Publié dans Travaux de logique 11, 91-160, 1997,
source qui doit être utilisée pour toute référence à ce travail

LOGIQUE COMBINATOIRE, TYPES, PREUVES ET LANGAGE NATUREL

Jean-Pierre DESCLÉS

Notre compréhension intime de la logique combinatoire de Curry nous la fait penser comme une *logique des processus opératoires abstraits*, ces processus opératoires étant, par ailleurs, éventuellement implantables sur les supports physiques d'une machine informatique et exécutables par les organes de calcul de cette machine. La logique combinatoire thématise les *modes intrinsèques de composition et de transformation* des opérateurs, analysés comme des opérateurs abstraits, appelés *combinateurs*. Ces modes de composition et de transformation sont définis *indépendamment de toute interprétation* et donc indépendamment des domaines d'interprétation comme par exemple l'arithmétique, la logique des propositions et des prédicats, la construction de programmes, la linguistique, la psychologie ou ... la philosophie. Par son grand pouvoir d'abstraction et d'expressivité, la logique combinatoire tend à acquérir un rôle unificateur puisqu'elle nous permet de jeter des ponts assez solides entre des domaines qui sont *a priori* aussi éloignés l'un de l'autre que les fonctions récursives, l'analyse des paradoxes, les types intuitionnistes de Martin-Löf, la programmation fonctionnelle, la théorie des preuves, la logique linéaire de Girard, les domaines réflexifs de Scott, la théorie des catégories cartésiennes fermées et des T[Σ]-algèbres, la théorie des topoï géométriques ... ou les théories des opérateurs linguistiques de Z.S. Harris ou de S.K. Shaumyan. A propos des sciences du langage, nous pouvons donc légitimement nous poser les questions suivantes: les langues naturelles, en tant que systèmes de représentations symboliques entretiennent-elles quelque rapport sémiotique avec d'autres systèmes sémiotiques que sont certains

langages logiques et certains langages de programmation? Les analyses linguistiques, en particulier les analyses syntaxiques et les constructions des interprétations sémantiques des énoncés, ont-elles quelque chose à voir avec les preuves pensées comme des objets d'étude? Les structurations internes des langues naturelles ont-elles quelque rapport avec des structurations logiques et géométriques (théorie des topoï par exemple)?

Historiquement, la logique combinatoire a pris naissance avec les travaux de M. Schönfinkel[1] (1924) qui cherchait à construire une logique de la quantification (ou logique du premier ordre) fondée sur la seule notion primitive de fonction. La logique combinatoire a été développée essentiellement sous forme d'une «prélogique» par H. B. Curry (1930, 1958, 1972) qui cherchait d'une part, à formaliser l'effectivité opératoire sous forme de règles analogues à «modus ponens» et d'autre part, à étudier, par des procédés logiques, l'émergence des paradoxes du genre de celui de Russell. Rappelons que la règle de «modus ponens» correspond, dans la «déduction naturelle» de Gentzen, à la *règle d'élimination* du symbole d'implication «->»:

$$[e\text{->}] \qquad \frac{A\text{->}B,\ A}{B}$$

Cette règle signifie que des deux prémisses «A->B» et «A», on peut en inférer la conclusion «B». Ce genre de règle d'inférence sert de modèle aux règles formelles. Un autre règle relative à la flèche «->» correspond à la *règle d'introduction* qui se présente comme suit:

$$[i\text{->}] \qquad \frac{\begin{array}{c}[A]\\ .\\ B\end{array}}{A\text{->}B}$$

1 Voir la traduction en français avec commentaires dans *Mathématiques, Informatique et Sciences Humaines* 112, 1990, 5-26.

Cette règle signifie que si d'une hypothèse A on peut déduire B, alors on peut se «libérer de l'hypothèse A» en introduisant, mais à un niveau hiérarchique supérieur, l'expression «A->B». Cette règle permet de relier la relation métalinguistique de déduction entre A et B à l'expression linguistique «A->B» interne au langage; c'est une version du théorème «classique» de la déduction.

M. Schönfinkel et H.B. Curry ont introduit des opérateurs abstraits, appelés «combinateurs», qui construisent dynamiquement des opérateurs complexes à partir d'opérateurs plus élémentaires. Indépendamment, A. Church (1944) est l'inventeur d'un calcul, le λ–calcul, formé d'expressions symboliques qui représentent les mécanismes opératoires de procédures associées à des opérateurs ou à des fonctions appliquées à des opérandes. La logique combinatoire tout comme le λ-calcul sont des systèmes applicatifs car fondés sur l'opération d'application, considérée comme une opération primitive qui construit un résultat à partir d'un opérateur et d'un opérande. D'autres langages applicatifs existent. Mentionnons, par exemple, les différents langages de programmation fonctionnelle comme ML. Dans l'application d'un opérateur à un opérande, l'opérateur peut être dans certains cas interprété comme une fonction ensembliste, l'opérande étant l'argument de la fonction et le résultat l'image de cette fonction. Cependant, il y a une différence de point de vue. Dans la conceptualisation ensembliste, opérateur – ou fonction –, opérande – ou argument –, résultat – ou image – sont *co-existentiels*, ils sont situés hors du temps, ils sont donc *atemporels*; de plus, étant donné une fonction ensembliste, on peut asserter l'existence d'une image à partir d'un argument sans que nécessairement on puisse toujours savoir construire effectivement cette image. Dans la conceptualisation applicative, l'opérateur et l'opérande doivent être d'abord construits avant de pouvoir appliquer l'un à l'autre pour en déduire la construction effective du résultat; de plus, l'application est conçue comme une *opération dynamique* qui implique une *asymétrie temporelle* entre opérateur, opérande d'une part et résultat, d'autre part. Un système applicatif, fondé sur l'application, est ainsi un système à la fois *effectif*, puisque les objets manipulés sont tous constructibles, et *dynamique*, puisque l'application est une opé-

ration qui s'exécute dans une certaine temporalité. Aucune restriction n'est posée *a priori* sur l'opération d'application. En particulier, on peut appliquer un opérateur à lui-même sans sortir pour autant des systèmes applicatifs. Cependant, on peut restreindre l'expressivité du formalisme applicatif en ne considérant que différents types d'entités manipulées; dans ce cas, les systèmes deviennent typés et alors les opérateurs typés ne peuvent s'appliquer qu'à certains types d'opérandes. Dans un langage typé, un opérateur quelconque ne peut généralement pas s'appliquer à lui-même sans violer certaines contraintes sur les types. Cependant, on peut étudier les mécanismes d'auto-application et faire apparaître les raisons profondes qui conduisent à une situation paradoxale.

Énumérons quelques caractéristiques de la logique combinatoire

1) *La logique combinatoire est une logique «sans variables» (liées).* Or, comme on sait, la notion de «variable» est complexe, délicate à manier et ambiguë. Historiquement, Schönfinkel et Curry ont du reste cherché à construire une logique «sans variables» pour rendre plus effectifs les systèmes formels. Bien que le λ-calcul de Church et la logique combinatoire de Curry soient extensionnellement équivalents, les expressions de la logique combinatoire, en dépit des difficultés apparentes et superficielles, sont plus simples à manipuler que les expressions du λ-calcul qui doit faire l'exacte gestion des variables liées pour éviter, entre autres problèmes, les «télescopages des variables» (comme dans le calcul des prédicats) et les «effets de bord» (en programmation).

2) *La logique combinatoire permet de ramener toutes les manipulations à des règles effectives et explicites* aussi simples, dans leurs formes, que les règles de «modus ponens» (élimination de l'implication dans le calcul propositionnel) et d'abstraction (introduction de l'implication dans le calcul propositionnel). Toutes les «constantes logiques» (d'un côté, les combinateurs et d'un autre côté, les opérateurs illatifs de

connexion propositionnelle et de quantification des prédicats par exemple) sont introduites ou éliminées par des règles dont la forme est très exactement analogue aux règles d'introduction et d'élimination de la flèche «->». On obtient ainsi des présentations «dans le style de Gentzen» de la logique combinatoire (voir, entre autres, le livre de Fitch). Ce genre de présentation permet une discussion sur la nature des règles de la logique, aboutissant aux généralisations obtenues dans la logique linéaire de J.Y Girard.

3) *La logique combinatoire est puissante* puisqu'elle permet d'exprimer, entre autres, les opérations de l'arithmétique élémentaire et la définition des fonctions récursives; elle permet donc une conceptualisation de l'effectif et du calculable, selon la thèse de Church. Les théorèmes de Gödel trouvent également une expression dans ce cadre applicatif.

4) *La logique combinatoire a un pouvoir fondationnel,* puisqu'elle permet une analyse logique des paradoxes, aussi bien du paradoxe de Russell, que celui du Menteur ou de la construction impossible de la surjection de Cantor entre un ensemble et l'ensemble de ses parties. Du reste, Curry a construit son système pour «analyser logiquement» les conditions qui faisaient émerger les paradoxes; il a ainsi montré que la plupart des paradoxes apparaissaient en construisant un opérateur particulier (le «combinateur paradoxal») qui met en jeu l'itération d'une «diagonalisation». Par ailleurs, la logique combinatoire est un langage interprétatif pour les programmes informatiques (dans la sémantique dénotationnelle de Strachey-Scott). Par son modèle topologique du λ-calcul, D. Scott a également montré quelle était l'importance du λ-calcul et des opérateurs constructeurs de point fixe dans les études sémantiques des langages de programmation de haut niveau, où certaines fonctions sont définies récursivement par des équations.

5) *La logique combinatoire a un pouvoir expressif d'analyse sémiotique des écritures symboliques*, en particulier, elle permet de passer de la notion extrinsèque d'opération à celle intrinsèque d'opérateur (Desclés 1980). Plusieurs exemples d'analyses sémantiques de concepts peuvent être donnés; citons par exemple le prédicat complexe «de qui rien de plus grand ne peut être

pensé» qu'utilise Anselme de Canterbury (X^{e} siècle) dans son fameux «unum argumentum» du *Proslogion* (Desclés 1991).

6) *La logique combinatoire se trouve à un carrefour* de disciplines formelles comme les mathématiques (théorie du point fixe, catégories cartésiennes fermées et topologies de Grothendieck), la logique (théorie de la calculabilité effective, théorie des démonstrations; théorie des fondements); philosophie des systèmes formels; approche intuitionniste des mathématiques; informatique fondamentale (programmation fonctionnelle; sémantique des langages de programmation).

7) *La logique combinatoire et le λ–calcul ont trouvé et trouvent des terrains d'application dans des disciplines qui s'appuient sur des données empiriques*. Ces applications sont certes encore timides mais cependant bien réelles dans certains secteurs des sciences humaines: psychologie cognitive (par exemple: analyse par L. Frey d'opérations cognitives du groupement de Piaget); philosophie (analyse de concepts philosophiques), musicologie (analyse de partitions, comme celle du *Clavier bien tempéré* de Bach); économie (travaux de Dupuy). Depuis les premiers travaux sur les Grammaires Catégorielles, à la suite des réflexions logico-philosophiques de Husserl et de S. Lesniewski, et surtout depuis les recherches menées par S.K. Shaumyan sur la Grammaire Applicative (1965), puis de Geatch, de Montague, de Dowty, de Moortgat, de Steedman, ..., les formalismes applicatifs de la logique combinatoire et du λ-calcul (avec types) ont été explicitement mis en oeuvre, de façon significative, dans plusieurs théories linguistiques: Grammaires Catégorielles étendues (avec des systèmes inférentiels de types fonctionnels de Lambek et une interprétation sémantique dans le λ-calcul par Moortgat, Steedman), syntaxe catégorielle et sémantique dénotative de la Grammaire Universelle de Montague, Grammaire des opérateurs et opérandes de Z. Harris (1983), Grammaire Applicative Universelle de S.K. Shaumyan (1977, 1987), Grammaire Applicative et Cognitive (Desclés 1990), ... D'autres linguistes, par exemple: A. Culioli ont également entrevu l'importance de la logique combinatoire pour la linguistique:

> On ramènera toutes les opérations unaires de prédication (à l'exclusion des transformations de composition de lexis) à une application (...) on ira jusqu'au bout de l'analyse, en y adjoignant [en en dérivant] une théorie des prédicats. (...) le linguiste aura parfois des concepts-clés à portée de main, parfois l'élaboration sera lente (ainsi en est-il de l'utilisation de la topologie en linguistique ou encore de la logique combinatoire) ...

Les questions qui nous préoccupent depuis quelques années peuvent se formuler de la façon suivante:

(i) Les langues naturelles étant des systèmes sémiotiques particuliers, peuvent-elles être pensées et organisées comme des langages applicatifs?

(ii) Est-il licite *a priori* de faire appel, pour une analyse linguistique, aux formalismes applicatifs de préférence à d'autres formalismes (certains étant mieux connus et plus utilisés comme les systèmes fondés sur la concaténation et sur les mécanismes d'unification de traits ou ceux qui, depuis quelques années, relèvent des réseaux connexionnistes)?

(iii) Les principales opérations linguistiques dégagées par les linguistes (comme les opérations de prédication, de détermination, de thématisation, d'énonciation, ...) peuvent-elles prendre une forme opératoire effective dans un langage applicatif?

(iv) Quelles sont les limites des formalismes applicatifs pour la description adéquate et fine des mécanismes langagiers?

(v) Quelles sont les relations entre la construction de la signification encodée par un énoncé et la théorie de la démonstration puisque certains travaux actuels explorent, depuis quelques années avec profit semble-t-il, cette voie?

(vi) Quelles conséquences peut-on déduire de l'analyse applicative des langues pour un traitement informatique des langues (par la programmation fonctionnelle, de nature applicative), à la fois sur le plan théorique et pratique?

(vii Les langages applicatifs sont-ils utilisables pour représenter plus adéquatement que les systèmes actuels – par

exemple, les réseaux sémantiques ou les graphes conceptuels de Sowa ou encore les modèles de Kamp – les connaissances et les représentations cognitives?

(viii) Les langages applicatifs sont-ils utilisables pour une formalisation, au moins partielle, des conceptualisations élaborées par les Grammaires Cognitives?

(ix) Quelles propriétés des formalismes applicatifs font progresser notre connaissance sur le fonctionnement du langage?

(x) Comment les langages applicatifs peuvent-ils nous aider à appréhender, représenter et manipuler des représentations cognitives hypothétiques qui sont non directement observables et par conséquent (re-)constructibles seulement par des processus abductifs explicites?

Quelques *points de vue sur les langues* peuvent justifier *a priori* l'utilisation du λ-calcul et de la logique combinatoire:

1. L'opération de base qui est constitutive des expressions linguistiques est l'application et non la concaténation.
2. Les unités des langues sont des unités opératives (ou fonctionnelles).
3. Les unités linguistiques sont des opérateurs de différents types.
4. La «currification» des opérateurs a une interprétation linguistique ainsi que, peut-être l'analogie de Curry-Howard.
5. La construction de l'interprétation fonctionnelle des énoncés peut être effectuée à partir, d'une part, une spécification sous forme d'une formule de types syntaxiques et, d'autre part, de preuves que cette formule est «cohérente», c'est-à-dire que cette formule est un théorème d'un système inférentiel sur les types.
6. L'introduction des opérateurs illatifs devrait permettre une meilleure analyse de la quantification dans les langues naturelles.

7. Les «prédicats complexes» sont construits à l'aide des combinateurs, ils sont essentiels pour l'analyse et la représentation sémantique d'un grand nombre d'énoncés.

Après avoir introduit quelques notations utiles, nous examinerons donc successivement chacun de ces différents points qui touchent au statut sémiotique des langues naturelles. Dès lors qu'elles sont comprises et acceptées après discussion, ces affirmations justifieraient, au moins à nos yeux, qu'un programme de recherche sur le langage puisse utiliser à bon escient la logique combinatoire en tant que système métalinguistique d'analyse des langues.

0. Notations

0.1. Application. L'application d'une expression Y, considérée comme un opérateur par rapport à une autre expression X, considérée comme un opérande de Y, construit un résultat Z. Nous désignons par «App(X, Y)» l'opération d'application. La relation entre cette opération «à exécuter» et le résultat Z obtenu «après exécution» est une réduction, notée: App(Y,X) β-> Z.

0.2. Présentation sémiotique de l'application. Le résultat de l'application de Y à X pourra être désigné par la simple juxtaposition «YX» de l'opérateur et de l'opérande. L'opération d'application est alors représentable par un arbre, dit «arbre applicatif»:

$$[\textbf{App}] \qquad \frac{Y \qquad X}{YX}$$

0.3. Expressions applicatives. L'ensemble des expressions applicatives est construit récursivement à l'aide de la règle:

> si Y et X sont des expressions applicatives, alors (YX) est une expression applicative.

Lorsqu'il n'y aura pas ambiguïté, les parenthèses externes seront omises. Il est clair que les deux expressions applicatives «X(YZ)» et «(XY)Z» (qui est égale à «XYZ») n'ont pas la même signification.

0.4. λ-expressions. Les λ-expressions sont des expressions applicatives particulières obtenues par un processus d'abstraction. L'ensemble des λ-expressions est défini comme suit:

(i) si X et Y sont des λ-expressions, alors (YX) est une λ-expression;

(ii) si X et Y sont des λ-expressions, et si x est une variable libre qui présente des occurrences dans Y, alors λX.Y est une λ-expression.

La λ–expression «λx.Y» est un opérateur qui a été obtenue par abstraction de «x» à partir de «Y». L'application de l'opérateur λx.Y à un opérande Z construit le résultat, noté «Y[x :=Z]», obtenu en substituant Z à *toutes les occurrences libres* de x dans Y, ce que nous désignons par la β-réduction:

$$((\lambda x.Y)\ (Z)\ \beta\text{->}\ Y[x :=Z].$$

Une présentation rigoureuse des λ-expressions et du λ–calcul associé consisterait à prendre certaines précautions sur la «gestion des abstractions» pour éviter les télescopages.

0.5. Expressions typées. Dans un système applicatif typé, les opérateurs, les opérandes et les résultats sont typés. Une entité X à laquelle est assigné un type α (soit par assignation initiale, soit par inférence) sera notée, en suivant la littérature actuelle: $[X :\alpha]$[2]. Désignons par la notation infixée «α->β» le type fonctionnel «de α vers β»[3]. L'opération d'application, dans un système applicatif typé, est représentée par l'arbre applicatif typé:

---

2 Une notation plus appropriée serait [α : X] ou: 'αX' notation de Curry qui considère le type comme un opérateur externe ou internalisable dans le système.

3 La notion préfixée 'Fαβ' de Curry est beaucoup plus commode mais la littérature actuelle ne l'a pas suivi sur ce point.

$$\textbf{[App}_t\textbf{]} \qquad \frac{[Y : \alpha \text{->} \beta] \qquad [X : \alpha]}{[YX : \beta]}$$

Un opérande absolu qui ne peut jamais être opérateur sera considéré comme un opérateur zéro-aire. Toutes les entités (typées ou non) d'un système applicatif sont donc des opérateurs. Chaque opérateur agit successivement sur éventuellement plusieurs opérandes pour construire un résultat (par exemple en linguistique un énoncé). Lorsqu'un, deux, ... n opérandes sont nécessaires pour qu'un opérateur construise une unité du système, on dit que l'opérateur est unaire, binaire, ... n aire.

Une λ-expression typée est construite comme suit: soit x une variable libre de type α et Y une expression de type β, la λ-expression λx.Y que l'on peut construire à partir de x et de Y est de type α->β, d'où l'abstraction:

$$\textbf{[Abst]} \qquad \frac{[Y : \beta], [x : \alpha]}{[\lambda x.Y : \alpha \text{->} \beta]} \qquad \text{(avec x une variable qui a des occurrences dans Y)}$$

Soient $\alpha_1, \ldots \alpha_n$ des types. Le type *produit cartésien* sera noté: α_1 x ...x α_n. Si l'on assigne les types α_i à chaque expression Y_i (i=1, ...n), alors nous avons la règle d'assignation du type «produit cartésien» au n-uple $<Y_1, \ldots, Y_n>$:

$$\textbf{[Cart}_t\textbf{]} \qquad \frac{[Y_1 : \alpha_1], \ldots\ldots\ldots, [Y_n : \alpha_n]}{[<Y_1, \ldots, Y_n> : \alpha_1 \text{ x } \ldots \text{x } \alpha_n]}$$

Remarque: Les notations choisies font apparaître explicitement les analogies entre d'une part l'application et l'élimination de '->' et d'autre part, entre l'abstraction et l'introduction de '->'. L'abstraction peut être exprimée, en logique combinatoire par l'introduction d'un combinateur. Cette analogie formelle est hautement significative, elle est à la base de l'isomorphisme de Curry-Howard, dont nous allons reparler, entre types et propositions, λ-expressions et preuves, spécifications et programmes.

1. Opération d'application en linguistique

Dans une analyse formelle des langues, deux points de vue se dégagent: (a) le point de vue strictement concaténationnel et (b) le point de vue applicatif. Selon le premier point de vue, les unités linguistiques sont des morphèmes et des mots qui sont juxtaposés à gauche ou à droite à d'autres unités linguistiques. Le linguiste doit alors définir les règles (c'est-à-dire la grammaire) qui engendrent, analysent et représentent tous, et rien que, les énoncés empiriques d'une langue. Dans ce cas, le linguiste travaille dans un monoïde libre engendré par l'opération de concaténation à partir des unités linguistiques de base. Selon cette conception, la grammaire devient un ensemble de règles de réécriture qui opèrent sur des suites du monoïde afin de caractériser exactement un langage formel comme une certaine partie d'un monoïde libre, ce langage correspond exactement aux suites finies qui sont en correspondance avec les phrases empiriques de la langue décrite. Les grammaires formelles syntagmatiques de N. Chomsky se sont constituées en adoptant explicitement ce point de vue. Les arbres syntagmatiques[4] sont directement associés à des classes d'unités linguistiques équivalentes (syntagmes nominaux, syntagmes verbaux, parties du discours, ...) organisées entre elles par des relations hiérarchiques d'emboîtement: telle classe (par exemple une phrase) est décomposée en une concaténation de classes plus fines (par exemple un syntagme nominal *suivi d'un* syntagme verbal). Les grammaires transformationnelles de Chomsky ne rompent pas avec la concaténation puisqu'elles continuent à opérer avec des suites concaténées de catégories et d'unités linguistiques minimales. Les modèles post-chomskyens (GPSG, HPSG, TAG) ne semblent pas avoir rompu non plus avec le point de vue concaténationnel. Ces derniers modèles ajoutent à l'opération fondamentale de concaténation, l'opération informatique d'unification.

Selon le second point de vue, les unités linguistiques sont des opérateurs ou des opérandes qui sont organisés dans un énoncé,

4 La structure d'arbre, en linguistique ou en informatique, est une structure que nous avons appelée dendron; deux ordres sont sous-jacents: un ordre hiérachique entre une catégorie et une sous-catégorie, un ordre concaténationnel entre sous-catégories et unités linguistiques minimales.

plus généralement dans un discours, par l'opération d'application. Le linguiste doit alors définir comment les unités s'organisent par l'application en précisant les contraintes sur l'application et éventuellement sur les types des unités linguistiques. L'application étant l'opération de base, elle est constitutive de toutes les expressions linguistiques représentées par des expressions applicatives qui sont alors encodées et présentées sous forme d'expressions linéaires en juxtaposant, selon des règles précises d'encodage, opérateurs et opérandes. La structure linéaire des énoncés est une structure non seulement superficielle[5] mais aussi apparente, elle ne correspond donc pas à une organisation significative des unités linguistiques. Selon cette approche non concaténationnelle, l'énoncé, plus généralement le discours, est une configuration organisée essentiellement par l'application (d'opérateurs à des opérandes), d'autres opérations pouvant venir s'ajouter aux opérations applicatives de base[6]. La juxtaposition des expressions linguistiques qui aboutit à une séquence apparente de symboles («la chaîne parlée») est simplement une *présentation* d'une organisation applicative sous-jacente que la grammaire, elle, doit révéler et analyser. De fait, cette présentation linéaire est imposée par le canal phonique qu'utilise le code oral. Quant aux langues des signes, qui ne font pas appel au même canal, ont, elles aussi, une structure applicative sous-jacente, elles diffèrent simplement des langues parlées par les *modes de présentation* imposées par les canaux utilisés mais les opérations constitutives des énoncés restent de même nature.

2. Unités opératives dans les langues

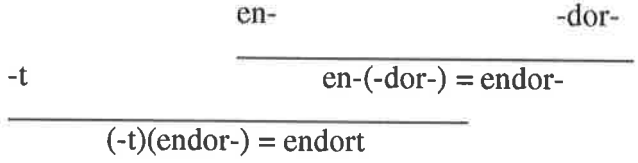
En adoptant le point de vue que les unités des langues sont des unités opératives (ou fonctionnelles), les unités linguistiques (selon le détail de l'analyse: des morphèmes, des mots, des

5 ... qui s'opposerait à une structure profonde; dans les premiers modèles chomskyens, la structure profonde a, elle aussi, une structure concaténationnelle.

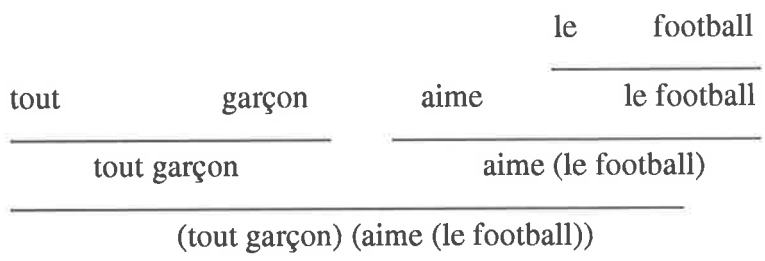
6 Les opérations prosodiques, dans les langues parlées, viennent se superposer aux autres organisations analysables par l'opération d'application. Cependant, il n'est pas exclu que les opérations prosodiques elles-mêmes puissent être analysées dans des termes applicatifs.

lexies – au sens de B. Pottier –, des syntagmes, des propositions, des phrases, des énoncés, des discours, ...) sont des opérateurs ou des opérandes: un opérateur s'applique à un opérande pour former une expression qui fonctionnera ou bien comme un opérande par rapport à un autre opérateur ou bien comme un opérateur par rapport à un autre opérande. Cette prise de position par rapport au langage n'est pas neutre puisqu'elle modélise toutes les unités linguistiques, non comme des symboles amorphes qui sont juxtaposés avec d'autres symboles amorphes, la signification ne surgissant que par l'assemblage final, mais comme des symboles opératoires qui agissent les uns sur les autres pour construire une signification qui est, elle-même, représentable en termes d'opérateurs et d'opérandes. Les mots ne sont donc pas de simples séquences juxtaposées de morphèmes grammaticaux et lexicaux mais des constructions décomposables en opérateurs et opérandes; les énoncés ne sont pas des séquences de mots juxtaposés mais des configurations dynamiques analysables en opérateurs et opérandes et représentables par des organisations géométriques (les arbres applicatifs).

Prenons un exemple très simple. Considérons le mot «endort»; il est décomposable en une organisation applicative représentable par l'arbre:



Prenons un deuxième exemple. Soit la phrase: *Tout garçon aime le football.* La décomposition applicative est donnée comme suit par l'arbre applicatif suivant:



De nombreuses théories linguistiques, tant en morphologie qu'en syntaxe, ont adopté, quelquefois implicitement, ce point de vue: les unités linguistiques des langues sont des symboles opératoires. Dans la Grammaire de Harris (dans ses différentes versions, depuis 1968), les unités linguistiques sont des opérateurs ou des opérandes, la décomposition d'une unité discursive (un discours) en opérateur ou opérande est essentielle dans la théorie. La Grammaire Applicative de S.K. Shaumyan (là aussi, dans ses différentes versions depuis 1963) considère explicitement les unités linguistiques (les sémions) comme des opérateurs ou des opérandes organisés par l'opération d'application d'où le nom de «Grammaire Applicative». La syntaxe structurale de Tesnière repose implicitement et en partie seulement sur cette notion applicative. En effet, dans cette approche, le verbe, unité centrale de la phrase, peut être considéré comme un opérateur qui s'applique (globalement) aux opérandes que sont les actants mais la notion de dépendance (par exemple, «l'article dépend du nom») n'exploite pas à fond la décomposition entre opérateur et opérande.

3. Grammaires catégorielles classiques

Les unités linguistiques ne sont pas homogènes, ni dans leurs morphologies, ni dans leurs fonctionnements. Il y a donc différents types d'unités linguistiques. Prenons le français par exemple. Une analyse, même très sommaire, fait reconnaître une différence entre les verbes et les noms, entre les noms et les adjectifs, entre les adjectifs et les adverbes, entre les prépositions et les conjonctions. Une des premières tâches du grammairien

revient donc à identifier les types des unités linguistiques manipulées: les types morphologiques, les types syntaxiques, les types sémantiques et, dans une approche cognitive du langage, les types cognitifs, ... Dans un certain nombre de cas, les types sont identifiées clairement par des classes d'unités linguistiques homogènes car ayant les mêmes propriétés formelles et un même fonctionnement dans les énoncés. Pour certaines unités cependant, la notion de type n'est pas toujours identifiable à une classe homogène d'unités, aussi n'assimilerons-nous pas *a priori* types et classes, la notion de type se prêtant à des abstractions opérationnelles et à des généralisations que n'autoriserait pas la notion de classe. Par exemple, la notion de type permet la construction de types, au moyen d'opérations précises – les constructeurs de types – à partir de types de base, puis d'établir des relations génétiques entre les types et parfois d'envisager un calcul inférentiel sur les types. Une entité typée est une entité à laquelle on a assigné un type; cela signifie que l'on connaît exactement: (i) quelles sont les opérations explicites que l'on peut faire sur cette entité; (ii) comment cette entité peut être représentée, dans certains cas même comment elle peut être physiquement implantée sur des supports matériels; (iii) comment cette entité peut être reliée à d'autres entités de type différent puisque connaissant son type, on connaît les relations génétiques que ce type entretient avec les autres types. Par exemple, le type **nat** des entiers naturels a un mode particulier d'implantation physique dans une machine informatique et ce mode diffère du type **réel** des réels. Si une entité est du type **bool** des entités booléennes, les opérateurs qui s'appliqueront à cette entité seront les opérateurs de négation, de conjonction, de disjonction, d'implication mais pas la soustraction arithmétique. Pour prendre une analogie, le type assigné correspond aux «dimensions» d'une expression du langage de la physique. Par exemple, dans l'expression linguistique du principe fondamental de la mécanique F = ma, les deux côtés du signe '=' doivent être de même type (principe d'isotypicalité), autrement dit la composition par multiplication d'une masse 'm' avec une accélération 'a' doit représenter une entité de même type qu'une force. En

prenant pour types de base: L (longueur), T (temps) et M (masse), le type d'une force est donc: M x (L/T^2).

En linguistique, les notions de type sémantique et syntaxique ont pris un sens technique et intuitif avec les Grammaires Catégorielles entrevues par Husserl. Ce formalisme a été ensuite développé en sémantique par Lesniewski, puis, en syntaxe, par Adjukiewicz, Curry, Bar Hillel pour être réexaminé et étendu par Lambek, Geatch, van Bethem, Bach et Partee, Morgaat, Steedman, ... Les types syntaxiques assignés aux unités et aux expressions linguistiques sont tous dérivés de types syntaxiques de base. Ces types syntaxiques (types de base et dérivés) représentent des catégories syntaxiques d'expressions linguistiques: noms, verbes intransitifs, verbes transitifs, adjectifs, ... Toutes les catégories syntaxiques en étant représentées par des types syntaxiques, sont analysées comme décomposables en catégories de base. On peut prendre selon les versions des Grammaires catégorielles, soit les deux catégories de base, «nom» et «phrase» ou soit les trois catégories de base, «syntagme nominal», «nom commun» et «phrase». Désignons ces dernières catégories de base par respectivement: 'N', 'NC' et 'P'. Les catégories dérivées sont construites à partir de deux constructeurs de types, les «divisions à droite et à gauche», désignées respectivement par '/' et '\'. On ajoute en général une autre constructeur, le «produit cartésien», désigné par 'x'.

L'ensemble $TYPE_{synt}$ de tous les *types syntaxiques dérivés* d'un ensemble de types syntaxiques de base $TYPE_{synt0}$ est défini récursivement comme suit:

(i) Les types de base de $TYPE_{synt0}$ sont des types de $TYPE_{synt}$;

(ii) Si: α et β sont des types de $TYPE_{synt}$
alors: 'α/β', '$\alpha\backslash\beta$', 'α x β' sont des types de $TYPE_{synt}$.

Les types 'α/β', '$\alpha\backslash\beta$' et 'α x β' se lisent respectivement «β sur α», «β sous α» et «α produit cartésien β».

A l'opération d'application [$\mathbf{App_t}$] sur les unités typées correspondent deux règles applicatives[7] [**AB>**] et [**AB<**]:

7 [**AB**] pour rappeler Adjukiewicz-Bar Hillel.

$$[\mathbf{AB>}]\ \frac{[X:\beta/\alpha] \quad [Y:\alpha]}{[X\text{-}Y:\beta]}> \qquad [\mathbf{<AB}]\ \frac{[Y:\alpha] \quad [X:\beta\backslash\alpha]}{[Y\text{-}X:\beta]}<$$

Dans la prémisse, les deux expressions X et Y ont des types assignés; dans la conclusion les expressions sont obtenues en juxtaposant, par l'opération notée '-', respectivement: l'opérateur X et l'opérande Y à sa droite[8] au moyen de la règle [**AB>**] et l'opérateur X et l'opérande Y à sa gauche au moyen de la règle [**AB<**]. Ces règles font apparaître l'interprétation fonctionnelle des types β/α et $\beta\backslash\alpha$.

Nous nous donnons également la règle d'assignation d'un type «produit» à un couple d'unités typées

$$[\mathbf{x_t}]\ \frac{[X:\alpha], [Y:\beta]}{[<X,Y>:\alpha \times \beta]}$$

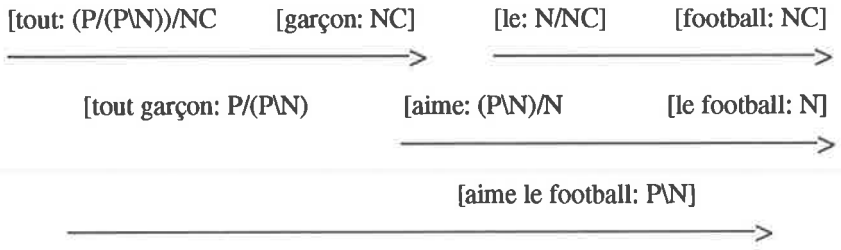
Cette règle nous invite à interpréter le type '$\alpha \times \beta$' comme un «produit cartésien fini».

Prenons, à titre d'exemple, l'ensemble $TYPE_0 = \{N, NC, P\}$ comme types de base; nous en déduisons des types dérivés qui représentent les catégories syntaxiques les plus classiques:

8 Les deux expressions en conclusion de la règle ne sont pas des expressions applicatives mais des expressions concaténées; on a désigné par '-' le symbole de concaténation.

Catégories syntaxiques	**Types syntaxiques**
nom propre	N
phrase	P
nom commun	NC
verbe intransitif	P\N
verbe transitif	(P\N)/N
adjectif	NC/NC
article	N/NC
adverbe	(P\N)\(P\N)
quantificateur	(P/(P\N))/NC

Assignons des types aux unités linguistiques de base. Nous pouvons reprendre l'analyse applicative de la phrase *tout garçon aime le football,* en assignant maintenant des types à chaque unité linguistique (ici, dans l'exemple, des mots lexicaux), nous obtenons, en utilisant les règles **[AB>]** et **[AB<]**, l'arbre suivant:



Les deux règles **[AB>]** et **[AB<]** sont analogues aux simplifications formelles du calcul sur les fractions ou encore aux simplifications formelles dans une vérification des «dimensions» d'une formule de physique. En se restreignant aux seuls types, les deux règles se ramènent formellement aux simplifications (à droite et à gauche) suivantes:

$$\frac{\alpha}{\beta} \times \beta = \alpha \qquad \beta \times \frac{\alpha}{\beta} = \alpha$$

Cette remarque a conduit J. Lambek à introduire un préordre de réduction entre les types, notés '=>', pour structurer le système des types TYPE engendré à partir des types de base $TYPE_0$ par les constructeurs de types /, \, x et en faire un système inférentiel (par la relation => entre types):

$< TYPE_0, /, \backslash, x, => >$.

Le **Calcul de Lambek** est un système de déduction sur les types où l'on se donne des schémas d'axiome et des schémas de règles qui permettent de déduire un ensemble de relations significatives entre types (les théorèmes du système de Lambek). En particulier, en se donnant, entre autres, comme schéma d'axiome et schéma de règle de déduction:

$$\alpha => \alpha$$

$$\frac{\alpha => \beta,\ \beta => \gamma}{\alpha => \gamma}$$

on en déduit, en tant que théorèmes de déduction sur les types, les analogues des règles applicatives **[AB>]** et **[AB<]**:

$$(\alpha/\beta) \times \beta => \alpha$$

$$\beta \times (\alpha\backslash\beta) => \alpha.$$

D'autres théorèmes sont également déduits comme, par exemple:

$$\gamma/(\alpha \times \beta) <=> \gamma/(\alpha/\beta)$$

$$\gamma/(\alpha \times \beta) <=> \gamma/(\beta/\alpha)$$

qui ont l'interprétation donnée par le principe de «currification» (voir plus loin). Ces nouvelles relations entre types étendent le calcul initial de Adjukiewicz et Bar-Hillel sur les types (dit sys-

tème **AB**) associé directement aux Grammaires Catégorielles classiques. Le Calcul de Lambek est donc plus riche que celui des catégories dans les Grammaires Catégorielles classiques.

Lorsque des types sont assignés aux unités linguistiques minimales, les Grammaires Catégorielles classiques substituent un calcul inférentiel sur les types à un calcul direct sur les unités linguistiques, de façon à vérifier que la formule de type qui spécifie l'énoncé est réductible au type caractéristique des énoncés: la formule du type de l'énoncé correspond au type P. A l'analyse catégorielle de la phrase *tout garçon aime une fille*, correspond donc une succession de réductions sur les types:

((P/(P\N))/NC) x NC x ((P\N)/N) x (N/NC) x NC =>
((P/(P\N)) x ((P\N)/N) x (N/NC) x NC =>
(P/(P\N)) x ((P\N)/N) x N =>
(P/(P\N)) x (P\N) =>
P.

Nous en déduisons le théorème: «le type initial est réductible au type P»:

((P/(P\N))/NC) x NC x ((P\N)/N) x (N/NC) x NC => P.

En inversant la relation de réduction sur les types, on obtient à partir du type de phrase P (et de l'axiome canoniquement associé: P => P), par expansions successives des types, une formule de type qui spécifie l'analyse syntaxique de toutes les phrases de même type. Ainsi, on obtient:

P => P
P => (P/(P\N)) x (P\N)
P => (P/(P\N)) x (((P\N)/N) x N)
P => (P/(P\N)) x (((P\N)/N) x ((N/NC) x NC)
P => (((P/(P\N))/NC) x NC) x (((P\N)/N) x ((N/NC) x NC).

Le type:

((P/(P\N))/NC) x NC x ((P\N)/N) x (N/NC) x NC

apparaît comme une *spécification syntaxique* d'une famille de phrases de même type c'est-à-dire organisées syntaxiquement de

la même façon: ces phrases sont toutes des instances du type donné.

Les notions d'application et de type permettent le déploiement des Grammaires Catégorielles. Le calcul inférentiel sur les types établit un lien entre l'analyse syntaxique applicative et la théorie de la preuve. En effet, effectuer une analyse applicative d'un énoncé, c'est construire explicitement, à partir d'une formule de type qui résulte d'une assignation de types aux unités minimales de l'énoncé, un arbre applicatif dont la racine est P, mais c'est aussi exhiber une preuve que la formule de type se réduit par la relation => au type P. Or, de l'arbre applicatif, on extrait une déduction sur les types et, réciproquement, d'une déduction sur les types on peut construire un arbre applicatif. Les présentations usuelles de ces Grammaires ne se sont cependant pas complètement dégagées de la conception concaténationnelle. Pour intégrer complètement la conception applicative avec celle des types, il faut étendre les Grammaires Catégorielles en exploitant la signification profonde des types fonctionnels. C'est ce que nous avons entrepris avec les Grammaires Catégorielles Combinatoires et Applicatives.

4. Analogie de Curry-Howard

4.1. Currification

Les Grammaires Catégorielles classiques ou étendues (dans le Calcul de Lambek) substituent, comme nous venons de le voir, un calcul inférentiel sur les types syntaxiques à un calcul direct sur les unités linguistiques agencées dans une phrase ou dans un énoncé. Lorsqu'on effectue un calcul sur les unités linguistiques en *se laissant guider* par les réductions de types, on construit progressivement l'énoncé en juxtaposant les expressions linguistiques à chaque fois qu'on opère une réduction sur les types selon les règles [**AB>**] ou [**AB<**] ou des règles plus complexes du Calcul de Lambek. Remarquons bien que si l'expression obtenue est une expression présentée par la juxtaposition linéaire des unités linguistiques, cette expression est

construite suivant une structure applicative qui décrit «l'histoire de sa construction». Les règles applicatives [**AB>**], [**AB<**] nous amènent directement à interpréter les opérateurs '/' et '\' comme des constructeurs de types fonctionnels et à interpréter l'opérateur 'x' comme un constructeur d'un type »produit cartésien». Ainsi, les types construis par les diviseurs *ont une signification fonctionnelle*: le type noté β/α exprime le type d'un opérateur qui construit à partir d'un opérande de type α une expression de type β; il en est de même du type $\beta\backslash\alpha$. En «oubliant» maintenant les positions à droite ou à gauche de l'opérande par rapport à l'opérateur, il apparaît que les types syntaxiques de la forme α/β et $\alpha\backslash\beta$ sont des types fonctionnels dont les instances sont nécessairement des opérateurs, avec la correspondance suivante:

type fonctionnel	diviseur à gauche	diviseur à droite
α->β	β/α	$\beta\backslash\alpha$.

Définissons directement les types fonctionnels et les types «produit cartésien» de la même façon que nous avions défini les types syntaxiques de $TYPE_{synt}$ (voir 3.). Soit $TYPE_0$ un ensemble de types de base. Les types fonctionnels et les types «produit cartésien» sont construits au moyen des règles suivantes:

(i) Les types de base de $TYPE_0$ sont des types de TYPE;

(ii) Si α et β sont des types de TYPE alors: 'α–>β' est un type de TYPE;

(iii) Si α et β sont des types de TYPE alors 'α x β' est un type de TYPE.

On peut considérer des opérateurs unaires, binaires, ... n-aires qui construisent des expressions de type β; ces opérateurs sont caractérisés par les types suivants:

type d'opérateurs unaires	α->β
type d'opérateurs binaires	α_1->(α_2->β)
...	
type d'opérateurs n-aires	α_1->(α_2->(...(α_n->β)...))

Puisque l'on peut former le produit cartésien de types 'α_1 x α_2 x...x α_n', le type d'un opérateur n-aire doit être également de la forme 'α_1 x α_2 x...x α_n->β'. Un tel opérateur, disons X_n, construit une expression de type β à partir de la donnée d'un n-uple d'opérandes de types respectifs α_1, ..., α_n. Plus précisément, nous avons en généralisant la règle d'application **[Appl]** au produit cartésien:

$$[\mathbf{App_{tn}}] \qquad \frac{[X_n : \alpha_1 \text{ x } \alpha_2 \text{ x...x } \alpha_n \text{->} \beta] \qquad [<Y^1,...,Y^n> : \alpha_1 \text{ x } \alpha_2 \text{ x...x } \alpha_n]}{[X_n(Y^1.....Y^n) : \beta]}$$

Quels sont les rapports entre les deux types d'un opérateur n-aire, c'est-à-dire entre d'une part, le type suivant: α_1 x α_2 x...x α_n->β et d'autre part, le type: α_1->(α_2->(...(α_n->β)...))? Sont-ils identiques, bien que construits différemment? Le principe de «currification» (ou de «curriage») répond à cette question en établissant une bijection entre ces deux types:

$$[\mathbf{CUR_t}] \qquad \alpha_1 \text{ x } \alpha_2 \text{ x...x } \alpha_n \text{->} \beta = \alpha_1 \text{->} (\alpha_2 \text{->} (...(\alpha_n \text{->} \beta)...)).$$

L'opération de «currification» d'un opérateur n-aire X_n, de type: α_1 x α_2 x...x α_n->β, revient à lui associer biunivoquement l'opérateur unaire $X_1' = \underline{Curr}(X_n)$, de type α->γ, où γ est le type fonctionnel α_2->(...(α_n->β)...). Le principe **[CUR_t]** de currification indique que l'on peut considérer des opérateurs n-aires comme des opérateurs unaires et que le type produit cartésien α_1 x α_2 x... α_n n'est plus nécessaire pour différencier le type des différents opérateurs unaires, binaires, ..., n-aires.

Quelles sont les raisons mathématiques du principe de currification **[CUR_t]**? Remarquons que ce principe, dans le cas n=2, est un théorème du système de Lambek puisque nous avons (voir précédemment) la déduction entre types:

$$\alpha_1 \text{ x } \alpha_2 \text{->} \beta \text{ <=> } \alpha_1 \text{->} (\alpha_2 \text{->} \beta).$$

Lorsque l'on considère un type de base comme une sorte d'ensemble, les types fonctionnels représentent des ensembles de fonctions entre ensembles. Dans le cas de la Catégorie des ensembles ***Ens***, A_1, A_2 et B étant des ensembles et *Ens*(X,Y)

désignant l'ensemble de toutes les fonctions de X dans Y, nous avons la bijection suivante:

$$\mathbf{Ens}(A_1 \times A_2, B) = \mathbf{Ens}(A_1, \mathbf{Ens}(A_2, B))$$

qui à toute fonction f_2 du produit cartésien d'ensembles $A_1 \times A_2$ dans B associe canoniquement la fonction f_1' de A_1 dans l'ensemble **Ens**(A_2, B) des fonctions de A_2 dans B, c'est-à-dire que l'on a, dans l'ensemble B, l'égalité suivante:

$f_2(a_1,a_2) = (f'_1(a_1))(a_2)$ pour chaque couple $<a_1,a_2>$ de $A_1 \times A_2$.

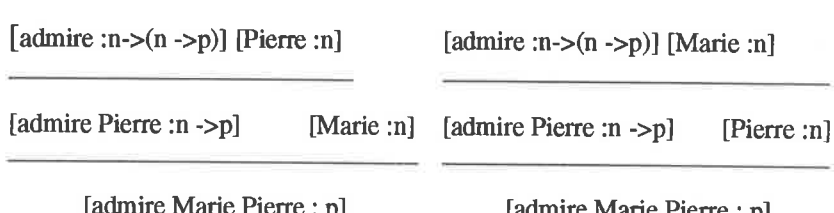
Cette propriété des ensembles se généralise aux produits finis cartésiens d'ensembles, elle est caractéristique des ensembles et plus généralement de toutes les Catégories cartésiennes fermées.

Du point de vue linguistique, l'opération de currification est importante puisqu'elle permet de ne considérer que des opérateurs unaires de type α->β et des opérateurs zéro-aires considérés comme des opérandes absolues. Ainsi, en admettant la théorie de la valence, des verbes bivalent ou trivalent sont considérés comme des opérateurs unaires qui à partir d'un opérande construisent des opérateurs unaires ou binaires. Le type d'un opérateur donne donc toutes les informations sur son arité. En associant des types fonctionnels aux unités linguistiques à partir des catégories d'une Grammaire Catégorielle classique, nous pouvons considérer que les trois arbres applicatifs (A_x) (A_1) et (A_2) suivants construisent une expression applicative de type p:

[admire :n x n ->p] [<Pierre, Marie> :n xn]

[admire (Pierre, Marie): p]

Arbre applicatif (A_x)



Arbre applicatif (A_1) **Arbre applicatif (A_2)**

Les résultats, qui sont de même type p, sont équivalents, c'est-à-dire:

admire (Pierre, Marie) = (admire Marie) Pierre.

Cependant, ces résultats ne sont pas construits de la même façon, ils n'ont pas la même histoire constitutive. Dans l'arbre applicatif (A_x), le prédicat verbal bivalent est appliqué globalement au couple des arguments <Pierre, Marie>, comme dans le calcul des prédicats classique. Dans l'histoire constitutive de l'arbre (A_1), il apparaît que le prédicat verbal, considéré comme un opérateur binaire, est appliqué dans un premier temps au terme qui fonctionne syntaxiquement comme «sujet», puis le résultat, considéré comme un opérateur unaire, est appliqué à son tour au terme qui fonctionne comme un «complément d'objet». Dans la constitution de l'arbre (A_2), la construction est effectuée en appliquant dans un premier temps le prédicat verbal au terme «objet» puis le résultat au terme «sujet», qui est ainsi caractérisé comme étant le dernier terme qui entre dans une relation prédicative, cette propriété servant à caractériser le terme «sujet». En explicitant l'histoire constitutive des résultats construits et en utilisant la λ-notation, nous avons:

$(\lambda<u_1,u_2>.\text{admire}_2\ (u_1,u_2))\ (<\text{Pierre, Marie}>) =$
admire (Pierre, Marie);

$((\lambda x.(\lambda y.\ \text{admire'}_1\ (x,y)))\ (\text{Pierre}))\ (\text{Marie}) =$
admire Marie Pierre;

$((\lambda y.(\lambda x.\ \text{admire'}_1\ (x,y)))\ (\text{Marie}))\ (\text{Pierre}) =$
admire Marie Pierre.

Divers arguments linguistiques montreraient que la construction (A_2) est celle qui doit être retenue dans une théorie linguistique des opérations de prédication. En effet, entre autres arguments, on pourrait montrer qu'il y a une relation «plus serrée» entre le prédicat verbal et le terme «objet» qu'entre le prédicat verbal et le terme «sujet»: la construction (A_2) reflète parfaitement cette hiérarchisation des arguments qui entrent successivement dans la construction applicative d'une relation prédicative, hiérarchisation qui n'est absolument pas exprimée par le formalisme du calcul des prédicats.

4.2. Analogie de Curry-Howard entre types et propositions

H. B. Curry (1958) a remarqué que les types engendrés par des types de base à l'aide du constructeur des types fonctionnels étaient en bijection avec les propositions de la logique positive des propositions dont le seul connecteur est l'opérateur d'implication, désigné ici par '->': toute formule du Calcul sur les types qui est un théorème peut ainsi être traduite immédiatement par un théorème du Calcul positif des propositions et réciproquement. Curry a également remarqué que tous les types des combinateurs de la logique combinatoire, dont nous reparlerons au paragraphe suivant, sont des formules qui sont analogues à des théorèmes du calcul propositionnel intuitionniste. En effet, il suffit d'interpréter le constructeur de fonctionnalité par le connecteur d'implication et d'interpréter les types comme des propositions. Ainsi, les formules comme:

(B->C)->((A->B)->(A->C))
(A->(B->C)) -> ((A->B)->(A->C))

s'interprètent aussi bien comme des types que comme des propositions; lorsque ce sont des types, la flèche '->' est le constructeur de fonctionnalité entre types; lorsque ce sont des propositions, la flèche '->' signifie l'implication entre propositions. Ces formules sont des théorèmes du Calcul propositionnel intuitionnisme et des théorèmes du Calcul sur les types. Dans une présentation par déduction naturelle dans le style de

Gentzen, on peut en donner des démonstrations en se servant des seules règles d'élimination et d'introduction de la flèche ->:

$$[e\text{->}] \quad \frac{\alpha\text{->}\beta \qquad \alpha}{\beta} \qquad\qquad [i\text{->}] \quad \frac{\begin{array}{c}[\alpha]\\ .\\ .\\ \beta\end{array}}{\alpha\text{->}\beta}$$

La preuve que (B->C)->((A->B)->(A->C)) est un théorème est donnée comme suit en «déduction naturelle»:

1.	B->C	hyp.
2.	A->B	hyp.
3.	A	hyp.
4.	A->B	rappel 2.
5.	B	[e->], 3., 4.
6.	B->C	rappel 1.
7.	C	[e->], 5., 6.
8.	A->C	[i->], 3.-7.
9.	(A->B)->(A->C	[i->], 2.-8
10.	(B->C)->((A->B)->(A->C))	[i->], 1.-9.

La preuve que (A->(B->C)) -> ((A->B)->(A->C)) est un théorème est donnée ci-dessous en «déduction naturelle»:

1.	A->(B->C)	hyp.
2.	A->B	hyp.
3.	A	hyp.
4.	A->(B->C)	rappel 1.
5.	B->C	[e->], 3., 4.
6.	A->B	rappel 2.
7.	B	[e->], 3., 6.
8.	C	[e->], 5., 7.
9.	A->C	[i->], 3.-8.
10.	(A->B)->(A->C)	[i->], 2.-9.
11.	(A->(B->C))->((A->B)->(A->C))	[i->], 1.-10.

A titre d'exercice, montrons que le type syntaxique:

((P/(P\N))/NC) x NC x ((P\N)/N) x (N/NC) x NC

(ou son analogue propositionnel) est bien une expression de type P. Pour simplifier les notations, remplaçons les symboles qui représentent des catégories élémentaires P, N et NC par les symboles abstraits respectifs A, B et C interprétés comme des types ou comme des propositions. Substituons la flèche '->' aux constructeurs de division / ou \ et interprétons le produit 'x' soit par le produit cartésien des types, soit par la conjonction de propositions. Le produit 'x', tout comme la conjonction propositionnelle, est gouvernée par les deux règles d'élimination (associées aux projections) et d'introduction suivantes:

$$[e_1\text{-}\mathbf{x}]\ \frac{A \times B}{A} \qquad [e_2\text{-}\mathbf{x}]\ \frac{A \times B}{B} \qquad [i\text{-}\mathbf{x}]\ \frac{A,\ B}{A \times B}$$

Après ces changements, le type syntaxique précédent devient:

(C->((B->A)->A) x C x (B->(B->A)) x (C->B) x C

Donnons la preuve que cette formule (type ou proposition) est une expression bien formée de type A.

1.	(C->((B->A)->A) x C x (B->(B->A)) x (C->B) x C	hyp.
2.	(C->((B->A)->A) x C	[e-x], 1.
3.	(B->(B->A)) x (C->B) x C	[e-x], 1.
4.	(C->((B->A)->A) x C	rappel 2.
5.	(C->((B->A)->A)	[e-x], 4.
6.	C	[e-x], 4.
7.	((B->A)->A)	[e->], 5., 6.
8.	(B->(B->A)) x (C->B) x C	rappel 3.
9.	(B->(B->A))	[e-x], 8.
10.	(C->B)	[e-x], 8.
11.	C	[e-x], 8.
12.	B	[e->], 10., 11.
13.	B->A	[e->], 9., 12.
14.	A	[e->], 5., 13.

On en déduit que:

(C->((B->A)->A) x C x (B->(B->A)) x (C->B) x C => A

est bien un théorème puisque A => A est un schéma d'axiome (dans le Calcul sur les types ou, par isomorphisme, dans le Calcul propositionnel).

Le Calcul sur les types et le Calcul positif sur les propositions étant isomorphes, toute démonstration de théorèmes du Calcul des propositions se transporte immédiatement dans le Calcul sur les types. En particulier, comme l'a montré J. Lambek, le théorème de Gentzen d'élimination de la coupure (Cut), qui donne une méthode effective pour démontrer des théorèmes dans le Calcul des propositions, est applicable au Calcul sur les types.

4.3. Interprétation fonctionnelle guidée par la syntaxe

Les preuves de théorèmes donnent la possibilité de construire directement une interprétation sémantique fonctionnelle dans le λ-calcul. Aux règles d'élimination [e->] et d'introduction [i->] du symbole '->' du Calcul des types sont canoniquement associées des règles de construction des expressions applicatives typées. Plus précisément, nous avons les deux règles de construction par application [e'->] et par abstraction [i'->] suivantes:

$$[e'\text{->}]\ \frac{X : \alpha\text{->}\beta \qquad Y : \alpha}{XY : \beta} \qquad\qquad [i'\text{->}]\ \frac{\begin{array}{c}[X : \alpha] \\ . \\ . \\ Y : \beta\end{array}}{\lambda X.Y : \alpha\text{->}\beta}$$

La première règle exprime que si un opérateur X de type α->β est appliqué à un opérande Y de type α alors le résultat XY est de type β. La seconde règle exprime que, avec l'hypothèse que X soit de type α, de l'expression Y de type β (dans laquelle apparaît une ou plusieurs occurrences de X), on peut abstraire une λ-expression λX.Y, c'est-à-dire un opérateur, de type α->β.

Reprenons les exemples précédents. Les preuves qui ont été données permettent de construire des expressions applicatives (des λ-expressions) typées. Nous avons ainsi:

1.	X :B->C	hyp.
2.	Y :A->B	hyp.
3.	Z :A	hyp.
4.	Y :A->B	rappel 2.
5.	YZ :B	[e'->], 3., 4.
6.	X :B->C	rappel 1.
7.	X(YZ) :C	[e'->], 5., 6.
8.	λZ.X(YZ) :A->C	[i'->], 3.-7.
9.	λY.λZ.X(YZ) :(A->B)->(A->C)	[i'->], 2.-8.
10.	λX.λY.λZ.X(YZ) :(B->C)->((A->B)->(A->C))	[i'->], 1.-9.

Nous venons de construire l'expression applicative (la λ-expression) λX.λY.λz.X(YZ) de type:

(B->C)->((A->B)->(A->C))

avec les variables abstraites X, Y et z de types respectifs: B->C, A->B, A. Cette expression applicative code directement la preuve que le type (B->C)->((A->B)->(A->C)) est un théorème.

De la même façon nous construisons l'expression applicative λX.λY.λZ.XZ(YZ) à partir de la preuve que son type ((A->(B->C))->((A->B)->(A->C)) est un théorème.

1.	X :A->(B->C)	hyp.
2.	Y :A->B	hyp.
3.	Z :A	hyp.
4.	X :A->(B->C)	rappel 1.
5.	XZ :B->C	[e'->], 3., 4.
6.	Y :A->B	rappel 2.
7.	YZ :B	[e'->], 3., 6.
8.	XZ(YZ) :C	[e'->], 5., 7.
9.	λZ.XZ(YZ) :A->C :	[i'->], 3.-8.
10.	λY.λZ.XZ(YZ) :(A->B)->(A->C)	[i'->], 2.-9.
11.	λXλY.λZ.XZ(YZ) :((A->(B->C))->((A->B)->(A->C))	[i'->],1.-10.

Toujours à titre d'exercice, reprenons la preuve de:

(C->((B->A)->A) x C x (B->(B->A)) x (C->B) x C => A

Supposons que nous ayons l'assignation suivante des types à des expressions applicatives X, Y, Z, T et U:

[X :(C->((B->A)->A)]; [Y :C]; [Z: (B->(B->A))]; [T :(C->B)]; [U :C]

Nous en déduisons la construction de l'expression applicative '((XY) (Z(TU)))' à partir de la preuve que son type '(C->((B->A)->A) x C x (B->(B->A)) x (C->B) x C' est réductible à A.

1.	[X :(C->((B->A)->A)] [Y :C]x [Z: (B->(B->A))] x [T :(C->B)]x [U :C]	hyp.
2.	[X :(C->((B->A)->A)] x [Y :C]	[e-x], 1.
3.	[Z: (B->(B->A))] x [T :(C->B)] x [U :C]	[e-x], 1.
4.	[X :(C->((B->A)->A)] x [Y :C]	rappel 2.
5.	[X :(C->((B->A)->A)]	[e-x], 4.
6.	[Y :C]	[e-x], 4.
7.	[XY :(B->A)->A]	[e'->], 5., 6.
8.	[Z: (B->(B->A))] x [T :(C->B)] x [U :C]	rappel 3.
9.	[Z: (B->(B->A))]	[e-x], 8.
10.	[T :(C->B)]	[e-x], 8.
11.	[U :C]	[e-x], 8.
12.	[TU :B]	[e'->], 10., 11.
13.	[Z(TU) : B->A]	[e'->], 9., 12.
14.	[(XY) (Z(TU))) : A]	[e'->], 5., 13.

En revenant aux substitutions [A := P], [B := N] et [C := NC] et en considérant que les expressions applicatives X, Y Z, T et U sont respectivement tout', garçon', aime', le', football', la preuve sur les types nous permet de construire directement l'expression applicative:

((tout' garçon') (aime' (le' football')))

qui est sous-jacente à la phrase: *tout garçon aime le football*. Le type '(NC->((N->P)->P) x NC x (N-> (N->P)) x (NC->N) x P' est une *spécification syntaxique* de cette phrase. La spécification syntaxique est insuffisante pour construire l'expression applicative sous-jacente, c'est-à-dire l'interprétation fonctionnelle de la phrase: il faut posséder une preuve que la spécification syntaxique est cohérente et réductible au type P des phrases. A par-

tir de cette preuve, on peut construire la représentation applicative sous-jacente à l'énoncé.

4.4. Théorie des types de Martin-Löf

L'analogie de Curry-Howard justifie les bases de l'approche intuitionniste de Martin-Löf qui identifie les types et les propositions: pour Martin-Löf, une proposition n'est pas le nom d'une «valeur de vérité» mais une proposition est un type dont les instances sont les preuves de cette proposition. Dans cette acceptation, une proposition A est vraie si et seulement si on peut exhiber une preuve de cette proposition A, autrement dit lorsque l'ensemble des preuves de la proposition A n'est pas vide. L'analogie entre types et propositions permet de donner également une signification fonctionnelle à l'implication propositionnelle. En effet, dans l'approche constructiviste de Martin-Löf, l'implication entre propositions, que nous avons désignée par: 'A->B', s'interprète comme suit: la signification de la proposition 'A->B' est donnée par une procédure fonctionnelle qui doit construire une preuve de la proposition B à partir d'une preuve de la proposition A. La théorie de Martin-Löf développe donc l'analogie de Curry-Howard en l'étendant aux autres connecteurs logiques ('&' et 'V'), à la négation (interprétée, comme dans l'approche intuitionniste de Heyting par l'absurdité) et aux quantificateurs d'existence et d'universalité. *Les types sont des spécifications de programmes.* A un type donné peut correspondre aucun programme (lorsque la spécification n'est pas correcte) ou un ou plusieurs programmes qui sont «équivalents en extension», puisqu'ils construisent le résultat ce qui est spécifié par le type, mais qui diffèrent «en intension» puisqu'ils construisent le résultat par des chemins différents. Les preuves que ces types sont des théorèmes du calcul inférentiel sur les types sont des instances de ces types; ces instances sont alors représentables par des λ-expressions, construites directement à partir de ces preuves; ces λ-expressions codent donc les preuves associées à un type, elles décrivent également les programmes dont le type est une spécification.

5. Composition des unités linguistiques

Les unités linguistiques, étant des opérateurs, sont composables entre elles. Les unités linguistiques étant, au moins pour certaines d'entre elles, des opérateurs typés, elles sont éventuellement composables entre elles, tout comme deux fonctions X et Y sont composables entre elles lorsque l'image de l'une est dans le domaine de définition de l'autre. Les combinateurs de la logique combinatoire étant des opérateurs abstraits, dont l'action intrinsèque est indépendante de tout domaine d'*interprétation*, ils servent à déterminer des modes de composition qui généralisent la composition entre fonctions. Un combinateur permet ainsi de construire des *opérateurs complexes* à partir d'opérateurs plus élémentaires. Chaque combinateur peut être représenté par une λ-expression close (c'est-à-dire sans variables libres) du λ-calcul de Church. Chaque combinateur peut aussi être inséré directement dans le système applicatif de la logique combinatoire de Curry au moyen de règles qui spécifient exactement son action sur les opérandes sur lesquelles il porte. Comme toutes les constantes logiques (flèche ou produit cartésien), la signification d'un combinateur est précisée par des règles d'élimination et d'introduction. Donnons quelques exemples de combinateurs.

5.1. Combinateurs élémentaires

Notation: Dans ce qui suit, les symboles X, Y, Z, ..., x, y, z, ... désignent des expressions applicatives quelconques qui représentent des opérateurs. Comme un combinateur construit un opérateur complexe à partir d'autres opérateurs, les symboles X, Y, Z désignent les opérateurs qui sont constitutifs de l'opérateur complexe construit par un combinateur et les symboles x, y, z, ... désignent les opérandes de l'opérateur complexe.

Chaque combinateur est défini soit par une λ–expression close soit par une règle d'élimination et une règle d'introduction. Nous donnons pour les combinateurs élémentaires les

λ-expressions associées et les règles d'élimination et d'introduction.

Combinateur I. Le plus simple des combinateurs est le combinateur d'identité **I**.

$$\mathbf{I} =_{def} \lambda X.\lambda x.Xx$$

$$[\text{e-}\mathbf{I}]\ \frac{(\mathbf{I}X)x}{Xx} \qquad [\text{i-}\mathbf{I}]\ \frac{Xx}{(\mathbf{I}X)x}$$

L'opérateur **I**X est construit à partir de l'opérateur X; l'action de **I**X sur un opérande x est la même que celle de X.

Combinateur B. Le combinateur **B** exprime un mode de composition qui s'interprète comme la simple composition de fonctions lorsque les opérateurs sont interprétés par des fonctions ensemblistes.

$$\mathbf{B} =_{def} \lambda X.\lambda Y.\lambda x.f(gx).$$

$$[\text{e-}\mathbf{B}]\ \frac{(\mathbf{B}XY)x}{X(Yx)} \qquad [\text{i-}\mathbf{B}]\ \frac{X(Yx)}{(\mathbf{B}XY)x}$$

L'opérateur **B**XY est un opérateur complexe obtenu en composant les deux opérateurs X et Y entre eux. L'action de cet opérateur complexe **B**XY sur un opérande x est précisée par les règles d'élimination et d'introduction. Lorsque les opérateurs X et Y sont interprétés comme des fonctions ensemblistes, le combinateur **B** représente exactement la composition des fonctions, d'où: $X \circ Y = \mathbf{B}XY$.

Combinateur S. Le combinateur **S** est une généralisation de **B**, il compose deux opérateurs X et Y, d'où l'opérateur complexe **S**XY:

$\mathbf{S} =_{def} \lambda X.\lambda Y.\lambda x.\ Xx(Yx)$

$$[\text{e-}\mathbf{S}]\ \frac{(\mathbf{S}XY)x}{Xx(Yx)} \qquad [\text{i-}\mathbf{S}]\ \frac{Xx(Yx)}{(\mathbf{S}XY)x}$$

L'opérateur complexe **SXY**, lorsqu'il a pour opérande x, donne le même résultat que l'action de X sur l'argument x et sur le résultat de l'opérateur Y sur x.

Combinateur W. Le combinateur de diagonalisation **W** duplique l'argument d'un opérateur binaire X.

$\mathbf{W} =_{def} \lambda X.\lambda x.Xxx$

$$[\text{e-}\mathbf{W}]\ \frac{(\mathbf{W}X)x}{Xxx} \qquad [\text{i-}\mathbf{W}]\ \frac{Xxx}{(\mathbf{W}X)x}$$

L'opérateur **WX** est un opérateur complexe construit à partir de l'opérateur X; l'action de **W** consiste à dupliquer un même argument x de X (processus de diagonalisation).

Combinateur C. Le combinateur **C** de conversion est défini comme suit:

$\mathbf{C} =_{def} \lambda X.\lambda x.\lambda y.\ Xyx$

$$[\text{e-}\mathbf{C}]\ \frac{(\mathbf{C}X)xy}{Xyx} \qquad [\text{i-}\mathbf{C}]\ \frac{Xyx}{(\mathbf{C}X)xy}$$

L'opérateur (**CX**) est un opérateur complexe (le converse de l'opérateur X); l'action de **CX** est précisée par les règles: si X opère sur les opérandes x puis y dans cet ordre applicatif, alors l'opérateur (**CX**) opère sur les opérandes y puis x dans cet ordre applicatif, autrement dit (**CX**) opère sur les arguments permutés de X.

Combinateur C*. Le combinateur **C*** (dit de "lifting") est défini comme suit:

$$\mathbf{C}* =_{def} \lambda Y.\lambda X.YX$$

$$[\text{e-}\mathbf{C}*]\ \frac{(\mathbf{C}*X)Y}{YX} \qquad [\text{i-}\mathbf{C}*]\ \frac{YX}{(\mathbf{C}*X)Y}$$

L'opérateur (**C***X) est un opérateur complexe construit à partir de X; son rôle opératoire est tel que si X était opérande par rapport à l'opérateur Y alors l'opérateur Y devient opérande de l'opérateur (**C***X).

Combinateur K. Le combinateur **K** (création d'argument fictif) est défini par:

$$\mathbf{K} =_{def} \lambda X.\lambda x.X$$

$$[\text{e-}\mathbf{K}]\ \frac{(\mathbf{K}X)x}{X} \qquad [\text{i-}\mathbf{K}]\ \frac{X}{(\mathbf{K}X)x}$$

Combinateur Φ. Le combinateur Φ (intrication d'opérateurs agissant en parallèle) est défini par:

$$\Phi =_{def} \lambda X.\lambda Y.\lambda Z.\lambda u.\ X(Yu)(Zu)$$

$$[\text{e-}\Phi]\ \frac{\Phi XYZu}{X(Yu)(Zu)} \qquad [\text{i-}\Phi]\ \frac{X(Yu)(Zu)}{\Phi XYZu}$$

Chaque combinateur introduit une relation *β-réduction élémentaire* (et sa converse une *β-expansion*) entre l'expression du programme de combinaison spécifié par le combinateur et le résultat de l'exécution du programme, autrement dit entre l'expression *avec* combinateur et le résultat de l'action du combinateur *sans* ce combinateur. Nous avons par exemple, les β-réduc-

tions (β-expansions) élémentaires suivantes, associées aux combinateurs élémentaires précédents:

IX	β=> X	X	<=β	**I**X
BXYz	β=> X(Yz)	X(Yz)	<=β	**B**XYz
WXy	β=> Xyy	Xyy	<=β	**W**Xy
CXyz	β=> Xzy	Xzy	<=β	**C**Xyz
C*XY	β=> YX	YX	<=β	**C***XY
SXYz	β=> Xz(Yz)	Xz(Yz)	<=β	**S**XYz
ΦXYZu	β=> X(Yu)(Zu)	X(Yu)(Zu)	<=β	**Φ**XYZu

La fermeture réflexive et transitive de la relation engendrée par les β-réductions (respectivement β-expansions) élémentaires constitue la relation de β-réduction (respectivement de β-expansion). Cette relation de β-réduction (et de β-expansion) structure l'ensemble des expressions applicatives avec combinateurs.

5.2. Propriétés des combinateurs

Les combinateurs ne sont pas tous indépendants. Par exemple, on a: [**I** = **BCC**]. Cette identité entre combinateurs signifie que «le converse du converse est la même chose que l'identité». En effet:

1.	**BCC**Xyx	hyp.
2.	**C**(**C**X)yx	[e-**B**], 1.
3.	**C**Xxy	[e-**C**], 2.
4.	Xyx	[e-**C**], 3.
5.	**I**Xyx	[i-**I**], 4.

Il existe une infinité dénombrable de combinateurs puisqu'il est possible de les faire agir les uns sur les autres par l'application et donc de construire des combinateurs dérivés. Par

exemple, on peut définir par récurrence les combinateurs $\mathbf{B^2}$, plus généralement $\mathbf{B^n}$, comme suit [$\mathbf{B^2} =_{def} \mathbf{BBB}$], [$\mathbf{B^n} =_{def} \mathbf{BBB^{(n-1)}}$]. On démontre que les actions de ces combinateurs sont définies par les réductions:

$\mathbf{B}XYZ_1 \quad => X(YZ_1)$
$\mathbf{B^2}XYZ_1Z_2 \quad => X(YZ_1Z_2)$
$\mathbf{B^n}XYZ_1Z_2 \ldots Z_n \quad => X(YZ_1Z_2 \ldots Z_n)$

On démontre qu'il est possible de définir tous les combinateurs à partir de combinateurs de base. On peut prendre pour combinateurs de base les combinateurs **S** et **K**, les autres combinateurs étant tous définissables en termes de **S** et **K**. Ainsi, nous avons la définition de I: [$\mathbf{I} =_{def} \mathbf{SKK}$]. En effet:

1. **SKK**x — hyp.
2. **K**x(**K**x) — [e-**S**], 1.
3. x — [e-**K**], 2.
4. **I**x — [i-**I**], 3.

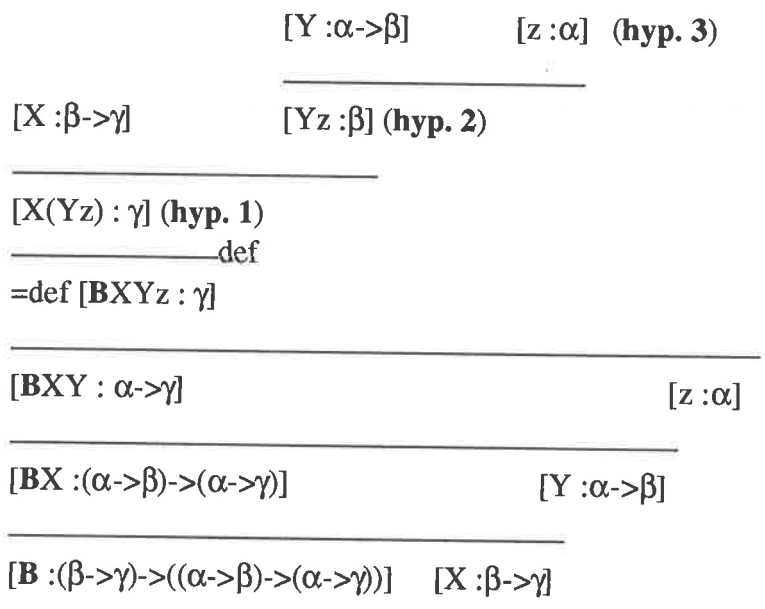
L'utilisation des combinateurs permet de *définir des propriétés intrinsèques d'opérateurs particuliers* sans faire appel à des variables. Ainsi, lorsque X est un opérateur binaire, exprimer qu'il est *intrinsèquement commutatif*, c'est affirmer que les actions de X et de son converse **C**X conduisent à des *résultats qui ont même signification*, en posant, *sans faire appel à des variables*, l'égalité intrinsèque relative à X: [$\mathbf{C}X =_{def} X$]. Donnons, à titre d'exemple, les formulations intrinsèques de la commutativité et de l'associativité de l'opérateur binaire, noté '+', défini aussi bien sur le domaine des entiers naturels que sur celui des nombres réels, en laissant au lecteur le soin d'en vérifier l'exactitude:

[**C**+ = +]
[**BC**(**BC**(**W**(**BB^2C**))+ = **WB^2**+]

Les combinateurs étant des opérateurs, ils sont typés non par des types constants mais par des *schémas de type*. Dans un contexte donné, le combinateur acquiert un type déterminé qui dépend directement du type assigné aux opérandes.

5.3. Schémas de type des combinateurs

Calculons par exemple le schéma de type du combinateur **B**. Avec les trois hypothèses: [X(Yzx) : γ], [Yz :β] et [z :α], dont on peut s'abstraire, nous avons les deux arbres suivants:



d'où le schéma de type assigné au combinateur **B**:

[**B** :(β->γ)->((α->β)->(α->γ))]

L'algorithme consiste à analyser la construction de l'expression qui définit l'action du combinateur, à savoir X(Yz), en décomposant, du sommet vers les feuilles, cette expression en faisant apparaître l'opérateur et l'opérande (arbre du haut). On assigne un type (en utilisant la règle [$\mathbf{App_t}$]) à partir de types assignés par hypothèse à certaines expressions. On analyse ensuite la construction du programme de construction défini par le combinateur, à savoir **BXY**, en assignant un type par abstraction, ce qui revient à se libérer progressivement des hypothèses introduites (arbre du bas).

Par la même méthode, calculons le schéma de type du combinateur **S**.

$[X : \alpha \to (\beta \to \gamma)]$ $[z : \alpha]$ (**hyp. 3**) $[Y : \alpha \to \beta]$ $[z : \alpha]$ (**hyp. 3**)

$[Xz : \beta \to \gamma]$ $[Yz : \beta]$ (**hyp. 2**)

$[Xz(Yx) : \gamma]$ (**hyp. 1**)

——— def

$[\mathbf{S}XYz : \gamma]$

$[\mathbf{S}XY : \alpha \to \gamma]$ $[z : \alpha]$

$[\mathbf{S}X : (\alpha \to \beta) \to (\alpha \to \gamma)]$ $[Y : \alpha \to \beta]$

$[\mathbf{S} : (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))]$ $[X : \alpha \to (\beta \to \gamma)]$

Le schéma de type du combinateur **S** est donc:

$$[\mathbf{S} : (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))]$$

D'après l'isomorphisme de currification $[\mathbf{CUR}_t]$, nous avons l'équivalence entre les schémas de type pour respectivement le combinateur **B** et pour **S**:

$$((\beta \to \gamma) \times (\alpha \to \beta)) \to (\alpha \to \gamma) = (\beta \to \gamma) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$$

$$((\alpha \times \beta \to \gamma) \times (\alpha \to \beta)) \to (\alpha \to \gamma)) = (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$$

La lecture de ces schémas de type et des équivalences donne des informations sur la syntaxe des combinateur **B** et **S**. Le combinateur **B** est un opérateur binaire qui compose entre eux deux opérateurs de types respectifs $\beta \to \gamma$ et $\alpha \to \beta$ de façon à construire un opérateur complexe de type $\alpha \to \gamma$. Le combinateur **S** est un opérateur qui compose entre eux deux opérateurs, le premier est

binaire α x β->γ, le second est unaire de type α->β; le résultat est un opérateur complexe de type α->γ.

Lorsqu'on interprète les types par des propositions, on remarque que les schémas de type de **B** et de **S** correspondent aux deux théorèmes, dont on a donné plus haut une démonstration, du Calcul propositionnel:

(B->C)->((A->B)->(A->C))
(A->(B->C))->((A->B)->(A->C))

On démontre que tout schéma de type assigné à un combinateur de la logique combinatoire est l'analogue d'un théorème du Calcul propositionnel.

5.4. Grammaire Catégorielle Combinatoire Applicative (GCCA)

Les combinateurs nous permettent de construire l'interprétation fonctionnelle d'un énoncé sous forme d'une représentation applicative normalisée sous-jacente à cet énoncé. En effet, les unités linguistiques étant des opérateurs, ils se voient assignés des types fonctionnels qui sont éventuellement composables entre eux puisque certains opérateurs, comme les fonctions, sont composables. Une extension des Grammaires Catégorielles classiques (avec le Calcul de Lambek notamment) a consisté à introduire des transformations sur les types (par exemple, l'opération de «type raising» (comme: α => $\beta/(\beta\backslash\alpha)$) et des compositions entre types fonctionnels composables (comme: (γ/β) x (β/α) => γ/α). Ces compositions entre les types d'unités linguistiques et les transformations de types sont justifiées lorsqu'on remarque que ces unités typées sont composées ou transformées *en tant qu'opérateurs*. La Grammaire Catégorielle Combinatoire et Applicative (GCCA) étend les Grammaires Catégorielles classiques en: 1°) exploitant systématiquement cette remarque ce qui revient à effectuer parallèlement des opérations sur les types qui guident les opérations sur les unités linguistiques; 2°) en donnant une stratégie heuristique, sous forme de métarègles, qui permettent de trouver une preuve de réduction du

type syntaxique au type des phrases. En effet, nous avons vu que dès que l'on avait obtenu une preuve de ce que le type de la suite, avec types assignés, était réductible au type des phrases (vérification de la correction syntaxique ou de la cohérence de la spécification syntaxique), on pouvait en déduire immédiatement la construction de l'expression applicative associée à la suite analysée. Cependant, l'obtention de la preuve n'est pas immédiate: quelle doit être la procédure que l'on doit suivre pour trouver cette preuve? Les métarègles de la GCCA (thèse de I. Biskri) donnent des éléments de réponse à cette question. Par ailleurs, lorsqu'on procède à l'analyse d'un énoncé, on doit passer progressivement d'une *expression concaténée typée*, c'est-à-dire à la présentation «en surface» de l'énoncé, à une *expression applicative typée*, c'est-à-dire l'interprétation fonctionnelle de l'énoncé. Pour cela, on introduit «localement» dans un premier temps des combinateurs qui correspondent à des compositions de types ou à des transformations de types nécessaires pour que l'analyse syntaxique puisse conclure. Ensuite, dans un second temps, la «normalisation» de l'expression applicative ainsi construite conduit, par une succession d'élimination des combinateurs introduits «localement», à construire, par des β-réductions, une représentation applicative normalisée, appelée forme normale de l'énoncé analysé. On sait que, d'après le théorème de Church-Rosser, cette forme normale est *unique* et, d'après le théorème de normalisation, qu'elle *existe*: toute expression applicative bien typée a une forme normale.

Pour résumer, on peut dire que les règles d'inférence sur les types, les introductions «locales» de combinateurs et les β-réductions entre expressions applicatives constituent un «univers de calcul sur les expressions applicatives typées» dans lequel sont situées les preuves de réduction (réduction entre types; normalisation) que l'on cherche. Les métarègles sont utilisées pour trouver et construire, dans un certain nombre de cas, une preuve particulière de réduction sur les types, cette preuve est codée par une expression applicative que l'on normalisera ensuite pour y trouver la forme normale.

Pour illustrer notre propos, donnons un échantillon de règles extraites de la GCCA (voir la totalité des règles dans: Biskri,

Desclés, Biskri 1996). Aux règles de vérification syntaxique de la Grammaire Catégorielle Combinatoire (GCC) de M. Steedman, on associe canoniquement, dans la GCCA, des règles de construction sémantique qui introduisent «localement» des combinateurs.

règle de vérification syntaxique (GCC: Steedman)	**règle de construction sémantique** (GCCA: Desclés-Biskri)
$(\beta/\alpha)\ x\ \alpha$	$[X : (\beta/\alpha)]\ x\ [Y : \alpha]$
$\longrightarrow[>]$	$\longrightarrow[>]$
β	$[XY : \beta]$
$(\alpha/\beta)\ x\ (\beta/\gamma)$	$[X : (\alpha/\beta)]\ x\ [Y : (\beta/\gamma)]$
$\longrightarrow[\mathbf{B}]$	$\longrightarrow[\mathbf{B}]$
α/γ	$[\mathbf{B}XY : \alpha/\gamma]$
α	$[X : \alpha]$
$\longrightarrow[\mathbf{T}]$	$\longrightarrow[\mathbf{C*}]$
$\beta/(\beta\backslash\alpha)$	$[\mathbf{C*}X : \beta/(\beta\backslash\alpha)]$

Les prémisses dans les règles de construction syntaxique sont des conjonctions d'expressions typées en relation biunivoque avec une juxtaposition d'expressions linguistiques (donc non applicatives) mais les conclusions sont des expressions applicatives typées et «locales».

Nous allons présenter un exemple d'analyse d'un énoncé dans le cadre des GCCA. en reprenant l'analyse syntaxique et sémantique de l'énoncé:

Tout garçon aime le football

que l'on comparera à l'analyse du même énoncé effectuée dans un précédent paragraphe (voir § 4). En assignant des types syntaxiques aux unités linguistiques élémentaires de cet énoncé, comme nous l'avons déjà effectué, nous avons obtenu une expression concaténée (ec) sous forme d'une juxtaposition

d'unités linguistiques (ici des mots) avec types assignés, l'opération de juxtaposition étant désignée par 'x':

(ec) [tout :((P/(P\N))/NC)] x [garçon :NC] x [aime :((P\N)/N):] x [le :(N/NC)] x [football :NC]

Les types assignés dans (ec) sont les types constitutifs du type τ de (ec):

(τ) τ = ((P/(P\N))/NC) x NC x ((P\N)/N) x (N/NC) x NC

Le calcul inférentiel sur les seuls types syntaxiques, avec les règles précédentes (inférence (**I**)), conduit à vérifier que l'expression est syntaxiquement «bien formée» puisque de type P:

((P/(P\N))/NC) x NC x ((P\N)/N) x (N/NC) x NC
——————>[>]
P/(P\N)
——————————>[**B**]
P/N
——————————>[**B**]
P/NC
——————————>[>]
P

Ce calcul est la preuve que relation (R):

(R) ((P/(P\N))/NC) x NC x ((P\N)/N) x (N/NC) x NC => P

est un théorème du système inférentiel des types. Des métarègles (que nous ne donnons pas ici) ont pour but de déclencher des changements de types à certains endroits et de composer les types à d'autres endroits. Ces métarègles nous conduisent à construire, parmi toutes les preuves possibles, une preuve particulière en adoptant une stratégie «quasi-incrémentale», c'est-à-dire en procédant, autant que faire se peut, à une analyse de la suite textuelle «de gauche à droite». Comme on peut le voir, la preuve proposée est différente de celle qui a été donnée dans un précédent paragraphe, cette dernière procédant par un balayage entier de la suite textuelle avec retour en arrière (c'est-à-dire

avec «backtracking»). A cette *inférence sur les types*, les règles de construction sémantique de la GCCA associent canoniquement *un calcul sur les unités linguistiques typées*: à la règle d'application [>] correspond une simple application d'un opérateur à un opérande, à l'utilisation inférentielle de la règle [**B**] sur les types correspond une composition fonctionnelle des unités linguistiques, considérées comme des opérateurs, ce qui conduit à l'introduction «locale» du combinateur **B**: le combinateur note dans le calcul lui-même la composition fonctionnelle des unités linguistiques. Ce calcul, de même structure que le calcul inférentiel sur les types, est effectué à l'aide des règles de construction sémantique. Nous avons:

1. [tout :(P/(P\N))/NC] x [garçon :NC] x [aime :(P\N)/N):] x [le :N/NC] x [NC :football] — hyp.
2. [(tout garçon): P/(P\N)] x [aime :((P\N)/N)] x [le : (N/NC) : le] x [football : NC] — [>], 1.
3. [**B**(tout garçon) (aime)] x [le : N/NC] x [football : NC] — [**B**], 2.
4. [**B**(**B**(tout garçon) (aime)) (le) : N/NC] x [football : NC] — [**B**], 3.
5. [**B**(**B**(tout garçon) (aime)) (le) (football): P] — [>], 4.

Ce processus de calcul (**C**) construit *progressivement une expression applicative globale* (ea) de type P *à partir de l'expression concaténée typée* (ec) de type τ, en introduisant «localement» des combinateurs et en composant «localement» les unités par l'opération d'application:

(ea) **B**(**B**(tout garçon) (aime)) (le) (football)

Toute l'information pour construire l'expression (ea) est entièrement déterminée par: (i) l'assignation de types aux unités linguistiques élémentaires de (ec) et donc par le type τ, et par (ii) l'inférence (**I**) du type τ au type P. Comme l'expression typée (ec) encode à la fois les informations données par (i), la construction (**C**) de (ea) à partir de (ec) a été effectuée, au moyen des règles de la GCCA, en se laissant entièrement guider par l'inférence (**I**) sur les types.

L'information syntaxique sur l'énoncé est entièrement fournie par le type τ. *Ce type τ code les contraintes d'agencement syntaxique des unités linguistiques avec des types assignés*. Si le

type τ constitue une spécification syntaxique de l'expression applicative (ea), sa seule connaissance n'est pas suffisante pour fournir la méthode de construction de l'expression applicative (ea) associée. Dans le cadre de la GCC de Steedman, avec les seules règles syntaxiques comme [>], [**B**] et [**T**], la construction de l'expression applicative sous-jacente à l'énoncé doit être effectuée dans un second temps en se donnant des règles d'interprétation dans le λ-calcul et des principes d'unification. En revanche, les règles constructives de la GCCA nous permettent de vérifier, par inférence, d'une part que le type τ est «bien formé» et réductible au type P des phrases, autrement dit, de vérifier que les contraintes d'agencement syntaxique codées par τ sont cohérentes et, si tel est le cas, d'autre part d'en déduire ensuite la construction de l'expression applicative (ea), qui donne l'interprétation fonctionnelle de l'énoncé. La construction (**C**) est entièrement guidée par la structure de l'inférence (**I**). Il apparaît alors que l'expression applicative (ea) est une certaine instance du type P, instance dont la construction effective est entièrement déterminée par (i) *la spécification syntaxique* τ et (ii) *une certaine preuve que la relation inférentielle* (R) 'τ => P' est bien un théorème du système inférentiel sur les types. D'autres instances de type P avec une spécification syntaxique de type τ seraient construites de la même façon. Par exemple, les expressions applicatives:

(ea') **B**(**B**(toute fille) (aime)) (la) (cuisine)
(ea") **B**(**B**(tout surveillant) (respecte)) (la) (réglementation)

sont des instances du type P avec une spécification syntaxique de type τ: les constructions de ces expressions sont analogues, seules les unités linguistiques avec une même assignation des types constitutifs de τ, changent.

Alors que l'expression (ec) n'est pas une expression applicative mais une *expression concaténée* typée – de type τ –, l'expression (ea) est une *expression applicative* typée – du type P obtenue par inférence à partir de la spécification syntaxique fournie par le type τ. Or, l'analyse de la structure applicative (ea) montre que cette expression applicative exprime *directement son propre mode de construction*, ce qui n'était pas le cas

de l'expression concaténée (ec). Ainsi, *la seule connaissance de la structure applicative d'une expression applicative de type P indique comment cette expression est construite.* Une telle expression applicative *code donc biunivoquement sa propre construction.*

Maintenant, étant donnée une expression applicative de type P avec des types syntaxiques assignés aux unités linguistiques qui la composent, on peut chercher à vérifier que la relation inférentielle entre le type syntaxique τ, déterminé par les seuls types assignés aux unités élémentaires, et le type P, est bien un théorème du système inférentiel sur les types; pour cela, on procède aux inférences élémentaires associées à chacune des étapes de la construction de l'expression applicative. Pour l'expression applicative (ea) qui nous sert d'exemple conducteur, nous avons:

1.	[**B**(**B**(tout garçon) (aime)) (le) (football) :P]	hyp.
2.	[**B**(**B**(tout garçon) (aime)) (le) :P/NC] x [football :NC]	[>], 1.
3.	[**B**(tout garçon) (aime):P/N] x [le:N/NC] x [football:NC]	[**B**], 2.
4.	[(tout garçon):P/(P\N)] x [aime:(P\N)/N)] x [le:N/NC] x [football :NC]	[**B**], 3.
5.	[tout :(P/(P\N))/NC] x [garçon :NC] x [aime :(P\N)/N)] x [le:N/NC]x[football :NC]	[>], 4.

Nous en déduisons le théorème sur les types: P <= τ. L'expression applicative (ea) de type P peut donc bien être également considérée comme une instance de type τ. On remarquera, au passage, que les schémas de types des occurrences du combinateur **B** seraient instanciés en tenant compte des types des opérateurs sur lesquels il agit.

Revenons maintenant à l'expression applicative (ea):

(ea) **B**(**B**(tout garçon) (aime)) (le) (football)

Cette expression contient des combinateurs que l'on peut chercher à éliminer. Nous en déduisons la β-réduction (considérée comme un processus de normalisation) suivante entre expressions applicatives de même type:

Processus de normalisation:

1. [**B**(**B**(tout garçon) (aime)) (le) (football): P] hyp.
2. [(**B**(tout garçon) (aime)) (le football) : P] [e-**B**], 1.
3. [(tout garçon) (aime (le football)) : P] [e-**B**], 2.

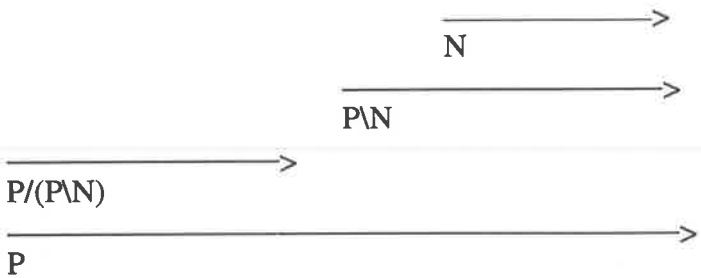
Le dernier pas est l'expression applicative:

(fn) (tout garçon) (aime (le football))

constitue de ce fait une «forme normale» (au sens technique de la logique combinatoire) de l'expression applicative (ea) car elle ne peut plus être réduite. Cette expression est aussi la forme normale sous-jacente à l'énoncé analysé *tout garçon aime le football.*

Remarquons que, pour les mêmes types assignés aux mêmes unités linguistiques élémentaires, la forme normale (fn) décrit un mode d'agencement applicatif différent de celui qui est exprimé par (ea); *cette forme normale code donc un autre mode de construction applicative* que celui qui est codé par (ea). Il s'ensuit que la forme normale (fn) correspond à une autre inférence entre les types; cette inférence est directement déduite de la construction applicative (fn). Nous avons:

((P/(P\N))/NC) x NC x ((P\N)/N) x (N/NC) x NC



On voit donc que les expressions applicatives (ea) et (fn) peuvent, toujours pour les mêmes assignations de types aux unités linguistiques élémentaires, être considérées comme deux instances différentes du même type τ. Elles expriment deux *programmes applicatifs distincts de même spécification syntaxique* τ. Cependant, ces deux programmes applicatifs ne sont pas

indépendants puisque le programme (ea) est réductible au programme (fn):

[**B**(**B**(tout garçon) (aime)) (le) (football): P]

β=> [(tout garçon) (aime (le football)): P].

Le programme (fn) exprime la construction applicative «normale» sous-jacente à l'énoncé (ec). Il a été obtenu au moyen d'un «processus de normalisation» portant sur (ea). Les deux expressions applicatives (ea) et (fn) représentent deux interprétations fonctionnelles du même énoncé *tout garçon aime le football.* Puisque ces deux interprétations sont intereliées par une β-réduction, elles expriment une *signification commune* dont un représentant canonique est justement la forme normale (fn). Ces programmes diffèrent uniquement par des modes de construction différents des mêmes unités linguistiques agencées. L'expression (ea) est un programme construit par une «stratégie incrémentale» à partir des informations contenues dans le type τ, c'est-à-dire en allant «de gauche à droite» et en composant le syntagme quantifié (*tout garçon*), considéré comme un opérateur, avec le verbe (*aime*) puis le résultat est de nouveau composé avec l'article (*le*), lui aussi opérateur, et l'expression ainsi obtenue est un opérateur qui est appliqué finalement à l'opérande nominal (*football*). Bien que l'expression (fn) soit de même type, elle exprime un programme de construction qui coïncide directement avec son interprétation fonctionnelle: dans un premier temps, un premier syntagme nominal (*le football*) est construit par application, puis ensuite le syntagme verbal (*aime* (*le football*)) est construit en appliquant le prédicat (*aime*) sur le syntagme nominal déjà construit (*le football*); le syntagme quantifié (*tout garçon*) est ensuite appliqué au syntagme verbal. Par conséquent, *ces deux programmes applicatifs* (ea) et (fn) construisent, par des moyens différents, *une même signification.* On dira que ces programmes constructifs diffèrent *en intension.* mais que leurs extensions sont équivalentes.

La construction sémantique de l'énoncé *tout garçon aime le football* est obtenue en «recollant» le processus de normalisation (pas 5 à 7) à la construction (pas 1 à 5) directement calquée sur l'inférence entre types: τ => P:

1. [tout :(P/(P\N))/NC] x [garçon :NC] x [aime :(P\N)/N):] x [le :N/NC] x [NC :football] hyp.
2. [(tout garçon): P/(P\N)] x [aime : ((P\N)/N)] x [le : (N/NC) : le] x [football : NC] [>], 1.
3. [**B**(tout garçon) (aime)] x [le : N/NC] x [football : NC] [**B**], 2.
4. [**B**(**B**(tout garçon) (aime)) (le) : N/NC] x [football : NC] [**B**], 3.
5. [**B**(**B**(tout garçon) (aime)) (le) (football): P] [>], 4.

==========

5. [**B**(**B**(tout garçon) (aime)) (le) (football): P] [>], 4.
6. [(**B**(tout garçon) (aime)) (le football) : P] [e-**B**], 5.
7. [(tout garçon) (aime (le football)): P] [e-**B**], 6.

6. Opérateurs illatifs

Nous allons passer à une représentation en «logique illative» (branche de la logique combinatoire obtenue en adjoignant aux combinateurs des opérateurs qui jouent le rôle inférentiel des quantificateurs et des connecteurs dans la logique classique du premier ordre). Introduisons les opérateurs de quantification universelle non restreinte et restreinte Π_1 et Π_2.

Soient deux expressions applicatives X et Y représentant deux opérateurs unaires, l'expression Π_1X (lire: «tout est X») et Π_2XY (lire: «tout X est Y» ou «tout ce qui est X est Y») sont deux expressions applicatives propositionnelles obtenues en appliquant respectivement le quantificateur Π_1 sur X ou le quantificateur Π_2 sur X, puis le résultat sur Y. L'opérateur **E** indiquant l'existence (dans un univers spécifié d'entités) d'opérandes de l'opérateur X sur lequel **E** porte (ce qui signifie que le domaine de définition de X n'est pas vide), les règles d'élimination des deux quantificateurs sont données comme suit:

$$[\text{e-}\Pi_1]\ \frac{\Pi_1 X,\ \mathbf{E}X}{Xx} \qquad [\text{e-}\Pi_2]\ \frac{\Pi_2 XY,\ Xx}{Yx}$$

La première règle signifie que si X est un prédicat (unaire) et si x est un élément (quelconque) qui existe dans le domaine

(univers de discours) de définition de X, ce que l'on représente par EX, alors le prédicat X s'applique à x. La seconde règle signifie que si l'on a la proposition Π_2XY et si l'opérateur X s'applique à x, alors l'opérateur Y s'applique également à x.

Désignons par 'p' le type particulier des expressions propositionnelles. Supposons que X et Y soient des opérateurs de type fonctionnel ι->p où ι est le type des entités individuelles. Ces opérateurs sont donc des constructeurs de propositions. Les types de Π_1 et de Π_2 sont définis comme suit

Π_1: (ι->p)->p

Π_2: (ι->p)->((ι->p)->p).

Le quantificateur Π_1 est un opérateur qui s'appliquant à un prédicat de type ι->p construit une proposition de type p. Le quantificateur Π_2 est un opérateur qui en s'appliquant à un prédicat de type ι->p construit un autre opérateur de type (ι->p)->p. Puisque, d'après le principe de currification, nous avons l'équivalence de types:

(ι->p) x (ι->p)->p = (ι->p)->((ι->p)->p)

nous pouvons dire que le quantificateur Π_2 s'applique à un couple de prédicats pour construire une proposition.

L'expression applicative obtenue précédemment à partir de l'énoncé *tout garçon aime le football*:

(tout garçon) (aime (le football))

est représentée maintenant par une autre expression applicative en utilisant le quantificateur restreint Π_2:

Π_2 (garçon) (aime (le football)).

Nous en déduisons l'inférence suivante entre expressions applicatives de même type P:

Π_2 (garçon) (aime (le football)), garçon (Julien)

---

aime (le football) (Julien)

En effet, nous avons:

1.	Π_2 (garçon) (aime (le football))	hyp.
2.	2.1. garçon (Julien)	hyp.
3.	2.2. Π_2 (garçon) (aime (le football))	rep. 1.
4.	2.3. (aime (le football)) (Julien)	[e-Π_2], 3., 2.

En français l'inférence:

> «si tout garçon aime le football et si Julien est un garçon, alors Julien aime le football»

exprime directement l'inférence précédente.

Etant donné un énoncé, dont la spécification syntaxique est exprimée par un type syntaxique τ. Nous avons un parallélisme étroit entre d'une part le choix d'une preuve syntaxique que le type τ se réduit au type P des phrases et d'autre part la construction de l'interprétation fonctionnelle qui est directement guidée par la preuve choisie choisies. A propos de l'exemple traité *tout garçon aime le football*, trouver une preuve de la relation entre types:

$$((P/(P\backslash N))/NC) \times NC \times ((P\backslash N)/N) \times (N/NC) \times NC \Rightarrow P$$

conduit directement à trouver une construction d'une expression applicative qui peut elle-même être réduite à sa forme normale. Cette forme normale rend possible le remplacement des quantificateurs linguistiques par des quantificateurs illatifs. Nous avons ainsi:

[tout: ((P/(P\N))/NC)]x[garçon :NC]x[aime :(P\N)/N]x [(N/NC) :le]x[NC :football]
-> [**B**(**B** (tout garçon) aime) (le football) : P]
β=> [(tout garçon) (aime (le football)) :P]
= [Π_2 (garçon) (aime (le football)) :P].

Nous avons des règles analogues aux règles relatives aux quantificateurs universels pour les quantificateurs existentiels Σ_1 et Σ_2. Soient X et Y deux prédicats unaires. Donnons les règles d'élimination:

$$[e\text{-}\Sigma_1]\ \frac{\Sigma_1 X,\ Xx \Rightarrow Z}{Z} \qquad [e\text{-}\Sigma_2]\ \frac{\Sigma_2 XY,\ Xx \Rightarrow Z,\ Yx \Rightarrow Z}{Z}$$

où 'x' est un élément non spécifié du domaine de définition de X (ou d'un sous domaine commun de définition de X et de Y).

Les règles d'introduction sont:

$$[i\text{-}\Sigma_1]\ \frac{Xx}{\Sigma_1 X} \qquad [i\text{-}\Sigma_2]\ \frac{\&(Xx)(Yx)}{\Sigma_2 XY}$$

Nous utiliserons plus tard ces règles.

Donnons un autre exemple d'énoncé: *Marie admire un certain garçon*. Nous avons l'assignation suivante de types syntaxiques:

[Marie :N] x [admire :(P\N)/N):] x [un:(P/(P\N))/NC] x [garçon :NC].

Nous en déduisons la construction de la forme normale avec un quantificateur existentiel:

1. [Marie :N] x [admire :(P\N)/N):] x [un.:(P\(P\N))/NC] x [garçon :NC]
2. [**C**∗Marie: P/(P\N)] x [admire :(P\N)/N):] x [un :(P\(P\N))/NC] x [garçon :NC]
3. [**B**(**C**∗Marie)(admire) :P/N] x [un.:(P\(P\N))/NC] x [garçon :N]
4. [**B**(**C**∗Marie)(admire) :P/N] x [un garçon :(P\(P\N))]
5. [(un garçon) (**B**(**C**∗Marie)(admire)) :P]
6. [(Σ_2 garçon) (**B**(**C**∗Marie)(admire)) :P]

7. Prédicat complexe

En linguistique, les combinateurs sont les opérateurs qui permettent de construire des «prédicats complexes» à partir de prédicats plus élémentaires.

7.1. Prédicat réflexif

Cette notion de prédicat complexe a été entrevue par G. Geatch pour rendre compte, entre autres, des énoncés avec réflexifs. En effet, prenons l'exemple suivant:

Satan s'aime.

Quel est le rôle sémantique du réflexif *se*? Ce clitique a-t-il un simple rôle anaphorique avec une fonction syntaxique de «complément d'objet» ou bien assume-t-il un autre rôle? Lorsqu'on substitue (à la position près) *Satan* au clitique *se*, on obtient bien la paraphrase:

Satan s'aime -> Satan$_i$ aime Satan$_i$.

On pourrait par conséquent penser que *se* est la trace, en position d'objet, d'une anaphore du sujet syntaxique. Or, deux difficultés surgissent: premièrement, on peut avoir l'anaphorique lui-même en position d'objet avec la co-présence de *se*:

Satan s'aime lui-même

ensuite, avec une quantification universelle, le réflexif *se* n'assume pas la fonction anaphorique de renvoi au sujet syntaxique puisque l'on a pas la paraphrase:

Chacun s'aime ≠ chacun aime chacun.

La solution entrevue par Geatch, à la suite de recherches logiques menées au moyen âge, consiste à construire le prédicat complexe «aimer soi-même» qui s'applique alors directement au terme sujet. Geatch ne propose pas une formalisation adéquate de cette notion de prédicat complexe. Selon nous, la logique combinatoire est un cadre formel tout désigné pour formaliser cette notion. Le prédicat «aimer soi-même» est construit à partir du prédicat plus élémentaire «aimer» à l'aide du combinateur **W**. Nous le définissons comme suit:

[«aimer-soi-même» $=_{def}$ **W** aimer].

La phrase *Satan s'aime* est analysée par l'application du prédicat complexe «aimer-soi-même» au terme *Satan*, d'où la représentation applicative préfixée (l'opérateur précède l'opérande):

(«aimer-soi-même») (Satan).

La définition du prédicat complexe permet de résoudre les difficultés signalées. En effet:

1.	(«aimer-soi-même») (Satan)	hyp.
2.	[«aimer-soi-même» $=_{def}$ **W** aimer]	def
3.	(**W** aimer) (Satan)	rempl. 2., 3.
4.	aimer (Satan) (Satan)	[e-**W**]

d'où la paraphrase (techniquement, c'est une β-réduction entre expressions applicatives):

(«aime-soi-même») (Satan) β => aime Satan Satan

qui correspond à l'inférence suivante: de *Satan s'aime* on peut déduire que *Satan aime Satan*, les deux occurrences de *Satan* ayant pour référence le même individu /Satan/.

Le passage de l'expression concaténée à la forme normale s'effectue comme suit:

1.	[Satan :N] x [se :((P\N)\N)/((P\N)/N)] x [aime :(P\N)/N]	hyp.
2.	[**C**∗ Satan :(P/N)/((P\N)\N)] x [se :((P\N)\N)/((P\N)/N)] x [aime :(P\N)/N]	[**C**∗]
3.	[**B**(**C**∗ Satan)(se) : (P/N)/((P\N)/N)] x [aime :(P\N)/N]	[**B**]
4.	[**B**(**C**∗ Satan)(se) (aime) :P]	[>]
5.	[(**C**∗ Satan) (se aime) :P]	[e-**B**]
6.	[(se aime)(Satan) :P]	[e-**C**∗]
7.	[se aime $=_{def}$ **W** aime]	def
8.	(**W** aime) (Satan)	rempl., 7., 6.
9.	aime Satan Satan	[e-**W**]

Prenons maintenant la quantification universelle:

Chacun s'aime.

Cette phrase est analysée à l'aide de l'opérateur illatif Π_1; cet opérateur s'applique au prédicat complexe «aimer-soi-même»

pour construire une proposition, d'où les représentations applicatives équivalentes suivantes:

(chacun) (se aime) = Π_1(**W** (aimer))

Nous avons en effet le passage de la présentation de l'énoncé sous forme d'une expression concaténée typée à sa forme normale:

1.	[chacun : P/(P\N)] x [[se :((P\N)/((P\N)/N)] x [aime :(P\N)/N]	
2.	[**B**(chacun)(se) : (P/((P\N)/N)] x [aime :(P\N)/N]	[**B**], 1.
3.	[**B**(chacun)(se) (aime) :P]	[>], 2.
4.	[chacun =def Π_1]	def. de chacun
4.	[**B**Π_1 (se) (aime) :P]	rempl. 4., 3.
5.	[Π_1 (se aime) :P]	[e-**B**], 4.
6.	[(se)(aime) =$_{def}$ **W** aime]	def. se aime
7.	Π_1 (**W** aimer)	rempl. 6., 5.
8.	**E**(**W** aimer)	hyp.
9.	(**W** aime) (Satan)	[e-Π_1], 7., 8.
10.	aime (Satan) (Satan)	[e-**W**], 9.

ce qui correspond à la relation d'inférence en français:

> si chacun s'aime, étant donné Satan, alors Satan s'aime et donc Satan aime Satan.

On voit que, dans le contexte de l'énoncé analysé, la *trace linguistique* du combinateur **W** de réflexivité est *se* et celle de *chacun* est le quantificateur Π_1. Dans une quantification restreinte nous avons le même genre d'analyse. Prenons l'énoncé suivant:

Chaque homme s'aime

dont la représentation applicative construite est:

Π_2 (homme) (**W** aimer).

où «homme» et «**W** aimer» sont deux prédicats unaires. Nous avons comme dans le calcul précédent:

1. [chaque: (P/(P\N)/NC] x [homme: NC] x [se :((P\N)\N)/((P\N)/N)] x [aime :(P\N)/N]
2. [chaque homme :(P/(P\N)]x[se :((P\N)/((P\N)/N)] x [aime :(P\N)/N]

.....

3. [chaque homme $=_{def} \Pi_2$ (homme)] def.
4. [se aime $=_{def}$ W aime] def.

.....

5. Π_2 (homme) (W aimer)
6. homme (Satan) hyp.
7. (W aimer) (Satan) [e-Π_2]
8. aimer (Satan) (Satan) [e-W]

ce qui correspond à l'inférence:

> si chaque homme s'aime et si Satan est un homme alors Satan s'aime et donc Satan aime Satan.

Avec la notion de prédicat complexe, on peut analyser des énoncés tels que:

> *Satan a pitié de lui-même*
> (a-pitié-de-soi)(Satan).
>
> *Seul Satan a pitié de lui-même*
> (est-le-seul-à-avoir-pitié-de-soi) (Satan).
>
> *Satan n'a pitié que de lui-même*
> (n'a-pitié-que-de) (Satan).
>
> *Seul, Satan n'a pitié que de lui-même*
> (est-le-seul-à-n'avoir-pitié-que-de-soi) (Satan).
>
> *Chacun n'a pitié que de soi-même*
> (Chacun)(n'a-pitié-que-de-soi).

Nous n'indiquons pas ici comment on construit les prédicats complexes mais les techniques de représentation à l'aide des combinateurs sont les mêmes que dans les exemples précédents.

7.2. Prédicat avec ellipse

On peut, par la même méthode, rendre ainsi compte de la signification de:

Pierre aime sa femme. Jacques aussi

qui est ambiguë. Cette suite textuelle peut signifier:

soit: *Pierre aime sa propre femme et Jacques aime sa propre femme,*
soit: *Pierre aime sa femme et Jacques aime la femme de Pierre.*

Dans la première interprétation, il faut analyser et représenter le prédicat complexe «aimer-sa-propre-femme» et le combinateur **W** intervient de nouveau dans sa construction. Montrons comment traiter la relation paraphrastique:

1 *Jean aime sa (propre) femme* β=> *Jean$_i$ aime (la femme de Jean$_i$)*

Soit t le type des termes et p le type des propositions. Exprimons dans un premier temps l'opérateur unaire «la-mère-de» de type: t->t. Nous avons la β-réduction suivante:

(est-la-mère-de T) x β=> la-mère-de T.

En effet, avec les assignations de types:

type (a-pour-mère) = t->(t->p)
type(la-mère-de) = t->t
type(**K**(la -mère-de)) = t->(t->p)

nous avons le calcul suivant:

1.	**I**(est-la-mère-de T) x	hyp.
2.	**BI**(est-la-mère-de) T x	[i-**B**], 1.
3.	**C**(**BI**(est-la-mère-de)) x T	[i-**C**], 2.
4.	[a-pour-mère =$_{def}$ **C**(**BI**(est-la-mère-de))]	def.
5.	a-pour-mère x T	rempl. 4., 3.
6.	[a-pour-mère =$_{def}$ **K**(la -mère-de)]	def.
6.	**K**(la -mère-de) x T	rempl. 6., 5.
7.	la-mère-de T	[e-**K**], 6.

Montrons maintenant comment on peut construire le prédicat complexe «aimer-sa-(propre)-mère». En procédant par β-expansions successives (c'est-à-dire par introduction successives de combinateurs), nous obtenons:

1. aime (la-mère-de Jean) Jean	hyp.
2. **B** aime (la-mère-de) Jean Jean	[i-**B**], 1.
3. **W**(**B** aime (la-mère-de)) Jean	[i-**W**], 2.
4. **BW**(**B** aime) (la-mère-de) Jean	[i-**B**], 3.
5. **B**(**BW**)**B** aime (la-mère-de)	[i-**B**], 4.
6. [$\mathbf{REFL_2}$ =$_{def}$ **B**(**BW**)**B**]	def de $\mathbf{REFL_2}$
7. $\mathbf{REFL_2}$ aime (la-mère-de) Jean	rempl., 5., 6.
8. [aime-sa-(propre)-mère =$_{def}$ $\mathbf{REFL_2}$ aime (la-mère-de)]	def. du prédicat
9. aime-sa-(propre)-mère Jean	rempl., 7., 8.

Nous pouvons analyser maintenant:

Chaque homme aime sa (propre) mère.

En effet, nous avons:

1. (chaque homme)(aime-sa-mère)	hyp.
2. Π_2 (homme) (aime-sa-mère)	représent. de 1.
3. 3.1. (homme)(Jean)	hyp.
3.2. Π_2 (homme) (aime-sa-mère)	rappel, 2.
3.3. (aime-sa-mère)(Jean)	[e-Π_2], 3.1., 3.2.
3.4. [aime-sa-mère =$_{def}$ $\mathbf{REFL_2}$ aime (la-mère-de)]	def. du prédicat
3.5. $\mathbf{REFL_2}$ aime (la-mère-de)(Jean)	rempl., 3.4., 3.3.
3.6. [$\mathbf{REFL_2}$ =$_{def}$ **B**(**BW**)**B**]	def de $\mathbf{REFL_2}$
3.7. **B**(**BW**)**B** aime (la-mère-de) Jean	rempl., 3.6., 3.5.
3.8. aime (la-mère-de Jean) Jean	[e-**B**(**BW**)**B**]

c'est-à-dire que nous avons le raisonnement:

> Si chaque homme aime sa (propre) mère et si Jean est un homme alors Jean aime sa (propre) mère et donc Jean aime la mère de Jean.

7.3. Prédicat avec anaphore

Donnons un autre exemple où la représentation de la signification peut être traitée par un prédicat complexe. Prenons l'énoncé:

J'ai téléphoné à un électricien, il viendra demain.

L'occurrence de l'anaphorique *il* n'est pas mise pour *un électricien* (selon la théorie du pronom paresseux) puisque en remplaçant le pronom *il* par l'antécédent *un électricien,* nous n'obtenons pas un énoncé de même signification:

J'ai téléphoné à un électricien, un électricien viendra demain.

La signification de l'énoncé avec l'anaphorique *il* peut être révélée par la paraphrase suivante:

J'ai téléphoné à un électricien et cet électricien auquel j'ai téléphoné viendra demain

ce que l'on peut également exprimer dans la logique des prédicats du premier ordre par une quantification existentielle d'une variable d'individu x:

[∃ x)[(électricien (x) & ((ai-téléphoné-à (je,x)) & vient-demain (x))].

Nous donnons une solution applicative en considérant le prédicat complexe unaire:

«ai-téléphoné-à-qui-vient-demain»

auquel s'applique l'opérateur qui correspond au syntagme quantifié existentiellement.

Σ_2 (électricien) («ai-téléphoné-à-qui-vient-demain»(je)).

Nous en déduisons, d'après la règle d'élimination du quantificateur existentiel Σ_2:

1. Σ_2 (électricien) («ai-téléphoné-à-qui-vient-demain» (je))
2. &((électricien)(x)) («ai-téléphoné-à-qui-vient-demain» (je)) (x)) [e-Σ_2]

c'est-à-dire:

il existe un certain électricien et c'est à lui que j'ai téléphoné et ce dernier viendra demain.

7.4. Problème des «donkey sentences»

Les «donkey sentences» posent des problèmes de composition des unités linguistiques. En effet, les représentations d'un syntagme nominal avec article indéfini *un* ne sont pas identiques selon les contextes dans lesquelles ce syntagme s'insère. Pour nous en rendre compte, considérons l'énoncé:

(a) *Pedro possède un âne*

dont la représentation classique dans le calcul des prédicats est:

(a') $(\exists x)$ [âne(x) & possède (Pedro,x)].

Le syntagme quantifié *un âne* se voit représenter à l'aide d'un quantificateur existentiel. Considérons maintenant l'énoncé suivant:

(b) *Si Pedro possède un âne, alors il* [= Pedro] *est heureux.*

Le connecteur *si ... alors* est étroitement associé, comme on le sait, à une quantification universelle restreinte. La représentation de cet énoncé dans le calcul des prédicats est donc exprimée par une quantification universelle restreinte:

(b') $(\forall x)$[âne (x) & possède (Pedro,x) -> est heureux (Pedro)].

Cependant, dans le calcul des prédicats, nous avons l'équivalence entre cette représentation (b'), avec quantificateur universel, et la représentation suivante (b"), avec quantificateur existentiel:

(b") {$(\exists x)$[âne (x) & possède (Pedro, x)]} -> est heureux (Pedro)

la portée des opérateurs de quantification étant différente dans (b') et (b"). On voit donc que la composition des significations dans la représentation sémantique est conservée puisque le syntagme nominal *un âne* est toujours représenté (à une équivalence

d'écritures près) à l'aide d'une quantification existentielle dans les deux énoncés (a) et (b). Prenons maintenant un autre contexte avec l'énoncé:

(c) *Si Pedro possède un âne, alors il le* [= cet âne] *bat.*

Remarquons immédiatement que la «théorie du pronom paresseux» ne fonctionne pas ici puisque si on remplace l'anaphorique *le* par son antécédent *un âne*, on obtient alors un énoncé (d) qui ne conserve plus la signification initiale:

(d) *Si Pedro possède un âne, alors il bat un âne.*

La signification de l'énoncé analysé *Si Pedro possède un âne alors il le bat* est rendue transparente par la glose suivante:

(c') Si Pedro possède un âne il bat l'âne que Pedro possède

que nous pouvons représenter dans le calcul des prédicats par une quantification universelle restreinte:

(c") $(\forall x)$[âne (x) & possède (Pedro,x) -> bat (Pedro,x)].

Cette représentation n'est cependant plus équivalente à la représentation avec quantification existentielle et changement de portée de la quantification:

(e) {$(\exists x)$[âne (x) & possède (Pedro,x)]} -> bat (Pedro,y).

En effet, l'occurrence de y est libre et n'est pas liée par la quantification existentielle, cette dernière représentation n'exprime donc plus la signification de (c).

Ces trois énoncés (a) *Pedro possède un âne*; (b) *Si Pedro possède un âne, il est heureux* et (c) *Si Pedro possède un âne, il le bat* font apparaître des représentations différentes pour le syntagme nominal *un âne* selon les contextes où ce syntagme apparaît; ce dernier est tantôt représenté par une quantification existentielle, tantôt représenté par une quantification universelle. Il s'ensuit que *ou bien* les langues ne sont pas compositionnelles au sens suivant: la syntaxe doit guider directement la construction sémantique, *ou bien* que *la composition est étroitement*

dépendante du formalisme utilisé. Dans ce dernier cas, le calcul des prédicats n'est pas un formalisme qui laisse transparaître la composition du français.

Plusieurs théories tentent d'apporter une solution à ce problème avec plus ou moins d'élégance.. Citons par exemple les tentatives dans le cadre de la DRT de Kamp ou dans l'interprétation dynamique qui prolonge les analyses linguistiques de la Grammaire Universelle de R. Montague. Une solution formelle élégante a été apportée par A. Ranta (1994) qui fait appel à la théorie intuitionniste des types de Martin-Löf. La logique combinatoire illative permet de donner une autre solution formelle en construisant, à l'aide de combinateurs et du quantificateur universel illatif Π_2, le prédicat complexe:

«dès-que-on$_1$-possède-âne-on$_1$-le-bat».

Pour construire ce prédicat complexe, nous allons analyser auparavant l'énoncé suivant:

(f) *Si Pedro possède un âne alors Juan le vend.*

Nous partons de la forme normale glosée et représentée dans le formalisme applicatif par respectivement:

«si x est un âne et si Pedro possède cet x alors Juan vend ce même x»

-> (& (âne x) (possède x Pedro)) (vend x Juan).

Par des β-expansions successives qui introduisent des combinateurs et le quantificateur Π_2, nous construisons le prédicat complexe «dès-que-on$_1$-possède-un-âne-(quelconque)-on$_2$-le-vend» de façon à avoir une représentation applicative «sans variables liées». Nous avons:

1. -> (& (âne x) (possède x Pedro)) (vend x Juan) hyp.
2. -> (& (âne x) (Cpossède Pedro x)) (Cvend Juan x) [i-C], 1.
3. -> (Φ& âne (Cpossède Pedro) x) (Cvend Juan x) [i-Φ], 2.
4. Φ-> (Φ& âne (Cpossède Pedro)) (Cvend Juan) x [i-Φ], 3.
5. Π_2 (Φ& âne (Cpossède Pedro))(Cvend Juan) [i-Π_2], 4.
6. Π_2 (**B**(Φ& âne) (Cpossède) Pedro)(Cvend Juan) [i-**B**], 5.
7. **B**Π_2 (**B**(Φ& âne) (Cpossède)) Pedro (Cvend Juan) [i-**B**], 6.
8. **C**(**B**Π_2 (**B**(Φ& âne) (Cpossède))) (Cvend Juan) Pedro [i-**C**], 7.
9. **B**(**C**(**B**Π_2 (**B**(Φ& âne) (Cpossède)))) (Cvend) Juan Pedro [i-**B**], 8.
10. 10.1. **B**(**C**(**B**Π_2 (**B**(Φ& âne) (Cpossède)))) sous expression
    10.2. **BBC**(**B**Π_2 (**B**(Φ& âne) (Cpossède))) [i-**B**], 10.1
    10.3. **B**(**BBC**)(**B**Π_2)(**B**(Φ& âne) (Cpossède)) [i-**B**], 10.2
    10.4. $\mathbf{B^2}$(**B**(**BBC**)(**B**Π_2)) **B** (Φ& âne) (Cpossède) [i-$\mathbf{B^2}$], 10.3
11. $\mathbf{B^2}$(**B**(**BBC**)(**B**Π_2)) **B** (Φ& âne) (Cpossède)
    (Cvend) Juan Pedro rempl., 10.3, 9.
12. [tout-âne-que-l'on$_1$-possède, on$_2$-le-vend =$_{def}$
    $\mathbf{B^2}$(**B**(**BBC**)(**B**Π_2)) **B** (Φ& âne) (Cpossède) (Cvend)] def.
13. (tout-âne-que-l'on$_1$-possède, on$_2$-le-vend)
    (Juan) (Pedro) rempl. 12., 11.

Le prédicat complexe «dès-que-on$_1$-possède-âne-on$_2$-le-vend» est donc défini par une expression applicative de la logique combinatoire illative:

dès-que-l'on$_1$-possède-un-âne-(quelconque), on$_2$-le-vend
= tout-âne-que-l'on$_1$-possède, on$_2$-le-vend
=$_{def}$ $\mathbf{B^2}$ (**B**(**BBC**)(**B**Π_2)) **B** (Φ& âne) (**C**possède) (**C**vend).

Par la même méthode, exprimons maintenant le prédicat complexe:

«dès-que-l'on$_1$-possède-âne, on$_1$-le-bat».

Nous avons:

1.	-> (& (âne x) (possède x Pedro)) (bat x Pedro)	hyp.
2.	Φ-> (Φ& âne (Cpossède Pedro)) (Cbat Pedro) x	[i-Φ], 1.
3.	Φ-> (**B** (Φ& âne) (Cpossède) Pedro) (Cbat Pedro) x	[i-**B**], 2.
4.	Π_2 (**B** (Φ& âne) (Cpossède) Pedro) (Cbat Pedro)	[i-Π_2], 3.
5.	ΦΠ_2 (**B** (Φ& âne) (Cpossède)) (Cbat) Pedro	[i-Φ], 4.
6.	[dès-que-on$_1$-possède-âne-on$_1$-le-bat =$_{def}$ ΦΠ_2 (**B** (Φ& âne) (Cpossède)) (Cbat)]	def.
7.	(dès-que-on$_1$-possède-âne-on$_1$-le-bat) (Pedro)	rempl., 6,. 5.

Le prédicat complexe «dès-que-on$_1$-possède-âne-on$_1$-le-bat» est donc défini par une expression applicative de la logique combinatoire illative:

tout-âne-que-l'on$_1$-possède, on$_1$-le-bat =$_{def}$
ΦΠ_2 (**B**(Φ& âne) (**C**possède)) (**C**bat).

Nous avons donc représenté l'énoncé *Si Pedro possède un âne alors il le bat* par l'application du prédicat complexe «dès-que-on$_1$-possède-âne-on$_1$-le-bat» au terme «Pedro» sans faire appel à des variables liées:

(dès-que-l'on$_1$-possède-âne, on$_1$-le-bat)(Pedro).

8. Conclusion

Nous avons montré comment *le type syntaxique d'un énoncé spécifie l'organisation syntaxique de l'expression* (énoncé). A l'énoncé correspond un programme applicatif dont *la spécification syntaxique est donnée par son type*; ce programme décrit comment les unités linguistiques agissent en tant qu'opérateurs sur les autres pour construire dynamiquement une signification. Cependant, plusieurs programmes applicatifs peuvent correspondre à un même énoncé, *la construction d'un programme particulier dépendant étroitement* (d'après l'isomorphisme de Curry-Howard) *de la réduction inférentielle du type syntaxique*, c'est-à-dire de la stratégie adoptée pour réduire le type syntaxique au type des phrases. Le programme normalisé, que l'on

déduit du programme applicatif particulier construit à partir de la réduction inférentielle du type syntaxique, décrit la représentation applicative sous-jacente à l'énoncé. Il existe une stratégie (donnée par les métarègles de la Grammaire Catégorielle Combinatoire Applicative) pour: 1° construire un programme applicatif correspondant à l'énoncé examiné à partir de type syntaxique de cet énoncé (utilisation de règles inférentielles sur les types, guidées par des métarègles); 2° en déduire le programme applicatif normal sous-jacent à l'énoncé, en normalisant, par des β-réductions, le programme déjà construit; ce programme normal représente l'interprétation sémantique fonctionnelle de l'énoncé analysé.

La discussion sur la composition (ou la non composition) des significations associées aux unités linguistiques dans les langues doit être reprise dans le nouveau cadre formel plus large de la logique combinatoire. Plus généralement, c'est la quantification des termes nominaux dans les langues naturelles qui doit être «revisitée», notamment en intégrant aux opérations de quantification, de détermination et de prédication les problèmes relatifs aux représentants objectaux, plus ou moins typiques et atypiques, d'un concept; la logique classique des prédicats est incapable de prendre en compte ces problèmes et donc de les formaliser alors que la psychologie et la linguistique cognitives en ont reconnu la pertinence dans chaque processus de catégorisation. Il apparaîtrait alors que le cadre post-frégéen, bien que très élégant et relativement adéquat à la description de la plupart des énoncés mathématiques, reste trop étroit et très insuffisant pour l'expression formelle des significations entre des énoncés qui sont paraphrastiquement très proches mais que la linguistique cherche à différencier explicitement pour mieux étudier les opérations langagières qui sont constitutives des énoncés. Les analyses, effectuées par Aristote, Ockham et les Messieurs de Port-Royal sont sans doute plus respectueuses des structures linguistiques que les analyses post-frégéennes de Montague et de Dowty, comme en témoigne le débat ouvert par Fred Sommers. Elles devraient être réexaminées, sans doute avec fruit, à l'aide du formalisme applicatif «sans variables» de la logique combi-

natoire typée de Curry (inconnu, bien entendu, des auteurs anciens).

Institut des Sciences Humaines Appliquées
Université de Paris-Sorbonne
96, boulevard Raspail
F 75006 PARIS

Références bibliographiques

AJDUKIEWICZ K. (1935). Die syntaktische Konnexität. *Studia Philosophica* 1, 1-27.

BACH E. (1988). Categorial grammars as theories of language. In: Oehrle (1988), 17-34.

BAR HILLEL Y. (1953). A quasi-arthmetical notation for syntacitc description. *Language* 29, 47-58.

BISKRI I. (1996). Logique combinatoire et linguistique: grammaire catégorielle combinatoire applicative. *Mathématiques, Informatique et Sciences Humaines* 32, 39-68.

BISKRI I., DESCLÉS J.-P. (1996). Du phénotype au génotype: la grammaire catégorielle combinatoire applicative. *Actes de TALN*. Marseille, 87-96.

CURRY H.-B. (1930). Grundlagen der kombinatorischen Logik (Inauguraldissertation). *Amer. J. Math.*, 509-536, 789-834.

CURRY H.-B., FEYS R. (1958). *Combinatory Logic*. Amsterdam: Nort Holland, vol. 1.

CURRY H.-B., HINDLEY J.R. & SELDIN J.-P. (1992). *Combinatory Logic*. Amsterdam: North Holland, vol. 2.

DESCLÉS J.-P. (1980). *Opérateurs / opérations: méthodes intrinsèques en informatique fondamentale. Applications aux bases de données relationnelles et à la linguistique*. Paris: Thèse de doctorat de l'Université René Descartes.

DESCLÉS J.-P. (1990). *Langages applicatifs, langues naturelles et cognition*. Paris: Hermès.

DESCLÉS J.-P. (1991). La double négation dans l'Unum Argumentum analysé à l'aide de la logique combinatoire. In:

La négation. Le rôle de la négation dans l'argumentation et la raisonnement. Université de Neuchâtel: Travaux du Centre de Recherches Sémiologiques n° 59, 33-74

FITCH F. (1974). *Elements of Combanitory Logic*. Yale: Univsersity Press.

FREY L. (1967). Langages logiques et processus intellectuels. In: *Modèles et formalisation du comportement*. Paris: Editions du CNRS, 327-345.

GEACH P.T. (1962). *Reference and Generality*. Ithaca: Cornell Univ. Press.

GEACH P.T. (1971). A program for syntac. *Synthese*22, 3-17.

GIRARD J.Y. (1995). Linear logic: its syntax and semantics. In: J.Y. Girard et al. (eds), *Advances in Linear Logic*. Cambridge: Cambridge University Press, 1-42.

HARRIS Z.S. (1982). *A Grammar of English on Mathematical Principles*. New York: Wiley.

HOWARD W.A. (1980). The formulae-as-types notion of construction. In: J.-P. Seldin & J.R. Heldin (eds), *To H.B. Curry: Essays on Combinatory Logic. Lambda Calculus and Formalism*. London: Academic Press, 479-490.

HUSSERL E. (1913). *Logische Untersuchungen*. Halle: M. Niemeyer.

LAMBEK J. (1958). The mathematics of sentence structure. *American Mathematical Monthly* 65, 154-165.

LAMBEK J. (1961). On the calculus syntactic types. *Proceedings of Symposia in Applied Mathematics* vol. XII. Providence: America Mathematical Society, 166-178.

LAMBEK J. (1972). Deductive systems and categories. III Cartesian closed categories, intuitionist propositional calculus and combinatory logic. *Lectures Notes in Mathematics* 274, 57-82.

LAMBEK J., SCOTT P.J. (1986). *An Introduction to Higher Order Categorical Logic*. Cambridge: Cambridge University Press.

LESNIEWSKI S. (1989). *Sur les fondements de la mathématique. Fragments (Discussions préalables, méréologie, ontologie)*. Trad. de G. Kalinowski. Paris: Hermès.

OEHRLE R.T. et al. (eds) (1988). *Categorial Grammars and Natural Languages Structures*. Dordrecht: Reidel.

MARTIN-LÖF P. (1984). *Intuitionistic Type Theory*. Naples: Bibliopolis.

MOORTGAT M. (1988). *Categorial Investigation. Logical and Linguistic Aspects of the Lambek Calculus*. Dordrecht, Providence RI: Foris Pub.

PARTEE B.H., MEULEN A. & WALL R.E. (1993). *Mathematical Methods in Linguistics*. Dordrecht: Kluwer Academic Pub.

SHAUMYAN S.K. (1977). *Appplicational Grammar as a Semantic Theory of Natural Language*. Edinburgh: University Press.

SHAUMYAN S.K. (1987). *A Semiotic Theory of Natural Language*. Bloomington: Indina University Press.

STEEDMAN M. (1988). Cominators and grammars. In: Oherle (1988), 207-263.

STEEDMAN M. (1989). *Work in Progress: Combinators and Grammars in Natural Language Understanding*. Tuscon University: Summer Institute of Linguistic.

STEEDMAN M. (1989). Constituency and coordination in a combinatory grammar. In: M. Baltin & J. Kroch, *Alternative Conceptions of Phrase Structure*. Chicago: University Press, 201-231.

VAN BENTHEM J. (1991). *Language in Action: Categories, Lambdas and Dynamic Logic*. Amsterdam: North Holland.