Université de Montréal

Un outil pour développer et tester les grammaires d'unification polarisées

par
Simon Richard

Département de linguistique et de traduction
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès arts (M.A.)
en linguistique

décembre, 2016

Résumé

Ce mémoire présente un outil informatique pour développer et tester des grammaires d'unification polarisées, conçu entre autres pour faciliter la validation expérimentale du formalisme de la grammaire d'unification Sens-Texte (GUST) de Kahane & Lareau. Les fondements théoriques du formalisme GUP sont d'abord expliqués, puis nous décrivons la conception du module et l'évaluons avec un fragment de la grammaire GUST.

Mots clés : grammaires d'unification polarisées, unification de graphes, théorie Sens-Texte, linguistique informatique.

Abstract

This thesis presents a computer tool to develop and test polarized unification grammars that was built, namely, to help validate experimentally Kahane & Lareau's Meaning-Text Unification Grammar (MTUG) formalism. We first describe the theory behind the PUG formalism, then we explain how the module was developed and we test it using a fragment of the MTUG grammar.

Keywords: polarized unification grammars, graph unification, Meaning-Text Theory, computational linguistics.

Table des matières

Liste des tableaux

Liste des figures

Remerciements

Tout d'abord, merci à mes parents qui m'ont donné l'amour du langage et m'ont toujours encouragé à poursuivre des études supérieures.

Je tiens à remercier mon directeur de recherche, François Lareau, qui m'a supporté tout au long de ma maîtrise et m'a permis d'obtenir mon tout premier emploi dans une boîte de linguistique. J'ai beaucoup appris sous sa direction et je termine maintenant ma maîtrise avec une trousse de linguiste, d'informaticien et de chercheur bien remplie.

Merci aux gens de l'Observatoire de linguistique Sens-Texte pour leur enthousiasme et leur support inconditionnel. Je les remercie également de m'avoir nommé récipiendaire de la bourse OLST en 2014. Leur support financier a été grandement apprécié.

Un gros merci finalement à Adriana qui m'a permis de m'évader en dansant, à Linda qui m'a tenu compagnie lors de mes nombreuses excursions à la bibliothèque, et à mes trois mousquetaires qui m'ont accompagné tout au long de mon baccalauréat.

Chapitre 1. Introduction

Ce mémoire porte sur l'implémentation informatique du formalisme des grammaires d'unification polarisées (GUP). Comme les autres formalismes basés sur l'unification, GUP permet au linguiste de représenter formellement des fragments de langue naturelle et de combiner ces fragments pour former des descriptions plus complexes.

Une des particularités des grammaires d'unification polarisées est qu'on attribue aux structures de traits des **polarités** qui guident le processus d'unification et permettent de vérifier facilement la bonne formation des structures. Nous décrivons brièvement le formalisme à la section 1.2 et plus en détail au chapitre 2.

1.1 Buts et intérêts de la recherche

Bien que nous implémentions ici le formalisme GUP, nous souhaitons éventuellement utiliser notre outil pour tester et valider expérimentalement la grammaire d'unification Sens-Texte (Kahane, 2002, Kahane & Lareau, 2005a, 2005b, Lareau, 2008), une grammaire d'unification polarisée de nature linguistique et basée sur la théorie Sens-Texte. Le formalisme servira également à évaluer notre outil en retour.

Nos buts sont les suivants :

— Créer un outil informatique permettant le développement et la vérification de grammaires d'unification polarisées ;

— Faire en sorte que cet outil se prête à la validation de la grammaire d'unification Sens-Texte tout en restant générique ;

— Faire en sorte que cet outil soit performant, même si ce n'est qu'un prototype. Le formalisme GUP présente certains obstacles à ce niveau.

Notre objectif est de bâtir un module semblable à `nltk.featstruct`, qui permet d'unifier des structures de traits et s'intègre à d'autres modules (Bird et al., 2009), plutôt qu'une

application graphique complète comme MATE (Bohnet et al., 2000) ou XLE (Maxwell & Kaplan, 1993). L'outil permettra d'importer et manipuler des grammaires en mode texte, et ensuite d'exporter les structures générées. Il ne permettra pas toutefois de calculer l'ensemble des structures bien formées obtenables à partir d'une grammaire.

Notre projet s'adresse donc au chercheur qui s'intéresse aux grammaires d'unification polarisées, pour la description linguistique ou autre, et à des questions comme l'unification d'ensembles, l'associativité des grammaires polarisées et bien plus. Il s'adresse en particulier au linguiste Sens-Texte qui souhaite mettre à l'épreuve la grammaire d'unification Sens-Texte et expérimenter, par exemple, avec différents systèmes de polarités.

1.2 Les grammaires d'unification polarisées

Les grammaires d'unification polarisées sont un formalisme mathématique introduit par Kahane (2004). Elles se distinguent des autres grammaires d'unification par l'ajout de **polarités** aux structures de traits (ou **objets**) qui guident le processus d'unification en indiquant si une structure est complète ou non. Le concept de polarisation des structures précède les GUP—Kahane (2004) mentionne certains de ses prédécesseurs, dont Nasr (1995), Duchier & Thater (1999) et Perrier (2002).

Dans la grammaire d'unification Sens-Texte, par exemple, un objet peut être polarisé en blanc ou en noir. La polarité blanche représente un besoin et la noire une ressource. Si un objet blanc et un objet noir sont unifiés, un besoin est comblé par une ressource et l'objet résultant est polarisé en noir. Cet objet ne peut plus être unifié avec un objet noir puisqu'il n'y a plus de besoin à combler. La polarité noire bloque donc certaines unifications.

Les grammaires d'unification polarisées sont aussi particulières puisqu'elles manipulent ce qu'on appelle des **structures polarisées**. Celles-ci sont des ensembles d'objets et leur combinaison est coûteuse car il faut envisager à priori toutes les manières de combiner leurs objets—Francez & Wintner (2012, p. 179) mentionnent d'ailleurs que l'unification d'ensembles pose certains problèmes computationnels.

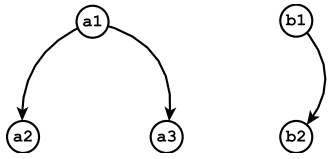


Figure 1 – Deux structures à trois et deux objets respectivement

Prenons comme exemple les deux structures de la figure 1. Les structures ont respectivement trois objets (a1, a2 et a3) et deux objets (b1 et b2). Le nom assigné à un objet (a1, b1, etc.) est arbitraire et nous sert uniquement à référer à cet objet. Les arcs entre les objets représentent des **fonctions structurantes**, qui sont équivalentes aux attributs à valeur complexe des grammaires d'unification classiques. Faute de connaître les paires fonction-valeur que contiennent les objets, nous devons envisager toutes les manières de combiner les objets des deux structures. Nous en montrons quelques-unes ici :

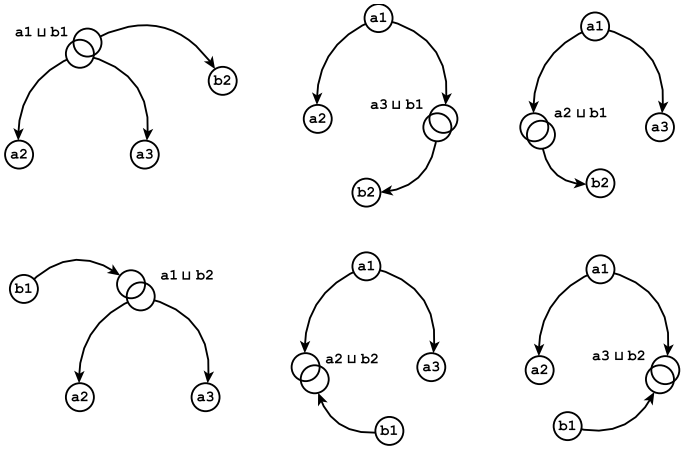


Figure 2 – Quelques combinaisons à envisager

Nous verrons plus tard que le nombre de combinaisons à envisager s'élève jusqu'à $2^n - 1$ où n est le produit du nombre d'objets dans les deux structures. Dans notre exemple, n est égal à $3 \times 2 = 6$ et les combinaisons théoriquement possibles s'élèvent donc à $2^6 - 1 = 63$.

Nous décrivons en détail le calcul des combinaisons à envisager au chapitre 3 et présentons quelques solutions trouvées pour réduire leur nombre (cf. § 3.4.1, p. 55). Ces solutions se veulent générales et applicables à n'importe quelle grammaire polarisée, et visent à retirer les combinaisons qui vont échouer ou donner des structures identiques :

— **Paires non viables :** si deux objets sont incompatibles, il est inutile de les inclure dans nos combinaisons ;

— **Unifications implicites :** lorsqu'une paire d'objets partage une fonction structurante, son unification déclenche celle des deux valeurs de la fonction. Si l'unification d'une paire A déclenche celle de B, nous savons d'avance qu'unifier seulement A donnera le même résultat qu'en unifiant A puis B. Nous pouvons donc nous contenter d'évaluer une seule de ces combinaisons.

— **Paires indirectes non viables :** si un objet est unifié successivement avec deux objets qui forment une paire non viable, nous savons déjà que la seconde unification va échouer et n'avons pas besoin d'évaluer la combinaison.

Nous décrivons en détail ces trois solutions au chapitre 3 et mesurons leur efficacité au chapitre 4. Celle-ci est cependant difficile à quantifier puisqu'elle dépend fortement du contenu des objets et varie donc selon les structures et les grammaires.

1.3 Structure de ce mémoire

Au chapitre 2, nous présentons la famille des grammaires d'unification, puis nous décrivons les grammaires d'unification polarisées que nous implémentons ici et la grammaire d'unification Sens-Texte qui servira à tester notre implémentation. Au chapitre 3, nous discutons de notre implémentation, en particulier de l'algorithme d'unification choisi et des méthodes utilisées pour réduire le coût computationnel mentionné plus haut. Au chapitre 4, nous décrivons notre évaluation de l'outil. Au chapitre 5, finalement, nous discutons des résultats de nos tests et effectuons une synthèse de notre mémoire.

1.4 Consulter le code source du module

Le code source du module est disponible dans un dépôt Git situé à l'adresse `https://github.com/simonrichard/guptools`. Des instructions pour télécharger, installer et utiliser le module sont également disponibles à la même adresse.

Chapitre 2. Les grammaires d'unification

Dans ce second chapitre, nous présentons brièvement la famille très diversifiée des grammaires d'unification ainsi que deux formalismes particuliers : les grammaires d'unification polarisées (Kahane, 2004, 2006) et la grammaire d'unification Sens-Texte (Kahane, 2002, Kahane & Lareau, 2005a, 2005b, Lareau, 2008).

2.1 Les grammaires d'unification traditionnelles

Les grammaires d'unification « traditionnelles » forment une famille diversifiée de formalismes qui partagent un mécanisme, l'**unification**. Notre description des grammaires d'unification n'est pas applicable à tous les formalismes, mais nous tâcherons tout de même de fournir une description aussi inclusive que possible en nous basant principalement sur Shieber (2003), Abeillé (2007) et Copestake (2002). Pour une présentation formelle des grammaires d'unification, voir l'ouvrage de Francez & Wintner (2012).

2.1.1 Les structures de traits

Les grammaires d'unification manipulent des **structures de traits**, soit des ensembles non ordonnés de paires attribut-valeur.

Un **attribut** est une fonction associée à une **valeur** dans une structure. Cette fonction est **partielle** puisqu'elle n'a pas forcément de valeur pour toutes les structures ; on dit aussi qu'elle est **déterministe** puisqu'elle retourne toujours la même valeur pour une même structure. Ce déterminisme fait en sorte qu'un attribut n'a jamais plus d'une valeur dans une structure bien formée. On représente une structure malformée, soit une structure avec des valeurs différentes pour un même attribut, par le symbole de l'échec « $\perp$ ».

Une valeur peut être **atomique** (une chaîne de caractères), **complexe** (une structure

enchâssée) ou **variable** (on ne connaît pas son contenu) [1].

Une valeur peut être partagée entre plusieurs attributs ; on parle alors de **réentrance** [2]. Celle-ci repose sur l'**identité** plutôt que l'**égalité** des valeurs—cela signifie que deux attributs peuvent pointer vers deux valeurs identiques mais distinctes, ou vers une seule et même valeur. Elle permet de rendre compte de phénomènes linguistiques tels que l'accord en nombre entre un verbe et son sujet. Par exemple, le verbe et son sujet partagent une même valeur pour l'attribut NOMBRE dans les figures 3 et 4.

Textuellement, une structure peut être représentée à l'aide d'une **matrice attribut-valeur** où chaque rangée correspond à une paire attribut-valeur. Lorsque la valeur est atomique, la colonne de droite comprend une simple chaîne de caractères, et lorsqu'elle est complexe, une matrice enchâssée. Une variable est représentée à l'aide de crochets vides [ ]. On étiquette parfois une valeur pour indiquer une réentrance—c'est le cas, notamment, pour la valeur des attributs NOMBRE (étiquetée 1) dans la figure 3.

$$\begin{bmatrix} \text{SUJET} & \begin{bmatrix} \text{NOMBRE} & \boxed{1} \end{bmatrix} \\ \text{ÉTIQUETTE} & \text{procrastiner} \\ \text{ACCORD} & \begin{bmatrix} \text{NOMBRE} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

Figure 3 – Matrice attribut-valeur

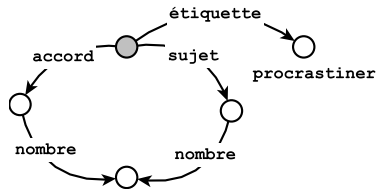


Figure 4 – Graphe orienté équivalent

Graphiquement, on utilise plutôt un **graphe orienté** [3]. Dans ce graphe, les nœuds terminaux correspondent à des valeurs atomiques, à des structures vides ou à des variables, et les nœuds non terminaux à des structures qui portent au moins une paire attribut-valeur. La **racine** du graphe (en gris) est le nœud qui correspond à la structure principale—tous les nœuds du graphe sont accessibles depuis cette racine. Chaque attribut est représenté par un arc qui pointe de son argument vers sa valeur, et la réentrance est indiquée par

1. Certains auteurs, dont Shieber (2003), voient les valeurs atomiques comme des structures de traits au même titre que les valeurs complexes. Nous supposons cependant ici que seule une valeur susceptible de porter des paires attribut-valeur est une structure de traits.
2. Abeillé utilise plutôt le terme **réentrée**.

plusieurs arcs qui pointent vers un même nœud, comme dans la figure 4.

Nous utiliserons parfois des **chemins** absolus pour référer à des valeurs dans une structure de traits, qu'elle soit représentée sous forme de matrice attribut-valeur ou de graphe. Un chemin est une liste d'attributs entre chevrons $\langle \ldots \rangle$ qui indique la route à suivre à partir de la structure principale. Dans un graphe, le chemin débute à la racine et indique les arcs à suivre. Il y a réentrance lorsque deux chemins pointent vers une même valeur.

Pour illustrer tout ce que nous avons vu jusqu'à présent, supposons que la structure de traits des figures 5 et 6 représente un individu. Nous savons plusieurs choses à son sujet, entre autres qu'il s'appelle Ethan et qu'il est programmeur. Ces deux attributs ont une valeur atomique—nous n'avons rien à ajouter au sujet de son prénom et de sa profession. Quant à ses parents, il ne suffit pas de dire que ce sont Jim et Carol, car nous avons des choses à dire à leur sujet. Les attributs PÈRE et MÈRE ont donc une valeur complexe.

$$\begin{bmatrix} \text{PRÉNOM} & \text{Ethan} \\ \text{PROFESSION} & \text{programmeur} \\ \boxed{1}\ \text{PÈRE} & \boxed{2}\begin{bmatrix} \text{PRÉNOM} & \text{Jim} \\ \text{FEMME} & \boxed{3} \end{bmatrix} \\ \text{MÈRE} & \boxed{3}\begin{bmatrix} \text{PRÉNOM} & \text{Carol} \\ \text{MARI} & \boxed{2} \end{bmatrix} \end{bmatrix}$$

Figure 5 – Ethan, représenté sous forme d'une matrice attribut-valeur

3. Le terme **graphe orienté acyclique** (DAG) est très courant dans la litérature même si la réentrance fait en sorte qu'une structure de traits peut parfois avoir des cycles. Par souci de clarté, nous éviterons ce terme et utiliserons plutôt les termes **graphe orienté** et **graphe**.

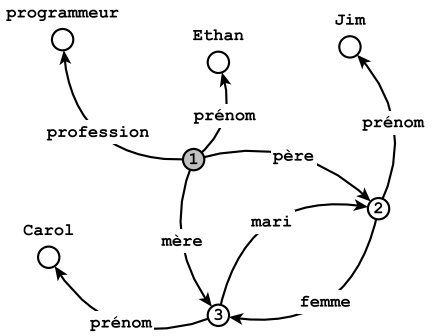


Figure 6 – Ethan, représenté sous forme d'un graphe orienté

La figure 5 contient trois matrices—une matrice principale et deux enchâssées—identifiées par une étiquette. La matrice [1] représente Ethan, tandis que les matrices [2] et [3] représentent respectivement Jim et Carol. Nous avons deux réentrances : ⟨PÈRE FEMME⟩ et ⟨MÈRE⟩ pointent vers la même valeur, et c'est aussi le cas pour ⟨MÈRE MARI⟩ et ⟨PÈRE⟩. Si le père de Ethan et le mari de Carol étaient, par hasard, deux hommes différents qui s'appellent Jim, alors la matrice ressemblerait plutôt à celle de la figure 7.

$$\boxed{1}\begin{bmatrix} \text{PRÉNOM} & \text{Ethan} \\ \text{PROFESSION} & \text{programmeur} \\ \text{PÈRE} & \boxed{2}\begin{bmatrix} \text{PRÉNOM} & \text{Jim} \end{bmatrix} \\ \text{MÈRE} & \boxed{3}\begin{bmatrix} \text{PRÉNOM} & \text{Carol} \\ \text{MARI} & \boxed{4}\begin{bmatrix} \text{PRÉNOM} & \text{Jim} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 7 – Deux invididus identiques mais distincts (Jim [2] et Jim [4])

Le graphe de la figure 6 est équivalent à la matrice de la figure 5. La racine, en gris, représente la structure de traits principale, Ethan, et les nœuds étiquetés [2] et [3] représentent Jim et Carol. La représentation en graphe permet de trouver rapidement les réentrances : les chemins ⟨PÈRE⟩ et ⟨MÈRE MARI⟩, qui partent tous deux de la racine, pointent vers un même nœud, et c'est aussi le cas pour les chemins ⟨MÈRE⟩ et ⟨PÈRE FEMME⟩. Ces deux réentrances sont celles identifiées dans le paragraphe précédent.

Est-il possible de décrire les (nombreux) amis de Ethan ? Nous avons vu qu'un attribut est déterministe—nous ne pouvons donc pas assigner plus d'une valeur à AMI. Une solution

possible est d'utiliser une structure comme contenant. Ce contenant est une **liste** s'il est ordonné et porte alors des attributs comme PREMIER, DEUXIÈME, etc. qui le lient aux différents éléments. Il est aussi possible de représenter une liste à l'aide de seulement deux attributs, PREMIER et RESTE. Une liste est alors représentée à l'aide d'une structure où PREMIER a pour valeur un élément de la liste et RESTE une autre structure enchâssée qui poursuit la liste (Abeillé, 2007, Francez & Wintner, 2012).

$$\begin{bmatrix} \text{PRÉNOM} & \text{Ethan} \\ \text{AMIS} & \begin{bmatrix} 1 & \begin{bmatrix} \text{PRÉNOM} & \text{Kaitlin} \end{bmatrix} \\ 2 & \begin{bmatrix} \text{PRÉNOM} & \text{Casper} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(a) Avec des attributs ordinaux

$$\begin{bmatrix} \text{PRÉNOM} & \text{Ethan} \\ \text{AMIS} & \begin{bmatrix} \text{PREMIER} & \begin{bmatrix} \text{PRÉNOM} & \text{Kaitlin} \end{bmatrix} \\ \text{RESTE} & \begin{bmatrix} \text{PREMIER} & \begin{bmatrix} \text{PRÉNOM} & \text{Casper} \end{bmatrix} \\ \text{RESTE} & \text{vide} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

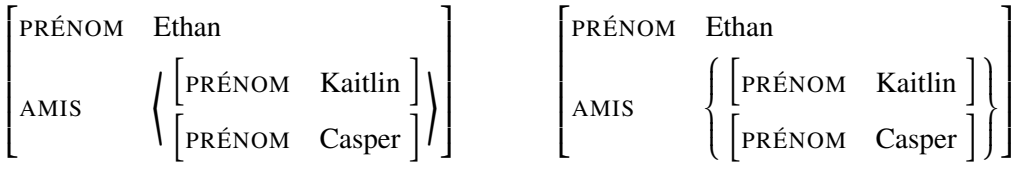
(b) Avec les attributs PREMIER et RESTE

Figure 8 – Deux manières de représenter une liste de valeurs

La seconde méthode (voir la figure 8b) est avantageuse puisqu'elle ne nécessite que deux attributs, alors que la première (8a) demande autant d'attributs ordinaux (1, 2, etc.) que d'éléments dans la liste. Si on considère qu'une grammaire a un nombre fini d'attributs (Francez & Wintner, 2012), la première méthode est également problématique parce qu'il n'existe pas à priori de limite au nombre d'attributs requis.

La figure 9a représente implicitement une liste. Les chevrons $\langle\rangle$ qui entourent les deux matrices enchâssées [4] indiquent que l'attribut AMIS pointe vers une séquence ordonnée de valeurs qui peut contenir une ou plusieurs valeurs ou même être vide. Dans notre exemple actuel, une liste vide indiquerait que l'individu décrit n'a aucun ami.

4. À ne pas confondre avec les chevrons utilisés pour indiquer un chemin.



(a) Liste entre chevrons ⟨⟩ (b) Ensemble entre accolades { }

Figure 9 – Représentations implicites d'une liste et d'un ensemble

Si le contenant n'est pas ordonné, nous avons un **ensemble** plutôt qu'une liste. À notre connaissance, aucune solution n'a été proposée pour représenter explicitement et efficacement un ensemble à l'aide de structures de traits [5]. Nous ne prendrons donc pas en compte les ensembles dans les sections suivantes puisqu'ils posent plusieurs problèmes [6] ; nous nous contentons ici de montrer une représentation implicite d'un ensemble (9b), mais le lecteur intéressé est invité à consulter Abeillé (2007) et Francez & Wintner (2012).

2.1.2 La subsomption et l'extension

La **subsomption** et l'**extension** sont des relations inverses qui ordonnent partiellement [7] des structures. De manière générale, on dit qu'une structure A subsume B (noté A ⊑ B) si B contient toute l'information contenue dans A. Plus spécifiquement, A subsume B et B est l'extension de A (noté B ⊒ A) si et seulement si :

1. Tous les chemins et réentrances qui existent dans A existent aussi dans B :
   Comme les chemins sont absolus, cette condition nous force à évaluer l'extension et la subsomption à partir des structures de traits principales.
2. Pour tous les chemins, la valeur dans A est égale à ou subsume celle dans B :
   Si la valeur dans A est atomique, la valeur dans B est la même. Si elle est complexe, alors elle subsume celle dans B. Cette contrainte n'est pas respectée si une des valeurs est complexe et l'autre atomique.

5. Les ensembles semblent être permis dans la plateforme XLE (Maxwell & Kaplan, 1993), mais nous ignorons comment ils y sont représentés explicitement.

6. Nous aborderons un de ces problèmes lorsque nous décrirons les grammaires d'unification polarisées (cf. § 2.2, p. 15) puisqu'une **structure polarisée** est un ensemble de structures de traits. La combinaison de structures polarisées est donc beaucoup plus complexe que l'unification de structures de traits.

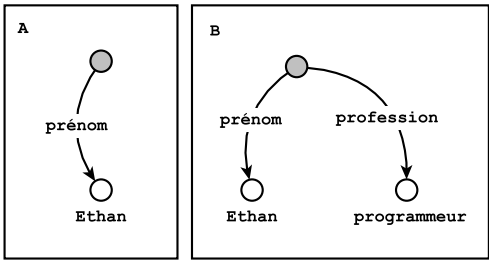


Figure 10 – Deux structures de traits en relation de subsomption

Dans la figure 10, la structure de gauche (A) subsume celle de droite (B). Le seul chemin qui existe dans A—⟨PRÉNOM⟩—existe également dans B, et la valeur est la même dans les deux structures. Le contraire n'est pas vrai ; B ne subsume pas A puisque le chemin ⟨PROFESSION⟩ n'existe pas dans A.

Comme les deux conditions sont satisfaites lorsqu'on compare une structure avec elle-même, on considère qu'une structure se subsume elle-même ($A \sqsubseteq A$) et qu'elle est une extension d'elle-même ($A \sqsupseteq A$). L'extension et la subsomption sont donc **réflexives**. Elles sont aussi **transitives**, c'est-à-dire que si A subsume B qui subsume à son tour C, alors A subsume nécessairement C par transitivité ($A \sqsubseteq B \wedge B \sqsubseteq C \Rightarrow A \sqsubseteq C$).

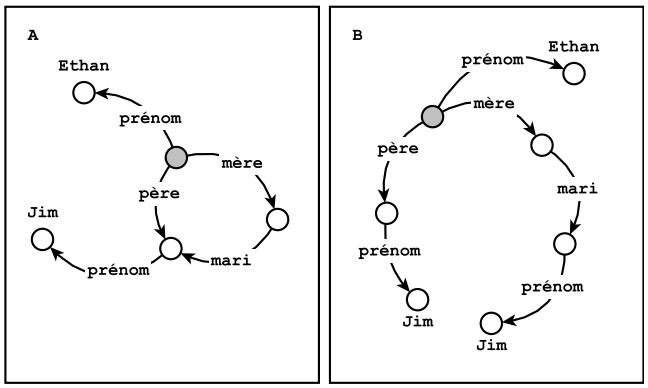


Figure 11 – Rôle de la réentrance dans la subsomption

7. L'extension et la subsomption ne donnent qu'un ordre partiel car elles sont réflexives alors qu'une relation d'ordre strict doit être antiréflexive. Par **réflexivité**, on entend par exemple que toute structure de traits est une extension d'elle-même ($A \sqsupseteq A$) et se subsume elle même ($A \sqsubseteq A$). Ce ne sont également pas toutes les structures qui sont en relation d'extension ou de subsomption.

Il faut tenir compte de la réentrance pour évaluer la subsomption. Dans la figure 11, par exemple, B subsume A puisque tous les chemins dans B sont présents dans A. Le contraire n'est pas vrai—A ne subsume pas B, car ⟨PÈRE⟩ et ⟨MÈRE MARI⟩ sont réentrants dans A mais pas dans B. La seconde condition à elle seule ne nous permet pas de faire cette distinction, puisque les deux chemins pointent vers des valeurs identiques mais distinctes dans B et les deux valeurs subsument donc celle dans A.

La subsomption et l'extension sont deux concepts importants dans les grammaires d'unification, puisque l'unification vise essentiellement à trouver une structure de traits « minimale » qui est subsumée par les deux structures originales (voir la section suivante).

2.1.3 L'unification

L'**unification** est l'opération de base dans les grammaires d'unification—d'où le nom—et consiste à combiner deux structures à partir de leurs racines afin d'obtenir une structure « **minimale** » qui est subsumée par les deux structures originales. Elle peut aussi être vue comme la consolidation de l'information des deux structures. Si les structures contiennent des traits contradictoires, la structure résultante est malformée et l'unification échoue. Par exemple, l'unification échoue si on combine des structures avec les valeurs 24 et 27 pour un attribut ÂGE, puisque les attributs sont déterministes.

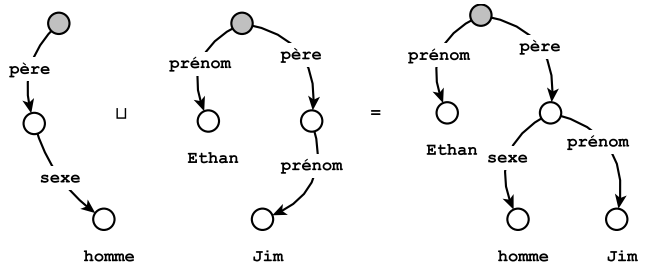


Figure 12 – Unification de deux structures de traits simples

La première structure de la figure 12 encode une généralisation, soit le fait qu'un père est un homme. Son unification avec la seconde structure donne une structure qui indique

que Jim, le père de Ethan, est lui aussi un homme. Lors du développement d'une grammaire, nous pouvons éviter de spécifier, pour chaque structure principale qui possède un attribut PÈRE, que la valeur complexe de cet attribut a un attribut SEXE avec la valeur *homme*, puisque cette information est prévisible et peut être ajoutée systématiquement par unification. Ce mécanisme s'appelle **sous-spécification** [8].

L'unification débute en superposant (en **unifiant**) les racines des deux structures. Chaque paire d'arcs sortants avec une même étiquette (qui représentent le même attribut) sont superposés à leur tour, et leurs cibles le sont également puisque les attributs sont déterministes. Pour superposer deux nœuds terminaux, ceux-ci doivent avoir des étiquettes compatibles—il y a ici trois scénarios possibles où les étiquettes sont compatibles :

— Les deux nœuds portent une étiquette et ces étiquettes sont identiques ;

— Aucun des nœuds ne porte d'étiquette ;

— Un seul nœud porte une étiquette.

Si les nœuds ont des étiquettes différentes, ils ne peuvent être superposés et l'unification échoue. Quant aux nœuds non terminaux, ils sont unifiés récursivement [9]. Nous traversons ainsi les deux structures et obtenons une nouvelle structure contenant toute l'information des structures originales. Si on retrouve plusieurs arcs pour un même attribut, la structure est malformée et l'unification est un échec, sinon l'unification est réussie.

À titre d'exemple, prenons la figure 12. En superposant les racines, nous obtenons deux arcs pour l'attribut PÈRE et superposons donc ces deux arcs et leurs cibles. Puisque les nœuds cibles ne partagent pas d'attribut, l'unification se termine avec succès.

8. Dans les grammaires d'unification polarisées, nous utilisons ce type de mécanisme pour « forcer » la présence de certaines informations en ajoutant des objets non saturés à une structure. Une propriété des structures polarisées nous permet également de simplifier ce mécanisme (cf. § 2.2.3, p. 21).

9. Chaque nœud est la racine d'un **sous-graphe**, soit l'ensemble des nœuds accessibles à partir du nœud. L'unification récursive de nœuds non terminaux consiste à unifier leurs sous-graphes, au même titre que l'unification des racines des graphes entiers.

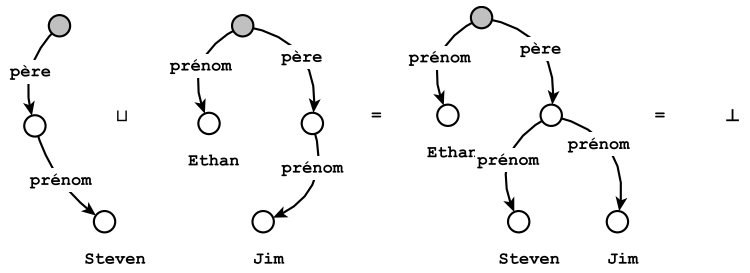


Figure 13 – Unification de deux structures de traits simples qui échoue

La figure 13 montre une unification qui échoue. Après avoir superposé les deux racines, les arcs qui représentent l'attribut PÈRE et les nœuds cibles, nous obtenons deux arcs PRÉNOM dont les cibles sont étiquetées différemment (*Jim* et *Steven*). Comme les deux nœuds cibles ne peuvent être superposés, le nœud résultant a deux arcs sortants pour un même attribut et l'unification échoue.

L'unification « traditionnelle » a une lacune importante—elle débute toujours aux racines des structures, ce qui nous empêche d'utiliser la sous-spécification à son plein potentiel. Nous ne pouvons pas, entre autres, utiliser la structure de gauche dans la figure 12 pour spécifier que le père de Jim est un homme, puisque le nœud qui représente Jim n'est pas la racine de la structure ; il nous faudrait plutôt la structure de la figure 14.

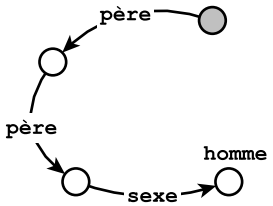


Figure 14 – Encodage d'une généralisation—un grand-père est un homme

Cette structure est encore une fois problématique car elle ne permet pas de spécifier le sexe d'un père à un niveau d'enchâssement arbitraire—par exemple, l'arrière-grand-père de Jim, autrement dit, le père du père du père du père de Ethan. Nous verrons sous peu que les grammaires d'unification polarisées viennent résoudre ce problème en relâchant la contrainte sur la minimalité de la structure résultante (cf. § 2.2.3, p. 21).

Formellement, l'unification non polarisée a trois propriétés importantes :

— Elle est **idempotente** : lorsqu'une structure est unifiée à elle-même, la structure résultante est identique à l'originale (A ⊔ A = A) puisqu'une structure de traits est toujours la plus petite extension d'elle-même ;

— Elle est **commutative** : comme l'addition et la multiplication en arithmétique, l'unification est symétrique, c'est-à-dire que l'ordre des opérandes (structures de traits) n'a pas d'impact sur le résultat final (A ⊔ B = B ⊔ A) ;

— Elle est **associative** : si on unifie plus de deux structures, l'ordre des unifications individuelles n'influence pas le résultat final : A ⊔ (B ⊔ C) = (A ⊔ B) ⊔ C.

L'unification est aussi **monotone**—elle peut ajouter ou non de l'information, mais jamais en ôter. Cela signifie notamment que si une structure A subsume une structure B, l'unification de A et B donne une structure identique à B (A ⊑ B, A ⊔ B = B). Cela veut également dire que les relations de subsomption sont conservées par l'unification, c'est-à-dire que si A ⊑ B, ∀ C, (A ⊔ C) ⊑ (B ⊔ C) (Abeillé, 2007, p. 41).

C'est ici que se termine notre présentation des grammaires d'unification « traditionnelles ». Dans la section suivante, nous décrirons une extension non stricte des grammaires d'unification, les grammaires d'unification polarisées. Nous verrons que certaines propriétés du formalisme rendent le processus d'unification beaucoup plus complexe.

2.2 Les grammaires d'unification polarisées

Les grammaires d'unification polarisées (GUP) sont une extension non stricte des grammaires d'unification traditionnelles, c'est-à-dire qu'elles n'adhèrent qu'en partie à notre description des grammaires d'unification traditionnelles. Elles conçoivent notamment l'unification d'une manière un peu différente.

La description qui suit est basée principalement sur les textes de Kahane (2004, 2006), Kahane & Lareau (2005a, 2005b) et Lareau (2008).

2.2.1 Les structures polarisées

Les grammaires d'unification polarisées manipulent des **structures polarisées**, soit des ensembles d'objets liés entre eux par des fonctions. Nous utiliserons dorénavant le terme **structure** pour référer à une structure polarisée et le terme **structure de traits** pour parler d'une structure propre aux grammaires d'unification traditionnelles.

Un **objet** correspond à peu près à une structure de traits telle que définie précédemment et porte des paires fonction-valeur. Une **fonction**, comme un attribut, assigne une valeur à certains objets de manière partielle et déterministe. Le type de cette valeur dépend du type de fonction. On distingue trois types de fonction :

— Les **fonctions structurantes** ont pour valeur un objet. Ce sont elles qui lient les objets d'une structure ;

— Les **fonctions d'étiquetage** ont pour valeur une valeur atomique ;

— Les **fonctions de polarisation** sont propres aux grammaires d'unification polarisées et ont pour valeur une polarité (cf. § 2.2.2, p. 19). Un même objet peut porter plus d'une polarité (à travers plusieurs fonctions de polarisation).

Alors que dans les grammaires d'unification traditionnelles, un même attribut peut avoir une valeur simple dans une structure de traits et complexe dans une autre, une fonction n'accepte qu'un type de valeur.

Chaque objet possède un **type**. Si une structure représente un arbre, par exemple, nous retrouvons des objets de type *nœud* et *arc*. Comme les structures polarisées représentent typiquement des graphes orientés dans les exemples de Kahane (2004), nous utiliserons également des graphes orientés à titre d'exemple, mais nous utiliserons une ligne pleine pour les objets de type *arc* et une ligne pointillée pour les fonctions, afin de distinguer un graphe orienté de la structure polarisée qui le représente.

Certaines grammaires d'unification sont **typées**. Dans de tels formalismes, les types de structure de traits peuvent être ordonnés en fonction de leur spécificité—autrement dit,

selon une relation de subsomption. Lorsque nous unifions deux structures de traits, cette hiérarchie de types nous permet d'inférer le type de la structure de traits résultante, soit le type le plus général qui est subsumé par les types des structures de traits originales. Si un tel type n'existe pas, l'unification échoue. Le typage permet également de forcer ou interdire la présence de certains attributs et de contraindre le type d'une valeur.

Ce typage n'est pas strictement nécessaire dans les GUP et nous verrons plutôt le type d'un objet comme une simple étiquette (Lareau, 2008, p. 220). Le lecteur intéressé par les grammaires typées peut cependant lire Carpenter (1992) et Copestake (2002).

$$\left\{ \begin{array}{ll} \boxed{1} & \left[\begin{array}{ll} \text{TYPE} & \text{nœud} \\ \text{ÉTIQUETTE} & \text{dormir} \\ \text{POLARITÉ} & \text{noir} \blacksquare \end{array} \right] \\ \boxed{2} & \left[\begin{array}{ll} \text{TYPE} & \text{arc} \\ \text{ÉTIQUETTE} & 1 \\ \text{POLARITÉ} & \text{noir} \blacksquare \\ \text{SOURCE} & \boxed{1} \\ \text{CIBLE} & \boxed{3} \end{array} \right] \\ \boxed{3} & \left[\begin{array}{ll} \text{TYPE} & \text{nœud} \\ \text{ÉTIQUETTE} & \text{chien} \\ \text{POLARITÉ} & \text{noir} \blacksquare \end{array} \right] \end{array} \right\}$$

Figure 15 – Une structure polarisée représentée sous forme de matrice

Textuellement, une structure peut être représentée par un ensemble de matrices attribut-valeur entre accolades où chaque matrice correspond à un objet. Les objets peuvent être étiquetés, en particulier lorsqu'ils sont la cible d'une fonction structurante. La figure 15 représente par exemple un graphe orienté qui contient deux nœuds et un arc, tous polarisés en noir. L'objet [2] représente un arc avec une source ([1]) et une cible ([3]).

Graphiquement, une structure peut être représentée à l'aide d'un graphe orienté où chaque nœud non terminal correspond à un objet. La figure 16 montre la structure polarisée de la figure 15 sous forme de graphe orienté. Ici, les nœuds gris représentent des objets et

chaque objet est étiqueté.

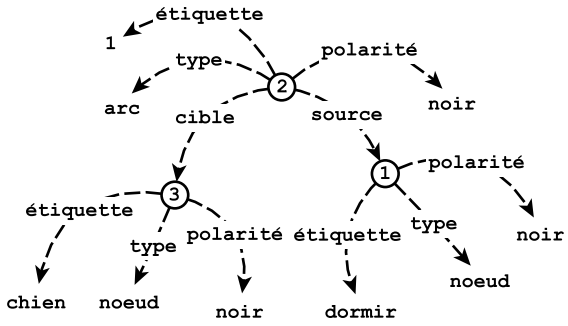


Figure 16 – Représentation explicite d'une structure polarisée

Notons que l'objet 2, même s'il représente un arc, n'est rien d'autre qu'un objet dans une structure polarisée, et donc un nœud dans sa représentation explicite. Le type d'un objet n'est qu'une étiquette après tout ; il n'influence donc pas la forme que prend cet objet dans une représentation explicite.

$$\left\{ \begin{array}{ll} \boxed{1} & \left[\begin{array}{ll} \text{PRÉNOM} & \text{Ethan} \\ \text{PROFESSION} & \text{programmeur} \\ \text{PÈRE} & \boxed{2} \\ \text{MÈRE} & \boxed{3} \end{array} \right] \\ \boxed{2} & \left[\begin{array}{ll} \text{PRÉNOM} & \text{Jim} \\ \text{FEMME} & \boxed{3} \end{array} \right] \\ \boxed{3} & \left[\begin{array}{ll} \text{PRÉNOM} & \text{Carol} \\ \text{MARI} & \boxed{2} \end{array} \right] \end{array} \right\}$$

Figure 17 – La structure de traits de la figure 5 convertie en structure plate

Pour illustrer la différence entre la configuration d'une structure de traits et celle d'une structure polarisée, prenons la structure plate de la figure 17—celle-ci est équivalente à la structure de traits de la figure 5 (p. 5). Alors que la structure de traits a une structure principale et deux enchâssées, les objets de la structure plate sont tous au même niveau

d'enchâssement—il n'y a pas d'objet « principal » [10].

2.2.2 Le système de polarités

Chaque grammaire d'unification polarisée est dotée d'un **système de polarités** P, soit un ensemble de **polarités** muni d'une opération binaire qu'on appelle le **produit**. Les systèmes de polarités correspondent plus ou moins aux magmas en algèbre ; nous utiliserons donc quelques termes propres aux mathématiques pour expliquer certaines propriétés des systèmes de polarités, mais tâcherons de définir les termes au préalable.

Chaque système est arbitraire. Le système de polarités « classique » n'a que deux polarités, **blanc** □ et **noir** ■. Les deux appartiennent à l'ensemble des polarités P et la polarité noire appartient aussi à un sous-ensemble de P, N, qui comprend les polarités **neutres** [11]. N est un **sous-ensemble strict** puisque P doit avoir au moins une polarité non neutre. N doit aussi être non vide. Une structure est **neutre** ou **saturée** quand les polarités portées par ses objets sont toutes neutres. Si la grammaire est bien conçue, toutes les structures neutres sont bien formées et toutes les structures bien formées sont neutres. La grammaire d'unification Sens-Texte vise par exemple à ne générer que des structures neutres qui sont linguistiquement correctes (cf. § 2.3, p. 23).

Un système de polarités bien conçu respecte quatre contraintes (Lareau, 2008) :

— **Dynamisme** : Un système est dynamique lorsqu'il possède au moins une polarité neutre et une polarité non neutre. Sans polarité neutre, une structure polarisée ne peut jamais atteindre la saturation.

— **Monotonicité** : Un système est monotone s'il est possible d'ordonner ses polarités de sorte que le produit de polarités donne toujours une polarité dans une position égale ou supérieure à celle de la plus « haute » des deux.

10. Comme une structure a typiquement plus d'un objet, il est nécessaire de spécifier l'objet de départ d'un chemin. Par exemple, les chemins ⟨ 1 TYPE⟩ et ⟨ 2 TYPE⟩ correspondent respectivement aux valeurs *nœudsém* et *arcsém* dans la structure de la figure 16.

11. Une polarité neutre n'est pas nécessairement un élément neutre au sens mathématique du terme mais peut l'être. La polarité grise ■ décrite dans Kahane (2004), par exemple, appartient à N et son produit avec une polarité A donne toujours A ($\forall A \in P, A . ■ = A$).

— **Finalité** : Un système est final lorsqu'il est monotone et que les polarités neutres occupent les plus hautes positions. La contrainte de finalité assure qu'un produit ne peut que laisser une polarité intacte ou lui faire approcher la saturation.

— **Déclarativité** : Un système est déclaratif si son produit est commutatif et associatif (cf. § 2.1.3, p. 15). Cette contrainte permet la combinaison de structures polarisées dans n'importe quel ordre sans changer les structures résultantes [12].

Le **produit** (noté « . ») [13] est une loi de composition interne sur un système de polarités. Par **loi de composition interne**, on entend une opération binaire qui prend deux éléments de P et leur associe un autre élément de P.

On représente un produit avec une table de Cayley. Le tableau 1 indique par exemple que □ et □ donne □, que □ et ■ (ou ■ et □ car le système est commutatif) donne ■ et que ■ et ■ donne un échec (■ . ■ = ⊥), c'est-à-dire que ■ ne peut être combiné avec lui-même. Rappelons que ce système de polarités et son produit sont arbitraires.

·	□	■
□	□	■
■	■	⊥

Tableau 1 – Produit d'un système de polarités classique

Cette table permet également de voir si le produit est commutatif. Le tableau 1, par exemple, est symétrique sur son axe diagonal (du coin supérieur gauche au coin inférieur droit) et le produit est donc commutatif. Nous décrirons une heuristique pour vérifier l'associativité du produit lorsque nous discuterons de notre outil.

Nous verrons dans la section suivante que le produit guide la combinaison des structures puisqu'on le calcule chaque fois que des objets polarisés sont unifiés, et que l'échec d'un

12. Cohen-Sygal & Wintner (2007) soutiennent qu'aucun système non trivial ne peut être associatif. Cela suggère que l'unification ne peut pas non plus être associative. Alors qu'on exige typiquement qu'au moins une paire d'objets soient unifiés dans les grammaires d'unification polarisées, relâcher cette contrainte permet de rendre l'unification associative.

produit fait également échouer l'unification entière.

2.2.3 La combinaison de structures polarisées

La **combinaison** de deux structures consiste à unifier une partie de leurs objets. Il ne faut pas confondre la **combinaison**, qui s'effectue au niveau des structures, avec l'**unification**, qui s'effectue au niveau des objets contenus dans les structures. Nous décrirons l'unification polarisée en premier puisqu'elle sous-tend le processus de combinaison.

Pour unifier deux objets, nous devons d'abord comparer les valeurs des fonctions partagées par les deux objets. Comme nous savons d'avance le type de valeur d'une fonction donnée, il est inutile de vérifier que le type est le même comme dans une unification traditionnelle. Les valeurs sont unifiées différemment selon le type de fonction :

— **Fonction d'étiquetage** : si les deux étiquettes sont identiques, l'objet résultant porte la même étiquette, sinon l'unification échoue ;

— **Fonction structurante** : si on réussit à unifier les deux valeurs de la fonction, la fonction pointe vers l'objet résultant, sinon l'unification échoue ;

— **Fonction de polarisation** : s'il existe un produit valide pour les deux polarités, l'objet résultant porte ce produit, sinon l'unification échoue.

Les valeurs de fonctions non partagées sont ensuite ajoutées au nouvel objet. Aucune unification n'est effectuée lors de cette étape—même si l'unification de la paire n'est pas encore complétée, sa réussite est essentiellement garantie. Une fois les valeurs copiées, l'unification est complétée.

Lorsque l'unification d'une paire d'objets échoue, la combinaison entière échoue aussi.

Nous avons vu que l'unification traditionnelle débute toujours aux racines des structures de traits. Comme une structure polarisée n'a pas d'objet principal, nous unifions plutôt

13. Le symbole « + » est aussi parfois utilisé pour noter une loi de composition. Comme les polarités peuvent être représentées comme des vecteurs binaires et le produit calculé en additionnant ces vecteurs, il serait peut-être plus approprié d'appeler « somme » la loi de composition de P. Pour une discussion de la décomposition des polarités en vecteurs binaires, voir l'annexe B (p. xv).

des paires d'objets et exigeons seulement qu'au moins une paire d'objets soit unifiée. On peut ainsi unifier une seule paire d'objets pour obtenir une structure à $\bar{\bar{A}} + \bar{\bar{B}} - 1$ objets où $\bar{\bar{A}}$ et $\bar{\bar{B}}$ sont le nombre d'objets dans les deux structures, mais tout aussi bien unifier tous les objets pour obtenir une structure à une seul objet si le contenu des objets le permet.

Cela fait décupler le nombre de combinaisons possibles pour une même paire de structures, contrairement aux grammaires d'unification traditionnelles où il n'y a que deux résultats possibles : une structure ou un échec. Pour illustrer ceci, prenons deux structures simples à deux objets chacune, où f et g sont des fonctions structurantes :

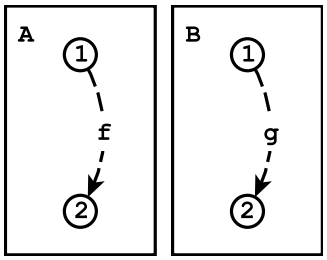


Figure 18 – Deux structures à deux objets chacune

Si nous unifions une seule paire (le strict minimum), il existe jusqu'à quatre structures résultantes, données par la figure 19. Même si certaines combinaisons peuvent échouer ou donner des structures identiques, elles sont toutes envisageables à priori, sans connaître le contenu des objets. Le nombre de combinaisons envisageables en n'unifiant qu'une seule paire d'objets est égal au produit du nombre d'objets dans les deux structures (ici, $2 \times 2 = 4$), soit la taille du produit cartésien des deux ensembles d'objets.

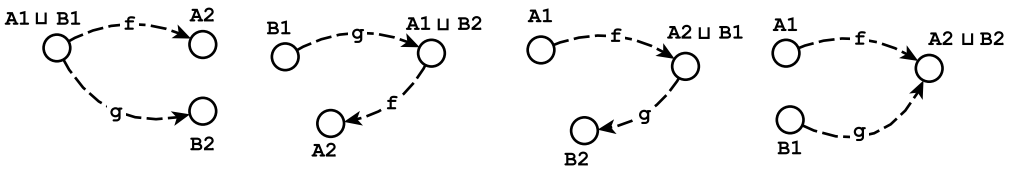


Figure 19 – Unification d'une seule paire d'objets

Nous ne voulons cependant pas nous limiter à une paire d'objets ; nous considérons les combinaisons d'une paire, mais aussi celles de deux paires, trois paires, etc. Au total,

il existe jusqu'à $2^n - 1$ façons de combiner les paires, où n est le nombre de paires, soit $2^4 - 1 = 15$ pour des structures à deux objets chacune. Nous en retirons une car nous ne considérons pas la combinaison vide (0 paires d'objets) [14].

Nous discuterons plus en détail des mathématiques derrière la combinaison des structures lorsque nous présenterons notre implémentation. Nous décrirons aussi diverses heuristiques employées pour réduire le nombre de combinaisons considérées et différentes techniques pour limiter l'impact des unifications qui échouent.

2.3 La grammaire d'unification Sens-Texte

La grammaire d'unification Sens-Texte est une grammaire d'unification polarisée de nature linguistique et basée sur la théorie Sens-Texte de Mel'čuk. Elle servira ici à tester notre implémentation de GUP, et en retour notre outil permettra d'évaluer la grammaire GUST, principalement d'un point de vue computationnel.

Puisque GUST ne servira qu'à des fins d'évaluation, nous ne ferons qu'une description de haut niveau de ses fondements théoriques et nous concentrerons sur une sous-partie du modèle. Cette description est basée sur Mel'čuk (1988, 1997), Polguère (1998, 2003), Kahane (2001, 2002), Kahane & Lareau (2005a, 2005b) et Lareau (2008).

2.3.1 Un survol de la théorie Sens-Texte

La **théorie Sens-Texte** (TST) remonte aux travaux de Žolkovskij et Mel'čuk à Moscou dans les années 1960 et vise à formaliser la correspondance entre des sens et des textes via sept niveaux de représentation contigus. Par **sens**, nous entendons toute information qui peut être exprimée par le biais du langage, et un **texte** est la réalisation physique d'un sens, que cette réalisation soit écrite ou orale (Mel'čuk, 1988, p. 44).

14. L'inclusion de la combinaison vide permettrait cependant de résoudre le problème d'associativité évoqué par Cohen-Sygal & Wintner (2007) et mentionné dans la note de bas de page 12 (p. 20). Comme la structure résultante est alors non connexe, l'unification peut se comporter de façon imprévue et nous préférons nous en tenir à la théorie telle que présentée dans les textes de Kahane et Lareau.

La TST compte sept **niveaux de représentation**, soit un niveau sémantique ainsi que deux niveaux—un niveau profond et un autre de surface—pour chaque type de représentation subséquent, soit la syntaxe, la morphologie et la phonologie. Les niveaux profonds servent d'interface entre les autres niveaux.

Au moment où nous écrivons ce mémoire, seuls les niveaux sémantique, syntaxique et textuel, ainsi que les interfaces entre eux, sont détaillés dans la formulation actuelle de GUST. Nous ne décrirons cependant ici que trois niveaux de la TST, soit les niveaux sémantique, syntaxique profond et syntaxique de surface, puisque le fragment de grammaire GUST qui servira à évaluer notre outil ne se rend qu'au niveau syntaxique.

2.3.1.1 La représentation sémantique

La représentation sémantique (RSém) est le point de départ de la théorie Sens-Texte pour la synthèse (Sens ⇒ Texte) [15]. On représente un sens avec un **réseau sémantique**, soit un ensemble de nœuds qui représentent des prédicats et des objets sémantiques, ainsi que d'arcs qui représentent des relations prédicatives entre les sémantèmes. Un **prédicat sémantique** est un concept qui implique d'autres participants (le sens ⟨manger⟩ implique par exemple un individu et un aliment) tandis qu'un **nom sémantique** dénote une entité et n'implique aucun autre participant (par exemple, le sens ⟨pomme⟩) [16].

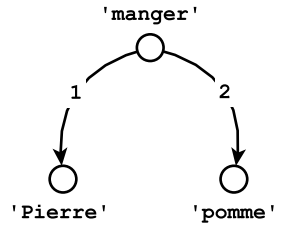


Figure 20 – Un réseau sémantique

Le réseau sémantique de la figure 20 a trois nœuds. ⟨manger⟩ est prédicatif et son nœud a

15. La représentation sémantique est particulière puisqu'elle n'est pas produite à partir d'une autre représentation propre à la TST, en synthèse du moins. Mel'čuk parle parfois d'une **représentation conceptuelle**, basée sur le domaine extralinguistique, mais n'élabore pas sur cette idée.

16. On utilise aussi le terme **objet sémantique** pour parler d'un nom sémantique.

donc des arcs sortants qui pointent vers ses arguments, ⟨Pierre⟩ et ⟨pomme⟩. L'étiquette d'un arc ne reflète pas nécessairement la position de l'argument en syntaxe ou dans la phrase ; on ordonne simplement les arguments selon leur saillance sémantique.

Il faut également noter que les étiquettes portées par les nœuds dans un réseau sémantique ne correspondent pas nécessairement aux mots qu'on retrouvera dans le texte final. C'est là un des intérêts de la TST—elle permet de générer pour un même sens tous les textes qui l'expriment. Le paraphrasage est rendu possible en grande partie par un « dictionnaire Sens-Texte » dont nous ne discutons pas ici, mais le lecteur est invité à consulter, entre autres, Mel'čuk (1997) et Polguère (1998). Pour une présentation des dictionnaires en GUST, voir également Kahane & Lareau (2016a).

2.3.1.2 La représentation syntaxique profonde

La représentation syntaxique profonde (RSyntP) est un **arbre de dépendance** non linéairement ordonné où les nœuds représentent des unités lexicales pleines et des fonctions lexicales, et les arcs des liens de dépendance syntaxique. Elle est obtenue en **arborisant** un réseau sémantique et on dit qu'elle est **non linéairement ordonnée** puisque l'ordre des nœuds dans sa forme graphique ne reflète pas l'ordre des mots en surface.

La correspondance entre un réseau sémantique et l'arbre syntaxique équivalent n'est pas biunivoque, c'est-à-dire qu'un nœud peut avoir un ou plusieurs nœuds correspondants, ou même ne pas en avoir.

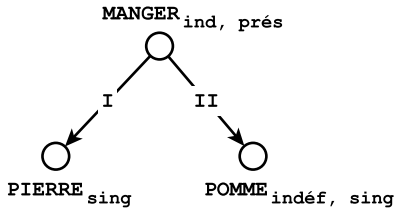


Figure 21 – Une représentation syntaxique profonde

La figure 21 est un exemple d'arbre de dépendance. Les trois nœuds MANGER, PIERRE et POMME correspondent respectivement aux nœuds ⟨manger⟩, ⟨Pierre⟩ et ⟨pomme⟩ qu'on

retrouve dans le réseau sémantique de la figure 20.

La structure syntaxique n'est pas complète—il manque notamment le déterminant associé à la lexie POMME. Cette information se retrouvera plutôt dans le niveau de représentation suivant, soit la représentation syntaxique de surface (RSyntS).

2.3.1.3 La représentation syntaxique de surface

La représentation syntaxique de surface (RSynP) est encore une fois un arbre de dépendance non ordonné et comprend toutes les lexies qu'on retrouvera dans le texte final—les prépositions, déterminants, etc. sont tous inclus sous une forme non fléchie.

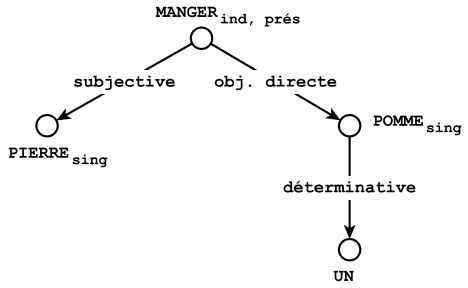


Figure 22 – Une représentation syntaxique de surface

Elle se distingue de la représentation syntaxique profonde par la présence de relations syntaxiques concrètes spécifiques à la langue modélisée comme les relations subjective et objective directe pour le français. Iordanskaja & Mel'čuk (2000) font une description formelle des relations syntaxiques de surface et un inventaire des relations de dépendance liées à la valence du verbe.

Cette représentation est prête à être linéarisée lors de la transition vers la représentation morphologique profonde, dont nous ne discuterons pas ici. Le lecteur intéressé est invité à consulter Kahane & Lareau (2016b).

2.3.2 L'application du formalisme GUP à la théorie Sens-Texte

Les grammaires d'unification polarisées ont été développées par Kahane spécifiquement pour formaliser la théorie Sens-Texte et ont été introduites dans Kahane (2004).

L'ajout d'une procédure de polarisation aux représentations de la TST permet de contrôler leur construction et de vérifier facilement leur bonne formation. Il existe par exemple une contrainte sur les arbres de dépendance qui oblige les nœuds à être la cible d'un seul arc, et donc de n'avoir qu'un seul **gouverneur**, sauf la racine qui ne doit avoir aucun gouverneur—cette contrainte est facilement vérifiée à l'aide de la grammaire d'arbres présentée dans Lareau (2008) [17] et une structure syntaxique qui ne respecte pas la contrainte ne sera donc jamais valide dans le formalisme GUST.

Tandis que la théorie Sens-Texte classique a sept niveaux de représentation, GUST n'en a que quatre : sémantique, syntaxique, morphotopologique et phonologique. Deux d'entre eux, les niveaux sémantique et syntaxique, sont décrits en détail dans la thèse de Lareau (2008), qui fait la synthèse de plusieurs articles de Kahane & Lareau. Puisque le fragment de grammaire du français que nous utiliserons pour évaluer notre outil exclut le niveau morphotopologique (cf. ch. 4, p. 68), nous nous contenterons ici également de décrire les niveaux sémantique et syntaxique et l'interface sémantique-syntaxe.

2.3.2.1 Le système de polarités de GUST

Le système de polarités propre à GUST n'a que deux polarités, soit les polarités **blanc** □ et **noir** ■ (P = {□, ■}). La polarité blanche est non neutre, tandis que la noire est neutre et appartient donc à N. Le système respecte la contrainte de **dynamisme** puisque N est un sous-ensemble strict de P (N ⊂ P), c'est-à-dire que les éléments de N sont tous présents dans P et qu'on retrouve au moins un élément dans P qui n'est pas dans N.

D'un point de vue linguistique, une polarité blanche indique qu'il manque de l'information linguistique à la structure. Dans une structure sémantique, par exemple, elle nous permet

17. La grammaire d'arbres a d'abord été présentée par Kahane (2004), qui s'est inspiré de Nasr (1995).

d'indiquer qu'un argument d'un prédicat est manquant. Lorsque l'argument est finalement instancié, l'objet est polarisé en noir pour indiquer que le besoin est comblé. Une structure est donc incomplète (elle n'est pas neutre) tant qu'elle porte une polarité blanche.

·	□	■
□	□	■
■	■	⊥

Tableau 2 – Le produit des polarités de GUST

Le tableau 2 montre le produit des polarités de GUST. Ce produit nous permet de vérifier que le système respecte la contrainte de **monotonicité**, c'est-à-dire qu'on peut ordonner P de manière à ce que le produit de deux polarités donne toujours une polarité égale à ou plus haute que la plus haute des deux—l'ordre en question est □ < ■. La contrainte de **finalité** est aussi respectée puisque le système est monotone et que la polarité qui occupe le plus haut rang, la polarité noire, appartient à N (elle est neutre).

Commutativité
□ . ■ = ■ . □ = ■

Associativité
(□ . ■) . □ = □ . (■ . □) = ■
(□ . □) . ■ = □ . (□ . ■) = ■
(□ . □) . □ = □ . (□ . □) = □
(■ . ■) . □ = ■ . (■ . □) = ⊥

Tableau 3 – La déclarativité du système de polarités de GUST

Le tableau 3 montre que le système respecte la contrainte de **déclarativité** puisque son produit est commutatif et associatif. Le système respecte donc les quatre contraintes de bonne formation et se prête ainsi à la description des langues (Lareau, 2008, p. 213).

2.3.2.2 L'articulation des modules de la grammaire

Les modules que nous présentons dans les sections suivantes sont **articulés**, c'est-à-dire qu'une règle porte des polarités associées aux modules adjacents. Par exemple, les objets des règles sémantiques portent une polarité d'interface blanche. En ajoutant ces polarités blanches, nous forçons l'application des règles d'interface pour chercher à les saturer. Les règles d'interface, à leur tour, portent des polarités syntaxiques blanches pour déclencher l'application des règles syntaxiques, et ainsi de suite.

Dans les sections suivantes, les polarités d'articulation sont implicites, c'est-à-dire qu'elles ne sont pas inclues dans nos structures, mais elles sont explicitées dans le fragment de grammaire utilisé pour tester notre implémentation du formalisme au chapitre 4.

L'articulation et son fonctionnement sont décrits plus en détail dans Kahane & Lareau (2005a, 2005b) et Lareau (2008).

2.3.2.3 La grammaire sémantique

Comme dans un modèle Sens-Texte traditionnel, la **grammaire sémantique** ($\mathcal{G}_{\text{sém}}$) est le point de départ de la transition Sens ⇒ Texte. Une structure polarisée représente ici un réseau sémantique et comporte deux types d'objets, des nœuds et des arcs sémantiques. Rappelons qu'il faut distinguer la structure polarisée de la structure qu'elle représente—dans une règle de $\mathcal{G}_{\text{sém}}$, les arcs représentent des fonctions et les objets portent une fonction TYPE qui a comme valeur *nœudsém* ou *arcsém*.

La figure 23 montre la différence entre les deux. La sous-figure de gauche a deux nœuds et un arc. La structure polarisée de la sous-figure de droite est équivalente—elle comprend deux objets de type *nœudsém* et un de type *arcsém*. Comme le réseau sémantique de gauche provient de la théorie Sens-Texte et n'est donc pas polarisé, la structure de droite ne comprend également aucune fonction de polarisation dans cet exemple. Dans une représentation sémantique typique, les objets doivent cependant porter une polarité sémantique, par le biais de la fonction de polarisation PSÉM.

Par souci de concision, nous utiliserons dorénavant une notation plus implicite dans nos règles de GUST. Dans cette notation, chaque objet de type *nœudsém* est représenté par un nœud, et chaque objet de type *arcsém* est représenté par un arc qui a comme source la valeur de la fonction SOURCE et comme cible, celle de la fonction CIBLE. La couleur d'un objet correspond à sa polarité sémantique.

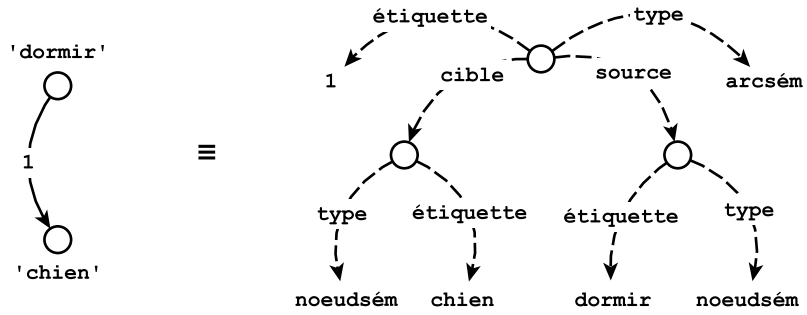


Figure 23 – Un réseau sémantique et la structure plate équivalente

La polarisation d'un réseau sémantique permet de vérifier que chaque sémantème prédicatif a le bon nombre d'arguments. Dans une règle de $\mathcal{G}_{\text{sém}}$, un sémantème prédicatif porte généralement la polarité noire et ses arguments la polarité blanche ; de cette manière, nous forçons l'instanciation—et donc la saturation—des arguments pour assurer la bonne formation de la structure. Rappelons-nous qu'une structure n'est bien formée que si elle est saturée, c'est-à-dire que tous ses objets portent une polarité neutre.

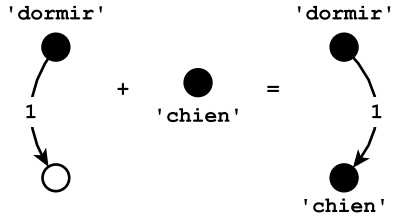


Figure 24 – L'instanciation du premier argument de ⟨dormir⟩

Dans la figure 24, par exemple, la règle de gauche correspond au sémantème ⟨dormir⟩ qui demande un argument quelconque. La règle instancie le sémantème ⟨dormir⟩ qui est donc polarisé en noir, tandis que son premier argument, qui n'est pas encore instancié, porte la

polarité blanche. Lorsque nous combinons cette règle avec celle du milieu, l'argument est instancié par le sémantème ⟨chien⟩ et le nœud résultant est alors saturé. La structure finale, comme tous ses objets portent la polarité noire, est saturée et donc bien formée.

Pour une description plus complète de la grammaire sémantique et un tableau récapitulatif des types d'objets et fonctions disponibles, voir Lareau (2008).

2.3.2.4 La grammaire syntaxique

Les structures de la grammaire syntaxique ($\mathcal{G}_{\text{synt}}$) représentent des arbres de dépendance non linéairement ordonnés. Les nœuds et les arcs correspondent respectivement aux valeurs *nœudsynt* et *arcsynt* de la fonction TYPE. Il existe un troisième type d'objet, les grammèmes, qui portent la valeur *grammème*. Un grammème fait la correspondance entre un sens flexionnel et sa réalisation—par exemple, le grammème de nombre prend comme valeur le singulier ou le pluriel, et cette valeur est plus tard réalisée morphologiquement (par l'ajout d'un suffixe *-s* à un nom, par exemple). Nous ne décrirons pas les grammèmes en détail ici, mais invitons le lecteur à consulter Mel'čuk (1993) et Lareau (2011).

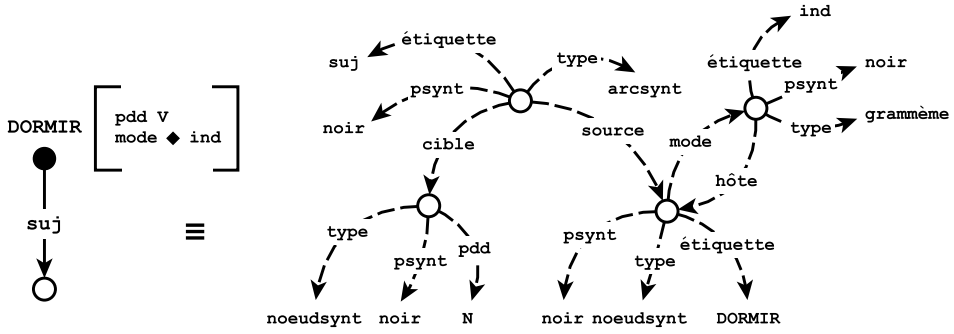


Figure 25 – Les versions implicite et explicite d'un arbre de dépendance

Dans une structure syntaxique, les nœuds représentent donc des lexèmes et les arcs des relations syntaxiques—plus précisément, des relations syntaxiques de surface telles que mentionnées précédemment (cf. § 2.3.1.3, p. 26). Ces objets portent tous une polarité syntaxique qui permet entre autres de vérifier que l'objet porte les fonctions nécessaires

(un nom a une valeur pour la fonction GENRE, etc.). Les objets portent également une polarité p_{arbre} qui permet de vérifier que l'arbre représenté par la structure polarisée est bien formé (voir la section suivante).

Dans la figure 25, nous utilisons encore une fois une notation implicite. Le nœud correspondant au lexème DORMIR est un objet de type *nœudsynt*, son étiquette DORMIR est la valeur de la fonction ÉTIQUETTE et sa couleur correspond à la polarité p_{synt}. La matrice adjacente contient une étiquette—la valeur de la fonction PDD—ainsi qu'un objet grammémique. Dans la notation explicite, le grammème a une fonction structurante HÔTE qui pointe vers le nœud syntaxique, et ce dernier a une fonction structurante MODE qui pointe vers le grammème en retour. La couleur du carreau porté par le grammème correspond à sa polarité p_{synt} et *ind* est la valeur de la fonction ÉTIQUETTE.

La grammaire d'arbres La grammaire d'arbres ($\mathcal{G}_{\text{arbre}}$) vérifie qu'une structure syntaxique est bien un arbre. Concrètement, elle permet de vérifier que chaque nœud est la cible de tout au plus un arc, sauf la racine qui ne doit être la cible d'aucun arc.

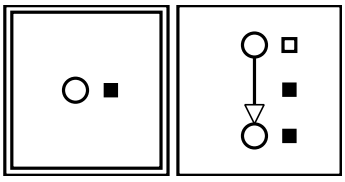


Figure 26 – La grammaire d'arbres

La grammaire ne contient que les deux règles de la figure 26. Dans ces règles, la couleur d'un objet correspond à sa polarité p_{synt}, et celle du carré adjacent, à sa polarité p_{arbre}. Comme le produit d'une polarité blanche avec une autre polarité donne toujours cette autre polarité en GUST, l'application des règles de $\mathcal{G}_{\text{arbre}}$ n'affecte que les polarités p_{arbre}. L'ajout d'une polarité p_{arbre} aux nœuds et arcs syntaxiques fait en sorte qu'une structure syntaxique n'est jamais saturée si elle ne respecte pas la contrainte structurale.

La règle de gauche n'est appliquée qu'une fois, sur la racine de l'arbre. Le double encadré indique que la règle est **initiale**, c'est-à-dire qu'elle ne peut être utilisée qu'une fois. La

règle de droite est ensuite appliquée au reste de l'arbre. Puisque la règle de droite est la seule qui peut saturer la polarité p_{arbre} d'un arc, son application échoue lorsqu'un arc a pour cible un nœud déjà saturé puisque le produit de deux polarités noires échoue.

Pour une description plus complète de la grammaire syntaxique et un tableau récapitulatif des types d'objets et fonctions disponibles, voir Lareau (2008).

2.3.2.5 L'interface sémantique-syntaxe

L'interface sémantique-syntaxe ($\mathcal{I}_{\text{sém-synt}}$) fait le pont entre les niveaux sémantique et syntaxique. Elle comprend des règles à trois parties : la partie de gauche est un fragment de réseau sémantique décrit par $\mathcal{G}_{\text{sém}}$, celle de droite, un fragment d'arbre de dépendance décrit par $\mathcal{G}_{\text{synt}}$, et celle du centre, un ensemble de correspondances entre certains objets de gauche (des signifiés) et de droite (des signifiants). Ces correspondances ne sont rien de plus que des objets de type *correspondance* dotés d'une polarité $p_{sém-synt}$.

Il faut noter que les règles de $\mathcal{I}_{\text{sém-synt}}$ ne transforment pas un réseau sémantique en un arbre de dépendance. L'application d'une règle de $\mathcal{I}_{\text{sém-synt}}$ à un réseau sémantique donne plutôt une structure plus complexe qui comprend à la fois des fragments de réseau sémantique et d'arbre de dépendance. Il est donc possible d'appliquer par la suite une règle qui n'affecte que la partie sémantique de la structure, puis d'utiliser cette nouvelle information pour appliquer une règle d'interface, et ainsi de suite.

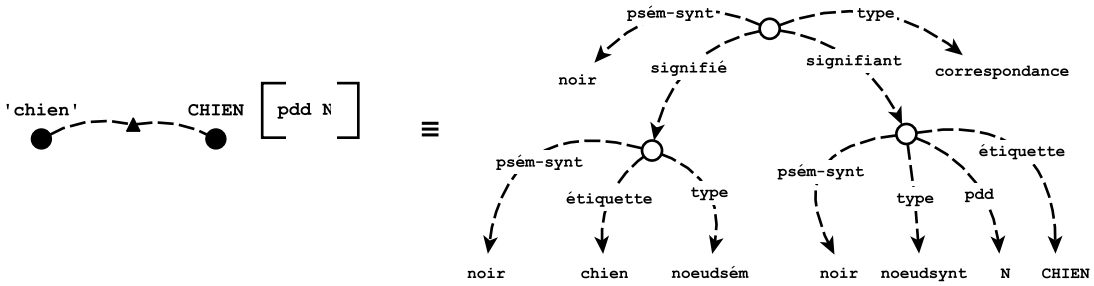


Figure 27 – Les versions implicite et explicite d'une règle de correspondance

Encore une fois, les règles de $\mathcal{I}_{\text{sém-synt}}$ peuvent être représentées de manière implicite,

comme dans la figure 27 où les nœuds triangulaires et les lignes pointillées représentent des correspondances. La couleur d'un objet représente sa polarité $p_{sém-synt}$.

Pour une description plus complète de la grammaire d'interface sémantique-syntaxe et un tableau récapitulatif des types d'objets et fonctions disponibles, voir Lareau (2008).

2.4 Synthèse

Dans ce chapitre, nous avons présenté la grande famille des grammaires d'unification ainsi qu'un formalisme particulier, celui des grammaires d'unification polarisées (GUP) qui se distingue des autres de deux grandes façons :

Alors que les grammaires d'unification manipulent typiquement des structures de traits où une structure est la principale, les grammaires polarisées manipulent des *ensembles* d'objets (de structures de traits) sans objet principal qu'on appelle des structures polarisées. Une des conséquences de cette configuration est qu'il faut envisager un nombre parfois élevé de combinaisons pour chaque paires de structures ; cela présente un problème computationnel important, comme nous le verrons dans les chapitres qui suivent.

Les objets se distinguent également des structures de traits du fait qu'elles portent une ou plusieurs polarités par le biais de fonctions de polarisation. Une polarité peut être non neutre et représenter un besoin, ou neutre et représenter une ressource. Pour unifier deux objets, il faut combiner les polarités qu'ils portent à l'aide d'une opération appelée « produit » qui peut échouer et entraîner l'échec de l'unification entière. Une structure n'est bien formée que lorsque toutes les polarités portées par ses objets sont neutres.

Finalement, nous avons présenté la théorie Sens-Texte et la grammaire d'unification Sens-Texte ; cette dernière est une grammaire d'unification polarisée de nature linguistique et nous permet d'évaluer notre implémentation du formalisme GUP au chapitre 4. Nous avons décrit trois grandes classes de règles de GUST : les règles sémantiques et syntaxiques représentent respectivement des réseaux sémantiques et des arbres syntaxiques, et les règles d'interface font le pont entre les deux classes précédentes.

Chapitre 3. Implémentation du formalisme

Dans ce troisième chapitre, nous décrivons une implémentation existante de l'interface sémantique-syntaxe de GUST, puis notre implémentation des grammaires d'unification polarisées. Nous discuterons d'abord de la définition et de l'importation d'une grammaire dans l'outil, puis nous présenterons différents algorithmes d'unification, incluant celui que nous avons développé pour la combinaison de structures polarisées.

3.1 Une implémentation existante de l'interface sémantique-syntaxe de GUST

Le mémoire de Lison (2006) décrit une implémentation de l'interface sémantique-syntaxe de GUST où les règles sont traduites en graphes XDG (Extensible Dependency Grammar) (Debusmann, 2006, Debusmann, Duchier, & Kruijff, 2004, Debusmann, Duchier, & Niehren, 2004). XDG est basé sur la **programmation par contraintes** [18] et se prête à l'implémentation d'une interface car chaque structure peut être représentée sous plusieurs dimensions, soit une par niveau de représentation. Ces graphes à strates multiples correspondent grossièrement aux règles d'interface que l'on retrouve en GUST.

Le graphe XDG de la figure 28 montre les représentations sémantique (haut) et syntaxique (bas) d'une même phrase [19]. Le nœud *zu* est en gris dans la représentation sémantique même s'il n'est pas porteur de sens puisque chaque niveau doit comprendre le même nombre de nœuds en XDG. Cette caractéristique présente un obstacle en GUST car la correspondance entre les niveaux n'est pas toujours biunivoque [20]. Lison règle ce problème à l'aide des nœuds « vides » proposés dans Debusmann (2004).

18. Nous utilisons nous aussi la programmation par contraintes (dans une forme beaucoup plus simple) pour décomposer des polarités en vecteurs (cf. annexe B, p. xv). Nous utilisons cependant la librairie `python-constraint` plutôt que l'environnement de développement XDG.

19. Cette figure est tirée directement de Lison (2006) et n'est reproduite qu'à titre d'exemple de graphe à strates multiples. Dans la grammaire d'unification Sens-Texte, nous considérons par exemple que *einen* prend *roman* comme argument, et non pas l'inverse. Merci à Sylvain Kahane pour cette remarque.

20. Un objet sémantique peut correspondre à plusieurs objets syntaxiques. Par exemple, ⟨patate⟩ peut correspondre à POMME, DE et TERRE.

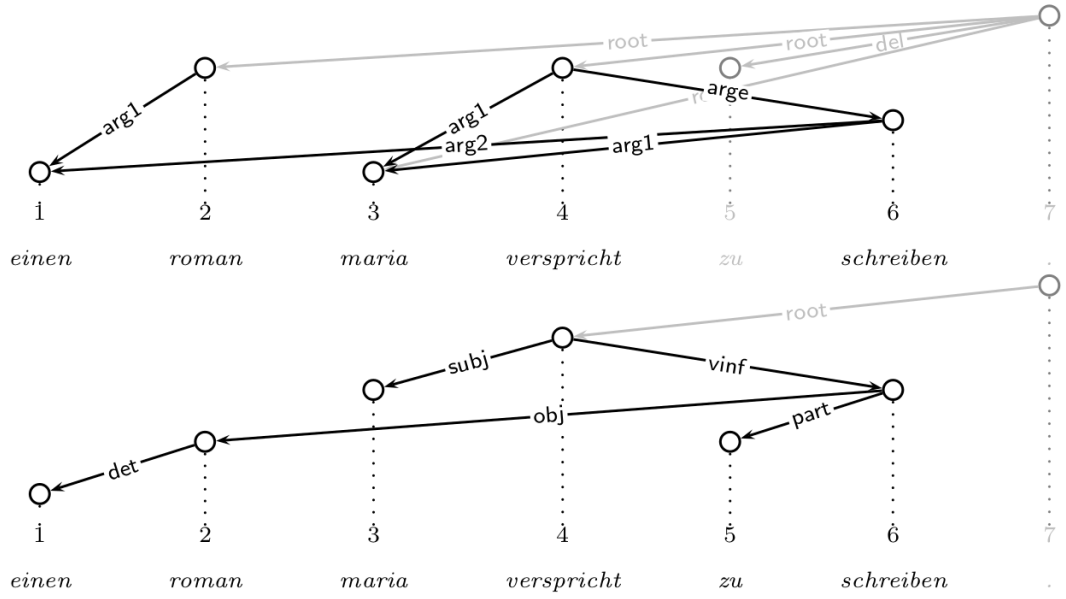


Figure 28 – Deux niveaux de représentation d'une même phrase (Lison, 2006, p. 71)

Le passage d'une représentation sémantique vers une représentation syntaxique peut donc être exprimé sous forme d'un problème de satisfaction de contraintes (CSP). XDG inclut un certain nombre de **contraintes**—par exemple, le résultat d'une analyse doit être un graphe orienté acyclique—et cet ensemble de contraintes peut être étendu. Pour GUST, par exemple, les objets doivent tous être saturés. Un **solveur** trouve des solutions qui respectent toutes les contraintes à l'aide d'algorithmes que nous n'abordons pas ici.

Même si notre implémentation et celle de Lison ont des buts similaires, celle de Lison est spécifique à GUST et vise à générer une représentation syntaxique saturée pour une représentation sémantique donnée. Notre outil permet aussi d'implémenter une grammaire GUST mais se veut générique et se prête davantage à l'expérimentation. Il est donc difficile de les comparer. L'approche est également différente—l'implémentation de Lison ne comprend aucune méthode d'unification et se base entièrement sur l'engin de résolution de contraintes qui vient avec le XDG Development Kit (XDK), tandis que nous avons implémenté une méthode de combinaison des structures polarisées et un algorithme d'unification tenant compte des polarités.

Le code source du compilateur GUST ⇒ XDG n'est plus disponible, sauf à l'intérieur

même du mémoire de Lison où on retrouve une fraction du code (P. Lison, communication personnelle, 22 janvier 2015) ; il n'a donc pas été possible d'essayer l'outil.

Nous n'élaborerons pas davantage sur l'approche de Lison puisqu'elle n'est pas nécessairement applicable dans le cadre de ce mémoire, mais nous invitons tout de même le lecteur intéressé à consulter Debusmann (2004), Debusmann, Duchier, & Kruijff (2004), Debusmann, Duchier, & Niehren (2004), Lison (2006) et Debusmann (2006).

3.2 Définition et importation d'une grammaire d'unification polarisée

Pour faciliter le développement et le partage des grammaires, nous avons développé une syntaxe de définition très simple et écrit un parseur avec le module `pyparsing` (McGuire, 2007). Comme notre outil est distribué librement, la syntaxe utilise des mots-clés anglais pour favoriser l'adoption de l'outil.

La définition d'une grammaire comprend trois grandes parties qui doivent être définies dans l'ordre suivant :

1. Les fonctions et le type de valeurs qu'elles retournent ;
2. Le système des polarités ;
3. Zéro ou plusieurs structures polarisées.

Il est possible d'écrire des commentaires dans la définition. Un commentaire est précédé d'un dièse (#) et peut occuper une ligne entière ou la seconde partie d'une ligne, c'est-à-dire que tout ce qui suit le dièse fait partie du commentaire. Le dièse est interdit dans tous les autres contextes (étiquette, nom d'objet, etc.).

Le parseur est également très permissif au niveau de l'espace dans le fichier de définition, c'est-à-dire qu'on peut par exemple mettre chaque paire fonction-valeur sur une ligne séparée ou mettre toutes les paires sur une même ligne.

Nous décrivons dans les sections suivantes les trois grandes parties d'une définition.

3.2.1 Les types de fonctions

Le parseur doit connaître le type d'une fonction pour savoir si une chaîne de caractères donnée doit être interprétée comme une étiquette, une polarité ou un identifiant d'objet dans une définition de structure.

```
structuring { source cible }
polarizing { psém }
labeling { type étiquette }
```

Figure 29 – Définition des fonctions

La définition des fonctions a trois sections—structuring, polarizing et labeling—qui correspondent respectivement aux ensembles de fonctions structurantes, de polarisation et d'étiquetage. L'ordre des fonctions à l'intérieur d'une section n'a aucune importance, et les sections peuvent également être définies dans n'importe quel ordre.

3.2.2 Le système de polarités

La définition d'un système de polarités comporte trois sections :

— `polarities` : L'ensemble des polarités. L'ordre n'est pas important.

— `neutral` : L'ensemble des polarités neutres. L'ordre n'est pas important.

— `product_table` : Une description partielle ou complète du produit à l'aide de triplets où la troisième polarité est le produit des deux premières.

```
polarities {
    blanc noir plus moins gris
}
neutral { noir gris }
product_table {
    ( gris gris gris )
    ( gris blanc blanc )
    ( gris moins moins )
    ( gris plus plus )
    ( gris noir noir )
    ( blanc blanc blanc )
    ( blanc moins moins )
    ( blanc plus plus )
    ( blanc noir noir )
    ( moins plus noir )
}
```

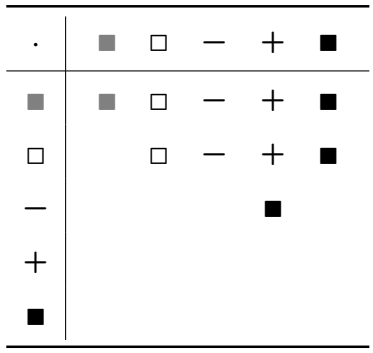


Figure 30 – Définition d'un système de polarités

Dans la définition de la figure 30, on indique que le produit de gris avec lui-même donne gris, gris avec blanc donne blanc, etc. Il n'est pas obligatoire de définir tous les produits. Pour chaque produit (p1 p2 p3) défini, le produit (p2 p1 p3) est ajouté automatiquement pour que le produit soit commutatif [21]. Il n'est pas nécessaire non plus d'indiquer qu'un produit échoue—tout produit manquant donne un échec.

Une fois le tableau des produits construit, le compilateur vérifie que le système de polarités respecte les quatre contraintes de « bonne formation » (cf. § 2.2.2, p. 19). L'instanciation du système de polarités est terminée dès que ces contraintes sont vérifiées.

21. L'outil émet tout de même un message d'erreur s'il trouve deux produits contradictoires.

3.2.2.1 Vérification des contraintes de bonne formation

Nous avons décrit précédemment quatre contraintes nécessaires pour qu'un système de polarités se prête à la description des langues (cf. § 2.2.2, p. 19). Ce sont les contraintes de dynamisme, de monotonicité, de finalité et de déclarativité. Seule la dernière peut être violée, même si ce n'est pas nécessairement souhaitable (Lareau, 2008, p. 213).

Lorsqu'une grammaire est importée, nous vérifions ces contraintes et notons pour chaque contrainte si elle est respectée ou non. L'outil peut être utilisé, même si une contrainte est violée, pour évaluer l'impact sur la combinaison et l'unification.

Contrainte de dynamisme Le système est dynamique s'il a au moins une polarité non neutre et une polarité neutre. Pour vérifier cette contrainte, nous vérifions tout simplement que N est un sous-ensemble strict de P et qu'il comporte au moins une polarité.

Contrainte de monotonicité Pour qu'un système soit monotone, on doit pouvoir ordonner ses polarités de sorte que pour toute paire de polarités, le produit occupe une position égale ou supérieure à la plus haute des deux polarités originales.

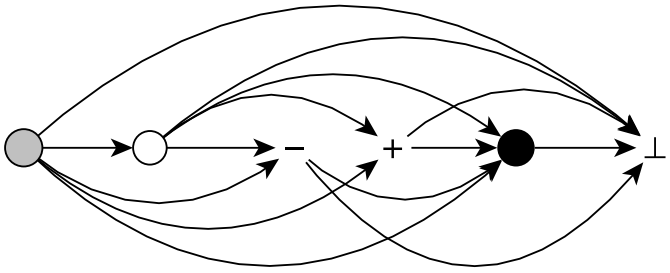


Figure 31 – Le produit représenté sous forme de graphe orienté acyclique

Pour vérifier cette contrainte, nous bâtissons un graphe orienté acyclique, et pour chaque produit (`p1 p2 p3`), nous ajoutons un arc qui part de `p1` et pointe vers `p3`. Comme la monotonicité permet que le produit occupe une position égale à celle d'une des polarités originales, nous éliminons les arcs dont la source et la cible sont le même nœud, évitant ainsi les cycles. Nous essayons ensuite de faire un tri topologique [22] pour ordonner les

nœuds du graphe de manière à ce que la source d'un arc précède toujours sa cible dans la hiérarchie. Le système est monotone si nous parvenons à ordonner les polarités.

Contrainte de finalité Le système respecte la contrainte de finalité s'il est motonone et que les polarités au sommet de sa hiérarchie sont neutres. Pour vérifier cette contrainte, nous récupérons donc la hiérarchie obtenue précédemment et vérifions que les n derniers éléments sont neutres, où n est la taille de N.

Contrainte de déclarativité Le système est déclaratif lorsque son produit est commutatif et associatif. Nous assurons déjà la commutativité du produit lorsque nous bâtissons le tableau des produits (cf. § 3.2.2, p. 38)—il nous suffit donc de vérifier que pour tout triplet de polarités (p1, p2, p3), (p1 . p2) . p3 = p1 . (p2 . p3)..

3.2.3 Les structures polarisées

Tout ce qui suit la définition du système de polarités sert à définir des structures polarisées. Chaque structure possède sa propre section, soit le nom de la structure suivi de la définition des objets qu'elle contient entre accolades :

```
chien # Le nom de la structure.
{
    chien # Le nom de l'objet.
    {
        type=noeudsém      # Une fonction et sa valeur, séparées
        étiquette=chien    # par un signe égal.
        psém=noir
    }
}
```

Figure 32 – Définition d'une structure dormir

22. Nous n'avons pas implémenté nous-même le tri topologique : le module `networkx` (Hagberg et al., 2008) comprend déjà une implémentation. La fonction `networkx.topological_sort` retourne une liste de polarités si le système de polarités est monotone ou lance une exception en présence de cycles.

Le nom de la structure est arbitraire, mais doit être unique. Si le parseur rencontre une structure avec un nom déjà pris, il se contente d'écraser la structure originale.

Les objets sont définis avec un nom arbitraire et un ensemble de paires fonction-valeur entre accolades. On utilise le nom d'un objet pour l'assigner comme valeur à une fonction structurante—la valeur de la fonction SOURCE de l'objet 1, par exemple, est le premier objet défini, qui porte le nom dormir. Lorsque le parseur rencontre un objet dont le nom est déjà pris, il écrase encore une fois l'objet original.

3.3 Sélection d'un algorithme d'unification

La sélection d'un algorithme d'unification adéquat est cruciale, en particulier pour notre implémentation des grammaires d'unification polarisées puisque celles-ci impliquent un très grand nombre d'unifications. Nous décrivons ici trois algorithmes, soit ceux de Pereira (1985), Wroblewski (1987) et Tomabechi (1991). Nous présentons également les travaux de van Lohuizen (2000), qui améliorent l'approche de Tomabechi.

D'autres algorithmes d'intérêt sont présentés dans les textes de Godden (1990), Kogure (1990) et Emele (1991) mais ne sont pas décrits ici.

3.3.1 Quelques concepts-clé

Avant de nous lancer dans la description des différents algorithmes d'unification, nous devons expliquer quelques concepts et expressions que nous utiliserons tout au long de ce chapitre. Certains de ces termes réfèrent à des propriétés des algorithmes d'unification, et d'autres à des concepts de programmation orientée objet (en Python).

3.3.1.1 Terminologie de programmation orientée objet

Une **classe** décrit un type d'entité (les structures polarisées, par exemple) et regroupe différents blocs de code qui servent à manipuler une **instance** de la classe (une structure po-

larisée spécifique). Chaque instance a des propriétés qu'on appelle des **attributs**—chaque instance de notre classe `Object`, par exemple, est dotée d'un attribut `neutral` qui indique si l'instance est neutre (toutes les polarités portées par l'objet sont neutres) ou non. Nous écrirons dorénavant attribut_1 pour parler d'un attribut porté par une structure de traits, et attribut_2 pour un attribut tel que défini ici.

Une **copie** est une instance créée à partir d'une instance existante. Cette nouvelle instance est identique à l'originale et possède les mêmes valeurs pour chaque attribut_2. Il est ensuite possible de modifier la copie sans affecter l'instance originale.

Un **dictionnaire** est une structure de données qui contient des paires clé-valeur. Une clé peut être une chaîne de caractères, un nombre, etc. Un des dictionnaires utilisés dans notre implémentation prend pour clé une instance de la classe `Object` et la valeur associée est un dictionnaire enchâssé de paires fonction-valeur. Dorénavant, le terme *dictionnaire* fera référence à cette structure de données plutôt qu'à une ressource lexicale.

3.3.1.2 Terminologie des algorithmes d'unification

Un algorithme d'unification peut être **destructif**, **quasi destructif** ou **non destructif**. On dit qu'une unification est **destructive** lorsqu'elle modifie les structures originales. Ainsi, si on unifie deux structures, au moins une d'elles est modifiée de façon permanente. Ce comportement n'est pas toujours souhaitable, en particulier dans un formalisme comme GUST où les structures manipulées représentent des règles.

Une unification **non destructive** laisse les structures originales intactes. Ce type d'unification pose souvent deux problèmes : la copie précoce et la copie excessive, définies plus bas. Finalement, une unification est **quasi destructive** si elle modifie temporairement les structures et crée seulement une copie si l'unification est un succès.

La **copie précoce** consiste à créer des copies lors d'une unification qui finit par échouer ; ces copies sont essentiellement gaspillées (Tomabechi, 1991). Wroblewski (1987) définit la copie précoce de manière légèrement différente et la limite à la copie des structures

initiales au début du processus d'unification. Nous sommes d'avis que toute copie qui précède un échec est problématique et adoptons donc ici la définition de Tomabechi.

La **copie excessive** consiste à créer trop de copies, par exemple lorsqu'on copie les deux structures initiales avant d'effectuer des opérations destructives. Une solution partielle est la **copie incrémentale** : plutôt que de copier les structures entières, on copie les objets au fur et à mesure qu'ils sont vus pendant l'unification. En cas d'échec, ces copies sont perdues, mais nous avons tout de même réduit le nombre de copies. L'inconvénient de la copie incrémentale est qu'il faut conserver une trace des copies pour éviter de copier un même objet plusieurs fois lorsque la structure contient des réentrances.

Pour notre implémentation des grammaires d'unification polarisées, il est crucial d'éviter autant que possible la copie excessive vu le nombre de combinaisons possibles pour une même paire de structures (cf. § 2.2.3, p. 21) et de favoriser un algorithme non destructif ou quasi destructif comme nous voulons souvent utiliser une règle plus d'une fois.

3.3.2 L'unification avec partage de structures de Pereira

L'unification proposée par Pereira (1985) est basée sur le partage de structures de Boyer & Moore (1972). Le **partage de structures** suppose qu'un DAG peut être représenté à l'aide d'un autre DAG et d'une série de modifications à lui apporter. Spécifiquement, on peut représenter un DAG avec un squelette et un environnement.

Le **squelette** pointe vers le DAG original et l'**environnement** contient les modifications à apporter au squelette pour obtenir le DAG. L'environnement comprend deux types de modifications, soit des redirections et des ajouts d'arcs. Un **ajout d'arc** est une paire attribut_1-valeur à ajouter à un nœud sous forme d'un arc. Une **redirection** indique qu'un nœud a été unifié avec un autre nœud et que ses paires attribut_1-valeur ont été ajoutées à ce dernier par le biais d'ajouts d'arcs. Si nous souhaitons manipuler ce nœud par la suite, nous devrons plutôt dorénavant manipuler celui vers lequel il a été redirigé.

Cette forme de représentation permet d'évaluer « paresseusement » le résultat d'une

unification—nous n'avons pas besoin d'appliquer les modifications au squelette si nous ne réutilisons pas le DAG dans une autre unification. Puisqu'un squelette est à son tour composé d'un squelette et d'un environnement, il faut cependant appliquer les modifications rétroactivement pour obtenir les squelettes lors de chaque unification.

Le processus d'unification est relativement simple. La première étape consiste à **déréférencer** les nœuds donnés en entrée, soit les racines des deux DAGs en début d'unification. Déréférencer un nœud consiste à vérifier s'il a été redirigé, et, le cas échéant, à récupérer le nœud-cible de la redirection. Le nœud-cible peut avoir été redirigé à son tour ; on déréférence donc récursivement jusqu'à ce qu'on trouve un nœud non redirigé.

Une fois les nœuds déréférencés, on regarde si les sous-graphes induits sont identiques, et, le cas échéant, on retourne un d'eux puisqu'aucune modification n'est nécessaire. Un sous-graphe est **induit** par un nœud lorsqu'il contient tous les nœuds accessibles depuis ce nœud—autrement dit, le nœud en question est la racine du sous-graphe.

Si les sous-graphes sont différents, on vérifie si un des nœuds est une feuille, c'est-à-dire un nœud terminal sans étiquette qui représente donc une variable (cf. § 2.1.1, p. 5), et si c'est le cas, on redirige ce nœud vers l'autre nœud. De cette façon, le chemin menant au nœud redirigé pointera désormais vers le sous-graphe induit par le nœud-cible.

N1	N2	Intersection	Différence
sujet catégorie accord	sujet accord	sujet accord	catégorie

Tableau 4 – L'intersection et la différence de deux nœuds

Finalement, si les deux nœuds ont des arcs sortants, nous essayons d'unifier les valeurs des arcs communs, soit les arcs qui portent une même étiquette, en appelant récursivement la procédure d'unification qui reçoit alors en entrée les cibles des deux arcs. Nous calculons ici l'**intersection** des deux ensembles d'arcs sortants. Si aucun échec ne survient pendant

cette opération, chacun des arcs associés au premier nœud mais pas au second est ajouté à l'environnement du second nœud. Nous calculons ici la **différence** du premier nœud et du second nœud. Le premier nœud est finalement redirigé vers le second. Le squelette et l'environnement sont tous les deux considérés à chaque étape de l'unification.

Si aucune de ces conditions n'est respectée—les graphes induits ne sont pas identiques, les nœuds ne sont pas des feuilles et n'ont pas tous les deux des arcs sortants—ou si un échec survient lors d'une unification, récursive ou non, l'unification entière échoue.

Nous reproduisons ici l'algorithme de Pereira. Nous nous sommes inspirés du pseudocode présenté dans Tomabechi (1993) mais avons omis certaines modifications apportées par l'auteur afin de rester fidèle à la description originale de l'algorithme par Pereira.

```
fonction UNIFIER(N1, N2):
    N1 <-- déréférencer(N1)
    N2 <-- déréférencer(N2)
    SI N1 et N2 sont identiques ALORS
        retourner N2
    OU SI N1 est de type feuille ALORS
        rediriger(N1, N2)
        retourner N2
    OU SI N2 est de type feuille ALORS
        rediriger(N2, N1)
        retourner N1
    OU SI N1 et N2 ont tous deux des arcs sortants ALORS
        intersect <-- intersection(N1, N2)
        diff <-- différence(N1, N2)
        POUR CHAQUE arc dans intersect FAIRE
            unifier(valeur de arc dans N1, valeur de arc dans N2)
        POUR CHAQUE arc dans diff FAIRE
            ajouter arc à N2
        rediriger(N1, N2)
    SINON
        retourner ÉCHEC
```

3.3.3 L'unification non destructive de Wroblewski

L'algorithme de Wroblewski (1987) est non destructif et vise à éliminer complètement la copie précoce et limiter autant que possible la copie excessive. Dans cette approche, chaque nœud du graphe a quatre attributs$_2$—forward, arc-list, copy et status.

— forward : pointe vers le nœud vers lequel le nœud qui porte l'attribut$_2$ a été redirigé, lorsqu'applicable ;

— arc-list : contient les paires attribut$_1$-valeur associées au nœud ;

— copy : pointe vers la copie du nœud lorsqu'applicable ;

— status : indique si le nœud appartient à un des graphes originaux, ou au graphe bâti par l'algorithme d'unification.

Quand deux nœuds sont unifiés, l'un d'eux est redirigé vers l'autre. S'il faut manipuler un nœud, on détermine si ce nœud a été redirigé, et le cas échéant, on manipule plutôt la valeur de l'attribut$_2$ forward. On **déréférence** le nœud lorsqu'on traverse un ou plusieurs attributs$_2$ forward.

L'algorithme utilise un compteur global et un cinquième attribut$_2$, mark, pour déterminer si le contenu des attributs$_2$ forward, copy et status d'un nœud est encore valide. Si mark n'est pas égal au compteur global, alors le contenu des attributs$_2$ remonte à une tentative d'unification antérieure et doit être ignoré.

L'intérêt de cette approche est son usage de la copie incrémentale pour éliminer la copie précoce et limiter la copie excessive. Le graphe résultant est bâti en copiant les nœuds au fur et à mesure qu'ils sont trouvés par l'algorithme pendant qu'il traverse les graphes originaux. Si l'unification échoue, toutes les copies sont gaspillées, mais le nombre de copies est moins élevé que lorsque les graphes originaux sont copiés en début d'unification. L'attribut$_2$ copy d'un nœud nous indique s'il existe déjà une copie de ce nœud et nous permet d'éviter la copie excessive dans la plupart des cas. L'approche ne fonctionne pas toujours parfaitement, notamment en présence de certains cycles.

Bien que l'approche de Pereira est plus efficace que celle de Wroblewski pour éliminer la copie excessive et la copie précoce, le mécanisme de partage de structures ralentit de façon considérable le processus d'unification. Il faut donc décider s'il est préférable de faire quelques copies de trop dans des cas très isolés avec la copie incrémentale ou bien procéder plus lentement mais ne jamais surcopier avec le partage de structures.

Examinons maintenant l'algorithme de Wroblewski proprement dit. La première étape consiste encore une fois à déréférencer les deux nœuds en entrée—les racines des DAGs en début d'unification. On regarde ensuite si les nœuds ont déjà été copiés en examinant la valeur de l'attribut$_2$ copy et en vérifiant que l'attribut$_2$ mark est égal au compteur global pour vérifier la validité des attributs$_2$ temporaires.

Si aucun des nœuds n'a de copie, on crée un nœud vide qui est assigné aux attributs$_2$ copy des nœuds originaux. Pour chaque arc commun aux deux nœuds, on essaie d'unifier les valeurs correspondantes, et si cette unification réussit, l'arc est ajouté à la copie, sinon l'unification entière échoue. Chaque arc présent dans un seul nœud est ajouté à la copie en remplaçant chaque valeur par sa copie lorsqu'applicable. On calcule ici la **différence symétrique** des deux ensembles d'arcs, c'est-à-dire l'ensemble des arcs présents dans un nœud mais pas l'autre. Finalement, on retourne la copie.

Si un seul des nœuds a une copie, on invoque une fonction d'unification *destructive* pour unifier cette copie avec le nœud sans copie. L'usage d'une fonction destructive n'est pas problématique ici puisque c'est la copie qui est modifiée et permet de limiter le nombre de copies créées. La copie est retournée en cas de réussite, sinon on retourne un échec.

Finalement, si les deux nœuds ont une copie, nous invoquons encore une fois la fonction destructive puisque les copies peuvent être modifiées en toute sécurité.

Nous reproduisons ci-dessous notre traduction de l'algorithme de Wroblewski, incluant la fonction destructive qui est très semblable à l'algorithme de Pereira. La différence, bien entendu, est qu'ici les changements sont apportés directement à la copie plutôt que stockés dans l'environnement comme dans l'approche de Pereira.

```
FONCTION unifier1(N1, N2):
    N1 <-- déréférencer(N1)
    N2 <-- déréférencer(N2)
    SI N1 et N2 sont identiques ALORS
        retourner N1 ou N2
    SINON
        rediriger(N1, N2)
        POUR CHAQUE arc dans intersection(N1, N2) FAIRE
            unifier1(valeur de arc dans N1, valeur de arc dans N2)
            SI unifier1 échoue ALORS
                retourner ÉCHEC
            SINON
                remplacer la valeur de arc dans N2
        POUR CHAQUE arc dans différence(N1, N2) FAIRE
            ajouter arc à N2
    retourner N2 ou N1

FONCTION unifier2(N1, N2):
    N1 <-- déréférencer(N1)
    N2 <-- déréférencer(N2)
    SI N1 et N2 n'ont aucune copie ALORS
        copie <-- un noeud vide
        N1.copy, N2.copy <-- copie
        POUR CHAQUE arc dans intersection(N1, N2) FAIRE
            unifier2(valeur de arc dans N1, valeur de arc dans N2)
            SI unifier2 échoue ALORS
                retourner ÉCHEC
            SINON
                ajouter arc à copie
        POUR CHAQUE arc dans différence-symétrique(N1, N2) FAIRE
            ajouter arc à   copie et remplacer la valeur
            par sa copie si nécessaire
    OU SI N1 ou N2 a une copie ALORS
        en supposant que N1 a la copie,
        unifier1(N1.copie, N2)
        retourner N1.copie
    OU SI N1 et N2 ont tous deux une copie ALORS
        unifier1(N1.copie, N2.copie)
```

3.3.4 L'unification quasi destructive de Tomabechi

L'unification quasi destructive de Tomabechi et ses différentes formes sont décrites dans Tomabechi (1991, 1992, 1993, 1995), mais nous nous contentons de décrire ici sa version sans partage, introduite dans Tomabechi (1991). L'algorithme se distingue de celui de Wroblewski du fait qu'il élimine complètement la copie excessive et la copie précoce.

Une des motivations derrière l'algorithme de Tomabechi est l'idée que toute copie qui précède un échec est gaspillée. Wroblewski et Tomabechi conçoivent donc la copie précoce de manière différente—Wroblewski définit la copie précoce comme la copie des structures initiales avant de les unifier, alors que pour Tomabechi, toute copie—partielle ou non—avant une unification réussie est inefficace et constitue de la copie précoce.

L'approche de Tomabechi n'est donc pas basée sur la copie incrémentale et utilise plutôt un attribut$_2$ comp-arc-list pour stocker les arcs temporaires associées au nœud. L'algorithme prend en considération le contenu de cet attribut$_2$ chaque fois qu'il dresse une liste des arcs sortants d'un nœud. Comme dans l'approche de Wroblewski, le contenu de l'attribut$_2$ est rendu obsolète en incrémentant un compteur global. comp-arc-list nous permet donc de laisser les graphes intacts—au long terme du moins—et de n'effectuer aucune copie avant de savoir que l'unification est un succès.

Tomabechi utilise en tout huit attributs$_2$ pour représenter un nœud :

- type : indique si le nœud correspond à une valeur atomique, à une variable ou à une valeur complexe. Le type est complexe si le nœud a des arcs sortants ;
- arc-list : contient les paires attribut$_1$-valeur permanentes associées au nœud. Pour un nœud de type atomique, cet attribut$_2$ contient la valeur atomique
- comp-arc-list : contient les paires attribut$_1$-valeur temporaires associées au nœud ;
- forward : pointe vers le nœud vers lequel le nœud qui porte l'attribut$_2$ a été redirigé, lorsqu'applicable ;
- copy : pointe vers la copie du nœud lorsqu'applicable ;
- comp-arc-mark : indique la valeur du compteur global associée à comp-arc-list ;

— forward-mark : indique la valeur du compteur global associée à forward ;

— copy-mark : indique la valeur du compteur global associée à copy.

(Les attributs$_2$ comp-arc-mark, forward-mark et copy-mark sont regroupés sous un même attribut$_2$ generation dans les versions suivantes de l'algorithme.)

Le processus d'unification est très similaire à celui proposé par Wroblewski. La première étape consiste à déréférencer les nœuds. Si les nœuds déréférencés sont identiques, on retourne tout simplement un succès.

Si les nœuds sont différents, on regarde si leurs types sont compatibles. Si un des nœuds est de type variable, ce nœud est redirigé vers l'autre nœud et on retourne un succès. Si les deux nœuds sont de type atomique et qu'ils ont la même valeur, l'un d'eux est redirigé vers l'autre et on retourne un succès, sinon on retourne un échec. Si un seul nœud est atomique, l'unification échoue.

Si aucune de ces conditions n'est remplie, les nœuds sont complexes et nous comparons donc le contenu de leurs attributs$_2$ arc-list et comp-arc-list. Pour chaque arc commun aux deux nœuds, nous lançons récursivement l'unification des valeurs correspondantes. Si toutes ces unifications sont réussies, le deuxième nœud est redirigé vers le premier, puis on assigne la valeur du compteur global à l'attribut$_2$ comp-arc-mark du premier nœud. Finalement, on calcule la différence entre le second nœud et le premier, soit l'ensemble des arcs présents dans le second nœud mais pas le premier, puis ces arcs sont ajoutés à l'attribut$_2$ comp-arc-list du premier nœud et on retourne un succès.

Si l'unification des deux racines est un succès, on copie la première racine. Le compteur global est ensuite incrémenté, peu importe le résultat de l'unification.

van Lohuizen (2000) suggère quelques modifications permettant de paralléliser l'unification et de résoudre plusieurs problèmes liés à l'approche de Tomabechi. Nous décrivons ces modifications dans la section suivante.

Nous reproduisons ici notre traduction de l'algorithme de Tomabechi. Nous avons pris la liberté de rassembler les fonctions unify-dg et unify1 sous une fonction unifier-dag.

```
FONCTION unifier-dag(RACINE1, RACINE2):
    résultat <-- unifier0(RACINE1, RACINE2)
    compteur <-- compteur + 1
    SI résultat est un succès ALORS
        retourner copier-dag(RACINE1)

FONCTION unifier1(N1, N2):
    N1, N2 <-- déréférencer(N1), déréférencer(N2)
    SI N1 et N2 sont identiques ALORS
        retourner SUCCÈS
    OU SI N1 est de type variable ALORS
        rediriger(N1, N2)
    OU SI N2 est de type variable ALORS
        rediriger(N2, N1)
    OU SI N1 et N2 sont tous deux de type atomique ALORS
        SI N1 et N2 ont la même valeur atomique ALORS
            retourner SUCCÈS
        SINON
            retourner ÉCHEC
    OU SI un seul des noeuds est de type atomique ALORS
        retourner ÉCHEC
    SINON
        intersect <-- intersection(N1, N2)
        POUR CHAQUE arc dans intersect FAIRE
            unifier1(valeur de arc dans N1, valeur de arc dans N2)
        rediriger(N2, N2)
        N1.comp-arc-mark <-- compteur
        N1.comp-arc-list <-- différence(N2, N1)
        retourner SUCCÈS
```

3.3.4.1 L'approche de van Lohuizen

Les algorithmes de Tomabechi et Wroblewski ont un problème important : les attributs temporaires occupent de la mémoire, même si la structure qui contient le nœud n'est pas en cours d'unification. Cela signifie que beaucoup de mémoire est allouée pour conserver de l'information obsolète.

La solution proposée par van Lohuizen (2000) est de stocker cette information dans des structures de données externes. Cette approche a deux avantages : les nœuds occupent

moins d'espace et il est possible de **paralléliser** l'unification, c'est-à-dire de lancer plus d'une unification dans des fils d'exécution différents.

Plus concrètement, van Lohuizen suggère d'utiliser deux tables de hachage temporaires :

— comp-arc-list prend pour clé un nœud et retourne les arcs (paires attribut$_1$-valeur) temporaires qui lui sont attribués en cours d'unification ;

— forward prend pour clé un nœud et, si celui-ci a été redirigé vers un autre nœud en cours d'unification, retourne le nœud-cible de la redirection.

comp-arc-list est consulté en premier lorsqu'on essaie d'accéder aux attributs$_1$ d'un nœud, et on déréférence un nœud avant d'essayer de le manipuler. Les tables sont simplement vidées après chaque unification, que celle-ci ait réussi ou échoué, et nous n'avons alors plus besoin d'un compteur global pour vérifier la validité des données.

Stocker cette information dans des structures de données externes permet également d'implémenter un algorithme **concurrent**. Plutôt que d'évaluer chaque pile d'unification une par une en effaçant les tables de hachage avant de passer à la pile suivante, nous pouvons évaluer plusieurs piles dans des fils d'exécution différents dotés de leurs propres tables comp-arc-list et forward. Cette approche est difficile à implémenter avec les algorithmes de Tomabechi et Wroblewski, puisque l'information y est stockée à l'intérieur des nœuds et qu'il n'est alors pas avisé de laisser plus d'un fil y accéder en même temps, à moins de donner aux nœuds des attributs temporaires pour chaque fil, ce qui devient difficile à gérer. Les tables de hachage permettent aux nœuds d'être en lecture seule et de servir à évaluer plusieurs piles d'unification simultanément.

Lorsqu'une unification réussit et que nous souhaitons copier la structure résultante, nous pouvons également utiliser une table copy pour garder une trace des nœuds déjà copiés. Encore une fois, stocker cette information dans une table de hachage permet d'éviter de conserver de l'information obsolète à l'intérieur des nœuds suite à la copie.

3.4 Le processus de combinaison des structures

Nous décrivons ici notre implémentation de la combinaison des structures polarisées. La première partie du processus—la génération des piles d'unification—est une de nos contributions originales, alors que l'algorithme d'unification est basée sur les travaux de Kahane & Lareau, ainsi que sur Tomabechi (1991) et van Lohuizen (2000).

Deux considérations ont guidé notre implémentation :

Une même structure peut servir à plusieurs reprises Dans le formalisme GUST, par exemple, chaque structure représente une règle linguistique, et on souhaite pouvoir utiliser une règle plus d'une fois. On doit alors éviter une approche destructive et favoriser plutôt une approche quasi destructive ou non destructive.

La combinaison de structures implique de nombreuses unifications Comme la combinaison de structures implique beaucoup d'unifications d'objets, il est important d'éliminer autant que possible la copie précoce et la copie excessive. Puisque l'approche de Wroblewski n'élimine que partiellement la copie excessive, il est préférable d'utiliser une approche comme celle de Tomabechi. L'approche de van Lohuizen permet également de diminuer la quantité de mémoire allouée pour unifier les objets et d'accélérer le processus de combinaison en rendant possible la parallélisation de l'évaluation des piles.

Nous avons donc implémenté une version simplifiée de l'algorithme de Tomabechi avec certaines des modifications proposées par van Lohuizen. Dans sa version actuelle, notre outil n'utilise pas le partage de structures, dont Tomabechi se sert dans toutes les versions subséquentes de son algorithme.

La section suivante décrit la génération des piles d'unification, un problème qui n'a pas été abordé précédemment dans le cadre des grammaires d'unification polarisées (à notre connaissance du moins). Cette étape est critique puisqu'elle a beaucoup plus d'impact sur le temps requis pour combiner des structures que l'algorithme choisi.

3.4.1 Génération des piles d'unification

La première étape de combinaison de structures est de générer une liste des piles d'unification. Par **pile d'unification**, nous entendons une liste de paires d'objets qu'on demande explicitement à l'outil d'unifier ; d'autres paires peuvent tout de même être unifiées par le biais d'une fonction structurante partagée par les objets d'une paire.

Nous obtenons ces paires en calculant le **produit cartésien** des deux structures à unifier. Le produit cartésien nous donne l'ensemble des paires où le premier objet appartient à la première structure et le deuxième à la deuxième structure. Si on unifie les structures A à 3 objets et B à 5 objets, alors on obtient $\bar{\bar{A}} \times \bar{\bar{B}} = 15$ paires, où $\bar{\bar{A}}$ et $\bar{\bar{B}}$ correspondent respectivement aux nombres d'objets dans les structures A et B.

	b1	b2	b3	b4	b5
a1	(a1, b1)	(a1, b2)	(a1, b3)	(a1, b4)	(a1, b5)
a2	(a2, b1)	(a2, b2)	(a2, b3)	(a2, b4)	(a2, b5)
a3	(a3, b1)	(a3, b2)	(a3, b3)	(a3, b4)	(a3, b5)

(a1-3 et b1-5 sont des noms arbitraires donnés aux objets.)

Tableau 5 – Produit cartésien de structures à 3 et 5 objets

On calcule ensuite l'**ensemble des parties** de l'ensemble de paires pour obtenir toutes les piles de longueur [1, n], où n est le nombre de paires. Cet ensemble contient 2^n piles, soit $2^{15} = 32\,768$ piles pour A et B. Chaque pile est unique—il n'en existe aucune autre de même taille qui contienne les mêmes paires, et ce, peu importe l'ordre des paires. On ne retrouvera par exemple qu'une seule pile parmi les suivantes : ⟨(a1, b1), (a2, b2)⟩ et ⟨(a2, b2), (a1, b1)⟩. La pile vide est retirée pour unifier au moins une paire d'objets. Nous avons donc tout au plus $2^n - 1$ piles à envisager.

Longueur	Exemple
1	⟨(a1, b1)⟩
2	⟨(a1, b1), (a1, b2)⟩
3	⟨(a1, b1), (a1, b2), (a1, b3)⟩
4	⟨(a1, b1), (a1, b2), (a1, b3), (a1, b4)⟩
...	...
15	⟨(a1, b1), (a1, b2), (a1, b3), (a1, b4), (a1, b5), (a2, b1), (a2, b2), (a2, b3), (a2, b4), (a2, b5), (a3, b1), (a3, b2), (a3, b3), (a3, b4), (a3, b5)⟩

Tableau 6 – Piles d'unification pour un ensemble de 15 paires

Le nombre de piles est élevé pour d'aussi petites structures et il double pour chaque paire d'objets additionnelle. On se retrouve rapidement avec plusieurs millions de piles à évaluer. Il devient alors nécessaire de sélectionner au préalable des piles susceptibles de réussir et de donner des structures uniques plutôt que d'attendre qu'elles échouent en cours d'évaluation et de filtrer les structures identiques suite à la combinaison.

Nous utilisons trois méthodes pour réduire le nombre de piles envisagées. La première consiste à filtrer les paires **non viables**, soit des paires d'objets dont les paires fonction-valeur sont incompatibles. Nous filtrons ces paires en vérifiant que les étiquettes présentes sont compatibles, que les polarités peuvent se combiner [23] et que les objets reliés par des fonctions structurantes forment une paire viable. S'il n'est pas possible de déterminer si une paire est viable ou non, nous assumons qu'elle l'est pour éviter de retirer des paires viables. Cette méthode est particulièrement efficace lorsque les objets portent de nombreuses étiquettes, puisqu'il suffit d'avoir deux étiquettes incompatibles pour qu'une paire ne soit pas viable.

Cette méthode est intéressante puisqu'elle est appliquée avant même de calculer l'en-

semble des parties. Chaque paire retirée réduit donc de moitié le nombre de piles.

Ensuite, une unification en déclenche une autre si les objets unifiés partagent une fonction structurante. La deuxième unification est alors impliquée par la première. Nous pouvons déduire qu'une pile qui ne contient que la première paire et une autre qui ne contient que les deux paires auront le même résultat et il est donc inutile de considérer les deux. Nous ne conservons que les piles explicites, soit celles qui énumèrent toutes les unifications impliquées. Cette méthode permet de retirer un bon nombre de piles lorsque les objets portent beaucoup de fonctions structurantes.

Finalement, certains objets sont unifiés indirectement. Dans la pile ⟨(a1, b1), (a1, b2)⟩, par exemple, on unifie d'abord a1 avec b1 puis avec b2. Si les objets b1 et b2 ne sont pas compatibles, nous savons que l'unification va échouer et qu'il est inutile de considérer la pile. Nous retirons donc les piles qui impliquent l'unification de paires indirectes non viables. Plus concrètement, nous bâtissons pour chaque pile un graphe dont les nœuds sont les objets contenus dans la pile et nous ajoutons un arc entre chaque paire à unifier. Pour chaque paire d'objets reliés par un ou plusieurs arcs dans le graphe, nous vérifions ensuite que la paire est viable, et si ce n'est pas le cas, la pile est exclue.

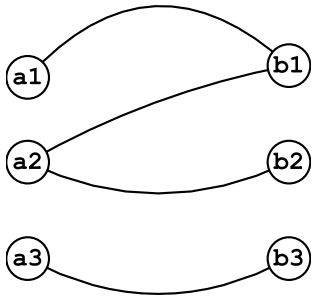


Figure 33 – Graphe pour la pile ⟨(a1, b1), (a2, b1), (a2, b2), (a3, b3)⟩

La pile ⟨(a1, b1), (a2, b1), (a2, b2), (a3, b3)⟩ donne par exemple le graphe de la figure 33. Celui-ci n'est pas entièrement connecté, c'est-à-dire qu'il n'existe pas un chemin entre

23. Nous utilisons ce critère car nous supposons ici qu'un système de polarités peut être décomposé en vecteurs binaires (cf. annexe B, p. xv). Si les polarités portées par les objets sont incompatibles lors du filtrage des paires non viables, alors elles seront toujours incompatibles et leur produit échouera donc toujours, même si les objets sont unifiés avec d'autres objets au préalable.

toutes les paires de nœuds. On retrouve plutôt deux groupes, soit les groupes [a1, a2, b1, b2] et [a3, b3]. On peut donc prédire que les objets du premier groupe seront tous unifiés, indirectement ou non, et que l'évaluation de la pile va échouer si une des paires est non viable. Dans notre exemple, il faut donc que les paires formées à partir du premier groupe—(a1, b1), (a1, b2), (a1, a2), (a2, b1), (a2, b2), (b1, b2)—soient toutes viables, et c'est le cas également pour la paire formée à partir du second groupe.

Nous pouvons difficilement quantifier l'efficacité de ces trois méthodes puisqu'elle varie selon le contenu des structures unifiées. De manière générale, la présence de nombreuses paires fonction-valeur permet de retirer un nombre important de piles d'unification qui échouent systématiquement ou qui donnent des structures identiques. Nous mesurons tout de même l'impact de ce filtrage avec une grammaire de test au chapitre 4 (p. 68).

3.4.2 Évaluation d'une pile d'unification

Chaque pile conservée est évaluée en unifiant chaque paire d'objets qu'elle contient une par une. Des paires d'objets sont parfois ajoutées à la pile en cours d'unification si deux objets partagent une fonction structurante. L'évaluation d'une pile est réussie si on parvient à unifier toutes les paires qu'elle contient et celles ajoutées en cours d'unification.

L'évaluation d'une pile débute en instanciant deux dictionnaires (cf. § 3.3.1), fwtab et compl, qui nous permettent de garder une trace des différentes unifications :

— fwtab prend pour clé un objet et, si celui-ci a été redirigé vers un autre objet en cours d'unification, retourne l'objet-cible de la redirection [24].

— compl prend pour clé un objet et retourne les paires fonction-valeur qui lui ont été attribuées lors d'une unification avec un autre objet.

Nous retirons ensuite la dernière paire de la pile [25] et lançons son unification. Le second objet est redirigé vers le premier en ajoutant une entrée dans fwtab. Dorénavant, si on

24. Ce dictionnaire est conçu de manière à suivre les chaînes de redirection, c'est-à-dire que si a est redirigé vers b et b vers c, alors fwtab retourne c pour la clé a. Nous avons sous-classé la classe dict pour que notre dictionnaire se comporte de cette façon (voir la classe `guptools.utils.Forward`).

tente de manipuler le second objet, on consultera d'abord fwtab et on manipulera plutôt l'objet vers lequel il a été redirigé.

On compare ensuite les fonctions partagées par les deux objets. Cette comparaison tient compte du contenu du dictionnaire compl—nous verrons pourquoi dans un instant. Pour chaque fonction partagée, on vérifie la compatibilité des deux valeurs. La compatibilité dépend du type de la fonction :

— **Fonction d'étiquetage :** Si les objets ont la même étiquette, on ne fait rien, sinon on lance une exception pour interrompre l'unification.

— **Fonction structurante :** On ajoute la paire de valeurs à la pile pour essayer de l'unifier plus tard. On procède ainsi plutôt que de lancer récursivement l'unification des deux objets puisque Python se prête mal à la récursion.

— **Fonction de polarisation :** Si le produit des polarités réussit, on ajoute une entrée dans compl avec cette valeur de la fonction pour le premier objet de la paire, sinon on lance une exception pour interrompre l'unification.

Finalement, chaque fonction qui n'est présente que dans le second objet est ajoutée avec sa valeur au premier objet par le biais du dictionnaire compl. Celui-ci est consulté chaque fois qu'on accède aux paires fonction-valeur d'un objet.

On vérifie ensuite s'il reste des paires à unifier. Si c'est le cas, on retire la dernière et on lance son unification, sinon la combinaison est réussie. Comme les modifications à apporter sont stockées dans compl plutôt que dans les structures et les objets, il faut lancer une procédure qui copie certaines parties des structures originales et applique les modifications. Cette procédure est décrite dans la section suivante.

Nous reproduisons ci-dessous notre algorithme de combinaison en pseudocode. Nous avons omis les fonctions de génération des piles et de copie des structures, mais le lecteur est invité à consulter le code source de notre outil.

25. En informatique, une pile (*stack*) suit le principe LIFO (« last in, first out » ou « dernier entré, premier sorti »), c'est-à-dire qu'on en retire toujours le dernier élément. On peut voir cette structure de données comme une pile littérale dont on ne peut enlever que l'élément du haut (le dernier ajouté).

```
FONCTION combiner(STRUCT_A, STRUCT_B) -> liste de structures
    structures <-- liste vide
    fwtab <-- dictionnaire vide
    compl <-- dictionnaire vide

    POUR CHAQUE pile dans piles(STRUCT_A, STRUCT_B) FAIRE
        ESSAYER
            TANT QUE pile n'est pas vide FAIRE
                A, B <-- prendre-première-paire(pile)
                A <-- déréférencer(A)
                B <-- déréférencer(B)
                SI A et B sont le même objet ALORS
                    retourner au début de la boucle
                rediriger(B, A)

                POUR CHAQUE fonction partagée par A et B FAIRE
                    SI fonction est structurante FAIRE
                        ajouter la paire de valeurs à la fin de pile
                    OU SI fonction est polarisante FAIRE
                        SI les polarités ont un produit FAIRE
                            ajouter le produit à A dans compl
                        SINON
                            lancer ECHEC_UNIFICATION
                    OU SI fonction est d'étiquetage FAIRE
                        SI les étiquettes sont différentes FAIRE
                            lancer ECHEC_UNIFICATION
                POUR CHAQUE fonction présente dans B mais pas A FAIRE
                    ajouter la valeur à A dans compl

        SI ECHEC_UNIFICATION n'a pas été lancé FAIRE
            ajouter copie(STRUCT_A, STRUCT_B, compl, fwtab) à structures

        vider fwtab et compl
    retourner structures
```

3.4.3 Copie suite à une évaluation réussie

On génère une structure résultante à partir des structures originales et des dictionnaires fwtab et compl. La première étape consiste à calculer l'ensemble des objets qu'on retrouvera dans la nouvelle structure. Pour ce faire, nous itérons à travers chacune des structures originales et conservons uniquement les objets qui n'ont pas été redirigés vers un autre objet, soit ceux qui n'ont pas d'entrée dans fwtab. Nous ne gardons donc pas les objets dont les paires fonction-valeur ont déjà été transférées vers un autre objet.

Chacun de ces objets est ensuite copié, et les paires fonction-valeur qui lui sont assignées dans compl sont ajoutées à la copie. Lorsqu'une fonction est présente à la fois dans l'objet et dans son entrée dans compl, c'est la valeur dans compl qui a préséance. En théorie, seule des fonctions de polarisation peuvent être dans un objet et son entrée dans compl.

On itère à travers les copies et pour chaque fonction structurante, on remplace la valeur par sa copie. On consulte bien sûr fwtab au préalable pour voir si la valeur a été redirigée vers un autre objet, et le cas échéant, c'est la copie de cet objet qu'on attribue comme valeur à la fonction structurante.

On crée finalement une nouvelle structure à partir de ces copies.

3.4.4 Résultat de la combinaison de structures

Une fois toutes les piles évaluées, l'ensemble des structures résultantes est filtré pour ne conserver que les structures uniques [26].

Deux structures sont identiques si elles sont **isomorphiques**, c'est-à-dire qu'il existe une correspondance biunivoque entre les objets des deux structures, de sorte que chaque paire d'objets correspondants portent les mêmes paires fonction-valeurs. Il faut également que les objets soient liés par les mêmes fonctions structurantes.

Dans notre outil, une structure est représentée par un graphe où chaque nœud est un objet et contient les étiquettes et polarités liées à l'objet. Chaque arc sortant représente une

fonction structurante et est étiqueté avec le nom de la fonction. Deux nœuds peuvent donc être mis en correspondance s'ils ont les mêmes polarités et étiquettes, et deux arcs peuvent être mis en correspondance s'ils portent la même étiquette et représentent donc la même fonction structurante. Concrètement, nous vérifions l'isomorphisme entre deux structures à l'aide du module `networkx.algorithms.isomorphism`.

Le processus de combinaison est terminé quand les structures uniques sont retournées.

3.4.5 Illustration du processus de combinaison

Nous illustrerons ici le processus de combinaison avec les structures de la figure 34. Dans ces structures, TYPE et POLARITÉ sont respectivement des fonctions d'étiquetage et de polarisation, tandis que SOURCE et CIBLE sont des fonctions structurantes.

26. Notre pré-filtrage des piles permet généralement de réduire considérablement le nombre de structures résultantes identiques. Nous tentons de quantifier l'efficacité de ce filtrage au chapitre 4 avec un fragment de grammaire GUST.

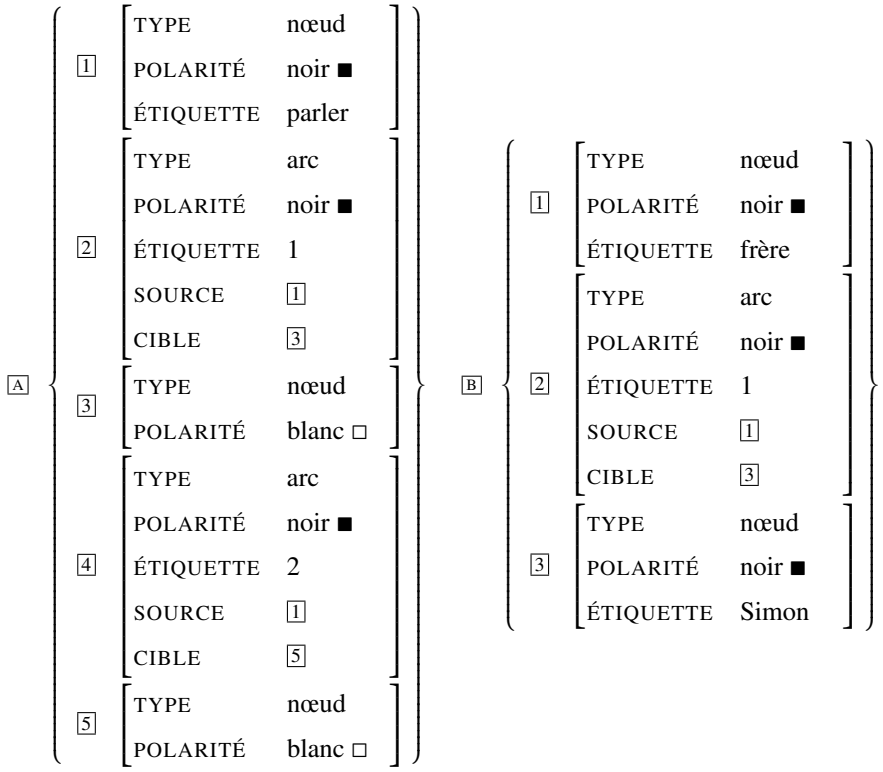


Figure 34 – Deux structures polarisées à unifier

	a1	a2	a3	a4	a5
b1	(a1, b1)	(a2, b1)	(a3, b1)	(a4, b1)	(a5, b1)
b2	(a1, b2)	(a2, b2)	(a3, b2)	(a4, b2)	(a5, b2)
b3	(a1, b3)	(a2, b3)	(a3, b3)	(a4, b3)	(a5, b3)

Tableau 7 – Produit cartésien des structures A et B

Nous débutons avec la génération des piles. Nous calculons d'abord le produit cartésien des structures A et B (qui ont respectivement 5 et 3 objets) pour obtenir les $5 \times 3 = 15$ paires du tableau 7. Pour limiter le nombre de piles générées, nous retirons les paires qui sont non viables, soit celles dont les objets portent des paires fonction-valeur incompatibles. La tableau 8 montre quelles paires sont viables ou non et pourquoi.

(a3, b1)	✓	
(a3, b3)	✓	
(a5, b1)	✓	
(a5, b3)	✓	
(a1, b1)	✗	Étiquettes et polarités incompatibles
(a1, b3)	✗	
(a4, b2)	✗	
(a1, b2)	✗	Types, étiquettes et polarités incompatibles
(a2, b1)	✗	
(a2, b3)	✗	
(a4, b1)	✗	
(a4, b3)	✗	
(a2, b2)	✗	Polarités incompatibles
(a3, b2)	✗	Types différents
(a5, b2)	✗	

Tableau 8 – Paires viables et non viables

Par exemple, (a1, b1) n'est pas viable puisque les objets ont des valeurs différentes pour la fonction ÉTIQUETTE et que les polarités ■ et ■ ne peuvent pas se combiner. La paire (a5, b2) n'est pas viable non plus car les objets ont des types différents. En tout, quatre paires sont viables. Pour générer les piles, nous calculons l'ensemble des parties de l'ensemble de paires viables et obtenons les $2^4 - 1 = 15$ piles des tableaux 9 et 10.

⟨(a3, b1)⟩	⟨(a3, b1), (a5, b1)⟩
⟨(a3, b3)⟩	⟨(a3, b1), (a5, b3)⟩
⟨(a5, b1)⟩	⟨(a3, b3), (a5, b3)⟩
⟨(a5, b3)⟩	⟨(a3, b1), (a3, b3)⟩

Tableau 9 – Piles d'unification à évaluer

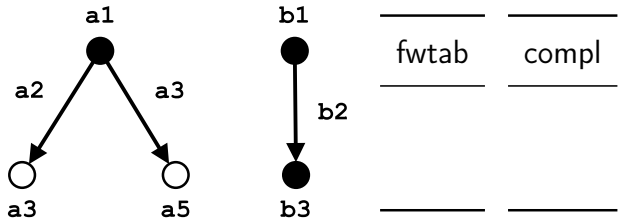
Notre méthode de filtrage basée sur les piles implicites et explicites n'est pas applicable ici puisqu'aucun des objets dans nos piles ne porte de fonction structurante. Celle basée sur les unifications indirectes est cependant applicable. Dans la pile ⟨(a3, b1), (a3, b3), (a5, b3)⟩, par exemple, a3 est unifié avec b1 puis avec b3. Comme les polarités et les étiquettes de b1 et b3 sont incompatibles, nous savons que l'évaluation va échouer. Nous retirons sept piles de cette manière et il n'en reste donc que huit à évaluer.

⟨(a3, b3), (a5, b1)⟩	b1 et b3 sont unifiés indirectement et forment une paire non viable
⟨(a5, b1), (a5, b3)⟩	
⟨(a3, b1), (a3, b3), (a5, b1)⟩	
⟨(a3, b1), (a3, b3), (a5, b3)⟩	
⟨(a3, b1), (a5, b1), (a5, b3)⟩	
⟨(a3, b3), (a5, b1), (a5, b3)⟩	
⟨(a3, b1), (a3, b3), (a5, b1), (a5, b3)⟩	

Tableau 10 – Piles d'unification filtrées avant même d'être évaluées

Nous n'évaluerons ici que la pile ⟨(a3, b1), (a5, b1)⟩. L'évaluation débute avec l'initialisation des dictionnaires fwtab et compl.

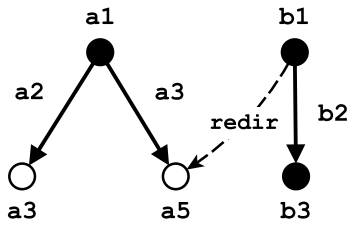
Pile : ⟨(a3, b1), (a5, b1)⟩



Nous retirons la dernière paire de la pile, (a5, b1), et redirigeons b1 vers a5 en ajoutant une entrée dans fwtab. Nous comparons ensuite les fonctions partagées par les objets. TYPE est une fonction d'étiquetage la même valeur dans les deux objets (*nœudsém*) ; nous n'avons rien à faire. POLARITÉ est une fonction de polarisation, alors nous calculons le

produit de □ et ■ et notons dans compl que la valeur de POLARITÉ est maintenant ■ pour l'objet a5. L'objet b1 porte une autre fonction, ÉTIQUETTE, que nous ajoutons avec sa valeur dans compl pour l'objet a5.

Pile : ⟨(a3, b1)⟩

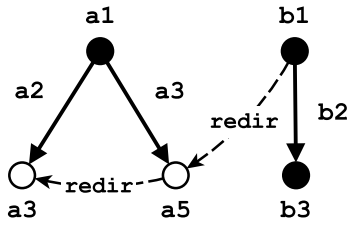


fwtab	compl
b1 ⇒ a5	polarité(a5) ⇒ ■
	étiquette(a5) ⇒ frère

Une fois l'unification terminée, nous vérifions s'il reste des paires dans la pile. Comme c'est le cas, nous retirons la dernière paire, (a3, b1), et consultons fwtab pour voir si le second objet a été redirigé. Nous voyons dans fwtab que b1 a été redirigé vers a5, qui lui n'a pas été redirigé vers un autre objet. Nous remplaçons donc b1 par a5 et nous ajoutons une entrée dans fwtab pour rediriger a5 vers a3.

Les objets partagent deux fonctions. TYPE est une fonction d'étiquetage et porte la même valeur dans les deux objets alors nous n'avons rien à faire. POLARITÉ est une fonction de polarisation, alors nous calculons le produit de □ et ■ et ajoutons le résultat dans compl pour l'objet a3. (On considère que a5 porte une polarité noire et non blanche puisque son entrée dans compl a préséance.) Toujours selon compl, a5 porte une autre fonction, ÉTIQUETTE, que nous ajoutons avec sa valeur dans compl pour l'objet a3.

Pile : ⟨⟩



fwtab	compl
b1 ⇒ a5	polarité(a5) ⇒ ■
a5 ⇒ a3	étiquette(a5) ⇒ frère
	polarité(a3) ⇒ ■
	étiquette(a3) ⇒ frère

Nous vérifions encore une fois s'il reste des paires dans la pile et ce n'est pas le cas. Il ne nous reste donc plus qu'à générer la structure résultante à partir des structures originales et des deux dictionnaires.

Nous faisons une liste de tous les objets dans les deux structures. Nous nous retrouvons avec 8 objets : a1, a2, a3, a4, a5, b1, b2 et b3. fwtab nous indique que parmi ces objets, deux ont été redirigés en cours d'unification, soit les objets a5 et b1. Ces deux objets sont retirés de notre liste d'objets et tous les autres objets sont copiés.

Nous consultons ensuite compl pour mettre à jour nos copies. On donne à la copie de a3 la valeur ■ pour la fonction POLARITÉ et la valeur *frère* pour la fonction CHIEN. Pour chaque objet qui porte une fonction structurante, on remplace la valeur par sa copie, en consultant au préalable fwtab pour avoir la version la plus récente de la valeur. Par exemple, on remplace la valeur de CIBLE dans a4 et celle de SOURCE dans b2 avec la copie de a3, car a5 et b1 ont tous deux été redirigés vers a3.

La génération de la structure résultante est ensuite terminée. Chaque pile est évaluée de la sorte et nous obtenons huit structures résultantes qui sont toutes uniques. Nous retournons finalement ces huit structures et le processus de combinaison est complété.

Chapitre 4. Évaluation

Afin d'évaluer notre outil, nous avons encodé un fragment de la grammaire d'unification Sens-Texte (§ 2.3, p. 23) tiré de Lareau (2008, pp. 239–241) et Kahane & Lareau (2005a). Celui-ci contient des règles des grammaires sémantique et syntaxique, ainsi que des règles d'interface sémantique-syntaxe. En tout, 23 règles ont été encodées.

À l'aide de ces règles, nous avons tenté de répondre aux questions suivantes :

— Nos méthodes de filtrage permettent-elles de retirer un nombre suffisant de piles ?

— Est-il plus rapide de filtrer les piles au préalable ou de simplement les évaluer et voir si l'évaluation échoue ou donne des structures identiques ?

— Les piles évaluées donnent-elles des structures valides et uniques ?

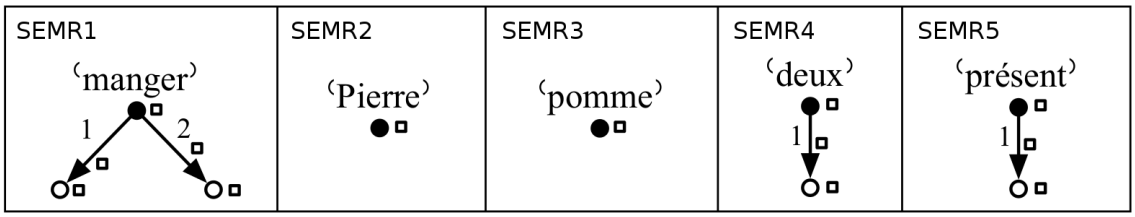


Figure 38 – Cinq règles de bonne formation sémantique

Les figures 38, 40 et 39 sont tirées de Lareau (2008) et les règles y sont représentées sous une forme implicite. Rappelons rapidement en quoi celle-ci consiste. Dans les règles de la figure 38, les nœuds sont des objets de type *nœudsém*, et les arcs, de type *arcsém*. Les chaînes de caractères entre guillemets comme ‹manger› et les numéros portés par les arcs (1, 2) sont des valeurs de la fonction ÉTIQUETTE. La couleur d'un objet et celle du carré adjacent correspondent respectivement à ses polarités sémantique et d'interface.

Dans les règles de la figure 39, les nœuds et les arcs représentent respectivement des objets de type *nœudsynt* et *arcsynt*. Les chaînes de caractères en majuscules près des nœuds et les chaînes de caractères à côté des arcs sont des valeurs de la fonction ÉTIQUETTE. Les crochets à côté de certains nœuds contiennent des étiquettes et des objets de type *grammème* liés au nœud par la fonction HÔTE. La couleur d'un objet et celle du carré adjacent correspondent ici respectivement à ses polarités syntaxique et d'interface.

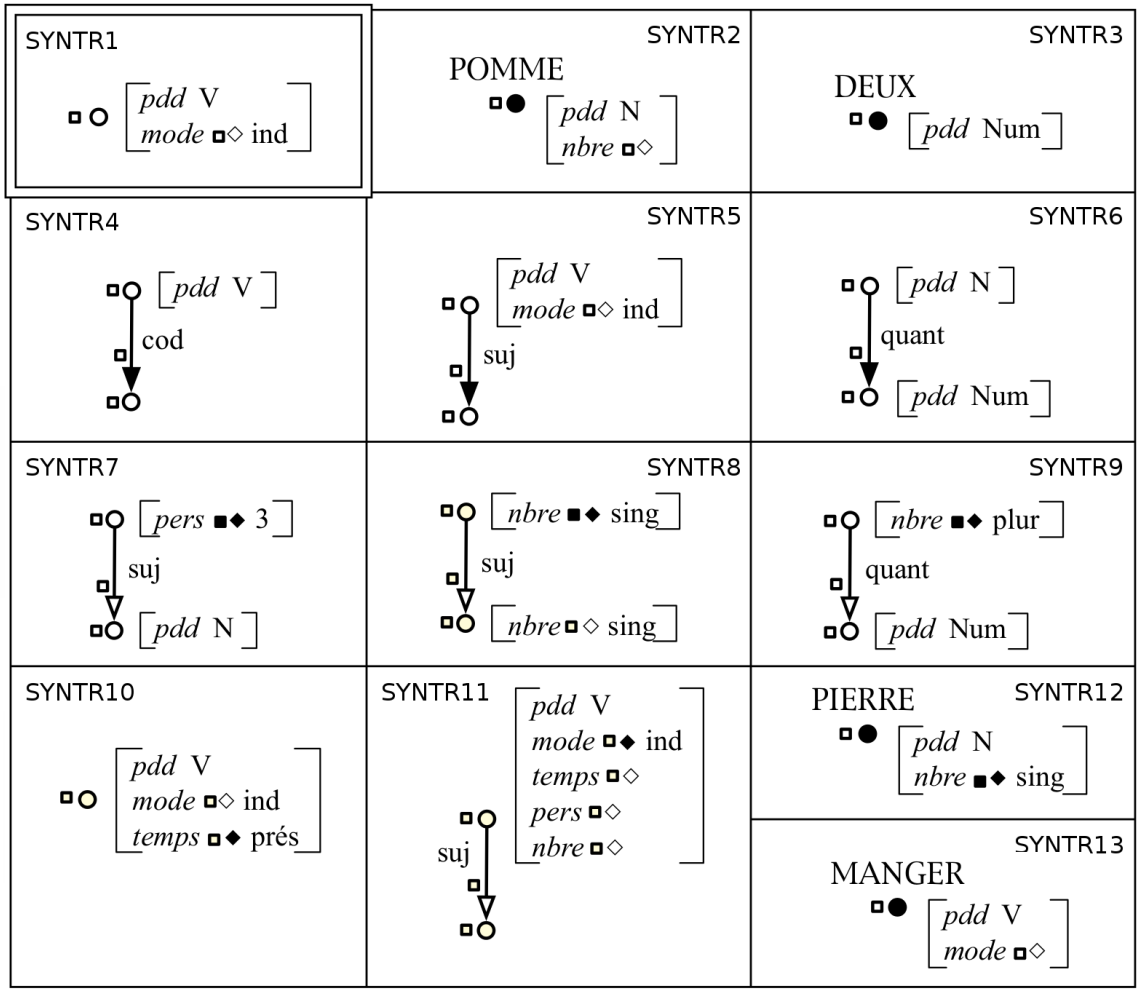


Figure 39 – Treize règles de bonne formation syntaxique

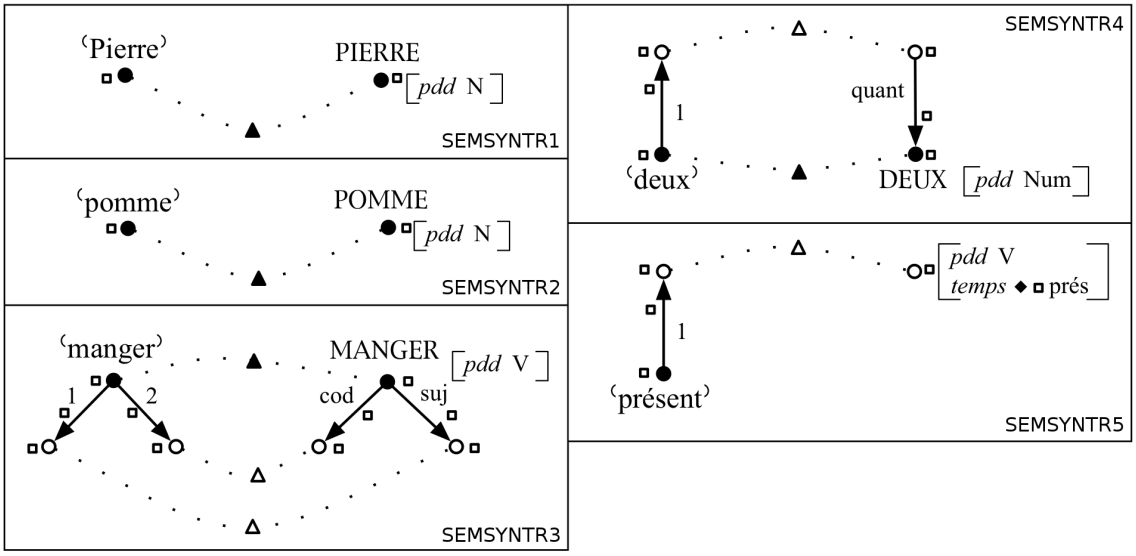


Figure 40 – Cinq règles d'interface sémantique-syntaxe

Dans les règles de la figure 40, finalement, les objets de gauche et de droite ont les mêmes types et fonctions que ceux des figures 38 et 39, respectivement. Les lignes pointillées correspondent à des objets de type *correspondance* liés à un nœud sémantique avec la fonction SIGNIFIÉ et à un nœud syntaxique avec la fonction SIGNIFIANT. La couleur du triangle correspond à la polarité d'interface du lien de correspondance.

4.1 Efficacité des méthodes de filtrage

Nous avons tout d'abord évalué l'efficacité de nos méthodes de filtrage, en commençant par le filtrage des paires non viables. Pour chaque paire de structures, nous avons calculé le nombre total de paires et le nombre de paires viables retenues. Nous avons ensuite calculé la moyenne pour chaque nombre total de paires. Notons que le nombre de paires retenues dépend davantage du contenu des objets que du nombre total de paires.

La figure 41 montre que le filtrage des paires non viables permet de retirer entre 75 et 100% des paires, c'est-à-dire que nous réduisons le nombre de piles à environ $2^{\frac{n}{4}} - 1$ ou moins, où n est le nombre total de paires. Il est donc possible par exemple de passer de 1 048 575 à 31 piles pour deux structures à 4 et 5 objets, soit 20 paires au total.

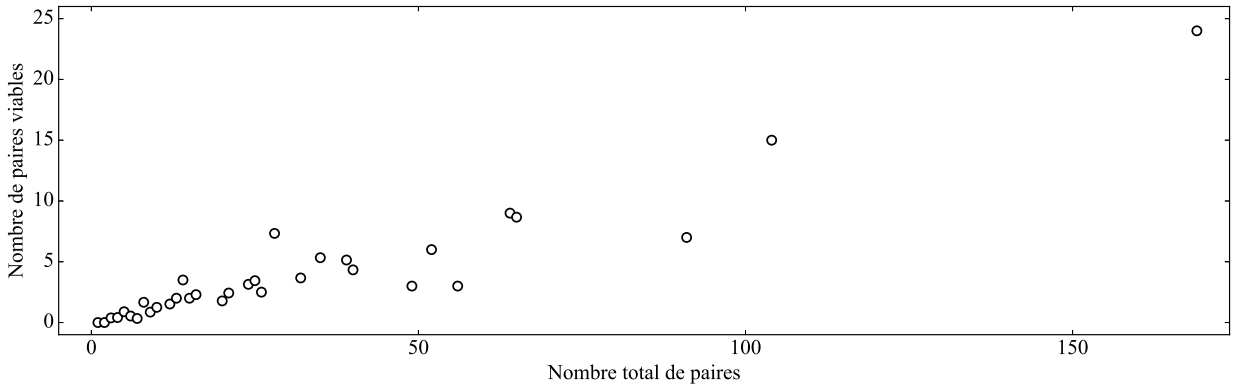


Figure 41 – Nombre de paires viables selon le nombre total de paires

Nous avons ensuite évalué l'efficacité de nos méthodes de filtrage des piles ; cette évaluation porte seulement sur les piles générées suite au filtrage des paires non viables. Pour ce test, nous avons combiné chaque paire de structures de notre grammaire et obtenu environ 2 000 structures supplémentaires. Nous avons ensuite sélectionné au hasard 30 000 paires de structures, et pour chaque paire, nous avons compté :

— Les piles exclues parce qu'elles ne sont pas explicites, c'est-à-dire qu'elles ne contiennent pas toutes les paires dont l'unification est prévisible ;

— Les piles exclues parce qu'elles impliquent indirectement l'unification d'objets qui ne sont pas compatibles ;

— Les piles qui ne sont pas retirées lors du filtrage.

La figure 42 montre la distribution moyenne de ces trois catégories de piles en fonction du nombre de paires viables [27]. Cette distribution est exprimée sous forme de pourcentage, mais rappelons-nous que le nombre total de piles est obtenu avec la formule $2^n - 1$ où n est le nombre de paires.

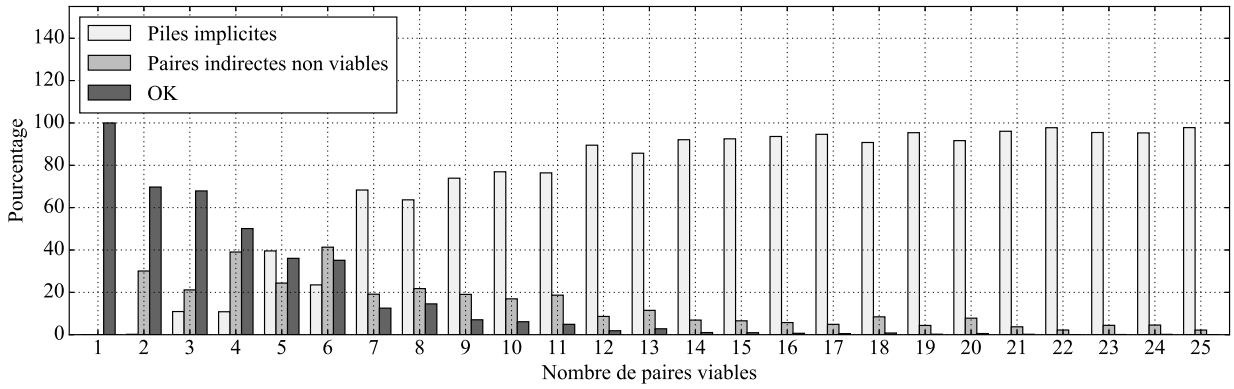


Figure 42 – Pourcentage de piles retirées et retenues selon le nombre de paires

On voit que l'efficacité du filtrage des piles implicites augmente avec le nombre de paires, ce qui est prévisible car les fonctions structurantes sont toujours plus nombreuses avec beaucoup d'objets. Le filtrage des piles qui impliquent indirectement l'unification de paires non viables peut sembler très peu efficace ; il faut cependant noter que les deux méthodes de filtrage mesurées ici sont appliquées séquentiellement, c'est-à-dire qu'on n'applique pas la deuxième méthode aux piles implicites. La deuxième méthode est en réalité très efficace—elle élimine toutes les piles dont l'évaluation échoue (dans nos tests du moins). Chaque pile restante après son application donne une structure valide mais pas nécessairement unique ; nous le verrons lors de notre dernier test (p. 74).

4.2 Temps de filtrage et d'évaluation des piles

Le test suivant consistait à comparer les temps de filtrage et d'évaluation d'une même pile. Ce test visait à déterminer s'il est plus efficace de filtrer les piles au préalable, ou de simplement les évaluer et voir si elles échouent ou donnent des structures identiques.

Pour chaque paire de structures (les structures originales et celles obtenues lors du test précédent) nous avons lancé le processus de génération des piles, puis noté pour chaque

27. Rappelons que ces résultats sont spécifiques à notre fragment de grammaire de test, et qu'ils ne sont pas nécessairement généralisables à toutes les grammaires polarisées envisageables. Nous nous attendons tout de même à des résultats similaires puisque nos méthodes de filtrage sont très génériques.

pile si elle était adéquate et le temps requis pour le déterminer. Chaque pile a ensuite été évaluée et le temps requis pour cette évaluation, réussie ou non, a été mesuré.

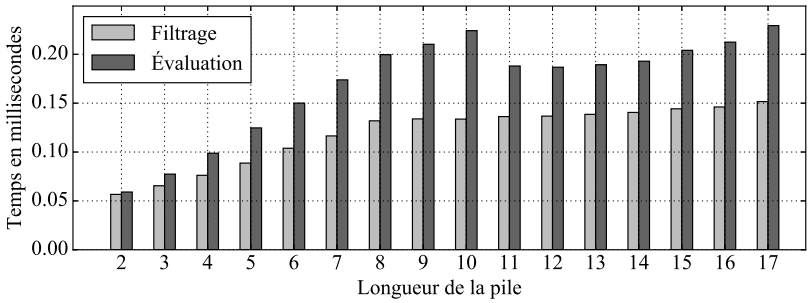


Figure 43 – Traitement des piles impliquant des paires indirectes non viables

La figure 43 examine les piles impliquant l'unification de paires indirectes non viables et dont l'évaluation échoue donc systématiquement. Le temps de filtrage ne représente ici qu'une fraction du temps d'évaluation, et ce, peu importe la longueur de la pile [28], cette méthode de filtrage est donc utile même si elle retire très peu de piles.

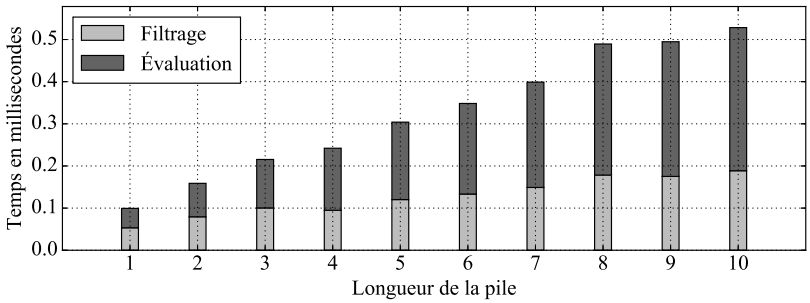


Figure 44 – Traitement des piles qui passent tous les filtres

La figure 44 montre les résultats du test pour les piles qui passent tous les filtres. Toutes ces piles donnent des structures valides dans le cadre de nos tests, c'est-à-dire qu'aucune d'elles n'échoue. Comme ces piles sont d'abord filtrées *et* ensuite évaluées lors d'une utilisation réelle de l'outil, ces deux temps s'additionnent tandis que le traitement des autres piles cesse dès qu'elles sont retirées par un filtre. Ce temps de traitement supplémentaire

28. Il faut prendre ces résultats avec réserve ; le nombre de paires dans une pile n'est pas toujours égal au nombre de paires unifiées en cours d'évaluation. Nous réglons en partie ce problème en retirant les piles implicites, mais on ne peut pas prédire le nombre total de paires unifiées avant d'évaluer une pile.

n'est cependant pas un problème car les piles qui passent tous les filtres sont typiquement très peu nombreuses, comme le montre la figure 42.

4.3 Structures valides et uniques

Notre dernier test visait à évaluer l'efficacité de nos méthodes de filtrage en calculant cette fois-ci le nombre de structures valides obtenues à partir des piles retenues, ainsi que le nombre de structures uniques parmi celles-ci. Ce test évaluait donc d'une part notre approche basée sur les paires non viables, et d'autre part, celle basée sur les piles implicites, qui vise à éviter les structures résultantes identiques.

Pour ce test, nous avons placé les règles de la grammaire dans deux groupes, soit un groupe de structures comprenant des objets sémantiques et un autre, des objets syntaxiques. Les règles d'interface se trouvaient donc dans les deux groupes. Nous avons divisé les règles de cette manière puisqu'une règle sémantique et une règle syntaxique ne partagent aucun type d'objet et ne peuvent donc jamais se combiner. Pour chaque groupe, nous avons ensuite combiné chaque paire et noté pour chacune d'elles le nombre de piles évaluées, le nombre de structures résultantes et le nombre de structures uniques parmi celles-ci.

Nous retrouvons les résultats dans les tableaux 11 et 12 ci-dessous. Par exemple, la première ligne indique qu'en combinant la structure SEMR1 avec elle-même, 255 piles sont évaluées, que l'évaluation de toutes ces piles est réussie mais que nous n'obtenons que 5 structures uniques. Rappelons que SEMR1, SEMSYNTR3, etc. sont des identifiants utilisés dans nos trois tables de règles (pp. 68-70). Par souci de concision, nous avons retiré les combinaisons où zéro ou une pile était évaluée.

Structure A	Structure B	Piles évaluées	Structures valides	uniques
SEMR1	**SEMR1**	**255**	**255**	**5**
	SEMR2	3	3	3
	SEMR3	3	3	3
	SEMR4	24	24	20
	SEMR5	24	24	20
	SEMSYNTR1	3	3	3
	SEMSYNTR2	3	3	3
	SEMSYNTR3	**831**	**831**	**162**
	SEMSYNTR4	24	24	20
	SEMSYNTR5	24	24	20
SEMR2	SEMSYNTR3	3	3	3
SEMR3	SEMSYNTR3	3	3	3
SEMR4	SEMR4	7	7	2
	SEMR5	6	6	6
	SEMSYNTR3	24	24	20
	SEMSYNTR4	19	19	13
	SEMSYNTR5	6	6	6
SEMR5	SEMR5	7	7	2
	SEMSYNTR3	24	24	20
	SEMSYNTR4	6	6	6
	SEMSYNTR5	19	19	13
SEMSYNTR1	SEMSYNTR3	24	24	23
	SEMSYNTR4	4	4	4
SEMSYNTR2	SEMSYNTR3	24	24	23
	SEMSYNTR4	4	4	4
SEMSYNTR3	**SEMSYNTR4**	**1378**	**1378**	**810**
	SEMSYNTR5	199	199	167
SEMSYNTR4	**SEMSYNTR4**	**214**	**214**	**5**
	SEMSYNTR5	13	13	13
SEMSYNTR5	SEMSYNTR5	7	7	2

Tableau 11 – Combinaison des structures comprenant des objets sémantiques

Structure A	Structure B	Piles évaluées	Structures valides	uniques
SEMSYNTR1	SYNTR7	3	3	3
	SYNTR8	3	3	3
SEMSYNTR2	SYNTR7	3	3	3
	SYNTR8	3	3	3
SEMSYNTR3	SYNTR1	7	7	7
	SYNTR2	3	3	3
	SYNTR3	3	3	3
	SYNTR4	79	79	38
	SYNTR5	79	79	38
	SYNTR6	8	8	8
	SYNTR7	27	27	23
	SYNTR8	79	79	38
	SYNTR9	24	24	20
	SYNTR10	7	7	7
	SYNTR11	79	79	38
	SYNTR12	3	3	3
	SYNTR13	7	7	7
SEMSYNTR4	SYNTR3	3	3	3
	SYNTR4	6	6	6
	SYNTR5	6	6	6
	SYNTR6	6	6	6
	SYNTR7	6	6	6
	SYNTR8	15	15	11
	SYNTR9	19	19	13
	SYNTR11	6	6	6
SEMSYNTR5	SYNTR4	3	3	3
	SYNTR5	3	3	3
	SYNTR8	3	3	3
	SYNTR11	3	3	3
	SYNTR13	2	2	2

Structure A	Structure B	Piles évaluées	Structures valides	uniques
SYNTR1	SYNTR4	3	3	3
	SYNTR5	3	3	3
	SYNTR8	3	3	3
	SYNTR11	3	3	3
SYNTR2	SYNTR7	5	5	5
	SYNTR8	3	3	3
SYNTR3	SYNTR8	3	3	3
	SYNTR9	3	3	3
SYNTR4	SYNTR4	15	15	2
	SYNTR5	15	15	11
	SYNTR6	2	2	2
	SYNTR7	6	6	6
	SYNTR8	15	15	11
	SYNTR9	6	6	6
SYNTR5	SYNTR5	15	15	2
	SYNTR6	2	2	2
	SYNTR7	7	7	7
	SYNTR8	19	19	13
	SYNTR9	6	6	6
SYNTR6	SYNTR6	3	3	1
	SYNTR7	5	5	5
	SYNTR8	8	8	8
	SYNTR9	6	6	6
SYNTR7	SYNTR7	7	7	2
	SYNTR8	19	19	13
	SYNTR9	6	6	6
SYNTR8	SYNTR8	19	19	2
	SYNTR9	3	3	3
SYNTR9	SYNTR9	7	7	2

Structure A	Structure B	Piles évaluées	Structures valides	uniques
SYNTR10	SYNTR4	3	3	3
	SYNTR5	3	3	3
	SYNTR8	3	3	3
	SYNTR11	3	3	3
SYNTR11	SYNTR4	15	15	11
	SYNTR5	19	19	13
	SYNTR6	2	2	2
	SYNTR7	7	7	7
	SYNTR8	19	19	13
	SYNTR9	6	6	6
	SYNTR11	7	7	2
	SYNTR13	3	3	3
SYNTR12	SYNTR7	3	3	3
SYNTR13	SYNTR4	3	3	3
	SYNTR5	3	3	3
	SYNTR7	2	2	2
	SYNTR8	8	8	6
	SYNTR9	2	2	2

Tableau 12 – Combinaison des structures comprenant des objets syntaxiques

Nous avons identifié en gras quelques résultats avec un nombre très bas de structures uniques comparativement au nombre de structures obtenues. Dans le cas de la combinaison de la structure SEMR1 avec elle-même, les structures uniques représentent moins de 2% des structures valides obtenues. Pour la combinaison de SEMR1 avec SEMSYNTR3, ou celle de SEMSYNTR3 avec SEMSYNTR4, il n'y a pas que ce ratio qui surprend, mais aussi le nombre de structures uniques (respectivement, 162 et 810).

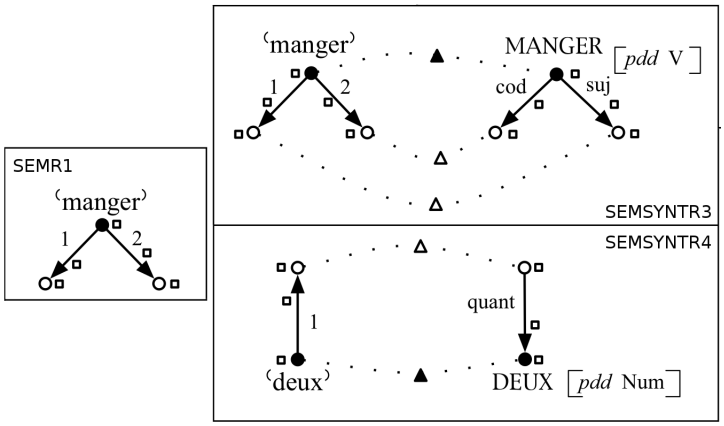


Figure 45 – Trois règles problématiques

Nous reproduisons ci-haut les structures impliquées. Le problème avec ces structures est le nombre élevé d'objets et l'absence d'information sur la plupart des nœuds (mis à part le type et les polarités blanches). En combinant SEMR1 et SEMSYNTR3, par exemple, le premier actant sémantique de ⟨manger⟩ dans SEMR1 peut donc s'unifier avec les premier et second actants sémantiques dans SEMSYNTR3, avec ⟨manger⟩ lui-même, et bien plus encore. Les nœuds « vides » sont donc problématiques puisqu'ils peuvent s'unifier avec n'importe quel autre objet du même type (*nœudsém* ou *nœudsynt*), ce qui réduit entre autres l'efficacité du filtrage des paires non viables. Ces nœuds permettent également à la structure de se « replier » sur elle-même, c'est-à-dire que ⟨manger⟩ peut être ses propres premier et second actants sémantiques dans une structure résultante.

Le problème ne relève donc pas de notre implémentation mais plutôt du fragment de grammaire que nous avons encodé. Dans une grammaire GUST plus complète, on peut donner à chaque actant sémantique une étiquette pour contraindre son sens ; le premier actant de ⟨manger⟩ serait par exemple étiqueté ⟨être vivant⟩ et le second ⟨aliment⟩. Dans notre fragment de grammaire, cependant, les actants n'ont pas été typés. Nous n'aborderons pas cette solution ici mais le lecteur est invité à consulter Lareau (2007, 2008).

Les résultats de ce test montrent que notre filtrage permet d'éliminer complètement les piles qui échouent avant même de les évaluer (dans nos tests du moins) mais que nous ne parvenons pas encore à identifier toutes les piles qui donnent des résultats identiques.

Chapitre 5. Discussion et conclusion

Dans ce dernier chapitre, nous discutons de quelques problèmes liés à notre implémentation des grammaires d'unification polarisées et à la grammaire d'unification Sens-Texte. Nous effectuons ensuite un retour sur ce mémoire.

5.1 Problèmes liés à la grammaire d'unification Sens-Texte

Un premier problème apparent de GUST est que les structures peuvent parfois se combiner de manière contre-intuitive. Dans la grammaire sémantique, par exemple, on s'attend à ce que l'unification de deux nœuds sémantiques, s'ils sont tous deux la source d'un arc étiqueté 1, entraîne également celle de leur premier actant sémantique respectif. Aucune fonction structurante ne lie cependant un prédicat à ses actants, et ce n'est donc qu'en unifiant une paire d'arcs sémantiques qu'on assure que des prédicats sont unifiés en même temps que leurs actants.

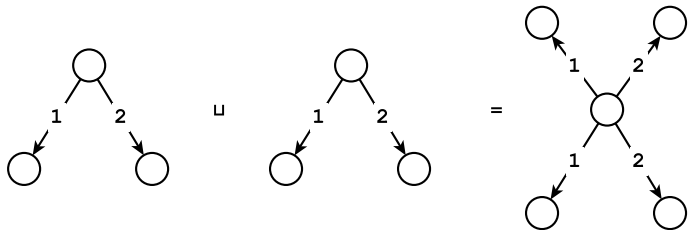


Figure 46 – Une combinaison possible de deux structures sémantiques

On peut donc obtenir la troisième structure de la figure 46 à partir des deux premières, alors qu'intuitivement, nous aimerions exclure ce genre de résultat. La polarisation des objets ne règle qu'en partie le problème—on ne peut empêcher ce résultat qu'en polarisant les deux nœuds sémantiques en noir.

Une solution possible serait de lier un prédicat à ses actants à l'aide de fonctions structurantes 1, 2, etc. L'unification de deux prédicats déclencherait donc toujours celle de leurs actants. Cette solution ne se prête malheureuement pas à une description linguistique plus

complexe puisqu'elle ne permet pas, notamment, d'appliquer certaines règles comme celle du réancrage des gouverneurs (Lareau, 2007, 2008) que nous avons reproduite dans la figure 47 [29]. Cette règle est particulière puisqu'un de ses nœuds est la source de deux arcs sémantiques portant la même étiquette. Son application forcerait donc l'unification d'un sémantème avec son genre prochain, ce qui n'est pas souhaitable. Pour une description plus complète de cette règle et de son usage, voir Lareau (2007, 2008).

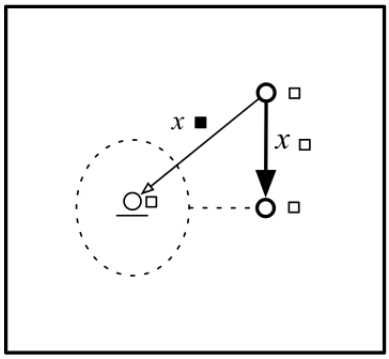


Figure 47 – La règle de réancrage des gouverneurs de Lareau (2007, 2008).

Une question qui s'impose est la suivante : la grammaire doit-elle à elle seule prévenir la génération de telles structures indésirables, neutres ou non, ou est-il acceptable de fournir des outils externes au formalisme pour limiter la capacité générative de la grammaire ? Nous n'abordons pas cette question, mais nous discutons d'une fonctionnalité qui pourrait régler partiellement ce problème dans la section suivante.

Le second problème avec GUST est que les règles de correspondance ne transforment pas une structure sémantique en structure syntaxique ; elles ajoutent simplement des objets à une structure polarisée existante. Ce problème, évidemment, est dû au fonctionnement des grammaires d'unification polarisées, mais n'en est pas moins dérangeant.

Même si les paires d'objets non viables sont retirées au préalable, la combinaison d'une règle d'interface avec une structure qui comprend des liens de correspondance devient problématique puisqu'il faut envisager toutes les combinaisons des objets sémantiques

29. On parle plutôt de « propagation des gouverneurs » dans Lareau (2007), mais la règle est la même.

des deux structures, puis des objets syntaxiques, ce qui peut faire exploser le nombre de piles à envisager.

Les règles d'interface ont le même problème que les structures sémantiques, c'est-à-dire qu'aucune fonction structurante ne lie un signifié à son signifiant. Cela signifie qu'une même paire d'objets sémantique et syntaxique peuvent se retrouver avec plusieurs liens de correspondance entre eux, alors qu'on s'attend à que les signifiants de nœuds sémantiques soient également unifiés automatiquement. Comme un même nœud sémantique peut correspondre à plus d'un nœud syntaxique (par exemple, ⁽patate⁾ a trois correspondants en syntaxe, soit POMME, DE et TERRE), ce problème est particulièrement complexe.

5.2 Problèmes liés à notre implémentation et fonctionnalités manquantes

Un problème important avec l'outil est qu'il ne gère pas toujours de manière satisfaisante la combinaison de structures avec de nombreux objets. Même si le filtrage des paires non viables permet typiquement d'éliminer 75 à 100% des paires et réduit drastiquement le nombre de piles (cf. § 4.1, p. 70), le nombre de paires retenues est parfois si élevé qu'on ne peut pas traiter toutes les piles dans un temps raisonnable.

Si on tente de combiner la structure SEMSYNTR3 avec elle-même, par exemple, nous parvenons à retirer 145 paires non viables sur un total de 169 paires. Nous avons donc près de 17 millions de piles à filtrer. Si nous considérons que le filtrage de chaque pile prend 0.05 millisecondes (la plus petite mesure obtenue lors de nos tests, pour une pile de deux paires), alors il pourrait nous prendre 15 minutes juste pour filtrer les piles.

Même si une part de responsabilité nous revient, puisque nos règles ne sont pas toujours suffisamment spécifiques, nous aimerions que l'outil gère ces combinaisons problématiques plus efficacement, quitte à n'évaluer qu'une partie des piles. Comme nos méthodes de filtrage n'éliminent qu'en partie les piles qui donnent des doublons, nous pourrions possiblement obtenir la plupart des structures résultantes à partir d'un sous-ensemble de piles. Ultimement, notre outil ne peut cependant pas faire le travail du chercheur.

5.2.1 Quelques fonctionnalités manquantes

Nous identifions ici quelques fonctionnalités qu'il serait intéressant d'ajouter dans une version future de notre outil. Trois de ces fonctionnalités sont liées à la performance de l'outil et une est en lien avec son usabilité. Cette liste n'est bien sûr pas exhaustive.

Interface graphique Notre outil permet seulement de visualiser les structures en mode texte. Cette représentation textuelle est difficile à interpréter, surtout lorsque la structure comporte un nombre élevé d'objets. Il serait donc souhaitable d'implémenter éventuellement une interface de visualisation graphique des structures.

Puisque certains formalismes comme GUST sont dotés d'une notation plus concise, il serait également utile de permettre à l'utilisateur de personnaliser l'affichage d'un objet selon la valeur d'une fonction, par exemple. Un objet de type *nœudsém* serait par exemple représenté sous forme de nœud et sa couleur dépendrait de sa polarité principale.

L'interface pourrait également servir à éliminer manuellement des structures suite à une combinaison, à créer et modifier des structures et tout simplement à rendre l'outil plus accessible pour un utilisateur moins habile avec les ordinateurs, par exemple.

Partage de structures Par souci de simplicité, nous avons implémenté un algorithme d'unification sans partage de structures. Cette fonctionnalité nous permettrait d'évaluer paresseusement les structures résultantes suite à une combinaison, c'est-à-dire que nous pourrions retourner une référence aux deux structures originales et les dictionnaires `compl` et `fwtab` pour chaque pile, et ne bâtir la structure résultante que si nécessaire.

Parallélisation La parallélisation nous permet d'exécuter plusieurs tâches simultanément. Le processus de génération des piles retourne présentement une pile à évaluer et son exécution est interrompue pendant l'évaluation de la pile. Il faut ensuite déterminer si la pile suivante est à évaluer, et le cas échéant, patienter pendant qu'on l'évalue, et ainsi de suite, jusqu'à ce que les piles soient épuisées.

La parallélisation pourrait nous permettre de faire trois choses :

— Générer les piles en arrière-plan, de manière à ce que le processus se poursuive même lorsqu'une pile est évaluée ;

— Unifier plus d'une paire d'objets à la fois pour accélérer l'évaluation d'une pile ;

— Évaluer plus d'une pile à la fois pour accélérer le processus de combinaison.

Comme notre outil est encore un prototype et que Python n'est pas nécessairement idéal pour la parallélisation à cause du Global Interpreter Lock (GIL) (Beazley, 2009), nous n'avons pas cherché à paralléliser le processus de combinaison. Nous croyons qu'il serait plus judicieux de perfectionner d'abord le filtrage des piles.

Élimination automatique de certains résultats L'outil retourne souvent beaucoup de structures suite à une combinaison. Cela n'est pas nécessairement un problème dans un cadre d'expérimentation, mais peut être un obstacle majeur lorsqu'on tente, par exemple, de générer toutes les structures syntaxiques bien formées pour une structure sémantique donnée en GUST, puisque le nombre de structures augmente très rapidement.

Il serait intéressant d'explorer différentes méthodes pour permettre à l'outil de choisir par lui-même si une structure est à conserver ou non, à l'aide de l'apprentissage machine ou en fournissant des outils supplémentaires pour filtrer les structures. Nous pourrions alors spécifier qu'un même nœud sémantique ne peut pas être la source de plus d'un arc portant la même étiquette, sauf dans un contexte très particulier [30]. La structure résultante de la figure 46 serait donc exclue automatiquement. Avec l'apprentissage machine, nous pourrions supprimer manuellement des structures suite à des combinaisons pour entraîner l'outil à mener la même tâche pour une grammaire donnée.

5.3 Conclusion

Nous avons présenté dans ce mémoire un outil qui facilite la conception et la validation de grammaires d'unification polarisées. Notre but était de créer un outil qui se prête à la

30. Cela nous permettrait entre autres d'appliquer la règle de réancrage des gouverneurs (Lareau, 2007). Il faudrait cependant également que l'outil permette le partage d'une étiquette entre plusieurs objets ou l'utilisation de variables. Ces fonctionnalités ne sont pas disponibles présentement.

validation du formalisme GUST tout en demeurant générique et flexible.

Notre plus grande préoccupation était en lien avec la combinaison des structures polarisées ; comme celles-ci sont des ensembles de structures de traits, il existe de nombreuses façons de combiner deux structures.

Nous avons décrit trois méthodes pour éliminer rapidement des combinaisons qui échouent ou donnent des structures identiques, avant même d'évaluer ces combinaisons. Ensemble, ces méthodes permettent d'éliminer plus de 99% des combinaisons dans certains cas, mais l'efficacité varie selon les paires fonction-valeur portées par les objets. Tandis que nous réussissons à éliminer toutes les combinaisons qui échouent, nous ne parvenons toujours pas à détecter toutes les combinaisons qui donnent des structures identiques et nous évaluons donc encore inutilement certaines combinaisons.

Pour évaluer les combinaisons restantes, nous avons implémenté un algorithme d'unification polarisée basé sur l'algorithme de Tomabechi (1991). Nous avons sélectionné un algorithme non destructif, c'est-à-dire que les structures originales ne sont pas modifiées, puisqu'on souhaite parfois utiliser une même structure plus d'une fois. L'algorithme élimine aussi complètement la copie précoce et la copie excessive.

Nous avons aussi implémenté certaines modifications suggérées par van Lohuizen (2000) pour simplifier l'algorithme et rendre possible la parallélisation éventuelle du processus de combinaison.

Notre implémentation des grammaires d'unification polarisées n'est encore qu'un prototype et le langage de programmation choisi pour concevoir l'outil (Python) reflète notre préférence mais n'est peut-être pas le meilleur choix avec le recul, puisqu'il se prête mal à la récursivité et au parallélisme. Néanmoins, notre prototype fonctionne et nous espérons qu'il sera utile aux informaticiens et linguistes, Sens-Texte ou autres.

Annexe A. Définition de notre fragment de grammaire

Nous reproduisons ici le contenu du fichier où est encodé notre fragment de grammaire GUST. Le format de ce fichier est décrit au chapitre 3 (cf. § 3.2, p. 37), et les structures définies correspondent à celles illustrées au début du chapitre 4 (p. 68).

```
structuring { source cible hôte signifié signifiant mode temps pers nbre }
polarizing { psém psémsynt psynt }
labeling { type étiquette pdd }

polarities { blanc noir }
neutral { noir }
product_table { ( blanc blanc blanc ) ( blanc noir noir ) }

SEMR1 {
    manger { type=noeudsém, étiquette=manger, psém=noir, psémsynt=blanc }
    N1 { type=noeudsém, psém=blanc, psémsynt=blanc }
    N2 { type=noeudsém, psém=blanc, psémsynt=blanc }
    E1 { type=arcsém, étiquette=1, psém=noir, psémsynt=blanc, source=manger, cible=N1 }
    E2 { type=arcsém, étiquette=2, psém=noir, psémsynt=blanc, source=manger, cible=N2 }
}

SEMR2 { Pierre { type=noeudsém, étiquette=Pierre, psém=noir, psémsynt=blanc } }

SEMR3 { pomme { type=noeudsém, étiquette=pomme, psém=noir, psémsynt=blanc } }

SEMR4 {
    deux { type=noeudsém, étiquette=deux, psém=noir, psémsynt=blanc }
    N1 { type=noeudsém, psém=blanc, psémsynt=blanc }
    E1 { type=arcsém, étiquette=1, psém=noir, psémsynt=blanc, source=deux, cible=N1 }
}

SEMR5 {
    présent { type=noeudsém, étiquette=présent, psém=noir, psémsynt=blanc }
    N1 { type=noeudsém, psém=blanc, psémsynt=blanc }
    E1 { type=arcsém, étiquette=1, psém=noir, psémsynt=blanc, source=présent, cible=N1 }
}

SYNTR1 {
    N1 { type=noeudsynt, pdd=V, mode=mode, psynt=blanc, psémsynt=blanc }
    mode { type=grammème, étiquette=ind, hôte=N1, psynt=blanc, psémsynt=blanc }
}
```

```
SYNTR2 {
    pomme { type=noeudsynt, étiquette=POMME, psynt=noir, psémsynt=blanc, pdd=N, nbre=nbre }
    nbre { type=grammème, hôte=pomme, psynt=blanc, psémsynt=blanc }
}

SYNTR3 { deux { type=noeudsynt, étiquette=DEUX, psynt=noir, psémsynt=blanc, pdd=Num } }

SYNTR4 {
    N1 { type=noeudsynt, pdd=V, psynt=blanc, psémsynt=blanc }
    cod { type=arcsynt, étiquette=cod, source=N1, cible=N2, psynt=noir, psémsynt=blanc }
    N2 { type=noeudsynt, psynt=blanc, psémsynt=blanc }
}

SYNTR5 {
    N1 { type=noeudsynt, pdd=V, mode=mode, psynt=blanc, psémsynt=blanc }
    mode { type=grammème, étiquette=ind, hôte=N1, psynt=blanc, psémsynt=blanc }
    suj { type=arcsynt, étiquette=suj, source=N1, cible=N2, psynt=noir, psémsynt=blanc }
    N2 { type=noeudsynt, psynt=blanc, psémsynt=blanc }
}

SYNTR6 {
    N1 { type=noeudsynt, pdd=N, psynt=blanc, psémsynt=blanc }
    quant { type=arcsynt, étiquette=quant, source=N1, cible=N2, psynt=noir, psémsynt=blanc }
    N2 { type=noeudsynt, pdd=Num, psynt=blanc, psémsynt=blanc }
}

SYNTR7 {
    N1 { type=noeudsynt, pers=pers, psynt=blanc, psémsynt=blanc }
    pers { type=grammème, étiquette=3, hôte=N1, psynt=noir, psémsynt=noir }
    suj { type=arcsynt, étiquette=suj, source=N1, cible=N2, psynt=blanc, psémsynt=blanc }
    N2 { type=noeudsynt, pdd=N, psynt=blanc, psémsynt=blanc }
}

SYNTR8 {
    N1 { type=noeudsynt, nbre=nbre, psynt=blanc, psémsynt=blanc }
    nbre { type=grammème, étiquette=sing, hôte=N1, psynt=noir, psémsynt=noir }
    suj { type=arcsynt, étiquette=suj, source=N1, cible=N2, psynt=blanc, psémsynt=blanc }
    N2 { type=noeudsynt, nbre=nbre2, psynt=blanc, psémsynt=blanc }
    nbre2 { type=grammème, étiquette=sing, hôte=N2, psynt=blanc, psémsynt=blanc }
}
```

```
SYNTR9 {
    N1 { type=noeudsynt, nbre=nbre, psynt=blanc, psémsynt=blanc }
    nbre { type=grammème, étiquette=plur, hôte=N1, psynt=noir, psémsynt=noir }
    quant { type=arcsynt, étiquette=quant, source=N1, cible=N2, psynt=blanc, psémsynt=blanc }
    N2 { type=noeudsynt, pdd=Num, psynt=blanc, psémsynt=blanc }
}

SYNTR10 {
    N1 { type=noeudsynt, pdd=V, mode=mode, temps=temps, psynt=blanc, psémsynt=blanc }
    mode { type=grammème, étiquette=ind, hôte=N1, psynt=blanc, psémsynt=blanc }
    temps { type=grammème, étiquette=présent, hôte=N1, psynt=noir, psémsynt=blanc }
}

SYNTR11 {
    N1 { type=noeudsynt, pdd=V, mode=mo, temps=tem, pers=per, nbre=nb, psynt=blanc, psémsynt=blanc }
    mo { type=grammème, étiquette=ind, hôte=N1, psynt=noir, psémsynt=blanc }
    tem { type=grammème, hôte=N1, psynt=blanc, psémsynt=blanc }
    per { type=grammème, hôte=N1, psynt=blanc, psémsynt=blanc }
    nb { type=grammème, hôte=N1, psémsynt=blanc, psynt=blanc }
    suj { type=arcsynt, étiquette=suj, source=N1, cible=N2, psynt=blanc, psémsynt=blanc }
    N2 { type=noeudsynt, psynt=blanc, psémsynt=blanc }
}

SYNTR12 {
    Pierre { type=noeudsynt, étiquette=PIERRE, pdd=N, nbre=nbre, psynt=noir, psémsynt=blanc }
    nbre { type=grammème, étiquette=sing, hôte=Pierre, psynt=noir, psémsynt=noir }
}

SYNTR13 {
    manger { type=noeudsynt, étiquette=MANGER, pdd=V, mode=mode, psynt=noir, psémsynt=blanc }
    mode { type=grammème, hôte=manger, psynt=blanc, psémsynt=blanc }
}

SEMSYNTR1 {
    Pierre { type=noeudsém, étiquette=Pierre, psém=blanc, psémsynt=noir }
    PIERRE { type=noeudsynt, étiquette=PIERRE, pdd=N, psémsynt=noir, psynt=blanc }
    corr { type=correspondance, psémsynt=noir, signifié=Pierre, signifiant=PIERRE }
}
```

```
SEMSYNTR2 {
    pomme { type=noeudsém, étiquette=pomme, psém=blanc, psémsynt=noir }
    POMME { type=noeudsynt, étiquette=POMME, pdd=N, psémsynt=noir, psynt=blanc }
    corr { type=correspondance, psémsynt=noir, signifié=pomme, signifiant=POMME }
}

SEMSYNTR3 {
    manger { type=noeudsém, étiquette=manger, psém=blanc, psémsynt=noir }
    N1 { type=noeudsém, psém=blanc, psémsynt=blanc }
    N2 { type=noeudsém, psém=blanc, psémsynt=blanc }
    E1 { type=arcsém, étiquette=1, source=manger, cible=N1, psém=blanc, psémsynt=noir }
    E2 { type=arcsém, étiquette=2, source=manger, cible=N2, psém=blanc, psémsynt=noir }
    MANGER { type=noeudsynt, étiquette=MANGER, pdd=V, psémsynt=noir, psynt=blanc }
    N1SYNT { type=noeudsynt, psémsynt=blanc, psynt=blanc }
    N2SYNT { type=noeudsynt, psémsynt=blanc, psynt=blanc }
    cod { type=arcsynt, étiquette=cod, source=MANGER, cible=N1SYNT, psémsynt=noir, psynt=blanc }
    suj { type=arcsynt, étiquette=suj, source=MANGER, cible=N2SYNT, psémsynt=noir, psynt=blanc }
    corr1 { type=correspondance, signifié=manger, signifiant=MANGER, psémsynt=noir }
    corr2 { type=correspondance, signifié=N2, signifiant=N1SYNT, psémsynt=blanc }
    corr3 { type=correspondance, signifié=N1, signifiant=N2SYNT, psémsynt=blanc }
}

SEMSYNTR4 {
    deux { type=noeudsém, étiquette=deux, psémsynt=noir, psém=blanc }
    N1 { type=noeudsém, psém=blanc, psémsynt=blanc }
    E1 { type=arcsém, étiquette=1, source=deux, cible=N1, psém=blanc, psémsynt=noir }
    DEUX { type=noeudsynt, étiquette=DEUX, pdd=Num, psémsynt=noir, psynt=blanc }
    N1SYNT { type=noeudsynt, psémsynt=blanc, psynt=blanc }
    quant { type=arcsynt, étiquette=quant, source=N1SYNT, cible=DEUX, psémsynt=noir, psynt=blanc }
    corr1 { type=correspondance, psémsynt=blanc, signifié=N1, signifiant=N1SYNT }
    corr2 { type=correspondance, psémsynt=noir, signifié=deux, signifiant=DEUX }
}

SEMSYNTR5 {
    présent { type=noeudsém, étiquette=présent, psém=blanc, psémsynt=noir }
    N1 { type=noeudsém, psém=blanc, psémsynt=blanc }
    E1 { type=arcsém, étiquette=1, source=présent, cible=N1, psém=blanc, psémsynt=noir }
    N1SYNT { type=noeudsynt, pdd=V, temps=temps, psémsynt=blanc, psynt=blanc }
    temps { type=grammème, étiquette=présent, hôte=N1SYNT, psémsynt=noir, psynt=blanc }
}
```

Annexe B. Décomposition du système de polarités

On peut concevoir la valeur d'une fonction de polarisation comme une cavité où s'insérent des pièces représentant chacune une polarité [31]. Une cavité vide représente l'absence complète de saturation et une cavité pleine, la saturation complète. Pour se combiner, deux polarités doivent donc être représentées par des pièces qui peuvent s'emboîter.

Dans un système à deux polarités bien formé, par exemple, nous n'avons que deux pièces, soit une pièce « vide » pour la polarité non neutre et une pièce pleine pour la polarité neutre. Avec cette configuration, une pièce vide peut se combiner avec une pièce vide ou pleine, tandis que deux pièces pleines ne peuvent pas se combiner.

Plus concrètement, chaque polarité ou pièce correspond à un **vecteur binaire**, soit une séquence de zéros et de uns. Le produit de deux polarités consiste à additionner leurs vecteurs, soit à additionner ensemble les premiers éléments, puis les seconds, et ainsi de suite, pour obtenir un nouveau vecteur. Il semblerait alors plus approprié d'appeler la loi de composition sur les polarités la « somme » plutôt que le produit.

k	2	3–4	5–8	9–16	17–32	33–49
éléments	1	2	3	4	5	6

Tableau 13 – Nombre d'éléments par vecteur selon le nombre de polarités

Le nombre d'éléments requis pour que chaque polarité ait un vecteur unique est égal à $\lceil \log_2 k \rceil$, soit le logarithme à base 2 du nombre k de polarités, arrondi vers le haut. Ainsi, pour un système à quatre polarités, il faut des vecteurs à 2 éléments pour distinguer les polarités : [0, 0], [0, 1], [1, 0] et [1, 1]. Pour un système à 12 polarités, il nous faut des vecteurs à 4 éléments, mais quatre de ces vecteurs sont de trop.

Le vecteur assigné à chaque polarité dépend du produit ; pour chaque paire de polarités, la somme des vecteurs assignés aux polarités doit être égale au vecteur assigné à la polarité

31. L'idée nous vient de François Lareau, qui a élaboré ce concept avec Sylvain Kahane.

résultante. Si la somme des vecteurs n'est pas binaire—elle contient des 2—alors il ne devrait pas être possible de combiner ces polarités.

·	A	B	C	D	E	F	G	H
A	A	B	C	D	E	F	G	H
B		⊥	D	⊥	F	⊥	H	⊥
C			⊥	⊥	G	H	⊥	⊥
D				⊥	H	⊥	⊥	⊥
E					⊥	⊥	⊥	⊥
F						⊥	⊥	⊥
G							⊥	⊥
H								⊥

Polarité	vecteur
A	[0, 0, 0]
B	[0, 0, 1]
C	[0, 1, 0]
D	[0, 1, 1]
E	[1, 0, 0]
F	[1, 0, 1]
G	[1, 1, 0]
H	[1, 1, 1]

Tableau 14 – Produit et décomposition d'un système à huit polarités

Le système de polarités du tableau 14 a par exemple huit polarités. Les vecteurs doivent être de longueur $\lceil \log_2 8 \rceil = 3$ pour que chaque polarité ait un vecteur unique. Nous voyons que le produit de A et A donne A, et le seul vecteur qui reste le même lorsqu'additionné avec lui-même ne contient que des zéros. On assigne donc le vecteur [0, 0, 0] à A. Pour les autres polarités, il faut procéder par recherche exhaustive (*brute force*) en assignant à chaque polarité un des vecteurs et en vérifiant que pour chaque paire de polarités dont le produit est défini, la somme de leurs vecteurs donne le vecteur assigné à leur produit.

Sans entrer dans les détails, nous pouvons trouver une configuration valide à l'aide de la programmation par contraintes. Dans le cadre de nos expérimentations, nous avons utilisé le module `python-constraint` et formulé deux contraintes :

— Pour chaque paire de polarités, les vecteurs assignés doivent être différents. Nous nous assurons ainsi que chaque polarité reçoit un vecteur unique ;

— Pour chaque produit défini, la somme des vecteurs assignés aux polarités originales doit être égale au vecteur assigné à leur produit.

Six configurations respectent les deux contraintes ; nous en montrons une ci-haut.

Cette manière de concevoir les polarités a certaines implications. On peut par exemple supposer que si deux polarités n'ont pas de produit (leur produit échoue), alors ces polarités ont des vecteurs incompatibles, c'est-à-dire qu'il existe au moins un 1 dans une même position dans les deux vecteurs. Le vecteur qui résulte lorsqu'on additionne ces vecteurs n'est alors plus binaire puisqu'il contient un 2.

Cette prémisse nous permet d'exclure certaines paires d'objets lors de la génération des piles d'unification : si les objets portent des polarités incompatibles, ces polarités seront *toujours* incompatibles car le produit de vecteurs binaires ne peut qu'ajouter des 1 ; les 1 qui « bloquent » le produit seront toujours présents. Nous supposons que cette prémisse est vraie et une paire d'objets avec des polarités incompatibles est donc marquée comme non viable dans notre outil ; notre outil ne tente cependant pas présentement d'assigner des vecteurs binaires aux polarité et considère que deux polarités sont incompatibles si leur produit n'a pas été défini.

Cette conception n'est pas parfaite—certains systèmes de polarités ne semblent pas se prêter à cette décomposition en vecteurs binaires. Ces systèmes de polarités ont peut-être des propriétés que nous ne connaissons pas encore. Il serait certainement intéressant d'explorer davantage cette approche dans des travaux futurs. Nous pourrions par exemple autocompléter un système de polarités à partir d'une description partielle du produit—en observant par exemple que la somme des vecteurs de deux polarités sans produit défini donne le vecteur d'une autre polarité—ou détecter qu'un système est problématique s'il n'existe aucune configuration où un produit défini est possible.

Bibliographie

Abeillé, A. (2007). *Les grammaires d'unification*. Paris, France : Lavoisier.

Beazley, D. (2009). *Inside the Python GIL.* Communication présentée lors du Python Concurrency Workshop, Chicago, Illinois.

Bird, S., Klein, E., & Loper, E. (2009). *Natural language processing with Python* (1re éd.). O'Reilly.

Bohnet, B., Langjahr, A., & Wanner, L. (2000). A development environment for an MTT-based sentence generator. Dans *Proceedings of the First International Conference on Natural Language Generation (INLG 2000)* (pp. 260–263). Mitzpe Ramon, Israël.

Boyer, R. S., & Moore, J. S. (1972). The sharing of structure in theorem-proving programs. Dans B. Meltzer & D. Michie (Eds.), *Machine intelligence* (Vol. 7, pp. 101–116). New York : John Wiley and Sons.

Carpenter, B. (1992). *The logic of typed feature structures*. Cambridge, Angleterre : Cambridge University Press.

Cohen-Sygal, Y., & Wintner, S. (2007). The non-associativity of polarized tree-based grammars. Dans *Proceedings of the 8th International Conference on Intelligent Text Processing and Computational Linguistics (CICLing-2007)* (pp. 208–217). Mexico, Mexique.

Copestake, A. (2002). *Implementing typed feature structures*. Stanford, Californie : CSLI Publications.

Debusmann, R. (2004). Multiword expressions as dependency subgraphs. Dans *Proceedings of the ACL 2004 Workshop on Multiword Expressions : Integrating Processing* (pp. 56–63).

Debusmann, R. (2006). *Extensible Dependency Grammar : a modular grammar formalism based on multigraph description* (Thèse de doctorat). Université de la Sarre, Sarrebruck, Allemagne.

Debusmann, R., Duchier, D., & Kruijff, G.-J. M. (2004). Extensible Dependency Grammar : a new methodology. Dans *Proceedings of the COLING 2004 Workshop on Recent Advances in Dependency Grammar* (pp. 78–84). Genève, Suisse.

Debusmann, R., Duchier, D., & Niehren, J. (2004). The XDG Grammar Development Kit. Dans *Proceedings of the 2nd International Conference on Mozart / Oz (MOZ 2004)* (pp. 188–199). Charleroi, Belgique.

Duchier, D., & Thater, S. (1999). Parsing with tree descriptions : a constraint-based approach. Dans *Proceedings of the 6th International Workshop on Natural Language Understanding and Logic Programming (NLULP 1999).* Las Cruces, Nouveau Mexique.

Emele, M. C. (1991). Unification with lazy non-redundant copying. Dans *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL 1991)* (pp. 323–330). Berkeley, Californie.

Francez, N., & Wintner, S. (2012). *Unification grammars*. Cambridge, Angleterre : Cambridge University Press.

Godden, K. (1990). Lazy unification. Dans *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics (ACL 1990)* (pp. 180—187). Pittsburgh, Pennsylvanie.

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using NetworkX. Dans G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th Python in Science Conference (SciPy 2008)* (pp. 11–15). Pasadena, Californie.

Iordanskaja, L., & Mel'čuk, I. (2000). The notion of surface-syntactic relation revisited (Valence-controlled surface-syntactic relations in French). Dans L. L. Iomdin &

L. P. Krysin (Eds.), *Slovo v tekste i v slovare* [Les mots dans le texte et dans le dictionnaire] (pp. 391–433). Moscou, Russie : Jazyki russkoj kul'tury.

Kahane, S. (2001). Grammaires de dépendance formelle et théorie Sens-Texte. Dans *Actes de la 8ème conférence sur le traitement automatique des langues naturelles (TALN 2001).* Tours, France.

Kahane, S. (2002). *Grammaire d'unification Sens-Texte : vers un modèle mathématique articulé de la langue* (Document de synthèse). Université de Paris 7.

Kahane, S. (2004). Grammaires d'unification polarisées. Dans *Actes de la 11ème conférence sur le traitement automatique des langues naturelles (TALN 2004)* (pp. 233–242). Fès, Maroc.

Kahane, S. (2005). Structure des représentations logiques, polarisation et sous-spécification. Dans *Actes de TALN 2005* (p. 163-173). Dourdan, France.

Kahane, S. (2006). Polarized unification grammars. Dans *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING / ACL 2006)* (pp. 137–144). Sidney, Australie.

Kahane, S., & Lareau, F. (2005a). Grammaire d'unification Sens-Texte : modularité et polarisation. Dans *Actes de la 12ème conférence sur le traitement automatique des langues naturelles (TALN 2005)* (pp. 23–32). Dourdan, France.

Kahane, S., & Lareau, F. (2005b). Meaning-Text Unification Grammar : modularity and polarization. Dans *Proceedings of the 2nd International Conference on Meaning–Text Theory (MTT 2005)* (pp. 163–173). Moscou, Russie.

Kahane, S., & Lareau, F. (2016a). Encoding a syntactic dictionary into a super granular unification grammar. Dans *Proceedings of the Workshop on Grammar and Lexicon : Interactions and Interfaces (GramLex 2016)* (pp. 92–101). Osaka, Japon.

Kahane, S., & Lareau, F. (2016b). Word ordering as a graph rewriting process. Dans A. Foret, G. Morrill, R. Muskens, R. Osswald, & S. Pogodalla (Eds.), *Formal grammar* (pp. 216–239). Springer.

Kogure, K. (1990). Strategic lazy incremental copy graph unification. Dans *Proceedings of the 12th International Conference on Computational Linguistics (COLING 1990)* (pp. 223–228). Helsinki, Finlande.

Lareau, F. (2007). Vers une formalisation des décompositions sémantiques dans la grammaire d'unification Sens-Texte. Dans *Actes de la 14ème conférence sur le traitement automatique des langues naturelles (TALN 2007).* Toulouse, France.

Lareau, F. (2008). *Vers une grammaire d'unification Sens-Texte du français : le temps verbal dans l'interface sémantique-syntaxe* (Thèse de doctorat). Université de Montréal, Montréal, Canada.

Lareau, F. (2009). Le temps verbal dans l'interface sémantique-syntaxe du français. Dans D. Beck, K. Gerdes, J. Milicevic, & A. Polguère (Eds.), *Proceedings of the 4th International Conference on Meaning–Text Theory (MTT 2009)* (pp. 225–232). Montréal, Canada : Observatoire de linguistique Sens-Texte.

Lareau, F. (2011). Grammemes. Dans I. Boguslavsky & L. Wanner (Eds.), *Proceedings of the 5th International Conference on Meaning–Text Theory (MTT 2011)* (pp. 144–153). Barcelone, Espagne.

Lison, P. (2006). *Implémentation d'une interface sémantique-syntaxe basée sur des grammaires d'unification polarisées* (Mémoire de maîtrise). Université catholique de Louvain, Louvain-la-Neuve, Belgique.

Maxwell, J., & Kaplan, R. (1993). The interface between phrasal and functional constraints. *Computational Linguistics*, *19*, 571–590.

McGuire, P. (2007). *Getting started with Pyparsing*. O'Reilly.

Mel'čuk, I. (1988). *Dependency syntax: theory and practice*. Albany, New York : State University of New York Press.

Mel'čuk, I. (1993). *Cours de morphologie générale*. Montréal, Canada : Les Presses de l'Université de Montréal.

Mel'čuk, I. (1997). *Vers une linguistique Sens-Texte.* Leçon inaugurale au Collège de France, Paris.

Nasr, A. (1995). A formalism and a parser for lexicalised dependency grammars. Dans *Proceedings of the 4th International Workshop on Parsing Technology (IWPT 1995)* (pp. 185–195). Prague, République tchèque.

Pereira, F. C. N. (1985). A structure-sharing representation for unification-based formalisms. Dans *Proceedings of the 23th Annual Meeting of the Association for Computational Linguistics (ACL 1985)* (pp. 137–144). Chicago, Illinois.

Perrier, G. (2002). Descriptions d'arbres avec polarités : les grammaires d'interaction. Dans *Actes de la 9ème conférence sur le traitement automatique des langues naturelles (TALN 2002).* Nancy, France.

Polguère, A. (1998). La théorie Sens-Texte. *Dialangue*, *8–9*, 9–30.

Polguère, A. (2003). *Lexicologie et sémantique lexicale : Notions fondamentales*. Montréal, Canada : Les Presses de l'Université de Montréal.

Shieber, S. M. (1990). Les grammaires basées sur l'unification. Dans P. Miller & T. Torris (Eds.), *Formalismes syntaxiques pour le traitement automatique du langage naturel.* Paris, France : Hermes.

Shieber, S. M. (2003). *An introduction to unification-based approaches to grammar*. Brookline, Massachusetts : Microtome Publishing.

Tomabechi, H. (1991). Quasi-destructive graph unification. Dans *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL 1991)* (pp. 315–322). Berkeley, Californie.

Tomabechi, H. (1992). Quasi-destructive graph unification with structure-sharing. Dans *Proceedings of the 14th International Conference on Computational Linguistics (COLING 1992)* (pp. 440–446). Nantes, France.

Tomabechi, H. (1993). *Efficient unification for natural language* (Thèse de doctorat). Université Carnegie-Mellon, Pittsburgh, Pennsylvanie.

Tomabechi, H. (1995). Design of efficient unification for natural language. *Journal of Natural Language Processing*, *2*(2), 23–58.

van Lohuizen, M. P. (2000). Memory-efficient and thread-safe quasi-destructive graph unification. Dans *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL 2000).* Hong Kong, Chine.

Wroblewski, D. A. (1987). Nondestructive graph unification. Dans *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI 1987)* (pp. 582–587). Seattle, Washington.