Définition du formalisme

L'objectif initial de cette thèse est la mise au point d'un formalisme comprenant d'une part un puissant langage de description linguistique, et capable d'autre part de produire une caractérisation formelle des langages résultants, permettant l'analyse ou la génération automatique d'énoncés. En particulier, le but est de pouvoir fournir à un linguiste un outil de description simple, permettant d'implémenter directement des phénomènes linguistiques (notamment sous la forme de contraintes de grammaticalité) ; et d'en obtenir automatiquement un analyseur syntaxique, capable – si possible – de travailler en un temps polynomial relativement à la taille de l'énoncé analysé. C'est la poursuite de ce double objectif qui a guidé les décisions de conception et le travail présentés dans ce chapitre.

À l'image des grammaires transformationnelles ou des ACG, le formalisme résultant construit d'abord la structure profonde (ou abstraite) d'un énoncé, et en déduit ensuite ses représentations phonologiques ou sémantiques (les structures concrètes). Il décrit dans un premier temps l'ensemble des structures abstraites comme des arbres, représentant de façon hiérarchique les composants syntaxiques de l'énoncé. La correction syntaxique (au sens linguistique) de ces structures est établie par un ensemble de contraintes de bonne formation. Celles-ci sont modélisées par des formules logiques, une structure abstraite étant considérée comme bien formée lorsqu'elle vérifie l'ensemble des formules logiques qui lui sont rattachées. Cette approche est similaire à celle de MTS[Rogers, 1996], hormis que les modèles acceptés représentent seulement la structure abstraite des phrases correctes, et non leur forme phonologique. En particulier, les questions liées à l'ordre des mots sont exclues de cette description, et sont traitées dans un second temps.

L'autre versant du formalisme est la description de linéarisations, c'est-à-dire de fonctions transformant une structure abstraite en sa représentation concrète. Cette dernière peut être une chaîne de caractères représentant la forme phonologique de l'énoncé, mais également une représentation sémantique (à la manière de Montague [1974]). Le terme de « linéarisation » sera

employé dans tous les cas par abus de langage. Tout comme les conditions de grammaticalité des énoncés, la relation entre structures abstraites et concrètes s'appuiera sur des contraintes logiques. L'emploi de plusieurs linéarisations associant à un même ensemble de structures abstraites plusieurs représentations concrètes peut alors permettre d'effectuer des tâches liées au traitement automatique des langues, comme l'extraction d'informations, la génération de texte ou la traduction automatique. Comme dans les ACG, la structure abstraite sert alors d'étape intermédiaire pour passer d'une représentation concrète à une autre.

Tout au long de ce chapitre, nous illustrerons notre méthodologie de description linguistique au travers d'un langage artificiel basique, formé d'expressions conditionnelles (si/alors/sinon) et arithmétiques (opérations binaires), qui servira de support pour montrer le fonctionnement du formalisme. L'application de celui-ci à la modélisation de phénomènes linguistiques réels fera ensuite l'objet d'un traitement à part, tout au long du chapitre 4.

3.1 Structures abstraites

Nos *structures abstraites* sont des arbres, dont les nœuds représentent les composants syntaxiques de l'énoncé, et les feuilles les éléments du lexique. Les arêtes sont étiquetées par des fonctions syntaxiques, dénotant la relation entre le sous-arbre qu'elles dominent et son nœud parent. Par convention, toutes les arêtes sortantes d'un même nœud portent des étiquettes différentes. Un inconvénient de ce choix est que les adjonctions deoivent se faire de manière récursive, plutôt que de permettre, par exemple, à de multiples adverbes de s'appliquer au même niveau. Cela facilite en revanche la visualisation de la forme générale des structures abstraites dans la grammaire d'approximation.

En raison de cette convention, et comme l'ensemble des étiquettes d'arêtes apparaissant dans une grammaire est fixé, tous les arbres d'une grammaire sont de degré borné ; ce qui permet de les considérer comme des termes. Tous les nœuds internes de ces termes sont construits au moyen d'un même symbole (noté $\bullet$) dont l'arité est égale au nombre d'étiquettes d'arête utilisées dans la grammaire. Par conséquent, chaque étiquette d'arête correspond à un entier positif inférieur ou égal à l'arité de $\bullet$, et les sous-termes correspondant à des sous-arbres inexistants (c'est-à-dire dont la fonction syntaxique n'est pas remplie) sont remplacés par un symbole $\bot$, d'arité nulle. Enfin, les éléments du lexique sont représentés par autant de symboles, également d'arité nulle.

La figure 3.1 illustre cette correspondance, en présentant à gauche une structure abstraite sous forme d'arbre, et à droite le terme qui l'encode. Cette structure abstraite appartient à notre langage d'exemple, et correspond à l'expression « si vrai alors 1 + 1, sinon 0 ». L'ensemble des étiquettes d'arête susceptibles d'apparaître est alors : $\{si, alors, sinon, op, arg_1, arg_2\}$; le symbole

• des nœuds internes est donc d'arité 6 dans le terme à droite. L'ensemble des feuilles utilisées (les éléments du lexique) est décrit dans la prochaine section.

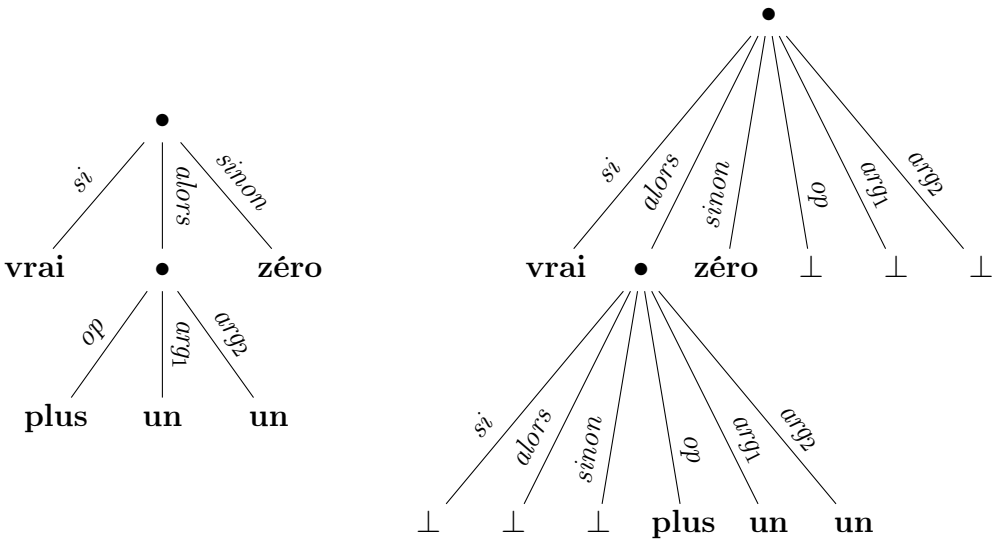


FIGURE 3.1 – Représentation et implémentation d'une structure abstraite

Dans la suite, nous emploierons systématiquement des arbres pour représenter les structures abstraites, en omettant les feuilles étiquetées par ⊥ et leurs arêtes entrantes des représentations, sauf pour expliciter des détails d'implémentation (nos résultats techniques s'appuyant sur l'encodage sous forme de termes).

Du point de vue des modélisations linguistiques à venir, nos choix de représentation quant à la syntaxe profonde des phrases nous conduisent à utiliser une étiquette d'arête spécifique pour représenter la tête d'un syntagme (c'est-à-dire sa projection lexicale) et, comme ici, à ne pas distinguer les nœuds internes de la structure (tous construits à l'aide du même symbole •). Une alternative aurait été de placer la tête syntagmatique directement sur les nœuds internes, à la manière de Tesnière [1959] (comme illustré par le déplacement de l'opérateur **plus** dans la figure 3.2) ; cela aurait toutefois compliqué la modélisation de l'adjonction, en forçant tous les adjoints à être rattachés au même niveau, et ainsi à utiliser un encodage vers les termes moins transparent. Une autre possibilité envisagée initialement était d'étiqueter les nœuds internes par des catégories syntaxiques, mais cet étiquetage est apparu comme artificiel et, par la suite, redondant avec l'usage des non-terminaux dans les grammaires d'approximation décrites plus bas.

3.1.1 Entrées lexicales

L'ensemble des étiquettes utilisées pour les feuilles est répertorié dans un *lexique*, dont les entrées associent chaque étiquette à un ensemble de traits,

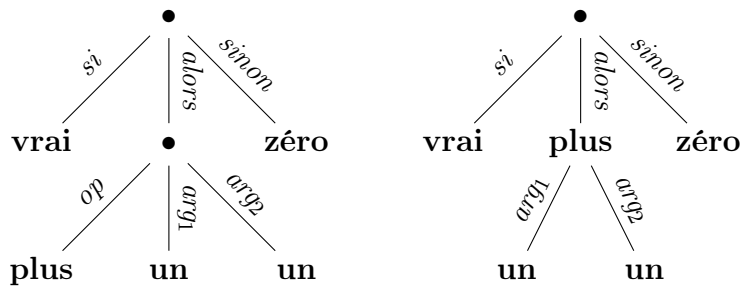


FIGURE 3.2 – Exemple de placement de l'information sur les nœuds internes

nommés *propriétés* de l'entrée lexicale. Ces propriétés seront utilisées pour contraindre la grammaticalité des structures abstraites : elles incluront dans le prochain chapitre (consacré aux descriptions linguistiques) des informations syntaxiques telles que les parties du discours, marques d'accord, sous-catégorisation, restrictions de sélection, *etc.*

Par hypothèse, un lexique contient un nombre fini d'entrées et de propriétés associées à chaque entrée. Un lexique minimal pour notre langage d'exemple est donné par la table 3.1, incluant des propriétés permettant de distinguer les types de valeurs (booléen, entier), les opérateurs et leur type (logique, arithmétique), ainsi que l'arité (unaire ou non) et les priorités (faible ou forte) de ces derniers.

Entrée	Propriétés
vrai	booléen
faux	booléen
zéro	entier
un	entier
ou	opérateur ; logique ; priorité −
et	opérateur ; logique ; priorité +
non	opérateur ; logique ; priorité + ; unaire
plus	opérateur ; arithmétique ; priorité −
moins	opérateur ; arithmétique ; priorité −
fois	opérateur ; arithmétique ; priorité +

TABLE 3.1 – Exemple de lexique

Nous donnons maintenant une définition formelle d'un lexique, construit à partir d'un ensemble fini de mots et d'un ensemble fini de propriétés.

Définition 3.1. Étant donné un ensemble fini de symboles L appelés *mots du lexique* et un ensemble fini de symboles P appelés *propriétés* (lexicales), une *entrée lexicale* est une paire (l, P_l), où $l \in L$ et $P_l \in \mathscr{P}(P)$.

Un *lexique* $\mathscr{L}$ est simplement défini comme un ensemble (fini) d'entrées lexicales.

3.2 Grammaire d'approximation

Afin de caractériser l'ensemble des structures abstraites valides – c'est-à-dire qui dénotent des énoncés syntaxiquement corrects – nous nous appuierons dans un premier temps sur une grammaire régulière de termes. Les termes qu'elle décrit correspondent à des structures abstraites arborescentes, selon l'encodage décrit précédemment. Cette grammaire vise uniquement à spécifier la forme générale des structures abstraites, sans modéliser toutes les règles de bonne formation des énoncés. Le langage ainsi généré illustre simplement la structure récursive de la langue, et est une grossière sur-approximation de l'ensemble des structures abstraites bien formées ; il sera affiné par la suite par l'ajout de contraintes de bonne formation. Dans la suite, nous désignerons cette grammaire indifféremment sous le terme de *grammaire d'approximation* ou de *grammaire support* (en raison des contraintes et des règles de linéarisation qui viendront décorer ses productions par la suite). Cette approche hybride n'est pas sans rappeler celle de Boral et Schmitz [2012], qui filtre les arbres de dérivation d'une grammaire hors-contexte à l'aide de la logique dynamique propositionnelle sur les arbres.

Les non-terminaux d'une grammaire d'approximation dénotent le type des structures abstraites qu'ils génèrent. En outre, plutôt que d'énumérer toutes les règles lexicales possibles, nous emploierons par souci de concision des propriétés issues du lexique en guise de symboles terminaux : implicitement, ces symboles dénotent n'importe quelle entrée du lexique possédant la propriété correspondante.

La figure 3.3 fournit une grammaire d'approximation possible pour notre langage d'exemple en listant ses productions. Elle possède un unique symbole non-terminal E, désignant une expression (arithmétique ou logique). La première production (en haut à gauche) permet de construire des expressions conditionnelles (« si [condition] alors [expression 1], sinon [expression 2] »), la production suivante (à droite) permettant de combiner une ou deux expression(s) au moyen d'un opérateur (la notation « (E) » est explicitée à la fin de cette section), et les deux productions terminales (en bas) permettent de réécrire une expression comme une valeur entière ou booléenne respectivement.

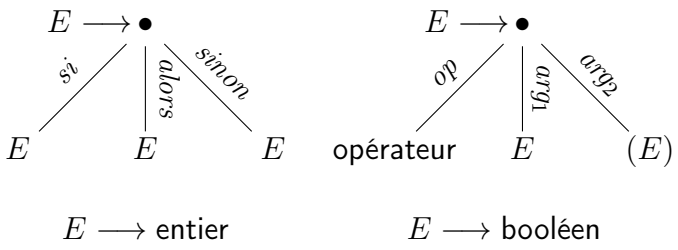


FIGURE 3.3 – Exemple de grammaire d'approximation

Observons que cette grammaire n'opère pas de distinction entre les expres-

sions et opérateurs logiques ou arithmétiques, ce qui permet de produire, entre autres, des expressions que l'on souhaiterait considérer comme mal formées, comme illustré par la figure 3.4. Celle-ci comporte deux structures abstraites appartenant au langage décrit par la grammaire d'approximation ; la seconde étant dénuée de sens, puisqu'elle combine deux nombres entiers au moyen d'un opérateur logique unaire portant sur des booléens. Ces structures seront par la suite filtrées au moyen des contraintes de bonne formation évoquées précédemment.

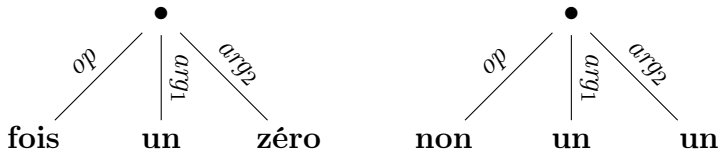


FIGURE 3.4 – Exemples de structures abstraites non-contraintes

Les productions d'une grammaire d'approximation seront toujours représentées avec leur membre droit sous sa forme arborescente pour faciliter la lecture. Pour des raisons pratiques, nous emploierons également un raccourci de notation dans les membres droits des productions : certains non-terminaux seront notés entre parenthèses. Cette écriture dénote l'optionnalité : le non-terminal correspondant peut soit être réécrit de la façon usuelle, soit être simplement absent de la structure résultante. Alternativement, la notation (A) peut être interprétée comme un non-terminal distinct de A, auquel sont implicitement associées les deux productions $(A) \to A$ et $(A) \to \bot$.

Dans le cadre du formalisme décrit ici, l'emploi d'une grammaire d'approximation pour spécifier la forme générale des structures abstraites n'est pas indispensable : il est possible d'obtenir le même ensemble de structures en utilisant directement des contraintes logiques (qui décrivent également la classe des langages réguliers de termes). Cependant, l'emploi d'une grammaire permet de disposer d'un « squelette », qui servira par la suite de support pour imposer des contraintes sur certaines réécritures et pour décrire plus aisément la linéarisation des structures abstraites vers des formes concrètes. En particulier, la lisibilité des contraintes logiques décrites par la suite bénéficie grandement de l'ancrage visuel fourni par les productions de la grammaire.

3.3 Contraintes logiques

Afin de contraindre la grammaticalité des structures abstraites, nous allons ensuite ajouter à la grammaire support des règles de bonne formation exprimées par des formules logiques. L'ensemble des arbres du langage abstrait sera alors restreint à l'ensemble des termes générés par la grammaire d'approximation qui satisfont également toutes les contraintes logiques associées. Le choix

d'un langage logique pour exprimer ces contraintes a d'importantes conséquences : il détermine l'expressivité du formalisme, ainsi que la complexité algorithmique des problèmes de décision associés. Dans notre cas, la logique monadique du second ordre sur une signature à k successeurs (MSOkS) s'impose comme un candidat (maximal) naturel : comme mentionné précédemment (voir section 2.3.4), l'ensemble des modèles satisfaisant à ses formules constitue un langage régulier, là où la satisfiabilité pour les logiques d'ordre supérieur n'est pas généralement décidable. En outre, il ressort de notre travail de modélisation décrit dans le chapitre 4 que de nombreuses règles de correction syntaxique issues de plusieurs langues sont exprimables au moyen de ce langage, ce qui suggère que cet outil permet de modéliser la syntaxe des langues en général. Nos formules seront donc construites en utilisant un sous-ensemble de MSOkS.

Nous allons d'abord détailler la nature exacte de ce vocabulaire logique, puis nous décrirons la façon dont les contraintes logiques enrichissent la grammaire d'approximation.

3.3.1 Vocabulaire logique

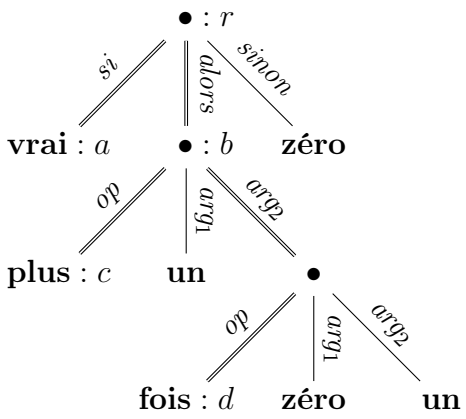
À l'usage, il apparaît que toute l'expressivité de la logique monadique du second ordre sur les termes n'est pas requise pour exprimer des contraintes sur la structure des phrases à un niveau abstrait. En particulier, la capacité de quantifier sur des ensembles de positions ne semble pas à première vue avoir d'application immédiate en linguistique. Toutefois, la restriction à une simple logique du premier ordre sur les termes ne permet pas d'exprimer de manière adéquate les phénomènes dits de dépendance à distance, tels que les contraintes d'îlot modélisées dans le chapitre suivant (voir section 4.4). Ces derniers requièrent d'une façon ou d'une autre la capacité d'exprimer des relations portant sur des chemin de longueur non-bornée entre deux positions – ce qui est exprimable en utilisant MSOkS. Aucun des autres phénomènes rencontrés jusqu'à présent n'ayant requis plus d'expressivité, nos contraintes logiques seront construites en utilisant la logique du premier ordre sur une signature à k successeurs, enrichie des ensembles de relations suivants :

— Pour toute expression régulière r construite sur l'ensemble des étiquettes d'arête, $r(x, y)$ est une relation binaire qui est vraie si et seulement si il existe un mot w dans le langage de r tel que que le chemin de x à y est étiqueté par w.

— Pour toute paire d'expressions régulières r_1, r_2 sur l'ensemble des étiquettes d'arête, $r_1 \uparrow r_2(x, y)$ est une relation qui est vraie si et seulement si le plus petit ancêtre commun z de x et y est tel que le chemin de z à x est étiqueté par un mot de r_1 et celui de z à y par un mot de r_2.

La figure 3.5 illustre le fonctionnement de ces relations. Elle représente une structure abstraite dans laquelle certaines positions sont étiquetées par

des variables (a, b, c, d et r pour la racine). Considérons l'expression régulière $(arg_1 \mid arg_2)^*\ op$: la relation $(arg_1 \mid arg_2)^*\ op\,(x, y)$ qu'elle définit est vérifiée pour toute paire de positions x, y telle que x domine y par une série de zéro ou plus arêtes étiquetées par arg_1 ou arg_2, suivies d'une arête étiquetée op. Dans la figure, cette relation est vérifiée pour les paires b, c et b, d, dont les chemins portent respectivement les séries d'étiquettes op et $arg_2\ op$.



$$si \uparrow alors\ (arg_1 \mid arg_2)^*\ op$$

FIGURE 3.5 – Structure abstraite et relations sur des chemins réguliers

Par suite, la relation plus complexe $si \uparrow alors\ (arg_1 \mid arg_2)^*\ op$ donnée par la figure est vérifiée pour les paires a, c et a, d : dans chaque cas, le plus petit ancêtre commun des deux nœuds (en l'occurrence la racine r) domine a par une arête étiquetée si et c ou d par un chemin étiqueté $alors\ op$ ou $alors\ arg_2\ op$ respectivement, satisfaisant les expressions régulières données de part et d'autre du symbole $\uparrow$; les chemins correspondants étant ceux marqués par une double barre dans la figure.

La signature MSOkS employée inclut en outre k relations de succession, qui correspondent aux k étiquettes d'arête utilisées dans la grammaire. Aussi, par confort de lecture dans nos contraintes logiques, nous dénoterons la relation S_i entre deux positions en utilisant l'étiquette d'arête associée à l'entier $i \in [k]$ dans les termes représentant les structures abstraites, en suivant la correspondance décrite plus haut (section 3.1, §2). Par exemple, nous écrirons $sinon(x, y)$ au lieu de $S_3(x, y)$, en suivant la correspondance suggérée par la figure 3.1.

Cette signature comprend également les prédicats unaires liés aux symboles de l'alphabet gradué ; lesquels incluent, dans notre cas, le symbole $\bullet$ d'arité k, la feuille $\perp$, et les feuilles associées aux entrées du lexique. Par la suite, les nœuds internes $\bullet$ ne véhiculant pas d'information, le prédicat $\bullet(x)$ ne sera pratiquement pas utilisé. En revanche, afin de se rapporter aisément aux pro-

priétés du lexique, nous ajouterons un ensemble de prédicats nommés d'après ces propriétés ; ceux-ci seront vérifiés pour toute position étiquetée par une entrée lexicale possédant la propriété correspondante. Ainsi, dans la figure 3.5, la position a satisfait le prédicat $\mathsf{booléen}(a)$, tandis que la position c satisfait les prédicats $\mathsf{opérateur}(c)$ et $\mathsf{arithmétique}(c)$.

Cette logique servira de support à la fois pour définir notre notion de grammaticalité et pour décrire les linéarisations vers des structures concrètes. Elle est notre principal outil de description linguistique et, à cet effet, doit pouvoir exprimer de manière concise n'importe quel concept linguistique à l'œuvre dans nos structures abstraites. Aussi, nous souhaitons éviter d'avoir à construire ou répéter des formules logiques complexes – en particulier lorsqu'elles dénotent un concept linguistique qui n'est pas immédiatement apparent dans nos structures abstraites. Pour cela, nous enrichirons notre vocabulaire logique au fur et à mesure de la conception d'une grammaire, au moyen de l'opérateur $\triangleq$. Classiquement, le prédicat introduit à gauche de l'opérateur sera par la suite interprété comme une abréviation pour la formule logique à droite. Ce mécanisme s'avérera très utile pour construire de nouvelles primitives linguistiques itérativement, en s'appuyant sur celles définies jusqu'alors. Le principal avantage de cette approche est que les formules logiques construites à partir de ces alias ont une bien meilleure lisibilité que celles bâties simplement sur la signature logique de départ, ce qui contribue à la facilité de compréhension et à la maintenabilité des grammaires.

Définition formelle Nous allons maintenant définir formellement la signature logique sur les termes utilisée par la suite, ainsi que les prédicats supplémentaires associés. Soit $\mathscr{L}$ le lexique utilisé, P l'ensemble de ses propriétés, et E l'ensemble des étiquettes d'arêtes. Pour commencer, nous rappelons la construction habituelle de l'ensemble des expressions régulières sur E.

Définition 3.2. L'ensemble $\mathcal{R}(E)$ des *expressions régulières sur* E est défini comme suit :

- $\emptyset$ est une expression régulière, qui dénote le langage vide : $\mathcal{L}(\emptyset) = \emptyset$.
- ε est une expression régulière, qui dénote le mot vide : $\mathcal{L}(\varepsilon) = \{\varepsilon\}$.
- Si $e \in E$, e est une expression régulière, qui dénote le langage $\mathcal{L}(e) = \{e\}$.
- Si $r_1, r_2 \in \mathcal{R}(E)$, $r_1 \mid r_2$ est une expression régulière qui dénote l'union des langages : $\mathcal{L}(r_1 \mid r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$.
- Si $r_1, r_2 \in \mathcal{R}(E)$, $r_1 r_2$ est une expression régulière, qui dénote la concaténation des langages : $\mathcal{L}(r_1 r_2) = \{w_1 w_2 \mid w_1 \in \mathcal{L}(r_1) \wedge w_2 \in \mathcal{L}(r_2)\}$.
- Si $r \in \mathcal{R}(E)$, r^* est une expression régulière, qui dénote l'itération du langage : $\mathcal{L}(r^*) = \{w^n \mid w \in \mathcal{L}(r) \wedge n \in \mathbb{N}\}$.

Nous décrivons maintenant l'ensemble des formules logiques sur la signature qui produit nos structures abstraites.

Définition 3.3. Étant donné un ensemble dénombrable de variables $\mathcal{X}$ notées $x, y, z, \ldots$:

— Si $l \in \mathscr{L}$, $l(x)$, $\bullet(x)$ et $\perp(x)$ sont des formules atomiques.
— Si $r \in \mathcal{R}(E)$, $r_1(x, y)$ est une formule atomique.
— Si $r_1, r_2 \in \mathcal{R}(E)$, $r_1 \uparrow r_2(x, y)$ est une formule atomique.
— Si ϕ et ψ sont des formules, $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \Rightarrow \psi$, $\phi \Leftrightarrow \psi$, $\exists x.\phi$ et $\forall x.\phi$ sont des formules.

Par souci de lisibilité, le prédicat $\perp(x)$ (correspondant à l'absence d'une entrée lexicale) sera dénoté par la formule sans(x), et sa négation sera abrégée par la formule avec(x). De plus, l'expression régulière formée par la disjonction de toutes les étiquettes d'arêtes présentes dans $E = \{e_1 \ldots e_n\}$ sera abrégée en dom $= e_1|\ldots|e_n$, afin de pouvoir exprimer simplement la dominance immédiate (ou au sens large) entre deux nœuds – en écrivant par exemple dom(x, y) ou dom$^*(x, y)$.

À l'exception des expressions régulières, l'ensemble des formules obtenues est un sous-ensemble strict de MSOkS n'utilisant pas de variables du second ordre ; la sémantique qui leur est associée est également la même que celle indiquée dans la section 2.1.5. Les formules atomiques $r(x, y)$ dénotent pour leur part que x domine y de telle sorte que les étiquettes des arêtes sur le chemin de x à y forment un mot appartenant à $\mathcal{L}(r)$. Cette interprétation peut-être obtenue en transformant les expressions régulières en une formule équivalente de la logique monadique du second ordre. Pour toute expression régulière $r \in \mathcal{R}(E)$, son interprétation $[\![r]\!]$ peut être obtenue inductivement comme suit :

— $[\![\emptyset(x, y)]\!] \overset{def}{\Leftrightarrow} Faux$
— $[\![\varepsilon(x, y)]\!] \overset{def}{\Leftrightarrow} x = y$
— $[\![e(x, y)]\!] \overset{def}{\Leftrightarrow} S_i(x, y)$
— $[\![r_1 \mid r_2(x, y)]\!] \overset{def}{\Leftrightarrow} [\![r_1(x, y)]\!] \vee [\![r_2(x, y)]\!]$
— $[\![r_1 r_2(x, y)]\!] \overset{def}{\Leftrightarrow} \exists z.\, [\![r_1(x, z)]\!] \wedge [\![r_2(z, y)]\!]$
— $[\![r^*(x, y)]\!] \overset{def}{\Leftrightarrow} \exists^2 Z.x \in Z \wedge \forall z.z \in Z \Rightarrow (\exists z' \neq z.z' \in Z \wedge [\![r(z, z')]\!]) \vee z = y$

Comme évoqué précédemment, les étiquettes d'arête $e(x, y)$ dénotent les relations de succession $S_i(x, y)$ (avec $i \in [k]$) correspondantes dans la définition usuelle de MSOkS – rappelons que l'égalité $\#(E) = ar(\bullet)$ permet d'établir cette correspondance.

La formule qui traduit l'itération utilise un quantificateur du second ordre pour construire un ensemble Z de positions intermédiaires entre x et y : cet ensemble inclut x, et tout nœud z qu'il contient doit soit dominer un autre nœud z', également inclus dans Z, par un chemin reconnu par r (effectuant un pas d'itération) ; soit être le nœud y lui-même (le « dernier » élément de Z).

Enfin, les formules atomiques construites à partir de l'opérateur $\uparrow$ décrivent un chemin montant dans l'arbre depuis x jusqu'à un plus petit ancêtre commun, puis redescendant vers y. Elles peuvent donc être interprétées de la manière suivante (la dernière clause traduisant le fait que z est le plus petit ancêtre

commun à x et y) :

$$[\![r_1 \uparrow r_2(x,y)]\!] \overset{def}{\Leftrightarrow} \exists z.[\![r_1(z,x)]\!] \wedge [\![r_2(z,y)]\!] \\ \wedge (\forall z'.\, [\![\mathrm{dom}^*(z',x)]\!] \wedge [\![\mathrm{dom}^*(z',y)]\!] \Rightarrow [\![\mathrm{dom}^*(z',z)]\!])$$

3.3.2 Contraintes de grammaticalité

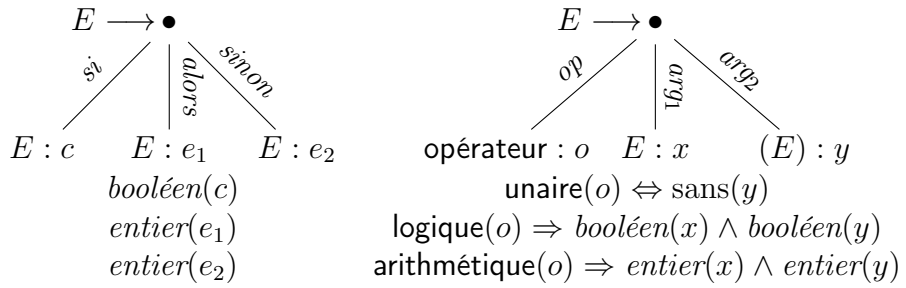
Maintenant que nous disposons d'un vocabulaire logique nous permettant de formuler des *contraintes logiques* de grammaticalité, nous allons décrire la manière dont ces contraintes vont enrichir nos grammaires d'approximation.

Nous décorons les productions de ces dernières par deux moyens : d'une part, les nœuds internes et feuilles du membre droit d'une production peuvent être étiquetés par des variables de $\mathcal{X}$; d'autre part, chaque production est associée à un ensemble de formules logiques, lesquelles peuvent contenir comme variables libres les étiquettes apparaissant dans le membre droit de leur production. Une grammaire ainsi décorée par des contraintes logiques est appelée une *grammaire enrichie*.

Reprenant notre exemple précédent, nous enrichissons la grammaire d'approximation permettant de construire des expressions au moyen de trois règles pour chacune des deux premières productions, comme illustré dans la figure 3.6. Les contraintes ajoutées s'appuient sur deux prédicats supplémentaires *booléen* et *entier*, définis au bas de la figure : une valeur v (correspondant à une position arbitraire dans l'arbre) satisfait le prédicat *booléen* lorsqu'elle correspond à une entrée lexicale possédant la propriété booléen, lorsqu'elle domine un opérateur logique (par le biais d'une arête étiquetée *op*), dont le résultat est, par hypothèse, une valeur booléenne, ou lorsqu'elle se réduit à $\perp$. Le prédicat *entier* est défini de manière similaire, en ajoutant un cas supplémentaire pour inclure le résultat d'une expression conditionnelle (par hypothèse un entier).

Par suite, les contraintes sur les expressions conditionnelles (formées par la première production) imposent que leurs arguments soient respectivement une valeur booléenne (c) et deux valeurs entières (e_1 et e_2). Les contraintes sur la seconde production imposent quant à elles, respectivement : que l'absence de l'argument y, optionnel, coïncide avec l'emploi d'un opérateur unaire, que l'emploi d'un opérateur logique suppose des arguments booléens, et de même qu'un opérateur arithmétique reçoit des arguments entiers. Le même langage abstrait aurait pu être obtenu en multipliant les symboles non-terminaux : en employant par exemple E_a et E_l comme symboles non-terminaux pour des expressions arithmétiques et logiques respectivement, et E^u ou E^b pour des expressions employant des opérateurs unaires ou binaires, et en spécifiant chacune des productions correspondantes. L'emploi de contraintes logiques vise à éliminer ces énumérations lorsqu'aucune interaction n'entre en jeu entre plusieurs concepts, comme ici entre l'arité et le type des opérateurs.

Lorsqu'une structure abstraite appartient au langage décrit par une grammaire d'approximation, il existe une dérivation de cette structure qui associe



$$
\begin{aligned}
\textit{booléen}(v) \;&\triangleq\; \mathsf{booléen}(v)\\
&\vee\; \exists o.\, op(v,o) \wedge \mathsf{logique}(o)\\
&\vee\; \mathrm{sans}(v)\\
\textit{entier}(v) \;&\triangleq\; \mathsf{entier}(v)\\
&\vee\; \exists o.\, op(v,o) \wedge \mathsf{arithmétique}(o)\\
&\vee\; \exists c.\, si(v,c) \wedge \mathrm{avec}(c)\\
&\vee\; \mathrm{sans}(v)
\end{aligned}
$$

FIGURE 3.6 – Exemple de grammaire enrichie

des productions de la grammaire à des nœuds internes de la structure : une production est associée à un nœud lorsque celui-ci a été obtenu par la réécriture d'un non-terminal qui est permise par cette production. Cette association permet également de faire correspondre une instance d'une variable décorant le membre droit d'une production à une position dans la structure abstraite résultante. Une dérivation d'une structure abstraite est alors dite *valide* si et seulement si toutes les contraintes logiques associées aux productions utilisées dans cette dérivation sont satisfaites. Cette notion de validité d'une dérivation s'étend naturellement aux structures abstraites : une structure abstraite est valide si et seulement si il existe une dérivation valide de cette structure d'après la grammaire enrichie considérée.

La figure 3.7 illustre la notion de validité à travers un exemple de structure abstraite, en donnant une dérivation complète de celle-ci conforme à la grammaire d'approximation donnée plus haut, et en listant en bas les contraintes qu'elle doit vérifier. Les arêtes en pointillé de l'arbre de dérivation représentent les choix de réécriture, en indiquant les productions utilisées : les productions de la grammaire d'approximation sont numérotées de p_1 à p_4, un $\mathscr{L}$ indique une règle lexicale (réécriture d'une propriété par une entrée lexicale correspondante) et la mention « opt. » indique l'omission d'un non-terminal optionnel. Les productions p_1 et p_2, utilisées une fois chacune, instancient leurs contraintes logiques aux positions correspondantes dans la structure abstraite (remarquons que la position 13, non représentée dans la structure abstraite, correspond à une feuille $\perp$.). Chacune de ces contraintes étant vérifiée dans la structure abs-

traite en vertu des définitions des prédicats *booléen* et *entier*, la structure est considérée comme valide.

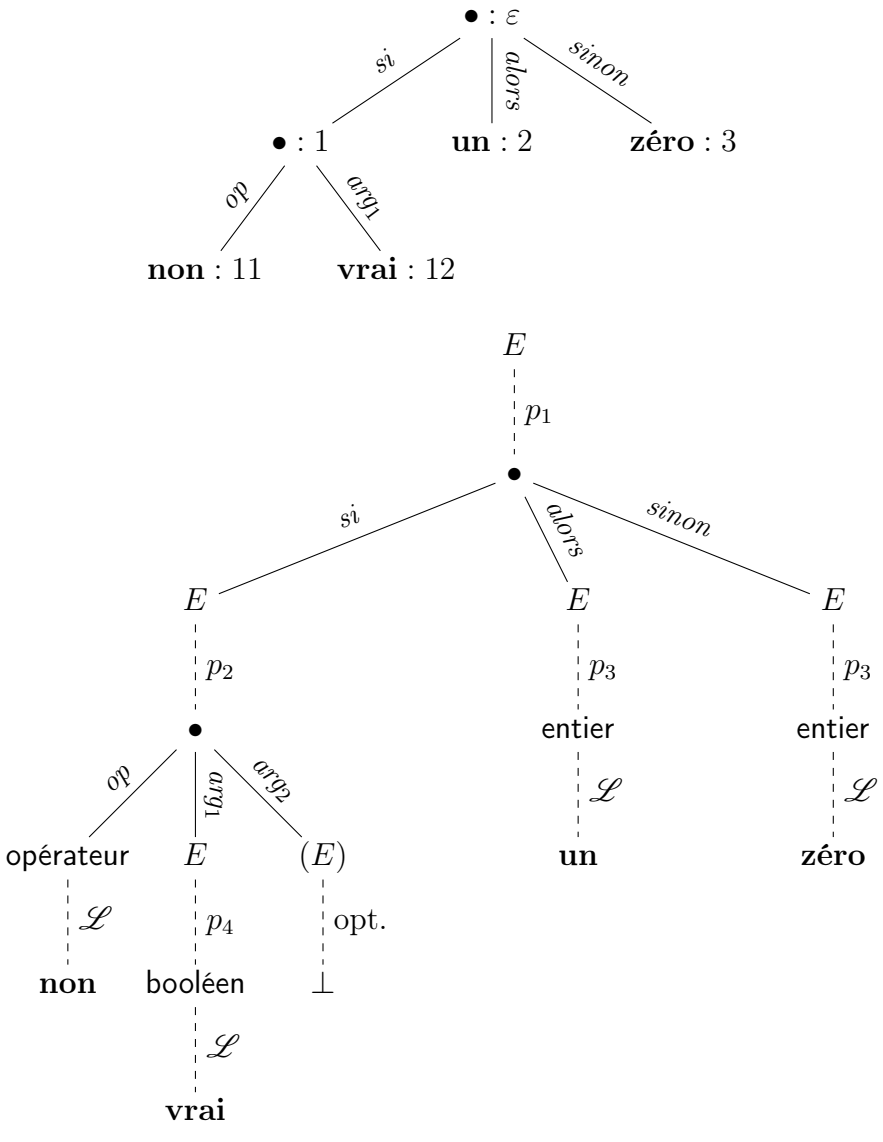
En pratique, ces contraintes logiques nous permettront de restreindre le langage abstrait en modélisant des contraintes linguistiques sur l'optionnalité de certains arguments, la sous-catégorisation, les restrictions de sélection et, le cas échéant, en imposant des dépendances à longue distance entre deux composants syntaxiques. En raison de la correspondance entre logique et automates décrite dans le chapitre 2, les grammaires enrichies par des contraintes ont le même pouvoir d'expression que les grammaires d'approximation dans l'absolu : la différence principale réside dans la concision qu'elles offrent pour décrire des langages. La prochaine sous-section illustre ce résultat en compilant une grammaire enrichie vers un langage régulier de termes.

Ce choix de compléter par des contraintes une grammaire d'approximation simple pour définir un langage abstrait est motivé par la volonté de ne pas surcharger la grammaire principale, et de conserver sa lisibilité. La construction d'une grammaire de réécriture modélisant plusieurs phénomènes du langage entraîne en effet rapidement, comme suggéré plus haut, une multiplication combinatoire des non-terminaux et des productions, qui doivent transmettre diverses informations sur le contexte de réécriture. Par contraste, nos contraintes logiques ajoutent aux productions de la grammaire une forme limitée (finie) de sensibilité au contexte. L'emploi de contraintes logiques permet ainsi de simplifier la construction de grammaires de grande taille, répondant aux mêmes besoins que la tradition des méta-grammaires évoquée section 1.3.

3.3.3 Compilation du langage abstrait

Une grammaire enrichie, telle que celle proposée ci-dessus, décrit un ensemble de structures abstraites valides, c'est-à-dire un langage de termes. Celui-ci est composé de l'ensemble des termes dérivables par la grammaire d'approximation qui satisfont l'ensemble des contraintes instanciées par leur dérivation. Ce langage de termes est en outre régulier, en raison de la connexion entre logique et automates décrite précédemment (voir section 2.3.4) ; nous en donnons maintenant une caractérisation effective, construite à partir d'une grammaire enrichie G.

La première étape est de convertir G afin que toutes ses contraintes logiques portent sur les racines de ses productions, puis d'interpréter les non-terminaux optionnels et règles lexicales. Par suite, l'idée centrale est de construire une grammaire régulière de termes G' (sans contraintes logiques) où les contraintes apparaissent directement dans les termes du langage, en construisant un alphabet gradué dont les symboles sont associés aux formules logiques de la grammaire. L'ensemble des contraintes de la grammaire G étant fini, nous pouvons alors construire une formule MSOkS qui vérifie que tout nœud étiqueté par une contrainte satisfait cette contrainte. Le langage résultant est



p_1	p_2
booléen(1)	unaire(11) $\Leftrightarrow$ sans(13)
entier(2)	logique(11) $\Rightarrow$ *booléen*(12) $\wedge$ *booléen*(13)
entier(3)	arithmétique(11) $\Rightarrow$ *entier*(12) $\wedge$ *entier*(13)

FIGURE 3.7 – Dérivation et validation d'une structure abstraite

régulier, et son intersection avec le langage sans contraintes de G' (également régulier) garantit que les formules ne sont instanciées que là où les production de G le permettent. Rappelons que les propriétés des langages réguliers de termes citées dans la section 2.3.1 incluent leur clôture par intersection et morphisme linéaire : le langage résultant de l'intersection précédente est donc régulier. De plus, par construction, il ne diffère du langage associé à G que par les symboles de son alphabet gradué (qui portent des formules logiques en plus des symboles utilisés dans G) ; or, un réétiquetage effaçant les formules pour ne conserver que l'information d'origine constitue un type simple de morphisme linéaire, ce qui achève la construction du langage régulier associé à G.

Nous détaillons maintenant cette construction. Formellement, la grammaire enrichie G que nous devons compiler est définie par son lexique, l'ensemble de ses productions enrichies et l'ensemble de ses symboles non-terminaux, incluant son symbole de départ. Son lexique $\mathscr{L}$ est construit sur un ensemble fini de mots du lexique L et un ensemble fini de propriétés P (voir définition 3.1). Soit $\mathcal{N}$ l'ensemble de ses symboles non-terminaux, et S son symbole de départ. Nous définissons maintenant précisément la nature de nos productions enrichies.

Définition 3.4. Soit $\mathcal{S} = \{\bullet, \bot\} \cup \{A, (A) \mid A \in \mathcal{N}\} \cup P$ l'alphabet gradué utilisé dans les productions enrichies, incluant l'ensemble des symboles non-terminaux, y compris optionnels (ceux apparaissant entre parenthèses), ainsi que toute propriété de P issue du lexique (employée comme symbole terminal) ; tous les symboles sont d'arité nulle, à l'exception de $\bullet$ (d'arité k). L'ensemble des variables pouvant être utilisées dans les contraintes est noté $\mathcal{X}$.

Une *production enrichie* est un objet de la forme $A \to t; \Lambda; \Phi$, où :
- $A \in \mathcal{N}$ est un symbole non-terminal, nommé *membre gauche*.
- t est un terme construit sur l'alphabet gradué $\mathcal{S}$, nommé *membre droit*.
- $\Lambda : \mathrm{Dom}(t) \mapsto \mathcal{X}$ est une fonction partielle dite *d'étiquetage* de t.
- Φ est une *contrainte*, c'est-à-dire une conjonction de formules logiques respectant la définition 3.3.

Une production enrichie est *bien formée* lorsque sa fonction d'étiquetage Λ est injective – afin d'interdire à une même variable d'étiqueter deux positions distinctes du membre droit. L'ensemble des productions enrichies d'une grammaire G est noté $\mathcal{R}$.

Le processus de compilation de G esquissé plus haut se décompose ainsi :

1. Interprétation des non-terminaux optionnels.
2. Ajout des règles lexicales.
3. Déplacement des contraintes logiques vers la racine des membre droits.
4. Construction de la grammaire régulière de termes G'.
5. Construction d'une formule MSOkS Val vérifiant les contraintes.
6. Construction du langage de G à partir de $\mathcal{L}(G')$ et de $\mathcal{L}(\mathrm{Val})$.

Non-terminaux optionnels Comme suggéré dans la section 3.2, nous pouvons interpréter tout non-terminal optionnel de la forme (A) au moyen de deux productions alternatives. Ainsi, pour tout symbole $A \in \mathcal{N}$, nous ajoutons le symbole (A) à $\mathcal{N}$, et les productions $(A) \to A; \emptyset; \emptyset$ et $(A) \to \bot; \emptyset; \emptyset$ à l'ensemble des productions de G, permettant de réécrire un (A) optionnel en A ou $\bot$, sans contrainte logique ou étiquette de variable associée ($\Lambda = \emptyset$ désignant la fonction d'étiquetage de domaine vide).

Règles lexicales Afin de factoriser un grand nombre de règles lexicales, nous avons proposé d'utiliser des propriétés du lexique comme symboles terminaux, pouvant être implicitement remplacés par n'importe quelle entrée lexicale possédant cette propriété. Nous interprétons donc maintenant toute propriété lexicale $p \in P$ comme un symbole non-terminal ; en ajoutant une règle $p \to (l, P_l); \emptyset; \emptyset$ à l'ensemble des productions pour toute entrée lexicale $(l, P_l) \in \mathscr{L}$ telle que $p \in P_l$ – ces nouvelles productions permettent de réécrire p en une entrée lexicale correspondante. Nous ajoutons également les paires (l, P_l) (les entrées lexicales) à l'alphabet gradué $\mathcal{S}$, en leur donnant l'arité zéro.

Déplacement des contraintes Nous modifions maintenant les contraintes associées aux productions de G afin de ne les faire dépendre que du nœud situé à la racine du membre droit. Pour ce faire, nous modifions chaque formule logique en liant par un quantificateur existentiel toutes les variables libres hormis celle étiquetant la racine du membre droit, et ajoutons la contrainte supplémentaire que les positions que dénotent les variables ainsi quantifiées doivent être situées dans la structure abstraite à une position relative (par rapport à la racine) correspondant à leur position dans l'arbre. La figure 3.8 illustre cette transformation sur une production simple (à gauche) : la partie droite de la figure donne le résultat du déplacement de la contrainte ϕ, à savoir une formule où seule la variable x demeure libre.

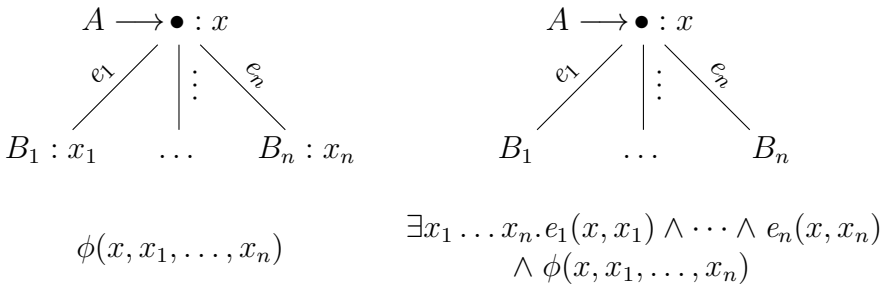


FIGURE 3.8 – Déplacement d'une contrainte vers la racine

Plus précisément, nous modifions la grammaire enrichie G comme suit : dans toute production $A \to t; \Lambda; \Phi$ telle que l'étiquette de la racine est $\Lambda(\varepsilon) =$

x, nous appliquons la séquence de transformations suivante : pour toute position valide $p \in \mathrm{Dom}(t)$ telle que $p \neq \varepsilon$ (excluant la racine), si l'étiquette de p est $\Lambda(p) = x_p$, nous remplaçons chaque formule $\phi \in \Phi$ par $\exists x_p.p(x, x_p) \wedge \phi$. La formule résultante dépend alors d'une variable libre x, qui dénote le nœud correspondant à la racine du membre droit dans la structure abstraite résultante. Nous supposons ici que la racine du membre droit de toute production est étiquetée par une variable ; le cas échéant, ajouter une étiquette x sur la racine si elle en est dénuée ne modifie pas le langage décrit par G.

Construction de la grammaire G' Nous pouvons maintenant construire la grammaire régulière de termes G' esquissée précédemment, qui construit des structures abstraites selon les dérivations permises par la grammaire d'approximation de G, en y incluant explicitement les contraintes qu'elles doivent satisfaire. Cette grammaire est obtenue à partir des productions modifiées de G (par la suite, G désigne implicitement la grammaire obtenue à la suite des transformations ci-dessus), en décorant les symboles terminaux et non-terminaux par des ensembles de contraintes devant être localement satisfaites.

Soit $\mathcal{S}_t = \{\bullet, \bot\} \cup \mathscr{L}$ l'ensemble des symboles terminaux de G et $\mathcal{N} \cup P$ l'ensemble de ses symboles non-terminaux, et soit $\mathcal{F} = \{\Phi \mid (A \to t; \Lambda; \Phi) \in \mathcal{R}\}$ l'ensemble des contraintes apparaissant dans G. Formellement, la grammaire $G' = (\mathcal{N}', \mathcal{S}', \mathcal{P}, S')$, qui produit l'ensemble des termes dérivables décorés par leurs contraintes dans G, est construite comme suit :

- $\mathcal{N}' = \{(A, F) \mid A \in \mathcal{N} \cup P \wedge F \subseteq \mathcal{F}\}$ est l'ensemble des paires formées d'un symbole non-terminal de G et d'un ensemble de contraintes de G.
- $\mathcal{S}' = \{(s, F) \mid s \in \mathcal{S}_t \wedge F \subseteq \mathcal{F}\}$ est l'ensemble des paires formées d'un symbole terminal de G et d'un ensemble de contraintes à satisfaire.
- $S' = (S, \emptyset)$ est le symbole de départ de G, sans contraintes associées.
- $\mathcal{P}$ est l'ensemble des productions de la forme :

$$(A, F) \to (s, F \cup \{\Phi\})\,(t'_1 \ldots t'_n)$$

telles que :

1. $A \to s(t_1 \ldots t_n); \Lambda; \Phi$ est une production enrichie de $\mathcal{R}$.
2. $t'_i = t_i[s \leftarrow (s, \emptyset)]$ pour tout $s \in \mathcal{S}_t \cup \mathcal{N} \cup P$ et tout $i \in [n]$.
3. $F \subseteq \mathcal{F}$.

Les conditions 1, 2 et 3 garantissent que l'ensemble F des contraintes associées au non-terminal A du membre gauche sont reportées sur le symbole s situé à la racine du membre droit, en y ajoutant la contrainte Φ apportée par la production enrichie de $\mathcal{R}$ utilisée ; le symbole s peut être terminal aussi bien que non-terminal. Les symboles des sous-termes éventuels dans le membre droit se voient associer un ensemble de contraintes vide, puisque toutes les contraintes ont été reportées sur la racine au cours de l'étape précédente.

Construction de la formule MSOkS Val Enfin, il nous reste à construire la formule logique Val, qui vérifie simplement que si un symbole à une position x dans un terme est associé à un ensemble de contraintes $F \subseteq \mathcal{F}$, alors la conjonction de toutes les formules de F est vérifiée. La formule résultante est :

$$\text{Val} \overset{def}{\Leftrightarrow} \forall x. \llbracket F \rrbracket (x) \Rightarrow \bigwedge_{\Phi \in F} \Phi(x) \qquad \text{avec :} \qquad \llbracket F \rrbracket (x) = \bigvee_{s \in \mathcal{S}_t} (s, F)(x)$$

Langage résultant Pour finir, le langage régulier de termes associés à G peut maintenant être construit à partir de $\mathcal{L}(G')$ et $\mathcal{L}(\text{Val})$, en considérant l'intersection de ces deux langages (G' assurant le respect des dérivations de G et Val la satisfaction des contraintes logiques associées), puis en effaçant par un morphisme linéaire l'information portant sur les formules logiques des termes du langage résultant (ce qui préserve la régularité). Ce morphisme est un simple réétiquetage, qui peut être défini par $h((s, F)(t_1 \dots t_n)) = s(h(t_1) \dots h(t_n))$.

Le langage $h(\mathcal{L}(G') \cap \mathcal{L}(\text{Val}))$ est alors effectivement l'ensemble des structures des structures abstraites valides sanctionnées par les règles de G.

3.4 Langage concret et linéarisation

Une fois l'ensemble des structures abstraites valides spécifié, il nous reste à décrire la relation entre ces dernières et leurs représentations concrètes, ou *réalisations*. Cette relation, nommée *linéarisation*, associe à toute structure abstraite valide un ensemble de réalisations. Nous la décrirons par l'intermédiaire de règles de construction ajoutées aux productions de la grammaire enrichie : la réalisation associée au non-terminal à gauche d'une production sera presque toujours (hormis dans le cas des requêtes logiques décrites plus bas) obtenue en combinant de diverses façons les réalisations associées aux non-terminaux du membre droit, à la manière des grammaires S-attribuées[Knuth, 1968]. Nous ferons à nouveau usage du langage logique décrit précédemment, pour conditionner des choix entre plusieurs réalisations.

De plus, un même langage abstrait pourra se voir associer plusieurs linéarisations différentes, par exemple dans le but de représenter différents aspects de l'énoncé (forme phonologique, sémantique, *etc.*). Afin d'unifier la présentation des linéarisations et de souligner la similarité avec l'approche des ACG, nous utiliserons le lambda-calcul simplement typé pour construire tous les types de réalisations, qu'il s'agisse de mots, de termes, de formules logiques ou d'opérations sur ces structures.

3.4.1 Règles de linéarisation

Dans leur version la plus simple, nos linéarisations sont décrites par des lambda-termes simplement typés, attachés aux productions de la grammaire

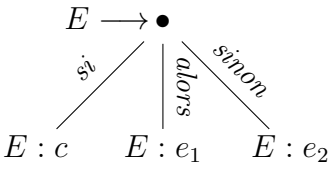
enrichie, et nommés *règles de linéarisation*. Ces lambda-termes décrivent la réalisation associée au non-terminal à gauche de la production. Chacune de leurs variables libres est associée à un non-terminal du membre droit, et dénote la réalisation associée à ce dernier. Puisque toute structure abstraite valide peut être associée à une dérivation selon ces productions, l'ensemble des réalisations d'une structure abstraite se définit naturellement à partir de l'ensemble des dérivations valides de cette structure : une réalisation valide est obtenue en considérant le lambda-terme associé à la production réécrivant le symbole de départ, et en y remplaçant itérativement toute variable associée à un non-terminal par le terme associé à la production utilisée pour réécrire cette occurrence du non-terminal.

Les lambda-termes associés aux productions terminales (construites implicitement à partir du lexique) sont dénotés par le nom l de l'entrée lexicale $(l, P_l) \in \mathscr{L}$, et leur interprétation dépend du langage concret visé. Chaque entrée lexicale pourra ainsi être associée par la suite à une constante ou à un lambda-terme clos représentant, suivant le cas, un mot sur un alphabet, une formule logique dénotant le sens du lexème, ou autre.

Nous établissons ensuite le lien entre les variables libres apparaissant dans les règles de linéarisation et les non-terminaux correspondants en étiquetant les membres droits des productions par des variables, de la même manière que précédemment pour les contraintes logiques. Par économie de notation, nous réutiliserons les mêmes variables que celles apparaissant dans les contraintes logiques ; à ceci près que seules les variables étiquetant des symboles non-terminaux sont susceptibles d'apparaître libres dans les lambda-termes décrivant les réalisations.

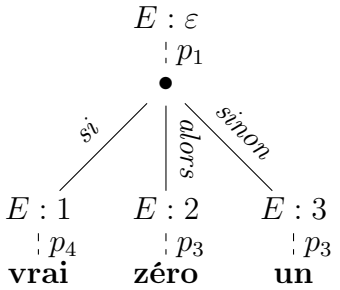
Afin d'illustrer ce fonctionnement la figure 3.10 montre la construction d'un lambda-terme à partir d'un arbre de dérivation, en suivant la règle de linéarisation donnée par la figure 3.9. Le langage concret utilisé est une transcription des expressions du langage abstrait utilisé comme exemple précédemment. Ces expressions sont représentées dans un langage de programmation fonctionnel fictif, utilisant les mots-clés `if`, `then` et `else`, ainsi que les opérateurs et constantes présents dans le lexique $\mathscr{L}$ (donné plus haut par la table 3.1). Le code source d'un programme écrit dans ce langage sera une chaîne de caractères, dénotée par un lambda-terme, suivant la manière évoquée au début de la section 2.2.4. Des réalisations intermédiaires sont attachées aux non-terminaux de la dérivation : dans notre exemple, ceux-ci correspondent exactement aux positions valides dans l'arbre représentant la structure abstraite (ε pour le symbole de départ et 1, 2 et 3 pour les autres). La réalisation de la structure est construite de manière ascendante : nous associons aux non-terminaux situés aux positions 1, 2 et 3 des lambda-termes correspondant aux entrées lexicales qui les réécrivent (respectivement `true`, `zero` et `one`). Puis, nous substituons ces derniers aux variables c, e_1 et e_2 dans la réalisation attachée à la production p_1, pour obtenir la réalisation attachée à la racine – c'est-à-dire au symbole de

départ.



$\lambda x^*.\texttt{if}\ (c\ (\texttt{then}\ (e_1\ (\texttt{else}\ (e_2\ x)))))$

FIGURE 3.9 – Règle de linéarisation



$1 : \texttt{true}\ ;\quad 2 : \texttt{zero}\ ;\quad 3 : \texttt{one}$
$\varepsilon : \lambda x^*.\texttt{if}\ (\texttt{true}\ (\texttt{then}\ (\texttt{zero}\ (\texttt{else}\ (\texttt{one}\ x)))))$

FIGURE 3.10 – Construction de la réalisation d'une structure abstraite

Le lambda-terme obtenu dans l'exemple ci-dessus est immédiatement en forme β-normale, et représente effectivement la chaîne de caractères « `if true then zero else one` ». Le cas échéant, la réalisation associée à une structure correspond au lambda-terme associé à sa racine modulo $=_{\beta\eta}$: pour illustrer ce fait, considérons une autre linéarisation possible pour notre langage, qui évalue directement les expressions du langage abstrait. Nous remplaçons la règle de linéarisation donnée par la figure 3.9 (associée à la production p_1) par : $c\ e_1\ e_2$. Cette règle applique simplement le lambda-terme associé à c aux lambda-termes associés à e_1 et e_2. En outre, nous associons aux entrées lexicales des lambda-termes dénotant des entiers naturels, dits entiers de Church[Church, 1940], et des opérations booléennes sur ces derniers. Les lambda-termes associés aux entrés lexicales devenant respectivement : $\lambda m^{(*\to*)\to*\to*}.\lambda n^{(*\to*)\to*\to*}.m$ (**vrai**), $\lambda f^{*\to*}.\lambda x^*.x$ (**zéro**) et $\lambda f^{*\to*}.\lambda x^*.f\ x$ (**un**). La réalisation associée à la racine ε est alors (en omettant, pour abréger, les annotations de type) :

$$\begin{aligned}(\lambda m.\lambda n.m)(\lambda f.\lambda x.x)(\lambda f.\lambda x.f\ x) &\to_{\beta\eta} (\lambda n.\lambda f.\lambda x.x)(\lambda f.\lambda x.f\ x)\\ &\to_{\beta\eta} \lambda f.\lambda x.x\\ &= \textbf{zéro}\end{aligned}$$

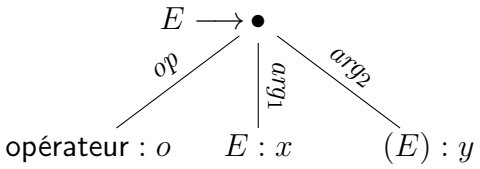
3.4.2 Réalisations multiples et conditions

La relation que nous voulons établir entre structures abstraites et concrètes n'est pas bijective en général. Nous souhaiterions pouvoir construire plusieurs paraphrases issues d'une même structure, et il se peut qu'à l'inverse, certaines structures abstraites n'aient pas de représentation valide dans un certain domaine – par exemple, dans le cas d'une grammaire synchrone multilingue. Afin

de tenir compte de ces faits, nous proposons d'associer un ensemble de lambda-termes représentant les diverses linéarisations possibles à chaque production (*réalisations multiples*), et d'ajouter la possibilité d'attacher à chacun d'entre eux une condition logique, qui doit être satisfaite pour permettre le choix de linéarisation correspondant (*condition de réalisation*).

Nous noterons les réalisations multiples en spécifiant directement l'ensemble des lambda-termes possibles ($\{M_1 \dots M_n\}$). Par facilité de lecture, ces lambda-termes seront énumérés ligne par ligne dans les figures. Les conditions de réalisation seront pour leur part notées en spécifiant une formule logique ϕ dénotant la condition à satisfaire et la réalisation M ainsi conditionnée, séparées par un symbole spécifique : $\longrightarrow$. La réalisation conditionnelle résultante $\phi \longrightarrow M$ se lit « phi permet M » et s'interprète comme « le choix de la réalisation M est permis lorsque la contrainte logique ϕ est vérifiée ».

La figure 3.11 exemplifie ces notations, en étendant la linéarisation proposée par la figure 3.9 à la production enrichie p_2 : la règle correspondante tient compte de l'optionnalité du nœud étiqueté par y dans la production, et produit une réalisation cohérente dans tous les cas. Les constantes `op` et `cp` dénotent respectivement une parenthèse ouvrante et fermante dans le code source.



$$\left\{ \begin{array}{lcl} \mathsf{unaire}(o) & \longrightarrow & \lambda a^*.o\ (x\ a) \\ \neg\mathsf{unaire}(o) & \longrightarrow & \lambda a^*.\mathtt{op}\ (x\ (o\ (y\ (\mathtt{cp}\ a)))) \end{array} \right\}$$

FIGURE 3.11 – Règle de linéarisation avec réalisations multiples et conditions

Remarquons que les conditions logiques de réalisation peuvent également s'appuyer sur les variables attachées aux membres droits des productions enrichies. Les variables libres apparaissant dans ces formules logiques sont interprétées de la même manière que précédemment dans les contraintes de bonne formation. Une réalisation est dite *possible* lorsque sa pré-conditions est localement satisfaite : tout comme pour la validité d'une structure abstraite, cette notion dépend des choix effectués dans la dérivation de la structure abstraite. L'ensemble des réalisations possibles pour une structure abstraite est naturellement l'union des réalisations possibles pour chacune de ses dérivations valides : cet ensemble n'est pas toujours un singleton, et peut éventuellement être vide.

Alternativement, une production $A \to t$ contrainte par un ensemble de formules Φ et munie d'une règle de linéarisation $\{\phi_1 \longrightarrow M_1 \dots \phi_n \longrightarrow M_n\}$ peut être interprétée simplement comme un ensemble de n productions $A \to t$, chacune contrainte par $\Phi \cup \{\phi_i\}$ et munie de la règle de linéarisation M_i.

3.4.3 Requêtes logiques

Finalement, nous souhaitons explorer la possibilité de prendre certaines libertés par rapport aux dérivations lors de la linéarisation, en construisant la réalisation d'une production à partir de réalisations attachées à des positions arbitrairement distantes dans la structure abstraite. Ces positions sont spécifiées au moyen de la même signature logique que précédemment, et les constructions résultantes sont appelées des *requêtes logiques*.

Ces requêtes s'avèrent particulièrement utiles lorsque le regroupement des syntagmes dans une phrase ne coïncide pas avec leurs relations sémantiques ; leur emploi apparaît cependant questionnable, dans la mesure où elles tendent à briser la corrélation existante entre la dérivation d'une structure abstraite et sa réalisation. Nous justifions leur existence par le fait que les contraintes de bonne formation des structure abstraites ont la possibilité de quantifier (grâce aux relations construites à partir d'expressions régulières) sur des nœuds arbitrairement distants de la production associée. Dans de tels cas, la réalisation attachée à ces nœuds distants peut impacter directement la réalisation de cette production.

En d'autres termes, nous considérons que la structure abstraite est soutenue par les contraintes logiques qui lui donnent forme, et non par la seule grammaire d'approximation. Cette situation est illustrée dans notre modélisation du mouvement (page 91), où l'usage de réalisations distantes est corrélé avec des contraintes logiques requérant l'existence des nœuds correspondants. Bien qu'il soit possible d'obtenir dans ce cas les mêmes réalisations en l'absence de requêtes logiques (par l'intermédiaire du lambda-calcul), les règles de linéarisation résultantes sont significativement plus complexes, nuisant à la maintenabilité de la grammaire.

La figure 3.12 illustre l'emploi d'une requête logique pour simplifier la réalisation associée à certaines structures de notre langage. Nous étendons la seconde linéarisation esquissée dans la section 3.4.1, qui évalue les expressions dénotées par nos structures abstraites, en donnant une règle de linéarisation pour la production p_2. Dans les deux premiers cas, cette règle de linéarisation applique la réalisation de l'opérateur aux réalisations de ses opérandes (en tenant compte de l'optionnalité de y). Dans le dernier cas, la réalisation associée à un nœud v est substituée à l'interprétation usuelle : ce choix n'est possible que si le nœud requis v satisfait la pré-condition $\mathit{chemin\_neutre}(r, v)$. Dans le but de rendre immédiatement visible dans les règles de linéarisation le fait qu'une variable x est issue d'une requête logique, nous utiliserons une police distincte ($\mathbf{x}$) pour la mettre en valeur.

La condition logique *chemin_ neutre* utilisée pour requérir le nœud v dénote l'existence d'une série d'opérations arithmétiques ou logiques neutres entre r et v, comme par exemple $1 * x$, $x - 0$, $x \wedge \mathit{vrai}$, *etc.* ; dans un tel cas, l'interprétation de r est identique à celle de v, quelle que soit la distance qui les

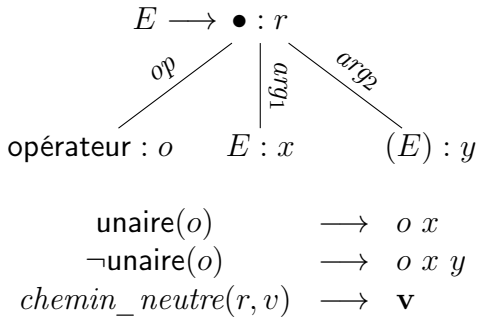


FIGURE 3.12 – Exemple de requête logique

sépare. Une telle condition peut être exprimée au moyen de notre langage logique comme illustré par la table 3.2. Ainsi, dire qu'il existe un chemin neutre entre r et v équivaut à dire que r domine v, et que pour toute expression e (qu'elle soit arithmétique ou logique) située entre r (inclus) et v (exclu), il existe trois nœuds o, a et n désignant respectivement l'opérateur de e et ses deux opérandes ; la notion d'opérande d'une expression étant encodée par la relation $arg(x, y)$, définie immédiatement en dessous. Par suite, a est l'opérande qui domine v, et n doit alors être un élément neutre pour l'opérateur o, ce que dénote la disjonction finale : 1 est neutre pour la multiplication, 0 pour l'addition, vrai pour la conjonction et faux pour la disjonction ; et 0 est neutre à droite d'une soustraction.

$$
\begin{array}{rcl}
\mathit{chemin\_neutre}(r, v) & \triangleq & \mathrm{dom}^+(r, v) \wedge \forall e.\, \mathrm{dom}^*(r, e) \wedge \mathrm{dom}^+(e, v) \Rightarrow \\
& & \exists o, a, n.\, \mathit{op}(e, o) \wedge \mathit{arg}(e, a) \wedge \mathrm{dom}^*(a, v) \wedge \mathit{arg}(e, n) \wedge \\
& & \big(\quad \mathbf{fois}(o) \wedge \mathbf{un}(n) \\
& & \vee \quad \mathbf{plus}(o) \wedge \mathbf{z\acute{e}ro}(n) \\
& & \vee \quad \mathbf{et}(o) \wedge \mathbf{vrai}(n) \\
& & \vee \quad \mathbf{ou}(o) \wedge \mathbf{faux}(n) \\
& & \vee \quad \mathbf{moins}(o) \wedge \mathbf{z\acute{e}ro}(n) \wedge \mathit{arg}_2(e, n) \quad \big) \\
\\
\mathit{arg}(x, y) & \triangleq & \mathit{arg}_1 \mid \mathit{arg}_2 \ (x, y)
\end{array}
$$

TABLE 3.2 – relation entre deux positions séparées par des opérations neutres

Il importe de remarquer que, utilisé sans restriction, ce mécanisme de requêtes logiques permet de décrire des réalisations circulaires, en requérant (directement ou indirectement) la réalisation associée à un nœud pour calculer la réalisation de ce même nœud. Plus généralement, le fait d'employer de telles requêtes, de préférence à une composition linéaire des non-terminaux respectant la structure de la dérivation, induit un risque de sanctionner des réalisations qui ignorent une fraction arbitrairement grande de la structure abstraite ou, à l'inverse, qui dupliquent accidentellement les réalisations associées à certains nœuds.

Ces problèmes peuvent être mitigés en imposant une relation d'ordre sur les positions de la structure abstraite, et en autorisant uniquement dans la réalisation d'une production les requêtes logiques portant sur des positions strictement inférieures à la position courante ; ce processus peut être vu comme comme l'ajout d'un ordre d'évaluation sur les nœuds de la structure abstraite pour la linéarisation. L'ordre partiel découlant des relation de dominance entre les positions du terme constitue un candidat naturel pour une telle relation, éliminant de façon plausible les problèmes de circularité.

3.4.4 Calcul d'une linéarisation

Nous expliquons maintenant comment obtenir une construction effective de toute linéarisation associée à une structure abstraite. Nous nous appuyons pour ce faire sur un ensemble de résultats issus de la littérature portant notamment sur les grammaires de remplacement d'hyper-arêtes (*hyperedge replacement grammars* ou HR, *cf.* Bauderon et Courcelle [1987]) et les grammaires catégorielles abstraites du second ordre ($\mathbf{ACG}(2, n)$).

Une réalisation se construit en partant d'un arbre de dérivation (enrichi par les conditions de bonne formation) associé à une structure abstraite valide. Cet arbre de dérivation est également décoré par les conditions de réalisation qu'il satisfait, selon le même processus que celui décrit dans la section 3.3.3 : les formules logiques sont explicitement inscrites sur les nœuds de l'arbre où elles sont instanciées. Par suite, le mécanisme des requêtes logiques est traité en ajoutant, dans l'arbre résultant, une arête reliant le nœud qui effectue la requête à un de ses descendants susceptible d'être le nœud requis. Le graphe résultant est un graphe orienté sans cycle (ou DAG), représentant un arbre dans lequel certains sous-arbres sont partagés ; le dépliage de ce DAG permet d'obtenir la structure du lambda-terme qui réalise la structure abstraite de la manière attendue. Il suffit alors d'y remplacer chaque nœud interne par un lambda-terme associé par la linéarisation pour obtenir la réalisation de la structure abstraite.

La figure 3.13 illustre le dépliage d'un DAG obtenu en ajoutant une arête correspondant à une requête logique dans une structure abstraite. En haut de la figure se trouve une structure abstraite, décorée par une arête supplémentaire étiquetée « REQ » qui relie deux positions séparées par un chemin neutre. Ce lien correspond à la requête logique $\mathit{chemin\_neutre}(r, v)$ figurant dans la règle de linéarisation donnée par la figure 3.12 et définie dans la table 3.2, et relie le second argument de la racine à une feuille (la requête garantit alors que sémantiquement, ces deux nœuds ont la même valeur). En bas de la figure se trouve le résultat du dépliage de ce DAG : le sous-arbre pointé par l'arête REQ est dupliqué et ajouté, et sa copie a pour nœud parent le nœud à l'origine de la requête. Remarquons que ce mécanisme de dépliage peut en général copier des sous-arbres de taille arbitraire (et que ces derniers peuvent eux-même contenir

des requêtes logiques, traitées de la même manière). La réalisation attendue de cette structure abstraite ignorerait ensuite l'ensemble du sous-arbre dominé par la première arête arg_2, à l'exception de la partie dominée par l'arête REQ : la valeur attribuée au second argument de cette expression est en effet **un**, puisque les opérations intermédiaires dans l'arbre ne modifient pas sa valeur (ce que garantit la formule *chemin_neutre*).

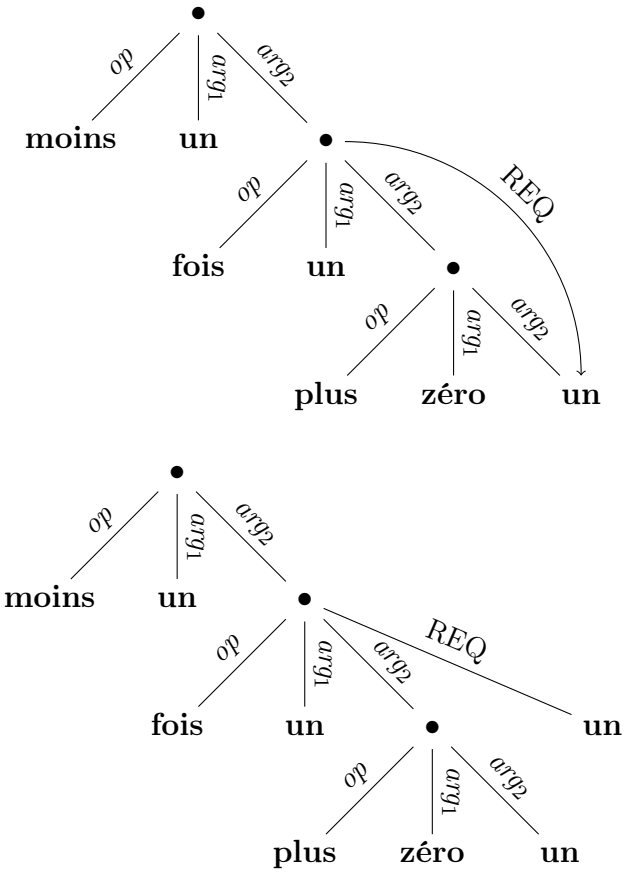


FIGURE 3.13 – Arbre avec partage (DAG) et son dépliage associé

Du point de vue des classes de langages impliquées, l'arbre de dérivation initial et définissable par MSOkS, et appartient donc à un langage régulier d'arbres. Par ailleurs, la classe de langages de graphes produite par les grammaires de remplacement d'hyper-arêtes inclut les langages réguliers d'arbres, et est notamment close par la classe des transductions dites MSO-définissables définies dans [*cf.* Courcelle et Engelfriet, 2011]. L'ajout des arêtes représentant les requêtes logiques constitue une telle transduction ; cependant le dépliage du DAG obtenu est susceptible de produire des copies dépassant le pouvoir d'expression des transductions MSO. La classe de langages d'abres ainsi produite est cependant obtenable par des transducteurs attribués avec anticipation (*look-ahead*)[*cf.* Bloem et Engelfriet, 1998], et donc par des ACG quasi-

linéaires du second ordre [*cf.* Kanazawa, 2009b]. Enfin, l'opération consistant à remplacer les nœuds internes (décorés par les conditions de réalisation qu'ils satisfont) par les lambda-termes que leur associe la linéarisation est un homomorphisme linéaire, et peut aisément être effectuée par une ACG du second ordre.

Il est à remarquer que les ACG quasi-linéaires ne partagent pas les propriétés des ACG décrites dans la section 2.4.2. Cependant, ce surcroît de complexité est essentiellement dû au pouvoir de copie potentiel de l'opération de dépliage de DAG qui permet de traiter les requêtes logiques. Comme les emplois linguistiques réels de ces dernières évitent la duplication des sous-termes impliqués, le coût algorithmique associé peut vraissemblablement être borné.

Sans effectuer en détail la construction que nous avons évoquée, nous détaillons maintenant ses principales étapes et les arguments qui les rendent effectives :

1. L'arbre de dérivation initial, décoré à la fois par les conditions de bonne formation et les conditions de linéarisations instanciées par chaque production, peut être construit en suivant exactement la procédure de compilation décrite dans la section 3.3.3. Celle-ci n'inclut que les conditions de bonne formation, mais les conditions de réalisation s'appuient sur les même variables qui étiquettent les productions, et peuvent donc également être rattachées à la racine des productions et intégrées dans un alphabet gradué pour figurer explicitement dans les termes d'un langage régulier.

2. L'arbre résultant est ensuite complété par des arêtes représentant les requêtes logiques. Celles-ci relient un nœud décoré par une règle de linéarisation comportant une requête logique à un de ses descendants qui satisfait la requête en question. L'ajout de ces arêtes peut être effectué par une transduction MSO-définissable, qui vérifie les informations présentes sur le nœud source, la satisfaction de la requête par le nœud cible (l'ensemble des requêtes logiques dans la linéarisation d'une grammaire est fini), et la relation de domination entre ces deux nœuds.

   De plus, l'emploi de paramètres dans le schéma de définition de cette transduction [*cf.* Courcelle et Engelfriet, 2011, p. 505] permet de ne sélectionner par la suite que les DAG où les requêtes logiques sont fonctionnelles (celles où le nœud qui effectue la requête est associé à un unique nœud requis parmi toutes les cibles possibles). L'ensemble de graphes résultants de cette transduction est définissable par une grammaire de remplacement d'hyper-arêtes.

3. Le dépliage des DAG résultants produit un langage d'arbres, définissable par une grammaire attribuée comme expliqué plus haut. Ces arbres reflètent la structure des lambda-termes qui sont des réalisations valides pour la structure d'origine.

4. Un réétiquetage de ces arbres permet de remplacer un nœud étiqueté par une condition de réalisation ϕ_i par son lambda-terme associé M_i. Le lambda-terme obtenu en considérant le terme M_i de chaque nœud, en y abstrayant les variables libres correspondant à ses arguments ou à ses requêtes logiques, et en l'appliquant aux lambda-termes associés à ses descendants immédiats forme alors une réalisation valide de la structure abstraite d'origine.

L'ensemble des réalisations ainsi décrites est donc obtenu par réétiquetage d'un langage d'arbres (de degré borné) défini par un transducteur attribué avec anticipation. Il est par conséquent définissable par une grammaire catégorielle abstraite quasi-linéaire du second ordre employant une signature de termes. Cette classe de langages est à son tour décidable ; et en l'absence de copie dans l'emploi des requêts logique, la linéarité de l'ACG résultante rend son langage reconnaissable en temps polynomial par rapport à la taille de l'énoncé, comme montré par Salvati [2005]. Il convient toutefois d'observer que, même en l'absence de copies arbitraires, la taille de la grammaire obtenue par ce biais sans optimisation, rend vraisemblablement l'analyse impraticable pour des applications de TAL. Nous reviendrons sur ce point en fin de chapitre.

3.5 Langage de macros pour les linéarisations

Plusieurs des linéarisations les plus complexes apparaissant dans nos modélisations contiennent des réalisations alternatives qui varient subtilement les unes par rapport aux autres (c'est en particulier le cas pour le traitement sémantique de l'antécédence, détaillé page 95). Dans ces cas, les lambda-termes dénotant les réalisations alternatives ne diffèrent que par un sous-terme (ou un contexte) de taille réduite, le reste demeurant identique entre tous les cas. Par souci de concision, nous souhaitons rendre explicites les généralisations que ces répétitions suggèrent, en factorisant les termes ou contextes identiques.

À cet effet, nous proposons un langage de macros mêlant de façon transparente les lambda-termes issus du langage concret avec les réalisations multiples et conditions logiques introduits dans la section 3.4.2, et proposons un système de réécriture permettant d'interpréter les termes résultants comme des ensembles de termes conditionnés, similaires à ceux de la section précédente.

3.5.1 Définition et usage

La réalisation d'une production sera maintenant décrite par des lambda-termes simplement typés sur la signature représentant le langage concret, pouvant inclure des conditions logiques et des ensembles de réalisations alternatives. Nous spécifions maintenant la syntaxe de ces nouvelles règles de linéarisation :

Définition 3.5. Soit $\mathcal{C}$ l'ensemble des constantes (typées) formant la signature du langage concret visé et $\mathcal{T}$ l'ensemble de ses types simples, puis $\mathcal{X} \times \mathcal{T}$ un ensemble dénombrable de variables typées, et $\mathcal{F}$ l'ensemble des conjonctions de formules logiques apparaissant dans les règles de linéarisation de la grammaire.

Le nouvel ensemble $\mathscr{R}$ des *règles de linéarisation* est formé à partir de celui des lambda-termes sur la signature $\mathcal{C}$, auxquels nous ajoutons deux constructions supplémentaires pour les réalisations conditionnelles et alternatives :

— Si $c \in \mathcal{C}$, alors $c \in \mathscr{R}$ (*constante*).
— Si $x^\alpha \in \mathcal{X} \times \mathcal{T}$, alors $x \in \mathscr{R}$ (*variable*).
— Si $M \in \mathscr{R}$, $x^\alpha \in \mathcal{X} \times \mathcal{T}$, alors $(\lambda x^\alpha.M) \in \mathscr{R}$ (*abstraction*).
— Si $M, N \in \mathscr{R}$, alors $(MN) \in \mathscr{R}$ (*application*).
— Si $M \in \mathscr{R}$ et $\phi \in \mathcal{F}$, alors $(\phi \longrightarrow M) \in \mathscr{R}$ (*condition*).
— Si $M_1, \ldots, M_n \in \mathscr{R}$, alors $\{M_1 \ldots M_n\} \in \mathscr{R}$ (*ensemble*).

Une règle de linéarisation ne contenant que des constantes, variables, abstractions et applications est qualifiée de *pure*.

Les deux dernières constructions de la définition précédente dénotent respectivement les réalisations conditionnelles et les réalisations alternatives. Leur interprétation découle naturellement de celle donnée plus haut pour les règles de linéarisation : l'interprétation d'une condition $\phi \longrightarrow M$ est que le sous-terme M est une réalisation possible si et seulement si ϕ est satisfaite ; et celle d'un ensemble $\{M_1 \ldots M_n\}$ est que chaque M_i est une réalisation possible pour $i \in [n]$. Remarquons que l'ensemble $\mathscr{R}$ résultant inclut en particulier les règles de linéarisation simples présentées dans la section 3.4.2, qui se présentent comme des ensembles de conditions portant sur des règles pures.

Afin d'illustrer l'usage de cette syntaxe, nous considérons tout d'abord le cas où l'on cherche à factoriser un contexte identique entre plusieurs alternatives, et où seul un sous-terme varie d'un cas à l'autre. Si C est le contexte à factoriser et $S = \{M_1 \ldots M_n\}$ l'ensemble de ses sous-termes possibles, la règle de linéarisation correspondante est simplement $C[S]$. Par exemple, la règle de linéarisation suivante permet d'employer au choix n'importe lequel des trois symboles $\times$, $*$ ou . pour construire la réalisation du produit de deux termes x et y :

$$\mathbf{fois}(o) \longrightarrow x \left\{ \begin{array}{c} \times \\ * \\ . \end{array} \right\} y$$

Elle abrège, dans la notation de la section précédente, la règle de linéarisation suivante :

$$\left\{ \begin{array}{lcl} \mathbf{fois}(o) & \longrightarrow & x \times y \\ \mathbf{fois}(o) & \longrightarrow & x * y \\ \mathbf{fois}(o) & \longrightarrow & x \ . \ y \end{array} \right\}$$

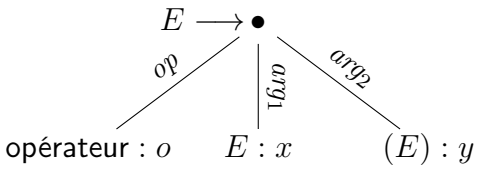
Par suite, dans le cas où l'élément à factoriser est un sous-terme, nous exploitons la sémantique usuelle du lambda-calcul pour en revenir au cas précédent. Ainsi, la règle $S\ N$ où $S = \{\phi_1 \longrightarrow \lambda x.M_1 \ldots \phi_n \longrightarrow \lambda x.M_n\}$ permet de

factoriser N dans chacun des termes M_i. Cette règle s'interprète, en concluant par une β-réduction, en $\{\phi_1 \longrightarrow M_1[x \leftarrow N] \ldots \phi_n \longrightarrow M_n[x \leftarrow N]\}$ (en supposant que x est une variable fraîche lors de la factorisation). Afin de ne pas alourdir nos réalisations avec de multiples abstractions toutefois, nous abrégerons par la suite la construction précédente en utilisant la notation :

$$\left\{ \begin{array}{c} \phi_1 \longrightarrow M_1 \\ \ldots \\ \phi_n \longrightarrow M_n \end{array} \right\} \textit{où } x = N$$

De plus, dans ces cas, x sera usuellement un nom de variable décrivant la fonction du terme N dans la réalisation. Cette possibilité de nommer les sous-termes factorisés constitue un avantage supplémentaire pour la modélisation linguistique et le développement de grammaires, en offrant l'opportunité d'expliciter la généralisation ainsi décrite.

La figure 3.14 illustre un emploi de ce mécanisme en proposant une règle pour le parenthésage des expressions, améliorant la règle proposée plus haut par la figure 3.11. Elle autorise l'omission des parenthèses autour d'un opérateur dont la priorité est forte (priorité $+$), tout en permettant dans tous les cas une réalisation incluant les parenthèses (dénotées par les constantes op et cp). L'expression elle même, incluant l'opérateur et ses opérandes, est factorisée par la variable *expr* entre ces deux cas, évitant la répétition de l'ensemble de réalisations qui apparaît à droite du signe $=$.



$$\left\{ \begin{array}{lcl} \textit{Vrai} & \longrightarrow & \lambda a^*.\mathsf{op}\ (\textit{expr}\ (\mathsf{cp}\ a)) \\ \mathsf{priorité}\ +(o) & \longrightarrow & \textit{expr} \end{array} \right\}$$

$$\textit{où } \textit{expr} = \left\{ \begin{array}{lcl} \mathsf{unaire}(o) & \longrightarrow & \lambda a^*.o\ (x\ a) \\ \neg\mathsf{unaire}(o) & \longrightarrow & \lambda a^*.x\ (o\ (y\ a)) \end{array} \right\}$$

FIGURE 3.14 – Ajout d'une règle de parenthésage des expressions

Enfin, nous emploierons parfois l'expression « *sinon* » en guise de condition logique. Cette condition particulière n'apparaîtra qu'en tant que dernier élément dans un ensemble de conditions : elle est associée à une linéarisation « par défaut », sélectionnée lorsqu'aucune autre condition de cet ensemble n'est vérifiée. En d'autres termes, *sinon* s'interprète comme la conjonction des négations

des autres conditions logiques de l'ensemble, comme illustré ci-dessous :

$$\left\{\begin{array}{lcl} \phi_1 & \longrightarrow & M_1 \\ & \dots & \\ \phi_n & \longrightarrow & M_n \\ sinon & \longrightarrow & N \end{array}\right\} \quad \Leftrightarrow \quad \left\{\begin{array}{lcl} \phi_1 & \longrightarrow & M_1 \\ & \dots & \\ \phi_n & \longrightarrow & M_n \\ \neg\phi_1 \wedge \dots \wedge \neg\phi_n & \longrightarrow & N \end{array}\right\}$$

3.5.2 Interprétation

Nous interpréterons les nouvelles règles de linéarisation de $\mathscr{R}$ en deux étapes : tout d'abord, nous réécrirons ces règles en interprétant les conditions et les ensembles ($\longrightarrow$ et $\{\}$) qu'elles contiennent, jusqu'à obtenir des termes en forme normale. Ces derniers auront la forme $\{\phi_1 \longrightarrow M_1 \dots \phi_n \longrightarrow M_n\}$ où chaque M_i est un terme pur (ne contenant ni ensemble, ni condition). La règle de linéarisation résultante sera ensuite interprétée de la même manière que dans la section précédente.

Nous détaillons maintenant le système de réécriture permettant d'interpréter les constructeurs $\longrightarrow$ et $\{\}$ dans les règles de linéarisation. Celui-ci se compose de deux relations $\rightarrow_C$ (pour les conditions logiques) et $\rightarrow_E$ (pour les ensembles), décrites respectivement par les figures 3.15 et 3.16. Intuitivement, ces systèmes permettent de faire « remonter » les constructeurs qu'ils réécrivent vers la racine du lambda-terme qui les inclut, en respectant l'interprétation suggérée dans la section 3.5.1.

$$\frac{\lambda x.\phi \longrightarrow M}{\phi \longrightarrow \lambda x.M}\, abs \qquad \frac{\phi \longrightarrow \psi \longrightarrow M}{\phi \wedge \psi \longrightarrow M}\, cond$$

$$\frac{(\phi \longrightarrow M)N}{\phi \longrightarrow MN}\, app,g \qquad \frac{M(\phi \longrightarrow N)}{\phi \longrightarrow MN}\, app,d$$

FIGURE 3.15 – Système de réécriture $\rightarrow_C$

Ainsi, dans le cas de $\rightarrow_C$, l'application ou l'abstraction d'un terme réalisé sous une certaine condition est elle-même réalisée sous cette condition (règles *abs*, *app*, *g* et *app*, *d*), et un terme réalisé sous deux conditions ϕ et ψ est réalisé sous la conjonction $\phi \wedge \psi$ de ces deux conditions (règle *cond*).

De même, la relation $\rightarrow_E$ dénote que l'application ou l'abstraction d'un ensemble de réalisation s'interprète comme l'ensemble des applications ou abstractions de chaque réalisation possible (règles *abs*, *app*, *g* et *app*, *d*). De façon transparente, un ensemble de réalisations conditionné par ϕ équivaut à l'ensemble de ces réalisations conditionnées chacune par ϕ (règle *cond*). Pour des raisons techniques qui seront apparentes dans la prochaine section, nous souhaitons également également exprimer le fait qu'un singleton équivaut au terme qu'il contient (règle *sgl*). Enfin, un ensemble de réalisations dont l'une

$$\frac{\lambda x.\,\{M_1 \dots M_n\}}{\{\lambda x.M_1 \dots \lambda x.M_n\}}\ abs \qquad \frac{\phi \longrightarrow \{M_1 \dots M_n\}}{\{\phi \longrightarrow M_1 \dots \phi \longrightarrow M_n\}}\ cond$$

$$\frac{\{M_1 \dots M_m\}\, N}{\{M_1 N \dots M_m N\}}\ app, g \qquad \frac{M\,\{N_1 \dots N_n\}}{\{M N_1 \dots M N_n\}}\ app, d$$

$$\frac{\{M\}}{M}\ sgl \qquad \frac{\{M_1 \dots M_{i-1}, \{M_{i,1} \dots M_{i,m}\}, M_{i+1} \dots M_n\}}{\{M_1 \dots M_{i-1}, M_{i,1} \dots M_{i,m}, M_{i+1} \dots M_n\}}\ ens$$

FIGURE 3.16 – Système de réécriture $\rightarrow_E$

des alternatives est un ensemble de réalisations s'interprète directement comme l'union de toutes les réalisations possibles (règle *ens*).

Enfin, la dernière composante de cette interprétation est la β-réduction usuelle sur les lambda-termes. Nous faisons ici le choix d'adopter une sémantique d'appels par valeur lors de la β-réduction : une application dont le membre gauche est une abstraction ne constitue un β-redex que lorsque son argument (à droite) est une règle de linéarisation pure, ne contenant ni ensemble ni condition.

3.5.3 Propriétés

Nous notons $\rightarrow_{\beta CE}$ le système de réécriture obtenu par combinaison de $\rightarrow_C$ et $\rightarrow_E$ avec la β-réduction usuelle restreinte décrite ci-dessus. Nous démontrons plusieurs propriétés de ce système de réécriture permettant d'obtenir l'interprétation souhaitée. Tout d'abord, nous montrons qu'il est localement confluent – c'est-à-dire que tout choix entre deux réécritures possibles dans un terme produit une paire de termes qui est dite *joignable* ; une paire de termes (R_1, R_2) étant joignable pour une relation $\rightarrow$ si il existe deux séquences de réécritures permettant de revenir au même terme : $R_1 \xrightarrow{*} R'$ et $R_2 \xrightarrow{*} R'$. Nous montrons ensuite que $\rightarrow_{\beta CE}$ est fortement normalisant, c'est-à-dire qu'il n'existe pas de terme pouvant être réécrit indéfiniment. Comme pour le lambda-calcul typé (*cf.* 2.2.3), la conjonction de ces deux propriétés induit la convergence de $\rightarrow_{\beta CE}$, c'est-à-dire l'existence et l'unicité d'une forme normale pour chaque règle de linéarisation. En outre, le système $\rightarrow_{\beta CE}$ possède également sous une forme faible la propriété de réduction du sujet : le type d'une réalisation (ou d'un fragment de réalisation) n'est pas modifié lors de sa réécriture.

Système de typage Afin de montrer ces résultats, nous dotons les termes de $\mathscr{R}$ d'un type selon le système donné par la figure 3.17. Ce système préserve les types classiques des lambda-termes à travers les conditions, et permet de typer un ensemble de termes de type uniforme ; en outre, un ensemble de termes de

types hétérogènes peut également être typé, au moyen d'un type atomique dédié ω, n'appartenant pas à $\mathcal{T}$. Cette dernière règle de typage est absorbante, en ce sens qu'un terme contenant un sous-terme de type ω ne peut qu'être un ensemble typé par ω (reflété par la condition $\alpha, \beta \neq \omega$, à l'exclusion des α_i).

$$\frac{x^\alpha \in \mathcal{X} \times \mathcal{T}}{x : \alpha}\, var \qquad \frac{M : \beta}{\lambda x^\alpha . M : \alpha \to \beta}\, abs \qquad \frac{M : \alpha \to \beta \quad N : \alpha}{MN : \beta}\, app$$

$$\frac{x \in C}{c : \tau(c)}\, cst \qquad \frac{M : \alpha \quad \phi \in \mathcal{F}}{\phi \longrightarrow M : \alpha}\, cond \qquad \frac{M_1 : \alpha \quad \cdots \quad M_n : \alpha}{\{M_1 \dots M_n\} : \alpha}\, ens$$

$$\frac{M_1 : \alpha_1 \quad \cdots \quad M_n : \alpha_n}{\{M_1 \dots M_n\} : \omega}\, ens, \omega$$

FIGURE 3.17 – Système de typage pour les règles de linéarisation ($\alpha, \beta \neq \omega$)

Le type ω reflète la possibilité de fournir des règles de linéarisation polymorphes : une production peut se voir associer une réalisation de type τ si une condition ϕ est vérifiée, et une autre réalisation de type σ si ψ est vérifiée ; c'est par exemple le cas lors de la linéarisation en sémantique des proposition relatives ou contrôlées dans le chapitre 4. Implicitement, une telle règle de linéarisation n'est acceptable que si $\phi \wedge \psi \Rightarrow \mathit{Faux}$: le non-terminal à gauche a alors le type τ dans le contexte de ϕ, et σ dans le contexte de ψ. Par suite, toute production dont le membre droit inclut ce symbole et dont les conditions sont compatibles avec ϕ et ψ doit tenir compte des différents types possibles, afin de ne produire que des termes bien typés. Le bon typage de toutes les réalisations possibles pour une structure abstraite donnée n'est pas garanti par le système décrit ici : par simplicité, nous avons fait le choix de rejeter comme invalides les réalisations qui ne sont pas typables.

Une autre possibilité aurait été de restreindre l'ensemble des grammaires possibles à celles dont les contraintes et règles de linéarisation ne sanctionnent que des réalisations bien typées. Compte tenu des avantages apportés par l'emploi de règles de linéarisation de type polymorphe (ω), ce choix requerrait l'emploi d'annotations de typage globales sur les non-terminaux, conditionnées par des formules logiques. Le système résultant n'entre pas dans le cadre de cette thèse, mais constituerait néanmoins un objet d'étude intéressant.

Réduction du sujet Nous commençons par montrer une forme restreinte de réduction du sujet, garantissant que la réduction d'un terme par $\to_{\beta CE}$ préserve son type. Nous qualifions cette propriété de restreinte, car elle repose sur l'emploi de variables typées à la Church : ainsi, une règle de linéarisation

peut contenir deux variables libres x^τ et x^σ, faisant toutes deux référence à la réalisation associée à un unique symbole non-terminal du membre droit. Une telle règle peut ne pas poser de problème, si x^τ et x^σ apparaissent sous deux conditions logiques distinctes (par exemple $\{\phi_1 \longrightarrow x^\tau; \phi_2 \longrightarrow x^\sigma\}$), mais est inacceptable si une des alternatives de réalisation combine les deux usages (par exemple $\phi \longrightarrow f\ x^\tau\ x^\sigma$).

Les termes contenant des typages incohérents seront filtrés ultérieurement, après la mise en forme normale des règles de linéarisation. Moralement, la propriété de réduction du sujet montrée ici garantit que le type d'un terme impliqué dans une règle de linéarisation n'est pas *modifié* lors de l'interprétation ; celle-ci peut cependant révéler des sous-termes incohérents, lesquels seront filtrés par la suite.

Théorème 3.1. *Si une règle de linéarisation $R \in \mathscr{R}$ est typable par $R : \alpha$ avec $\alpha \in \mathcal{T}$ et que $R \xrightarrow{*}_{\beta CE} R'$, alors $R' : \alpha$.*

Démonstration. Par induction sur la longueur de la réduction, nous montrons simplement ce résultat pour $R \rightarrow_{\beta CE} R'$.

Dans le cas où $R \rightarrow_\beta R'$, la démonstration suit le même schéma que celle de la réduction du sujet pour le lambda-calcul simplement typé.

Pour chacune des des règles $\rightarrow_C$ ou $\rightarrow_E$, le résultat est immédiat à partir de la forme des règles (voir figures 3.15, 3.16 et 3.17). Observons que la condition $\alpha \in \mathcal{T}$ est nécessaire : la règle *sgl* de $\rightarrow_E$ ne permet pas de conserver le typage $\{M\} : \omega$ si M n'est pas de type ω (elle préserve en revanche la possibilité de lui donner le type de M). □

Normalisation forte Nous montrons par la suite que toute séquence de réécritures par $\rightarrow_{\beta CE}$ est de longueur finie, par un argument similaire à celui de la normalisation forte du lambda-calcul typé. La preuve, assez longue, de ce résultat se trouve dans la deuxième section de l'annexe A.

Théorème 3.2. *Le système de réécriture $\rightarrow_{\beta CE}$ est fortement normalisant.*

Démonstration. Voir annexe A.2. □

Confluence locale Nous démontrons maintenant la confluence (locale) du système de réécriture interprétant les règles de linéarisation combiné à la β-réduction classique.

Lemme 3.3. *Le système de réécriture $\rightarrow_{\beta CE}$ est localement confluent.*

Démonstration. La confluence locale de $\rightarrow_\beta$ pour nos termes s'obtient de la même manière que pour le lambda-calcul (théorème 2.1).

Par suite, la confluence locale pour le système entier s'obtient en identifiant toutes les paires dites *critiques* de $\rightarrow_{\beta CE}$, formées lorsqu'un fragment d'un

terme de $\mathscr{R}$ peut se réécrire selon deux règles différentes, et en montrant qu'elles sont joignables. La confluence locale s'ensuit immédiatement[*cf.* Kirchner et Kirchner, 2006, p. 200].

Les paires critiques du système $\rightarrow_C$ sont dues à un choix entre *cond* et n'importe quelle règle de $\rightarrow_C$ ou à un choix entre app, g et app, d. Similairement, les paires critiques de $\rightarrow_E$ impliquent la règle *ens* avec n'importe quelle règle, ou les règles app, g et app, d ; la règle *sgl* entre également en conflit (immédiatement résolu) avec les autres règles du système. Ensuite, la combinaison de ces deux systèmes produit des paires critiques entre la règle *cond* de $\rightarrow_E$ et n'importe quelle règle de $\rightarrow_C$, ainsi qu'entre les règles app, g et app, d de ces deux systèmes. Enfin, la combinaison avec $\rightarrow_\beta$ introduit deux nouvelles paires critiques, entre la β-contraction et les règles *abs* de chacun des systèmes.

Observons que sémantique d'appels par valeur donnée plus haut pour la β-réduction interdit à un ensemble ou une condition d'appartenir au membre droit d'un β-redex, ce qui prévient l'existence de paires critiques non-joignables entre β et app, d.

La liste exhaustive des paires critiques de $\rightarrow_{\beta CE}$, ainsi que les stratégies permettant de les joindre, est donnée par l'annexe A.1. □

Forme normale

Théorème 3.4. *Le système de réécriture $\rightarrow_{\beta CE}$ induit une forme normale unique $|R|_{\beta CE}$ pour tout terme $R \in \mathscr{R}$.*

Démonstration. L'existence d'une forme normale est due au théorème 3.2, et son unicité est alors une conséquence immédiate du lemme 3.3. □

Conséquences Les propriétés listées dans cette section entraînent que les règles de linéarisation construites au moyen de notre langage de macros s'interprètent comme des ensembles de conditions de règles de linéarisation pures, tels que décrites précédemment. En effet, la mise en forme normale impose que tout terme contenu dans un ensemble est une condition portant sur un terme pur, ou un terme pur (sans quoi il contiendrait un redex selon le système $\rightarrow_{CE}$). L'unicité de cette forme normale et la normalisation forte impliquent que toute stratégie de réduction conduit à la même interprétation.

En outre, la duplication des termes et formules logiques apportée par les règles de réduction de $\rightarrow_E$ permet à nos règles de linéarisation de représenter de façon compacte des règles pures contenant de nombreux sous-termes et/ou formules logiques identiques.

3.6 Conclusion

À l'issue de ce chapitre, nous disposons d'un formalisme permettant de décrire des langages, formé de deux composantes essentielles.

La première est un mécanisme de description du langage à un niveau abstrait. La structure des énoncés résulte d'une grammaire régulière de termes, dite grammaire support, qui décrit grossièrement un sur-ensemble du langage d'arbre visé. Les productions de la grammaire support sont ensuite étiquetées et décorées par des formules logiques, lesquelles s'appuient sur une signature du premier ordre sur les termes enrichie par des relations régulières sur les chemins, à des fins de description linguistique. Ces formules sont instanciées et interprétées comme des contraintes de grammaticalité, filtrant les structures abstraites ne satisfaisant pas leurs contraintes associées. La signature sur laquelle elles s'appuient peut être exprimée à partir d'une signature du second ordre monadique à k successeurs, ce qui nous a permis de donner une caractérisation effective du langage régulier de termes résultant. Il est à noter que l'automate correspondant est susceptible d'être de très grande taille, même si les contraintes et la grammaire support qui le caractérisent sont assez brèves, en raison de l'expressivité de la logique utilisée pour formuler les contraintes. Cet obstacle peut toutefois être mitigé en substituant au processus de compilation naïve décrit dans la section 3.3.3 les techniques évoquées par Morawietz et Cornell [1997], en s'appuyant sur des outils logiciels dédiés tels que MONA [Klarlund et Møller, 2001].

La seconde composante de notre formalisme consiste en un processus de linéarisation, qui calcule une représentation concrète d'un énoncé du langage abstrait. Cette représentation, nommée réalisation, est obtenue à l'aide du lambda-calcul simplement typé ; elle consiste en un lambda-terme qui dénote un aspect sémantique ou syntaxique de l'énoncé. Ce terme est calculé de manière ascendante, en suivant la dérivation de la structure abstraite qu'offre la grammaire support, par le biais de règles de linéarisations rattachées à ses productions. Celles-ci combinent généralement les réalisations associées aux non-terminaux du membre droit (à la manière d'une grammaire d'attributs), et exploitent le langage logique que nous avons proposé pour les contraintes de grammaticalité, afin de sélectionner certaines réalisations en fonction du contexte de la structure abstraite. Nous avons également proposé un langage de macros étendant le lambda-calcul simplement typé, pour décrire de manière compacte des règles de linéarisation dont certains fragments dépendent de préconditions logiques. Nous avons ensuite montré son bon fonctionnement dans le cadre d'une sémantique d'appels par valeur. Enfin, nous avons introduit un mécanisme de requêtes logiques, permettant le déplacement à travers la structure abstraite de composants de sa réalisation : ces requêtes s'appuient une fois encore sur notre langage logique, et permettent de s'échapper partiellement du cadre offert par les dérivations de la grammaire support.

Les langages ainsi décrits sont effectivement des ACG du second ordre produisant des lambda-termes ; lesquels sont en correspondance immédiate avec les structures qu'ils représentent (mots, formules logiques, *etc.*). Bien que décidables en temps polynomial par rapport à la taille d'un énoncé, le problème

de la taille des représentations des langages ainsi obtenus se pose une fois de plus. Les résultats théoriques qui permettent leur construction n'offrent pas de garantie explicite en termes de complexité algorithmique au delà de la décidabilité. Cependant, l'existence de modèles approximatifs des langages naturels dans d'autres formalismes algorithmiquement abordables, ainsi que le nombre limité de concepts linguistiques à décrire, permettent de supposer qu'il est possible de fournir un modèle acceptable de la compétence linguistique sans requérir les pires cas de complexité atteignables par le formalisme. En outre, notre emploi de résultats et d'outils simples issus de l'informatique fondamentale offre un avantage supplémentaire, à savoir que la mise au point de techniques de compilation dont la complexité se limite à celle qui est réellement requise par une grammaire (et sa linéarisation) est étroitement liée aux connaissances existantes sur la structure et la complexité propres de la logique et du lambda-calcul.

Observons finalement que, bien que nous ayons choisi d'illustrer dans ce chapitre le fonctionnement du formalisme par un langage artificiel basique, certains des choix de conception effectués répondent directement à des besoins linguistiques spécifiques : c'est en particulier le cas des requêtes logiques, introduites pour faciliter la modélisation des phénomènes de mouvement. D'autres pans du formalisme, comme le langage de macros des règles de linéarisation, visent à faciliter l'ingénierie grammaticale. L'objectif directeur de ces choix est la capacité de proposer des descriptions concises et de haut niveau de divers phénomènes linguistiques, dans la continuité de l'emploi de notre langage logique sur les structures abstraites. Afin de confronter cet objectif au formalisme que nous avons décrit jusque là, le prochain chapitre illustre le fonctionnement de nos outils de description à travers un ensemble de modélisations linguistiques, incluant celles qui ont guidé notre travail de conception.