3.1 Le principe de compositionnalité

3.1.1 Compositionnalité du sens

Le principe de compositionnalité est en général attribué à Frege, bien que cette paternité soit en fait sujette à caution. Ce principe, qu'on devrait appeler en toute rigueur principe de compositionnalité du sens, est motivé par l'observation que les humains, équipés d'un système cognitif nécessairement fini, sont capables de produire et de comprendre une infinité de phrases. L'ensemble de toutes les phrases possibles ne peut donc pas pré-exister sous forme de liste, il doit être le produit d'un mécanisme récursif (ou inductif). Par conséquent, l'association entre une phrase donnée et un sens, qui n'est pas aléatoire, ne peut pas non plus pré-exister sous forme de liste. Il faut supposer l'existence d'un mécanisme de **calcul** du sens des énoncés. Comme par ailleurs on peut observer que les phrases sont des objets structurés à partir d'éléments atomiques composés ensemble (par la syntaxe), il devient raisonnable de supposer que

> *Le sens d'une expression est une fonction du sens de ses parties,*
> *et de leur mode de composition*

Ce principe de compositionnalité est donc essentiellement une **hypothèse** d'ordre philosophique que l'on peut faire sur la nature de notre faculté de langage.

En pratique, il est facile de trouver des cas simples où, en français, ce principe semble être à l'œuvre. Par exemple, pour comprendre la phrase (1a), il faut (et il suffit) de connaître le sens de chacun des trois mots qui apparaissent dans (1a), **ainsi que** la façon dont ils sont combinés : ce sens n'est pas le même que celui de la phrase (1b), qui fait intervenir les mêmes composants.

(1) a. Jean aime Marie
b. Marie aime Jean

Au moins jusqu'à un certain point, les "sens" mis en jeu sont indépendants de la construction particulière considérée. C'est la raison pour laquelle on appelle quelquefois le principe de compositionnalité "principe de localité" : le sens est déterminé de manière locale, indépendante du contexte.

Il est important de souligner que ces "sens" élémentaires doivent se **combiner** : ainsi, le sens de *aime* correspond à un prédicat à deux places, lesquelles sont occupées respectivement par les sens de *Jean* et de *Marie* (à la bonne position).

En faisant une hypothèse minimale sur la structure syntaxique de (1a), et en utilisant une représentation en logique des prédicats de son "sens", on peut représenter tout cela dans la figure 3.1. Nous verrons plus loin une manière de formaliser rigoureusement cette "composition des sens".

Une conséquence de cette façon de voir est que l'on est conduit à attribuer un "sens" à des syntagmes postulés par la syntaxe. Ainsi, dans la figure 3.1, on aboutit à la conclusion que le sens de *aime Marie* est quelque chose comme $aime(x, m)$, où m est une constante, donc déterminée, et x une variable, c'est-à-dire ici une "place libre".

La mise en œuvre concrète du principe de compositionnalité que nous proposons ici suppose (1) que l'on fixe la méthode de composition syntaxique (en ce sens, on ne peut pas postuler,

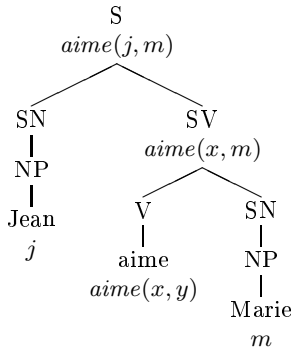

FIG. 3.1 – Décomposition et calcul du sens pour (1a)

comme on le fait quelquefois pour la syntaxe, une « autonomie de la sémantique »), et (2) que l'on couple rigoureusement chaque composition syntaxique avec une composition sémantique.

Ainsi, la figure 3.1 pourrait correspondre à la grammaire suivante :

S	$\rightarrow$	SN	SV
$P(a,b)$	$\leftarrow$	a	$P(x,b)$
SN	$\rightarrow$	NP	
a	$\leftarrow$	a	
SV	$\rightarrow$	V	SN
$P(x,b)$	$\leftarrow$	$P(x,y)$	b

On peut observer que les langages formels usuels, comme la logique des prédicats, les langages de programmation impératifs, ou encore le langage de l'arithmétique, vérifient tous rigoureusement ce principe de compositionnalité.

Par constraste, il est bien connu des linguistes que de nombreuses expressions, apparemment composées selon les règles de la syntaxe, n'obéissent pas au principe de compositionnalité du sens. Par exemple, les *expressions figées*, comme (2a), les collocations, comme (2b), sont clairement en contradiction avec le principe de compositionnalité : il n'est pas suffisant de connaître le sens de chacun des constituants pour comprendre le sens du tout : le sens ne peut être connu que si on l'a appris, exactement comme c'est le cas pour les mots du lexique.

(2) a. Paul a cassé sa pipe
b. Léa était dans une colère noire

Cette observation ne remet pas en cause l'hypothèse de Frege : il est clair qu'un locuteur du français doit connaître sous la forme d'une liste le sens de nombreux items linguistiques (mots, morphèmes, et, nous venons de le voir, expressions figées, etc.). Mais de nombreuses phrases (en fait, vraisemblablement, une infinité) se distinguent de ces items, en ce qu'il n'est pas nécessaire de les avoir apprises pour les comprendre. C'est dans ce cas que s'applique le principe de compositionnalité.

On peut aussi considérer que la compréhension d'énoncés comme les suivants remet en cause le principe de compositionnalité :

(3) a. Est-ce que tu peux me passer le sel ?
b. [En réponse à la question "Veux-tu rentrer ?"] J'ai un peu froid

En effet, pour comprendre ces énoncés, il faut vraisemblablement connaître plus que le sens des constituants : il faut connaître les conventions d'usage qui nous conduisent à interpréter, par défaut, ces énoncés comme signifiant autre chose que leur sens littéral. On considère que cet aspect de la signification relève du domaine de la *pragmatique*. Il faut noter que ce type d'exemple n'est qu'apparamment en contradiction avec le principe de compositionnalité : rien n'empêche (et on a même de bonnes raisons pour cela) de considérer que l'effet pragmatique évoqué se produit **à partir** du sens littéral, comme une sorte d'effet secondaire. Dans ce cas, le sens littéral est bien déterminé par composition des sens des parties.

C'est d'ailleurs le cas aussi pour les *métaphores*, dont on peut considérer que le sens se calcule en deux temps : d'abord un sens littéral, purement sémantique, et ensuite un sens métaphorique. Si ce deuxième sens n'obéit clairement pas toujours au principe de compositionnalité, rien ne nous oblige en revanche à exclure que le sens littéral s'obtient par un calcul compositionnel.

C'est ainsi que l'on peut, à l'intérieur du domaine de la sémantique, distinguer deux champs d'investigation assez radicalement distincts. Soit l'on s'intéresse à la partie compositionnelle du calcul du sens, on s'intéresse alors à la **grammaire** (dans un sens large), aux **règles** du type de celles qui ont été données en exemple plus haut. Le projet scientifique dans lequel on s'engage est un projet de **modélisation** d'une compétence, démarche qui s'apparente à la démarche du physicien qui modélise la réalité matérielle. Soit, en revanche, on s'intéresse à la partie non compositionnelle de la compétence linguistique, celle qu'il faut apprendre, celle qui pose le plus de problème en traduction. Alors la démarche est une démarche de **description**, qui consiste à établir une liste raisonnée de phénomènes pour l'essentiel arbitraires (au sens de Saussure). Cette démarche peut vraisemblablement être comparée à celle de l'entomologiste, tentant de mettre de l'ordre dans le foisonnement de la vie animale.

Le domaine de la sémantique formelle, qui finalement ne se distingue pas très nettement de ce que l'on appelle de plus en plus la question de l'interface syntaxe-sémantique, tombe principalement dans le premier cas évoqué : on y cherche des généralisations falsifiables concernant notre procédure de calcul du sens.

On parle quelquefois, d'une façon sans doute approximative, de sémantique non lexicale pour ce domaine. Ce n'est pas approprié dans le sens où la frontière entre compositionnel et non compositionnel ne peut pas être confondue avec la frontière entre lexical et grammatical : d'une part, à l'intérieur du lexique existent des règles d'organisation qui permettent de prédire des effets sémantiques ; d'autre part, comme nous l'avons vu, des expressions apparemment régies par la grammaire doivent essentiellement être vues comme des items lexicaux, dont le sens ne se calcule pas à partir de leur organisation interne.

3.1.2 Formalisation de la combinaison sémantique

On se place dans l'hypothèse où on représente la sémantique de phrases en langage naturel par une formule de la logique du premier ordre (cf. plus loin le programme montagovien).

Si on reprend la grammaire donnée plus haut, on voit qu'interviennent au niveau de la syntaxe et de la sémantique des opérations. Au niveau syntaxique, il s'agit simplement de

la concaténation, conformément aux définitions classiques d'une grammaire hors-contexte. Au niveau sémantique, il faut définir l'opération qui permet de combiner des "morceaux de formule". Par exemple, on a dit que la combinaison de $P(x, y)$ et de b donnait $P(x, b)$. Il est clair qu'il ne s'agit pas d'une simple concaténation, mais d'une opération de combinaison qui s'appuie sur la structure syntaxique des formules. De plus, il est nécessaire d'expliciter des aspects laissés implicites dans l'exemple précédent : il faut expliciter le fait que y (et aussi x, d'ailleurs) représente une « place » dans la formule résultante, et que b est suceptible de prendre cette place. Il faut aussi garantir, dans la combinaison précédente, que b va bien prendre la place marquée par y et non celle marquée par x.

On pourrait bien entendu recourir à un langage impératif dans lequel ces différents éléments seraient exprimés explicitement :

```
combiner(f1, f2)
{
  remplacer le 2e argument de f1 par f2 et renvoyer f1
}
```

C'est ce qu'on ferait avec un outil comme `yacc`, par exemple.

En fait, si on regarde de plus près, on voit que deux notions sont nécessaires pour formaliser la composition de morceaux de formules : d'une part, une notion de substitution (il faut être capable de requérir le remplacement de toutes les occurrences d'une variable par le même terme), et d'autre part, une notion d'application d'une formule incomplète à une autre.

On peut pour s'en convaincre considérer l'exemple suivant.

(4) a. Un enfant dort

b. [un(enfant)](dort)

c. $[\exists a(Ea \wedge Da)(\text{enfant}(x))]\,(\text{dormir}(y))$

Le λ-calcul est un outil formel extrêmement puissant (c'est-à-dire expressif) qui va précisément nous permettre de spécifier ces différents aspects, d'une manière que l'on peut qualifier de **déclarative**.

3.2 λ-calcul

3.2.1 Le langage pur (non typé)

Le lambda-calcul a été inventé par Alonzo Church, pour représenter la notion de fonction mathématique, d'une façon qui évite les problèmes de l'approche basée sur la théorie des ensembles.[1]

[1] En particulier, le fait que la notation $3x^2 + 7$ est ambiguë entre la définition de la fonction qui à x associe $3x^2 + 7$ et l'application de la fonction à un x particulier.

3.2.1.1 Syntaxe

Le **vocabulaire** est composé d'un ensemble dénombrable de symboles de variables, de parenthèses, du point, et du symbole λ.

La **syntaxe** est définie par induction :

- Si x est une variable, alors x est un λ-*terme*.
- Si x est une variable, et t un terme, alors $\lambda x.t$ est un λ-*terme* (λ-abstraction)
- Si t_1 et t_2 sont des termes, alors $(t_1)t_2$ est un λ-*terme* (application).

3.2.1.1.1 Remarques

- L'application d'un terme φ à un terme ψ est notée $(\varphi)\psi$, à l'inverse de la notation traditionnelle $f(x)$.
- Il n'y a pas de distinction entre variables fonctionnelles et variables arguments. On peut donc former des terme de la forme $\lambda f.(f)f$ (application à soi-même).
- Comme les quantificateurs en logique du premier ordre, l'abstracteur λ est un lieur de variables. On peut définir par conséquent les notions de variables libres et liées : [2]
  (a) La **portée** d'un abstracteur λx dans le terme $\lambda x.\varphi$ est le terme φ.
  (b) Une occurrence d'une variable x dans le terme φ est dite **libre** si elle n'est pas dans la portée d'un abstracteur λx apparaissant dans φ.
  (c) Si $\lambda x.\varphi$ est un sous-terme de ψ et que x est libre dans φ, alors cette occurrence de x est dite **liée** par l'abstracteur λx.

  Exemple : dans le terme $((\lambda x.\lambda y.(x)y)z)x$, la variable x a deux occurrences, l'une libre et l'autre liée.
- La grammaire définissant les λ-termes est non ambigüe, il n'y a qu'une façon de décomposer un terme. On peut définir la notion de sous-terme. Les notations (point et parenthèses) pouvant s'avérer assez lourdes, on adopte les conventions :

$$\begin{aligned} \lambda x_1.\lambda x_2 \ldots \lambda x_n.t &= \lambda x_1 x_2 \ldots x_n.t \\ (\ldots((t)t_1)t_2\ldots)t_m &= t\, t_1 t_2 \ldots t_m \end{aligned}$$

  Exemple : $\lambda xy.xy$ se lit $\lambda x.\lambda y(x)y$.
- L'application fonctionnelle est toujours monadique (s'applique à un argument). Toutes les fonctions sont "curryfiées" : une fonction à deux arguments est vue comme une fonction à un argument qui s'applique à un argument.

3.2.1.2 Conversions

3.2.1.2.1 Équivalence On définit la relation $\equiv$ entre termes. Intuitivement $t_1 \equiv t_2$ indique que t_1 et t_2 dénotent la même fonction.

Plus rigoureusement, on définit $\equiv$ comme
- une relation d'équivalence entre termes
- une congruence pour la λ-abstraction et l'application, i.e. :

[2] Version inductive :
- $VL(x) = \{x\}$ (où x est une variable)
- $VL(\ (t_1)t_2\ ) = VL(t_1) \cup VL(t_2)$ (où t_1 et t_2 sont des termes)
- $VL(\lambda x.t) = VL(t) \setminus \{x\}$

- $M \equiv N \Rightarrow \lambda x.M \equiv \lambda x.N$ pour toute variable x.
- $M \equiv N \Rightarrow (M)P \equiv (N)P$ & $(P)M \equiv (P)N$ pour tout terme P.

3.2.1.2.2 α-conversion Il est naturel, comme on le fait pour les formules quantifiées en logique du premier ordre, de considérer des termes qui ne diffèrent que par le nom des variables liées comme équivalents. C'est l'α-équivalence :

$$\lambda x.\varphi \equiv \lambda z._{[x:=z]}\varphi$$

La notation ${}_{[x:=z]}\varphi$ signifie que toutes les occurrences liées de x dans φ sont renommées z. Il ne s'agit pas d'un λ-terme du langage, mais d'une abbréviation pour le processus de renommage des variables, que l'on peut définir par induction :

- ${}_{[x:=z]}x \rightsquigarrow z$
- ${}_{[x:=z]}y \rightsquigarrow y$ si $y \neq x$
- ${}_{[x:=z]}(M)N \rightsquigarrow ({}_{[x:=z]}M)_{[x:=z]}N$
- ${}_{[x:=z]}\lambda x.M \rightsquigarrow \lambda z._{[x:=z]}M$
- ${}_{[x:=z]}\lambda y.M \rightsquigarrow \lambda y._{[x:=z]}M$ si $x \neq y$

Convention sur les variables Soit M un terme, x une variable ; les occurrences de x dans M sont soit libres soit toutes liées. On montre aisément que tout terme construit sans respecter la convention sur les variables est équivalent, à une α-conversion près, à un terme la respectant.

3.2.1.2.3 Substitution de termes On introduit une "procédure" de substitution d'un terme à une variable, notée de façon analogue au renommage des variables, et définie inductivement comme suit (on prend les précautions d'usage concernant les variables libres ou liées : aucune variable libre dans le terme que l'on substitue ne doit devenir lié dans le terme résultant) :

- ${}_{[x:=t]}x \rightsquigarrow t$
- ${}_{[x:=t]}y \rightsquigarrow y$ si $y \neq x$
- ${}_{[x:=t]}(M)N \rightsquigarrow ({}_{[x:=t]}M)_{[x:=t]}N$
- ${}_{[x:=t]}\lambda y.M \rightsquigarrow \lambda y._{[x:=t]}M$ si y n'est pas libre dans t.

Ce sont les seuls cas de substitution admissibles.

Remarque Les substitutions (de terme à une variable) se comportent comme les affectations d'un langage impératif.

Exemples
- ${}_{[x:=\lambda x.x]}\lambda y.(x)y \rightsquigarrow \lambda y.(\lambda x.x)y$
- ${}_{[x:=\lambda xy.xy]}\lambda y.(xz)xy \rightsquigarrow \lambda y.((\lambda x.\lambda y.(x)y)z)(\lambda x.\lambda y.(x)y)y$
  (repasser par la forme entièrement parenthésée)

3.2.1.2.4 Beta-conversion (ou β-équivalence) :

$$(\lambda x.M)N \equiv {}_{[x:=N]}M$$

(suivant la convention sur les variables, la substitution est toujours valide.)

La β-conversion est une règle de **calcul** (on parle de β-réduction lorsque l'on "réduit" $(\lambda x.M)N$ en ${}_{[x:=N]}M$) ; c'est même la **seule** règle primitive de calcul du λ-calcul (modulo l'α-conversion), et celle que nous étudierons le plus longuement.

Exemples
- $(\lambda x.x)y \equiv y$ ("identité")
- $(\lambda xy.x)zt \equiv z$ ("projection")
- $(\lambda x.xx)y \equiv yy$ ("application à soi-même")
  d'où $(\lambda x.xx)\lambda x.xx \equiv (\lambda x.xx)\lambda x.xx$: la β-réduction ne réduit pas nécessairement la taille du terme.

Remarques
- On appelle **(β)-redex** un terme de la forme $(\lambda x.M)N$, et **contractum** (ou forme contractée) le terme ${}_{[x:=N]}M$.
- Un terme est dit **en forme normale** s'il ne contient pas de redex comme sous-terme. Par exemple $\lambda z.(z)x$, et pas $(\lambda z.z)x$.
- Théorème : il existe des redex qui n'ont pas de forme normale. Par exemple $(\lambda x.xx)\lambda x.xx$.
- Théorème (existence d'un point fixe) : $\forall M \exists N$ t.q. $MN \equiv N$.
  Démonstration Soit $W = \lambda x.(M)xx$ et $N = WW$, $x \notin VL(M)$. Alors

$$N = (\lambda x.(M)xx)W = (M)WW = MN)$$

□

3.2.1.3 Combinateurs

Le λ-calcul, tel qu'il a été présenté, est extrêmement abstrait, et, à ce titre, peu maniable ; on va s'intéresser ici à des « constructeurs » dérivés.

Définition Un **combinateur** est un λ-term clos, i.e. sans variable libre.

Remarque : un combinateur se comporte de façon uniforme (indépendante du contexte), et ainsi peut être assimilié à une *constante* du langage. À noter qu'un combinateur est défini (comme tout λ-terme) à une α-conversion près.

Quelques combinateurs et leurs propriétés

Identité $\mathsf{I} =_{\text{def}} \lambda x.x$

Pour tout terme t, on a $(\mathsf{I})t \equiv t$.

Composition $\mathsf{C} =_{\text{def}} \lambda f.\lambda g.\lambda x(f)(g)x$

Pour tous termes M et N, on a : $\mathsf{C}MN(= ((\mathsf{C})M)N) \equiv \lambda x.(M)(N)x$.

Booléens $\mathsf{T} =_{\text{def}} \lambda x.\lambda y.x$
$\mathsf{F} =_{\text{def}} \lambda x.\lambda y.y$

Ces définitions, conventionnelles, s'expliquent par la forme très simple que reçoit alors la définition par cas : $\texttt{if}\ P\ \texttt{then}\ Q\ \texttt{else}\ R =_{\text{def}} PQR$.

En effet, si P se réduit en T (ie $P \equiv \mathsf{T}$), alors $\texttt{if}\ P\ \texttt{then}\ Q\ \texttt{else}\ R \equiv \mathsf{T}QR \equiv Q$. De même, si $P \equiv \mathsf{F}$, alors on obtient R.

On peut alors définir relativement aisément les combinateurs booléens, en passant par la définition précédente : $\neg P = \texttt{if}\ P\ \texttt{then}\ \mathsf{F}\ \texttt{else}\ \mathsf{T} = P\mathsf{FT}$ (ou $((P)\mathsf{F})\mathsf{T}$). D'où par λ-abstraction : $\mathsf{NOT} =_{\text{def}} \lambda x.((x)\mathsf{F})\mathsf{T}$.

Alors, on peut vérifier que si $P \equiv \mathsf{T}$, alors $\mathsf{NOT}P \equiv ((P)\mathsf{F})\mathsf{T} \equiv ((\mathsf{T})\mathsf{F})\mathsf{T} \equiv \mathsf{F}$.

En exercice (p. 13), on peut définir les opérateurs $\wedge$, $\vee$, et vérifier les propriétés usuelles.

Paires Trois notations : $[M, N] =_{\text{def}} \lambda z.zMN$
$(M)_0 =_{\text{def}} M\mathsf{T}$
$(M)_1 =_{\text{def}} M\mathsf{F}$

Alors on a $([M, N])_0 \equiv M$ et $([M, N])_1 \equiv N$

On peut généraliser :

Listes finies $[M_0, M_1, \ldots M_n] =_{\text{def}} [M_0, [M_1, [\ldots, M_n]\ldots]]$
« $M_0, M_1, \ldots M_n$ » $=_{\text{def}} \lambda z.zM_0M_1 \ldots M_n$

Entiers Il existe plusieurs façons de représenter les entiers par des λ-termes ; la plus naturelle est celle de Church, où les entiers sont conçus comme des itérateurs :
$\mathsf{0} =_{\text{def}} \lambda f.\lambda x.x$
$\mathsf{1} =_{\text{def}} \lambda f.\lambda x.(f)x$
$\mathsf{n} =_{\text{def}} \lambda f.\lambda x.(f)(f)\ldots(f)x$, avec f n fois.

On peut alors représenter la fonction successeur, l'addition et la multiplication :
$\mathsf{Succ} =_{\text{def}} \lambda n.\lambda f.\lambda x.(f)((n)f)x$
$+ \equiv \lambda m.\lambda n.\lambda f.\lambda x.((m)f)((n)f)x$
$* \equiv \lambda m.\lambda n.\lambda f.(m)(n)f$

Récursion Toute définition récursive peut être considérée comme une équation définissant le point fixe d'un opérateur d'ordre supérieur.

Par exemple, la définition classique de la factorielle, « fact n : si $n = 0$ alors 1 sinon $n\times$fact$(n-1)$ » est un point fixe de l'opérateur Φ défini par : $\Phi(f)(0) = 1$
$\Phi(f)(n) = n \times f(n-1)$

La puissance du λ-calcul (ainsi que certains de ses aspects paradoxaux) est illustrée par le fait que :
- Tout λ-terme possède un point fixe
- Il existe un combinateur (en fait une infinité) qui "produit" le point fixe de tout λ-terme

Définition Un combinateur de point fixe P est un combinateur tel que

$$\forall M \quad PM \equiv (M)PM$$

Un tel combinateur peut être extrait de la démonstration du thèorème du point fixe : on a vu que si $W = \lambda x.(F)(x)x$ alors
$WW \equiv (\lambda x.(F)(x)x)\lambda x.(F)(x)x \equiv FWW$.
En posant alors $\mathsf{Y} =_{\text{def}} \lambda F.WW \equiv \lambda f.(\lambda x.(f)(x)x)\lambda q.(f)(x)x$, on a
$\mathsf{Y}M \equiv WW_{[F:=M]} \equiv (M)\mathsf{Y}M$ en vertu du théorème du point fixe.

Exemple boucle `while` :
La définition (récursive) de `while` P `do` Q `else` R `od` est :
`while` PQR = `if` NOT P `then` R else $Q \circ$ `while` PQR
`while` est donc le point fixe de l'opérateur :
$\Phi = \lambda F.\lambda x.\lambda y.\lambda z(\mathsf{NOT})xz\mathsf{C}yFxyz$[3]
et donc `while` $\equiv \mathsf{Y}\Phi$

Remarque Y n'est pas le seul combinateur de point fixe. Un autre combinateur "classique" de point fixe est dû à Turing : $\mathsf{T} \equiv \mathrm{AA}$, où $\mathrm{A} \equiv \lambda xy.(y)((x)x)y)$ (erreur de parenthèses)

3.2.2 Langage typé

Le lambda-calcul non typé est très (trop) puissant pour les aspects de sémantique compositionnelle qui nous intéressent ici. Ce qui a fait du lambda-calcul un outil précieux en sémantique contemporaine est la reconnaissance que la plupart des constructions de base du français peuvent être interprétées compositionnellement comme impliquant une application fonctionnelle, et que de nombreuses constructions apparemment moins basiques peuvent recevoir une sémantique compositionnelle impliquant la lambda-abstraction.

3.2.2.1 Théorie des types

Un système de types, dans le sens où nous allons l'utiliser ici, est un système de catégories motivées sémantiquement de telle manière que des restrictions sur la bonne formation formulées en termes de types peuvent garantir que toutes les expressions bien formées ont une sémantique bien définie.[4] Le système que nous proposons ici est un des plus simples et des plus courants :

L'ensemble des types est défini par induction :

1. e est un type
2. t est un type
3. Si a et b sont des types, alors $\langle a, b\rangle$ est un type

Un langage typé est un langage dont chaque *fbf* se voit assigner un type par une syntaxe compositionnelle dont la sémantique se conforme aux principes suivants (où D_a dénote l'ensemble des dénotations possibles des expressions de type a) :

Soit A un domaine d'entités.
– $D_e = A$
– $D_t = \{0, 1\}$
– $D_{\langle a,b\rangle}$ = l'ensemble des fonctions de D_a dans D_b.
Le calcul des prédicats peut être défini comme un language typé dans le sens précédent : les constantes individuelles et les variables sont de type e, les formules (fermées) sont de type t, les prédicats à une place (qui dénotent des ensembles) sont de type $\langle e, t\rangle$, les prédicats à deux places sont de type $\langle e, \langle e, t\rangle\rangle$, etc.

[3] C est l'opérateur de composition vu p. 9.

[4] Un tel système a été introduit par Russel dans la théorie des ensembles (et des fonctions) pour rendre les ensembles paradoxaux impossibles à exprimer.