référence à l'entité (l'homme) introduite par le groupe nominal « a man » marqué par un 1 en exposant, et il en va de même pour le pronom « it » marqué d'un 2 en indice pour faire référence à l'entité introduite par le groupe nominal « a donkey » annoté par un 2 en indice.

Précisons que la convention de Barwise interdit que deux groupes nominaux introduisant des entités soient annotés avec le même nombre en exposant, ce qui introduirait une ambiguïté dans la relation anaphorique. Ainsi, un texte comme

$$\text{A}^1 \text{ man owns *a}^1 \text{ donkey.}$$

ne respecte pas la convention de Barwise. De la même façon, cette convention interdit aux pronoms d'être marqués par des nombres n'apparaissant en exposant d'aucun groupe nominal préalablement rencontré. Des textes tels que

$$\text{A}^1 \text{ man owns a}^2 \text{ donkey. *He}_3 \text{ beats *it}_4$$

ne respectent donc pas davantage la convention de Barwise.

Insistons donc sur le fait que la proposition de (Muskens, 1996c) consiste à associer de façon compositionnelle des boîtes à des textes respectant la convention de Barwise, et uniquement à de tels textes. Cela signifie en particulier que la résolution des anaphores n'est pas envisagée ici comme faisant partie du processus de construction de boîtes mais comme une étape extérieure à ce processus [1].

Par ailleurs, nous avons beaucoup insisté sur la convention de Barwise car, comme on va le voir dans les sous-sections qui suivent, les nombres utilisés pour annoter les groupes nominaux jouent un rôle crucial dans les représentations associées aux entités introduites ou auxquels les groupes nominaux font référence.

Représentations des entrées lexicales

Nous donnons les représentations proposées par (Muskens, 1996c) pour les entrées lexicales, puis faisons quelques commentaires à leur sujet :

– déterminants indéfinis annotés (a^n) :

$$\lambda P' P.([u_n|]; P'(u_n); P(u_n))$$

– déterminants négatifs annotés (no^n) :

$$\lambda P' P.[|\neg([u_n]; P'(u_n); P(u_n))]$$

– déterminants universels annotés (every^n)]

$$\lambda P' P.[|(([u_n|]; P'(u_n)) \rightarrow P(u_n))]$$

– noms propres annotés (*e.g.* Mary^n) :

$$\lambda P.P(\text{Mary})$$

– pronoms annotés (he_n) :

$$\lambda P.P(\delta)$$

où δ est le référent de discours contenant l'antécédent du pronom

– pronoms relatifs (who, that) :

$$\lambda P' P.\lambda v.(P(v); P'(v))$$

– noms (*e.g.* man) :

$$\lambda v.[|\text{man}(v)]$$

– verbes intransitifs (*e.g.* walk) :

$$\lambda v.[|\text{walk}(v)]$$

[1] (Muskens, 1996c) propose néanmoins un algorithme permettant de vérifier qu'une DRS qui vient d'être construite vérifie les contraintes d'accessibilité. Nous dirons quelques mots de cet algorithme à la sous-section suivante.

– verbes intransitifs (*e.g.* love) :

$$\lambda Q.\lambda v.(Q(\lambda v'.[|\text{love}(v, v')]))$$

– verbe auxiliaire négatif does not :

$$\lambda P.\lambda Q.[|\neg Q(P)]$$

– construction conditionnelle if :

$$\lambda pq.[|(p \rightarrow q)]$$

Commençons par indiquer que, dans toutes ces règles, les variables P et P' sont de type $\pi \rightarrow s \rightarrow s \rightarrow t$, c'est-à-dire que P et P' sont des fonctions prenant en argument un référent de discours et construisant une DRS. La première règle associe un λ-terme à un déterminant indéfini annoté selon la convention de Barwise, comme nous l'avons expliqué précédemment. Le terme associé aux déterminants indéfinis annotés est une fonction prenant deux fonctions des registres vers les boîtes en arguments et renvoyant une DRS complexe. Plus précisément, la boîte qui est construite consiste en la fusion de trois boîtes simples : l'une introduisant un référent de discours u_n et ne comportant aucune condition, et deux autres obtenues en appliquant P' puis P au référent de discours qui vient d'être introduit. Le fait que l'indice n apparaissant dans le référent de discours u_n soit le même que celui annotant le déterminant indéfini dans la convention de Barwise n'est pas une coïncidence. En effet, dans (Muskens, 1996c), les nombres explicitant les relations anaphoriques selon la convention de Barwise sont utilisés pour choisir les noms des référents de discours à utiliser pour représenter les entités associées. Ces référents de discours sont des *constantes* qu'il convient d'ajouter à l'environnement de typage au fur et à mesure qu'on les rencontre. Or, comme nous supposons que les textes considérés vérifient la convention de Barwise, nous en déduisons que si un texte annoté contient deux occurrences d'un déterminant indéfini, celles-ci seront annotées avec des nombres différents en exposant, comme dans l'exemple

$$\text{A}^1 \text{ man owns a}^2 \text{ donkey.}$$

dont nous avions déjà fait état précédemment. D'après la règle que nous venons de donner pour associer un λ-terme à un déterminant indéfini, et grâce aux explications qui viennent d'être données, il est clair que la première occurrence du déterminant indéfini aura pour représentation $\lambda P'P.[u_1|]; P'(u_1); P(u_1)$, où u_1 est une constante à ajouter à l'environnement de typage, tandis que la seconde occurrence du déterminant indéfini aura pour représentation $\lambda P'P.[u_2|]; P'(u_2); P(u_2)$, où u_2 est une autre constante, différente de u_1 et qui doit aussi être ajoutée à l'environnement de typage. On constate alors que les deux occurrences du déterminant indéfini ont des représentations qui ne sont pas α-équivalentes, et c'est précisément à cause de cette non-α-équivalence que nous avons dû modifier `Nessie` pour passer de la première à la seconde version de l'algorithme de construction sémantique, comme nous l'avons expliqué au chapitre 2. Sans cette modification, il n'aurait pas été possible de construire à l'aide de `Nessie` des représentations sémantiques non α-équivalentes pour des occurrences distinctes d'un même lexème comme le fait (Muskens, 1996c). Grâce à la modification décrite au chapitre 2 nous pourrons, comme nous le verrons au chapitre suivant, rendre effective la proposition de (Muskens, 1996c) et construire automatiquement les DRSs associées à des textes simples.

Pour en revenir aux termes associés aux entrées lexicales, nous aimerions faire une dernière remarque au sujet des termes associés aux déterminants indéfinis. Nous avions vu en section 5.1.1 que, pour pouvoir construire les DRSs de façon compositionnelle, il est nécessaire de disposer d'un opérateur de fusion de DRSs noté ;. La nécessité d'introduire cet opérateur avait alors été motivée par le fait qu'il permettait, lors de la construction du sens d'un texte constitué de plusieurs phrases, d'associer une DRS à chaque phrase puis de fusionner les DRSs pour en obtenir une représentant le sens du texte complet. Or, ce que montrent les termes associés aux indéfinis (et cela est confirmé par les termes associés aux autres déterminants, aux verbes *etc.*), c'est que l'opérateur de fusion de DRSs est amené à apparaître bien plus souvent dans une DRS complexe que ce que laissait supposer la section 5.1.1. En effet, cet opérateur ne sert pas seulement à fusionner les DRSs représentant les phrases. En réalité, dans un contexte où les représentations sémantiques sont envisagées comme des conjonctions de prédicats appliqués à des arguments, l'opérateur ; de fusion de DRSs joue un rôle capital, puisque, en quelque sorte, cet opérateur peut être vu comme l'équivalent dans le langage de la DRT de la conjonction dans le langage de la logique. De ce fait, comme nous le verrons au chapitre suivant, l'opérateur de fusion de DRSs apparaît à de multiples reprises dans les DRSs complexes que l'on souhaite construire. On comprend ainsi pourquoi la règle permettant d'éliminer cet opérateur en procédant à la fusion effective de DRSs et qui a été présentée en section 5.1.1 joue un rôle capital. Nous reviendrons à cette règle dans la section suivante, puis de nouveau au chapitre suivant.

Pour le moment, terminons la présentation des principales règles d'association entre entrées lexicales et λ-termes.

Compte tenu des explications qui viennent d'être données quant aux termes associés aux indéfinis, ceux associés à aux autres types de déterminants devraient être faciles à comprendre. Les remarques qui ont été faites sur les indéfinis demeurent valables. Par ailleurs, bien que les DRSs soient un peu plus complexes que celles utilisées pour les indéfinis, les idées utilisées sont essentiellement les mêmes. La deuxième règle traite le cas des déterminants négatifs. La boîte qu'elle construit consiste en un univers vide (c'est-à-dire qu'elle n'introduit pas de nouveau référent de discours) et en une condition qui est la négation d'une boîte. La boîte à laquelle s'applique la négation est, quant à elle, similaire à celle utilisée dans la première règle. On pourrait se demander s'il est vraiment nécessaire de construire une boîte aussi compliquée pour les négations, et si l'on ne pourrait pas les représenter par des termes plus simples. Pourquoi ne pas utiliser, par exemple, quelque chose comme $[u_n|\neg(P'(u_n); P(u_n))]$? La raison pour laquelle se terme n'est pas utilisé est liée aux contraintes d'accessibilité. En effet, avec la représentation que nous venons de proposer, le référent de discours introduit par la négation est rendu accessible au niveau global, ce qui permet d'y faire référence plus tard dans le discours. Concrètement, cela signifie que le pronom « he » dans

No man walks. *He smokes

pourrait être relié anaphoriquement à l'entité introduite par l'homme apparaissant dans la première phrase et utilisé pour affirmer qu'aucun homme ne marche, ce qui n'est clairement pas conforme à nos intuitions. La DRS indiquée pour les déterminants négatifs n'a pas ce problème, car aucun référent de discours n'est ajouté au niveau global [2]. En d'autres termes, la portée des

[2]Nous verrons cependant un peu plus loin que (Muskens, 1996c) n'interdit pas de construire des boîtes qui ne respectent pas les contraintes d'accessibilité.

référents de discours introduits par les négations est limitée. Ce résultat est atteint en construisant une DRS dont l'univers est vide et qui ne contient qu'une condition, qui est la négation d'une DRS complexe.

Des remarques d'un genre similaire peuvent être faites pour la troisième règle associant un λ-terme aux déterminants universels. Comme le montre la DRS associée, la sémantique du déterminant universel est construite à partir de l'implication, tout comme dans des représentations montagoviennes plus classiques. De plus, ici aussi la portée du référent de discours introduit par cette règle est contrôlée de façon précise, grâce à des techniques similaires à celles expliquées précédemment et bien connues en DRT. Grâce à ces mécanismes de contrôle de la portée des référents de discours, des discours tels que

Every man has a nose. *He has a really big nose.

ne devraient en principe pas être traités correctement, car le pronom n'a pas d'antécédent.

La quatrième règle associe un λ-terme aux noms propres. La représentation qui leur est associée est, au moins dans l'esprit, proche de celle utilisée dans les approches montagoviennes. Le λ-terme est une fonction prenant en argument une fonction des registres (référents de discours) vers les boîtes et renvoyant la boîte obtenue en appliquant cette fonction au référent de discours introduit par le nom propre. Signalons à ce propos que les noms propres donnent lieu non pas à une seule, mais à deux constantes dan (Muskens, 1996c). L'une est le nom propre vu comme entité, l'autre est un registre ou référent de discours spécifique qui contient, quelque soit l'état (c'est-à-dire pour tous les enchâssements), l'entité que nous venons d'introduire. Ceci se comprend aisément en revenant à la sémantique des DRSs et aux liens que nous avons fait avec la traduction de la satisfaction des formules du premier ordre dans le λ-calcul typé. Un nom propre est une constante qui doit être interprétée, et qui peut donc être vue, du point de vu logique, à la fois comme objet syntaxique et comme élément du domaine d'interprétation dénoté par l'objet syntaxique. Remarquons par ailleurs que, bien que la convention de Barwise requière que les noms propres soient annotés par un nombre en exposant, ce nombre n'est pas utilisé dans les termes qui les représentent. Cette annotation n'est cependant pas superflue, puisqu'elle permet à un pronom apparaissant ultérieurement dans le discours d'établir un lien anaphorique avec le nom propre considéré. Le nombre annotant le nom propre ne va donc pas être utilisé directement dans la représentation de celui-ci, mais dans la représentation d'éventuels pronoms y faisant référence, comme le montre la cinquième règle. Pour cette règle, comme pour les autres, les idées qui l'inspirent sont claires. Elle est en effet relativement similaire à celle utilisée pour la règle précédente. Cependant, la règle est un peu différente de celle à laquelle on aurait pu s'attendre, puisqu'elle utilise un référent de discours δ au lieu d'utiliser u_n. Ceci est dû précisément au fait que le référent de discours à utiliser peut avoir été introduit par un nom propre, auquel cas son nom n'est pas u_n mais un nom dérivé de celui du nom propre. Mais il s'agit là de points de détail.

Les règles restantes associent des λ-termes aux pronoms relatifs, aux noms, verbes intransitifs et transitifs, au verbe auxiliaire « does not » et à des constructions de la forme « if ... then ... ». La représentation des pronoms relatifs devrait être facile à comprendre, car elle est similaire aux représentations que nous venons de détailler. Les règles associant des représentations au nom « man », au verbe intransitif « walk » et au verbe transitif « love » ont des généralisations évidentes à tous les noms, verbes intransitifs et transitifs. Suivant en cela les choix montagoviens,

(Muskens, 1996c) représente les noms et verbes transitifs par des prédicats ou relations unaires, tandis que les verbes transitifs sont représentés par des relations binaires. De ce fait, les représentations qui leur sont associées sont des fonctions qui, à partir des paramètres qui conviennent à chacune de ces trois familles syntaxiques, construisent des boîtes contenant une seule condition, qui n'est autre que la relation considérée appliquée aux référents de discours passés en paramètres. Ces boîtes seront fusionnées aux autres pour former la représentation sémantique finale grâce à l'opérateur de fusion.

La représentation donnée au verbe auxiliaire « does not » demande elle aussi quelques explications. Son deuxième argument, Q, doit être de type $(\pi \rightarrow s \rightarrow s \rightarrow t) \rightarrow s \rightarrow s \rightarrow t$. Pour comprendre cela, il est utile de remarquer que ce type n'est autre que celui qu'il convient de donner aux groupes nominaux. Le type du premier argument, P, est celui des groupes verbaux. Insistons sur le fait que, du point de vue des types, un groupe verbal constitué d'un verbe intransitif et un groupe verbal constitué d'un verbe transitif suivi d'un groupe nominal sont absolument identiques.

Enfin, la dernière règle associe une représentation aux constructions de la forme « if ... then ». La lecture de cette représentation nous paraît relativement aisée, puisque la boîte associée à la construction est une boîte dont l'univers est vide et contenant une seule condition qui est une implication sur les boîtes. Remarquons que cette règle respecte elle aussi les contraintes d'accessibilité de la DRT. En particulier, les référents de discours introduits dans l'univers de l'antécédent de la conditionnelle sont accessibles aux conditions du conséquent de cette même conditionnelle, mais pas au-delà, puisque la construction conditionnelle tout entière est placée dans une boîte dont l'univers est vide.

En conclusion de cette sous-section sur les représentations des entées lexicales, notons que nous venons de donner presque toutes les règles apparaissant dans (Muskens, 1996c). La seule qui a été omise est celle qui permet le « quantifier rising » et que nous ignorerons ici. En effet, cette règle permet de gérer les ambiguïtés de portée des quantificateurs, problème qui ne sera pas abordé par cette thèse.

Règles de calcul des représentations de syntagmes complexes

Le calcul de la boîte associée à un texte d'entrée respectant la convention de Barwise se fait en deux étapes, conformément à la tradition montagovienne. Dans un premier temps, l'analyse syntaxique du texte d'entrée est mise en œuvre à l'aide d'un formalisme grammatical approprié. Cette analyse produit un arbre syntaxique dont les feuilles sont étiquetées par des entrées lexicales et dont les nœuds sont, dans (Muskens, 1996c), unaires ou binaires. Dans un second temps, une boîte est associée à l'arbre syntaxique. Les règles données par Muskens et que nous allons reproduire ici concernent uniquement la seconde étape du traitement visant à produire une boîte à partir d'un arbre d'analyse. Ces règles de traduction, données par Muskens, sont au nombre de 5. Cependant, l'une d'entre elles s'occupe du « quantifier rising » et nous allons donc l'ignorer ici. Nous nous retrouvons donc avec les quatre règles de traduction suivantes :

1. La traduction d'une feuille est donnée par les règles vues à la sous-section précédente ;
2. La traduction d'un nœud unaire est la traduction de son unique sous-arbre ;
3. La traduction d'un arbre binaire est obtenue en appliquant celle de l'un des sous-arbres à

celle de l'autre, le sens de l'application étant donné de manière non ambiguë par les types. La traduction d'un texte représenté par la boîte K_1 suivi d'une phrase représentée par la boîte K_2 est donnée par la boîte $K_1; K_2$;

4. Deux règles de réduction :
   a) La règle de β-réduction ;
   b) La règle associée au lemme de fusion de boîtes, qui permet de remplacer le terme

$$[u_1, \ldots, u_n | \gamma_1, \ldots, \gamma_m]; [u'_1, \ldots, u'_k | \delta_1, \ldots, \delta_q]$$

   par le terme

$$[u_1, \ldots, u_n, u'_1, \ldots, u'_k | \gamma_1, \ldots, \gamma_m, \delta_1, \ldots, \delta_q]$$

   si les conditions du lemme de fusion de boîtes sont vérifiées.

Exemples

Nous allons montrer ici comment les règles qui ont été données dans les deux paragraphes précédents peuvent être utilisées pour associer des boîtes à des discours simples. Considérons pour commencer le discours suivant :

$$\text{John}^1 \text{ walks.}$$

Remarquons tout d'abord que ce texte respecte bien la convention de Barwise, bien que, en l'absence de pronoms, cela ne soit pas très important ici. D'après les règles données précédemment, la boîte associée à ce texte est obtenue en appliquant la représentation de John notée $\langle \text{John} \rangle$ à celle de walk notée $\langle \text{walk} \rangle$. Nous obtenons alors le λ-terme suivant :

$$(\lambda P.P(\text{John}))(\lambda v.[|\text{walk}(v)])$$

Une première étape de β-réduction conduit au terme

$$(\lambda v.[|\text{walk}(v)])\text{John}$$

qui se réduit à

$$[|\text{walk}(\text{John})]$$

Remarquons que l'univers de cette DRS est vide. Nous faisons en effet l'hypothèse que les référents de discours utilisés pour décrire les noms propres sont toujours disponibles.

Par ailleurs, si nous nous souvenons que cette notation avec crochets n'est qu'une abréviation et qu'elle peut par conséquent être étendue pour obtenir un authentique λ-terme, nous obtenons d'après nos clauses de traduction le terme suivant :

$$\lambda ij.(i[]j \wedge \text{walk}(V(\text{John}, j)))$$

Rappelons que dans ce terme, l'expression $i[]j$ est elle-même une abréviation pour le terme $\forall \delta : \pi.V(\delta, i) = V(\delta, j)$. Nous procédons donc au remplacement de la forme abrégée par la forme étendue, ce qui nous conduit au λ-terme final suivant :

$$\langle \text{John}^1 \text{ walks} \rangle = \lambda i, j.((\forall \delta : \pi.V(\delta, i) = V(\delta, j)) \wedge \text{walk}(V(\text{John}, j)))$$

Comme on peut le constater, même pour un texte aussi simple que celui considéré, sa représentation sémantique complètement étendue est un λ-terme quelque peu complexe et dans lequel la structure de la DRS qu'il représente n'est plus très visible. C'est pourquoi, lorsque nous présenterons des exemples plus complexes, nous nous en tiendrons le plus souvent aux formes abrégées, beaucoup plus évocatrices de véritables DRSs à notre avis. Nous reviendrons bientôt sur les liens entre forme abrégée et étendue des boîtes.

Considérons maintenant un exemple un tout petit peu plus compliqué, à savoir

$$\text{A}^1 \text{ man walks.}$$

La nouveauté ici est le fait que le groupe nominal ne consiste plus seulement en un nom propre, mais en un nom commun accompagné d'un déterminant indéfini. Comme cela est requis par le nombre qui annote le déterminant, ce déterminant introduit le référent de discours constant u_1. La représentation sémantique de la phrase est obtenue en appliquant celle du groupe nominal à celle du groupe verbal, la représentation du groupe nominal étant elle-même obtenue en appliquant celle du déterminant à celle du nom. Nous obtenons ainsi le terme suivant : $(\langle a^1 \rangle \langle \text{man} \rangle) \langle \text{walk} \rangle$ ce qui donne, compte tenu des règles de traduction données précédemment :

$$(\lambda P', P.([u_1|]; P'(u_1); P(u_1))\lambda v.[|\text{man}(v)])\lambda w.[|\text{walk}(w)]$$

En procédant à toutes les β-réductions possibles, on obtient :

$$[u_1|]; [|\text{man}(u_1)]; [|\text{walk}(u_1)]$$

À ce point, la seule règle de réduction que nous pouvons appliquer est celle qui permet de fusionner des boîtes. En l'appliquant deux fois nous obtenons la DRS finale :

$$[u_1|\text{man}(u_1), \text{walk}(u_1)]$$

Il va de soi que la règle de fusion de boîtes est indispensable pour passer de l'avant-dernier terme au dernier. Un système de réduction ne comportant que la β-réduction ne pourrait pas produire la forme la plus réduite. Cela signifie notamment que la règle de fusion de DRSs ne peut être exprimée en termes de β-réduction.

Nous nous en tenons là pour les exemples pour le moment, mais nous en verrons de plus élaborés au chapitre suivant, lorsque nous pourrons utiliser `Nessie` pour automatiser la construction de boîtes.

5.1.5 Quelques remarques

Cette partie comporte quelques remarques quant à la formalisation de la DRT dans le λ-calcul que nous venons de présenter.

Types de base utilisés

Nous l'avons dit, (Muskens, 1996c) propose d'utiliser TY3 pour encoder la DRT, en prenant comme types de base autres que t un type e pour les entités, un type π pour les registres ou

référents de discours, et un type s pour les états ou enchâssements. Or, comme nous l'avons expliqué précédemment, les enchâssements ne sont rien d'autre que des fonctions des registres vers les entités. On peut donc tout à fait envisager de se passer du type s et de le remplacer par le type $\pi \rightarrow e$. Si l'on procède ainsi, l'entité associée à un registre δ par un enchâssement i est donnée simplement par l'expression $i(\delta)$. Cette expression peut alors être utilisée à la place de l'expression $V(\delta, i)$ utilisée dans la proposition d'origine, ce qui permet de se passer également du symbole V. On peut alors se demander pourquoi Muskens n'a pas eu recours à cette solution, qui semble plus simple que la sienne. La raison de ce choix est qu'il donne un ensemble d'axiomes permettant, si l'on souhaite construire des modèles pour les interprétations des DRSs, de construire des modèles convenables. Or, dans ces axiomes, des quantifications sur les enchâssements apparaissent. Si ceux-ci n'étaient pas présents en tant que type de base, cela voudrait dire que les axiomes devraient utiliser des quantifications sur des fonctions, c'est-à-dire des quantifications du deuxième ordre. C'est précisément cela que Muskens a souhaité éviter, et c'est pour cela qu'il a eu recours au type s en tant que type de base.

Comme on peut le constater, cette préoccupation a peu de chose à voir avec la construction compositionnelle de DRSs qui nous occupe ici, et donc on aurait sans doute tout intérêt à préférer la version de la proposition de Muskens sans le type s et sans la constante V qui ne sont pas utiles dans notre contexte. Nous les conserverons cependant tous deux dans l'implantation que nous allons réaliser au chapitre suivant, d'une part pour que celle-ci soit la plus fidèle possible à la théorie originelle, d'autre part parce que le gain représenté par l'élimination du type s et du symbole V est purement cosmétique.

Contraintes d'accessibilité

Comme nous l'avons laissé entendre, le fait qu'un texte soit annoté suivant la convention de Barwise ne garantit pas que la DRS qui sera construite pour ce texte d'après la proposition de (Muskens, 1996c). Pour illustrer ce point, on peut considérer l'exemple

$$\text{No}^1 \text{ man walks. *He}_1 \text{ runs.}$$

Ce texte respecte la convention de Barwise, et on est donc en mesure de lui associer une DRS en utilisant les règles vues précédemment. Après simplification, on obtient la boîte suivante :

$$[|\neg[u_1|\text{man}(u_1)], \text{run}(u_1)]$$

On constate alors que u_1 est utilisé dans la seconde condition, alors qu'il n'est pas accessible puisqu'il apparaît dans l'univers d'une DRS apparaissant dans la première condition.

La convention de Barwise ne suffisant pas à garantir que les boîtes construites vérifient les contraintes d'accessibilité de la DRT, il est nécessaire de proposer un algorithme permettant de déterminer si une boîte satisfait les contraintes d'accessibilité, et c'est ce qui a été fait dans (Muskens, 1996c).

Nous n'allons pas donner les détails de cet algorithme ici. Disons seulement qu'il prend en entrée un terme de type $s \rightarrow s \rightarrow t$ et explore récursivement ce terme pour calculer les ensembles de référents de discours accessibles depuis chacune des boîtes et conditions qui y figurent. Ce qu'il nous paraît important de remarquer concernant cet algorithme, c'est que la structure de

terme qui est utilisée pour mener à bien l'exploration n'est pas celle donnée par le λ-calcul (variables, constantes, abstractions et applications), mais la structure induite par les abréviations vues précédemment. Les ensembles de référents accessibles sont ainsi définis pour des termes de la forme $[u_1, \ldots, u_n|\gamma_1, \ldots, \gamma_m]$, $K_1; K_2$, *etc.*Ceci signifie que l'entrée de cet algorithme ne peut pas être un terme quelconque de type $s \rightarrow s \rightarrow t$, car l'algorithme ne peut opérer que sur des termes où la structure des DRSs qu'ils représentent est toujours visible, c'est-à-dire sur des termes dans lesquels les abréviations vues précédemment n'ont pas encore été « dépliées ».

Ce constat est important d'un point de vu pratique. En effet, il signifie que, si l'on souhaite mettre en œuvre la construction de DRSs telle qu'elle est proposée dans (Muskens, 1996c) en y intégrant la vérification des contraintes d'accessibilité, on devra prendre en compte les abréviations utilisées dans la proposition originelle. On ne pourra pas, par exemple, utiliser pour représenter les éléments du lexique des termes où les abréviations n'apparaissent plus, comme on pourrait être tenté de le faire.

Abréviations, β-réduction et règle de fusion

Comme nous venons de l'expliquer, l'algorithme de vérification des contraintes d'accessibilité ne peut être appliqué qu'à des termes dans lesquels la substitution des macros par leur définition n'a pas encore été faite. Cet algorithme montre donc que les termes avec macros et ceux sans macros ne sont pas tout à fait équivalents. Il faut donc, avant de mettre en œuvre la construction compositionnelle de DRSs telle qu'elle a été proposée dans (Muskens, 1996c), réfléchir à la façon de gérer ces macros. Cela est à notre avis d'autant plus nécessaire que ces macros interagissent non seulement avec la règle de β-réduction, mais aussi avec la règle de fusion de DRSs qui, tout comme l'algorithme de vérification des contraintes d'accessibilité, ne peut être appliquée qu'à un terme où les macros n'ont pas encore été remplacées par leurs définitions.

Dans cette sous-section, nous souhaitons proposer un cadre théorique permettant de mener à bien étude fine des diverses interactions qui viennent d'être mentionnées. Ce cadre nous est donné par la réécriture, et plus particulièrement par la réécriture d'ordre supérieur. On peut en effet voir les macros, la règle de fusion de DRSs et la règle de β-réduction comme autant de règles de réécriture. Il est alors possible d'étudier les propriétés de terminaison et de confluence des systèmes obtenus en choisissant différentes combinaisons de règles.

Nous ne donnerons ici que les règles à considérer, ainsi qu'un résultat de non confluence. Nous terminerons cette présentation en suggérant quelques propriétés qui mériteraient selon nous d'être étudiées.

L'ensemble T des termes à considérer est celui des λ-termes de TY3 avec pour types de base autres que t les types e, π et s, cet ensemble étant enrichi par les constructions syntaxiques suivantes :

1. Macros définissant les DRSs :
   a) Si $u_1, \ldots, u_n$ sont de type π et $\gamma_1, \ldots, \gamma_m$ sont de type $s \rightarrow t$, alors
   $$[u_1, \ldots, u_n|\gamma_1, \ldots, \gamma_m] \in T$$
   est un terme de type $s \rightarrow s \rightarrow t$;

b) Si K_1 et K_2 sont de type $s \rightarrow s \rightarrow t$, alors $K_1; K_2 \in T$ est un terme de type $s \rightarrow s \rightarrow t$;

2. Macros définissant les conditions : soient K, K_1 et K_2 des termes de type $s \rightarrow s \rightarrow t$. Alors les termes suivants appartiennent à T et sont de type $s \rightarrow t$:
   a) $\neg' K$
   b) $K_1 \rightarrow' K_2$
   c) $K_1 \vee' K_2$

L'ensemble de termes T ainsi défini est muni des règles de réécriture suivantes, dans lesquelles les objets ont les mêmes types que précédemment :

1. Règles définissant les macros sur les DRSs :

   a) $[u_1, \ldots, u_n|\gamma_1, \ldots, \gamma_m]$
   $\rightarrow \lambda ij : s.((\forall \delta : \pi.(\delta \neq u_1 \wedge \ldots \wedge \delta \neq u_n)$
   $\rightarrow V(\delta, i) = V(\delta, j)) \wedge \gamma_1(j) \wedge \ldots \wedge \gamma_m(j))$
   b) $K_1; K_2 \rightarrow \lambda ij : s.\exists k : s.K_1(i,k) \wedge K_2(k,j)$

2. Règles définissant les macros sur les conditions :

   a) $\neg' K \rightarrow \lambda i : s.\neg \exists j : s.K(i,j)$
   b) $(K_1 \rightarrow' K_2) \rightarrow \lambda i : s.\forall j : s.K_1(i,j) \rightarrow \exists k : s.K_2(j,k)$
   c) $K_1 \vee' K_2 \rightarrow \lambda i : s.\exists j : s.K_1(i,j) \vee \exists j : s.K_2(i,j)$

3. Autres règles :
   a) La β-réduction
   b) La règle de fusion de DRSs, à savoir

$$[u_1, \ldots, u_n|\gamma_1, \ldots, \gamma_m]; [u'_1, \ldots, u'_p|\gamma'_1, \ldots, \gamma'_q]$$
$$\rightarrow [u_1, \ldots, u_n, u'_1, \ldots, u'_p|\gamma_1, \ldots, \gamma_m, \gamma'_1, \ldots, \gamma'_q]$$

Dans un premier temps, nous considérons le système de réécriture obtenu à partir de toutes les règles que nous venons de mentionner. Nous pouvons le caractériser en disant qu'il s'agit d'un système d'ordre supérieur, puisque les règles de réécriture qui le constituent font intervenir des variables liées. Nous conjecturons que ce système termine, mais la preuve formelle de sa terminaison reste à faire. Le système n'est cependant pas normalisant, car pas confluent. Pour le constater, considérons le terme $M = [u_1|\gamma_1]; [u_2|\gamma_2]$. Nous avons les deux réductions suivantes :
D'une part :

$$M \rightarrow [u_1, u_2|\gamma_1, \gamma_2]$$
$$\rightarrow \lambda ij : s.(\forall \delta : \pi.(\delta \neq u_1 \wedge \delta \neq u_2) \rightarrow V(\delta, i) = V(\delta, j)) \wedge \gamma_1(j) \wedge \gamma_2(j)$$

D'autre part, en posant $K_1 = [u_1|\gamma_1]$ et $K_2 = [u_2|\gamma_2]$ on a :

$$\begin{aligned} M &\rightarrow \lambda ij : s.\exists k : s.K_1(i,k) \wedge K_2(k,j) \\ &\rightarrow \lambda ij : s.\exists k : s.(\lambda ij : s.i[u_1]j \wedge \gamma_1(j))(i,k) \wedge (\lambda ij : s.i[u_2]j \wedge \gamma_2(j))(k,j) \\ &\rightarrow \lambda ij : s.\exists k : s.(i[u_1]k \wedge \gamma_1(k)) \wedge (k[u_2]j \wedge \gamma_2(j)) \end{aligned}$$

Nous obtenons ainsi deux termes irréductibles et qui ne sont pas équivalents modulo ce système de réécriture, ce qui prouve que le système considéré n'est pas confluent. Il serait cependant intéressant de tenter de compléter ce système, c'est-à-dire de chercher des règles qui, si elles lui étaient ajoutées, le rendraient confluent et donc normalisant.

Par ailleurs, les deux séquences de réductions de M qui viennent d'être présentées illustrent deux approches possibles pour simplifier les DRSs. Dans la première séquence de réduction, c'est la règle de fusion de DRSs qui est utilisée en premier. On obtient une DRS résultat de cette fusion, que l'on choisit ensuite d'étendre grâce à la règle de réécriture appropriée. Mais on pourrait très bien s'en tenir à cette DRS réduite et ne pas l'étendre. Un tel choix peut être modélisé par un système de réécriture où les seules règles disponibles sont la β-réduction et la fusion de DRSs. Les règles de substitution des macros ne sont alors plus prises en compte, et celles-ci restent donc présentes dans les termes obtenus. Ce système à deux règles [3] nous permet donc d'obtenir des termes qui reflètent fidèlement la structure de la DRS qu'ils représentent, le prix à payer pour obtenir cette structure étant que les calculs qui y aboutissent ne se font plus au moyen de la seule règle de β-réduction, mais en lui adjoignant une règle de réécriture, à savoir celle fusionnant deux DRSs.

La seconde séquence de réductions de M illustre une autre façon de traiter les DRSs et qui consiste à substituer les macros par leurs définitions. Cette seconde possibilité peut être modélisée par un système de réécriture comportant toutes les règles énumérées précédemment, sauf la règle de fusion de DRSs. Notre intuition est que ce système est lui aussi normalisant, mais ce résultat reste également à établir formellement. En tout cas, ce second système montre qu'il est possible de se passer de la règle de fusion de DRSs et que la proposition de (Muskens, 1996c) peut être mise enœuvre en recourant seulement à la β-réduction, et éventuellement à l'expansion de macros (on peut choisir de les utiliser ou de ne pas les utiliser). Cependant, si l'on choisit cette solution qui n'utilise que le λ-calcul, les termes que l'on obtiendra ne rendront pas copte de la structure de la DRS qu'ils représentent, et cette structure sera irrémédiablement perdue.

Il faut donc, avant de mettre en œuvre la proposition de (Muskens, 1996c), choisir laquelle des deux stratégies qui viennent d'être présentées on souhaite utiliser : soit celle produisant des termes reflétant la structure de la DRS qu'ils représentent mais nécessitant le recours à une règle de réécriture qui ne peut être exprimée dans le λ-calcul, soit celle n'utilisant que le λ-calcul (et la substitution de macros) mais aboutissant à des termes dans lesquels la structure de la DRS qu'ils représentent est perdue. Compte tenu des fonctionnalités disponibles dans `Nessie` et qui ont été présentées au chapitre 2, il est clair que la méthode la plus facile à mettre en œuvre est la seconde, puisque les mécanismes sur lesquels elle repose (expansion de macros et β-réduction) sont tous deux directement disponibles dans `Nessie`. Nous utiliserons cependant la première stratégie, qui nous permettra de construire des termes reflétant clairement la structure des DRSs

[3] Nous avons de bonnes raisons de croire que ce système est normalisant. La preuve formelle de ce résultat reste néanmoins à établir.

qu'ils représentent. Ce choix nous obligera certes à concevoir un outil supplémentaire mettant en œuvre la règle de fusion de DRSs, mais nous obtiendrons en échange des DRSs qu'il sera possible de réutiliser dans des expérimentations futures, l'expansion des macros restant elle aussi toujours possible une fois que les traitement qui ont besoin d'une structure de DRS ont été effectués.

5.2 Traitement Compositionnel de la Dynamicité

Dans cette section, nous allons présenter une proposition due à de Groote et qui a été introduite dans (de Groote, 2006). Cette proposition vise à traiter le caractère dynamique d'un discours de façon complètement compositionnelle. Pour y parvenir, l'auteur introduit une notion de contexte que nous décrivons en section 5.2.1. La section 5.2.2 explique comment les idées présentées dans la section précédente sont utilisées pour le calcul effectif de représentations tenant compte de la dynamicité, en partant des représentations des éléments lexicaux. Enfin, les sections 5.2.3 et 5.2.4 mettent l'approche préalablement introduite en relation d'une part avec la DRT, d'autre part avec `Nessie`.

5.2.1 Contextes

La DRT peut être vue comme une tentative pour prendre en compte une notion de contexte. En effet, l'intuition sous-jacente à la DRT était qu'un énoncé ne se réduit pas à l'affirmation d'une proposition, mais qu'il a aussi une action modifiant le contexte courant. Par exemple, une phrase comme « A man walks » n'affirme pas seulement que la propriété de marcher est vraie pour un individu qui se trouve être aussi un homme. Cette phrase introduit aussi un individu qui est un homme, qui marche et, peut-être plus important encore, auquel on peut faire référence dans la suite du discours. Ainsi une phrase comme « He is happy » serait parfaitement légitime précisément parce que le contexte courant contient un individu auquel le pronom personnel « he » peut faire référence. La même phrase apparaissant au début d'un discours créerait sans doute un certain sentiment de malaise, dû à l'incapacité de l'interpréter, précisément parce qu'il n'y a pas d'entité dans le contexte courant à laquelle le pronom pourrait se rapporter. Donc, l'intuition fondatrice de la DRT à propos de l'importance des contextes semble justifiée. Cependant, plusieurs questions demeurent posées :

1. Qu'est-ce qu'un contexte ?
2. Comment une phrase peut-elle utiliser un contexte ?
3. Comment une phrase peut-elle le mettre à jour et le transmettre ?

L'approche proposée dans (de Groote, 2006) apporte des réponses à toutes ces questions et nous allons voir lesquelles ci-dessous.

D'abord, comme en témoigne l'exemple de discours précédent, l'une des raisons pour lesquelles on peut souhaiter recourir à une notion de contexte est de garder une trace des entités qui ont été mentionnées, ces entités pouvant être utilisées par exemple pour la résolution des anaphores pronominales. Cette remarque suggère qu'un contexte pourrait contenir une liste d'entités précédemment mentionnées. Et ceci est en fait le choix de (de Groote, 2006). Dans son article,

il propose d'envisager les contextes comme des ensembles finis d'individus et affecte le type γ à ces objets.

Bien sûr, la structure des objets de type γ pourrait aussi être plus complexe, mais dans cette présentation nous nous en tiendrons à la proposition originale qui constitue à notre avis une réponse à la fois simple et élégante à la première question que nous avons posée ci-dessus.

La réponse à la seconde question est presque aussi simple que la précédente. Il y a en réalité deux aspects qu'il convient d'envisager ici. Dans un premier temps, il faut déterminer comment rendre le contexte accessible aux phrases qui ont besoin de le manipuler. Dans (de Groote, 2006), ceci est fait grâce à une λ-abstraction. En effet, au lieu d'affecter un type propositionnel aux représentations des phrases, celles-ci peuvent être considérées comme des fonctions prenant un contexte en argument et renvoyant une proposition (elles pourraient aussi renvoyer un objet d'un autre type, comme par exemple une fonction des mondes possibles vers les propositions, sans que cela modifie notre discussion). Cette vision fonctionnelle des phrases correspond bien à l'idée suggérée par notre premier exemple et selon laquelle les phrases doivent être interprétées dans un contexte. Donc, pendant la construction de la représentation sémantique d'une phrase, nous pouvons faire l'hypothèse que l'on dispose d'une variable de type γ, contenant le contexte courant et qui peut être utilisée par la représentation sémantique de la phrase. Pour compléter la réponse à la question 2, des opérateurs permettant de sélectionner un individu dans un contexte sont définis. Ces opérateurs sont des fonctions de type $\gamma \rightarrow \iota$, où ι est le type des entités. Ces fonctions prennent donc un contexte en argument et retournent l'individu choisi. Ce sont ces fonctions qui sont utilisées pour représenter les pronoms, leur argument étant un contexte contenant tous les antécédents possibles pour le pronom. Comme le fait de Groote, nous supposerons dans ce chapitre nous disposons d'une fonction différente pour chaque pronom qui nous intéresse. Nous utiliserons donc les fonctions (opérateurs de sélection) suivantes : $\texttt{sel}_{\text{he}}$, $\texttt{sel}_{\text{she}}$, $\texttt{sel}_{\text{it}}$, $\texttt{sel}_{\text{him}}$et $\texttt{sel}_{\text{her}}$. La mise en œuvre effective de ces opérateurs sera discutée au chapitre suivant, section 6.4.

Voyons maintenant la troisième question, peut-être la plus intéressante. Dans cette présentation, nous n'utiliserons qu'un seul opérateur de mise à jour du contexte, à savoir l'ajout à un contexte existant e d'un individu x, qui sera noté $x{::}e$, ce qui signifie que l'opérateur :: est une constante de type $\iota \rightarrow \gamma \rightarrow \gamma$. De plus, l'expression $x{::}(y{::}e)$ sera notée de façon plus concise $x{::}y{::}e$.

La véritable question qui demeure est de savoir comment faire en sorte que le contexte mis à jour soit transmis et devienne utilisable pour y interpréter la suite du discours. Bien sûr, il peut être passé en argument à la représentation de la phrase suivante, puisque les phrases ont désormais le type des fonctions des contextes vers les propositions. Mais d'où cet argument peut-il provenir, sachant qu'une phrase ne renvoie qu'une proposition et pas un contexte. La solution proposée dans (de Groote, 2006) pour résoudre ce problème est d'utiliser des *continuations*. Plus précisément, cela consiste à donner à chaque phrase un argument supplémentaire, argument qui n'est rien d'autre que la représentation de la phrase suivante, c'est–dire une fonction des contextes vers les propositions. Alors, passer le contexte mis à jour à la phrase suivante se ramène à appliquer au contexte la continuation représentant la sémantique de la phrase suivante.

Pour résumer, l'idée explorée dans (de Groote, 2006) consiste en les deux points suivants :

1. Utiliser des contextes (des objets de type γ) ;

2. Utiliser des continuations (c'est-à-dire des objets de type $\gamma \rightarrow o$, où o est le type des propositions), pour passer les contextes entre les représentations sémantiques. Les continuations représentant en réalité la sémantique de la suite du discours, elles sont appelées des *contextes droits* dans (de Groote, 2006). Pour notre part, nous utiliserons indifféremment les appellations *continuations* et *contextes droits* et nous posons en outre $\delta = \gamma \rightarrow o$.

Il en résulte que les phrases ne sont plus traduites par de simples propositions, mais par des fonctions prenant deux arguments et renvoyant une proposition, le premier argument étant un contexte et le second une continuation. Les phrases seront donc représentées par des termes de type $\gamma \rightarrow \delta \rightarrow o = \gamma \rightarrow (\gamma \rightarrow o) \rightarrow o$, type que nous noterons O par la suite.

Pour en revenir au premier exemple de cette section, voici une représentation possible pour la première phrase « A man walks. » :

$$\lambda e\phi.\exists x.(\text{man } x) \wedge (\text{walk } x) \wedge (\phi x{::}e)\,.$$

Les deux abstractions qui apparaissent au début du terme correspondent aux deux arguments que nous avons mentionnés précédemment : e est un contexte et ϕ une continuation. Puis vient la proposition représentant la phrase. La quantification existentielle est celle introduite par le déterminant indéfini, tandis que la signification des deux premiers conjoints devrait être claire. Le dernier conjoint, $\phi x{::}e$, paraîtra peut-être plus obscur. En fait, ϕ est la variable représentant le reste du discours (le contexte droit), et l'appliquer au contexte $x{::}e$ signifie interpréter le reste du discours dans un contexte contenant toutes les entités présentes dans le contexte avant que cette phrase ait été interprétée (ce qui correspond à e), plus l'individu x introduit par cette phrase. D'un point de vue sémantique, cela signifie que le reste du discours sera interprété dans un contexte où l'homme introduit par cette phrase (et représenté par la variable x) sera présent.

Pour voir comment l'homme introduit par la première phrase pourrait être rendu accessible à un pronom apparaissant dans une phrase ultérieure, considérons la phrase « He loves Mary ». Une représentation possible pour cette phrase pourrait être :

$$\lambda e\phi.(\text{love } (\texttt{sel}_{\text{he}} e) m) \wedge \phi e\,.$$

Cette représentation est relativement similaire à la précédente. La seule nouveauté est l'utilisation de l'opérateur $\texttt{sel}_{\text{he}}$qui sélectionne une entité dans un contexte. Ce second conjoint est même plus simple que le précédent. En effet, comme cette phrase ne modifie pas le contexte, sa continuation, la variable ϕ, peut être appliquée au contexte d'entrée de la phrase courante, à savoir e [4].

Il nous reste à voir comment ces deux représentations peuvent être fusionnées, c'est-à-dire en d'autres termes comment il est possible à partir de ces représentations d'en obtenir une pour le discours complexe « A man walks. He loves Mary. ».

Etant données deux représentations M_1 et M_2 de type $O = \gamma \rightarrow \delta \rightarrow o$, la représentation obtenue en les fusionnant doit avoir le même type. Voici le terme permettant de réaliser la fusion :

$$\lambda e\phi.(M_1 e(\lambda e'.M_2 e'\phi))$$

[4] On fait l'hypothèse ici que Mary fait déjà partie du contexte e.

Avant d'appliquer cet opérateur aux deux représentations précédentes, faisons quelques commentaires à son sujet. Les deux premières abstractions jouent exactement le même rôle dans le cas de l'opérateur de fusion que dans le cas des représentations vues précédemment : le premier est le contexte gauche du discours constitué par les deux discours fusionnés (qui est égal au contexte de la première phrase), tandis que le second est sa continuation. En ce qui concerne le corps du terme, la première phrase S_1 est appelée avec son contexte en argument. Le second argument de l'application est une fonction. En effet, on doit faire intervenir la seconde phrase, mais on ne connaît pas encore le contexte avec lequel il convient de l'appeler (le contexte à passer à la seconde phrase dépend de ce que la première aura ajouté au contexte d'entrée), d'où la λ-abstraction. Finalement, la continuation passée comme deuxième argument à M_2 est la continuation du discours complet, à savoir ϕ.

Nous pouvons maintenant calculer la représentation du discours constitué des deux phrases :

$$\begin{aligned}
\langle S_1 + S_2\rangle &= \lambda e\phi.\langle S_1\rangle e(\lambda e'.(\langle S_2\rangle e'\phi)) \\
&= \lambda e\phi.((\lambda e\phi.\exists x.(\text{man } x) \wedge (\text{walk } x) \wedge (\phi x{::}e))\lambda e'.\langle S_2\rangle e'\phi) \\
&\rightarrow_\beta \lambda e\phi.((\lambda\phi.\exists x.(\text{man } x) \wedge (\text{walk } x) \wedge (\phi x{::}e))\lambda e'.\langle S_2\rangle e'\phi) \\
&\rightarrow_\beta \lambda e\phi.\exists x.(\text{man } x) \wedge (\text{walk } x) \wedge ((\lambda e'.\langle S_2\rangle e'\phi))x{::}e \\
&\rightarrow_\beta \lambda e\phi.\exists x.(\text{man } x) \wedge (\text{walk } x) \wedge (\langle S_2\rangle x{::}e\phi)
\end{aligned}$$

Cette séquence de réductions explique comment le contexte, mis à jour par la première phrase, est passé à la seconde par le biais de la continuation. Nous pouvons maintenant remplacer $\langle S_2\rangle$ par sa valeur. Ceci nous donne la représentation suivante :

$$\begin{aligned}
\langle S_1 + S_2\rangle &= \lambda e\phi.\exists x.(\text{man } x) \wedge (\text{walk } x) \wedge ((\lambda e\phi.(\text{love } (\texttt{sel}_{\text{he}}e)m) \wedge \phi e)x{::}e\phi) \\
&\rightarrow_\beta \lambda e\phi.\exists x.(\text{man } x) \wedge (\text{walk } x) \wedge (\text{love } (\texttt{sel}_{\text{he}}x{::}e)m) \wedge \phi x{::}e
\end{aligned}$$

En supposant que l'opérateur $\texttt{sel}_{\text{he}}$effectue le bon choix, nous obtenons finalement :

$$\lambda e\phi.\exists x.(\text{man } x) \wedge (\text{walk } x) \wedge (\text{love } xm) \wedge \phi x{::}e\,.$$

Le dernier conjoint apparaissant dans ce terme, à savoir $\phi x{::}e$, montre que si une troisième phrase était ajoutée à ce discours, elle serait, comme la seconde, interprétée dans un contexte contenant tous les éléments initialement présents ainsi que x.

5.2.2 Calcul effectif de représentations sémantiques

Dans la sous-section précédente, nous avons introduit les contextes. Ils peuvent être vus comme des conteneurs d'information. Nous avons aussi vu que les continuations offrent un moyen naturel de transporter les contextes, (c'est-à-dire de l'information) d'une phrase à une autre. Dans cette section, nous expliquons comment des représentations sémantiques telles que celles données précédemment peuvent être construites en partant de λ-termes représentant les entrées lexicales. Dans (de Groote, 2006), l'auteur présente deux tentatives pour atteindre ce

but, et nous allons nous aussi les présenter toutes deux. De façon générale, les deux tentatives consistent à paramétrer les représentations montagoviennes classiques des différents composants syntaxiques par des contextes et des continuations. En effet, être en mesure de faire passer un contexte d'une phrase à l'autre n'est pas suffisant. Il faut aussi être en mesure de transporter des contextes mis à jour *'pendant* la construction *d'une* phrase. Par exemple, dans une phrase comme « A farmer who owns a donkey beats it. », le groupe nominal « a donkey » ajoute une entité au contexte, et le contexte ainsi mis à jour doit être disponible pendant la construction sémantique du reste de la phrase, de sorte que la résolution d'anaphore pour le pronom « it » puisse avoir lieu.

Les deux paramétrisations des représentations montagoviennes par des contextes proposées dans (de Groote, 2006) consistent donc à utiliser deux abstractions (l'une portant sur le contexte et l'autre sur la continuation) dans les représentations des noms, des déterminants et des autres catégories syntaxiques. Comme nous le verrons, la seule différence entre les deux paramétrisations est la façon dont celle-ci est faite.

Première tentative

Comme nous l'avons expliqué, il est nécessaire de permettre aux constituants syntaxiques des phrases d'accéder aux contextes et de les mettre à jour. Pour les groupes nominaux, par exemple, le type que Montague proposait de leur affecter était $(\iota \to o) \to o$. L'idée sous-jacente à ce type est qu'un groupe nominal est une fonction dont le seul argument est la propriété que l'on souhaite donner à l'entité portée par le groupe nominal. Dans une phrase comme « John walks. » par exemple, la fonction passée en argument au groupe nominal « John » serait « walk ». La représentation sémantique du groupe nominal est alors obtenue en appliquant cette fonction à l'entité introduite par le groupe nominal, ici « John ».

Maintenant, pour faire en sorte que les groupes nominaux prennent en compte la notion de contexte, nous pouvons paramétrer chaque occurrence de o par γ, le type des contextes gauches. Le type des groupes nominaux devient alors $(\iota \to \gamma \to o) \to \gamma \to o$. Un tel type permet au groupe nominal à la fois d'accéder à un contexte et de le modifier. L'accès est possible parce que le groupe nominal prend un contexte gauche comme deuxième argument. La modification, quant à elle, est possible grâce au premier argument. En effet, le premier argument est une fonction prenant un contexte en argument. Dans ces conditions, rien n'empêche un groupe nominal d'appeler son premier argument avec un contexte différent de celui qu'il a reçu comme second argument. Pour rendre ces explications plus concrètes, voici des représentations possibles pour les groupes nominaux John, Mary, he et her.

$$\langle \text{John} \rangle = \lambda \psi e.(\psi jj{::}e)$$
$$\langle \text{Mary} \rangle = \lambda \psi e.(\psi mm{::}e)$$
$$\langle \text{he} \rangle = \lambda \psi e.(\psi(\mathtt{sel}_{\text{he}} e)e)$$
$$\langle \text{her} \rangle = \lambda \psi e.(\psi(\mathtt{sel}_{\text{her}} e)e)$$

Il est clair que les interprétations des noms propres modifient le contexte avant de le transmettre à leur continuation, alors que les pronoms ne font que le consulter avant de le passer, inchangé, à leur continuation.

L'interprétation des verbes transitifs, quant à elle, n'est qu'une adaptation de la représentation utilisée par Montague aux contextes et aux continuations. Les verbes transitifs sont envisagés ici comme des fonctions prenant deux groupes nominaux en argument (d'abord le groupe nominal objet, ensuite le groupe nominal sujet), et renvoyant une phrase. Voici les λ-termes représentant les verbes « love » et « smile at » :

$$\langle \text{love} \rangle = \lambda ose\phi.s(\lambda xe'.o(\lambda y.\lambda e''.\text{love } xy \wedge \phi e'')e')e$$

$$\langle \text{smile at} \rangle = \lambda ose\phi.s(\lambda xe'.o(\lambda ye''.(\text{smile at } xy) \wedge \phi e'')e')e$$

Nous pouvons alors calculer la représentation de la phrase « John loves Mary » :

$$\begin{aligned}
\langle \text{John loves Mary} \rangle &= \langle \text{love} \rangle \langle \text{Mary} \rangle \langle \text{John} \rangle \\
&= (\lambda ose\phi.(s\lambda x.\lambda e'.o(\lambda y.\lambda e''.\text{love } xy \wedge \phi e'')e')e)\langle \text{Mary} \rangle \langle \text{John} \rangle \\
&\rightarrow^*_\beta \lambda e\phi.(\langle \text{John} \rangle (\lambda xe'.\langle \text{Mary} \rangle (\lambda ye''.\text{love } xy \wedge \phi e'')e')e) \\
&= \lambda e\phi.(\lambda \psi e'''.\psi j(j{::}e'''))(\lambda xe'.\langle \text{Mary} \rangle (\lambda ye''.\text{love } xy \wedge \phi e'')e')e \\
&\rightarrow_\beta \lambda e\phi.(\lambda e'''.(\lambda xe'.\langle \text{Mary} \rangle (\lambda ye''.\text{love } xy \wedge \phi e'')e')j(j{::}e'''))e \\
&\rightarrow^*_\beta \lambda e\phi.(\lambda e'''.\langle \text{Mary} \rangle (\lambda ye''.\text{love } jy \wedge \phi e'')(j{::}e'''))e \\
&\rightarrow_\beta \lambda e\phi.\langle \text{Mary} \rangle (\lambda ye'''.(\text{love } jy) \wedge \phi e'')(j{::}e) \\
&= \lambda e\phi.(\lambda \psi e'''.(\psi m(m{::}e'''))(\lambda ye''.(\text{love } jy) \wedge \phi e'')(j{::}e) \\
&\rightarrow_\beta \lambda e\phi.(\lambda e'''.(\lambda e''.\text{love } jy \wedge \phi e'')m)m{::}e'''))(j{::}e) \\
&\rightarrow_\beta \lambda e\phi.(\lambda ye''.(\text{love } jy) \wedge \phi e'')m(m{::}j{::}e) \\
&\rightarrow^*_\beta \lambda e.\lambda \phi.(\text{love } jm) \wedge \phi m{::}j{::}e
\end{aligned}$$

Avec cette paramétrisation des propositions par les contextes, il est aussi possible de donner des interprétations pour les substantifs et les déterminants. Si, comme dans les travaux de Montague, les noms sont vus comme des propriétés d'entités, on peut utiliser leur représentation traditionnelle :

$$\begin{aligned}
\langle \text{man} \rangle &= \lambda x.(\text{man } x) \\
\langle \text{woman} \rangle &= \lambda x.\text{woman } x\,.
\end{aligned}$$

Les déterminants, quant à eux, peuvent être vus comme des fonctions des noms vers les groupes nominaux. Étant données les affectations de types précédentes, cela signifie que le type envisagé pour les déterminants est le suivant :

$$(\iota \rightarrow o) \rightarrow (\iota \rightarrow \gamma \rightarrow o) \rightarrow \gamma \rightarrow o$$

Et voici les représentations proposées pour le déterminant indéfini et le déterminant universel :

$$\begin{aligned}
\langle \text{a} \rangle &= \lambda n\psi e.\exists x.nx \wedge \psi xx{::}\, e \\
\langle \text{every} \rangle &= \lambda n\psi e.\forall x.nx \supset \psi xx{::}e
\end{aligned}$$

Ces représentations sont, elles aussi, proches de celles proposées par Montague. La seule différence réside dans le fait que ces représentations ajoutent les entités introduites par les quantificateurs au contexte qu'elles reçoivent en entrée et passent le contexte ainsi mis à jour à leur continuation.

Il devient alors possible de calculer la représentation de la phrase « Every man loves a woman » :

$$\begin{aligned}\langle\text{Every man loves a woman}\rangle &= \langle\text{love}\rangle(\langle\text{a}\rangle\langle\text{woman}\rangle)(\langle\text{every}\rangle\langle\text{man}\rangle)\\ &\rightarrow^*_\beta \lambda e\phi.\forall x.(\text{man } x) \supset \exists y.(\text{woman } y) \wedge (\text{love } xy) \wedge \phi y::x::e\end{aligned}$$

Comme le montre cette représentation, des phrases subséquentes seraient interprétées dans un contexte où x et y sont présents. Par exemple, dans une phrase comme « He smiles at her. », les pronoms « he » (*resp.* « her ») se verraient associés aux antécédents x (*resp.* y) par une procédure de résolution d'anaphores. Cependant, ce n'est pas exactement ce que nous souhaiterions. En effet, étant donné l'interprétation que nous venons de calculer pour « Every man loves a woman. », ni x ni y ne devraient être accessibles pour la résolution d'anaphores dans les phrases suivantes. Et même si, étant donnée la phrase « He smiles at her » on considère que la première phrase devrait être interprétée en donnant la portée la plus large au groupe nominal objet (c'est-à-dire qu'il y aurait une femme très spéciale y aimée de tout homme x), cela permettrait certes à y d'être accessible aux phrases futures, mais pas à x !

Force est donc de constater qu'il y a bien un problème avec les représentations sémantiques choisies : elles ne nous permettent pas de contrôler avec assez de précision la portée des variables introduites par les quantificateurs. Et ce n'est là que l'une des limitations imposées par les choix qui ont été faits quant aux représentations sémantiques. Une autre limitation concerne les noms. Les interpréter comme des propriétés est certainement une solution satisfaisante pour des noms simples. Mais qu'en est-il des noms complexes, comme par exemple ceux suivis d'un groupe prépositionnel ou d'une clause relative ? Par exemple, le fragment « man who loves a woman » peut être vu comme un nom. Cependant, si ce fragment est interprété par un terme de type $\iota \rightarrow o$, il ne lui sera possible ni d'accéder au contexte ni de le modifier. Ceci constitue une seconde limitation de cette première tentative. La seconde, que nous allons présenter maintenant, résout les deux problèmes que nous venons d'évoquer.

Deuxième tentative

Avant d'entrer dans les détails techniques relatifs à cette seconde tentative de paramétrisation des valeurs sémantiques par les contextes et les continuations, nous aimerions essayer de faire partager au lecteur notre compréhension intuitive de ce qui n'a pas fonctionné dans la première paramétrisation. Nous pensons que cela apportera également un éclairage sur la seconde tentative, que nous présenterons par la suite. Lorsque nous avons examiné le cas des groupes nominaux, nous avons commencé par considérer le type qui leur était affecté dans la proposition originelle de Montague ($(\iota \rightarrow o) \rightarrow o$) et nous avons paramétré chaque occurrence de o dans ce type par γ. Mais, si nous nous rappelons que dans la proposition de Montague les phrases étaient interprétées par des propositions, nous remarquons que le type sémantique des groupes nominaux pourrait aussi s'écrire $(\iota \rightarrow s) \rightarrow s$, où s est le type sémantique des phrases. Et, si nous

nous rappelons que nous avons convenu, au tout début de cette section, d'associer aux phrases des termes de type $\gamma \to \delta \to o$, cela suggère que nous pourrions remplacer chaque occurrence de s apparaissant dans le type sémantique des groupes nominaux par ce type. Le type que nous obtenons pour les groupes nominaux après une telle substitution se présente alors comme suit : $(\iota \to \gamma \to \delta \to o) \to \gamma \to \delta \to o$, soit plus simplement $(\iota \to O) \to O$.

De même, considérons les noms et la discussion que nous avons eue à leur sujet à la fin de la sous-section précédente. Nous avions remarqué alors qu'un nom pouvait dépendre d'une phrase, par exemple lorsqu'il est suivi d'une proposition relative. Cette remarque suggère que notre intuition à propos du remplacement de o par s dans les types a du sens. En procédant à ce remplacement dans le type des noms tel qu'il a été initialement proposé par Montague, nous obtenons le type suivant : $\iota \to s = \iota \to \gamma \to \delta \to o$. Et avec un tel type, on constate qu'un nom sera en effet en mesure à la fois d'accéder au contexte et de le modifier.

Pour résumer, nous pensons que le problème de la première tentative était principalement lié à une certaine dose de confusion entre le type o des propositions, d'une part, et le type sémantique des phrases, d'autre part. À notre avis, cette confusion est due au fait que dans la proposition initiale de Montague ces deux types coïncidaient. Avec la proposition de (de Groote, 2006), ceci n'est plus vrai et il convient d'être attentif, lorsqu'on écrit un type, de sorte à bien choisir entre le type des phrases, celui-ci devant être paramétré par un contexte et une continuation, et le type o qui n'a pas besoin d'être paramétré. C'est précisément ce qui est fait dans la deuxième tentative que nous allons présenter maintenant, et c'est pourquoi cette deuxième tentative va fonctionner et éviter les écueils de la première [5].

En ce qui concerne les détails techniques de cette seconde tentative, nous avons en fait déjà commencé à les donner. En effet, les types que nous avons construits pour les noms et les groupes nominaux sont précisément ceux utilisés dans cette approche. Nous pouvons donc donner sans plus tarder les nouvelles représentations sémantiques correspondant à ces types.

Noms Comme nous l'avons vu, le type des noms est $\iota \to \gamma \to \delta \to o$. L'interprétation de « man » est donnée par : $\langle \text{man} \rangle = \lambda x e \phi.(\text{man } x) \wedge \phi e$. D'après ce terme, il est clair que la représentation ne modifie pas le contexte, ce qui correspond bien à ce qu'il doit se passer pour des noms simples tels que « man ».

Groupes nominaux Nous l'avons déjà indiqué, le type que nous souhaitons donner aux représentations des groupes nominaux est le suivant :

$$(\iota \to \gamma \to \delta \to o) \to \gamma \to \delta \to o$$

L'interprétation de « Mary » est alors la suivante : $\langle \text{Mary} \rangle = \lambda \psi e \phi.(\psi m e \lambda e'.\phi(m :: e'))$.

L'interprétation des pronoms est très similaire à la précédente, la seule différence étant que l'individu est extrait du contexte passé en entrée, et que celui-ci n'est pas modifié. Par exemple : $\langle \text{he} \rangle = \lambda \psi e \phi.(\psi(\mathtt{sel}_{\text{he}} e) e \lambda e'.\phi(m :: e'))$.

[5] Bien que, comme nous l'avons vu, la première tentative n'est pas complètement satisfaisante, elle est intéressante parce qu'elle montre avec précision comment les contextes et les continuations fonctionnent, tout en gardant des représentations relativement simples des éléments lexicaux, comparées à celles qui vont suivre. Nous pensons que de Groote a utilisé la première tentative à des fins pédagogiques.

Verbes transitifs Bien sûr, l'interprétation des verbes transitifs doit aussi être modifiée, de sorte à prendre en compte les nouveaux types donnés aux groupes nominaux. L'interprétation de « love » est la suivante : $\langle \text{love} \rangle = \lambda o.\lambda s.s(\lambda x.o\lambda ye\phi.(\text{love } xy) \wedge \phi e)$

Pour bien comprendre la forme de cette représentation, il peut être utile de garder à l'esprit l'idée suivante : les contextes doivent « parcourir » le texte d'entrée, de gauche à droite, exactement comme le fait un lecteur en train de lire le texte, de sorte qu'ils puissent accumuler les entités au fur et à mesure que celles-ci apparaissent dans le texte d'entrée. Cette intuition explique pourquoi c'est le sujet qui est appelé en premier dans la représentation du verbe : il en est ainsi parce que le sujet apparaît avant l'objet syntaxiquement (en tout cas lorsque la forme du verbe est active), donc c'est à lui que le contexte doit être passé en premier, de sorte qu'il puisse le modifier avant de le passer à l'objet.

Déterminants Comme nous l'avons vu précédemment, les interprétations des déterminants doivent être de type $n \to np$, que l'on peut développer partiellement pour obtenir le type suivant :

$$n \to (\iota \to s) \to \gamma \to \delta \to o$$

Les déterminants sont donc des fonctions de 4 arguments dont les types sont respectivement n (le type des noms), $\iota \to s$, γ et $\gamma \to o$. Le terme interprétant le déterminant indéfini est le suivant : $\langle \text{a} \rangle = \lambda n\psi e\lambda\phi.\exists x.(nxe\lambda e'.(\psi xx{::}e'\phi))$. Et celui interprétant le déterminant universel est : $\langle \text{every} \rangle = \lambda n\psi e\phi.(\forall x.\neg(nxe\lambda e'.\neg(\psi xx{::}e'\lambda e''.\top))) \wedge \phi e$

La double négation qui est utilisée ici permet de donner au déterminant universel la bonne sémantique, tout en garantissant que l'entité qu'il introduit n'est passée qu'à la continuation qui en a besoin, et non à celle représentant la suite du discours. Cette représentation modélise ainsi ce qui est modélisé habituellement en DRT par les contraintes d'accessibilité.

Pronoms relatifs Nous sommes intéressés ici par des constructions du type « A man who walks loves a woman ». Comme le montre cet exemple, le pronom relatif prend comme premier argument un groupe verbal, comme second argument un nom et renvoie un nom. Un groupe verbal peut être vu comme une phrase à laquelle manque un groupe nominal. Par conséquent, le type des pronoms relatifs peut s'écrire de la façon suivante :

$$(\langle \text{NP} \rangle \to \langle \text{S} \rangle) \to \langle \text{N} \rangle \to \langle \text{N} \rangle = (\langle \text{NP} \rangle \to \langle \text{S} \rangle) \to \langle \text{N} \rangle \to \iota \to \gamma \to \delta \to o$$

La représentation sémantique du pronom relatif est alors donnée par le terme suivant : $\langle \text{who} \rangle = \lambda rnxe\phi.(nxe\lambda e'.(r(\lambda\psi'.\psi' x)e'\phi))$. Pour comprendre ce terme, souvenons-nous que le pronom relatif coordonne deux fragments syntaxiques, à savoir le nom qui le précède et la clause relative qui le suit. Par conséquent, le nom apparaît dans le texte *avant* la clause relative. C'est pour cette raison que le contexte gauche de toute la construction doit être passé au nom en premier, la clause relative ne pouvant être appelée qu'ensuite avec en argument un contexte qui n'est pas connu à l'avance, puisqu'il s'agit du contexte initial éventuellement mis à jour par le nom. Nous trouvons ces remarques utiles à la compréhension du terme ci-dessus. En effet, comme le montre ce terme, le nom, représenté par la variable n est appelé en premier. Il reçoit comme arguments l'individu x considéré, le contexte gauche de toute la construction syntaxique, et comme troisième argument une continuation que nous allons expliciter maintenant. Elle débute par une

abstraction $\lambda e'$ sur un contexte. Ceci correspond, comme nous l'avons indiqué, au contexte mis à jour par le nom représenté par n et qui devra être passé à r d'une façon ou d'une autre. Puis, comme la clause relative apparaît juste après le pronom relatif qui l'introduit dans le discours, elle doit être appelée en premier. De plus, elle prend trois arguments : le groupe nominal qu'il lui manque pour construire une phrase, un contexte et une continuation. Le groupe nominal constituant le premier argument est construit par type-raising en partant de l'individu considéré x, d'où le terme $\lambda\psi.\psi x$. Le contexte attendu en second argument est le contexte après mise à jour par le nom et qui n'est donc pas disponible directement, c'est pourquoi e' est utilisé. Et finalement, le troisième argument de r est sa continuation, qui est aussi la continuation de toute la construction, à savoir ϕ.

Exemple Pour illustrer cette seconde tentative, réussie celle-là, nous reproduisons l'exemple donné dans (de Groote, 2006), à savoir la célèbre « donkey sentence » « Every farmer who owns a donkey beats it ».

$$
\begin{aligned}
&\langle\text{Every farmer who owns a donkey beats it}\rangle\\
&= \langle\text{beats}\rangle\langle\text{it}\rangle(\langle\text{every}\rangle(\langle\text{who}\rangle(\langle\text{owns}\rangle(\langle\text{a}\rangle\langle\text{donkey}\rangle))\langle\text{farmer}\rangle))
\end{aligned}
$$

$$
\begin{aligned}
&\langle\text{a donkey}\rangle\\
&= (\lambda n\psi e\phi.\exists y.(nye(\lambda e'.(\psi yy{::}e'\phi))))\langle\text{donkey}\rangle\\
&\rightarrow_\beta \lambda\psi e\phi.\exists y.(\langle\text{donkey}\rangle ye(\lambda e'.(\psi yy{::}e'\phi)))\\
&= \lambda\psi e\phi.\exists y.((\lambda xe\phi.(\text{donkey}x) \wedge (\phi e))ye(\lambda e'.(\psi yy{::}e'\phi)))\\
&\rightarrow_\beta \lambda\psi e\phi.\exists y.((\lambda\phi.(\text{donkey } y) \wedge (\phi e))(\lambda e'.(\psi yy{::}e'\phi)))\\
&\rightarrow_\beta \lambda\psi e\phi.\exists y.(\text{donkey } y) \wedge ((\lambda e'.(\psi yy{::}e'\phi))e)\\
&\rightarrow_\beta \lambda\psi e\phi.\exists y.(\text{donkey } y) \wedge (\psi yy{::}e\phi)
\end{aligned}
$$

$$
\begin{aligned}
&\langle\text{owns a donkey}\rangle\\
&= (\lambda os.s(\lambda x.o(\lambda ye\phi.(\text{own}xy) \wedge (\phi e))))\langle\text{a donkey}\rangle\\
&\rightarrow_\beta (\lambda s.s(\lambda x.\langle\text{a donkey}\rangle(\lambda ye\phi.(\text{own}xy) \wedge (\phi e))))\\
&= (\lambda s.s(\lambda x.(\lambda\psi e\phi.\exists y.(\text{donkey } y) \wedge (\psi yy{::}e\phi))(\lambda ye\phi.(\text{own}xy) \wedge (\phi e))))\\
&\rightarrow_\beta (\lambda s.s(\lambda x.(\lambda e\phi.\exists y.(\text{donkey } y) \wedge (\lambda ye\phi.(\text{own } xy) \wedge \phi e)yy{::}e\phi)))\\
&\rightarrow_\beta \lambda s.s(\lambda x.(\lambda e\phi.\exists y.(\text{donkey } y) \wedge (\text{own}xy) \wedge (\phi y{::}e)))
\end{aligned}
$$

$\langle$who owns a donkey$\rangle$
$= (\lambda rnxe\phi.(nxe(\lambda e'.(r(\lambda\psi.\psi x)e'\phi))))\langle$owns a donkey$\rangle$
$\rightarrow_\beta \lambda nxe\phi.(nxe(\lambda e'.(\langle$owns a donkey$\rangle(\lambda\psi.\psi x)e'\phi)))$
$= \lambda nxe\phi.(nxe(\lambda e'.\,(\lambda s.s(\lambda x.(\lambda e\phi.\exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge (\phi y::e))))\,(\lambda\psi.\psi x)e'\phi)))$
$\rightarrow_\beta \lambda nxe\phi.(nxe(\lambda e'.(\lambda\psi.\psi x)(\lambda xe\phi.\exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge (\phi y::e))e'\phi))$
$\rightarrow_\beta \lambda nxe\phi.(nxe(\lambda e'.(\lambda xe\phi.\exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge (\phi y::e))xe'\phi))$
$\rightarrow^*_\beta \lambda nxe\phi.(nxe(\lambda e'.\exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge (\phi y::e')))$

$\langle$farmer who owns a donkey$\rangle$
$= \langle$who owns a donkey$\rangle\langle$farmer$\rangle$
$= \lambda nxe\phi.(nxe(\lambda e'.\exists y.(\text{donkey})y \wedge (\text{own } xy) \wedge (\phi y::e')))\langle$farmer$\rangle$
$\rightarrow_\beta \lambda xe\phi.(\langle$farmer$\rangle xe(\lambda e'.\exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge (\phi y::e')))$
$= \lambda xe\phi.((\lambda xe\phi.(\text{farmer } x) \wedge \phi e)xe(\lambda e'.\exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge (\phi y::e')))$
$\rightarrow_\beta \lambda xe\phi.((\lambda\phi.(\text{farmer } x) \wedge \phi e)(\lambda e'.\exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge (\phi y::e')))$
$\rightarrow_\beta \lambda xe\phi.(\text{farmer } x) \wedge (\lambda e'.\exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge (\phi y::e'))e$
$\rightarrow_\beta \lambda xe\phi.(\text{farmer } x) \wedge \exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge (\phi y::e)$

$\langle$every farmer who owns a donkey$\rangle$
$= \langle$every$\rangle\langle$farmer who owns a donkey$\rangle$
$= (\lambda n\psi e\phi.(\forall x.\neg(nxe(\lambda e.\neg(\psi xx::e(\lambda e.\top))))) \wedge \phi e)$
$\langle$farmer who owns a donkey$\rangle$
$\rightarrow_\beta \lambda\psi e\phi.(\forall x.\neg(\langle$farmer who owns a donkey$\rangle xe(\lambda e.\neg(\psi xx::e(\lambda e.\top))))) \wedge \phi e$
$\rightarrow_\beta \lambda\psi e\phi.(\forall x.\neg((\lambda xe\phi.(\text{farmer } x) \wedge \exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge$
$(\phi y::e))xe(\lambda e.\neg(\psi xx::e(\lambda e.\top))))) \wedge \phi e$
$\rightarrow_\beta \lambda\psi e\phi.(\forall x.\neg((\lambda\phi.(\text{farmer } x) \wedge \exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge$
$(\phi y::e))(\lambda e.\neg(\psi xx::e(\lambda e.\top))))) \wedge \phi e$
$\rightarrow_\beta \lambda\psi e\phi.(\forall x.\neg((\text{farmer } x) \wedge \exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge$
$(\lambda e.\neg(\psi xx::e(\lambda e.\top)))y::e)) \wedge \phi e$
$\rightarrow_\beta \lambda\psi e\phi.(\forall x.\neg((\text{farmer } x) \wedge \exists y.(\text{donkey } y) \wedge (\text{own } xy) \wedge$
$\neg(\psi xx::y::e(\lambda e.\top)))) \wedge \phi e$
$\equiv \lambda\psi e\phi.(\forall x.(\text{farmer } x) \supset (\forall y.((\text{donkey } y) \wedge (\text{own } xy))$
$\supset (\psi xx::y::e(\lambda e.\top)))) \wedge \phi e$

$$
\begin{aligned}
&\langle\text{beats it}\rangle \\
&= \langle\text{beat}\rangle\langle\text{it}\rangle \\
&= (\lambda os.s(\lambda x.o(\lambda ye\phi.(\text{beat } xy) \wedge \phi e)))\langle\text{it}\rangle \\
&\rightarrow_\beta \lambda s.s(\lambda x.\langle\text{it}\rangle(\lambda ye\phi.(\text{beat } xy) \wedge \phi e)) \\
&= \lambda s.s(\lambda x.(\lambda\psi e\phi.(\psi(\mathtt{sel}_{\text{it}}e)e\phi))(\lambda ye\phi.(\text{beat } xy) \wedge \phi e)) \\
&\rightarrow_\beta \lambda s.s(\lambda x.(\lambda e\phi.((\lambda ye\phi.(\text{beat } xy) \wedge \phi e)(\mathtt{sel}_{\text{it}}e)e\phi))) \\
&\rightarrow_\beta \lambda s.s(\lambda xe\phi.(beatx(\mathtt{sel}_{\text{it}}e)) \wedge \phi e)
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{S}\rangle &= \langle\text{beats it}\rangle\langle\text{every farmer who owns a donkey}\rangle \\
&= (\lambda s.s(\lambda xe\phi.(\text{beat } x(\mathtt{sel}_{\text{it}}e)) \wedge \phi e))\langle\text{every farmer who owns a donkey}\rangle \\
&\rightarrow_\beta (\lambda\psi e\phi.(\forall x.(\text{farmer } x) \supset (\forall y.(\text{donkey } y \wedge (\text{own } xy)) \supset (\psi xx{::}y{::}e(\lambda e.\top)))) \wedge \phi e) \\
&\qquad (\lambda xe\phi.(\text{beat } x(\mathtt{sel}_{\text{it}}e)) \wedge \phi e) \\
&\rightarrow_\beta (\lambda e\phi.(\forall x.(\text{farmer } x) \supset (\forall y.(\text{donkey } y \wedge (\text{own } xy)) \\
&\qquad \supset ((\lambda xe\phi.(\text{beat } x(\mathtt{sel}_{\text{it}}e)) \wedge \phi e)xx{::}y{::}e(\lambda e.\top)))) \wedge \phi e) \\
&\rightarrow_\beta \lambda e\phi.(\forall x.(\text{farmer})x \supset (\forall y.(\text{donkey } y \wedge (\text{own } xy)) \\
&\qquad \supset (\text{beat } x(\mathtt{sel}_{\text{it}}x{::}y{::}e)) \wedge \top)) \wedge \phi e)
\end{aligned}
$$

5.2.3 Liens avec la DRT

Comme il est remarqué dans (de Groote, 2006), il est possible de représenter une DRS par un λ-terme de type $\gamma \rightarrow \delta \rightarrow o$. Ainsi, la DRS $[x_1, \ldots, x_n|\gamma_1, \ldots, \gamma_m]$ peut être représentée par le λ-terme suivant : $\lambda e.\lambda\phi.\exists x_1 \ldots x_n.\gamma_1 \wedge \ldots \wedge \gamma_m \wedge \phi(e')$, où e' est un contexte construit à partir de e et en y ajoutant les variables $x_1, \ldots, x_n$. Notons toutefois que la correspondance qui vient d'être donnée entre les DRSs et certains termes de type O n'est valide que dans le cadre de la DRT standard. En particulier, elle ne s'étend pas trivialement à la DRT considérée dans (Muskens, 1996c), à cause du fait que les référents de discours peuvent être des constantes dans cette version étendue.

En outre, une traduction en sens inverse, c'est-à-dire un encodage des termes de type O en DRT est beaucoup moins aisé. En effet, les continuations de (de Groote, 2006) permettent certes de modéliser les contraintes d'accessibilité de la DRT, comme on a pu s'en rendre compte jusqu'ici, mais elles permettent aussi de modéliser d'autres contraintes, que la DRT ne permet pas d'exprimer à elle seule. Comme exemple de telles contraintes qui ne peuvent être modélisées en DRT et qui peuvent l'être dans le cadre qui vient d'être présenté, nous pouvons citer le travail décrit dans (Pogodalla, 2008). Dans ce travail, on montre comment l'utilisation de deux contextes permet d'isoler les entités introduites par des articles définis et que l'on peut souhaiter traiter d'une façon spécifique. Pour reprendre l'exemple des auteurs de ce dernier travail, considérons le discours « John does not own a car. *It is red. ». Ici, la seconde phrase ne peut être interprétée car, du fait de la négation, la voiture introduite à la première phrase n'est plus accessible.

Cependant, John doit l'être, car affirmer qu'il ne possède pas de voiture ne remet pas en question son existence et l'on pourrait continuer le discours en affirmant : « He owns a bike », suite parfaitement acceptable.

Comme la DRT ne considère dans sa définition de l'accessibilité qu'un seul univers de référents de discours, il n'est pas possible en DRT de distinguer John de sa voiture, comme on aimerait pouvoir le faire dans l'exemple précédent. Le travail précédemment cité montre que le système de type introduit dans (de Groote, 2006) rend cette distinction possible. D'autres distinctions le sont également. On peut par exemple imaginer de placer dans des contextes distincts les entités qui ont été introduites comme sujet des verbes et celles introduites comme objet, ce qui facilite grandement l'implantation de méthodes de résolution des anaphores utilisant des notions de parallélisme syntaxique. Le système de type introduit dans (de Groote, 2006) pourrait donc aussi être utilisé pour rendre compte de théories telles que la théorie du liage introduite dans (Büring, 2005).

5.2.4 Liens avec `Nessie`

Comme nous le verrons au chapitre suivant, il est facile avec `Nessie` de calculer des représentations sémantiques telles que celles qui ont été calculées précédemment. En attendant de voir comment y parvenir, nous aimerions faire remarquer deux points qui nous semblent importants.

D'une part, rappelons que nous avons donné au chapitre 2 deux versions de l'algorithme de construction de représentations sémantiques. La première version faisait l'hypothèse que toutes les occurrences d'un lexème dans un arbre syntaxique avaient des représentations α-équivalentes, tandis que la deuxième permettait de traiter des théories sémantiques qui affectent des représentations non α-équivalentes à différentes occurrences d'un même lexème. Il convient de remarquer que, dans le traitement compositionnel de la dynamicité, toutes les occurrences d'un lexème ont des représentations α-équivalentes. De ce fait, la première version de l'algorithme présenté au chapitre 2 aurait été suffisante pour traiter le formalisme qui a été introduit dans (de Groote, 2006) et ce n'est pas ce formalisme qui a nécessité le passage de la première à la seconde version de l'algorithme de construction sémantique. Ainsi, bien que l'implantation dont nous allons parler au chapitre suivant utilise la seconde version de l'algorithme de construction sémantique, ce n'est que pour des raisons de simplicité, et on aurait tout aussi bien pu utiliser la première version de l'algorithme pour traiter les représentations à base de contextes et de continuations.

D'autre part, nous aimerions établir des rapprochements entre (de Groote, 2006) et la seconde version de l'algorithme de construction sémantique, c'est-à-dire celle effectivement implantée dans `Nessie`. Plus précisément, ces rapprochements concernent les environnements de typage introduits au chapitre 2 et les contextes, objets de type γ, utilisés dans (de Groote, 2006). Rappelons qu'un environnement donne un type à des constantes et à des variables. Or, nous avons vu qu'un contexte, en tout cas tel qu'envisagé jusqu'ici, n'est rien d'autre qu'un ensemble fini d'entités. En d'autres termes, un contexte peut être vu comme un ensemble de constantes et de variables de type ι, c'est-à-dire une sorte d'environnement dans lequel tous les objets seraient de type ι. Un environnement peut donc être compris comme une généralisation de la notion de contexte introduite jusqu'ici, dans le sens où un environnement ne se restreint pas à des objets d'un seul type. Rien n'empêche cependant de définir des contextes plus riches, contenant des

objets de plusieurs types. Il n'est cependant pas évident que l'on pourrait simuler à l'aide de contextes notre notion d'environnement, puisqu'il faudrait pour cela que les contextes puissent contenir des constantes et variables de types *arbitraires* non prévus à l'avance. Cependant, le fait que les contextes ne disposent pas d'une telle propriété ne suffit pas à rendre les environnements préférables, et cela pour deux raisons.

La première raison est que nous ne connaissons pas, pour l'instant, d'application où l'on pourrait avoir besoin de contextes pouvant contenir des données de n'importe quel type, c'est-à-dire de contextes véritablement polymorphes. La deuxième raison est que les contextes ont sur les environnements de `Nessie` un avantage décisif, que nous allons tâcher d'expliquer maintenant. Comme nous l'avons vu précédemment, l'encodage de la DRT de (Muskens, 1996c) permet certes de vérifier qu'une DRS vérifie les contraintes d'accessibilité, mais n'empêche pas la construction de DRSs qui ne les vérifient pas. Ainsi, pour le discours « John does not own a donkey. He beats *it. », si l'individu introduit par « a donkey » est représenté par la constante u_2 de type π, rien n'empêche de faire référence à cette constante dans la seconde phrase et en particulier de l'utiliser lors de la résolution anaphorique du pronom « it ». Il sera certes possible de prévoir qu'une telle DRS viole les contraintes d'accessibilité, mais on ne peut pas garantir, sans vérification explicite, qu'un terme donné vérifie *par construction* les contraintes d'accessibilité.

Or il en va tout autrement dans la proposition de (de Groote, 2006), où les contextes sont internalisés. Dans cette théorie, l'ensemble des entités accessibles et l'ensemble des constantes et variables apparaissant dans un environnement de typage sont distingués l'un de l'autre, et l'ensemble des individus accessibles, c'est-à-dire le contexte, devient un « citoyen de première classe », manipulable au sein de la théorie exactement comme le sont les autres objets. Il est donc possible de passer à chaque représentation sémantique un contexte, c'est-à-dire l'ensemble des individus accessibles, ce qui permet de garantir par construction que les contraintes d'accessibilité sont respectées, et cela indépendemment des algorithmes utilisés pour résoudre les anaphores. Dans l'exemple qui vient d'être donné, les continuations et les contextes permettent de modéliser l'individu introduit par « a donkey » par une variable que l'on peut ou non faire apparaître dans le contexte passé à la phrase suivante. Si cette variable n'apparaît pas dans le contexte passé à « He beats it. », on peut garantir que, *par construction*, le terme représentant cette phrase respecte les contraintes d'accessibilité, et cela en particulier parce que le contexte utilisé lors de la résolution anaphorique du pronom « it » ne contient pas l'individu introduit par « a donkey ».

En résumé, nous pouvons dire que la notion d'environnement global et « dynamique » (c'est-à-dire s'enrichissant de nouvelles constantes pendant le parcours de l'arbre syntaxique) qui a été introduite dans `Nessie` est certes utile pour des théories traitant la dynamicité en donnant des représentations non α-équivalentes à différentes occurrences d'un même lexème, mais que cette notion d'environnement n'est pas nécessaire à tous les traitements de la dynamicité. En effet, comme le montre l'approche introduite dans (de Groote, 2006), la dynamicité peut être traitée de façon complètement compositionnelle, sans qu'il soit nécessaire de donner à différentes occurrences d'un même lexème des représentations non α-équivalentes, et en distinguant, d'une part les environnements de typage et, d'autre part, les contextes qui gardent trace d'un ensemble d'entités accessibles à un moment donné du processus de construction sémantique.

Chapitre 6

Calculer automatiquement la représentation des discours

Au début du chapitre précédent, nous avons expliqué que, pour poursuivre les expérimentations qui doivent nous permettre de répondre à la question de la pertinence de TYn en sémantique computationnelle, il était nécessaire de disposer d'un moyen de construire compositionnellement la sémantique de discours. C'est cette constatation qui nous a amenés à présenter, au chapitre précédent, deux outils développés dans le cadre de la sémantique formelle et qui permettent de calculer compositionnellement la sémantique de discours constitués de plusieurs phrases, à savoir la DRT compositionnelle introduite dans (Muskens, 1996c) et le traitement compositionnel de la dynamicité introduit dans (de Groote, 2006). La présentation de ces outils qui a été donnée nous a certes permis d'en établir certaines propriétés, mais une mise en œuvre effective demeure à notre avis indispensable pour se faire une idée précise de l'intérêt que peuvent avoir ces outils sur le plan computationnel. En outre, si nous voulons les utiliser pour entreprendre des expérimentations telles que celles qui ont été menées au chapitre 4, disposer d'une version opérationnelle de ces outils est nécessaire.

C'est donc à la mise en œuvre de la DRT compositionnelle et du traitement compositionnel de la dynamicité qu'est consacrée une partie importante de ce chapitre. Puisque cette mise en œuvre est effectuée au sein de la version modifiée de `Curt` développée au chapitre 3, nous commençons, à la section 6.1, par présenter les quelques modifications qu'il a été nécessaire d'apporter à la grammaire obtenue à la fin du chapitre 3 pour que les arbres syntaxiques qu'elle produit puissent être utilisés avec les deux formalismes considérés ici. Cette section sera aussi l'occasion d'une discussion d'ordre plus général sur la conception de grammaires utilisables avec plusieurs formalismes sémantiques. Nous montrerons en particulier en quoi l'utilisation d'un outil tel que `Nessie` facilite aussi celle d'une même grammaire avec plusieurs formalismes sémantiques, et nous ferons quelques suggestions permettant de rendre une grammaire aussi indépendante que possible d'un formalisme sémantique spécifique. Dans les sections 6.2 et 6.3, nous présentons tour à tour l'implantation de la DRT compositionnelle puis celle du traitement compositionnel de la dynamicité. Enfin, dans la section 6.4, nous nous appuyons sur le travail réalisé dans la section précédente pour entreprendre l'étude de la résolution d'anaphores dans TYn. Ce faisant, nous poursuivons les expérimentations entreprises au chapitre 4, mais cette fois dans un cadre plus général, puisque nous ne sommes plus limités à la construction de la sémantique de phrases isolées et pouvons désormais construire celle de discours constitués de plusieurs phrases. Comme on pourra le constater, la résolution d'anaphores, au-delà de son utilité immédiate, nous apportera d'autres éléments de réponse à la question de la pertinence de

TYn en sémantique computationnelle, centrale dans cette thèse. En section 6.5, nous ferons un bilan de ces éléments de réponse, ainsi que des enseignements qui peuvent être tirés de la mise en œuvre des deux théories sémantiques.

6.1 La grammaire et les arbres syntaxiques associés

Pour illustrer les approches qu'ils proposent pour calculer compositionnellement des représentations sémantiques de discours, Muskens et de Groote s'appuient chacun sur une grammaire de très petite taille qui leurs permet de montrer comment se fait le passage de la syntaxe à la sémantique. Pour notre part, nous faisons ici le choix de n'utiliser aucune de ces deux grammaires, mais plutôt de nous en tenir à celle que nous avons utilisée précédemment pour la validation de `Nessie`. Ce choix a plusieurs justifications, dont la première est des plus pragmatiques. Déjà en possession d'une grammaire opérationnelle et capable de produire des arbres utilisables par `Nessie`, pourquoi en adopterions-nous une autre, alors que notre but n'est pas de tester l'interface entre syntaxe et sémantique, mais bien la construction sémantique pour plusieurs théories. De plus, si nous arrivons, à partir d'une même grammaire, à produire des représentations sémantiques dans plusieurs formalismes différents, cela devrait constituer une preuve assez convaincante de l'intérêt de `Nessie` pour la comparaison de théories sémantiques.

Nous nous en tenons donc à la grammaire déjà utilisée par Blackburn et Bos dans `Curt`, que nous avions réutilisée au chapitre 3 après l'avoir modifiée pour qu'elle soit en mesure de produire des arbres utilisables par `Nessie`. Il est cependant nécessaire d'apporter à cette grammaire deux modifications avant d'être en mesure de l'utiliser pour les tâches qui nous intéressent dans ce chapitre. L'une consiste en une amélioration des arbres produits par les coordinations, l'autre est une extension de la grammaire aux pronoms personnels. Cette extension ne sera pas utilisée dans la section 6.2, mais nous commencerons à l'utiliser à la section 6.3 et surtout à la section 6.4.

6.1.1 Amélioration des arbres pour les coordinations

Lorsque nous avons voulu reconstruire les représentations sémantiques construites initialement par `Curt` à l'aide de `Nessie`, nous avons dû pour y parvenir choisir un système de types approprié. Nous avions alors choisi TY1 et nous avons décidé de représenter les phrases par des λ-termes de type t, c'est-à-dire des propositions. Considérons cependant les trois règles de DCG intervenant dans la construction de l'arbre d'analyse des phrases du type « if ... then ... » :

```
s([ast:binary(s,AstS1,AstS2),coord:yes,sem:Sem])-->
   s([ast:AstS1,coord:ant,sem:S1]),
   s([ast:AstS2,coord:con,sem:S2]),
   {combine(s:Sem,[s:S1,s:S2])}.

s([ast:nary(s,'lam B. lam S : t. (B=>S)',AstS),coord:ant,sem:Sem])-->
   [if],
   s([ast:AstS,coord:no,sem:S]),
   {combine(s:Sem,[if:S])}.

s([ast:unary(s,AstS),coord:con,sem:Sem])-->
```

```
[then],
s([ast:AstS,coord:no,sem:S]),
{combine(s:Sem,[then:S])}.
```

La première règle affirme que la phrase « If ... then ... ! » est la concaténation d'un antécédent « if ... » et d'une conséquence « then ». En outre, cette règle nous apprend que l'arbre syntaxique qui va être construit est un arbre binaire ayant pour fils gauche l'arbre de l'antécédent et pour fils droit l'arbre du conséquent. Et, compte tenu de ce que nous savons sur la façon dont `Nessie` construit la représentation sémantique de tels arbres binaires, nous pouvons déduire que la représentation sémantique de l'arbre qui nous intéresse sera obtenue en appliquant celle du sous-arbre de gauche (celui de l'antécédent) à celle du sous-arbre de droite (celui du conséquent). Pour que cela soit possible, il faut que les types des termes représentant les deux sous-arbres soient compatibles avec l'application, ce qui est bien le cas, comme on peut le déduire des deux règles de DCG suivantes. Considérons en effet la deuxième règle, construisant l'arbre de l'antécédent « if ... ! ». Cet arbre est un arbre n-aire obtenu en appliquant le λ-terme donné dans la DCG à la représentation de la phrase suivant le « if ». Comme la structure suivant le « if » est une phrase, sa représentation est un terme de type t, et donc le λ-terme construit pour représenter l'antécédent « if ... » est de type $t \rightarrow t$. Enfin, d'après la troisième règle, la conséquence « then ... » est représentée par un terme de type t, qui peut bien être passé en argument à la représentation de l'antécédent, comme l'exige la première règle.

Cette façon de construire les arbres d'analyse pour les conditionnelles peut paraître complexe, mais, au chapitre 3, elle ne pouvait être évitée. En effet, notre objectif était alors que les représentations sémantiques construites à l'aide de `Nessie` soient α-équivalentes à celles produites par `Curt`, et cela *avant* β-réduction, ce qui imposait des contraintes très fortes sur la construction des arbres syntaxiques dont découle celle des représentations sémantiques.

Ces contraintes ayant disparu, nous pouvons maintenant porter un regard critique sur cette façon de construire les arbres syntaxiques. En particulier, nous aimerions citer deux inconvénients de cette méthode. D'une part, on pourrait souhaiter que toutes les occurrences d'un non-terminal soient représentées par des termes de même type. Par exemple, il paraît cohérent d'envisager que toutes les phrases soient représentées par des λ-termes de type t. Or, nous venons de le voir, les arbres construits au chapitre 3 ne remplissent pas cette condition. En effet, les structures « if ... » sont vues comme étant des phrases d'un point de vue syntaxique, et pourtant leur représentation sémantique est un terme de type $t \rightarrow t$. D'autre part, et ce second inconvénient est à notre avis le plus important, il convient de remarquer que les arbres syntaxiques construits au chapitre 3 font *explicitement* l'hypothèse que les phrases sont représentées par des termes de type t. Or, nous l'avons vu au chapitre précédent, cela n'est pas toujours le cas. Concrètement, cela signifie que les arbres construits au chapitre 3 ne peuvent être utilisés qu'avec des théories sémantiques représentant les phrases par des objets de type t. Ceci représente évidemment une limitation importante des arbres syntaxiques tels qu'ils sont construits actuellement. Nous pourrions même dire que de tels arbres ne sont pas purement syntaxiques, puisqu'ils font aussi des hypothèses concernant la sémantique. Pour prolonger cette remarque, notons que, comme nous le verrons au chapitre suivant, si l'on construit de tels arbres, la propriété de simulation de `Nessie` par une grammaire catégorielle abstraite est perdue, ce qui constitue une raison de plus de les éviter autant que faire se peut.

La modification que nous allons proposer résout ce problème et permet d'utiliser les arbres syntaxiques avec des théories sémantiques arbitraires. Cette modification consiste essentiellement à enlever toute hypothèse sémantique des arbres syntaxiques. Pour y parvenir, nous utilisons le fait que `Nessie` permet non seulement de déclarer des constantes, mais aussi, comme nous l'avons vu au chapitre 2, de *définir* des constantes, c'est-à-dire de leur associer, en plus de leur type, une valeur par laquelle elles seront remplacées lors de la construction sémantique.

Pour en revenir un instant au chapitre 3, nous pourrions par exemple ajouter au lexique de `Nessie` la déclaration suivante :

```
const implication : t -> t -> t := lam S1, S2 : t. (S1 => S2);
```

La constante `implication` ainsi définie peut ensuite figurer dans l'arbre syntaxique, par exemple dans les λ-termes utilisés pour construire la représentation des arbres n-aires. Ainsi, les trois règles de DCG que nous avons donné précédemment peuvent être remplacées par les règles suivantes :

```
s([
   ast:nary(s,'lam S1, S2. (implication(S1,S2))',AstS1,AstS2),
   coord:yes,sem:Sem
  ])-->
   s([ast:AstS1,coord:ant,sem:S1]),
   s([ast:AstS2,coord:con,sem:S2]),
   {combine(s:Sem,[s:S1,s:S2])}.

s([
   ast:unary(s,AstS),
   coord:ant,sem:Sem
  ])-->
   [if],
   s([ast:AstS,coord:no,sem:S]),
   {combine(s:Sem,[if:S])}.

s([ast:unary(s,AstS),coord:con,sem:Sem])-->
   [then],
   s([ast:AstS,coord:no,sem:S]),
   {combine(s:Sem,[then:S])}.
```

Comme on peut le constater, la deuxième et la troisième règle reconnaissant respectivement un antécédent et une conséquence se contentent de faire remonter les représentations sémantiques des phrases sans les modifier, par le biais d'arbres unaires. Quant à la première règle, l'arbre qu'elle construit est un arbre n-aire obtenu en fournissant à la constante `implication` les deux arguments dont elle a besoin, à savoir un antécédent et une conséquence. Si `implication` est remplacée par sa définition pendant la construction sémantique, la représentation sémantique obtenue *après* β-réduction sera exactement la même que celle obtenue avec l'arbre initial. Certes, l'α-équivalence des représentations avant réduction est perdue, mais cela n'est pas un problème. En outre, l'arbre ainsi construit ne fait plus aucune hypothèse quant aux types des représentations sémantiques manipulées et peut donc être utilisé avec n'importe quelle théorie

sémantique. Le seul pré-requis est que chaque théorie définisse une constante `implication` prenant en arguments des représentations sémantiques de phrases et renvoyant une autre représentation sémantique de phrase, les types des arguments et de la valeur retournée pouvant varier d'une théorie à l'autre.

Remarquons que les arbres construits au chapitre 3 pour les disjonctions de la forme « either ... or ... » posent exactement le même genre de problèmes et que ces problèmes peuvent être résolus exactement comme nous venons de le faire pour les implications. Nous apportons donc à la grammaire une modification similaire à celle que nous venons de présenter, mais cette fois concernant les disjonctions.

Avant d'examiner l'extension de la grammaire aux pronoms personnels, nous aimerions nous arrêter un instant sur les modifications que nous venons d'apporter pour faire quelques remarques à leur sujet.

D'une part, nous pouvons tirer des modifications que nous venons de présenter un enseignement précieux. En effet, ces modifications nous montrent que, plus un arbre syntaxique sera abstrait et de haut niveau, plus cet arbre pourra être réutilisé facilement avec des théories sémantiques très différentes les unes des autres. En particulier, si l'on souhaite construire, à partir d'une même grammaire, des représentations sémantiques dans des formalismes différents, il est indispensable d'éviter autant que possible de faire apparaître dans l'arbre des éléments spécifiques à une théorie, comme par exemple des informations de typage.

D'autre part, nous aimerions faire remarquer que `Nessie` facilite, par sa conception, la réutilisation d'arbres syntaxiques. De façon plus précise, le fait que les termes utilisés dans les arbres n-aires puissent faire appel à des constantes sans avoir à donner explicitement des informations de typage est une incitation non négligeable à concevoir des arbres abstraits et donc non liés à une théorie sémantique particulière.

6.1.2 Pronoms

L'extension de notre DCG aux pronoms est réalisée en y ajoutant des clauses stipulant qu'un groupe nominal peut prendre la forme d'un pronom, ce qui nécessite d'ajouter aux groupes nominaux un attribut (trait) `fun` qui peut prendre les valeurs `sub` et `obj` pour gérer la distinction entre pronoms sujets et objets. On ajoute par ailleurs une famille pour les pronoms au lexique syntaxique de `Curt` :

```
lexEntry(pro,[symbol:selhe,syntax:[he],fun:sub]).
lexEntry(pro,[symbol:selshe,syntax:[she],fun:sub]).
lexEntry(pro,[symbol:selit,syntax:[it],fun:_]).
lexEntry(pro,[symbol:selhim,syntax:[him],fun:obj]).
lexEntry(pro,[symbol:selher,syntax:[her],fun:obj]).
```

Ces déclarations associent à chaque pronom l'opérateur de choix d'antécédent qui lui correspond. Ce sont donc ces opérateurs qu'il conviendra de définir dans les lexiques sémantiques de `Nessie` pour que le support des pronoms soit complet. Comme nous l'avons indiqué plus haut, nous ne chercherons pas à construire de représentations sémantiques pour des textes avec pronoms en DRT compositionnelle, pour les raisons qui ont été données au chapitre précédent. Il faudra donc attendre la section 6.3 pour voir des exemples utilisant des pronoms.

6.2 Implantation de la DRT compositionnelle de Muskens

Nous allons maintenant voir comment construire des DRSs pour les phrases engendrées par la grammaire précédente dans le formalisme de Muskens. Comme nous l'avons indiqué lors de la présentation de ce formalisme, celui-ci peut associer des représentations distinctes à deux occurrences distinctes d'une même entrée lexicale. Cela se produit par exemple pour les déterminants, chacune de leurs occurrences étant supposée annotée de façon unique par un nombre qui est utilisé pour générer un référent de discours, lui aussi unique.

Or, nous avons expliqué au chapitre 2 comment il est possible de faire en sorte que `Nessie` construise des représentations différentes pour une même entrée lexicale. Il suffit de donner aux feuilles correspondant aux entrées lexicales dont les occurrences doivent être distinguées des arguments qui seront vus comme des constantes et pourront à ce titre figurer dans la représentation de la feuille correspondante. Cependant, les arbres syntaxiques construits par la grammaire que nous utilisons ne permettent pas, pour le moment, de distinguer deux occurrences d'un déterminant. Par exemple, l'arbre produit par notre grammaire lors de l'analyse de la phrase « A man loves a woman. » est le suivant :

```
unary(t,
      binary(s,
             binary(np,
                    leaf(indef, det),
                    unary(n, leaf(man, noun))
             ),
             binary(vp,
                    leaf(love, tv),
                    binary(np,
                           leaf(indef, det),
                           unary(n, leaf(woman, noun))
                    )
             )
      )
)
```

Comme on peut le constater, les deux occurrences du déterminant indéfini sont représentées par la même feuille `leaf(indef, det)`, ce qui supprime toute possibilité pour `Nessie` de distinguer les occurrences entre elles et donc de produire des représentations spécifiques à chaque occurrence. Notre première tâche est donc de résoudre ce problème et de faire en sorte que l'arbre syntaxique soit porteur d'informations permettant de distinguer les occurrences des lexèmes pour lesquels on souhaite construire une représentation distincte pour chaque occurrence.

Ce résultat peut être obtenu grâce à un prédicat `Prolog` procédant à l'annotation d'arbres syntaxiques. Ce prédicat prend en entrée un arbre tel que celui que nous venons de montrer et renvoie en sortie un arbre dont les feuilles portant des déterminants ont un argument, cet argument étant propre à chaque feuille. Pour garantir l'unicité de l'argument, les feuilles contenant des déterminants sont comptées et la i-ième feuille reçoit l'argument u_i.

Si nous appelons ce prédicat d'annotation sur l'arbre précédent, nous obtenons en sortie l'arbre suivant :

```
unary(t,
      binary(s,
             binary(np,
                    leaf(indef, det,u1),
                    unary(n, leaf(man, noun))
             ),
             binary(vp,
                    leaf(love, tv),
                    binary(np,
                           leaf(indef, det,u2),
                           unary(n, leaf(woman, noun))
                    )
             )
      )
)
```

Cette fois, les feuilles représentant les deux occurrences du déterminant indéfini sont bien distinctes l'une de l'autre, puisque la première reçoit comme argument `u1` tandis que la seconde reçoit en argument `u2`. Notons que toutes les feuilles appartenant à la famille des déterminants sont ainsi comptées et annotées, ce qui signifie que, conformément aux préconisations de Muskens, on distingue non seulement toutes les occurrences des déterminants indéfinis, mais aussi toutes celles des déterminants universels et des déterminants négatifs.

Nous pouvons maintenant passer au lexique. Comme nous l'avons vu lorsque nous avons présenté l'approche de Muskens, il est indispensable de n'utiliser dans le lexique que les formes abrégées des λ-termes. En effet, si l'on ne se plie pas à cette discipline, les termes que l'on obtient sont non seulement quasi-illisibles, mais encore impossibles à simplifier, puisque la procédure de fusion de DRSs ne peut s'appliquer qu'aux formes abrégées. Nous commençons par déclarer les types dont nous avons besoin :

```
type e; # Entities
type pi; # Registers
type s; # States
type drs = s -> s -> t;
type condition = s -> t;
```

Viennent ensuite les déclarations de plusieurs symboles utiles : l'égalité et la différence entre entités, la constante V permettant de trouver le contenu d'un registre dans un état donné, l'égalité entre états (dont une définition extensionnelle est donnée) et la propriété pour deux états de différer en au plus un registre, dont nous aurons besoin par la suite :

```
const eq : e -> e -> t;
const neq : e -> e -> t;
const V : pi -> s -> e;
const seq : s -> s-> t :=
  lam s1, s2 : s.
  forall delta : pi. (V(delta,s1) = V(delta,s2));
const diff1 : s -> s -> pi -> t :=
  lam s1, s2 : s. lam delta0 : pi.
```

```
  forall delta : pi.
  (delta != delta0 => V(delta,s1) = V(delta,s2));
```

Le lexique se poursuit par la déclaration de constantes qui peuvent être vues comme des constructeurs de DRSs et de conditions, au sens donné à ce terme lorsque nous avons introduit le langage `OCaml` au chapitre 2[1].Comme on va le voir, la plupart de ces constructeurs reflètent fidèlement les abréviations introduites par Muskens et dont nous avons laissé les références en commentaires lorsque cela était pertinent. Voici donc ces constructeurs :

```
const drs_of_dr : pi -> drs :=
  lam u : pi. lam s1, s2 : s. (diff1(s1,s2,u));

const drs_of_condition : condition -> drs :=
  lam c : condition.
  lam i, j : s.
  (seq(i,j) && c(j));
const merge : drs -> drs -> drs := # ABB4
  lam D1, D2 : drs.
  lam i, j : s.
  exists k : s.
  (D1(i,k) && D2(k,j));

const condition_of_urel : (e -> t) -> pi -> condition := # ABB1
  lam P : e -> t. lam v : pi. lam i : s.
  ( P(V(v,i)) );
const condition_of_brel :
  (e -> e -> t) -> pi -> pi -> condition := # ABB1
  lam P : e -> e -> t. lam v1, v2 : pi. lam i : s.
  ( P(V(v1,i),V(v2,i)) );

const drsnot : drs -> condition := # ABB2
  lam D : drs.
  lam i : s. ! exists j : s. (D(i,j));
const drsor : drs -> drs -> condition := # ABB2
  lam D1, D2 : drs.
  lam i : s. exists j : s.
  (D1(i,j) || D2(i,j));

const drsimp : drs -> drs -> condition := # ABB2
  lam D1, D2 : drs.
  lam i : s. forall j : s.
  (D1(i,j) => (exists k : s. (D2(j,k))));
```

Les deux premières constantes permettent de construire des DRSs contenant soit un référent de discours, soit une condition. Ainsi, si u_1 est un référent de discours et γ_1 une condition, alors la DRS notée $[u_1|]$ par Muskens sera représentée dans notre lexique par le terme

[1]Il faut cependant remarquer que les constructeurs dont il est ici question ne permettent pas de filtrage et ne définissent d'ailleurs pas un type inductif.

`drs_of_dr(u1)`, tandis que la DRS que Muskens note $[|\gamma_1]$ sera représentée dans notre lexique par le terme
`drs_of_condition(gamma1)`. La troisième constante, `merge`, permet de construire une DRS à partir de deux DRSs existantes. Elle correspond en fait à l'opérateur ; de Muskens, et nous aurions donc tout aussi bien pu l'appeler `seq` pour indiquer qu'elle représente la séquence de deux DRSs. Le nom `merge` ne nous paraît cependant pas tout à fait illégitime, puisque les deux DRSs qui sont ainsi « séquencées » sont appelées à être fusionnées.

Les deux constantes suivantes, `condition_of_urel` et `condition_of_brel` permettent de construire des conditions atomiques à partir de relations unaires et binaires. Ces constructions sont des cas particuliers de l'abréviation 1 de Muskens. Ces deux constantes sont suffisantes, car les seules relations utilisées en pratique sont unaires ou binaires.

Enfin, les trois dernières constantes permettent de construire des conditions à partir de DRSs.

Le lexique se poursuit par la définition des constantes `implication` et `disjunction` que nous avons introduites dans la section précédente. Rappelons que ces constantes apparaissent dans les arbres d'analyse produits pour les phrases du type « if ... then... » et « either ... or ... ». Ces constantes sont définies de la façon suivante :

```
const implication : drs -> drs -> drs :=
  lam D1, D2 : drs.
  (drs_of_condition(drsimp(D1,D2)));
const disjunction : drs -> drs -> drs :=
  lam D1, D2 : drs.
  (drs_of_condition(drsor(D1,D2)));
```

Enfin, le lexique se termine par la déclaration de toutes les familles et de tous les lemmes nécessaires. Il s'agit en fait des mêmes familles et des mêmes lemmes que ceux rencontrés au chapitre 3, puisque la grammaire utilisée est la même. Seules les représentations associées aux lemmes changent, mais sans réserver de surprises particulières. Contentons-nous donc de présenter les représentations des déterminants, qui utilisent les arguments d'occurrences issus de l'étape d'annotation d'arbres syntaxiques mentionnée plus haut.

```
family det;

lemma indef {
  family = det;
  const $1 : pi;
  term =
  lam P0, P : pi->drs.
  ( merge(drs_of_dr($1),merge(P0($1),P($1))) )
};

lemma uni {
  family = det;
  const $1 : pi;
  term =
  lam P0, P : pi->drs.
  ( drs_of_condition(drsimp(merge(drs_of_dr($1),P0($1)),P($1))) )
};
```

```
lemma neg {
  family = det;
  const $1 : pi;
  term =
  lam P0, P : pi->drs.
  ( drs_of_condition(
    drsnot(merge(drs_of_dr($1),merge(P0($1),P($1)))))
  )
};
```

On le voit, les termes associés aux trois lemmes sont des traductions fidèles de ceux proposés par Muskens dans (Muskens, 1996c).

Pour en terminer vraiment avec les remarques se rapportant au lexique, signalons que les observations relatives aux coordinations qui ont été faites au chapitre 3 restent de mise et que le traitement des coordinations qui est proposé ici est une adaptation à la théorie de Muskens de celui que nous avons utilisé au chapitre 3, à savoir que nous utilisons autant de familles de coordinations qu'il y a de catégories syntaxiques pouvant être coordonnées, donc ici une pour les noms, une pour les groupes nominaux et une pour les groupes verbaux.

Grâce à ce lexique, à la grammaire et au prédicat d'annotation d'arbres qui ont été présentés précédemment, nous sommes enfin en mesure de calculer automatiquement une représentation sémantique pour la phrase sur laquelle se sont appuyés nos exemples jusqu'ici, à savoir « A man loves a woman. ». Si nous transmettons son arbre syntaxique annoté à `Nessie` et demandons à ce dernier d'afficher son résultat sous la forme d'un terme `Prolog`, voici ce que nous obtenons :

```
merge(
  drs_of_dr(u1),
  merge(
    drs_of_condition(condition_of_urel(man,u1)),
    merge(
      drs_of_dr(u2),
      merge(
        drs_of_condition(condition_of_urel(woman,u2)),
        drs_of_condition(condition_of_brel(love,u1,u2))
      )
    )
  )
).
```

Bien qu'il soit possible, au prix d'un petit effort, de se convaincre que ce terme représente bien la DRS recherchée, on ne peut s'empêcher de ressentir, en le contemplant, une certaine frustration. En effet, ce terme est complexe, surtout au vu de la phrase au demeurant fort simple dont il est sensé représenter le sens ! En effet, si une phrase aussi simple que « A man loves a woman. » produit un terme si peu lisible que celui-ci, qu'en sera-t-il pour des phrases plus complexes, et *a fortiori* pour de véritables discours tels que ceux que l'on peut rencontrer dans le cadre d'applications réalistes ? Heureusement, la situation n'est pas aussi désespérée qu'il y paraît. En effet, en observant bien le terme que nous venons d'obtenir, on s'aperçoit qu'il contient plusieurs occurrences de la constante `merge` qui représente la fusion des deux DRSs

qu'elle reçoit en argument. Or, comme nous l'avons vu précédemment, sous certaines conditions deux DRSs peuvent être fusionnées, la DRS obtenue regroupant tous les référents de discours et toutes les conditions présents dans les DRSs à fusionner. En outre, nous avons vu que la simplification de DRS que Muskens utilise dans son algorithme de traduction fait intervenir non seulement la β-réduction, mais aussi cette règle de fusion de DRSs. En d'autres termes, avec la représentation que nous venons d'obtenir, nous pourrions dire que nous n'avons parcouru que la moitié du chemin conduisant à une DRS telle qu'envisagée par Muskens. En effet, la seule règle qui a été exploitée pour simplifier les DRSs est la β-réduction. La règle permettant de fusionner des DRSs n'a, pour sa part, pas encore été utilisée.

Il nous faut donc trouver un moyen d'appliquer cette règle jusqu'à ce que la DRS obtenue ne contienne plus d'occurrence du symbole `merge`. Pour ce travail, il est clair que `Nessie` ne peut nous être d'aucune utilité. En effet, la seule règle de réécriture connue de `Nessie` est la β-réduction et, on vient de le dire, la fusion de DRSs ne peut être obtenue par l'intermédiaire de β-réductions. Nous devons donc trouver un autre moyen de simplifier les DRSs. Pour y parvenir, nous allons utiliser `Prolog`, qui permet de manipuler facilement des termes tels que celui que nous venons de présenter.

Nous allons procéder en deux passes successives. Lors d'une première passe, nous allons procéder à une simplification (ou normalisation) des DRSs visant à faciliter leur fusion, réalisée pendant la seconde passe. Pour comprendre ce que nous entendons par simplification, il convient de réfléchir à la fusion de DRSs proprement dite. Comme nous l'avons indiqué, fusionner deux DRSs revient essentiellement à fusionner leurs ensembles de référents de discours d'une part, et leurs ensembles de conditions d'autre part. Pour que cela soit faisable, nous avons besoin d'exprimer toutes les DRSs « atomiques » en termes d'ensembles de référents de discours et de conditions, et cela de sorte que les DRSs ainsi exprimées soient faciles à utiliser dans le contexte d'une fusion.

Or, les DRSs produites par la phase de construction sémantique ne sont pas exprimées en termes d'ensembles de référents de discours et de conditions. Ou du moins, cette information n'est pas donnée de façon explicite. La première passe consiste donc essentiellement à rendre explicite les ensembles de référents de discours et de conditions constituant chaque DRS. Pour y parvenir, nous choisissons de représenter une DRS par un terme `Prolog` ayant pour foncteur `drs` d'arité 2. Les arguments de ce foncteur seront deux listes, la première étant une liste de référents de discours, la seconde une liste de conditions. On l'aura compris, il s'agit ici d'adopter une notation qui s'approche des abréviations utilisées par Muskens. Le but de la première passe est donc de transformer des représentations sémantiques telles que celle qui a été vue précédemment en des représentations faisant intervenir `drs/2`. Voici les règles `Prolog` que nous utilisons pour parvenir à ce résultat :

```
simplifyDrs(drs_of_dr(X),drs([X],[])).
simplifyDrs(drs_of_condition(C),D) :- !,
  simplifyCondition(C,Cout),
  D = drs([],[Cout]).

simplifyDrs(merge(D1,D2),D) :- !,
  simplifyDrs(D1,D1out),
  simplifyDrs(D2,D2out),
```

```
  D = merge(D1out,D2out).

simplifyCondition(condition_of_urel(P,A),Cout) :- !,
  Cout =.. [P,A].

simplifyCondition(condition_of_brel(P,X,Y),Cout) :- !,
  Cout =.. [P,X,Y].

simplifyCondition(drsimp(D1,D2),C) :- !,
  simplifyDrs(D1,D1out),
  simplifyDrs(D2,D2out),
  C = drsimp(D1out,D2out).
simplifyCondition(drsor(D1,D2),C) :- !,
  simplifyDrs(D1,D1out),
  simplifyDrs(D2,D2out),
  C = drsor(D1out,D2out).
simplifyCondition(drsnot(D),Cout) :- !,
  simplifyDrs(D,Dout),
  Cout = drsnot(Dout).
```

Les deux premières clauses définissant `simplifyDrs` sont celles simplifiant les DRSs atomiques. Ce sont ces clauses, ainsi que celles où figurent les constantes `condition_of_urel` et `condition_of_brel` qui font effectivement quelque chose, les autres clauses ne servant qu'à traverser récursivement les DRSs et conditions. Si nous utilisons les prédicats que nous venons de définir sur la représentation sémantique précédemment construite, voici la DRS que nous obtenons :

```
merge(drs([u1], []), merge(drs([], [man(u1)]), merge(drs([u2], []),
merge(drs([], [woman(u2)]), drs([], [love(u1, u2)])))).
```

Nous remarquons que le terme ainsi obtenu est déjà plus simple que le précédent. Et surtout : ce terme montre — peut-être encore mieux que le précédent — tout le bénéfice que l'on pourra tirer de la fusion de DRSs et à quel point fusionner les DRSs peut conduire à une simplification de la représentation obtenue. Avant de montrer comment cette simplification est réalisée, il convient cependant de faire remarquer que, lors de notre première passe, nous nous sommes délibérément affranchis des contraintes de typage. Considérons en effet les trois sous-termes `man(u1)`, `woman(u2)` et `love(u1,u2)` apparaissant dans le terme simplifié. Dans le lexique qui a été utilisé pour la construction sémantique, man et woman ont été déclarés comme étant de la famille des noms et sont donc des prédicats de type $e \rightarrow t$. De la même façon, `love` étant un lemme de la famille des verbes transitifs, la constante qu'il introduit est de type $e \rightarrow e \rightarrow t$. Du point de vue du typage, les applications de ces prédicats à des registres (de type π) sont donc illicites. Nous faisons cependant le choix d'y recourir, pour obtenir des termes plus simples. Si nous avions voulu garder des termes typables tout au long du processus de simplification de DRSs, il aurait fallu continuer d'utiliser les constructeurs `condition_of_urel` et `condition_of_brel`, et cela n'aurait pas été suffisant. Il aurait aussi fallu faire en sorte que le symbole `drs` puisse être typé. Or, quel type associer à `drs` ? Dans un système de types permettant le polymorphisme, on pourrait apporter à cette question une réponse simple : `drs` prend en

arguments une liste de référents de discours, une liste de conditions et renvoie une DRS. Ceci n'est cependant pas exprimable ainsi dans TYn. La seule solution que l'on pourrait apporter à ce problème dans TYn consisterait à déclarer deux types de listes spécialisées, à savoir un type pour les listes de référents de discours et un autre type pour les listes de conditions, chacun de ces types devant fournir ses propres constructeurs « liste vide » et « ajout d'un élément à une liste ».

Tout cela est possible, certes, mais nous pensons que, compte tenu du but qui est le nôtre ici, nous n'avons pas un grand intérêt à garder à tout prix des termes typés tout au long du processus de simplification de DRSs. Nous passons donc à la deuxième étape de simplification, à savoir la fusion de DRSs proprement dite. Celle-ci est implantée à l'aide de trois prédicats `Prolog` : `mergeDrs/2` qui procède à toutes les fusions de DRSs possibles en une étape de réécriture, `hasMerge/1` qui teste si un terme contient ou non des occurrences du symbole `merge`, et enfin `rewriteDrs/2` qui utilise les deux prédicats précédents pour réécrire une DRS tant qu'elle contient au moins une occurrence du symbole `merge`. Voici les clauses `Prolog` définissant ces trois prédicats :

```
mergeDrs(merge(drs(Dr1,C1),drs(Dr2,C2)),drs(Dr,C)) :- !,
  appendLists(Dr1,Dr2,Dr),
  appendLists(C1,C2,C).
mergeDrs(Term1,Term2) :- var(Term1), !, Term1=Term2.
mergeDrs(Term1,Term2) :-
  nonvar(Term1), !,
  Term1 =.. [F|Args],
  maplist(mergeDrs,Args,NewArgs),
  Term2 =.. [F|NewArgs].

hasMerge(merge(_,_)).
hasMerge(Term) :-
  nonvar(Term), !,
  Term =.. [_|Args],
  hasMergeList(Args).

hasMergeList([]) :- fail.
hasMergeList([X|Xs]) :- hasMerge(X); hasMergeList(Xs).

rewriteMerge(Term1,Term3) :-
  hasMerge(Term1) ->
    ( mergeDrs(Term1,Term2), rewriteMerge(Term2,Term3) );
    Term1 = Term3.
```

La première clause définissant `mergeDrs` est celle effectuant la réécriture proprement dite. La seconde clause permet de gérer correctement les variables et la troisième permet le parcours récursif des termes. Il est nécessaire d'appeler plusieurs fois le prédicat `mergeDrs` parce que des occurrences imbriquées de `merge` ne peuvent être simplifiées par un seul appel. Pour parachever ce travail de simplification, nous ajoutons des prédicats permettant d'afficher des DRSs sous une forme condensée qui s'inspire de celle utilisée par Muskens. En d'autres termes, un terme `Prolog` de la forme `drs([u1,u2],[c1,c2])` est affiché `[u1,u2|c1,c2]`.

Ainsi, la représentation de « A man loves a woman. » que nous obtenons après les deux passes de simplification est la suivante :

```
[u1,u2|man(u1),woman(u2),love(u1, u2)].
```

Cette représentation est, bien sûr, bien plus facile à lire que celle obtenue avant la simplification. Elle correspond très exactement à celle que Muskens propose, mais a cependant été obtenue automatiquement. Forts de ce premier succès, nous nous empressons de calculer d'autres DRSs : Pour la phrase « Every man loves a woman. » nous obtenons :

```
[|drsimp([u1|man(u1)], [u2|woman(u2),love(u1, u2)])].
```

Ou, pour un exemple un petit peu plus compliqué encore, nous pouvons calculer la représentation de « If every man loves mia then vincent loves mia ». Nous obtenons :

```
[|
  drsimp([|
    drsimp([u1|man(u1)], [|love(u1, mia)])],
    [|love(vincent, mia)]
  )
].
```

Nous sommes donc parvenus à construire exactement les représentations proposées par Muskens. En plus de `Nessie`, nous avons dû recourir pour atteindre ce résultat à un prétraitement des arbres syntaxiques pour en annoter les feuilles contenant des déterminants, et à de la réécriture pour réaliser les fusions de DRSs.

6.3 Implantation du traitement compositionnel de la dynamicité

Dans cette section, nous allons montrer comment construire des représentations sémantiques « à la de Groote » pour les phrases engendrées par la grammaire présentée en section 6.1. Comme on pourra le constater, la construction de représentations sémantiques dans ce formalisme est beaucoup plus directe que celle présentée à la section précédente, puisqu'elle ne nécessite ni prétraitement des arbres d'analyse ni recours à des règles de réécriture autres que la β-réduction. La seule tâche qui nous incombe ici est donc la mise au point d'un lexique adapté, tâche à laquelle nous nous attelons sans plus tarder, avant de donner quelques exemples illustrant la construction automatique de représentations sémantiques à l'aide de la DCG, du lexique et de `Nessie`.

6.3.1 Lexique

Comme d'habitude, le lexique commence par la déclaration des types nécessaires :

```
type o = t;
type iota;
type gamma;
```

```
type delta = gamma -> o;
type drs = gamma -> delta -> o;
type n = iota -> drs;
type np = (iota -> drs) -> drs;
```

On pourrait se passer de la déclaration du type `o` qui n'est qu'un alias pour le type `t` toujours présent dans TYn, mais nous choisissons de la conserver, de sorte à rester aussi proches que possible des notations utilisées dans (de Groote, 2006). C'est pour cette même raison qu'est introduit le type ι pour représenter les entités, alors que dans toutes les applications précédentes nous avions utilisé `entity` ou `e`.

Les types `gamma` et `delta` correspondent respectivement aux contextes gauches et droits, tandis que le type `drs` est celui des λ-termes représentant les phrases et les discours. Nous l'appelons ainsi pour souligner que c'est le type « équivalent » dans l'approche de de Groote au type `s->s->t` de l'approche de Muskens, mais il est important de se souvenir que les objets construits ici ne sont pas de véritables DRSs, contrairement aux termes construits à la section précédente. Enfin, les types `n` et `np` sont des alias donnant les types sémantiques des catégories syntaxiques homonymes.

Le lexique se poursuit par la déclaration des constantes spécifiant les symboles et opérations disponibles :

```
const top : o;
const not : o -> o;
const emptyContext : gamma;
const cons : iota -> gamma -> gamma;
const emptyRightContext : delta :=
  lam x : gamma. (top);
const merge : drs -> drs -> drs :=
  lam D, S : drs. lam e : gamma. lam phi : delta.
  ( D(e) @ lam e0 : gamma. (S(e0,phi)) );
const dynneg : drs -> drs :=
  lam D : drs. lam e : gamma. lam phi : delta.
  (not(D(e,emptyRightContext)) && phi(e));
const dynconj : drs -> drs -> drs :=
  lam D1, D2 : drs. lam e : gamma. lam phi : delta.
  ( D1(e) @
    lam e0 : gamma. (D2(e0,phi))
  );
const implication : drs -> drs -> drs :=
  lam D1, D2 : drs.
  (dynneg(dynconj(D1,dynneg(D2))));
```

La plupart de ces constantes ne requièrent, nous semble-t-il, aucun commentaire particulier. Notons simplement que la constante `implication` est définie à l'aide des constantes `dynneg` et `dynconj`, conformément à ce qui est fait dans (de Groote, 2006). Cependant, nous demanderons à `Nessie` de remplacer toutes ces constantes par leur valeur avant de procéder à la β-réduction des représentations sémantiques, et ces constantes n'apparaîtront donc plus dans la représentation sémantique finale.

Le lexique se termine par la déclaration des familles et des lemmes usuels, ces déclarations ne réservant aucune surprise particulière. Notons simplement que, comme nous l'avons déjà fait remarquer, les λ-termes utilisés pour représenter les mots du lexique sont relativement complexes. À titre d'exemple, voici comment est défini le déterminant universel :

```
family det;

const uni_helper :
  (iota->drs) -> (iota->drs) -> gamma -> delta -> o :=
  lam n, psi : iota -> drs. lam e : gamma. lam phi : delta.
  forall x : iota. !
  (
    n(x,e) @
    lam e0 : gamma. ! ( psi(x,cons(x,e0),emptyRightContext))
  );

lemma uni {
  family = det;
  term =
  lam n, psi : iota -> drs. lam e : gamma. lam phi : delta.
  (uni_helper(n,psi,e,phi) && phi(e))
};
```

On le voit, le λ-terme est si complexe que nous avons besoin d'introduire une constante simplement pour garder des représentations de taille raisonnable et faciles à lire.

Donnons également la représentation des pronoms. Celle qui est incluse dans le lexique de `Nessie` est exactement celle de (de Groote, 2006) et complète le support des pronoms qui avait été entrepris en section 6.1. Nous ferons évoluer cette représentation à la section suivante.

```
family pro {
  type = gamma -> iota;
  pattern =
  lam psi : iota -> drs. lam e : gamma. lam phi : delta.
  ( psi(_(e),e,phi) )
};

lemma selhe, selshe, selit, selhim, selher : pro;
```

6.3.2 Calcul de représentations du sens

Grâce à ce lexique, il nous est possible de faire calculer par `Nessie` une représentation pour la phrase « Vincent eats a burger. ». Nous obtenons :

```
lam(E,lam(Phi,some(X,and(burger(X),
and(eat(vincent,X),app(Phi,cons(vincent,cons(X,E)))))))).
```

Comme le montre le sous-terme `app(Phi,cons(vincent,cons(X,E)))`, les individus `X` et `vincent` sont passés à la continuation `Phi` et seront donc présents dans le contexte fourni aux phrases suivantes. Ainsi, si nous demandons la représentation du petit discours « Vincent eats a burger. He likes it. », nous obtenons le terme suivant :

```
lam(E,lam(Phi,
some(X,and(burger(X),
  and(eat(vincent,X),
  and(like(
      selhe(cons(vincent,cons(X,E))),
      selit(cons(vincent,cons(X,E)))
  ),app(Phi,E)))
))))).
```

ce qui nous permet de constater que `vincent` et `X` sont tous deux présents dans les contextes passés en argument aux pronoms dans la seconde phrase. En revanche, si nous demandons la représentation de « If vincent eats a burger then he likes it. », alors nous obtenons le terme suivant :

```
lam(E,lam(Phi,
  and(
    not(some(X,and(
      burger(X),
      and(eat(vincent,X),and(not(and(
        like(
      selhe(cons(vincent,cons(X,E))),
      selit(cons(vincent,cons(X,E)))
        ),
        top)),top))
    ))),
    app(Phi,E)
  )
)).
```

Ici le contexte contenant `vincent` et `X` est certes passé aux pronoms apparaissant dans la seconde phrase, mais pas à la continuation `Phi` qui représente une éventuelle suite du discours. On vérifie ainsi que les entités introduites par l'antécédent d'une conditionnelle sont accessibles depuis la conséquence de celle-ci, mais pas depuis la suite du discours. De la même façon, nous pouvons calculer la représentation de la phrase « Every man who eats a burger likes it », qui est similaire à « Every farmer who owns a donkey beats it », utilisée comme exemple final dans (de Groote, 2006). La représentation que nous obtenons pour cette phrase est al suivante :

Remarquons pour clore cette section que les représentations que nous venons de construire ne sont pas utilisables en l'état à des fins d'inférence. En effet, ces représentations sont des λ-termes, et non des formules du premier ordre prêtes à être manipulées par des outils d'inférence. Pour passer des λ-termes aux formules, il nous faut nous débarrasser des deux abstractions figurant dans les représentations sémantiques. On y parvient en appliquant à chaque représentation que l'on souhaite transformer en formule du premier ordre un contexte vide et une continuation vide. Pour obtenir de tels termes, nous procédons en deux étapes. D'une part, nous ajoutons au lexique de `Nessie` les déclarations suivantes :

```
const drsToO : drs -> o :=
  lam P : drs. ( P(emptyContext,emptyRightContext) );

family firstOrderLogic;

lemma firstOrderLogic {
  family = firstOrderLogic;
  term = lam D : drs. ( drsToO(D) )
};
```

D'autre part, on modifie légèrement les arbres produits par la DCG de la section 6.1. Au lieu d'associer à une phrase ou à un discours un arbre `t`, on renvoie l'arbre suivant :

```
binary(
  leaf(firstOrderLogic, firstOrderLogic),
  t
)
```

Ainsi, la représentation calculée pour le discours « Vincent eats a burger. He likes it. » est maintenant la suivante :

```
some(X,and(burger(X),
  and(eat(vincent,X),
  and(like(
      selhe(cons(vincent,cons(X,emptyContext))),
      selit(cons(vincent,cons(X,emptyContext))
  )),top))
)).
```

Deux remarques finales s'imposent à propos de cette représentation. D'une part, on pourrait souhaiter la simplifier, en supprimant l'occurrence inutile de `top`. Il serait possible de parvenir à ce résultat en utilisant des règles de réécriture similaires à celles vues en section 6.2, où nous les avions utilisées pour fusionner des DRSs. Ce qu'il nous paraît important de retenir, c'est que, de la même façon que la fusion de DRSs ne pouvait être mise en œuvre uniquement grâce à la β-réduction, la simplification que nous venons d'évoquer, aussi simple qu'elle paraisse, ne peut elle non plus être effectuée au moyen de la β-réduction. Dans les deux cas, il est nécessaire de recourir à un système de réécriture plus puissant que celui reposant seulement sur la β-réduction. Nous reviendrons sur cette constatation dans la conclusion de ce chapitre, puis dans la conclusion de la thèse. D'autre part, on peut remarquer que la représentation qui vient d'être construite n'est pas, en l'état, utilisable par `Curt`. En effet, elle fait intervenir des termes du premier ordre utilisant des symboles de fonction, alors que les seuls termes du premier ordre utilisables par `Curt` sont les constantes et les variables. On remarque cependant que les seuls symboles de fonction apparaissant dans la représentation ci-dessus sont ceux introduits par les représentations des pronoms. Plus précisément, il s'agit des fonctions de résolution d'anaphores. Comme on va le voir dans la section qui suit, résoudre les anaphores permettra précisément d'obtenir des représentations ne faisant plus intervenir de symboles de fonction, et donc utilisables par `Curt`.

6.4 Résolution d'anaphores dans TYn

Dans son traitement compositionnel de la dynamicité, de Groote propose de représenter les pronoms à l'aide de fonctions prenant en argument un contexte qui contient l'ensemble des antécédents possibles pour ce pronom et renvoyant « le bon antécédent ». Cette section commence par discuter des présupposés de cette hypothèse et se poursuit par la présentation d'une mise en œuvre possible d'une telle approche. Nous en viendrons ainsi à proposer une architecture logicielle permettant de résoudre les anaphores dans TYn. Une fois cette architecture mise en place, nous verrons comment elle peut être utilisée pour résoudre des anaphores pronominales simples suivant deux techniques, l'une algorithmique, l'autre logique.

6.4.1 Ce que présuppose la proposition de de Groote

La représentation des pronoms à l'aide de fonctions supposées capables de choisir dans un contexte le bon antécédent fait à notre avis trois hypothèses. La première d'entre elles est qu'il est possible de calculer pendant la construction sémantique (en utilisant *seulement* la β-réduction) l'ensemble des antécédents possibles pour une occurrence d'un pronom, et de passer cet ensemble comme argument à la fonction représentant le pronom. Le bien fondé de cette première hypothèse est attesté par la proposition de de Groote elle-même, puisqu'il parvient à construire de tels ensembles (les contextes) de façon purement compositionnelle. Cette hypothèse nous semble donc légitime, et nous la reprenons à notre compte, supposant que, pour tous les types de relations anaphoriques que l'on peut souhaiter résoudre dans TYn, il est possible de construire pour tout syntagme appelé à être remplacé par un autre un ensemble de ses antécédents possibles, grâce à la règle de β-réduction.

La seconde hypothèse que nous aimerions mettre en évidence tient à la valeur renvoyée par le processus de résolution anaphorique. Pour de Groote, les fonctions représentant les pronoms sont vues comme des oracles, renvoyant une entité qui est précisément celle désignée par le pronom. Ce qui nous semble sous-entendu ici, c'est qu'il est possible de choisir parmi l'ensemble des entités passé en paramètre celle qui est « la bonne », sans avoir pour cela à examiner au préalable les diverses représentations sémantiques obtenues pour chaque valeur possible de l'antécédent. Pour bien comprendre ce qui vient d'être affirmé, il peut être utile de considérer le petit discours suivant : « Vincent eats a burger. He likes it. », discours sur lequel nous nous appuierons tout au long de cette section. La représentation que nous avions obtenu pour ce texte à la section précédente était :

```
some(X,and(burger(X),
  and(eat(vincent,X),
  and(like(
      selhe(cons(vincent,cons(X,emptyContext))),
      selit(cons(vincent,cons(X,emptyContext))
  )),top))
)).
```

Ici, la résolution anaphorique est facilitée par le fait que « Vincent » et X sont de genres différents. Par conséquent, les fonctions $\texttt{sel}_{\text{he}}$et $\texttt{sel}_{\text{it}}$peuvent trouver facilement l'antécédent

pour les pronoms qu'elles représentent, puisqu'à chaque fois un seul individu ayant le genre qui convient est présent dans le contexte. Il n'en n'irait pas ainsi, en revanche, si le discours considéré était écrit en français. En effet, dans ce cas, « Vincent » et X seraient tous deux de genre masculin. On pourrait alors avoir besoin, pour déterminer à quels antécédents se rapportent les pronoms, de produire les 4 représentations sémantiques possibles, si l'on tient compte du fait que chaque pronom a deux antécédents possibles et que les choix d'antécédents sont *a priori* indépendants.

Or l'approche de de Groote semble difficile à concilier avec cet exemple, puisque selon cette approche les opérateurs de résolution d'anaphores devraient pouvoir fournir leur résultat *avant* que la représentation sémantique ne soit calculée dans sa totalité, alors qu'ici nous avons *d'abord* généré plusieurs représentations sémantiques, obtenues en remplaçant chaque pronom par un antécédent possible, avant de choisir une représentation parmi celles-ci, ce choix étant lui-même une indication implicite des antécédents choisis pour chaque pronom.

On comprend dès lors que, si l'hypothèse de de Groote était particulièrement utile et simplificatrice pour un travail théorique tel que celui présenté au chapitre précédent (et qui n'avait pas pour objet spécifique la résolution des anaphores), elle ne nous semble pas particulièrement pertinente lorsqu'il s'agit de mettre en œuvre, concrètement, des techniques de résolution des anaphores. Dans la suite de cette section, nous allons donc supposer que les fonctions recherchant un antécédent parmi une liste d'antécédents possibles peuvent en réalité en proposer plusieurs, donnant lieu à autant de représentations sémantiques possibles dans lesquelles chaque appel de fonction est successivement remplacé par l'un des résultats possibles de cet appel.

Enfin, après avoir évoqué une hypothèse sur les arguments des fonctions de résolution d'anaphores et une sur leur résultat, nous aimerions en mentionner une sur la façon dont ces fonctions font leurs calculs et qui est à notre avis implicite dans la présentation qui en est donnée par de Groote. Cette hypothèse porte sur les règles de réduction autorisées pendant le calcul. Nous avons en effet vu que la proposition de de Groote permet de calculer des contextes modélisant les contraintes d'accessibilité de la DRT, et cela en utilisant uniquement la β-réduction comme règle de réécriture. Cependant, rien n'est précisé quant à la façon dont pourrait être implantée une fonction de choix d'antécédent, et nous pensons qu'écrire une telle fonction seulement à l'aide de β-réductions serait une tâche difficile, voire impossible. C'est pourquoi, il nous semble que le choix de l'antécédent d'un pronom ne peut être effectué seulement à l'aide de réductions β et que d'autres règles de réécriture doivent être autorisées. Nous allons donc utiliser cette hypothèse par la suite, ce qui augmentera considérablement nos possibilités quant à ce qu'une telle fonction peut calculer.

6.4.2 Une architecture logicielle pour la résolution d'anaphores

Pour résumer la discussion de la sous-section précédente, nous allons proposer une architecture logicielle permettant la résolution des anaphores dans TYn et basée sur les hypothèses suivantes, qui sont une adaptation des présupposés présents dans l'approche de de Groote. Premièrement, nous supposons que les syntagmes qui sont appelés à être remplacés par d'autres syntagmes peuvent être représentés par des fonctions prenant en argument une liste d'antécédents possibles et renvoyant « le bon antécédent ». Deuxièmement, nous allons implanter ces fonctions de résolution d'anaphores en leur faisant renvoyer non pas un seul antécédent, mais plusieurs, à

partir desquels nous construirons un ensemble de représentations sémantiques plausibles. Troisièmement, nous nous autorisons pour l'implantation des fonctions de résolution d'anaphores à utiliser des règles autres que la β-réduction du λ-calcul simplement typé.

Concrètement, c'est au sein de `Curt` que nous allons réaliser ce travail d'implantation, comme nous l'avons fait aux chapitres 3 et 4. Ce choix a en particulier l'avantage de nous permettre de profiter des capacités de `Curt` à éliminer des représentations sémantiques inconsistantes, pour ne retenir que celles qui sont cohérentes avec une théorie donnée. C'est en effet ce mécanisme que nous allons utiliser pour choisir parmi plusieurs représentations sémantiques celles qui sont cohérentes du point de vue des anaphores, ce qui nous conduira à réfléchir à l'axiomatisation des relations anaphoriques.

La résolution d'anaphores est implantée par un programme baptisé `AnaR`, et qui a été développé pendant cette thèse. L'entrée du programme est un terme de TYn dont certains sous-termes sont les applications de fonctions de résolution d'anaphores (dont les noms sont connus du programme) à un argument qui est une liste d'antécédents possibles. En sortie, `AnaR` fournit une liste de représentations sémantiques dans lesquels tous les appels à des fonctions de résolution d'anaphores ont été remplacés par les résultats possibles de ces appels, tels qu'ils ont été calculés par les fonctions de résolution d'anaphores présentes dans le programme.

Pour bien comprendre comment `AnaR` fonctionne, nous pouvons par exemple commencer par associer aux noms `selhe` et `selit` la fonction identité, ce qui signifie que pour ce solveur d'anaphores extrêmement naïf, tous les antécédents qui lui sont passés en argument sont des antécédents possibles. Dans ce cas, si nous appelons `AnaR` en lui passant en entrée le terme

```
selhe(cons(vincent,cons(X,emptyContext)))
```

alors nous obtenons en sortie :

```
[vincent,X]!
```

ce qui signifie que le terme donné en entrée peut être remplacé soit par `vincent`, soit par `X`.

De la même façon, lorsque le solveur est appelé avec en entrée le terme

```
selit(cons(vincent,cons(X,emptyContext)))
```

la sortie produite est exactement la même, à savoir

```
[vincent,X]
```

Donc, si le solveur est appelé sur une représentation sémantique contenant à la fois les deux termes vus précédemment, on doit obtenir en sortie 4 représentations sémantiques, puisque les résultats des fonctions de résolution d'anaphores sont supposés indépendants les uns des autres. Ainsi, lorsque `AnaR` est appelé avec pour entrée

```
like(
  selhe(cons(vincent,cons(X,emptyContext)))
  selit(cons(vincent,cons(X,emptyContext)))
)
```

la sortie produite est la suivante :

```
[
  like(vincent,vincent),
  like(vincent,X),
  like(X,vincent),
  like(X,X)
].
```

Un tel programme de résolution d'anaphores peut être facilement inséré dans la version modifiée de `Curt` qui utilise `Nessie` pour la construction sémantique. Il suffit en effet, au lieu de passer le résultat de `Nessie` à `Curt` directement, de le passer d'abord à `AnaR` pour passer ensuite à `Curt` les résultats de ce dernier. Une fois cette modification effectuée, nous pouvons avoir avec `Knowledgeable Curt` le dialogue suivant :

```
> vincent eats a burger. he likes it.
Message (consistency checking): mace found a result.
Message (consistency checking): mace found a result.
Message (consistency checking): mace found a result.
Message (consistency checking): mace found a result.
Message (informativeness checking): mace found a result.
Message (informativeness checking): mace found a result.
Message (informativeness checking): mace found a result.
Message (informativeness checking): mace found a result.
Curt: OK.
> readings
1 some(A, and(burger(A), and(eat(vincent, A), and(like(A, A), top))))
2 some(A, and(burger(A), and(eat(vincent, A), and(like(A, vincent),
top))))
3 some(A, and(burger(A), and(eat(vincent, A), and(like(vincent, A),
top))))
4 some(A, and(burger(A), and(eat(vincent, A), and(like(vincent,
vincent), top))))
```

Bien sûr, le fait d'obtenir 4 lectures possibles pour ce petit texte alors qu'on n'en voudrait qu'une ne semble pas, à première vue, très satisfaisant. Il constitue cependant une preuve du bon fonctionnement de l'architecture logicielle mise en place jusqu'ici, et au sein de laquelle nous allons pouvoir raffiner le processus de résolution d'anaphores.

6.4.3 Approche algorithmique

Le programme de résolution d'anaphores vu jusqu'ici ne tient pas compte du genre des entités manipulées. Pour qu'il puisse le faire, il s'agit d'abord de lui rendre cette information accessible, puis de le modifier pour qu'il puisse en tenir compte. Dans la mesure où la seule structure accessible pendant la résolution d'anaphores est celle des contextes, ce sont eux qui doivent, d'une manière ou d'une autre, rendre compte du genre des entités qu'ils contiennent. Une première solution pour y parvenir consiste à envisager un contexte non plus comme un seul ensemble mais comme un triplet d'ensembles, chaque ensemble regroupant les entités d'un genre donné. Cependant, dans la mesure où les triplets ne sont pas très faciles à manipuler dans le cadre du λ-calcul simplement typé, nous allons utiliser une solution légèrement différente dans son

implantation, bien que similaire dans l'esprit. Au lieu de n'utiliser qu'un seul constructeur de contextes comme de Groote le faisait dans sa proposition, nous allons en utiliser 3 : un ajoutant une entité de genre féminin à un contexte, un autre pour les entités de genre masculin, et enfin un troisième pur les entités de genre neutre. Dans le lexique sémantique de `Nessie`, nous remplaçons donc la déclaration

```
const cons : iota -> gamma -> gamma;
```

par les trois déclarations suivantes :

```
const consm : iota -> gamma -> gamma;
const consf : iota -> gamma -> gamma;
const consn : iota -> gamma -> gamma;
```

Avec de telles constantes, le contexte rencontré précédemment et représenté jusqu'ici par le terme

```
cons(vincent,cons(X,emptyContext))
```

peut être représenté par

```
consm(vincent,consn(X,emptyContext))
```

ce qui permet au programme de résolution d'anaphores de calculer pour un genre donné l'ensemble des entités ayant ce genre et présentes dans le contexte qui lui est passé en argument. Or, comme les fonctions de résolution d'anaphores connaissent le genre des individus recherchés, il leur est possible de ne renvoyer que les individus de ce genre présents dans le contexte.

Il reste à voir comment modifier les représentations sémantiques de sorte à construire des contextes comme ceux que nous venons de présenter. Les modifications à apporter concernent d'une part les noms propres et d'autre part les groupes nominaux constitués d'un déterminant suivi d'un substantif.

Noms propres On commence par ajouter une information de genre concernant les noms propres au lexique syntaxique de `Curt`. Pour ce faire, nous remplaçons les déclarations originales de la forme :

```
lexEntry(pn,[symbol:vincent,syntax:[vincent]]).
```

par des déclarations telles que :

```
lexEntry(pn,[symbol:vincent,syntax:[vincent],gen:mas]).
```

Puis nous modifions la grammaire de `Curt` de sorte que l'information de genre des noms propres soit incluse dans les arbres syntaxiques construits pour ces derniers. Ainsi, au lieu de construire l'arbre `leaf(vincent, pn)`, on construit l'arbre `leaf(vincent,pn,mas)`. On fait donc le choix d'utiliser les arguments d'occurrences pour stocker dans l'arbre d'analyse l'information de genre, qui est nécessaire pour ajouter l'entité au contexte. Enfin, dans le lexique sémantique de `Nessie`, on définit :

```
const mas := consm;
```

qui permet d'interpréter l'information de genre par un constructeur de contexte, et l'on modifie la déclaration de la famille des noms propres de la façon suivante :

```
family pn {
  type = iota;
  pattern = lam psi : iota->drs. lam e : gamma. lam phi : delta.
  ( psi(_,e) @
    lam e0 : gamma.
      ( phi($1(_,e0)) )
  )
};
```

Avec ces modifications, le terme calculé par `Nessie` pour l'arbre `leaf(vincent, pn, mas)` est le suivant :

```
lam(Psi,lam(E,lam(Phi,
  app(app(app(Psi,vincent),E),lam(E0,app(Phi,consm(vincent,E0))))))
)))
```

Ce terme est certes un peu difficile à lire, mais la seule information réellement importante est que le genre (masculin) est bien pris en compte lors de l'ajout au contexte, comme en témoigne le sous-terme `consm(vincent,E0)`.

Déterminants et noms communs La représentation des groupes nominaux constitués d'un déterminant et d'un nom commun doit elle aussi être modifiée. Les modifications à effectuer ici sont essentiellement du même ordre que celles qui viennent d'être décrites pour les noms propres : modifications du lexique syntaxique, de la DCG et du lexique sémantique. Cependant, on doit prendre en compte ici un fait nouveau. Dans la proposition de de Groote, ce sont les déterminants qui ajoutent l'entité qu'ils introduisent au contexte. Or, en anglais, les déterminants ne portent aucune information de genre, et un déterminant n'est donc pas en mesure de connaître le genre de l'entité qu'il introduit. Nous avons donc été amenés à modifier tant la représentation des déterminants que celles des noms communs, pour que ce soient ces derniers qui ajoutent les entités introduites par les déterminants au contexte. Remarquons que si cette approche est possible ici, elle serait plus difficile à envisager lorsque le contexte auquel il convient d'ajouter une entité dépend du déterminant qui l'introduit, comme c'est le cas dans (Pogodalla, 2008), où les entités introduites par les déterminants définis sont traitées différemment de celles introduites par les déterminants indéfinis, de sorte à prendre en compte les différences d'accessibilité de ces deux classes de déterminants. Remarquons aussi qu'une langue telle que le français, où les déterminants sont marqués en genre (au moins au singulier) ne poserait pas ce type de problème.

Une fois ces modifications effectuées et le programme de résolution d'anaphores rendu sensible au genre, nous pouvons avoir avec `Knowledgeable Curt` le dialogue suivant :

```
> vincent eats a burger. he likes it.
Message (consistency checking): mace found a result.
Message (informativeness checking): mace found a result.
Curt: OK.
> readings
1 some(A, and(burger(A), and(eat(vincent, A),
and(like(vincent, A), top))))
```

ce qui est bien le résultat escompté.

6.4.4 Axiomatisation des relations anaphoriques

Nous supposons ici que nous repartons des lexiques (syntaxique et sémantique) et de la grammaire tels qu'ils se présentaient au début de cette section, c'est-à-dire avant les modifications apportées à la sous-section précédente. Comme nous l'avons annoncé précédemment, nous allons ici tirer parti de la capacité de `Curt` à éliminer les lectures d'une phrase qui sont inconsistantes avec une théorie donnée. Pour parvenir à ce résultat, il nous faut axiomatiser les propriétés des relations anaphoriques de sorte que toutes les lectures d'une phrase dans lesquelles le genre des pronoms ne correspond pas avec celui de l'antécédent qui leur est associé contredisent un axiome. Il est donc clair que ce qu'il faut axiomatiser, c'est l'idée selon laquelle le genre d'un pronom et de son antécédent doivent coïncider. Pour pouvoir énoncer des propriétés concernant les genres, la première chose à faire est de réifier cette notion qui n'était pas présente dans la proposition de de Groote. Nous commençons donc par ajouter au lexique sémantique de `Nessie` un type pour le genre, trois constantes `mas`, `fem` et `neu` de ce type, ainsi qu'un prédicat binaire `gen` permettant de lier une entité et son genre. Nous ajoutons donc au lexique `Nessie` les déclarations suivantes :

```
type gender;
const mas : gender;
const fem : gender;
const neu : gender;
const gen : iota -> gender -> o;
```

Une fois en possession de ces outils, il est nécessaire de modifier la représentation des pronoms de sorte à établir, grâce au prédicat `gen`, un lien entre le genre de l'entité à laquelle se réfère un pronom et le genre du pronom, auquel on peut accéder par l'intermédiaire de la constante `$1`, puisque d'après nos conventions celui-ci est passé comme premier argument des feuilles représentant les pronoms. Une modification possible de la représentation des pronoms est la suivante :

```
family pro {
  type = gamma -> iota;
  pattern =
  lam psi : iota -> drs. lam e : gamma. lam phi : delta.
  ( psi(_(e),e,phi) && gen(_(e),$1) )
};
```

Cette modification ajoute bien la propriété désirée (il s'agit du terme `gen(_(e),$1)`), mais elle ne nous satisfait pas vraiment. En effet, comme on peut le constater, elle fait intervenir par deux fois le caractère _, qui est appelé à être remplacé, lors de l'instanciation, par l'opérateur de choix d'antécédent implanté par notre solveur d'anaphores. Cette représentation des pronoms donnera donc lieu à deux appels au solveur d'anaphores pour chaque occurrence d'un pronom, ce qui n'est pas souhaitable. Nous allons donc utiliser une autre représentation des pronoms qui ne fait intervenir qu'un seul appel à la fonction de résolution d'anaphores. On utilise pour cela une variable liée existentiellement et dont on énumère toutes les propriétés, ce qui donne la représentation suivante :

```
family pro {
  type = gamma -> iota;
  pattern =
  lam psi : iota -> drs. lam e : gamma.
  lam phi : delta. exists x : iota.
  ( x=_(e) && gen(x,$1) && psi(x,e,phi) )
};
```

Comme on peut le constater, le caractère _ n'apparaît plus qu'une fois dans cette représentation, et donc la fonction de résolution des anaphores ne sera invoquée qu'une fois, ce qui résout le problème de la représentation précédente.

Il nous reste à voir comment spécifier les informations sur le genre des noms propres et des noms communs, et quelle théorie utiliser. Comme nous allons le voir, ces deux tâches sont liées, puisque c'est au moyen d'axiomes que nous allons spécifier tant le genre des noms propres que celui des noms communs.

La théorie que nous utilisons comporte deux parties. La première est celle générée automatiquement à partir du lexique de `Nessie`, à l'aide de l'outil `TheoGen` introduit au chapitre 4. Rappelons qu'il s'agit d'une théorie traduisant sous forme d'axiomes toutes les contraintes imposées implicitement aux représentations sémantiques par le typage. La deuxième partie de la théorie peut à son tour être subdivisée en deux parties : une comportant des axiomes généraux sur les genres, et une spécifiant le genre de chaque nom propre et de chaque nom commun. Notre formalisation des genres est basée sur les cinq axiomes suivants :

gender_unique	$\forall ABC.((\text{gen}(A,B) \wedge \text{gen}(A,C)) \rightarrow B = C)$
not_mas_and_fem	$\neg \text{mas} = \text{fem}$
not_mas_and_neu	$\neg \text{mas} = \text{neu}$
not_fem_and_neu	$\neg \text{fem} = \text{neu}$
genders_partition_entities	$\forall A.(\text{iota}(A) \rightarrow (\text{gen}(A,\text{mas}) \vee \text{gen}(A,\text{neu}) \vee \text{gen}(A,\text{fem}))$

À ces axiomes indépendants de la langue (ils sont utilisables avec toutes les langues comportant 3 genres, donc en particulier l'anglais et l'allemand), nous en ajoutons un par nom propre ou substantif qui nous intéressent. Pour le traitement de notre exemple, nous ajoutons :

vincent_gender	$\text{gen}(\text{vincent}], \text{mas})$
burger_gender	$\forall A.(\text{burger}(A) \rightarrow \text{gen}(A,\text{neu}))$

Cette théorie étant construite, nous pouvons la tester en lançant `Curt`. On obtient le dialogue suivant :

```
> vincent eats a burger. he likes it.
Message (consistency checking): otter found a result.
Message (consistency checking): mace found a result.
Message (consistency checking): otter found a result.
Message (consistency checking): otter found a result.
Message (informativeness checking): mace found a result.
Curt: OK.
> readings
1 some(A, and(burger(A), and(eat(vincent, A), some(B, and(and(eq(B,
vincent), gen(B, mas)), some(C, and(and(eq(C, A), gen(C, neu)),
and(like(B, C), top)))))))
```

Pour faciliter la lecture de ce petit dialogue, rappelons que `Mace` est un constructeur de modèle, tandis que `Otter` est un prouveur de théorèmes. Compte tenu de la façon dont ces outils d'inférence sont utilisés, il convient de comprendre que lorsque c'est `Otter` qui trouve un résultat à un test de consistance, cela signifie que ce test a échoué, autrement dit la représentation et la théorie considérées sont contradictoires. Comme on peut le constater, 4 tests de consistance sont réalisés, ce qui n'a rien de surprenant puisque nous savons que la version naïve du programme de résolution d'anaphores renvoie 4 lectures possibles pour notre exemple de test. Parmi ces 4 lectures, 3 sont jugées inconsistantes avec la théorie utilisée. Une seule passe donc le test d'informativité, qu'elle réussit, puisque c'est `Mace` qui trouve un résultat pour ce test et non `Otter`, ce qui signifie qu'un modèle satisfaisant simultanément cette lecture et la théorie considérée a pu être construit. Enfin, la commande `readings` nous permet de nous assurer que la lecture trouvée est bien celle que nous souhaitions. On remarquera que la représentation ainsi obtenue est légèrement plus complexe que celle obtenue avec la première technique de résolution des anaphores, utilisant un solveur sensible au genre. Il est clair que, pour tenir compte des genres des noms, la première méthode est plus efficace et plus élégante que celle qui vient d'être présentée. Cette dernière approche utilisant des outils d'inférence n'en demeure pas moins une méthode possible pour résoudre d'autres types d'anaphores, ou pour tenir compte d'autres contraintes d'une nature moins locale que les genres.

Remarquons aussi qu'au cours de l'élaboration de la théorie qui vient d'être présentée, une erreur s'était glissée dans l'un des axiomes : `burger` était déclaré de genre masculin, au lieu d'être neutre. Avec une telle théorie, les 4 lectures de notre texte d'exemple ont été rejetées comme contradictoires par `Curt`, ce qui est parfaitement légitime puisque dans une telle théorie, aucune entité n'est de genre neutre et donc le pronom « it » ne saurait avoir d'antécédent.

6.5 Conclusion

Nous avons montré dans ce chapitre comment les formalismes présentés au chapitre précédent peuvent être utilisés pour calculer effectivement des représentations sémantiques de discours. Le fait que nous soyons parvenus à construire des représentations sémantiques dans ces deux formalismes pourrait donner à penser qu'ils sont, en fin de compte, équivalents du point de vue de leur utilisabilité en pratique. Cependant, il n'en est rien et nous en voulons pour preuve les efforts qu'il a fallu déployer pour parvenir à construire des représentations « à la Muskens », alors que `Nessie` est suffisant pour construire à lui seul les représentations proposées par de Groote.

Au-delà de cette comparaison entre deux approches de la sémantique des discours basées sur la théorie des types, ce chapitre nous conduit à nous interroger sur la pertinence de TYn en tant que langage de spécification des représentations du sens. Nous avons en effet, par trois fois dans ce chapitre, été confrontés à des situations où nous aurions aimé faire certaines simplifications des représentations sémantiques obtenues, alors que ces simplifications ne pouvaient être spécifiées dans TYn muni uniquement de la β-réduction. La première situation à laquelle nous faisons allusion est celle de la section 6.2, où nous avons dû recourir à des règles de réécriture supplémentaires implantées en `Prolog` pour effectuer des fusions de DRSs. La seconde situation, vue en section 6.3, est celle où l'on aurait aimé réécrire des formules du type `and(F,top)` en `F`, et où nous avons renoncé à mettre en place cette simplification, jugée trop coûteuse à implanter

en regard des bénéfices que l'on aurait tirés d'une telle simplification. La troisième situation est bien sûr celle vue en section 6.4, où nous avons implanté en `Caml` les fonctions de résolution d'anaphores, faute de pouvoir les spécifier directement dans TYn.

Bien que les solutions qui ont été proposées dans ce chapitre aux problèmes que nous venons de rappeler puissent, dans un premier temps, être jugées satisfaisantes, nous ne pouvons nous empêcher de les trouver quelque peu inélégantes, et pour tout dire un peu frustrantes. En effet, la solution au problème des fusions de DRSs et celle utilisée pour implanter les opérateurs de choix d'antécédents ont en commun le fait d'éparpiller et parfois de dupliquer les informations relatives aux objets manipulés. Dans le cas de la fusion de DRSs par exemple, une partie de la définition de l'opérateur de fusion est stockée dans le lexique sémantique de `Nessie`, tandis que la règle de réécriture permettant de fusionner deux DRSs fait partie du programme `Prolog` qui la met en œuvre, et où elle n'apparaît d'ailleurs pas « en clair », mais noyée au milieu du code. Il en va de même pour la résolution d'anaphores : on a besoin de formaliser les genres à la fois dans le lexique de `Nessie` *et* dans le solveur d'anaphores.

On comprend dès lors que, si la logique utilisée pour spécifier les représentations sémantiques était suffisamment expressive pour décrire, en plus des représentations elles-mêmes, les processus permettant de les simplifier, cela pourrait permettre d'éviter les dispersions et duplications qui viennent d'être soulignées. Ce constat appelle à notre avis une remarque et une question.

D'une part, on peut remarquer que les facilités offertes par un formalisme logique pour exprimer le sens ne constituent pas nécessairement un critère suffisant pour décider que ce formalisme est adapté au traitement automatique de représentations sémantiques. À ce premier critère, nous sommes tentés d'en ajouter un autre qui pourrait être la capacité du formalisme à manipuler les représentations sémantiques, et notamment à les simplifier ou à faire des calculs de haut niveau les utilisant. D'autre part, si l'on admet que ce chapitre tend à prouver que TYn ne satisfait pas le second critère qui vient d'être énoncé, il faut alors se demander quelle autre logique plus expressive pourrait le remplacer, étant donné que le gain en expressivité ne saurait être trop grand, sous peine de perdre des propriétés importantes de la logique choisie, telles que la terminaison et la confluence. Nous poursuivrons cette discussion au chapitre 8, où nous verrons d'autres éléments que ceux présentés dans ce chapitre et susceptibles de nous aider à trouver un candidat pour un éventuel remplacement de TYn.

Chapitre 7

Nessie vu comme une grammaire catégorielle abstraite du second ordre

Nous avons introduit au chapitre 2 un outil appelé `Nessie` permettant de construire des représentations sémantiques à partir de lexiques et d'arbres syntaxiques. Les chapitres suivants nous ont permis d'intégrer `Nessie` à `Curt` (chapitre 3), d'utiliser la version ainsi modifiée de `Curt` pour entreprendre une première expérimentation en construction sémantique dans TYn (chapitre 4), puis de construire un cadre permettant de calculer compositionnellement la sémantique des discours et où les expérimentations entreprises précédemment ont été poursuivies (chapitres 5 et 6). Nous pouvons donc dire que, pour l'instant, c'est surtout l'aspect sémantique qui a été exploré, le côté syntaxique n'étant qu'effleuré. En particulier, nous nous sommes contentés jusqu'ici de construire des représentations sémantiques pour de très petits fragments de langues, pour lesquels nous avons écrit des DCGs capables à la fois de les reconnaître et de construire les arbres d'analyse nécessaires à `Nessie`. Il va cependant de soi que cette approche est limitée : écrire une DCG pour analyser des textes et produire des arbres permettant d'en construire le sens est certes envisageable pour de petits langages de tests, mais cette approche s'avère très rapidement limitée lorsqu'il s'agit de construire le sens d'un langage de taille importante. Pour de tels langages, des formalismes syntaxiques bien plus expressifs que les DCGs ont été proposés, comme par exemple les grammaires d'arbres adjoints, les grammaires fonctionnelles lexicalisées, *etc.*

Or, pour la plupart de ces formalismes, des analyseurs syntaxiques ont été développés. Il paraît donc naturel de se demander avec lesquels de ces analyseurs on pourrait interfacer `Nessie`, c'est-à-dire, étant donné les formalismes utilisés, quels analyseurs seraient en mesure de produire des arbres utilisables par `Nessie`.

L'une des démarches possibles pour répondre à cette question pourrait consister à choisir un formalisme grammatical et un analyseur pour ce formalisme, puis à essayer de le modifier de sorte qu'il produise les arbres syntaxiques que `Nessie` peut utiliser pour la construction sémantique. Cette démarche nous paraît cependant doublement insatisfaisante. D'une part, si elle échoue, comment savoir si cet échec est dû à l'analyseur syntaxique, ou à des raisons plus profondes, inhérentes au formalisme grammatical sur lequel est fondé cet analyseur ? D'autre part, si elle réussit, que nous apprendra-t-elle de réellement intéressant sur `Nessie` ? Certes, nous saurons qu'il est en mesure de construire des représentations à partir d'arbres produits en utilisant un certain formalisme syntaxique, mais c'est là le seul apprentissage que nous pouvons espérer faire, et cela en courant le risque d'un échec.

Plutôt que de procéder ainsi, nous allons adopter dans ce chapitre une démarche plus théo-

rique et réfléchir à l'expressivité de `Nessie` dans un cadre formel. Ce cadre nous est donné par les ACGs (Grammaires Catégorielles Abstraites) introduites par de Groote dans (de Groote, 2001). Schématiquement, et bien que les ACGs soient un formalisme relativement récent, nous sommes tentés de présenter ce formalisme basé sur le λ-calcul simplement typé en disant que les ACGs occupent dans la modélisation de l'interface syntaxe-sémantique la même place que les machines de Turing en calculabilité. Cela signifie qu'une ACG permet de spécifier un langage, et que les langages reconnus par les ACGs ont des propriétés particulières que nous expliciterons. C'est pourquoi il est utile de montrer que d'autres formalismes grammaticaux peuvent être simulés à l'aide d'ACGs ; cela permet en effet à la fois de connaître l'expressivité de ces autres formalismes, et de prouver des propriétés des langages reconnus.

Nous allons donc dans ce chapitre mettre en relation les langages reconnus par `Nessie` et ceux reconnus par une certaine classe d'ACGs, à savoir les ACGs d'ordre deux, dont nous donnerons au préalable une définition. Cela nous permettra d'une part de déduire certaines propriétés théoriques des langages reconnus par `Nessie`, et d'autre part de savoir avec quels formalismes syntaxiques il est envisageable de combiner `Nessie`. En effet, si nous parvenons à établir un lien entre `Nessie` et les ACGs, alors nous serons à même d'en déduire le lien entre le langage reconnu par `Nessie` et tous les formalismes grammaticaux qui peuvent être simulés à l'aide d'ACGs, à savoir notamment les formalismes faiblement contextuels comme les grammaires TAG ou LCFRS dont la simulation à l'aide d'ACGs est décrite dans (de Groote et Pogodalla, 2004).

Nous commençons (section 7.1) par introduire les ACGs. Nous en donnerons d'abord la définition, puis nous en énoncerons quelques propriétés. Une fois les ACGs introduites, nous serons en mesure d'expliquer de façon plus précise le lien que nous cherchons à établir entre ces grammaires et `Nessie`. Nous poursuivrons en montrant comment spécifier le langage reconnu par `Nessie` en termes d'ACGs, et cette démonstration se fera en deux étapes. Dans un premier temps, nous montrerons que certains mécanismes utilisés par `Nessie` peuvent être exprimés à l'aide d'autres mécanismes, construisant ainsi une sorte de langage minimal. Dans un second temps, nous montrerons comment construire une ACG permettant de passer des arbres spécifiés dans ce langage minimal à leur représentation sémantique.

7.1 Définition et propriétés des grammaires catégorielles abstraites

Nous commençons par donner la définition générale des ACGs, puis de celles d'ordre deux. Nous énonçons ensuite les propriétés de ces dernières dont nous aurons besoin par la suite, et concluons cette section en posant de manière précise la problématique que nous souhaitons aborder dans ce chapitre.

7.1.1 Définitions

Nous l'avons dit, les ACGs sont basées sur le λ-calcul simplement typé. Nous en donnons ici la définition, qui a été donnée pour la première fois dans (de Groote, 2001). On commence par définir l'ensemble des types considérés dans ce formalisme.

Définition 7. *Soit A un ensemble fini de types dits atomiques. Alors on définit l'ensemble $\mathscr{T}(A)$ des types sur A de façon inductive :*
- *Si $\tau \in A$, alors $\tau \in \mathscr{T}(A)$;*
- *Si $\alpha, \beta \in \mathscr{T}(A)$, alors $\alpha \to \beta \in \mathscr{T}(A)$.*

En outre, si $\tau_1, \ldots, \tau_n \in \mathscr{T}(A)$ et $\tau_0 \in A$, alors on dit que le type $\tau_1 \to \ldots \to \tau_n \to \tau_0$ est d'arité n.

Définition 8. *On appelle signature linéaire d'ordre supérieur ou plus simplement vocabulaire un triplet $\Sigma = \langle A, C, \tau \rangle$ tel que :*
- *A est un ensemble fini de types atomiques ;*
- *C est un ensemble fini de constantes ;*
- *τ est une fonction qui à chaque constante de C associe un type de $\mathscr{T}(A)$.*

Définition 9. *Soit par ailleurs un ensemble dénombrable X de λ-variables. Alors l'ensemble $\Lambda(\Sigma)$ des λ-termes linéaires sur une signature d'ordre supérieur $\Sigma = \langle A, C, \tau \rangle$ est défini inductivement de la façon suivante :*
- *si $c \in C$, alors $c \in \Lambda(\Sigma)$;*
- *si $x \in X$, alors $x \in \Lambda(\Sigma)$;*
- *si $x \in X$, $t \in \Lambda(\Sigma)$, et x a exactement une occurrence libre dans t, alors $\lambda x.t \in \Lambda(\Sigma)$;*
- *Si $t, u \in \Lambda(\Sigma)$, et si les ensembles de variables libres de t et u sont disjoints, alors $(tu) \in \Lambda(\Sigma)$.*

Étant donné un vocabulaire $\Sigma = \langle A, C, \tau \rangle$, il est possible d'associer un type linéaire implicatif de $\mathscr{T}(A)$ à tout terme de $\Lambda(\Sigma)$. Cette affectation de type est régie par un système d'inférence dont les jugements sont des séquents de la forme $\Gamma \vdash_\Sigma t : \alpha$, avec :

1. Γ un ensemble fini de déclarations de types de variables de la forme $x : \beta$, avec $x \in X$ et $\beta \in \mathscr{T}(A)$, chaque variable apparaissant au plus une fois ;
2. $t \in \Lambda(\Sigma)$;
3. $\alpha \in \mathscr{T}(A)$.

Les axiomes et règles d'inférence utilisés sont les suivants :

$$\vdash_\Sigma c : \tau(c) \quad \text{(cons)}$$

$$x : \alpha \vdash_\Sigma x : \alpha \quad \text{(var)}$$

$$\frac{\Gamma, x : \alpha \vdash_\Sigma t : \beta}{\Gamma \vdash_\Sigma (\lambda x.t) : (\alpha \to \beta)} \quad \text{(abs)}$$

$$\frac{\Gamma \vdash_\Sigma t : (\alpha \to \beta) \quad \Delta \vdash_\Sigma u : \alpha}{\Gamma, \Delta \vdash_\Sigma (tu) : \beta} \quad \text{(app)}$$

Nous définissons à présent l'ordre d'un type, que nous utiliserons pour définir un peu plus tard l'ordre d'une grammaire catégorielle abstraite.

Définition 10. *Pour tout type τ, son ordre, noté $o(\tau)$ est défini inductivement comme suit :*
- *Si τ est atomique, alors $o(\tau) = 1$;*
- *Si $\tau = \tau_1 \to \tau_2$, alors $o(\tau) = max(1 + o(\tau_1), o(\tau_2))$.*

Par extension, étant donné un vocabulaire $\Sigma = \langle A, C, \tau \rangle$, et une constante $c \in C$, on dira que c est d'ordre k pour dire que son type $\tau(c)$ est d'ordre k.

Remarque 1. *Si $\tau_0, \tau_1, \ldots, \tau_n$ sont des types atomiques, alors*

$$o(\tau_1 \to \tau_2 \to \ldots \to \tau_n \to \tau_0) = 2$$

Ce résultat se prouve facilement par récurrence :
- *Si $n = 1$, alors on s'intéresse au type $\tau_1 \to \tau_0$, sachant que τ_1 et τ_0 sont atomiques et donc d'ordre 1 par définition. On a alors :*

$$o(\tau_1 \to \tau_0) = max(1 + o(\tau_1), o(\tau_2)) = 2$$

- *Pour $n > 1$, d'après l'hypothèse de récurrence on a*

$$o(\tau_2 \to \tau_3 \to \ldots \to \tau_n \to \tau_0) = 2$$

d'où l'on déduit le résultat annoncé, à l'aide du fait que τ_1 est atomique et donc d'ordre 1.

Ce résultat montre que le type d'une fonction peut être d'ordre 2 quelle que soit l'arité de cette fonction. La seule chose qui importe, c'est que les arguments de cette fonction soient eux-mêmes de type atomique. Autrement dit, pour qu'une fonction soit d'ordre 2, il faut que ses arguments ne soient pas eux-mêmes des fonctions, ce qui signifie que cette fonction n'est pas une fonction d'ordre supérieur ou fonctionnelle. On le voit, la définition d'ordre qui vient d'être donnée est à rapprocher de ce que l'on entend par ordre lorsqu'on parle de « logique du premier ordre » pour insister sur le fait que dans ces logiques on ne peut quantifier que sur les éléments du domaine, et pas sur des objets tels que les fonctions ou relations.

Nous pouvons maintenant définir le concept de lexique, qui est un morphisme entre vocabulaires. Plus formellement :

Définition 11. *Soient deux vocabulaires $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ et $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$. Alors, un lexique $\mathscr{L}$ de Σ_1 vers Σ_2 (ce que nous noterons par la suite $\mathscr{L} : \Sigma_1 \to \Sigma_2$) est un couple $\mathscr{L} = \langle F, G \rangle$ tel que :*
- *$F : A_1 \to \mathscr{T}(A_2)$ est une fonction qui à chaque type atomique de A_1 associe un type (non nécessairement atomique) construit à partir de A_2 ;*
- *$G : C_1 \to \Lambda(\Sigma_2)$ est une fonction qui à chaque constante de C_1 associe un terme linéaire sur le vocabulaire Σ_2 ;*
- *Les fonctions F et G sont compatibles avec la relation de typage, c'est-à-dire que pour toute constante $c \in C_1$ le jugement de typage suivant est dérivable :*

$$\vdash_{\Sigma_2} G(c) : \hat{F}(\tau_1(c))$$

où $\hat{F}$ est l'unique prolongement homomorphe de F.

Nous pouvons maintenant définir les grammaires catégorielles abstraites, ainsi que les grammaires catégorielles abstraites d'ordre 2 :

Définition 12. *Une ACG est un quadruplet $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ tel que :*
- $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ *et* $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ *sont deux vocabulaires ;* Σ_1 *est appelé le vocabulaire* abstrait *de* $\mathscr{G}$*, tandis que* Σ_2 *est appelé le vocabulaire* objet *de* $\mathscr{G}$ *;*
- $\mathscr{L} : \Sigma_1 \to \Sigma_2$ *est un lexique du vocabulaire abstrait* Σ_1 *vers le vocabulaire objet* Σ_2 *;*
- $s \in \mathscr{T}(A_1)$ *est un type du vocabulaire abstrait que l'on appelle le* type distingué *de* $\mathscr{G}$.

En outre, on dit que $\mathscr{G}$ est d'ordre k si et seulement si k est le plus petit entier tel que toutes les constantes du vocabulaire abstrait de $\mathscr{G}$ sont d'ordre au plus k.

On peut alors définir deux langages engendrés par une ACG : un langage abstrait, et un langage objet. Intuitivement, le langage abstrait consiste en l'ensemble des termes linéaires clos sur le vocabulaire abstrait qui ont pour type le type distingué de la grammaire. Le langage objet est alors l'image par le lexique du langage abstrait. Formellement, on a :

Définition 13. *Soit $\mathscr{G}$ une ACG telle que définie précédemment. Alors :*

$$\mathcal{A}(\mathscr{G}) = \{t \in \Lambda(\Sigma_1) \mid \vdash_{\Sigma_1} t : s \textit{ est dérivable}\}$$

et

$$\mathcal{O}(\mathscr{G}) = \{u \in \Lambda(\Sigma_2) \mid \exists t \in \mathcal{A}(\mathscr{G}).\ u = \mathscr{L}(t)\}$$

À l'aide de ces définitions, nous pouvons expliquer de quelle manière les grammaires catégorielles abstraites peuvent être utilisées pour modéliser à la fois l'analyse syntaxique et la construction sémantique. L'analyse syntaxique consiste à associer à une chaîne d'entrée sa structure syntaxique. Elle utilise donc deux langages : l'un, noté L_1, pour exprimer les structures syntaxiques, et l'autre, noté L_2, qui est l'ensemble des chaînes d'entrée possibles. Cette étape d'analyse syntaxique peut être décrite par une grammaire catégorielle abstraite $\mathscr{G}_{12}$ dont le langage abstrait est exactement L_1 et dont le langage objet est L_2. Un texte d'entrée T peut alors être vu comme un terme t du langage objet L_2. Réaliser l'analyse syntaxique de T revient alors à se demander s'il existe un terme u dont l'image par le lexique de l'ACG est t.

C'est donc de cette première utilisation des ACGs que provient l'appellation de « problème du parsing » utilisée pour faire référence au problème qui vient d'être mentionné et consistant à chercher, à partir d'un terme t du langage objet, un terme u du langage abstrait dont l'image par le lexique est t. Dans la sous-section suivante, nous indiquerons quelques résultats portant sur la décidabilité de ce problème.

La construction sémantique, quant à elle, consiste à construire une représentation sémantique à partir d'une structure syntaxique. Elle fait donc elle aussi appel à deux langages : celui utilisé précédemment pour représenter les structures syntaxiques et noté L_1, et un langage L_3 pour spécifier les représentations sémantiques. La construction sémantique peut donc être modélisée à l'aide d'une ACG $\mathscr{G}_{13}$, dont le langage abstrait est L_1 et dont le langage objet est L_3, ce qui signifie que les grammaires $\mathscr{G}_{12}$ et $\mathscr{G}_{13}$ ont le même langage abstrait, celui représentant les structures syntaxiques [1].

[1] Un exemple illustrant la modélisation de l'analyse syntaxique et de la construction sémantique à l'aide de grammaires catégorielles abstraites est donné dans (de Groote, 2001). Les notations utilisées ici nous ont été directement inspirées par cet exemple.

Avec cette modélisation de l'analyse syntaxique et de la construction sémantique, on peut aussi se demander s'il est possible, à partir d'une représentation sémantique, de produire un texte d'entrée correspondant à cette représentation. En termes d'ACGs, cela revient à trouver (à l'aide de la grammaire $\mathscr{G}_{13}$), pour une représentation sémantique donnée, une structure syntaxique ayant pour image cette représentation sémantique, puis à calculer, à l'aide de la grammaire $\mathscr{G}_{12}$ une chaîne d'entrée, image par le lexique de cette grammaire de la structure syntaxique abstraite. Le passage de la structure syntaxique à son image étant trivial, le problème réside surtout dans la recherche d'une structure syntaxique à partir de sa représentation sémantique, problème qui est souvent appelé « problème de la réalisation syntaxique » [2] et qui, dans le cadre des ACGs, est exactement le problème du parsing introduit précédemment, dans lequel le langage objet, au lieu d'être celui des chaînes de caractères, est celui des représentations sémantiques.

Terminons en donnant deux autres appellations possibles pour ce problème du parsing. La première, naturelle, est celle de « réversibilité » d'une ACG. La seconde découle du fait que, lorsque l'on cherche si un terme t a un antécédent par une ACG, on ne fait en réalité rien d'autre que déterminer si t appartient ou non au langage objet de cette ACG. C'est pour cette raison que le problème du parsing est aussi appelé parfois problème de l'appartenance (d'un terme au langage objet d'une ACG) ou « membership problem » en anglais.

7.1.2 Quelques propriétés remarquables des ACGs

Dans la définition de TYn qui a été donnée en section 1.2.2, aucune contrainte de linéarité n'avait été imposée sur les termes, ce qui signifie que la définition de TYn autorise chaque variable liée à avoir plusieurs occurrences dans le terme apparaissant après le lieur. La possibilité d'utiliser des termes non linéaires a d'ailleurs été exploitée par la suite, notamment lorsque les déterminants indéfinis ont été représentés par des termes tels que $\lambda PQ.\exists x.(Px) \rightarrow (Qx)$, où la variable liée x a deux occurrences.

On peut alors se demander pourquoi la définition des ACGs exige que les termes considérés soient linéaires. La réponse à cette question est double. D'une part, la condition de linéarité est vérifiée par la plupart des formalismes syntaxiques (voir (de Groote et Pogodalla, 2004)). D'autre part, la condition de linéarité est importante pour l'étude du problème du parsing qui a été défini précédemment. En effet, pendant longtemps, le seul résultat de décidabilité qui avait pu être obtenu pour ce problème l'avait été pour des ACGs linéaires (*cf.* (Salvati, 2005)), la décidabilité du parsing restant une question ouverte pour les ACGs non linéaires en général. Il a cependant été prouvé dans (Salvati, 2007) que, dans le cas particulier des grammaires du second ordre, la condition de linéarité n'est plus nécessaire pour garantir la décidabilité du parsing. Dans la mesure où, comme nous l'avons indiqué précédemment, les représentations sémantiques manipulées par `Nessie` ne sont pas nécessairement linéaires, c'est ce travail qui donne tout sons sens à la preuve d'équivalence entre `Nessie` et une ACG du second ordre qui va être faite dans ce chapitre.

[2] Un abus de langage à la fois courant et relativement commode consiste à utiliser le terme de « problème de la génération« pour faire référence au problème de la réalisation syntaxique. Nous utiliserons nous aussi cette appellation, mais il convient de garder à l'esprit que le terme génération peut aussi désigner le processus de production d'une représentation sémantique sous forme logique, ce processus prenant en compte des questions d'ordonnancement et ayant lieu avant la réalisaton syntaxique.

7.1.3 Problématique abordée

Compte tenu des définitions qui ont été données et des propriétés des ACGs d'ordre 2 qui viennent d'être énoncées, il devient possible de poser de manière plus précise la problématique que nous souhaitons aborder dans ce chapitre. Lorsque nous avons présenté Nessie, nous l'avons décrit, schématiquement parlant, comme une fonction $\mathscr{N}$ prenant en arguments un lexique et un arbre syntaxique et renvoyant comme résultat un terme de TYn. La fonction $\mathscr{N}$ pourrait donc avoir comme type lexicon $\rightarrow$ tree $\rightarrow$ term. Cependant, comme on a coutume de le faire en λ-calcul, nous pouvons aussi nous intéresser à l'application partielle de $\mathscr{N}$ à un lexique Nessie particulier L_N, application que l'on peut représenter par le terme $(\mathscr{N}\, L_N)$ de type tree $\rightarrow$ term. On peut alors se demander s'il est possible de construire à partir de L_N une ACG d'ordre 2 $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ simulant la fonction $(\mathscr{N}\, L_N)$. Dans la suite de ce chapitre, nous allons montrer qu'une telle grammaire existe, et qu'en outre elle vérifie la propriété suivante : pour tout terme t appartenant à la variante de TYn spécifiée par L_N il existe un arbre a tel que $(\mathscr{N}\, L_N a) = t$ si et seulement si il existe un terme $u \in \mathcal{A}(\mathscr{G})$ tel que $\mathscr{L}(u) = t$.

Comme nous l'avions annoncé dans l'introduction, nous allons aborder la construction de $\mathscr{G}$ et la preuve de la propriété que nous venons d'énoncer en deux étapes. Dans un premier temps, nous allons montrer qu'il est possible de se passer d'un certain nombre d'outils fournis par Nessie pour spécifier des représentations sémantiques et des arbres, sans toutefois diminuer l'expressivité obtenue. Dans un second temps, nous allons procéder à la construction de l'ACG simulant $(\mathscr{N}\, L_N)$ pour la partie restante de Nessie.

7.2 Épuration de Nessie

Dans cette section, nous allons donner une spécification plus précise de l'ensemble des représentations sémantiques et des arbres à considérer pour la preuve de la propriété énoncée à la fin de la section précédente. Nous commençons par expliquer pourquoi nous n'allons pas nous intéresser aux arbres dont les feuilles font appel à des arguments d'occurrences. Nous montrerons ensuite qu'il est possible de se dispenser à la fois des alias sur les types et des macros (constantes définies à l'aide de l'opérateur := dans les lexiques). Enfin, nous verrons que, bien que Nessie puisse utiliser plusieurs sortes d'arbres d'analyse (unaires, binaires et n-aires), il pourrait n'utiliser que des arbres binaires.

7.2.1 Non prise en compte des constantes d'occurrences

Nous avions présenté au chapitre 2 deux versions de l'algorithme de construction sémantique de Nessie. Le passage de la première à la seconde version de l'algorithme avait alors été motivé par la volonté de traiter des théories sémantiques qui représentent des occurrences différentes d'un même mot par des λ-termes non α-équivalents, comme par exemple la DRT compositionnelle de Muskens qui a été présentée dans les deux chapitres précédents. Cependant, nous avons aussi pu constater qu'il est possible de traiter la dynamicité de façon complètement compositionnelle, c'est-à-dire sans qu'il soit nécessaire pour y parvenir d'associer à des occurrences différentes d'un même mot des représentations non α-équivalentes. C'est pourquoi, nous pensons que le fait de ne pas couvrir dans ce chapitre la partie de Nessie permettant de traiter les

théories telles que celle de Muskens ne limite pas véritablement la portée et l'intérêt des résultats qui vont être montrés. En effet, comme le prouve le traitement de la dynamicité à bas de continuations proposé par de Groote et qui a été implémenté grâce à `Nessie`, la partie de `Nessie` que nous allons traiter est suffisante pour calculer de façon compositionnelle la sémantique de discours constitués de plusieurs phrases.

Nous allons donc nous en tenir à la première version de `Nessie` et supposer qu'aucune feuille ne comporte de constante d'occurrence. Ceci revient à supposer que le lexique d'entrée L_N de `Nessie` n'utilise pas les constructions du type `$1` dans les représentations des lemmes et familles qu'il définit.

7.2.2 Élimination des alias et des macros

Nous supposerons par la suite que le lexique utilisé pour construire l'ACG ne contient ni alias sur les types, ni définitions de constantes à l'aide de l'opérateur `:=`. En faisant cette hypothèse, nous ne diminuons en rien l'expressivité des lexiques, puisque tant les alias sur les types que les macros ne sont en fait que des facilités syntaxiques, certes très utiles en pratique, mais qui n'apportent rien en termes d'expressivité.

7.2.3 Élimination des arbres unaires et n-aires

Nous allons montrer ici que les arbres unaires et les arbres n-aires n'ajoutent rien à l'expressivité du langage d'arbres de `Nessie`. Ainsi, lorsque nous prouverons la propriété sur la simulation de `Nessie` par une ACG, il nous suffira dans la preuve de considérer le cas des feuilles et celui des arbres binaires, les autres types d'arbres n'augmentant pas l'expressivité de `Nessie`.

Cette façon de procéder est similaire à celle utilisée en logique du premier ordre et consistant à montrer que trois connecteurs et quantificateurs suffisent pour écrire toutes les formules du premier ordre.

Arbres unaires Lorsque nous avons donné les règles permettant d'associer une représentation sémantique à un arbre, nous avions indiqué que si un arbre t a pour représentation sémantique M, alors l'arbre `unary(label,t)` a aussi M pour représentation sémantique. Ceci revient à dire qu'aucun calcul sémantique n'est fait dans les arbres unaires, et que par conséquent l'arbre `unary(label,t)` pourrait être remplacé par l'arbre `t` sans que cela porte à conséquence sur la représentation sémantique finale.

Arbres n-ires Soit $n \geq 1$, $t_1, \ldots, t_n$ des arbres syntaxiques dont les représentations sémantiques sont notées $M_1, \ldots, M_n$. Soit c une macro du lexique, c'est-à-dire une constante ayant une *définition* et pas seulement une déclaration. On suppose que c est définie dans le lexique par l'entrée suivante :

```
const c := M;
```

On considère alors l'arbre `nary(label,c,t1,...,tn)`. Nous avons vu que la représentation sémantique d'un tel arbre est donnée par le terme $(MM_1 \dots M_n)$. Nous cherchons ici à construire ce terme à partir d'un arbre comportant seulement des feuilles et des nœuds binaires.

Nous remarquons tout d'abord que nous pouvons ajouter au lexique les déclarations suivantes :

```
family c_family;
lemma c_lemma = {
  family = c_family;
  term = c
};
```

Grâce à cette famille fermée et à ce lemme, nous obtenons la possibilité de faire référence à la constante `c` dans le langage des arbres, grâce à la feuille `leaf(c_lemma,c_family)`. En effet, par construction, la représentation sémantique de cette feuille est exactement la constante `c`. Cela nous permet de construire un arbre dont la représentation sémantique est exactement la même que celle de l'arbre n-aire vu précédemment. Il s'agit de l'arbre

```
binary(...,
  binary(
    leaf(c_lemma,c_family),
    t1),...,
  tn
)
```

Remarquons que si l'on peut utiliser comme règles de construction associées aux arbres n-aires des λ-termes arbitraires, et pas seulement des constantes du lexique, alors il n'est plus possible de procéder comme nous venons de le faire, car cela demanderait d'ajouter au lexique une infinité de constantes.

7.3 Construction de l'ACG et preuve de la propriété de simulation

Jusqu'à présent, nous avons montré que certains des mécanismes offerts par `Nessie` pour écrire des représentations sémantiques et des arbres n'étaient pas nécessaires, du point de vue de l'expressivité. Nous avons ainsi dégagé un langage minimal suffisant pour spécifier à la fois tous les arbres et toutes les représentations sémantiques susceptibles d'être manipulés par `Nessie`. Nous allons maintenant utiliser ce langage minimal pour prouver la propriété énoncée à la fin de la section 1. Pour cela, nous commençons par introduire quelques notations, puis nous construisons l'ACG dont nous avons besoin. Nous poursuivons en prouvant l'équivalence entre `Nessie` et l'ACG préalablement construite et terminons en donnant un exemple montrant comment construire une ACG pour un lexique donné.

7.3.1 Notations

Comme nous l'avons annoncé, la construction de l'ACG

$$\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$$

est faite à partir du lexique `Nessie` L_N.

Pour cette construction, on envisage L_N comme un triplet $L_N = \langle T_N, C_N, F_N \rangle$ tel que :

- T_N est l'ensemble des types atomiques ;
- C_N est l'ensemble des constantes « sémantiques », c'est-à-dire l'ensemble des constantes introduites par les déclarations de lemmes ouverts et celles déclarées à l'aide des directives `const`. Les types de toutes ces constantes sont connus et on note $\tau(c)$ le type de la constante $c \in C_N$;
- F_N est l'ensemble des familles (constantes « syntaxiques »), chaque famille étant elle-même vue comme un ensemble de lemmes.

Étant donnée une famille $f \in F_N$ et un lemme $l \in f$, on note $\langle l_f \rangle$ la représentation sémantique associée à ce lemme par L_N et $\tau(\langle l_f \rangle)$ le type de cette représentation tel qu'il peut être calculé à partir des informations de typage contenues dans L_N. Remarquons que nous prenons la précaution d'indicer chaque lemme par sa famille pour rappeler le fait (déjà mentionné au chapitre 2) qu'un lemme peut appartenir à plusieurs familles et qu'on a donc besoin de savoir à quelle famille appartient le lemme pour en connaître la représentation sémantique.

Soit τ un type. On définit alors Decompose(τ) par induction de la façon suivante :

- Si τ est atomique, alors Decompose$(\tau) = \{\tau\}$;
- Si $\tau = \tau_1 \rightarrow \tau_2$, alors Decompose$(\tau) = \{\tau\} \cup \{\tau_1\} \cup$ Decompose(τ_2).

Soit T un ensemble fini de types. On définit alors :

- Base$(T) = \bigcup_{\tau \in T}$ Decompose(τ) ;
- Atome(T) est un ensemble en bijection avec Base(T). On note $:=$ la bijection entre ces deux ensembles, ainsi que son unique prolongement compatible avec $\rightarrow$, ce dernier n'étant pas lui-même une bijection.

7.3.2 Construction de $\mathscr{G}$

Nous allons construire tour à tour les quatre composants de $\mathscr{G}$, à savoir son vocabulaire abstrait Σ_1, son vocabulaire objet, Σ_2, son lexique $\mathscr{L}$ et son type distingué s. Comme on pourra le constater, parmi tous ces objets c'est le vocabulaire abstrait dont la spécification est la plus complexe. Cette complexité tient au fait que l'on souhaite que les constantes soient d'ordre 2, ce qui oblige à introduire beaucoup de types et beaucoup de constantes pour chaque lemme de chaque famille de L_N.

Vocabulaire abstrait L'ensemble des types atomiques est construit à partir de l'ensemble des types des représentations sémantiques des familles et lemmes, soit K ce dernier ensemble qui est défini par :

$$K = \bigcup_{f \in F_n} \cup_{l \in f} \tau(\langle l_f \rangle).$$

L'ensemble des types atomiques vaut alors $A_1 =$ Atome(K) et l'on pose de plus $B =$ Base(K).

Soit $\tau \in B$. On définit alors l'ensemble Pa(τ) des profils atomiques de τ par induction de la façon suivante :

- Si τ est atomique, alors $\text{Pa}(\tau) = \{\tau'\}$, où τ' est l'unique élément de A ayant pour image τ (autrement dit l'antécédent de τ par la bijection $:=$ sur les types) ;
- Si $\tau = \tau_1 \to \tau_2$, α antécédent de τ et β antécédent de τ_1, alors

$$\text{Pa}(\tau) = \{\alpha\} \cup \{\beta \to \gamma | \gamma \in \text{Pa}(\tau_2)\}$$

Remarque 2. *Supposons que $\tau = \tau_1 \to \ldots \to \tau_k \to \tau_0$, avec $k >= 1$ et τ_0 atomique. Montrons par récurrence que le cardinal de Pa(τ) vaut $k+1$. Comme τ_0 est atomique, on a que Pa(τ_0) $= \{\tau_0'\}$, où τ_0' est l'antécédent de τ_0.*

- *Cas de base : on suppose $\tau = \tau_1 \to \tau_0$. Alors par définition du profil atomique on a :*

$$Pa(\tau) = \{\alpha\} \cup \{\beta \to \gamma | \gamma \in Pa(\tau_0)\}$$

  dont le cardinal vaut bien 2 puisque le cardinal de l'ensemble des profils atomiques de τ_0 est 1.
- *Hypothèse de récurrence : on suppose que la propriété est vraie au rang k et on veut la montrer au rang $k+1$. Comme précédemment, on utilise la définition du profil atomique et l'on obtient $k+2$ comme cardinal en remplaçant le cardinal du profil atomique de $\tau_2 \to \ldots \tau_{k+1} \to \tau_0$ par $k+1$, par hypothèse de récurrence.*

Remarque 3. *Pour tout $\alpha \in Pa(\tau)$, son image par $:=$ est τ. On le prouve en raisonnant par induction sur la structure de τ :*

- *Si τ est atomique, son ensemble de profils atomiques se réduit à son antécédent par la bijection, donc le résultat est évident ;*
- *Si $\tau = \tau_1 \to \tau_2$, alors Pa(τ) contient l'antécédent de τ par la bijection, dont l'image est τ par construction. Les autres profils atomiques sont de la forme $\beta \to \gamma$ avec β tel que $\beta := \tau_1$ et γ un profil atomique de τ_2. Par l'hypothèse d'induction on a que $\gamma := \tau_2$, et donc par définition de $:=$ on a la propriété voulue.*

On peut enfin donner l'ensemble des constantes du vocabulaire abstrait ainsi que leurs types. Pour toute famille $f \in F_N$, pour tout lemme $l \in f$ dont la représentation a pour type τ d'arité k, on introduit les $k+1$ constantes (l_f, α) de type $\alpha \in \text{Pa}(\tau)$.

Vocabulaire objet Le vocabulaire objet consiste en l'ensemble des constantes logiques et types définis par L_N, à savoir que son ensemble de types de base est T_N et que ses constantes sont celles de C_N, chacune d'elles ayant le type qui lui a été affecté par L_N.

Lexique Le morphisme sur les types est la fonction $:=$ introduite précédemment. Pour les constantes, leur image est définie par :

$$\mathscr{L}(l_f, \alpha) = \langle l_f \rangle$$

Type distingué Dans `Nessie`, aucune contrainte n'est imposée sur le type des valeurs sémantiques à construire. Il serait cependant facile de modifier le système pour que, par exemple, seuls des arbres construisant des valeurs sémantiques d'un certain type soient reconnus. Nous allons supposer que `Nessie` fonctionne ainsi et que l'on cherche à construire des arbres ayant des représentations sémantiques de type τ. Alors, le type distingué de $\mathscr{G}$ doit être l'antécédent de τ par la bijection $:=$ sur les types.

7.3.3 Preuve d'équivalence

Rappelons que, comme nous l'avons annoncé en section 1, nous voulons prouver en utilisant la grammaire qui vient d'être construite que pour tout terme $t \in \Lambda(\Sigma_2)$ l'équivalence suivante est vérifiée : $\exists u$. clos et de type atomique, tel que $\mathscr{L}(u) = t$ ssi $\exists a$ un arbre tel que $(\mathscr{N} L_N a) = t$.

Sens direct

On suppose qu'il existe u un terme clos de type atomique tel que $\mathscr{L}(u) = t$ et on veut montrer qu'alors il existe un arbre a tel que $(\mathscr{N} L_N a) = t$.

On raisonne par induction sur la structure de u :

- Si u est une constante, alors par construction du vocabulaire abstrait de $\mathscr{G}$ il existe une famille f, un lemme l de cette famille et un type α tels que $u = (l_f, \alpha)$. L'arbre correspondant est la feuille étiquetée par le lemme l et la famille f ;
- Si $u = (M_1 M_2)$, alors par hypothèse de récurrence il existe deux arbres A_1 et A_2 tels que $(\mathscr{N} L_N A_1) = M_1$ et $(\mathscr{N} L_N A_2) = M_2$. L'arbre recherché est alors l'arbre binaire ayant A_1 comme sous-arbre gauche et A_2 comme sous-arbre droit ;
- Comme u est supposé de type atomique, il ne peut commencer par une λ-abstraction ;
- Enfin, comme nous nous restreignons à des termes clos, u ne saurait être une variable.

Sens réciproque

On suppose qu'il existe un arbre a tel que $(\mathscr{N} L_N a) = t$ et on veut montrer qu'alors il existe u clos, de type atomique et tel que $\mathscr{L}(u) = t$.

On raisonne par induction sur la profondeur de l'arbre et la propriété que l'on prouve est la suivante : Soit τ le type de t et k son arité. En supposant l'existence de l'arbre a tel que $(\mathscr{N} L_N a) = t$, on va montrer qu'il existe non pas un seul mais $k+1$ termes u_i de types distincts appartenant à l'ensemble $\mathrm{Pa}(\tau)$ et tels que $\mathscr{L}(u_i) = t$. Remarquons au préalable que comme $\tau \in B$, son profil atomique est bien défini.

Si $n = 0$, les arbres considérés sont les feuilles et alors les termes cherchés existent par construction du vocabulaire abstrait de $\mathscr{G}$: ce sont les constantes.

Si $n > 0$, a est un arbre binaire. Soit b son sous-arbre gauche et c son sous-arbre droit. On peut alors poser $M_1 = (\mathscr{N} L_N b)$ et $M_2 = (\mathscr{N} L_N c)$. On a alors $t = (M_1 M_2)$. Comme M_1 peut prendre M_2 en argument, son type est de la forme $\tau_1 \to \tau$, tandis que M_2 est de type τ_1. Or par définition, si l'on appelle α l'antécédent de $\tau_1 \to \tau$ et β l'antécédent de τ_1, l'ensemble des profils atomiques de $\tau_1 \to \tau$ s'écrit :

$$\mathrm{Pa}(\tau_1 \to \tau) = \{\alpha\} \cup \{\beta \to \gamma | \gamma \in \mathrm{Pa}(\tau)\}$$

Cet ensemble contient un type atomique (α) et $k+1$ types non atomiques. D'après l'hypothèse d'induction, il existe donc $k+1$ termes u_i^b ($0 \leq i \leq k$) dont l'image est M_1 et dont les types sont fonctionnels, ce qui signifie que ces termes peuvent être appliqués à d'autres termes. Par ailleurs, la profondeur de l'arbre c étant inférieure à n, nous pouvons lui appliquer l'hypothèse de récurrence. Quelle que soit la forme du type τ_1 de M_2 (représentation sémantique de l'arbre c), on sait que l'ensemble des profils atomiques de τ_1 contient au moins 1 type atomique α_1 et

d'après l'hypothèse d'induction il existe donc un terme u^c de type α_1 dont l'image est M_2. On peut donc construire $k+1$ termes donc l'image est t, à savoir les termes de la forme $(u_i^b u^c), 0 \leq i \leq k$.

7.3.4 Exemple

Pour terminer, nous allons donner un exemple concret montrant comment construire une ACG à partir d'un lexique `Nessie`. Pour cet exemple, nous allons réutiliser le lexique donné au chapitre 2. Pour mémoire, ce lexique se présentait de la façon suivante :

```
type e;
type pred = e->t;
family det;
lemma indef {
  family = det;
  term = lam P, Q : pred. exists x:e. ( P(x) => Q(x) )
};

family iv {
  type = pred;
  pattern = lam x:e. (  _(x) )
};

lemma walk,dance : iv;

family noun {
  type = pred;
  pattern = lam x:e. (  _(x) )
};

lemma man, woman : noun;
```

Nous supposons que toutes les occurrences de `pred` sont remplacées par `e->t` et calculons l'ACG correspondant à ce lexique.

Pour commencer, voici un tableau associant à chaque famille le type de ses représentations sémantiques et qui nous sera utile par la suite :

Famille	Type
noun	$e \to t$
iv	$e \to t$
det	$(e \to t) \to (e \to t) \to t$

À partir de ce tableau, nous pouvons calculer l'ensemble des types dont nous aurons besoin pour calculer l'ensemble des types abstraits et que nous avions appelé K lors de la construction de $\mathscr{G}$. On obtient :

$$K = \{e \to t, (e \to t) \to (e \to t) \to t\}$$

On en déduit l'ensemble des types de base, calculé en décomposant les types complexes comme indiqué précédemment :

$$\text{Base}(K) = \text{Decompose}(e \to t) \cup \text{Decompose}((e \to t) \to (e \to t) \to t)$$

Après calcul, on obtient

$$\text{Base}(K) = \{(e \to t) \to (e \to t) \to t, e \to t, e, t, (e \to t) \to t\}$$

Nous construisons alors l'ensemble des types atomiques du vocabulaire abstrait, en bijection avec l'ensemble qui vient d'être présenté :

$$\text{Atome}(K) = \{\overline{(e \to t) \to (e \to t) \to t}, \overline{(e \to t) \to t}, \overline{e \to t}, \overline{e}, \overline{t}\}$$

Nous allons maintenant calculer l'ensemble des constantes du vocabulaire abstrait. Comme le lexique que nous considérons ne contient pas de lemme appartenant à plusieurs familles, nous ne prendrons pas la peine d'écrire les noms de familles en indice, comme cela a été fait dans la preuve précédente. On calcule le profil atomique des types $e \to t$ et $(e \to t) \to (e \to t) \to t$:

$$\text{Pa}(e \to t) = \{\overline{e \to t}, \overline{e} \to \overline{t}\}$$

qui est bien un ensemble contenant 2 éléments, à savoir un élément de plus que l'arité du type considéré.

De même comme l'arité du type $(e \to t) \to (e \to t) \to t$ est 2, ce type doit avoir 3 profils atomiques. Or, on a :

$$\begin{aligned}\text{Pa}((e \to t) \to (e \to t) \to t) \quad = \{&\overline{(e \to t) \to (e \to t) \to t},\\ &\overline{e \to t} \to \overline{(e \to t) \to t},\\ &\overline{e \to t} \to \overline{e \to t} \to \overline{t}\}\end{aligned}$$

Ces profils atomiques nous permettent de définir les constantes constituant le vocabulaire abstrait de l'ACG. Voici ces constantes, accompagnées de leur type :

Constante	Type
walk_1	$\overline{e \to t}$
walk_2	$\overline{e} \to \overline{t}$
dance_1	$\overline{e \to t}$
dance_2	$\overline{e} \to \overline{t}$
man_1	$\overline{e \to t}$
man_2	$\overline{e} \to \overline{t}$
woman_1	$\overline{e \to t}$
woman_2	$\overline{e} \to \overline{t}$
indef_1	$\overline{(e \to t) \to (e \to t) \to t}$
indef_2	$\overline{e \to t} \to \overline{(e \to t) \to t}$
indef_3	$\overline{e \to t} \to \overline{e \to t} \to \overline{t}$

Remarquons qu'ici, pour distinguer les constantes provenant d'un même lemme l appartenant à la famille f les unes des autres, nous avons préféré utiliser un nombre écrit en indice, plutôt que d'utiliser la notation (l_f, α) avec α le type de la constante, comme nous l'avions fait dans la preuve donnée précédemment.

Nous pouvons maintenant donner le vocabulaire objet de l'ACG, que nous obtenons directement à partir du lexique. L'ensemble des types est $\{e, t\}$, tandis que les constantes et leurs types sont données dans le tableau suivant :

Constante	Type
walk	$e \to t$
dance	$e \to t$
man	$e \to t$
woman	$e \to t$
indef	$(e \to t) \to (e \to t) \to t$

Quant au lexique de l'ACG, l'image d'un type de la forme $\overline{\tau}$ est le type τ, tandis que l'image d'une constante de la forme l_i où l est un lemme est la représentation sémantique associée à ce lemme par le lexique de `Nessie`.

Enfin, si nous faisons l'hypothèse que nous cherchons à reconnaître des arbres construisant des phrases (donc des valeurs de type t), le type distingué de la grammaire est $\overline{t}$.

Nous pouvons maintenant considérer à nouveau l'arbre d'analyse que nous avions donné au chapitre 2 pour la phrase « A man walks », à savoir :

```
binary(s,
  binary(np,leaf(indef,det),leaf(man,noun)),
  unary(vp,leaf(walk,iv))
)
```

Puisque cet arbre existe, d'après le théorème précédent il doit exister un terme de type $\overline{t}$ dont l'image par le lexique est la représentation sémantique produite par `Nessie` pour cet arbre. Considérons le terme $M = ((\text{indef}_3\text{man}_1)\text{walk}_1)$. Il est facile de vérifier que ce terme est bien de type $\overline{t}$. De plus, on peut calculer son image par le lexique :

$$\mathscr{L}(M) = ((\mathscr{L}(\text{indef}_3)\mathscr{L}(\text{man}_1))\mathscr{L}(\text{walk}_1))$$

ce qui donne en utilisant la définition de $\mathscr{L}$:

$$\mathscr{L}(M) = (((\lambda PQ.\exists x.(Px) \wedge (Qx))(\lambda x.(\text{man } x)))(\lambda x.(\text{walk } x)))$$

qui après β-réduction donne bien la représentation sémantique associée par `Nessie` à l'arbre précédent, à savoir

$$\exists x.(\text{man } x) \wedge (\text{walk } x)$$

7.4 Conclusion

Dans ce chapitre, nous avons montré qu'un lexique `Nessie` peut être assimilé à une ACG du second ordre : les déclarations de types et constantes du lexique décrivent le vocabulaire objet

de l'ACG, tandis que les déclarations de familles et de lemmes définissent simultanément les constantes du vocabulaire abstrait et leur image par le lexique de la grammaire.

Nous avons également montré que `Nessie` dispose de plusieurs mécanismes permettant de simplifier l'écriture d'arbres et de représentations sémantiques, mais que ceux-ci – bien que très utiles en pratique – n'augmentent pas l'expressivité que l'on obtient avec un langage minimal de représentations sémantiques et de spécification d'arbres. Comme toutes les constructions de ce dernier peuvent être traduites vers les formules sémantiques qu'elles représentent à l'aide d'une ACG du second ordre, nous pouvons en conclure, d'après les propriétés d'inversibilité de ces grammaires, qu'il est aussi possible de construire, à partir d'une représentation sémantique, un arbre engendrant cette représentation. Aussi, si l'analyseur syntaxique est capable, à partir de cet arbre, de produire la chaîne d'entrée dont cet arbre est issu, cela signifie qu'il est possible à partir d'une représentation sémantique de produire un texte correspondant à cette représentation. Ce chapitre nous permet donc d'affirmer que la fonction $(\mathscr{N}L_N)$ qui traduit un arbre en sa représentation sémantique est « réversible ». Nous en déduisons que les lexiques de `Nessie` pourraient être utilisés en génération aussi bien qu'en analyse.

Chapitre 8

Conclusion

Nous commençons par résumer les éléments présentés dans cette thèse en soulignant ses contributions, puis suggérons quelques pistes de recherche.

8.1 Contributions

Au chapitre 1, nous avons commencé par présenter les travaux fondateurs pour la sémantique formelle, à savoir ceux de Tarski, Church et surtout Montague, dont les idées, bien qu'elles remontent aux années 70, continuent d'inspirer les travaux portant sur la sémantique formelle des langues. En particulier, sa méthode des fragments est toujours très utilisée, tout comme les concepts de lexique et d'arbre qu'il a introduits. Nous avons ensuite présenté les quatre directions de recherche les plus pertinentes pur cette thèse, parmi toutes celles qui ont été explorées suite aux travaux de Montague, à savoir les entités abstraites, TYn, la DRT et les ACGs. Nous avons ensuite fait un petit tour d'horizon des contributions en sémantique computationnelle, qui est le domaine sur lequel porte cette thèse. Lors de ce survol, nous nous sommes attardé sur `Curt`, le système introduit par Blackburn et Bos et que nous avons utilisé pour mener à bien diverses expérimentations en construction sémantique dans la suite de cette thèse.

Notre objectif étant d'étudier l'intérêt de TYn pour la sémantique computationnelle, nous avons commencé par développer, au chapitre 2, un outil nommé `Nessie` et qui permet de construire des représentations sémantiques dans TYn à partir de lexiques et d'arbres, conformément à l'approche montagovienne. Deux caractéristiques distinguent `Nessie` de ses prédécesseurs. D'une part, `Nessie` est écrit en `OCaml`, un langage de programmation *fonctionnel* typé *statiquement*, alors que des programmes tels que `Curt` sont écrits en `Prolog`, un langage *logique* typé *dynamiquement*. Les conséquences principales de ce choix sont, notamment, la possibilité de détecter plus d'erreurs de programmation à la compilation, ainsi qu'un gain d'efficacité à l'exécution. D'autre part, les fonctionnalités de `Nessie` sont plus génériques que celles offertes par les programmes tels que `Curt`, et cela à deux niveaux. Au niveau syntaxique, la structure utilisée pour guider la construction sémantique est une structure d'arbre abstrait. Cette structure nous paraît bien adaptée à la représentation de la syntaxe d'une phrase, dans la mesure où elle peut être engendrée par un large éventail de formalismes syntaxiques (par exemple tous ceux qui sont faiblement contextuels) et n'est pas attachée à un formalisme syntaxique spécifique. Au niveau sémantique, l'intérêt de `Nessie` est qu'il permet à l'utilisateur de spécifier complètement le langage logique (c'est-à-dire la variante de TYn) à utiliser (déclaration des types et constantes) ainsi que les représentations des mots, ces dernières pouvant être données en tenant compte de leurs similitudes éventuelles, grâce aux notions de classes de mots ouvertes

et fermées dont on peut tenir compte dans le lexique de `Nessie`. Enfin, nous avons dégagé au chapitre 2 deux degrés de compositionnalité des théories sémantiques (selon que les représentations sémantiques associées à différentes occurrences d'un même mot sont ou non congruentes modulo α-équivalence), et nous avons montré que `Nessie` permet de traiter ces deux sortes de compositionnalité.

Au chapitre 3, nous avons conçu une version modifiée de `Curt` dans laquelle le calcul sémantique n'était plus réalisé en `Prolog` comme dans la version initiale de `Curt`, mais confié à `Nessie`. Ce travail, qui a nécessité une modification de la DCG de `Curt` pour qu'elle produise les arbres utilisables par `Nessie`, nous a permis de nous assurer du bon fonctionnement de `Nessie`. En l'intégrant à `Curt`, nous avons en effet pu montrer que les représentations sémantiques construites par `Nessie` étaient α-équivalentes à celles construites par `Curt`. Le travail d'intégration de `Nessie` à `Curt` constitue également une base sur laquelle s'appuient les expérimentations en construction sémantique réalisées au chapitre 6 et, plus généralement, il montre comment on peut procéder pour modifier une grammaire de sorte qu'elle produise des arbres utilisables par `Nessie`, enseignement qui a été exploité au chapitre 4. Cette première expérience de construction sémantique a aussi été l'occasion d'une première confrontation avec les limites d'expressivité de TYn. Nous avons en effet montré en section 3.3 que, dans le λ-calcul simplement typé, le traitement des coordinations était plus complexe qu'en l'absence de types, dans la mesure où les conjonctions de coordination peuvent coordonner plusieurs familles de constituants (noms, groupes nominaux, groupes verbaux...). Nous avions résolu ce problème en introduisant une coordination pour chaque famille qui peut être coordonnée, mais cette solution n'est pas entièrement satisfaisante, notamment parce qu'elle nécessite, lors de l'analyse syntaxique d'une conjonction de coordination, de choisir la conjonction appropriée, démarche qui n'est pas nécessaire en l'absence de types.

Ces premiers résultats ayant été obtenus, nous avons entrepris une seconde expérience, plus ambitieuse, et dont le but était de fournir de nouveaux éléments de réponse à la question de la pertinence de TYn pour la construction sémantique et l'inférence. En l'occurrence, il s'agissait de construire des modèles restituant le contenu temporel et aspectuel de phrases polonaises simples, ce que nous avons fait au chapitre 4. Le chapitre 4 est donc le premier où nous avons pu étudier l'impact du passage à TYn sur l'inférence, le chapitre précédent ne portant, pour sa part, que sur la construction sémantique. Dans ce chapitre, nous avons montré que l'utilisation de TYn présente des avantages tant pour la construction sémantique que pour l'inférence. Au niveau de la construction sémantique, l'utilisation de TYn nous a incités à associer aux éléments du lexique tels que les verbes des représentations sémantiques plus riches que celles qui leur avaient été associées dans le cas du λ-calcul non typé. L'avantage d'un tel choix est double. D'une part, si la représentation sémantique associée à un mot restitue toute l'information sémantique contenue dans ce mot, on évite le recours à l'arbre syntaxique pour faire remonter l'information qui n'avait pu être incluse dans la représentation sémantique (nous faisons ici référence au traitement de la sémantique temporelle des verbes vu en section 4.3). D'autre part, nous avons pu remarquer que, plus l'information sémantique était traitée localement, plus les représentations sémantiques obtenues étaient susceptibles d'être réutilisées : une fois le traitement du temps et des événements confiné aux verbes, il a été possible d'intégrer leur représentation à celles associées aux autres catégories syntaxiques dans TY1). Au niveau de l'inférence, nous avons vu que le fait d'utiliser des représentations sémantiques typées permet de générer auto-

matiquement, à partir des informations de typage, des axiomes qui facilitent sensiblement la construction de modèles pertinents. Insistons sur le fait que cette possibilité est sans équivalent lorsque l'on travaille en logique du premier ordre. Le chapitre 4 a aussi été l'occasion d'une réflexion à propos du choix des axiomes à passer aux outils d'inférence, et qui tend à montrer que ce choix doit être fait avec précaution, d'une part parce que la complexité des tâches d'inférence est exponentielle en la taille de la théorie considérée, ce qui signifie qu'ajouter un axiome rend le travail d'inférence beaucoup plus complexe, d'autre part parce qu'il faut veiller, lorsqu'on passe un axiome à un constructeur de modèle, à ce que cet axiome ne déclenche pas la construction de modèles infinis, qu'un tel outil ne saurait produire. Enfin, les deux derniers résultats significatifs de ce chapitre sont une théorie du temps et des événements indépendante de la langue, et un algorithme permettant, à partir d'une telle théorie, d'une représentation sémantique et d'un modèle minimal consistant à la fois avec la théorie et la représentation, de trouver tous les modèles sémantiquement pertinents (du point de vue du temps et de l'aspect) consistants avec la théorie et la représentation.

Au chapitre 5, nous avons remarqué que, si l'on souhaitait étudier d'autres phénomènes sémantiques que celui vu au chapitre 4, il était indispensable d'être capable de construire des représentations sémantiques non plus seulement pour des phrases isolées comme on l'avait fait jusqu'alors, mais aussi pour des discours comportant plusieurs phrases. Suivant la démarche adoptée au début de la thèse pour choisir un système logique et qui avait abouti au choix de TYn, nous avons choisi d'étudier les outils mis à notre disposition par la sémantique formelle pour calculer de façon compositionnelle la sémantique des discours. Nous en avons présenté deux, à savoir la DRT compositionnelle introduite dans (Muskens, 1996c) et le traitement compositionnel de la dynamicité introduit dans (de Groote, 2006). Lors de la présentation de ces deux théories, nous avons montré qu'elles n'ont pas le même degré de compositionnalité, dans la mesure où la DRT compositionnelle associe des représentations non α-équivalentes à différentes occurrences de certaines entrées lexicales, tandis qu'avec le traitement compositionnel de la dynamicité, toutes les occurrences d'un même lexème ont toujours des représentations α-équivalentes. Nous avons également établi une propriété de la DRT compositionnelle qui nous paraît importante : bien que cette théorie se propose de traduire aussi fidèlement que possible les boîtes par des λ-termes de TYn, on ne peut construire des termes reflétant fidèlement la structure des boîtes qu'ils représentent en se restreignant à la seule règle de réduction fournie par TYn, à savoir la β-réduction. On est donc en présence de l'alternative suivante : ou bien on veut construire des termes reflétant fidèlement la structure des boîtes qu'ils représentent, et alors il faut sortir de TYn et accepter d'adjoindre à la β-réduction une règle de réécriture d'ordre supérieur permettant de fusionner les boîtes ; ou bien on souhaite que la construction se passe uniquement dans TYn, mais alors il faut accepter que les termes que l'on construit ne contiennent pas la structure de la boîte qu'ils représentent.

Forts de ces résultats théoriques, nous passons, au chapitre 6, à la mise en œuvre des deux théories qui viennent d'être présentées. En ce qui concerne la DRT compositionnelle, son implantation permet de voir quelles sont les conséquences pratiques des remarques qui ont été faites à son sujet au chapitre précédent. Ainsi, le fait que les occurrences de certains éléments du lexique tels que les déterminants doivent être distinguées les unes des autres a pour conséquence que les arbres d'analyse doivent être annotés pour que cette information de distinction puisse parvenir à la couche sémantique. Le fait que le calcul des DRSs nécessite le recours à une

règle de réécriture autre que la β-réduction, quant à lui, oblige à mettre en œuvre cette règle, ce qui ne peut être fait qu'à l'extérieur de TYn et donc de `Nessie`. L'approche de (de Groote, 2006) peut, quant à elle, être mise en œuvre en n'ayant recours qu'à `Nessie`. Nous avons cependant observé que, lorsque les représentations à base de continuations sont traduites en logique du premier ordre, elles contiennent des occurrences de top (valeur propositionnelle toujours vraie) qui ne peuvent être éliminées au moyen de la seule règle de β-réduction. Enfin, nous avons utilisé les représentations qui viennent d'être mentionnées comme base pour mener une réflexion sur la résolution d'anaphores dans TYn. Cette réflexion poursuit les investigations sur les intérêts computationnels de TYn entreprises au chapitre 4. Insistons sur le fait que cette réflexion n'avait pas pour but de proposer de nouvelles heuristiques de résolution d'anaphores, mais plutôt de proposer une architecture logicielle permettant d'adapter les heuristiques existantes à TYn. L'architecture qui a été proposée permet d'utiliser deux stratégies pour résoudre les anaphores. La première suppose que chaque syntagme à remplacer peut être représenté par l'application d'une fonction à un argument qui est une liste d'antécédents possibles pour ce syntagme. Cette fonction est alors implantée au sein d'un solveur d'anaphores capable de produire à partir d'une représentation sémantique avec problèmes anaphoriques toutes les représentations dans lesquelles ces problèmes ont été résolus. La seconde approche est basée sur l'inférence. Elle s'appuie sur une théorie axiomatisant les propriétés de la relation anaphorique, comme par exemple le fait que pour tout pronom, son genre et celui de son antécédent doivent coïncider. Les lectures d'un discours ne respectant pas cette contrainte peuvent alors être éliminées car elles sont inconsistantes avec la théorie considérée. La mise en œuvre de cette seconde approche a été grandement facilitée par l'utilisation du générateur de théorie qui avait été introduit au chapitre 4 et apporte donc un nouvel exemple de l'intérêt de TYn en sémantique computationnelle. Nous avons cependant aussi mis en évidence certaines limitations à l'expressivité de TYn. En effet, La règle de fusion de DRSs de (Muskens, 1996c), la suppression des occurrences de top dans les représentations de (de Groote, 2006) et les fonctions utilisées pour la résolution d'anaphores ont pour point commun de ne pouvoir être exprimées au sein de TYn, ce qui nous a conduits à recourir à des outils externes manipulant les représentations préalablement construites.

Enfin, le chapitre 7 propose une modélisation de l'interface syntaxe-sémantique qui a été mise en œuvre dans `Nessie`. Nous y montrons qu'un lexique `Nessie` est équivalent à une grammaire catégorielle abstraite du second ordre, cette équivalence étant à entendre au sens fort : étant donné un lexique `Nessie` L_N, il existe une grammaire catégorielle abstraite du second ordre $\mathscr{G}$ telle que pour tout terme t de TYn, dire qu'il existe un arbre `Nessie` dont la sémantique pour L_N est t est équivalent à dire que t appartient au langage objet de $\mathscr{G}$. Les grammaires catégorielles du second ordre ont deux propriétés qui donnent une importance particulière à ce résultat. D'une part, il a été montré que ces grammaires ont une expressivité équivalente à celle de plusieurs formalismes syntaxiques largement utilisés en Traitement Automatique des Langes, comme par exemple les grammaires d'arbres adjoints et plus généralement les grammaires reconnaissant les langages faiblement contextuels. Comme cette équivalence préserve elle aussi les structures de dérivation, ce résultat donne une preuve théorique du fait que `Nessie` pourra être interfacé avec les analyseurs syntaxiques pour ces formalismes. Intuitivement, ce qui est exprimé ici est que les structures d'arbres dont `Nessie` a besoin en entrée peuvent être produites lors de l'analyse syntaxique pour les formalismes qui viennent d'être mentionnés. D'autre part, il existe un résultat de réversibilité pour les grammaires catégorielles abstraites du second ordre.

D'après ce résultat, il est possible, étant donné un terme u du langage objet, de produire un terme t du langage abstrait et dont l'image est u (autrement dit, le problème dit du parsing est décidable pour les grammaires du second ordre). Du point de vue de `Nessie`, cela signifie que ses lexiques sont eux aussi réversibles, et qu'il est donc possible, étant donné un terme, de trouver un arbre dont ce terme est la représentation sémantique. Or, comme les formalismes syntaxiques qui viennent d'être cités peuvent être exprimés à l'aide d'une ACG, cela signifie qu'il est facile, à partir d'un arbre, de produire la chaîne d'entrée dont cet arbre représente l'analyse (d'après la modélisation de l'analyse syntaxique donnée au chapitre précédent, la chaîne d'entrée est l'image par le lexique de l'ACG de son arbre d'analyse). On en déduit qu'il est possible, à partir d'une représentation sémantique, de produire un texte dont la sémantique est la représentation de laquelle on est parti. Autrement dit, cela signifie que les lexiques de `Nessie` peuvent être utilisés aussi bien en génération qu'en analyse.

8.2 Pistes pour des recherches ultérieures

Un premier travail restant à accomplir porte sur le chapitre 5. Nous y avions modélisé la DRT compositionnelle à l'aide de trois systèmes de réécriture d'ordre supérieur. Il reste à prouver la terminaison de ces trois systèmes, ainsi que la confluence des deux derniers, modélisant d'une part la construction de termes reflétant la structure des boîtes qu'ils représentent au prix d'un détour hors de TYn, d'autre part la construction, à l'intérieur de TYn, de termes dans lesquels la structure de la boîte qu'ils représentent n'apparaît plus après β-réduction.

Par ailleurs, les résultats du chapitre 7 peuvent être prolongés dans au moins deux directions. D'une part, nous avons montré que les lexiques de `Nessie` sont réversibles, c'est-à-dire qu'il est possible, à partir d'un lexique `Nessie` et d'une représentation sémantique M sur ce lexique, de produire un arbre ayant M pour représentation sémantique. Il serait intéressant, nous semble-t-il, de développer un programme mettant en œuvre cette possibilité. L'arbre ainsi obtenu pourrait être passé à l'analyseur syntaxique, qui doit pouvoir à partir de l'arbre produire un texte d'entrée. On obtiendrait ainsi une chaîne complète de génération, partant d'une représentation sémantique et aboutissant à un texte dans une langue quelconque. Bien sûr, ceci suppose que les arbres d'analyse contiennent suffisamment d'information pour produire une chaîne d'entrée, ce qui nous amène à une deuxième direction de recherche liée aux résultats du chapitre 7, à savoir le branchement de `Nessie` à un analyseur syntaxique. En effet, après avoir montré qu'il est possible, en théorie, d'utiliser `Nessie` avec des analyseurs basés sur des formalismes grammaticaux d'expressivité équivalente à celle d'une ACG du second ordre, il nous paraît souhaitable de mettre en œuvre de façon concrète le branchement de `Nessie` avec un tel analyseur. Il s'agit donc de modifier un analyseur existant (ou les arbres d'analyse qu'il produit) de sorte que ceux-ci puissent être utilisés par `Nessie`. On comprend dès lors que les deux tâches qui viennent d'être citées sont intimement liées, puisqu'elles portent toutes deux sur les arbres qui servent d'interface entre la phase d'analyse syntaxique et celle de construction sémantique.

La possibilité d'utiliser `Nessie` conjointement à un analyseur syntaxique pour lequel des grammaires à large couverture sont disponibles serait, à notre avis, de nature à encourager l'utilisation de TYn pour le calcul sémantique. Disposer d'une telle interface permettrait aussi de pouvoir utiliser `Nessie` avec des grammaires couvrant plus de phénomènes que les grammaires

considérées pendant cette thèse. Il serait intéressant d'explorer la construction sémantique pour de telles grammaires, et d'observer le comportement de `Nessie` dans cette nouvelle situation. On pourrait par exemple se demander si les lexiques `Nessie` tels qu'ils se présentent actuellement conviennent bien à une utilisation avec des grammaires plus conséquentes que celles vues ici. En outre, une fois le branchement de `Nessie` à un analyseur syntaxique réalisé, une telle infrastructure pourrait être utilisée pour construire des représentations sémantiques plus complexes que celles vues jusqu'à présent, poursuivant ainsi les expérimentations menées aux chapitres 4 et 6. En particulier, un tel cadre logiciel pourrait être utilisé pour explorer des théories sémantiques plus complexes que celles vues jusqu'ici, comme par exemple des versions de TYn intégrant le traitement de la dynamicité, les événements, ou d'autres entités abstraites. D'autres expérimentations en construction sémantique pourraient porter sur des phénomènes tels que la résolution d'anaphores, le calcul des présuppositions, les ellipses, les ambiguïtés de portée et les interactions qui peuvent exister entre tous ces phénomènes. On pourrait aussi s'intéresser à des formalismes tels que la SDRT introduite dans (Asher, 1993) et se demander si son traitement est possible à l'intérieur de TYn et, le cas échéant, si un tel traitement est computationnellement intéressant.

Des expérimentations telles que celles que nous venons de proposer nous ramènent quant à elles au questionnement qui a été ébauché dans la conclusion du chapitre 6. En effet, quelle stratégie adopter pour remédier au manque d'expressivité de TYn ? Une réponse très générale à cette question pourrait être qu'il y a deux stratégies possibles. L'une consiste à décider qu'un formalisme de représentation sémantique ne doit pas nécessairement rendre compte de tous les calculs que l'on souhaite faire sur les représentations sémantiques, et que ces calculs peuvent être pris en charge par des outils externes. Cette stratégie a été illustrée à la fois par la première technique mise en œuvre pour résoudre des anaphores (à savoir celle utilisant un programme externe de résolution d'anaphores) et par le traitement des DRSs de (Muskens, 1996c). Une deuxième stratégie consiste à considérer que les formalismes logiques doivent être suffisamment expressifs, non seulement pour que l'on puisse y exprimer la sémantique des expressions des langues humaines, mais aussi pour rendre compte de tous les calculs que peut induire la construction sémantique (simplifications, résolution des anaphores et des ellipses…). La seconde approche mise en œuvre pour résoudre les anaphores au chapitre 6 s'apparente à cette seconde stratégie. En d'autres termes, les deux stratégies que nous venons d'exposer posent la question de la représentationnalité : est-il préférable, lors de la construction sémantique, de recourir à une représentation intermédiaire qui peut être manipulée par des outils externes, ou vaut-il mieux n'utiliser que des formules logiques, non représentationnelles, appartenant à une logique suffisamment expressive pour que l'on puisse y exprimer *tous* les calculs nécessaires à la construction sémantique.

Nous pensons que l'une des orientations possibles pour les recherches consécutives à cette thèse pourrait être de chercher une réponse à cette question. Il reste alors à se poser la question de la démarche à adopter pour explorer ces deux voies. Pour notre part, nous pensons que chacune des deux stratégies qui ont été mentionnées plus haut mérite d'être explorée.

En ce qui concerne la recherche d'une logique expressive, les limitations de l'expressivité de TYn auxquelles nous avons été confrontés dans cette thèse, et qui ont été rappelées à la section précédente, nous laissent penser que, peut-être, un système de type plus expressif pourrait être intéressant. On peut par exemple penser à des systèmes « à la ML » qui permettent l'utilisation

de types polymorphes comme ceux dont nous déplorions l'absence lorsque nous avons évoqué le traitement des coordinations au chapitre 3. Une autre piste possible est le recours au Calcul des Constructions Inductives (Coquand et Huet, 1988; Paulin-Mohring, 1993), un λ-calcul typé suffisamment expressif pour permettre la représentation et la manipulation de programmes et de preuves. Un tel système nous semble *a priori* intéressant, par exemple parce que la capacité de détecter l'inconsistance de représentations sémantiques par typage croît avec l'expressivité du système de type utilisé. Il sera cependant nécessaire, si l'on décide de mettre en œuvre de telles logiques, de s'interroger sur la meilleure façon d'y parvenir. En effet, on peut par exemple remarquer que l'algorithme de typage utilisé par `Nessie` et qui a été présenté en section 2.2.1 fonctionne de façon ascendante, déduisant le type d'un terme de celui de ses sous-termes. Or un tel algorithme n'est pas adapté à des systèmes de types plus expressifs que TYn. On devrait donc se demander si, pour explorer de telles logiques, il est préférable de remplacer l'algorithme actuel par un algorithme plus générique (par exemple basé sur la résolution d'équations en types), où s'il est préférable de recourir à des approches radicalement différentes de celle proposée dans `Nessie`, consistant par exemple à recourir plus systématiquement à des outils spécialisés dans la manipulation efficace de λ-termes et qui pourraient être appelés, tels des « boîtes noires », par un outil de construction sémantique similaire à `Nessie` dans son principe, mais plus spécialisé encore.

Cette approche visant à utiliser une logique très expressive peut cependant avoir également un certain nombre d'inconvénients. D'une part, pour que le calcul sémantique puisse être effectué au sein d'une logique telle que le Calcul des Constructions Inductives évoqué ci-dessus, il est nécessaire d'encoder dans cette logique tous les algorithmes utilisés. Outre la difficulté de cette tâche, il est important de signaler son coût en termes de calculs. Par exemple, la certification rend l'accès à un tableau linéaire en sa taille, alors que la même opération effectuée dans un cadre non certifié peut être faite en temps constant. D'autre part, demander que toutes les transformations des représentations sémantiques se fassent au sein d'une logique donnée peut avoir pour conséquence l'impossibilité d'utiliser des outils existants, alors même que ceux-ci pourraient être d'une grande aide. Rappelons en effet que dans le contexte du Traitement Automatique des Langues qui nous intéresse ici, l'efficacité peut être un critère important, que ce soit l'efficacité en termes de temps de développement, ou celle de l'exécution des processus de traitement proprement dits.

Ces constatations sont l'une des justifications pour explorer l'autre voie mentionnée ci-dessus, à savoir celle du recours aux outils externes et donc à des représentations que l'on s'autorise à manipuler. Des outils spécialisés ont en effet été développés pour résoudre des tâches telles que la résolution d'anaphores ou des ambiguïtés de portée. L'une des conditions qu'il est nécessaire de satisfaire pour utiliser de tels outils est de construire les représentations dont ils ont besoin, et, dans la mesure du possible, de le faire de façon compositionnelle. En ce qui concerne la construction de représentations sous-spécifiées à partir desquelles les ambiguïtés de portée peuvent être résolues, par exemple, il n'existe à notre connaissance aucune méthode de construction complètement compositionnelle de telles représentations. Il nous paraît donc souhaitable de rechercher de telles méthodes. Cette recherche pourrait d'ailleurs bénéficier du traitement de la dynamicité proposé dans (de Groote, 2006). Les contextes pourraient en effet être utilisés aussi pour accumuler des contraintes de portée lors de la construction sémantique, la β-réduction permettant ensuite de les regrouper, de la même façon qu'elle permettait de calculer l'ensemble des

antécédents possibles pour un pronom dans le travail qui vient d'être mentionné.

On le voit, les deux stratégies présentent des avantages et des inconvénients, et chacune peut conduire à des résultats intéressants. C'est pourquoi, nous pensons que chacune d'elle mérite d'être étudiée, et que cette étude devrait être menée avec le moins de préjugés possible quant aux résultats qui pourraient être obtenus. En particulier, l'idée selon laquelle il faudrait à tout prix trouver une logique suffisamment expressive pour encoder tous les calculs dont on a besoin en construction sémantique constitue, à notre avis, un biais méthodologique. Pour nous, la question de l'existence d'une telle logique demeure ouverte, et nous pensons que présupposer de son existence est prématuré.

Bibliographie

K. ADUKIEWICZ – « Die syntaktische Konnexität », *Studia Philosophica* **1** (1935), p. 1–27, English translation in *Polish Logic* (S. McCall éd.), Oxford, 1967.

A. ALSTEIN et P. BLACKBURN – « An aspectual classification of polish verbs », Unpublished manuscript, 2007.

E. ALTHAUS, D. DUCHIER, A. KOLLER, K. MEHLHORN, J. NIEHREN et S. THIEL – « An efficient graph algorithm for dominance constraints », *Journal of Algorithms* **48** (2003), no. 1, p. 194–219.

N. ASHER et A. LASCARIDES – *Logics of conversation*, Cambridge University Press, 2003.

N. ASHER – *Reference to abstract objects in discourse*, Kluwer Academic Publishers, 1993.

E. BACH – *Informal lectures on formal semantics*, State University of New York Press, 1989.

Y. BAR-HILLEL, C. GAIFMAN et E. SHAMIR – « On categorial and phrase structure grammars », *Bulletin of the Research Council of Israel* **9F** (1960), p. 1–11.

J. BARWISE – « Noun phrases, generalized quantifiers and anaphora », Generalized Quantifiers : Linguistic and Logical Approaches (P. Gärdenfors, éd.), Reidel, Dodrecht, 1987, p. 1–29.

P. BLACKBURN, M. DE RIJKE et Y. VENEMA – *Modal logic*, Cambridge University Press, 2001.

P. BLACKBURN et J. BOS – *Representation and inference for natural language. A first course in computational semantics*, CSLI, 2005.

P. BLACKBURN et J. BOS – *Representation and inference for natural language*, CSLI Press, 2005.

P. BLACKBURN et S. HINDERER – « Generating models for temporal representations », *RANLP 2007, Proceedings of the international conference on Recent Advances in Natural Language Processing*, 2007, p. 69–75.

P. BLACKBURN et S. HINDERER – « From TYn to DRT : an implementation », *3rd International Language & Technology Conference - L&TC'07* (Poznam Pologne), 2007, p. 384–388.

P. BLACKBURN, C. GARDENT et M. DE RIJKE – « Back and forth through time and events », *Proceedings of the Ninth Amsterdam Colloquium*, 1993, p. 161–175.

P. BLACKBURN, J. BOS et K. STREIGNITZ – *Prolog, tout de suite !*, College Publications, 2007.

J. BOS – « Predicate Logic Unplugged », *Proceedings of the Tenth Amsterdam Colloquium* (P. Dekker et M. Stokhof, éds.), 1996, p. 133–143.

J. BOS, E. MASTENBROEK, S. MCGLASHAN, S. MILLIES et M. PINKAL – « A Compositional DRS-based Formalism for NLP Applications », *IWCS-1, Proceedings of the First International Workshop on Computational Semantics* (H. Bunt, R. Muskens et G. Rentier, éds.), 1994.

J. BOS – « Towards wide-coverage semantic interpretation », *IWCS-6, Proceedings of Sixth International Workshop on Computational Semantics*, 2005, p. 42–53.

J. BOS et K. MARKERT – « When logical inference helps determining textual entailment (and when it doesn't) », *Pascal, Proceedings of the Second Challenge Workshop on Recognizing Textual Entailment*, 2006.

D. BÜRING – *Binding theory*, Cambridge Textbooks in Linguistics, Cambridge University Press, 2005.

A. CHURCH – « A formulation of the simple theory of types », *Journal of Symbolic Logic* **5** (1940), p. 56–68.

A. COPESTAKE, D. FLICKINGER, R. MALOUF, S. RIEHEMANN et I. SAG – « Translation using Minimal Recursion Semantics », *TMI-95, Proceedings of the Sixth International Conference on Theoretical and Methodological Issues in Machine Translation*, 1995, p. 15–32.

T. COQUAND et G. HUET – « The calculus of constructions », *Information and Computation* **76** (1988).

N. DE BRUIJN – « Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation », *Indagationes Mathematicae* **34** (1972), p. 381–392.

P. DE GROOTE – « Tree-adjoining grammars as abstract categorial grammars », *TAG+6, Proceedings of the Sixth International Workshop on Tree Adjoining Grammars and Related Frameworks*, 2003, p. 145–150.

P. DE GROOTE (éd.) – *The curry-howard isomorphism*, Cahiers du Centre de Logique, Universite Catholique de Louvain, vol. 8, Academia-Bruyland, 1995.

P. DE GROOTE – « Towards abstract categorial grammars », *ACL '01, Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics and of the 10th Conference of the European Chapter of the Association for Computational Linguistics*, 2001, p. 148–155.

P. DE GROOTE – « Towards a Montagovian Account of Dynamics », *Semantics and Linguistic Theory XVI, to appear*, CLC Publications, 2006.

P. DE GROOTE et S. POGODALLA – « On the expressive power of abstract categorial grammars: Representing context-free formalisms », *Journal of Logic, Language and Information* **13** (2004), no. 4, p. 421–438.

D. DOWTY – *Word meaning and Montague grammar*, Springer, 1979.

G. FREGE – « Über Sinn und Bedeutung », *Zeitschrift für Philosophie und philosophische Kritik* **100** (1892), p. 25–50.

J. FRIEDMAN et D. S. WARREN – « λ-normal forms in an intensional logic for english », *Studia Logica* **39** (1979), p. 311–324.

D. GALLIN – *Intentional and higher-order modal logic*, North Holland, 1975.

GEACH – *Reference and generality*, Cornell University Press, 1962.

J. GROENENDIJK et M. STOKHOF – « Questions », Handbook of Logic and Language (J. Van Benthem et A. Ter Meulen, éds.), MIT Press and North-Holland, 1997, p. 1055–1124.

J. GROENENDIJK et M. STOKHOF – « Studies in the semantics of questions and the pragmatics of answers », Thèse, University of Amsterdam, 1984.

J. GROENENDIJK – « Questions and answers : Semantics and logic », *Proceedings of the 2nd CologNET-ElsET Symposium* (M. M. R. Bernardi, éd.), 2003.

T. GUNJI – « Towards a computational theory of pragmatics — discourse, presupposition, and implicature », Thèse, Ohio State University, 1981.

I. HEIM – « The semantics of definite and indefinite noun phrases », Thèse, University of Massachusetts, 1982.

I. HEIM et A. KRATZER – *Semantics in Generative Grammar*, Blackwell Publishing, 1998.

L. HENKIN – « Completeness in the simple theory of types », *Journal of Symbolic Logic* **15** (1950), p. 81–91.

J. HOBBS et S. ROSENSCHEIN – « Making computational sense of Montague's intensional logic », *Artificial Intelligence* **9** (1978), no. 3, p. 287–306.

H. KAMP – « A Theory of Truth and Semantic Representation », Truth, Interpretation and Information: Selected Papers from the Third Amsterdam Colloquium (J. Groenendijk, T. Janssen et M. Stokhof, éds.), Foris, 1984, p. 1–41.

H. KAMP et U. REYLE – *From discourse to logic*, Kluwer, Dordrecht, 1993.

A. KOLLER, J. NIEHREN et S. THATER – « Bridging the gap between underspecification formalisms : Hole semantics as dominance constraints », *EACL'2003, Proceedings of the 11th Meeting of the European Chapter of the Association for Computational Linguistics*, 2003, p. 195–202.

J. LAMBEK – « The mathematics of sentence structure », *American Mathematical Monthly* **65** (1958), p. 154–169.

J. LAMBEK – « On the calculus of syntactic types », Structure of Language and its Mathematical Aspects (R. Jakobsen, éd.), Providence, 1961, p. 166–178.

J. LAMBEK – « Categorial and categorical grammars », Categorial Grammars and Natural Language (B. Oehrle et Wheeler, éds.), Reidel, 1988, p. 297–317.

J. LANDSBERGEN – « Machine translation based on logically isomorphic Montague Grammars », *Proceedings of COLING 82* (J. Horecky, éd.), North-Holland, 1982.

P. LETOUZEY – « Programmation fonctionnelle certifiée – l'extraction de programmes dans l'assistant Coq », Thèse, Université Paris-Sud, 2004.

M. MAIN et D. BENSON – « Denotational Semantics for "Natural" Language Question-Answering Programs », *American Journal of Computational Linguistics* **9** (1983), no. 1, p. 11–21.

A. MARTINICH (éd.) – *The philosophy of language*, Oxford University Press, 1996.

R. MITKOV – *Anaphora resolution*, Longman, 2002.

A. MŁYNARCZYK – « Aspectual Pairing in Polish », Thèse, University of Utrecht, 2004, LOT Dissertation Series 87.

R. MONTAGUE – « On the nature of certain philosophical entities », *The Monist* **53** (1969), p. 159–194, Reprinted in Montague (1974), 148–187.

R. MONTAGUE – « English as a Formal Language », Linguaggi nella Società e nella Tecnica (B. Visentini et al., éds.), Milan, Edizioni di Comunita, 1970, Reprinted in Montague (1974), 188–221, p. 189–224.

R. MONTAGUE – « Pragmatics », Contemporary Philosophy : a Survey (R. Klibansky, éd.), Florence, La Nuova Italia Editrice, 1968, Reprinted in Montague (1974), 95–118, p. 102–122.

R. MONTAGUE – « Pragmatics and intensional logic », *Synthese* **22** (1970), p. 68–94, Reprinted in Montague (1974), 119–147.

R. MONTAGUE – « Universal Grammar », *Theoria* **36** (1970), p. 373–398, Reprinted in Montague (1974), 222-246.

R. MONTAGUE – « The Proper Treatment of Quantification in Ordinary English », Approaches to Natural Language (J. Hintikka, J. Moravcsik et P. Suppes, éds.), Reidel, Dordrecht, 1973, Reprinted in Montague (1974), 247–270, p. 221–242.

R. MONTAGUE – *Formal Philosophy. Selected papers of Richard Montague. Edited and with an introduction by Richmond H. Thomason*, Yale University Press, 1974.

R. MUSKENS – *Meaning and partiality*, Studies in Logic, Language and Information, CSLI Publications, 1996.

R. MUSKENS – « Combining montague semantics and discourse representation », *Linguistics and Philosophy* **19** (1996), p. 143–186.

R. MUSKENS – « Tense and the logic of change », Lexical Knowledge in the in the Organisation of Language (U. Egli, P. Pause, C. Schwartz, A. von Stechow et G. Wienold, éds.), Benjamins, 1995, p. 147–183.

R. MUSKENS – « Language, lambdas, and logic », Resource Sensitivity in Binding and Anaphora (G.-J. Kruijff et R. Oehrle, éds.), Kluwer, 2003, p. 23–54.

R. MUSKENS – « Combining Montague Semantics and Discourse Representation », *Linguistics and Philosophy* **19** (1996), p. 143–186.

R. OEHRLE – « Term-labeled categorial type systems », *Linguistics and Philosophy* **17** (1994), p. 633–678.

R. OEHRLE – « Some 3-dimensional system of labelled deduction », *Bulletin of the IGPL* **3** (1995), p. 429–488.

B. PARTEE – « The Development of Formal Semantics in Linguistic Theory », The Handbook of Contemporary Semantic Theory (S. Lappin, éd.), Blackwell, 1997, p. 11–38.

B. PARTEE – « Montague Grammar », Handbook of Logic and Language (J. Van Benthem et A. Ter Meulen, éds.), Elsevier Science, 1997, p. 5–91.

C. PAULIN-MOHRING – « Inductive definitions in the system Coq - rules and properties », *Proceedings of the conference Typed Lambda Calculi and Applications* (M. Bezem et J.-F. Groote, éds.), Lecture Notes in Computer Science, vol. 664, Springer-Verlag, 1993, LIP research report 92-49.

S. POGODALLA – « Exploring a type-theoretic approach to accessibility constraint modelling », *Actes du colloque Journées sémantique et modélisation* (P. M. Nicholas Asher et L. Roussarie, éds.), 2008.

F. POTTIER – « Cαml », 2005.

S. SALVATI – « Problèmes de filtrage et problèmes d'analyse pour les Grammaires Catégorielles Abstraites », Thèse, Institut National Polytechnique de Lorraine, 2005.

S. SALVATI – « On the membership problem for non-linear abstract categorial grammars », *Proceedings of New Directions in Type-theoretic Grammars* (R. Muskens, éd.), 2007.

L. SCHUBERT et F. PELLETIER – « From English to Logic : Context-Free Computation of 'Conventional' Logical Translation », *Computational Linguistics* **8** (1982), no. 1, p. 26–44.

M. R. SHINWELL et A. M. PITTS – « Fresh Objective Caml user manual », Tech. Report 621, University of Cambridge, feb 2005.

A. TARSKI – « Der Wahrheitsbegriff in den formalisierten Sprachen (German translation of a book in Polish, 1933) », *Studia philosophica* **1** (1935), p. 261–405, English translation in A. Tarski, *Logic, Semantics, and Metamathematics*, Oxford University Press, 1956.

J. VAN BENTHEM – *Essays in logical semantics*, Reidel, 1986.

R. VAN DER SANDT – « Presupposition Projection as Anaphora Resolution », *Journal of Semantics* **9** (1992), p. 333–377.