N° d'ordre:

THESE DE DOCTORAT

présentée à

l'UNIVERSITE PARIS VII

Spécialité : Informatique

présentée par

Yves LAFONT

Sujet de la thèse :

LOGIQUES, CATEGORIES & MACHINES

Implantation de Langages de Programmation guidée par la Logique Catégorique

Soutenue le 7 janvier 1988, devant la Commission d'examen composée de :

MM.	J.L. Krivine	Président
	S. Abramsky G. Berry G. Cousineau P.L. Curien J.Y. Girard G. Huet	Examinateurs

N° d'ordre:

THESE DE DOCTORAT

présentée à

l'UNIVERSITE PARIS VII

Spécialité : Informatique

présentée par

Yves LAFONT

Sujet de la thèse :

LOGIQUES, CATEGORIES & MACHINES

Implantation de Langages de Programmation guidée par la Logique Catégorique

Soutenue le 7 janvier 1988, devant la Commission d'examen composée de :

MM.	J.L. Krivine	Président
	S. Abramsky G. Berry G. Cousineau P.L. Curien J.Y. Girard G. Huet	Examinateurs

LOGIQUES, CATEGORIES & MACHINES

Implantation de Langages de Programmation guidée par la Logique Catégorique

(Thèse de Doctorat)

Y. Lafont
Université de Paris 7

11 Janvier 1988

Résumé

Cette thèse (doctorat d'université) comporte trois chapitres:

De la Déduction Naturelle à La Machine Catégorique

CAML est un *langage fonctionnel interactif compilé* particulièrement adapté au développement de logiciels complexes, tels que:

- langages de programmation
- systèmes de preuve
- maquettes de systèmes experts ...

L'implantation de CAML est fondée sur la *Machine Catégorique Abstraite* [CouCurMau], elle-même issue des *combinateurs catégoriques* de J. Lambek.

Nous donnons les fondements mathématiques de ce type d'implantation:

- Le mécanisme d'évaluation catégorique est obtenu à partir de la preuve d'un résultat mathématique.
- L'exécution d'un programme *n'est pas* un processus de normalisation.

Ce premier chapitre constitue une vue d'ensemble sur les rapports entre *logique*, *théorie des catégories* et *informatique* dans le cas de la programmation fonctionnelle.

La Machine Linéaire Abstraite

Dès 1986, Jean-Yves Girard commence à populariser sa *Logique Linéaire* en participant à plusieurs colloques d'informatique théorique. Son travail [Girard87] a suscité beaucoup d'intérêt dans la communauté informatique, car il s'agit d'une approche radicalement nouvelle dans le domaine des fondements de la programmation.

A la lumière de nos travaux sur la *Machine Catégorique*, nous y trouvons la solution de *problèmes* qui accompagnent la *programmation fonctionnelle* depuis son origine:

- Les *traits impurs* (effets de bord, affectations, ...)
- La récupération de mémoire (*garbage collection*).
- Les deux modes d'évaluation (strict et paresseux)

Le deuxième chapitre contient une introduction à cette nouvelle logique, ainsi que les principes de la *Machine Linéaire*.

Un Langage Linéaire

LIVE (la *version linéaire* de CAML) est un nouveau langage qui intègre des caractéristiques non fonctionnelles, tout en conservant la nature *déclarative* des *langages fonctionnels purs*. Il est implanté sur la *Machine Linéaire* (pas besoin de *garbage collector*!).

La description de cette implantation constitue le troisième et dernier chapitre, qui présuppose une certaine familiarité avec CAML.

Mots-clefs

- Calcul des Séquents
- Catégorie Cartésienne Fermée
- Catégorie Monoïdale
- Combinateur Catégorique
- Déduction Naturelle
- Effet de Bord
- Evaluation
- λ-Calcul
- Logique Intuitionniste
- Logique Linéaire
- Machine Abstraite
- Programmation Fonctionnelle
- Récupération de la Mémoire

Remerciements (Acknowledgments)

Le premier et le troisième chapitre de cette thèse puisent largement dans les travaux du projet Formel (en particulier Guy Cousineau, Pierre-Louis Curien, Michel Mauny et Ascánder Suárez) sur la *Machine Catégorique* et l'implantation de CAML.

J'ai eu la chance d'assister à la naissance de la *Logique Linéaire*, et Jean-Yves Girard, qui est mon directeur de thèse, m'a beaucoup encouragé dans ce travail. J'ai apprécié sa compétence inégalée dans le domaine de la *Théorie de la Démonstration*.

Les idées importantes sont apparues au cours de l'année 1985-86, but the thesis was written the following year in London, so I would like to thank the members of the *Theory and Formal Methods* Group and the Computing Department of Imperial College who invited me as research assistant during my French National Service. There I found a very good environment to finish this work. I am especially grateful to Samson Abramsky who agreed to be my local supervisor and whose outstanding competence I much appreciate.

Pour l'ensemble de la thèse, j'ai été motivé par des discussions fructueuses avec plusieurs membres du projet Formel, notamment Thierry Coquand, Pierre-Louis Curien et Ascánder Suárez. En ce qui concerne la *Théorie des Catégories*, j'ai également profité de la science de Jean Bénabou, Albert Burroni et Jacques Penon.

Enfin je dois évidemment mentionner le soutien infiniment précieux que Gérard Huet m'a apporté tout au long de ce travail, et malgré l'éloignement (lorsqu'il était aux Etats-Unis, puis lorsque j'étais au Royaume-Uni).

Table des matières

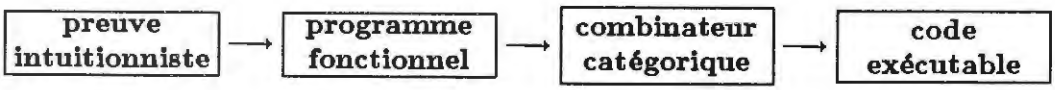
Chapitre 1

De la Déduction Naturelle à la Machine Catégorique

Le titre de ce chapitre résume trois *slogans*:

- La *Déduction Naturelle*, c'est du *λ-Calcul* [CurFeys].
- Le *λ-Calcul*, c'est la théorie des *Catégories Cartésiennes Fermées* [Lambek68].
- La théorie des *Catégories Cartésiennes Fermées*, c'est du *langage machine* [CouCurMau].

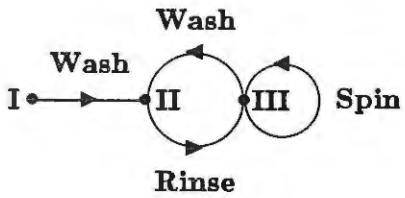
On réalise ainsi le *Programme de Constructivisation des Mathématiques*:



Toutefois, on se restreint ici au *calcul propositionnel*. Il serait intéressant d'étendre ces résultats aux diverses *théories des types*[1] (voir par exemple [Martinloef,Girard71,LamSco]).

Initialement, nous voulions définir une sorte de *sémantique opérationnelle* des programmes.

Un programme est censé produire des résultats, par exemple en envoyant un flot d'informations à un terminal ou à une autre machine. On a donc un *langage d'actions primitives*:



On considère la *catégorie cartésienne fermée* engendrée par ce graphe: Les objets sont des *formules du calcul propositionnel*, considérés comme des *types de donnée*, et les morphismes sont représentés par des *Combinateurs Catégoriques*, considérés comme des *programmes* (section 1.2).

Pour que le *combinateur* $\varphi : A \to B$ soit exécutable, il faut que A et B soient des formules atomiques (ici, les formules atomiques sont **I**, **II** et **III**). Dans ce cas, il existe un unique $\alpha : A \to B$

[1] On a des formulations catégoriques pour le *système de Martin-Löf* [Seely84,Cartmell] et le *λ-calcul d'ordre supérieur* [Seely86,CoqEhr]. Voir aussi [LamSco].

composé d'actions primitives et congru à φ modulo les *équations catégoriques*. C'est ce qu'exprime le *Théorème de Cohérence Hiérarchique* (section 1.3).

C'est pour *démontrer* ce théorème que nous introduisons les *sémantiques* des *formules* et des *combinateurs* qui expliquent le *mécanisme d'évaluation* de la *Machine Catégorique* (section 1.4).

Historiquement, la *Machine Catégorique* n'est pas apparue de cette façon. Mais nous disposons désormais d'une *méthode* pour aborder d'autres problèmes d'implantation. C'est dans cet esprit que la *Machine Linéaire* a été introduite (voir le deuxième chapitre).

Les deux premières sections présentent les différents systèmes de preuves intuitionnistes: *Déduction Naturelle*, *λ-Calcul Typé* et *Combinateurs Catégoriques*. Vient ensuite la section la plus importante avec la preuve du *Théorème de Cohérence Hiérarchique*, version *élémentaire* de la preuve "catégorique" donnée en annexe. La section 1.4 présente la *Machine Catégorique*.

Pour comprendre les sections suivantes, il est utile d'avoir une certaine familiarité avec CAML. Il est difficile d'expliquer la compilation de la conditionnelle sans la notion de *type dépendant* pour laquelle nous n'avons pas de formalisme satisfaisant (section 1.5). Quant aux *effets de bords* et aux *échappements*, nous renonçons tout simplement à les intégrer dans notre formalisme (section 1.6). La dernière section contient quelques remarques sur les *optimisations*.

1.1 La Déduction Naturelle

La *Déduction Naturelle* [Prawitz] est un *système formel* de preuves *intuitionnistes*, assez proche du raisonnement mathématique.

Une *déduction* est un *arbre* dont les *feuilles* sont les hypothèses et la *racine* est la conclusion. Cet arbre est construit au moyen de *règles de déductions*.

Voici par exemple une *déduction* de B à partir de l'*hypothèse* $A \wedge (A \Rightarrow B)$:

$$\dfrac{\dfrac{A \wedge (A \Rightarrow B)}{A \Rightarrow B} \quad \dfrac{A \wedge (A \Rightarrow B)}{A}}{B}$$

Certaines règles permettent de *décharger* des hypothèses, comme dans cette déduction de $B \Rightarrow (A \wedge B)$ à partir de A:

$$\dfrac{\dfrac{A \quad [B]}{A \wedge B}}{B \Rightarrow (A \wedge B)}$$

Dans cet exemple, la formule B est *déchargée* (notation: $[B]$): Elle n'est plus considérée comme une hypothèse.

1.1.1 Règles de déduction

On se limite aux connecteurs propositionnels: $\top$ (vrai), $A \wedge B$ (conjonction), $\bot$ (absurde), $A \vee B$ (disjonction) et $A \Rightarrow B$ (implication). Les *règles de déduction* sont très *naturelles*:

(A, B, C dénotent des formules quelconques.)

$$\frac{}{\top} \qquad \frac{\vdots}{\dfrac{A \wedge B}{A}} \qquad \frac{\vdots}{\dfrac{A \wedge B}{B}} \qquad \frac{\vdots \quad \vdots}{\dfrac{A \quad B}{A \wedge B}}$$

$$\frac{\vdots}{\dfrac{\bot}{A}} \qquad \frac{\vdots}{\dfrac{A}{A \vee B}} \qquad \frac{\vdots}{\dfrac{B}{A \vee B}} \qquad \frac{\vdots \quad [A] \quad \vdots \quad [B] \quad \vdots}{\dfrac{A \vee B \qquad C \qquad C}{C}}$$

$$\frac{\vdots \quad \vdots}{\dfrac{A \Rightarrow B \quad A}{B}} \qquad \frac{[A]}{\dfrac{\vdots}{\dfrac{B}{A \Rightarrow B}}}$$

- On a toujours $\top$.
- De $A \wedge B$, on déduit A (on déduit aussi B).
- De A et B, on déduit $A \wedge B$.
- De $\bot$, on déduit n'importe quoi.
- De A on déduit $A \vee B$ (de B aussi).
- Si on a $A \vee B$ et si on a prouvé C à partir A et de même à partir de B, alors on a prouvé C (preuve par cas).
- De $A \Rightarrow B$ et A on déduit B (modus ponens).
- Si on a prouvé B à partir de A, on a prouvé $A \Rightarrow B$.

Chaque règle fait intervenir ***une seule occurrence de connecteur***, soit en conclusion (règle d'introduction), soit en hypothèse (règle d'élimination). La formule contenant cette occurrence est appelée ***formule principale*** (de l'introduction ou de l'élimination).

1.1.2 Coupures

On peut toujours prouver les choses simples de façon compliquée. Voici par exemple une déduction ***compliquée*** de $B \Rightarrow (A \wedge B)$ à partir de A:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A] \quad [B]}{A \wedge B}}{A \Rightarrow (A \wedge B)}}{B \Rightarrow (A \Rightarrow (A \wedge B))} \quad [B]}{A \Rightarrow (A \wedge B)} \quad A}{A \wedge B}}{B \Rightarrow (A \wedge B)}$$

Cette déduction contient une *coupure*[2], c'est à dire une règle d'*élimination* dont la *formule principale* est *conclusion* d'une règle d'*introduction*.

Il y a cinq cas de *coupure* de ce genre:

$$\dfrac{\dfrac{\vdots\ \vdots}{A\quad B}{\scriptstyle \wedge I}}{A}{\scriptstyle \wedge E_1} \qquad \dfrac{\dfrac{\vdots\ \vdots}{A\quad B}{\scriptstyle \wedge I}}{B}{\scriptstyle \wedge E_2} \qquad \dfrac{\dfrac{\vdots}{A}{\scriptstyle \vee I_1}\quad \begin{matrix}[A]\\ \vdots\\ C\end{matrix}\quad \begin{matrix}[B]\\ \vdots\\ C\end{matrix}}{C}{\scriptstyle \vee E} \qquad \dfrac{\dfrac{\vdots}{B}{\scriptstyle \vee I_2}\quad \begin{matrix}[A]\\ \vdots\\ C\end{matrix}\quad \begin{matrix}[B]\\ \vdots\\ C\end{matrix}}{C}{\scriptstyle \vee E} \qquad \dfrac{\begin{matrix}[A]\\ \vdots\\ B\end{matrix}\quad \vdots}{A \Rightarrow B\quad A}{\scriptstyle \Rightarrow E}{\scriptstyle \Rightarrow I}{B}$$

Comme l'une des règles élimine ce que l'autre introduit, la déduction peut être "simplifiée" (*élimination* de la coupure):

$$\dfrac{\dfrac{\vdots\ \vdots}{A\quad B}{\scriptstyle \wedge I}}{A}{\scriptstyle \wedge E_1} \longrightarrow \begin{matrix}\vdots\\ A\end{matrix} \qquad \dfrac{\dfrac{\vdots}{A}{\scriptstyle \vee I_1}\quad \begin{matrix}[A]\\ \vdots\\ C\end{matrix}\quad \begin{matrix}[B]\\ \vdots\\ C\end{matrix}}{C}{\scriptstyle \vee E} \longrightarrow \begin{matrix}\vdots\\ A\\ \vdots\\ C\end{matrix} \qquad \dfrac{\dfrac{\begin{matrix}[A]\\ \vdots\\ B\end{matrix}\quad \vdots}{A \Rightarrow B\quad A}{\scriptstyle \Rightarrow E}{\scriptstyle \Rightarrow I}{B} \longrightarrow \begin{matrix}\vdots\\ A\\ \vdots\\ B\end{matrix}$$

En général, l'élimination d'une coupure fait apparaître d'autres coupures que l'on peut éliminer à nouveau, et ainsi de suite ...

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]\quad [B]}{A \wedge B}{\scriptstyle \wedge I}}{A \Rightarrow (A \wedge B)}{\scriptstyle \Rightarrow I}\quad [B]}{B \Rightarrow (A \Rightarrow (A \wedge B))}{\scriptstyle \Rightarrow I}\quad A}{A \Rightarrow (A \wedge B)}{\scriptstyle \Rightarrow E}}{A \wedge B}{\scriptstyle \wedge E_1}}{B \Rightarrow (A \wedge B)}{\scriptstyle \Rightarrow I} \longrightarrow \dfrac{\dfrac{\dfrac{[A]\quad [B]}{A \wedge B}{\scriptstyle \wedge I}}{A \Rightarrow (A \wedge B)}{\scriptstyle \Rightarrow I}\quad A}{A \wedge B}{\scriptstyle \Rightarrow E}}{B \Rightarrow (A \wedge B)}{\scriptstyle \wedge E_1} \longrightarrow \dfrac{A\quad [B]}{A \wedge B}{\scriptstyle \wedge I}}{B \Rightarrow (A \wedge B)}{\scriptstyle \Rightarrow I}$$

Il n'est pas du tout évident que le processus d'*élimination des coupures* termine toujours. En éliminant une coupure, on peut augmenter la taille d'une déduction en *dupliquant* des branches:

$$\dfrac{\dfrac{\dfrac{[A]\quad [A]}{A \wedge A}{\scriptstyle \wedge I}}{A \Rightarrow (A \wedge A)}{\scriptstyle \Rightarrow I}\quad \vdots\ A}{A \wedge A}{\scriptstyle \Rightarrow E} \longrightarrow \dfrac{\vdots\ \vdots}{A\quad A}{\scriptstyle \wedge I}}{A \wedge A}$$

Les connecteurs $\bot$ et $\vee$ compliquent un peu la situation. Il faut également considérer les *coupures commutatives* (annexe B).

[2]La notion de *coupure* a été introduite par Gentzen, pour son *Calcul des Séquents* (annexe A). La *coupure* est l'une des règles de son calcul.

1.1.3 Le Théorème de Normalisation

Nous pouvons maintenant énoncer une propriété fondamentale des déductions *sans coupure*:

Proposition 1 *(Propriété de la Sous-formule)*
Dans une déduction sans coupure, il n'y a que des sous-formules de la conclusion ou des hypothèses.

Démonstration: Par récurrence sur la déduction:
Si la dernière règle est une *introduction*, c'est évident (il faut traiter à part l'introduction de $\Rightarrow$, à cause de l'hypothèse déchargée).
Si la dernière règle est une *élimination* de formule principale A, on montre que A est sous-formule d'une hypothèse. En effet, comme il n'y a pas de coupure, la règle dont provient A n'est pas une introduction. On conclue en remarquant qu'une élimination de $\wedge$ ou de $\Rightarrow$ produit toujours une sous-formule de la formule principale[3]. □

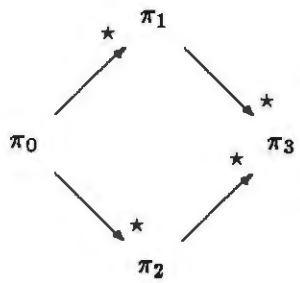
En particulier, on a:

Corollaire 1 *La dernière règle d'une déduction sans hypothèse et sans coupure est toujours une introduction.*

On va montrer qu'il est toujours possible d'éliminer les coupures:
On écrit $\pi \longrightarrow \pi'$ lorsque, par élimination d'une coupure à l'intérieur de π, on obtient π'. On note $\longrightarrow^*$ la fermeture reflexive et transitive de $\longrightarrow$.

Théorème 1 *(Normalisation)*
La relation $\longrightarrow$ est noethérienne. Autrement dit, le processus d'élimination des coupures termine toujours.

Théorème 2 *(Confluence)*
La relation $\longrightarrow$ est confluente: Si $\pi_0 \longrightarrow^* \pi_1$ et $\pi_0 \longrightarrow^* \pi_2$, il existe π_3 telle que $\pi_1 \longrightarrow^* \pi_3$ et $\pi_2 \longrightarrow^* \pi_3$.



Le *théorème de normalisation* exprime que toute déduction π peut se réduire en une déduction sans coupure (avec même conclusion et mêmes hypothèses), appelée *forme normale* de π. La démonstration utilise une récurrence sur les déductions et sur les formules.

Le *théorème de confluence* assure l'*unicité* de cette forme normale. En effet, si π a deux formes normales ν_1 et ν_2, il existe ν_3 telle que $\nu_1 \longrightarrow^* \nu_3$ et $\nu_2 \longrightarrow^* \nu_3$. Comme ν_1 et ν_2 sont sans coupures, on a $\nu_1 = \nu_3 = \nu_2$. Le *théorème de confluence* se montre par récurrence sur les déductions.

Le corollaire 1 et le *théorème de normalisation* ont pour conséquences:

[3] Ce n'est pas vrai pour les éliminations de $\bot$ et $\vee$. C'est pourquoi on doit introduire la notion de *coupure commutative*.

Corollaire 2 *(Cohérence de la Déduction Naturelle)*
Il n'y a pas de déduction de $\bot$ sans hypothèse .

Corollaire 3 *D'une déduction de $A \vee B$ sans hypothèse, on peut extraire une déduction de A ou une déduction de B.*

L'hypothèse que la déduction soit ***sans hypothèse*** est évidemment cruciale.

On trouvera plus d'informations sur le ***théorème de normalisation*** dans [Girard87a], par exemple.

1.2 λ-Calcul Typé et Combinateurs Catégoriques

1.2.1 λ-calcul typé

Pour décrire rigoureusement le mécanisme de ***décharge*** d'hypothèses, on donne une nouvelle présentation de la *Déduction Naturelle.*

Les hypothèses sont maintenant ***déclarées*** avec un ***nom***, et la liste des hypothèses est appelée ***environnement*** de la déduction. Toutes les hypothèses de l'environnement sont accessibles[4] mais seule la ***dernière*** hypothèse peut être déchargée.

Une déduction est ***représentée*** par un terme: On écrit $x_1 : A_1, \ldots, x_n : A_n \vdash \mu : B$ lorsque μ est un terme représentant une déduction de B dans l'***environnement*** $x_1 : A_1, \ldots, x_n : A_n$ (les $x_1, \ldots, x_n$ sont les ***noms*** des hypothèses ou ***variables*** de l'environnement).

Pour les termes, on utilise la syntaxe suivante:

- Une variable x est un terme.
- Si μ est un terme, $\mathbf{fst}\ \mu$ et $\mathbf{snd}\ \mu$ sont des termes (projections).
- Si μ et ν sont des termes, (μ, ν) est un terme (couple).
- Si μ et ν sont des termes, $\mu\,\nu$ est un terme (application).
- Si x est une variable et μ un terme, $\mathbf{fun}\ x \rightarrow \mu$ est un terme (abstraction).

Les règles de la *Déduction Naturelle* s'expriment facilement dans ce cadre[5]. Pour simplifier, on se limite aux connecteurs $\wedge$ et $\Rightarrow$ (la disjonction est traitée dans la section 1.5):

(Γ, Δ désignent des environnements, x, y des variables et μ, ν des termes.)

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash x : A}{\Gamma, y : B \vdash x : A}$$

$$\frac{\Gamma \vdash \mu : A \wedge B}{\Gamma \vdash \mathbf{fst}\ \mu : A} \qquad \frac{\Gamma \vdash \mu : A \wedge B}{\Gamma \vdash \mathbf{snd}\ \mu : B} \qquad \frac{\Gamma \vdash \mu : A \quad \Gamma \vdash \nu : B}{\Gamma \vdash (\mu, \nu) : A \wedge B}$$

[4] Si, dans l'environnement, plusieurs hypothèses portent le même nom, la dernière hypothèse portant ce nom *cache* les autres, mais on peut toujours éviter cette situation en ***renommant*** les hypothèses.

[5] Ne pas confondre cette ***présentation*** de la ***Déduction Naturelle*** avec le ***Calcul des Séquents Intuitioniste*** (annexe A).

$$\frac{\Gamma \vdash \mu : A \Rightarrow B \quad \Gamma \vdash \nu : A}{\Gamma \vdash \mu\,\nu : B} \qquad \frac{\Gamma, x : A \vdash \mu : B}{\Gamma \vdash \mathbf{fun}\, x \rightarrow \mu : A \Rightarrow B}$$

Ainsi, notre déduction de $B \Rightarrow (A \wedge B)$ à partir de A devient:

$$\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}}{x : A, y : B \vdash x : A} \quad x : A, y : B \vdash y : B}{x : A, y : B \vdash (x, y) : A \wedge B}}{x : A \vdash \mathbf{fun}\, y \rightarrow (x, y) : B \Rightarrow (A \wedge B)}$$

Les déductions apparaissent ainsi comme des *programmes*, et les formules propositionnelles comme des *types* (A ∧ B étant le produit cartésien, et $A \Rightarrow B$ le type des fonctions de A dans B) dans un petit langage fonctionnel: le *λ-Calcul Typé avec paires*[6].

On compare souvent l'*élimination des coupures* avec l'*exécution* d'un programme. Mais l'*élimination des coupures* est un *mécanisme de reécriture*, alors que l'exécution est un *mécanisme d'évaluation* (sections 1.3 et 1.4).

1.2.2 Combinateurs Catégoriques

La *Déduction Naturelle* (comme le *Calcul des Séquents Intuitioniste*) manipule des jugements de la forme $A_1, \ldots, A_n \vdash B$ (sous les hypothèses $A_1, \ldots, A_n$, on montre B). Il est tentant de remplacer l'environnement $A_1, \ldots, A_n$ par la conjonction de $A_1 \wedge \ldots \wedge A_n$ pour se limiter à des séquents de la forme $A \vdash B$.

On obtient un système dont les preuves sont des *Combinateurs Catégoriques*. Ici, on abandonne le symbole de la *Théorie de la Démonstration* (⊢) pour adopter celui de la *Théorie des Catégories* (→):

$$\frac{}{\mathbf{Id} : A \rightarrow A}\,(\text{identité}) \qquad \frac{\varphi : A \rightarrow B \quad \psi : B \rightarrow C}{\psi \circ \varphi : A \rightarrow C}\,(\text{composition})$$

$$\frac{}{\mathbf{Fst} : A \wedge B \rightarrow A} \qquad \frac{}{\mathbf{Snd} : A \wedge B \rightarrow B} \qquad \frac{\varphi : A \rightarrow B \quad \psi : A \rightarrow C}{\langle \varphi, \psi \rangle : A \rightarrow B \wedge C}$$

$$\frac{}{\mathbf{App} : (A \Rightarrow B) \wedge A \rightarrow B} \qquad \frac{\varphi : A \wedge B \rightarrow C}{\mathbf{Cur}(\varphi) : A \rightarrow B \Rightarrow C}$$

Ces combinateurs sont les *constructeurs* d'une *théorie équationnelle* dont les modèles sont les *Catégories Cartésiennes Fermées* (voir par exemple [Maclane,Huet,Curien86,LamSco]):

$$\frac{\varphi : A \rightarrow B}{\mathbf{Id} \circ \varphi = \varphi : A \rightarrow B \quad (Idl)} \qquad \frac{\varphi : A \rightarrow B}{\varphi \circ \mathbf{Id} = \varphi : A \rightarrow B \quad (Idr)}$$

$$\frac{\varphi : A \rightarrow B \quad \psi : B \rightarrow C \quad \chi : C \rightarrow D}{(\chi \circ \psi) \circ \varphi = \chi \circ (\psi \circ \varphi) : A \rightarrow D \quad (Ass)}$$

[6] Pour l'abstraction, nous utilisons la notation $\mathbf{fun}\, x \rightarrow \mu$ (empruntée à CAML) plutôt que $\lambda x.\mu$.

$$\frac{\varphi : A \to B \quad \psi : A \to C}{\mathbf{Fst} \circ \langle \varphi, \psi \rangle = \varphi : A \to B \quad (Fst)} \qquad \frac{\varphi : A \to B \quad \psi : A \to C}{\mathbf{Snd} \circ \langle \varphi, \psi \rangle = \psi : A \to C \quad (Snd)}$$

$$\frac{\chi : A \to B \quad \varphi : B \to C \quad \psi : B \to D}{\langle \varphi, \psi \rangle \circ \chi = \langle \varphi \circ \chi, \psi \circ \chi \rangle : A \to C \wedge D \quad (DPair)}$$

$$\frac{}{\langle \mathbf{Fst}, \mathbf{Snd} \rangle = \mathbf{Id} : A \wedge B \to A \wedge B \quad (UPair)}$$

$$\frac{\varphi : A \to B \quad \psi : A \wedge B \to C}{\mathbf{App} \circ \langle \mathbf{Cur}(\psi), \varphi \rangle = \psi \circ \langle \mathbf{Id}, \varphi \rangle : A \to C \quad (App)}$$

$$\frac{\varphi : A \to B \quad \psi : B \wedge C \to D}{\mathbf{Cur}(\psi) \circ \varphi = \mathbf{Cur}(\psi \circ \langle \varphi \circ \mathbf{Fst}, \mathbf{Snd} \rangle) : A \to C \Rightarrow D \quad (DCur)}$$

$$\frac{}{\mathbf{Cur}(\mathbf{App}) = \mathbf{Id} : A \Rightarrow B \to A \Rightarrow B \quad (UCur)}$$

L'équation suivante est conséquence de (Idr), (Ass), (Fst), (Snd), $(DPair)$, (App) et (Cur):

$$\frac{\varphi : A \to B \quad \psi : A \to C \quad \chi : B \wedge C \to D}{\mathbf{App} \circ \langle \mathbf{Cur}(\chi) \circ \varphi, \psi \rangle = \chi \circ \langle \varphi, \psi \rangle : A \to D \quad (App')}$$

Si $\mathbf{C}$ est une catégorie, on peut construire la *catégorie cartésienne fermée* $\mathcal{L}(\mathbf{C})$ *librement* engendrée par $\mathbf{C}$ et le foncteur canonique $J : \mathbf{C} \to \mathcal{L}(\mathbf{C})$.

$\mathcal{L}(\mathbf{C})$ est une catégorie dont les objets sont des formules propositionnelles (construites à partir des objets de $\mathbf{C}$), et les morphismes des classes d'équivalence de combinateurs typés (construits à partir des morphismes de $\mathbf{C}$).

C'est en quelque sorte la *théorie* construite sur un ensemble d'axiomes très rudimentaires (les axiomes sont de la forme $A \to B$, où A, B sont des formules atomiques), ou encore l'*extension fonctionnelle* d'un langage très primitif.

1.2.3 Equivalence des deux formalismes

On vérifie:

Théorème 3 *Les deux formalismes (λ-Calcul Typé et Combinateurs Catégoriques) sont équivalents:*

- *A tout combinateur $\varphi : A \to B$, on associe un terme $x : A \vdash \mu : B$.*
- *A tout terme $x_1 : A_1, \ldots, x_n : A_n \vdash \mu : B$, on associe un combinateur $\varphi : (\cdots((K \wedge A_1) \wedge A_2)\ldots) \wedge A_n \to B$, où K est une formule quelconque (pas nécessairement $\top$).*

Notez que les deux traductions ne sont pas inverses l'une de l'autre. La traduction des termes en combinateurs sera utilisée pour la compilation des langages fonctionnels (section 1.4):

(Si Γ est l'environnement $x_1 : A_1, \ldots, x_n : A_n$, on note $\hat{\Gamma}$ la formule $(\cdots((K \wedge A_1) \wedge A_2)\ldots) \wedge A_n$.)

$$\frac{}{\Gamma, x : A \vdash x : A \mapsto \mathbf{Snd} : \hat{\Gamma} \wedge A \to A} \qquad \frac{\Gamma \vdash x : A \mapsto \varphi : \hat{\Gamma} \to A}{\Gamma, y : B \vdash x : A \mapsto \varphi \circ \mathbf{Fst} : \hat{\Gamma} \wedge B \to A}$$

$$\frac{\Gamma \vdash \mu : A \wedge B \mapsto \varphi : \hat{\Gamma} \to A \wedge B}{\Gamma \vdash \mathbf{fst}\, \mu : A \mapsto \mathbf{Fst} \circ \varphi : \hat{\Gamma} \to A} \qquad \frac{\Gamma \vdash \mu : A \wedge B \mapsto \varphi : \hat{\Gamma} \to A \wedge B}{\Gamma \vdash \mathbf{snd}\, \mu : B \mapsto \mathbf{Snd} \circ \varphi : \hat{\Gamma} \to B}$$

$$\frac{\Gamma \vdash \mu : A \mapsto \varphi : \hat{\Gamma} \to A \qquad \Gamma \vdash \nu : B \mapsto \psi : \hat{\Gamma} \to B}{\Gamma \vdash (\mu, \nu) : A \wedge B \mapsto \langle \varphi, \psi \rangle : \hat{\Gamma} \to A \wedge B}$$

$$\frac{\Gamma \vdash \mu : A \Rightarrow B \mapsto \varphi : \hat{\Gamma} \to A \Rightarrow B \qquad \Gamma \vdash \nu : A \mapsto \psi : \hat{\Gamma} \to A}{\Gamma \vdash \mu\nu : B \mapsto \mathbf{App} \circ \langle \varphi, \psi \rangle : \hat{\Gamma} \to B}$$

$$\frac{\Gamma, x : A \vdash \mu : B \mapsto \varphi : \hat{\Gamma} \wedge A \to B}{\Gamma \vdash \mathbf{fun}\, x \to \mu : A \Rightarrow B \mapsto \mathbf{Cur}(\varphi) : \hat{\Gamma} \to A \Rightarrow B}$$

1.2.4 Déclarations locales

Dans la section 1.1, nous avons utilisé la *substitution*:

$$\begin{array}{ccc} & A & \\ \vdots & \vdots & \\ A & B & \end{array} \quad \longrightarrow \quad \begin{array}{c} \vdots \\ A \\ \vdots \\ B \end{array}$$

Bien qu'il ne s'agisse pas d'une règle de la *Déduction Naturelle*, on peut l'ajouter au *λ-Calcul Typé*:

$$\frac{\Gamma \vdash \mu : A \quad \Gamma, x : A \vdash \nu : B}{\Gamma \vdash \mathbf{let}\, x = \mu\, \mathbf{in}\, \nu : B}$$

Le terme $\mathbf{let}\, x = \mu\, \mathbf{in}\, \nu$ peut être considéré comme une abréviation de $(\mathbf{fun}\, x \to \nu)\, \mu$, mais on a la traduction directe:

$$\frac{\Gamma \vdash \mu : A \mapsto \varphi : \hat{\Gamma} \to A \qquad \Gamma, x : A \vdash \nu : B \mapsto \psi : \hat{\Gamma} \wedge A \to B}{\Gamma \vdash \mathbf{let}\, x = \mu\, \mathbf{in}\, \nu : B \mapsto \psi \circ \langle \mathbf{Id}, \varphi \rangle : \hat{\Gamma} \to B}$$

1.3 Théorie de l'Evaluation

1.3.1 Le Théorème de Cohérence Hiérarchique

Dans le formalisme catégorique, il n'y a pas de *théorème de normalisation.* Il est impossible d'*éliminer* la *composition* comme on élimine la règle de coupure dans le *Calcul des Séquents* (annexe A).

Plus fondamentalement, il n'y a pas de *Propriété de Sous-formule.* Par exemple, pour construire un combinateur $\varphi : A \to (B \Rightarrow A)$, on a besoin de la conjonction[7]:

$$\frac{\overline{\mathbf{Fst} : A \wedge B \to A}}{\mathbf{Cur(Fst)} : A \to (B \Rightarrow A)}$$

On a cependant une *propriété de sous-formule* pour les combinateurs $\varphi : A \to B$ lorsque A et B sont des formules atomiques. Plus précisément:

Théorème 4 *(Cohérence Hiérarchique)*
Pour toute catégorie $\mathbf{C}$*, le foncteur canonique* $J : \mathbf{C} \to \mathcal{L}(\mathbf{C})$ *est plein et fidèle*[8]*. Autrement dit, si* A *et* B *sont des objets de* $\mathbf{C}$*,* J *induit une bijection:*

$$\mathbf{Hom}_{\mathbf{C}}(A, B) \xrightarrow{\sim} \mathbf{Hom}_{\mathcal{L}(\mathbf{C})}(A, B)$$

Ainsi, dans $\mathcal{L}(\mathbf{C})$, on ne *crée* pas de morphisme entre objets de $\mathbf{C}$, ni d'équation entre morphismes de $\mathbf{C}$. C'est une propriété d'*extension conservatrice*[9].

Nous donnons en annexe C une démonstration de ce théorème avec des techniques "catégoriques" (plongement de *Yoneda, glueing* ...).

Dans un premier temps, on se limitera au cas où $\mathbf{C}$ est la catégorie terminale $\mathbf{T}$ (réduite à un seul objet K avec son identité). La *fidélité* de J est alors évidente.

Théorème 5 *(Cohérence Hiérarchique pour* $\mathbf{T}$*)*
$\mathbf{Hom}_{\mathcal{L}(\mathbf{T})}(K, K)$ *est réduit à* $\{id_K\}$*. Autrement dit, tout combinateur* $\varphi : K \to K$ *est congru à* $\mathbf{Id}$ *(modulo les équations catégoriques).*

1.3.2 Sémantique fonctionnelle

On donne ici une preuve "élémentaire" du théorème 5, directement issue de la preuve "catégorique" (annexe C). Ce n'est pas tellement le résultat qui nous intéresse, mais la démonstration, dont on va extraire le *mécanisme d'évaluation.*

On se limite donc au calcul propositionnel construit sur une seule formule atomique K, et on note $\mathbf{Comb}(A, B)$ l'ensemble des combinateurs typés[10] $\varphi : A \to B$.

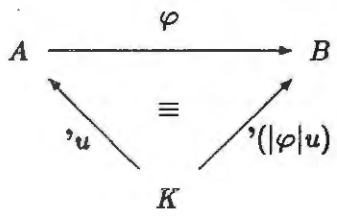
Par récurrence, on construit, pour toute formule A, un ensemble de *valeurs abstraites* noté $\mathbf{Val}(A)$ et une fonction $' : \mathbf{Val}(A) \to \mathbf{Comb}(K, A)$. $\mathbf{Val}(A)$ est une "interprétation sémantique", $\mathbf{Comb}(K, A)$ une "interprétation syntaxique", et $'$ est la "colle" entre la sémantique et la syntaxe.

[7] Dans le formalisme catégorique, l'implication est définie *à partir de la conjonction.* Dans les autres formalismes (*Déduction Naturelle* et *Calcul des Séquents*), la conjonction apparaît implicitement dans l'environnement.
[8] *Surjectif* et *injectif* sur les morphismes.
[9] Des résultats similaires ont été énoncés dans le cadre du *Calcul des Séquents* [Szabo] et du λ-*Calcul typé* [BreaCoq].
[10] On distingue les *combinateurs typés* des *morphismes* (qui sont des classes d'équivalence de combinateurs).

De même, on construit pour tout combinateur typé $\varphi : A \to B$ une fonction $|\varphi| : \mathbf{Val}(A) \to \mathbf{Val}(B)$ (sémantique de φ) *cohérente* avec ', c'est à dire vérifiant:

$$'(|\varphi|\, u) \equiv \varphi \circ 'u \quad (quote)$$

($\equiv$ est la *congruence* engendrée par les axiomes catégoriques)



Pour construire les $\mathbf{Val}(A)$, on se laisse guider par la nécessité de définir '.

Puisqu'on veut montrer que $\mathbf{Id}$ est le seul morphisme $K \to K$, il est naturel de poser:

- $\mathbf{Val}(K) = \{*\}$ (singleton) et $'* = \mathbf{Id} : K \to K$.

La conjonction ne pose aucune difficulté:

- $\mathbf{Val}(A \wedge B) = \mathbf{Val}(A) \times \mathbf{Val}(B)$ et $'(u, v) = \langle 'u, 'v\rangle : K \to A \wedge B$.

Pour $\mathbf{Val}(A \Rightarrow B)$ il est tentant de considérer l'ensemble des fonctions de $\mathbf{Val}(A)$ dans $\mathbf{Val}(B)$. Mais il faut définir ': Pour toute valeur de $A \Rightarrow B$, on veut un combinateur $K \to A \Rightarrow B$. Un $\mathbf{Cur}(\varphi)$, avec $\varphi : K \wedge A \to B$ ferait l'affaire. D'où la définition:

- $\mathbf{Val}(A \Rightarrow B) = \{(\varphi, f);\ \varphi \in \mathbf{Comb}(K \wedge A, B),\ f : \mathbf{Val}(A) \to \mathbf{Val}(B),\ '(f\ u) \equiv \varphi \circ \langle \mathbf{Id}, 'u\rangle\}$ et $'(\varphi, f) = \mathbf{Cur}(\varphi) : K \to A \Rightarrow B$.

Une valeur de $A \Rightarrow B$ est donc un combinateur $K \wedge A \to B$ (composante *syntaxique*) avec une fonction de $\mathbf{Val}(A)$ dans $\mathbf{Val}(B)$ (composante *sémantique*) vérifiant une condition de cohérence (version *interne* de (*quote*)).

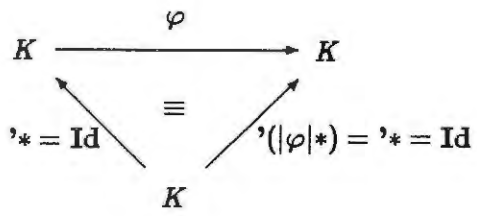
On définit inductivement la *sémantique* des combinateurs:

- $|\mathbf{Id}|\, u = u$. On vérifie $'(|\mathbf{Id}|\, u) \equiv \mathbf{Id} \circ 'u$ par (Idl).
- $|\varphi \circ \psi|\, u = |\varphi|\, (|\psi|\, u)$. On vérifie $'(|\varphi \circ \psi|\, u) \equiv (\varphi \circ \psi) \circ 'u$ par (Ass).
- $|\mathbf{Fst}|\, (u, v) = u$. On vérifie $'(|\mathbf{Fst}|\, (u, v)) \equiv \mathbf{Fst} \circ '(u, v)$ par (Fst).
- $|\mathbf{Snd}|\, (u, v) = v$ ...
- $|\langle \varphi, \psi\rangle|\, u = (|\varphi|\, u, |\psi|\, u)$. On vérifie $'(|\langle \varphi, \psi\rangle|\, u) \equiv \langle \varphi, \psi\rangle \circ 'u$ par $(DPair)$.
- $|\mathbf{App}|\, ((\varphi, f), u) = f\ u$. On vérifie $'(|\mathbf{App}|\, ((\varphi, f), u)) \equiv \mathbf{App} \circ '((\varphi, f), u)$ par (App) et par la condition de cohérence sur les valeurs de $A \Rightarrow B$.
- $|\mathbf{Cur}(\varphi)|\, u = (\varphi \circ \langle 'u \circ \mathbf{Fst}, \mathbf{Snd}\rangle, f\ u)$ où $f\ u\ v = |\varphi|\, (u, v)$. On vérifie d'abord que c'est une valeur (condition de cohérence pour les valeurs de $A \Rightarrow B$) par (Ass), (Idr), (Fst), (Snd) et $(DPair)$, puis $'(|\mathbf{Cur}(\varphi)|\, u) \equiv \mathbf{Cur}(\varphi) \circ 'u$ par (Cur).

Achevons la preuve du théorème 5:

Soit $\varphi \in \mathbf{Comb}(K, K)$. Comme $\mathbf{Val}(K) = \{*\}$, on a $|\varphi| * = *$, donc, par (Idr) et $(quote)$:

$$\varphi \equiv \varphi \circ \mathbf{Id} = \varphi \circ {}'* \equiv {}'(|\varphi| *) = {}'* = \mathbf{Id}$$



Dans cette preuve, on n'a pas utilisé les équations $(UPair)$ et $(UCur)$. Il est possible de réduire encore la liste des axiomes nécessaires.

1.3.3 Sémantique relationnelle

La preuve du théorème 5 introduit la notion de *valeur abstraite* et de *sémantique des combinateurs* qui suggèrent un mécanisme d'évaluation. Mais les *valeurs abstraites* de $A \Rightarrow B$, avec leur composante fonctionnelle, ne semblent guère "mécanisables"[11]. De plus, la sémantique de $\mathbf{Cur}(\varphi)$ est assez compliquée.

Ici intervient une remarque de bon sens: Pourquoi évaluer $|\mathbf{Cur}(\varphi)|\ u$? Un jour, on devra lui appliquer le combinateur **App**, ce qui va tout simplifier. Une valeur de $A \Rightarrow B$ pourrait être un combinateur $\varphi \in \mathbf{Comb}(X \wedge A, B)$ avec une valeur $u \in \mathbf{Val}(X)$, où X est une formule quelconque.

Mais alors, pour construire $\mathbf{Val}(A \Rightarrow B)$, on utilise tous les $\mathbf{Val}(X)$, si bien que la définition est circulaire!

Pour résoudre ce problème de circularité, on remplace la *sémantique fonctionnelle* par une sémantique *relationnelle*[12].

On a une nouvelle notion de *valeur*:

- $*$ est une valeur.
- Si u et v sont des valeurs, (u, v) est une valeur (paire).
- Si φ est un combinateur et u est une valeur, $(\varphi \cdot u)$ est une valeur (fermeture).

On définit inductivement une *relation de typage* $u \circ\!\!- A$, et à toute *valeur typée* $u \circ\!\!- A$, on associe ${}'u : K \to A$:

$$\frac{}{* \circ\!\!- K \qquad {}'* = \mathbf{Id} : K \to K}$$

$$\frac{u \circ\!\!- A \qquad v \circ\!\!- B}{(u, v) \circ\!\!- A \wedge B \qquad {}'(u, v) = \langle {}'u, {}'v \rangle : K \to A \wedge B}$$

[11] Dans le système CDS de Berry et Curien par exemple, les *algorithmes* sont représentés comme des données, mais ce sont en général des objets infinis.

[12] Ce principe général a déjà été mis en évidence par G. Plotkin.

$$\frac{\varphi : A \wedge B \to C \qquad u \circ\!\!: A}{(\varphi \cdot u) \circ\!\!: B \Rightarrow C \qquad '(\varphi \cdot u) = \mathbf{Cur}(\varphi) \circ 'u : K \to B \Rightarrow C}$$

On définit de même une *relation d'évaluation* (ternaire) $\varphi\ u \triangleright v$ pour $\varphi : A \to B$, $u \circ\!\!: A$ et $v \circ\!\!: B$:

$$\frac{u \circ\!\!: A}{\mathbf{Id}\ u \triangleright u} \qquad \frac{\varphi : A \to B \quad \psi : B \to C \quad u \circ\!\!: A \quad \varphi\ u \triangleright v \circ\!\!: B \quad \psi\ v \triangleright w \circ\!\!: C}{(\psi \circ \varphi)\ u \triangleright w}$$

$$\frac{u \circ\!\!: A \quad v \circ\!\!: B}{\mathbf{Fst}\ (u, v) \triangleright u} \qquad \frac{u \circ\!\!: A \quad v \circ\!\!: B}{\mathbf{Snd}\ (u, v) \triangleright v}$$

$$\frac{\varphi : A \to B \quad \psi : A \to C \quad u \circ\!\!: A \quad \varphi\ u \triangleright v \circ\!\!: B \quad \psi\ u \triangleright w \circ\!\!: C}{\langle \varphi, \psi \rangle\ u \triangleright (v, w)}$$

$$\frac{\varphi : A \wedge B \to C \quad u \circ\!\!: A \quad v \circ\!\!: B \quad \varphi\ (u, v) \triangleright w \circ\!\!: C}{\mathbf{App}\ ((\varphi \cdot u), v) \triangleright w} \qquad \frac{\varphi : A \wedge B \to C \quad u \circ\!\!: A}{\mathbf{Cur}(\varphi)\ u \triangleright (\varphi \cdot u)}$$

Evaluer un combinateur $\varphi : A \to B$ sur une valeur $u \circ\!\!: A$, c'est trouver $v \circ\!\!: B$ telle que $\varphi\ u \triangleright v$. Le système de règles d'évaluation est *non ambigu*, d'où:

Proposition 2 *(unicité du résultat de l'évaluation)*
Pour tout $\varphi : A \to B$ et $u \circ\!\!: A$, il existe au plus une valeur $v \circ\!\!: B$ telle que $\varphi\ u \triangleright v$.

D'autre part, en utilisant (Idl), (Ass), (Fst), (Snd), $(DPair)$ et (App'), on vérifie aisément:

Proposition 3 *(cohérence du résultat de l'évaluation)*
Si $\varphi\ u \triangleright v$, alors $'v \equiv \varphi \circ 'u$.

Il reste à montrer:

Théorème 6 *(Convergence)*
Pour tout combinateur $\varphi : A \to B$ et pour toute valeur $u \circ\!\!: A$, il existe une valeur (unique) $v \circ\!\!: B$ telle que $\varphi\ u \triangleright v$.

Démonstration: Pour toute formule A, on construit un ensemble $\mathbf{Conv}(A)$ (*convergence* de A) de valeurs $u \circ\!\!: A$:

- $\mathbf{Conv}(K) = \{*\}$
- $\mathbf{Conv}(A \wedge B) = \{(u, v);\ u \in \mathbf{Conv}(A),\ v \in \mathbf{Conv}(B)\}$
- $\mathbf{Conv}(A \Rightarrow B) = \{(\varphi \cdot u); \varphi : X \wedge A \to B, u \circ\!\!: X, (\forall v \in \mathbf{Conv}(A))(\exists w \in \mathbf{Conv}(B))\varphi\ (u, v) \triangleright w\}$

On vérifie (par récurrence sur les combinateurs, puis sur les valeurs):

Lemme 1 *Pour tout combinateur $\varphi : A \to B$ et pour toute valeur $u \in \mathbf{Conv}(A)$, il existe une valeur $v \in \mathbf{Conv}(B)$ telle que $\varphi\ u \triangleright v$.*

Lemme 2 *Pour tout A, $\mathbf{Conv}(A) = \{u;\ u \circ\!\!: A\}$.*

Ainsi, la circularité a été contournée, et le théorème 6 est démontré. □

La proposition 3 et le *Théorème de Convergence* permettent de raffiner le théorème 5:

Corollaire 4 *Pour tout combinateur* $\varphi : K \to K$, *on a* $\varphi \circ \mathbf{Id} \equiv \mathbf{Id}$ *modulo* (Idl), (Ass), (Fst), (Snd), $(DPair)$ *et* (App') [13].

La proposition 2 et le *Théorème de Convergence* rétablissent le caractère *fonctionnel* de l'évaluation. Dans la suite, on écrira $v = \varphi\, u$ plutôt que $\varphi\, u \triangleright v$.

1.4 La Machine Catégorique

La *relation d'évaluation* de la section 1.3 peut être considérée comme une *machine abstraite* dont le code est un *combinateur catégorique.*

A quelques détails près, notre présentation de la *Machine Catégorique Abstraite* est la même que [CouCurMau]. L'originalité de notre approche réside dans la justification mathématique qui précéde (section 1.3).

1.4.1 Description de la Machine

La *Machine Catégorique Abstraite* possède un *registre* (l'*environnement*) et une *pile.*

Le code est une liste d'instructions (contigües en mémoire) exécutées en séquence. Ainsi, le combinateur **Id** donne le code [] (liste vide), et la composition $\varphi \circ \psi$ correspond à la *concaténation* de ψ avec φ.

Les combinateurs **Fst** et **Snd** sont les instructions d'accès aux composantes d'une paire.

Le combinateur $\langle \varphi, \psi \rangle$ n'est pas une instruction primitive: Pour l'exécuter sur une valeur u, il faut:

- calculer $\varphi\, u$
- calculer $\psi\, u$ (cela suppose que l'on ait conservé u)
- construire une paire avec $\varphi\, u$ et $\psi\, u$ (cela suppose que l'on ait conservé $\varphi\, u$).

En utilisant la pile pour conserver les valeurs, on peut:

- empiler u
- exécuter φ
- échanger le résultat avec le sommet de la pile
- exécuter ψ
- construire une paire avec le sommet de la pile et le résultat.

Ainsi, chacun des trois symboles $\langle\ ,\ \rangle$ est une instruction primitive (**Push**, **Swap** et **Cons**).

L'instruction $\mathbf{Cur}(\varphi)$ construit une fermeture avec le code φ.

L'instruction **App** attend une valeur de la forme $((\varphi \cdot u), v)$. Elle fabrique la paire (u, v) et *appelle* le code φ. L'appel de φ consiste à:

[13] En fait, les équations catégoriques sont toujours utilisées de gauche à droite: $\varphi \circ \mathbf{Id}$ se *reécrit* en **Id**.

- empiler l'adresse de l'instruction suivante (*adresse de retour*)
- se brancher sur φ.

Cela suppose que le code φ se termine par l'instruction **Return** (qui *dépile* l'*adresse de retour*).

Notez que la pile a deux fonctions distinctes:

- conserver des valeurs (**Push**, **Swap** et **Cons**)
- conserver des *adresses de retour* (**App** et **Return**)

Résumons le fonctionnement de la *Machine Catégorique Abstraite*:

($x_0 :: [x_1; \cdots; x_n]$ dénote la liste $[x_0; x_1; \cdots; x_n]$)

La Machine Catégorique					
Avant			Après		
code	environnement	pile	code	environnement	pile
Fst $:: C$	(u, v)	S	C	u	S
Snd $:: C$	(u, v)	S	C	v	S
Push $:: C$	u	S	C	u	$u :: S$
Swap $:: C$	u	$v :: S$	C	v	$u :: S$
Cons $:: C$	u	$v :: S$	C	(v, u)	S
App $:: C$	$((C' \cdot u), v)$	S	C'	(u, v)	$C :: S$
Cur(C') $:: C$	u	S	C	$(C' \cdot u)$	S
Return $:: C$	u	$C' :: S$	C'	u	S

1.4.2 Supplément à la Machine Catégorique

Le *valeurs* introduites dans la section 1.3 ne sont pas très parlantes! Pour utiliser notre *Machine* de façon plus concrète, nous allons introduire un *type de base* (les entiers) avec des *constantes* et des *opérations de base*.

Plutôt qu'une instruction pour chaque constante entière, on utilise une instruction générique **Quote**$(-)$[14]:

Avant			Après		
code	environnement	pile	code	environnement	pile
Quote$(u) :: C$	v	S	C	u	S

L'instruction **Quote**$(-)$ permet d'introduire des constantes de base, mais aussi des *variables globales* de n'importe quel type qui ont été calculées auparavant (par exemple des valeurs fonctionnelles).

On a une instruction primitive pour chaque opération, par exemple:

Avant			Après		
code	environnement	pile	code	environnement	pile
$+ :: C$	(u, v)	S	C	$u + v$	S
$\times :: C$	(u, v)	S	C	$u \times v$	S

[14]Notez la similitude entre l'instruction **Quote**(u) et le combinateur 'u de la section 1.3.

1.4.3 Compilation des λ-termes.

La compilation est donnée par la *traduction des λ-termes en combinateurs catégoriques* (voir section 1.2).

Si μ est un terme construit dans l'environnement $x_1 : A_1, \ldots, x_n : A_n$, on représente la liste des variables par l'expression $X = (\cdots((\_, x_1), x_2)\ldots, x_n)$, et on note $[\![\mu]\!]_X$ le *code* de μ relativement à X (si on exécute $[\![\mu]\!]_X$ sur une valeur $(\cdots((v, v_1), v_2)\ldots, v_n)$, on obtient la valeur de $\mu[v_1/x_1, \ldots, v_n/x_n]$):

- $[\![x]\!]_{(X,x)} = [\mathbf{Snd}]$
- $[\![x]\!]_{(X,y)} = [\mathbf{Fst}] \mathbin{@} [\![x]\!]_X$
- $[\![\mathbf{fst}\ \mu]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Fst}]$
- $[\![\mathbf{snd}\ \mu]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Snd}]$
- $[\![(\mu, \nu)]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![\nu]\!]_X \mathbin{@} [\mathbf{Cons}]$
- $[\![\mu\,\nu]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![\nu]\!]_X \mathbin{@} [\mathbf{Cons}; \mathbf{App}]$
- $[\![\mathbf{fun}\ x \to \mu]\!]_X = [\mathbf{Cur}([\![\mu]\!]_{(X,x)} \mathbin{@} [\mathbf{Return}])]$
- $[\![\mathbf{let}\ x = \mu\ \mathbf{in}\ \nu]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Cons}] \mathbin{@} [\![\nu]\!]_{(X,x)}$
- $[\![c]\!]_X = [\mathbf{Quote}(u)]$ (c est une constante ou une variable globale de valeur u)
- $[\![\mu + \nu]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![\nu]\!]_X \mathbin{@} [\mathbf{Cons}; +]$
- $[\![\mu * \nu]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![\nu]\!]_X \mathbin{@} [\mathbf{Cons}; \times]$

On constate que la séquence [**Push**; **Swap**] est équivalente à [**Push**]. C'est pourquoi on supprime l'instruction **Swap** dans la compilation de **let** $x = \mu$ **in** ν. On verra d'autres *optimisations* dans la section 1.7.

Voici un exemple emprunté à [CouCurMauSu]:

$[\![\mathbf{let}\ f = \mathbf{fun}\ x \to x * x\ \mathbf{in}\ \mathbf{let}\ x = 3\ \mathbf{in}\ f(f\ x)]\!]_{\_} = [\mathbf{Push}; \mathbf{Cur}(C); \mathbf{Cons}; \mathbf{Push}; \mathbf{Quote}(3); \mathbf{Cons}; \mathbf{Push}; \mathbf{Fst}; \mathbf{Snd}; \mathbf{Swap}; \mathbf{Push}; \mathbf{Fst}; \mathbf{Snd}; \mathbf{Swap}; \mathbf{Snd}; \mathbf{Cons}; \mathbf{App}; \mathbf{Cons}; \mathbf{App}]$ où

$C = [\![x * x]\!]_{(\_,x)} \mathbin{@} [\mathbf{Return}] = [\mathbf{Push}; \mathbf{Snd}; \mathbf{Swap}; \mathbf{Snd}; \mathbf{Cons}; \times; \mathbf{Return}]$

Le code exécuté (figure 1.1) donne le résultat escompté (81).

1.4.4 Le récupérateur de mémoire

Pour la machine, une *paire* est un *pointeur* vers un *doublet* de cases adjacentes:

instruction	environnement	pile		
Push	$-$			
Cur(C)	$-$	$-$		
Cons	$(C \cdot \_)$	$-$		
Push	$(\_, (C \cdot \_))$			
Quote(3)	$(\_, (C \cdot \_))$	$(\_, (C \cdot \_))$		
Cons	3	$(\_, (C \cdot \_))$		
Push	$((\_, (C \cdot \_)), 3)$			
Fst	$((\_, (C \cdot \_)), 3)$	$((\_, (C \cdot \_)), 3)$		
Snd	$(\_, (C \cdot \_))$	$((\_, (C \cdot \_)), 3)$		
Swap	$(C \cdot \_)$	$((\_, (C \cdot \_)), 3)$		
Push	$((\_, (C \cdot \_)), 3)$	$(C \cdot \_)$		
Fst	$((\_, (C \cdot \_)), 3)$	$((\_, (C \cdot \_)), 3)$	$(C \cdot \_)$	
Snd	$(\_, (C \cdot \_))$	$((\_, (C \cdot \_)), 3)$	$(C \cdot \_)$	
Swap	$(C \cdot \_)$	$((\_, (C \cdot \_)), 3)$	$(C \cdot \_)$	
Snd	$((\_, (C \cdot \_)), 3)$	$(C \cdot \_)$	$(C \cdot \_)$	
Cons	3	$(C \cdot \_)$	$(C \cdot \_)$	
App	$((C \cdot \_), 3)$	$(C \cdot \_)$		
Push	$(\_, 3)$	[**Cons**; **App**]	$(C \cdot \_)$	
Snd	$(\_, 3)$	$(\_, 3)$	[**Cons**; **App**]	$(C \cdot \_)$
Swap	3	$(\_, 3)$	[**Cons**; **App**]	$(C \cdot \_)$
Snd	$(\_, 3)$	3	[**Cons**; **App**]	$(C \cdot \_)$
Cons	3	3	[**Cons**; **App**]	$(C \cdot \_)$
×	$(3, 3)$	[**Cons**; **App**]	$(C \cdot \_)$	
Return	9	[**Cons**; **App**]	$(C \cdot \_)$	
Cons	9	$(C \cdot \_)$		
App	$((C \cdot \_), 9)$			
Push	$(\_, 9)$	[]		
Snd	$(\_, 9)$	$(\_, 9)$	[]	
Swap	9	$(\_, 9)$	[]	
Snd	$(\_, 9)$	9	[]	
Cons	9	9	[]	
×	$(9, 9)$	[]		
Return	81	[]		
	81			

Figure 1.1: Exemple d'exécution

On utilise également ce mode de représentation pour les *fermetures*.

Ainsi, tous les objets ont la même taille (une adresse mémoire) et la création d'une *paire* (par les instructions **Cons**, **App**) ou d'une *fermeture* (par l'instruction **Cur**$(-)$) ne nécessite aucune recopie. Mais ces créations *consomment* des *doublets*, et aucune instruction de la *Machine Catégorique* ne peut les *récupérer*, car tout noeud de l'environnement est susceptible d'être *partagé* (à cause de l'instruction **Push**).

D'autre part, certaines instructions (**Fst**, **Snd** et **Quote**$(-)$) "oublient" une partie (ou la totalité) de l'environnement. Là encore, le partage des noeuds interdit toute tentative de récupération!

Lorsqu'il n'y a plus de doublet disponible, on est obligé d'interrompre l'évaluation pour récupérer tous les doublets abandonnés. On peut aussi indiquer le degré de partage de chaque noeud au moyen d'un *compteur de références* (récupération "à la volée"). De toute façon, on exécute un algorithme de *récupération de mémoire* (*garbage collector*) distinct du *code catégorique*[15].

1.5 Conditionnelle et Types Concrets

1.5.1 Le coproduit

La notion catégorique correspondant à la disjonction est le *coproduit* (construction duale du produit cartésien), avec les combinateurs suivants:

$$\frac{}{\mathbf{Inl} : A \to A \vee B} \qquad \frac{}{\mathbf{Inr} : B \to A \vee B} \qquad \frac{x : A \to C \quad y : B \to C}{\mathbf{Case}(x, y) : A \vee B \to C}$$

Le *Théorème de Cohérence Hierarchique* s'étend sans difficulté aux *catégories bicartésiennes fermées*. En particulier, on construit $\mathbf{Val}(A \vee B) = \mathbf{Val}(A) \amalg \mathbf{Val}(B)$ (union disjointe) et:

$$\frac{u \circ A}{(\mathbf{inl}\, u) \circ A \vee B} \qquad \frac{u \circ B}{(\mathbf{inr}\, u) \circ A \vee B}$$

Pour la *Machine Catégorique*, on a les instructions suivantes:

Avant			Après		
code	environnement	pile	code	environnement	pile
$\mathbf{Inl} :: C$	u	S	C	$(\mathbf{inl}\, u)$	S
$\mathbf{Inr} :: C$	u	S	C	$(\mathbf{inr}\, u)$	S
$\mathbf{Case}(C', C'') :: C$	$(\mathbf{inl}\, u)$	S	C'	u	$C :: S$
$\mathbf{Case}(C', C'') :: C$	$(\mathbf{inr}\, u)$	S	C''	u	$C :: S$

Les règles de *Déduction Naturelle* pour la disjonction donnent:

$$\frac{\Gamma \vdash \mu : A}{\Gamma \vdash \mathbf{inl}\, \mu : A \vee B} \qquad \frac{\Gamma \vdash \mu : B}{\Gamma \vdash \mathbf{inr}\, \mu : A \vee B} \qquad \frac{\Gamma \vdash \lambda : A \vee B \quad \Gamma, x : A \vdash \mu : C \quad \Gamma, y : B \vdash \nu : C}{\Gamma \vdash \mathbf{case}\, \lambda\, \mathbf{of}\, (\mathbf{inl}\ x) \to \mu \mid (\mathbf{inl}\ y) \to \nu : C}$$

[15] L'existence du *garbage collector* impose certaines contraintes. Ainsi, l'ensemble des adresses de paires doit être *disjoint* de l'ensemble des atomes (condition qui n'est pas requise par la *Machine Catégorique*, puisque le *contrôle des types* est effectué à la compilation).

Mais la traduction de case λ **of** $x \to \mu \mid y \to \nu$ en *combinateur catégorique* est compliquée[16]. On pourrait la simplifier en utilisant une construction plus générale:

$$\frac{\varphi : A \wedge B \to D \quad \psi : A \wedge C \to D}{\mathbf{Case}(\varphi, \psi) : A \wedge (B \vee C) \to D}$$

C'est la caractérisation *paramétrique* du coproduit (équivalente à la caractérisation usuelle *dans le cas des catégories cartésiennes fermées*).

En fait, nous allons abandonner le coproduit comme *construction primitive*.

Pour l'implantation, en effet, il faut que tous les objets aient la même taille. On représente donc une valeur de $A \vee B$ par une paire (m, u), où m est un *marqueur* (indiquant si on est dans A ou dans B) et u est une valeur de A ou de B (selon la valeur de m). Cette représentation est d'ailleurs conforme à la construction "ensembliste" de l'union disjointe:

$$A \amalg B = \{0\} \times A \cup \{1\} \times B$$

Ainsi, le type *booléen* semble plus *primitif* que le coproduit.

1.5.2 Booléens

Lorsqu'on a un objet final $\top$, le type **Bool** est simplement $\top \vee \top$, mais ici:

- On considère le type **Bool** (et non le coproduit) comme primitif.
- On veut une caractérisation *paramétrique* des booléens.

Les combinateurs catégoriques pour **Bool** sont alors:

$$\frac{}{\mathbf{True} : A \to \mathbf{Bool}} \qquad \frac{}{\mathbf{False} : A \to \mathbf{Bool}} \qquad \frac{\varphi : A \to B \quad \psi : A \to B}{\mathbf{Branch}(\varphi, \psi) : A \wedge \mathbf{Bool} \to B}$$

On a évidemment $\mathbf{Val(Bool)} = \{\mathbf{true}, \mathbf{false}\}$ et:

$$\frac{}{\mathbf{true} : \mathbf{Bool}} \qquad \frac{}{\mathbf{false} : \mathbf{Bool}}$$

Il n'y a qu'une instruction pour la machine catégorique (les valeurs **true** et **false** sont introduites par **Quote(true)** et **Quote(false)**):

Avant			Après		
code	environnement	pile	code	environnement	pile
$\mathbf{Branch}(C', C'') :: C$	$(u, \mathbf{true})$	S	C'	u	$C :: S$
$\mathbf{Branch}(C', C'') :: C$	$(u, \mathbf{false})$	S	C''	u	$C :: S$

Notez la similitude avec $\mathbf{Case}(-,-)$.

On introduit la *conditionnelle* dans le *λ-calcul typé*:

[16] $\mathbf{App} \circ \langle \mathbf{Case}(\mathbf{Cur}(\varphi \circ \langle \mathbf{Snd}, \mathbf{Fst} \rangle), \mathbf{Cur}(\psi \circ \langle \mathbf{Snd}, \mathbf{Fst} \rangle)) \circ \chi, \mathbf{Id} \rangle$, où χ, φ et ψ sont les traductions respectives de λ, μ et ν.

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}} \qquad \frac{\Gamma \vdash \lambda : \mathbf{Bool} \quad \Gamma \vdash \mu : A \quad \Gamma \vdash \nu : A}{\Gamma \vdash \mathbf{if}\ \lambda\ \mathbf{then}\ \mu\ \mathbf{else}\ \nu : A}$$

Cette fois, la traduction ne pose pas de difficulté:

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool} \mapsto \mathbf{True} : \hat{\Gamma} \to \mathbf{Bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool} \mapsto \mathbf{False} : \hat{\Gamma} \to \mathbf{Bool}}$$

$$\frac{\Gamma \vdash \lambda : \mathbf{Bool} \mapsto \chi : \hat{\Gamma} \to \mathbf{Bool} \qquad \Gamma \vdash \mu : A \mapsto \varphi : \hat{\Gamma} \to A \qquad \Gamma \vdash \nu : A \mapsto \psi : \hat{\Gamma} \to A}{\Gamma \vdash \mathbf{if}\ \lambda\ \mathbf{then}\ \mu\ \mathbf{else}\ \nu : A \mapsto \mathbf{Branch}(\varphi, \psi) \circ \langle \mathbf{Id}, \chi \rangle : \hat{\Gamma} \to A}$$

On en déduit la compilation de la conditionnelle:

- $[\![\,\mathbf{true}\,]\!]_X = [\mathbf{Quote(true)}]$
- $[\![\,\mathbf{false}\,]\!]_X = [\mathbf{Quote(false)}]$
- $[\![\,\mathbf{if}\,\lambda\,\mathbf{then}\,\mu\,\mathbf{else}\,\nu\,]\!]_X = [\mathbf{Push}]@[\![\,\lambda\,]\!]_X@[\mathbf{Cons;Branch}([\![\,\mu\,]\!]_X@[\mathbf{Return}], [\![\,\nu\,]\!]_X@[\mathbf{Return}])]$

Comme pour $\mathbf{let}\ x = \mu\ \mathbf{in}\ \nu$, on a remplacé la séquence [**Push;Swap**] par [**Push**].

1.5.3 Types énumérés

De la même façon, on peut considérer n'importe quel *type fini*:

$$\mathbf{Suit} = \mathbf{spade} \mid \mathbf{heart} \mid \mathbf{club} \mid \mathbf{diamond}$$

Avant			Après		
code	environnement	pile	code	environnement	pile
$\mathbf{Branch}_4(C_0, C_1, C_2, C_3) :: C$	$(u, 0)$	S	C_0	u	$C :: S$
$\mathbf{Branch}_4(C_0, C_1, C_2, C_3) :: C$	$(u, 1)$	S	C_1	u	$C :: S$
$\mathbf{Branch}_4(C_0, C_1, C_2, C_3) :: C$	$(u, 2)$	S	C_2	u	$C :: S$
$\mathbf{Branch}_4(C_0, C_1, C_2, C_3) :: C$	$(u, 3)$	S	C_3	u	$C :: S$

- $[\![\,\mathbf{spade}\,]\!]_X = [\mathbf{Quote(0)}]$
- $[\![\,\mathbf{heart}\,]\!]_X = [\mathbf{Quote(1)}]$
- $[\![\,\mathbf{club}\,]\!]_X = [\mathbf{Quote(2)}]$
- $[\![\,\mathbf{diamond}\,]\!]_X = [\mathbf{Quote(3)}]$
- $[\![\,\mathbf{case}\,\lambda\,\mathbf{of\ spade} \to \alpha \mid \mathbf{heart} \to \beta \mid \mathbf{club} \to \gamma \mid \mathbf{diamond} \to \delta\,]\!]_X = [\mathbf{Push};] @ [\![\,\mu\,]\!]_X @ [\mathbf{Cons;Branch}_4([\![\,\alpha\,]\!]_X @ [\mathbf{Return}], [\![\,\beta\,]\!]_X @ [\mathbf{Return}], [\![\,\gamma\,]\!]_X @ [\mathbf{Return}], [\![\,\delta\,]\!]_X @ [\mathbf{Return}])]$

On a utilisé un *codage* des atomes (**spade**, **heart**, **club**, **diamond**) par des entiers. Les *chaînes de caractères* sont plus "parlantes" (pour les humains) mais les *entiers* sont plus "compacts" (pour la machine).

1.5.4 Le coproduit comme type concret

On représente le coproduit $A \vee B$ par le *type concret* [CAML]:

$$\mathbf{Sum} = \mathbf{inl\ of}\ A \mid \mathbf{inr\ of}\ B$$

On peut interpréter cette définition de différentes façons:

- Comme une construction *primitive*. Mais alors, on ne rend pas compte du fait qu'une valeur de **Sum** est un *couple* (m, u) où m est un marqueur et u une valeur de A ou de B.
- Comme l'*union* $\{\mathbf{inl}\} \wedge A \cup \{\mathbf{inr}\} \wedge B$. Mais comment donner un sens "catégorique" à l'union de deux types quelconques?
- Comme la somme dépendante $\sum_{m \in \mathbf{inl} \mid \mathbf{inr}} \mathbf{case}\ m\ \mathbf{of}\ \mathbf{inl} \to A \mid \mathbf{inr} \to B$. Dans un *système de types dépendants*, le produit cartésien $A \wedge B$ est un cas particulier de somme dépendante $(\sum_{x \in A} B)$. Cela permet de considérer les *types énumérés*, et la *somme dépendante* comme des constructions primitives, et les types concrets (en particulier, le coproduit) comme une construction dérivée. Mais nous n'avons pas de formalisme satisfaisant pour cela[17].

En l'absence d'un formalisme satisfaisant, nous proposons d'adopter:

- Le premier point de vue (les *types concrets* comme construction primitive) pour un *contrôle des types* rigoureux.
- Le troisième point de vue (*somme dépendante*) pour l'intuition, et pour justifier la compilation.

Le point de vue de la *somme dépendante* permet les identifications suivantes:

- $\mathbf{inl}\ \mu$ avec $(\mathbf{inl}, \mu)$
- $\mathbf{inr}\ \mu$ avec $(\mathbf{inr}, \mu)$
- $\mathbf{case}\ \lambda\ \mathbf{of}\ (\mathbf{inl}\ x) \to \mu \mid (\mathbf{inl}\ y) \to \nu$ avec
  $\mathbf{let}\ z = \lambda\ \mathbf{in\ case}\ (\mathbf{fst}\ z)\ \mathbf{of}\ \mathbf{inl} \to \mu[(\mathbf{snd}\ z)/x] \mid \mathbf{inr} \to \nu[(\mathbf{snd}\ z)/y]$
- $\mathbf{fun}\ (\mathbf{inl}\ x) \to \mu \mid (\mathbf{inl}\ y) \to \nu$ avec
  $\mathbf{fun}\ z \to \mathbf{case}\ (\mathbf{fst}\ z)\ \mathbf{of}\ \mathbf{inl} \to \mu[(\mathbf{snd}\ z)/x] \mid \mathbf{inr} \to \nu[(\mathbf{snd}\ z)/y]$

D'où la compilation:

- $[\![\ \mathbf{inl}\ \mu\ ]\!]_X = [\mathbf{Push}; \mathbf{Quote}(0); \mathbf{Swap}] \mathbin{@} [\![\ \mu\ ]\!]_X \mathbin{@} [\mathbf{Cons}]$
- $[\![\ \mathbf{inr}\ \mu\ ]\!]_X = [\mathbf{Push}; \mathbf{Quote}(1); \mathbf{Swap}] \mathbin{@} [\![\ \mu\ ]\!]_X \mathbin{@} [\mathbf{Cons}]$
- $[\![\ \mathbf{case}\ \lambda\ \mathbf{of}\ (\mathbf{inl}\ x) \to \mu \mid (\mathbf{inl}\ y) \to \nu\ ]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\ \lambda\ ]\!]_X \mathbin{@} [\mathbf{Cons}; \mathbf{Push}; \mathbf{Snd}; \mathbf{Fst}; \mathbf{Cons}; \mathbf{Branch}_2([\![\ \mu\ ]\!]_{(X,(\_,x))} \mathbin{@} [\mathbf{Return}], [\![\ \nu\ ]\!]_{(X,(\_,y))} \mathbin{@} [\mathbf{Return}])]$
- $[\![\ \mathbf{fun}\ (\mathbf{inl}\ x) \to \mu \mid (\mathbf{inl}\ y) \to \nu\ ]\!]_X = [\mathbf{Cur}([\mathbf{Push}; \mathbf{Snd}; \mathbf{Fst}; \mathbf{Cons}; \mathbf{Branch}_2([\![\ \mu\ ]\!]_{(X,(\_,x))} \mathbin{@} [\mathbf{Return}], [\![\ \nu\ ]\!]_{(X,(\_,y))} \mathbin{@} [\mathbf{Return}]); \mathbf{Return}])]$
- $[\![\ x\ ]\!]_{(X,(\_,x))} = [\mathbf{Snd}; \mathbf{Snd}]$
- $[\![\ x\ ]\!]_{(X,(\_,y))} = [\mathbf{Fst}] \mathbin{@} [\![\ x\ ]\!]_X$

Nous avons utilisé l'instruction $\mathbf{Branch}_2(-,-)$, conformément à la convention établie pour les *types énumérés* (1.5.3). Au renommage près, c'est l'instruction $\mathbf{Branch}(-,-)$.

[17] Notez que la construction $\sum_{m \in \mathbf{inl} \mid \mathbf{inr}} \mathbf{case}\ m\ \mathbf{of}\ \mathbf{inl} \to A \mid \mathbf{inr} \to B$ n'existe pas dans le *système* de [Martinloef].

1.5.5 Autres types concrets

On étend ces constructions aux autres *types concrets* [CAML].

Par exemple, la définition **Nat = zero | succ of Nat** peut être considérée comme une abréviation pour **Nat = zero of** $\top$ **| succ of Nat**, où $\top$ est le type énuméré avec un seul élément (). C'est le *type dépendant récursif* $\mathbf{Nat} = \sum_{m \in \mathbf{zero} \mid \mathbf{succ}} \mathbf{case}\ m\ \mathbf{of\ zero} \rightarrow \top \mid \mathbf{succ} \rightarrow \mathbf{Nat}$.

Ainsi, on a :
$[\![\mathbf{fun\ zero} \rightarrow \mathbf{zero} \mid (\mathbf{succ}\ x) \rightarrow x]\!]_{\_} = [\mathbf{Cur}([\mathbf{Push}; \mathbf{Snd}; \mathbf{Fst}; \mathbf{Cons}; \mathbf{Branch_2}(C', C''); \mathbf{Return}])]$
où:

- $C' = [\![\mathbf{zero}]\!]_{(\_ \rightarrow (\_ \rightarrow \_))}\ @\ [\mathbf{Return}] = [\mathbf{Quote}((0,0)); \mathbf{Return}]$
- $C'' = [\![x]\!]_{(\_ \rightarrow (\_ \rightarrow x))}\ @\ [\mathbf{Return}] = [\mathbf{Snd}; \mathbf{Snd}; \mathbf{Return}]$

Les *types récursifs* sont utilisés avec des *fonctions récursives*. La récursion ne nécessite pas d'instruction spécifique, mais le code d'une fonction récursive est *bouclé* (section 1.7), si bien que la *terminaison* de l'évaluation (*Théorème de Convergence* de la section 1.3) n'est plus assurée[18].

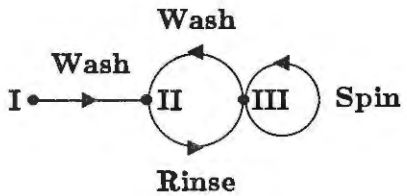
Nous ne prétendons pas donner une justification mathématique du mécanisme d'évaluation dans le cas des *programmes récursifs*[19]. Nous nous contentons d'une approche naïve.

1.6 Effets de Bord et Echappements

Dans cette section nous étudions les *traits impurs* de la programmation fonctionnelle, pour lesquels il est impossible d'étendre les résultats de la section 1.3. On fait du "rafistolage".

1.6.1 Actions primitives

On suppose ici qu'un programme produit une suite d'*actions primitives*[20], qui sont les flêches d'un graphe Σ:



Par exemple [**Wash**; **Rinse**] et [**Wash**; **Rinse**; **Wash**; **Rinse**; **Spin**] sont des *actions* de type $\mathbf{I} \rightarrow \mathbf{III}$, c'est à dire des éléments de $\mathbf{Hom}_{\Sigma^*}(\mathbf{I}, \mathbf{III})$ où Σ^* est la catégorie librement engendrée par Σ.

D'après le *Théorème de Cohérence Hiérarchique* (section 1.3) tout combinateur catégorique $\varphi : \mathbf{I} \rightarrow \mathbf{III}$ est congru à une *action* (unique) $\alpha : \mathbf{I} \rightarrow \mathbf{III}$.

On va donc étendre le langage:

[18] Il suffit d'avoir des *types récursifs* pour écrire des programmes (non récursifs) qui ne terminent pas (avec D tel que $D \simeq D \Rightarrow D$, on peut typer tous les termes du *λ-calcul pur*).

[19] La notion de *point fixe* est incompatible avec celle de *coproduit* [HuPoi].

[20] On retrouve cette idée dans les formalismes du parallélisme, comme CCS.

Tout d'abord **I**, **II**, **III** sont des formules atomiques.

Si on s'intéresse uniquement aux actions de type $\mathbf{I} \to ?$, on a les constructions suivantes:

$$\frac{}{\vdash \mathbf{skip} : \mathbf{I}}$$

$$\frac{\Gamma \vdash \mu : \mathbf{I}}{\Gamma \vdash \mathbf{wash}\ \mu : \mathbf{II}} \qquad \frac{\Gamma \vdash \mu : \mathbf{II}}{\Gamma \vdash \mathbf{rinse}\ \mu : \mathbf{III}} \qquad \frac{\Gamma \vdash \mu : \mathbf{III}}{\Gamma \vdash \mathbf{wash}\ \mu : \mathbf{II}} \qquad \frac{\Gamma \vdash \mu : \mathbf{III}}{\Gamma \vdash \mathbf{spin}\ \mu : \mathbf{III}}$$

A tout terme $x_1 : A_1, \ldots, x_n : A_n \vdash \mu : B$, on associe un combinateur $\varphi : (\cdots((\mathbf{I} \wedge A_1) \wedge A_2)\ldots) \wedge A_n \to B$:

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{I} \mapsto \mathbf{Id} : \mathbf{I} \to \mathbf{I}} \qquad \frac{\Gamma \vdash \mu : \mathbf{I} \mapsto \varphi : \hat{\Gamma} \to \mathbf{I}}{\Gamma \vdash \mathbf{wash}\ \mu : \mathbf{II} \mapsto \mathbf{Wash} \circ \varphi : \hat{\Gamma} \to \mathbf{II}} \qquad \ldots$$

En particulier, un terme clos $\vdash \mu : A$ donne un combinateur $\varphi : \mathbf{I} \to A$.

Naturellement, on a des instructions primitives **Wash**, **Rinse**, **Spin** pour la machine catégorique, et la compilation:

- $[\![\,\mathbf{skip}\,]\!]_{\_} = []$
- $[\![\,\mathbf{skip}\,]\!]_{(X,x)} = [\mathbf{Fst}] \mathbin{@} [\![\,\mathbf{skip}\,]\!]_X$
- $[\![\,\mathbf{wash}\ \mu\,]\!]_X = [\![\,\mu\,]\!]_X \mathbin{@} [\mathbf{Wash}]$
- $\cdots$

Par exemple, si la variable globale *white* vaut **True**, on a:
$[\![\,\mathbf{let}\ f = \mathbf{fun}\ x \to \mathbf{rinse}\ (\mathbf{wash}\ x)\ \mathbf{in\ if}\ \mathit{white}\ \mathbf{then\ spin}\ (f(f\ \mathbf{skip}))\ \mathbf{else}\ (f\ \mathbf{skip})\,]\!]_{\_} =$
$[\mathbf{Push}; \mathbf{Cur}(C); \mathbf{Cons}; \mathbf{Push}; \mathbf{Quote(True)}; \mathbf{Cons}; \mathbf{Branch}(C', C'')]$ où

- $C = [\![\,\mathbf{rinse}\ (\mathbf{wash}\ x)\,]\!]_{(\_,x)} \mathbin{@} [\mathbf{Return}] = [\mathbf{Snd}; \mathbf{Wash}; \mathbf{Rinse}; \mathbf{Return}]$
- $C' = [\![\,\mathbf{spin}\ (f(f\ \mathbf{skip}))\,]\!]_{(\_,f)} \mathbin{@} [\mathbf{Return}] =$
  $[\mathbf{Push}; \mathbf{Snd}; \mathbf{Swap}; \mathbf{Push}; \mathbf{Snd}; \mathbf{Swap}; \mathbf{Fst}; \mathbf{Cons}; \mathbf{App}; \mathbf{Cons}; \mathbf{App}; \mathbf{Spin}; \mathbf{Return}]$
- $C'' = [\![\,f\ \mathbf{skip}\,]\!]_{(\_,f)} \mathbin{@} [\mathbf{Return}] = [\mathbf{Push}; \mathbf{Snd}; \mathbf{Swap}; \mathbf{Fst}; \mathbf{Cons}; \mathbf{App}; \mathbf{Return}]$

La liste des actions primitives exécutées par la machine est [**Wash**; **Rinse**; **Wash**; **Rinse**; **Spin**], comme on l'attendait.

Malheureusement, un programme très simple comme **let** x = **rinse** (**wash skip**) **in skip** n'est pas exécuté correctement[21] (on obtient [**Wash**; **Rinse**] au lieu de []). Or il n'y a aucune raison de refuser un tel programme.

Cette difficulté est liée à la nature *stricte* (non *paresseuse*) du mécanisme d'évaluation. Pour être correcte, la *Machine Catégorique* ne devrait pas *exécuter* les *actions primitives*, mais *calculer* une action (exécutée *après* l'évaluation).

[21] Le combinateur catégorique correspondant à ce programme est **Fst** ∘ ⟨**Id**, **Rinse** ∘ **Wash**⟩ qui est congru à **Id**.

1.6.2 Actions calculées

Suivant la preuve catégorique du *Théorème de Cohérence Hiérarchique* (annexe C), on pose:

- $\mathbf{Val(A) = Hom_{\Sigma^*}(I, A)}$ pour $A = \mathbf{I}, \mathbf{II}, \mathbf{III}$.

Une valeur d'un type de base est donc simplement une suite d'actions primitives:

$$\frac{}{\mathbf{skip} : \mathbf{I}} \qquad \frac{u : \mathbf{I}}{\mathbf{wash}\ u : \mathbf{II}} \qquad \frac{u : \mathbf{II}}{\mathbf{rinse}\ u : \mathbf{III}} \qquad \frac{u : \mathbf{III}}{\mathbf{wash}\ u : \mathbf{II}} \qquad \frac{u : \mathbf{III}}{\mathbf{spin}\ u : \mathbf{III}}$$

Les *instructions primitives* sont *mémorisées* au lieu d'être *exécutées*:

Avant			Après		
code	environnement	pile	code	environnement	pile
Wash :: C	u	S	C	**wash** u	S
...					

Cette fois, let x = **rinse (wash skip) in skip** est exécuté correctement (**rinse (wash skip)** est *calculé*, mais pas *exécuté*).

Notez que les valeurs de type **I, II, III** pourraient être représentées au moyen du type concret récursif:

$$\mathbf{Action = skip \mid wash\ of\ Action \mid rinse\ of\ Action \mid spin\ of\ Action}$$

C'est pourquoi *on n'introduit pas* ces constructions dans les *langages fonctionnels*. Par contre, *on autorise les effets de bord*, c'est à dire les actions exécutées au cours de l'évaluation d'une expression. Ici, nous ne considérons que les *effets externes* (*entrées/sorties*), mais on peut également admettre des *effets de bord* qui modifient l'environnement interne (avec des *références*).

Comme nous l'avons remarqué, cette *sémantique opérationnelle* est fondamentalement *incorrecte*. Plus exactement, elle est incompatible avec les *équations catégoriques*. Mais, sans cette licence, il serait difficile d'écrire des *programmes interactifs* dans un langage fonctionnel.

Toutefois, le concept *d'action calculée* apparaît dans un grand nombre de programmes comportant deux phases successives:

- *calcul* (sans impression) d'une valeur d'un type concret.
- *impression* (sans calcul essentiel) de cette valeur.

Mais la syntaxe des langages fonctionnels n'impose pas ce schéma.

1.6.3 Instructions pour les effets de bord

Les *effets de bord* étant autorisés, on utilise l'evaluation en séquence pour programmer en *style procédural*:

$$\frac{\Gamma \vdash \mu : A \quad \Gamma \vdash \nu : B}{\Gamma \vdash \mu ; \nu : B}$$

$\mu ; \nu$ peut être considéré comme une abréviation pour **snd** (μ, ν), mais on a une compilation directe avec l'instruction **Pop**:

Avant			Après		
code	environnement	pile	code	environnement	pile
$\mathbf{Pop} :: C$	u	$v :: S$	C	v	S

- $[\![\mu ; \nu]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Pop}] \mathbin{@} [\![\nu]\!]_X$

D'autre part, on utilise des *fonctions primitives* pour les actions primitives. Une *fonction primitive* est une valeur de la forme $(C \cdot \_)$, où C est du code *non catégorique*. Lorsqu'il n'y a ni argument, ni résultat, on utilise le type $\top \to \top$.

Avec ces conventions, notre programme s'écrira:

$$\mathbf{let}\ f = \mathbf{fun}\ () \to (\mathit{wash}\ (); \mathit{rinse}\ ())\ \mathbf{in\ if}\ \mathit{white}\ \mathbf{then}\ (f\ (); f\ (); \mathit{spin}\ ())\ \mathbf{else}\ f\ ()$$

1.6.4 Instructions pour les échappements

Comme les *effets de bords*, les *échappements* sont incompatibles avec les *équations catégoriques*. Mais on peut facilement les ajouter à la *Machine Catégorique*.

Pour simplifier, on se limite à *une seule exception* qui ne renvoie pas de valeur [CAML]:

$$\frac{}{\Gamma \vdash \mathbf{fail} : A} \qquad \frac{\Gamma \vdash \mu : A \quad \Gamma \vdash \nu : A}{\Gamma \vdash \mu\, ?\, \nu : A}$$

Il est nécessaire d'ajouter une *pile d'échappement*[22] pour sauvegarder les états de la machine:

Avant				Après			
code	env^{nt}	pile	pile d'éch$^{\text{nt}}$	code	env^{nt}	pile	pile d'éch$^{\text{nt}}$
$\mathbf{Fail} :: C$	u	S	$(C', u', S') :: T$	C'	u'	S'	T
$\mathbf{PopTrap} :: C$	u	S	$(C', u', S') :: T$	C	u	S	T
$\mathbf{PushTrap}(C', C'') :: C$	u	S	T	C'	u	$C :: S$	$(C'', u, S) :: T$

- $[\![\mathbf{fail}]\!]_X = [\mathbf{Fail}]$
- $[\![\mu\, ?\, \nu]\!]_X = [\mathbf{PushTrap}([\![\mu]\!]_X \mathbin{@} [\mathbf{PopTrap}; \mathbf{Return}], [\![\nu]\!]_X \mathbin{@} [\mathbf{Return}])]$

1.7 Quelques Optimisations

1.7.1 Amélioration de la Machine Catégorique

Dans le code compilé, certaines instructions (**App**, **Branch**$(-,-)$, et les primitives binaires $+$, $\times$, ...) sont toujours précédées de l'instruction **Cons** qui, on l'a vu, est coûteuse en place mémoire.

On remplace les séquences [**Cons**; **App**], [**Cons**; **Branch**$(-,-)$], ... par des instructions primitives (**App**, **Branch**$(-,-)$, ...) qui *dépilent* leur premier argument:

[22] En fait, on n'a pas besoin d'une deuxième pile. Il suffit d'un *registre* supplémentaire. La *pile d'échappement* peut être représentée par une *liste chaînée* imbriquée dans la pile principale [Suarez].

Avant			Après		
code	environnement	pile	code	environnement	pile
$\mathbf{App} :: C$	u	$(C' \cdot v) :: S$	C'	(v, u)	$C :: S$
$\mathbf{Branch}(C', C'') :: C$	**True**	$u :: S$	C'	u	$C :: S$
$\mathbf{Branch}(C', C'') :: C$	**False**	$u :: S$	C''	u	$C :: S$
$+ :: C$	u	$v :: S$	C	$u + v$	S
$\times :: C$	u	$v :: S$	C	$u \times v$	S

- $[\![\mu\, \nu]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![\nu]\!]_X \mathbin{@} [\mathbf{App}]$
- $[\![\mathbf{if}\ \lambda\ \mathbf{then}\ \mu\ \mathbf{else}\ \nu]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\lambda]\!]_X \mathbin{@} [\mathbf{Branch}([\![\mu]\!]_X \mathbin{@} [\mathbf{Return}], [\![\nu]\!]_X \mathbin{@} [\mathbf{Return}])]$
- $[\![\mu + \nu]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![\nu]\!]_X \mathbin{@} [+]$
- $[\![\mu * \nu]\!]_X = [\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![\nu]\!]_X \mathbin{@} [\times]$

1.7.2 Fonctions locales et récursion

Dans la compilation d'une expression $\mathbf{let}\ f = \mathbf{fun}\ x \to \mu\ \mathbf{in}\ \nu$, il n'est pas nécessaire d'ajouter f à l'environnement. En effet, la valeur de f est une fermeture $(C \cdot u)$, où C est calculé à la compilation, et u est l'environnement de cette expression.

On utilise des *environnements annotés*:

- $[\![\mathbf{let}\ f = \mathbf{fun}\ x \to \mu\ \mathbf{in}\ \nu]\!]_X = [\![\nu]\!]_{X\{f \to C\}}$ où $C = [\![\mu]\!]_{(X,x)} \mathbin{@} [\mathbf{Return}]$
- $[\![x]\!]_{X\{x \to C\}} = [\mathbf{Cur}(C)]$
- $[\![x]\!]_{X\{y \to C\}} = [\![x]\!]_X$

Cette compilation est plus efficace, car elle n'augmente pas la taille de l'*environnement réel*[23]. De plus, elle se généralise aux fonctions récursives:

- $[\![\mathbf{let\ rec}\ f = \mathbf{fun}\ x \to \mu\ \mathbf{in}\ \nu]\!]_X = [\![\nu]\!]_{X\{f \to C\}}$ où $C = [\![\mu]\!]_{(X\{f \to C\},x)} \mathbin{@} [\mathbf{Return}]$

On produit du code *bouclé*, ce qui est raisonnable pour la récursion.

Par exemple, on a[24]:

$[\![\mathbf{let\ rec}\ f = \mathbf{fun\ zero} \to 0 \mid (\mathbf{succ}\ y) \to f\ y + 1\ \mathbf{in}\ f\ x]\!]_{(\neg x)} = [\mathbf{Push}; \mathbf{Cur}(C); \mathbf{Swap}; \mathbf{Snd}; \mathbf{App}]$
où

- $C = [\mathbf{Push}; \mathbf{Snd}; \mathbf{Fst}; \mathbf{Branch}_2(C', C''); \mathbf{Return}]$
- $C' = [\![0]\!]_{((\neg x)\{f \to C\},(\neg \_))} \mathbin{@} [\mathbf{Return}] = [\mathbf{Quote}(0); \mathbf{Return}]$
- $C'' = [\![f\ y + 1]\!]_{((\neg x)\{f \to C\},(\neg y))} \mathbin{@} [\mathbf{Return}] =$
  $[\mathbf{Push}; \mathbf{Fst}; \mathbf{Cur}(C); \mathbf{Swap}; \mathbf{Snd}; \mathbf{Snd}; \mathbf{App}; \mathbf{Push}; \mathbf{Quote}(1); +; \mathbf{Return}]$[25]

Cette méthode s'applique également aux fonctions *mutuellement récursives.*

[23] L'environnement qui apparaît *à l'exécution.* Les annotations n'apparaissent qu'*à la compilation.*

[24] On utilise ici les *nouvelles* instructions primitives (1.7.1).

[25] On a utilisé ici une autre optimisation:

$[\![\mu + \nu]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Push}] \mathbin{@} [\![\nu]\!]_X \mathbin{@} [+]$ plutôt que $[\mathbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![\nu]\!]_X \mathbin{@} [+]$ lorsque ν est une expression *close.*

1.7.3 Termes non fonctionnels, variables locales et fonctions globales

Ce qui fait la puissance des langages fonctionnels, c'est l'existence des *types fonctionnels*. Si on n'utilise que des *fonctions globales*, il n'y a aucune raison de représenter l'environnement sous forme d'un arbre. On a une *compilation directe* (sans passer par les combinateurs catégoriques) en utilisant la pile.

A tout terme *non fonctionnel* μ, avec $x_n : A_n, \ldots, x_1 : A_1 \vdash \mu : B$, on associe $[\![\mu]\!]_{[x_1;\cdots;x_n]}$. Si on exécute ce code avec une pile $u_1 :: \cdots :: u_n :: S$, la valeur obtenue est $\mu[u_1/x_1, \ldots, u_n/x_n]$, et *la pile est inchangée*[26].

On introduit une instruction d'accès *en profondeur*:

Avant			Après		
code	valeur	pile	code	valeur	pile
$\mathbf{Acces}_i :: C$	u	$u_1 :: \cdots :: u_i :: S$	C	u_i	$u_1 :: \cdots :: u_i :: S$

- $[\![x_i]\!]_{x_1::\cdots::x_i::X} = \mathbf{Acces}_i$
- $[\![\mathbf{let}\ x = \mu\ \mathbf{in}\ \nu]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Push}] \mathbin{@} [\![\nu]\!]_{x::X} \mathbin{@} [\mathbf{Pop}]$
- $[\![\mathbf{fst}\ \mu]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Fst}]$
- $[\![\mathbf{snd}\ \mu]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Snd}]$
- $[\![(\mu, \nu)]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Push}] \mathbin{@} [\![\nu]\!]_{\_::X} \mathbin{@} [\mathbf{Cons}]$
- $[\![c]\!]_X = [\mathbf{Quote}(u)]$ (c est une constante ou une variable globale de valeur u)
- $[\![\mathbf{if}\ \lambda\ \mathbf{then}\ \mu\ \mathbf{else}\ \nu]\!]_X = [\![\lambda]\!]_X \mathbin{@} [\mathbf{Branch}([\![\mu]\!]_X \mathbin{@} [\mathbf{Return}], [\![\nu]\!]_X \mathbin{@} [\mathbf{Return}])]$
- $[\![\mu + \nu]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Push}] \mathbin{@} [\![\nu]\!]_{\_::X} \mathbin{@} [+]$
- $\ldots$

Pour les *fonctions globales*, on ajoute les instructions:

Avant			Après		
code	valeur	pile	code	valeur	pile
$\mathbf{Call}(C') :: C$	u	S	C'	u	$C :: S$
$\mathbf{Pop}_n :: C$	u	$u_1 :: \cdots :: u_n :: S$	C	u	S

A toute fonction globale n-aire f définie par $\mathbf{let}\ f\ x_n \ldots x_1 = \mu$, on associe le code:

$$C = [\![\mu]\!]_{[\_;x_1;\cdots;x_n]} \mathbin{@} [\mathbf{Return}]^{27}$$

- $[\![f\ \mu_n \ldots \mu_1]\!]_X =$
  $[\![\mu_n]\!]_X \mathbin{@} [\mathbf{Push}] \mathbin{@} [\![\mu_{n-1}]\!]_{\_::X} \mathbin{@} [\mathbf{Push}] \mathbin{@} \ldots \mathbin{@} [\![\mu_1]\!]_{\_::\cdots::\_::X} \mathbin{@} [\mathbf{Push}; \mathbf{Call}(C); \mathbf{Pop}_n]$

[26] Il s'agit d'un mode de compilation très classique pour des langages qui n'admettent que les *fonctions globales* (comme PASCAL). Le cas de LISP est plus compliqué, car c'est un langage *non typé* qui, à l'origine, manipule des *expressions* plutôt que des *fonctions*.

[27] Le premier emplacement de l'environnement $[\_; x_1; \cdots; x_n]$ est occupé par l'adresse de retour

Naturellement, ce type de compilation ne s'étend pas aux *termes fonctionnels*[28]. Par exemple, pour évaluer $\mathbf{let}\, x = 3\, \mathbf{in}\, \mathbf{fun}\, y \rightarrow x + y$, on ne peut pas garder x sur la pile (x est une variable locale *rémanente*). Mais on peut traiter ainsi toute *variable locale qui n'apparaît pas libre à l'intérieur d'une abstraction* [Suarez].

On pourrait également optimiser la compilation des *fonctions globales*, mais le code obtenu ne serait pas utilisable lorsque la fonction est appliquée *partiellement* à ses arguments.

1.7.4 Autres optimisations

Il y a beaucoup d'autres optimisations dans [Suarez], notamment la reconnaissance des *expressions closes* qui simplifie la gestion de l'environnement.

Un autre type d'optimisation (plus dépendante de l'implantation), qui permet d'économiser de la place, est l'élimination des *constructeurs superflus*.

Considérons par exemple le type concret (section 1.5):

$$\mathbf{Nat} = \mathbf{zero} \mid \mathbf{succ\ of\ Nat}$$

Il s'agit, rappelons le, d'une abréviation pour $\mathbf{Nat} = \mathbf{zero\ of}\ \top \mid \mathbf{succ\ of\ Nat}$. Mais comme l'ensemble des adresses de paires est *disjoint* de l'ensemble des atomes (remarque 15, section 1.4), on peut représenter **zero**() par un atome plutôt que par une paire[29]. On utilise alors une instruction qui reconnaît les atomes, au lieu de $\mathbf{Branch}_2(-,-)$.

Comme les valeurs de **Num** sont des atomes, on peut appliquer la même optimisation au type:

$$\mathbf{Expr} = \mathbf{atom\ of\ Num} \mid \mathbf{pair\ of\ Expr} \wedge \mathbf{Expr}$$

Dans ce cas, on peut éliminer les *constructeurs superflus* **atom** et **pair**.

1.7.5 Quelques avantages de la compilation catégorique

Le formalisme catégorique donne un sens à des expressions qui ne proviennent pas directement de la *Déduction Naturelle*, comme $\mathbf{let}\ (x, y) = \mu\ \mathbf{in}\ \nu$.

L'environnement est un arbre de variables ou *motif*:

- Les variables sont des motifs.
- _ est un motif.
- Si X et Y sont des motifs, (X, Y) est un motif.

On compile avec ces nouveaux environnements:

- $[\![x]\!]_x = []$
- $[\![x]\!]_y$ échoue, ainsi que $[\![x]\!]_\_$
- $[\![x]\!]_{(X,Y)} = [\mathbf{Snd}] \mathbin{@} [\![x]\!]_Y$ ou, en cas d'échec $[\mathbf{Fst}] \mathbin{@} [\![x]\!]_X$
- $[\![\mathbf{fun}\, Y \rightarrow \mu]\!]_X = [\mathbf{Cur}([\![\mu]\!]_{(X,Y)} \mathbin{@} [\mathbf{Return}])]$

[28] En fait, il est possible de construire des fermetures en *recopiant la pile*. Par rapport à la *Machine Catégorique*, cette implantation présente à la fois des avantages (accès plus directs) et des inconvénients (fermetures plus coûteuses).

[29] Ce qui justifie l'abréviation **Nat = zero | succ of Nat**

- $[\![\textbf{let } Y = \mu \textbf{ in } \nu]\!]_X = [\textbf{Push}] \mathbin{@} [\![\mu]\!]_X \mathbin{@} [\textbf{Cons}] \mathbin{@} [\![\nu]\!]_{(X,Y)}$

De ce fait, **fst** μ est équivalent à **let** $(x, \_) = \mu$ **in** x (et **snd** μ à **let** $(\_, x) = \mu$ **in** x). On n'a donc pas besoin de **fst** $-$ et **snd** $-$ dans la syntaxe du *λ-Calcul Typé.*

Si on admet aussi des *atomes* comme *motifs*, on obtient la *définition par filtrage* (*pattern matching*) qui généralise la *définition par cas* (section 1.5).

1.8 Conclusion

Nons n'avons pas abordé tous les aspects théoriques de l'implantation des langages fonctionnels (polymorphisme, synthèse des types, modules, ...). Nous avons seulement *expliqué* le *mécanisme d'évaluation stricte* des *langages fonctionnels* à partir de la théorie des *Catégories Cartésiennes Fermées*[30].

Nous proposons une méthode générale pour implanter d'autres langages:

- Donner une présentation axiomatique simple du langage (de préférence une théorie équationnelle).
- Démontrer un *théorème de cohérence hiérarchique.*
- De la preuve de ce théorème, extraire un mécanisme d'exécution des programmes[31].

Le chapitre suivant[32] contient une application de ce principe.

[30] Nous n'avons pas de justification analogue pour le *mécanisme d'évaluation paresseuse* [Mauny]. De plus, les *effets de bord, références, échappements,* ... n'entrent pas dans ce cadre.

[31] Pour certaines théories, par exemple la *Théorie des Topos* [LamSco], on a un *théorème de cohérence hiérarchique*, mais il semble assez difficile d'en extraire un mécanisme d'exécution.

[32] On n'y trouve pas la formulation précise d'un *théorème de cohérence hiérarchique*, mais il s'agit bien de cela.

Chapitre 2

La Machine Linéaire Abstraite

Considérons quelques problèmes qui se posent dans le domaine de la *programmation fonctionnelle*:

Cellules partagées

La *programmation fonctionnelle pure* est élégante, mais bien souvent inefficace. En général, on ajoute des *références* pour des raisons pratiques, au détriment de la transparence sémantique.

En LISP, il n'est pas besoin de *références* pour modifier les données. Par exemple, la "fonction" *nconc* (concaténation physique de listes, figure 2.1) remplace le *nil* final de son premier argument par son second argument, altérant toutes les données qui *partagent* la dernière cellule de ce premier argument.

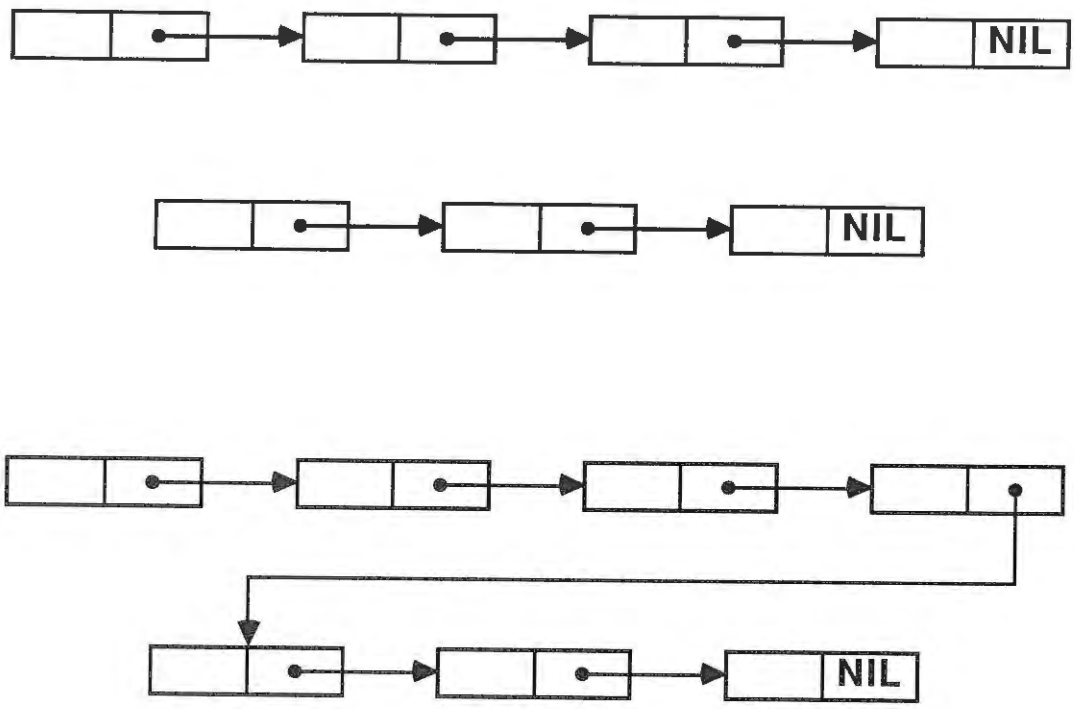


Figure 2.1: nconc

Peut-on intégrer ces *fonctions impures* (qui altérent ou détruisent leurs arguments) dans un style déclaratif?

Autogestion

Dans certains dialectes LISP, la *liste libre* est accessible par l'utilisateur qui peut ainsi gérer lui-même la récupération de sa mémoire. Parfois, il est clair que les données ne seront plus jamais utilisées, et il est raisonnable de récupérer de l'espace, mais les conséquences d'une éventuelle méprise sont dramatiques!

Peut-on assurer que les données ne seront plus jamais utilisées?

Cohabitation

On connaît deux principaux mécanismes d'évaluation pour les *langages fonctionnels*. Le mécanisme *strict* (*appel par valeur*) est généralement plus performant, mais le mécanisme paresseux (*appel par nécessité*) permet d'élégantes constructions infinies.

Peut-on faire cohabiter harmonieusement ces deux mécanismes?

On peut s'attaquer à ces problèmes avec des outils d'analyse *ad hoc*, mais cela devient très *ardu* lorsqu'on admet des *fonctions d'ordre supérieur* .

Nous proposons plutôt un *calcul typé* qui tient compte de ces *détails d'implantation*. Du coup, nous quittons le domaine de la *programmation fonctionnelle*, proposant:

- un nouveau type de langage
- une nouvelle technique d'implantation (la *Machine Linéaire Abstraite*).

La *Logique Linéaire Intuitionniste* de J.Y. Girard est à ce nouveau type de langage ce que la *Logique Intuitionniste* est aux langages fonctionnels (paradigme de Curry-Howard). Il semble que la *Logique Linéaire* [Girard87] soit la clef de questions cruciales en informatique (voir [Girard86,Lafont87] par exemple).

Comme le lecteur n'est peut-être pas familiarisé avec la *Logique Linéaire*, nous en proposons une introduction illustrée (sections 2.1 à 2.3). Ensuite, nous présentons la *Machine Linéaire Abstraite*, principal objet de ce chapitre (sections 2.4 et 2.5) et nous donnons une réponse positive aux questions précédentes (section 2.6). Le *λ-Calcul Linéaire* (section 2.7) est la base d'un *langage de programmation* qui peut être implanté sur ce type de machine. Les rappels et détails techniques sont rassemblés en annexe.

Ce chapitre est une version remaniée et complétée de [GirLaf].

2.1 Calcul des Séquents

2.1.1 Calcul des Séquents de Gentzen

Moins connu que la *Déduction Naturelle*, le *Calcul des Séquents* de Gentzen [Gentzen,Kleene] est sans doute la présentation la plus *parfaite* de la *Logique Intuitionniste*. Ici, on a la claire distinction entre les règles *structurelles* et *logiques* (voir annexe A).

Les règles *structurelles* sont: l'*échange*, l'*identité* (habituellement considérée comme un axiome, elle peut être restreinte aux formules atomiques), la *coupure*, la *contraction* et l'*affaiblissement*. La principale propriété du *Calcul des Séquents* est le théorème d'*élimination des coupures* (*Hauptsatz*): Dans la preuve d'un séquent $\Gamma \vdash A$, la règle de *coupure* peut être éliminée.

De l'*Hauptsatz*, on déduit la *cohérence* ($\vdash \bot$ n'est pas prouvable) et la *propriété de la sous-formule* (tout séquent prouvable $A_1, \ldots, A_n \vdash B$ admet une preuve qui ne contient que des sous-formules de $A_1, \ldots, A_n, B$).

Si vous oubliez les règles de la conjonction, vous pouvez hésiter entre les deux suivantes:

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad \text{ou} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

Elles sont équivalentes, car vous retrouvez la seconde à partir de la première:

$$\cfrac{\cfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma, \Gamma \vdash A \wedge B}}{\begin{array}{c}\vdots \\ \text{(contractions)} \\ \vdots\end{array}}{\Gamma \vdash A \wedge B}$$

... et la première à partir de la seconde:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash A}{\begin{array}{c}\vdots \\ \text{(affaiblissements)} \\ \vdots\end{array}}}{\Gamma, \Delta \vdash A} \quad \cfrac{\cfrac{\Delta \vdash B}{\begin{array}{c}\vdots \\ \text{(affaiblissements)} \\ \vdots\end{array}}}{\Gamma, \Delta \vdash B}}{\Gamma, \Delta \vdash A \wedge B}$$

2.1.2 Calcul des Séquents de Girard

Si vous retirez la *contraction* et l'*affaiblissement* du *Calcul des Séquents*, les règles précédentes ne sont plus équivalentes, et correspondent à des connecteurs distincts.

Vous obtenez le *Calcul des Séquents* de Girard pour la *Logique Linéaire Intuitionniste*. Les connecteurs sont $\otimes$ (produit tensoriel), $\mathbf{1}$ (unité tensorielle), $\multimap$ (implication linéaire), & (produit direct), $\mathbf{t}$ (unité directe), $\oplus$ (somme directe) and $\mathbf{0}$ (zéro direct).

Règles Structurelles:

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,(\text{échange}) \qquad \frac{}{A \vdash A}\,(\text{identité}) \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\,(\text{coupure})$$

(pas de contraction) (pas d'affaiblissement)

Règles Logiques:

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} \qquad \frac{}{\vdash \mathbf{1}} \qquad \frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B} \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \,\&\, B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \,\&\, B \vdash C} \qquad \frac{}{\Gamma \vdash \mathbf{t}}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C} \qquad \frac{}{\Gamma, \mathbf{0} \vdash A}$$

Dans le *Calcul des Séquents* de Gentzen, un séquent $A_1, \ldots, A_n \vdash B$ signifie que B est conséquence de $A_1 \wedge \ldots \wedge A_n$. Ici, il signifie que B est conséquence de $A_1 \otimes \ldots \otimes A_n$.

Notez que les règles pour $\otimes$, $\mathbf{1}$ et $\multimap$ sont particulièrement simples (les prémisses ne partagent pas d'environnement). Ce sont les connecteurs *multiplicatifs*. $\&$, $\mathbf{t}$, $\oplus$ et $\mathbf{0}$ sont les connecteurs *additifs*.

Théorème 7 *(Haupsatz pour la Logique Linéaire Intuitionniste)*
Dans la preuve d'un séquent $\Gamma \vdash A$, *la règle de* coupure *peut être éliminée.*

La démonstration est semblable à celle de Gentzen, et même plus simple, car il n'y a ni *contraction* ni *affaiblissement*.

2.1.3 Exemples de preuves

A partir de maintenant, *linéaire* sous-entend *linéaire intuitionniste*.

On a une traduction immédiate des formules *linéaires* dans les formules *intuitionnistes*, remplaçant $\otimes$ et $\&$ par $\wedge$ (conjonction), $\mathbf{1}$ et $\mathbf{t}$ par $\top$ (vrai), $\multimap$ par $\Rightarrow$ (implication), $\oplus$ par $\vee$ (disjonction) et $\mathbf{0}$ par $\bot$ (faux).

Si une formule *linéaire* est prouvable, sa traduction est une formule *intuitionniste* prouvable. Par exemple, la formule *linéaire* $(A \,\&\, B) \multimap A$ (où A et B sont des formules atomiques) admet la preuve suivante:

$$\frac{\dfrac{A \vdash A}{A \,\&\, B \vdash A}}{\vdash (A \,\&\, B) \multimap A}$$

Bien sûr, sa traduction $(A \wedge B) \Rightarrow A$ est prouvable.

Mais la réciproque est fausse: La formule *linéaire* $(A \otimes B) \multimap A$ n'est pas prouvable, alors que sa traduction $(A \wedge B) \Rightarrow A$ l'est toujours.

Montrons que $(A \otimes B) \multimap A$ n'est pas prouvable. Une preuve *sans coupure* de $\vdash (A \otimes B) \multimap A$ se termine nécessairement comme ceci:

$$\frac{\dfrac{\dfrac{\vdots}{A, B \vdash A}}{A \otimes B \vdash A}}{\vdash (A \otimes B) \multimap A}$$

Mais sans *affaiblissement*, il est évidemment impossible de déduire $A, B \vdash A$.

On a la propriété de distributivité:

$$A \otimes (B \oplus C) \equiv (A \otimes B) \oplus (A \otimes C)$$

Ici $A \equiv B$ veut dire que $A \vdash B$ et $B \vdash A$ sont toutes les deux prouvables.

Bien sûr, ce n'est pas vrai si on remplace $\otimes$ par $\&$.

Il est très facile de construire une preuve *sans coupure* en *remontant* l'arbre de déduction. Par exemple:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A} \quad \dfrac{}{B \vdash B}}{A, B \vdash A \otimes B}}{A, B \vdash (A \otimes B) \oplus (A \otimes C)} \quad \dfrac{\dfrac{\dfrac{}{A \vdash A} \quad \dfrac{}{C \vdash C}}{A, C \vdash A \otimes C}}{A, C \vdash (A \otimes B) \oplus (A \otimes C)}}{A, B \oplus C \vdash (A \otimes B) \oplus (A \otimes C)}$$
$$\overline{A \otimes (B \oplus C) \vdash (A \otimes B) \oplus (A \otimes C)}$$

2.1.4 Logique Linéaire Classique

Dans [Girard87], vous ne trouverez pas la *Logique Linéaire Intuitionniste* mais *Classique*.

A première vue, la *Logique Linéaire Classique* est à la *Logique Linéaire Intuitionniste* ce que la *Logique Classique* est à la *Logique Intuitionniste*. Par exemple, on a la négation comme connecteur primitif, et la *loi du tiers exclus*. Cependant, la *Logique Linéaire Classique* est *constructive*, ce qui n'est pas le cas de la *Logique Classique*!

Nous sommes maintenant certains que la *Logique Linéaire Classique* est la voie la plus prometteuse, mais le cas *intuitionniste* est probablement plus accessible, et il fut le premier *implanté*.

2.2 Logique Catégorique Combinatoire

On a vu dans le premier chapitre que la *Logique Catégorique Combinatoire* (section 1.2) est une présentation alternative de la *Logique Intuitionniste*.

Rappelons qu'un *combinateur catégorique* $\varphi : A \to B$ est la dénotation d'une preuve de $A \vdash B$ (une seule formule à gauche). On peut dire que la *Logique Catégorique Combinatoire* est plus *élémentaire* que le *Calcul des Séquents*, et de ce fait, elle est bien adaptée à l'implantation des langages fonctionnels.

2.2.1 Signification opérationnelle de la Logique Linéaire

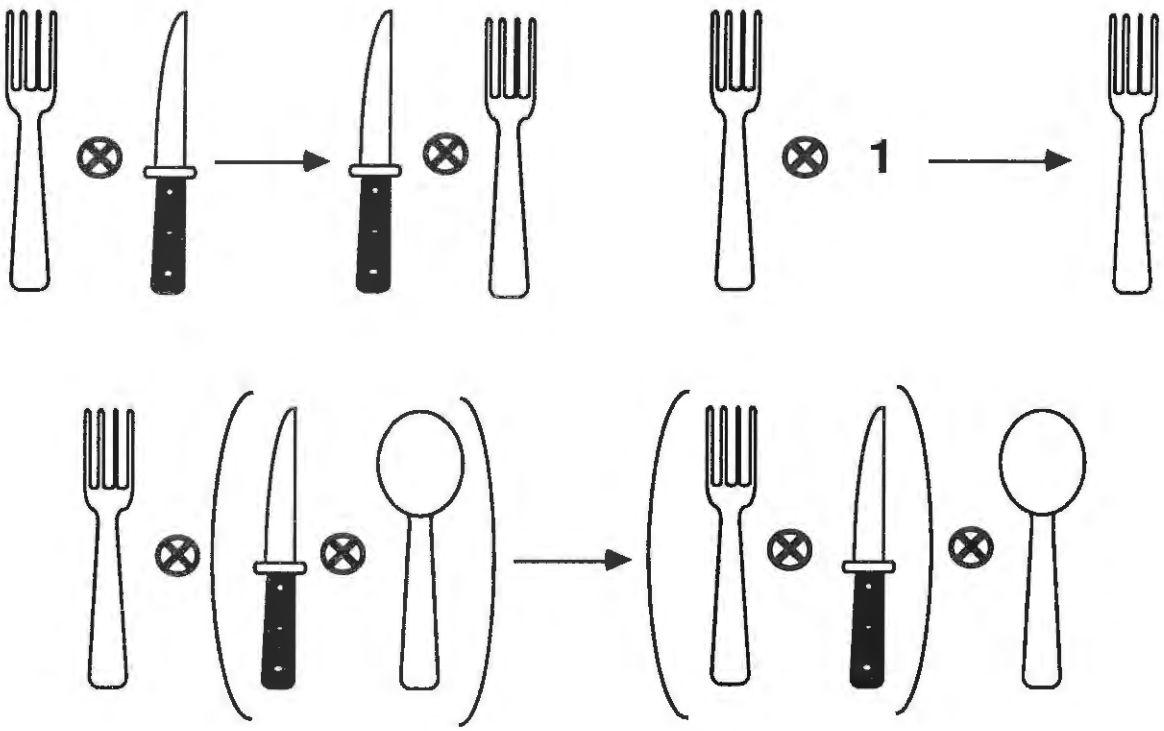
La *Logique Catégorique Combinatoire* est commode pour expliquer la signification *opérationnelle* des *formules linéaires*.

Dans le cas intuitionniste, une formule représente un *type de donnée*, et un combinateur $\varphi : A \to B$ dénote une *fonction* de A vers B.

Dans le cas linéaire, les données sont des *objets concrets* et les combinateurs dénotent des *transformations concrètes*. Le résultat est *construit* à partir des données, et rien n'est perdu, rien n'est créé.

Produit tensoriel

Une donnée de type $A \otimes B$ est la *juxtaposition* d'une donnée de type A et d'une donnée de type B. Vous pouvez *arranger* ou *réordonner* les données comme vous l'entendez, mais vous ne pouvez ni les *dupliquer* ni les *effacer* (figure 2.2).



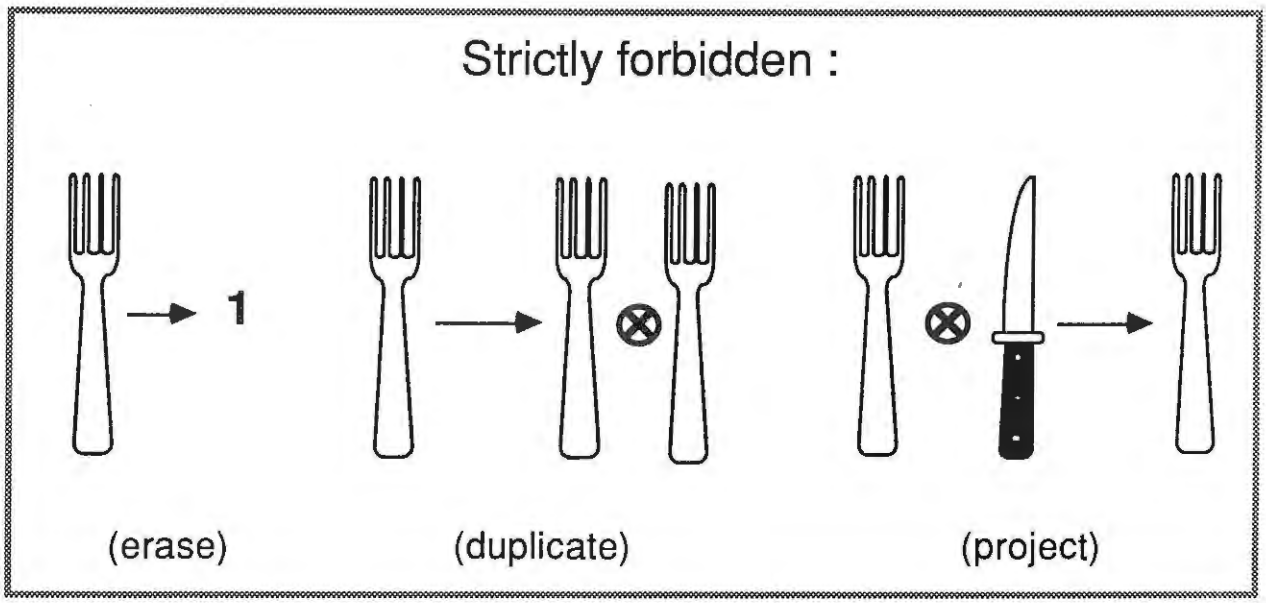


Figure 2.2: produit tensoriel

Somme directe

Pour expliquer les autres connecteurs, nous avons besoin de *chapeaux*. Par exemple, une donnée de type $A \oplus B$ est un *chapeau* qui peut contenir une donnée de type A ou une donnée de type B. Il est très facile d'obtenir une donnée de type $A \oplus B$ à partir d'une donnée de type A (figure 2.3). Le *chapeau* lui-même n'est pas considéré comme une donnée.

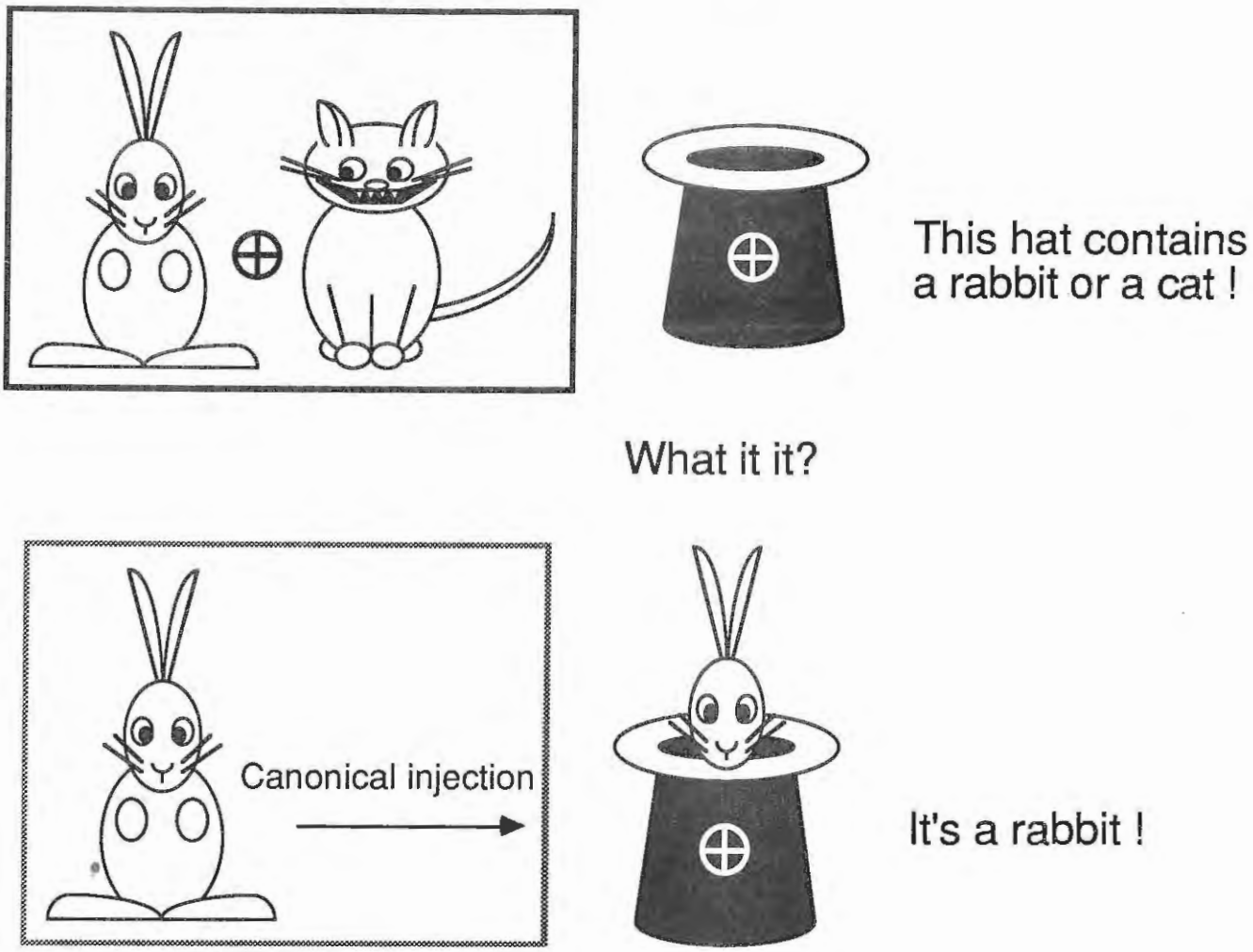


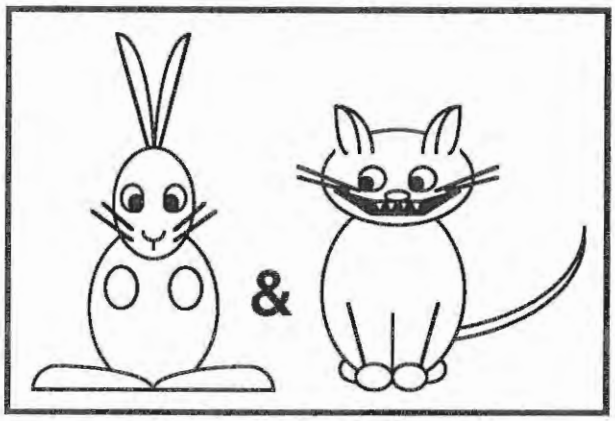
Figure 2.3: somme directe

Produit direct

Pour le *produit direct*, il nous faut un *chapeau magique*. Une donnée de type $A \& B$ est un *chapeau* qui peut contenir une donnée de type A ou une donnée de type B, mais (c'est magique!) vous pouvez demander si vous désirez une donnée de type A ou une donnée de type B (figure 2.4).

Implication linéaire

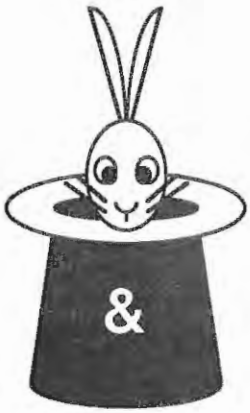
Le *chapeau* pour $A \multimap B$ n'est plus magique, mais toujours mystérieux. En effet, il contient quelque chose pour produire une donnée de type B à partir d'une donnée de type A, mais vous ne saurez jamais ce que c'était au juste (figure 2.5).



Do you want a rabbit or a cat ?

I want a rabbit!

Here is a rabbit !



Strictly forbidden :

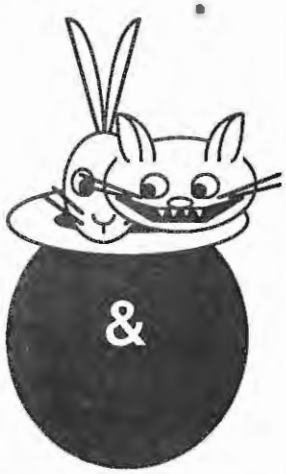
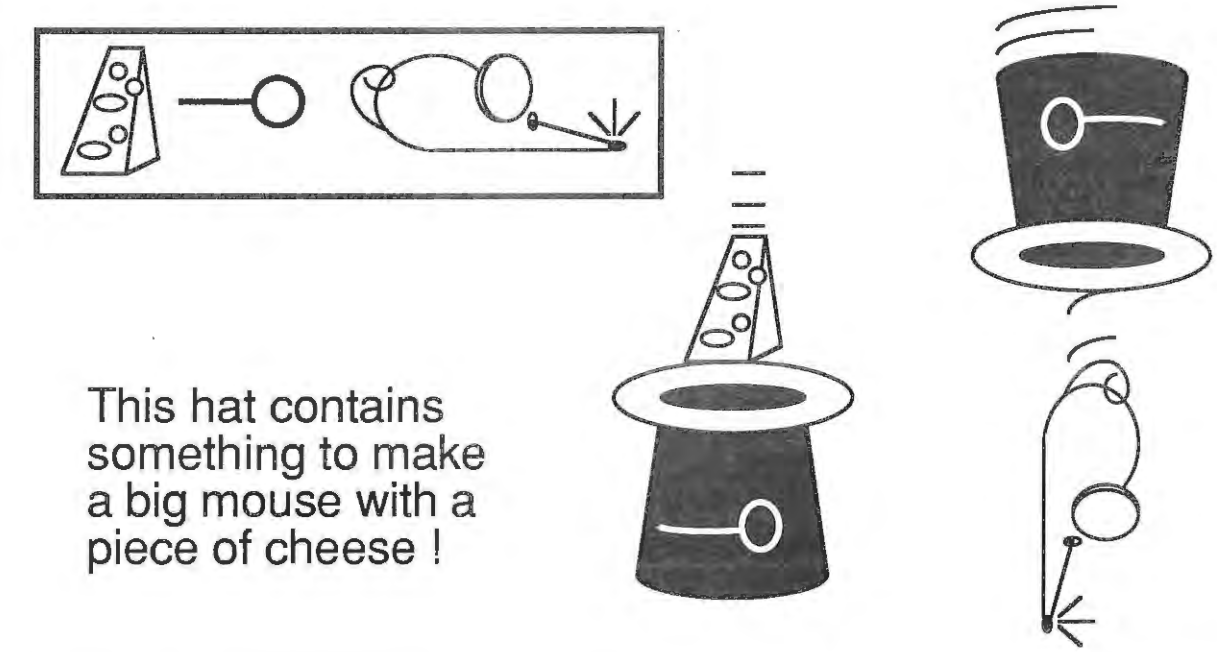


Figure 2.4: produit direct



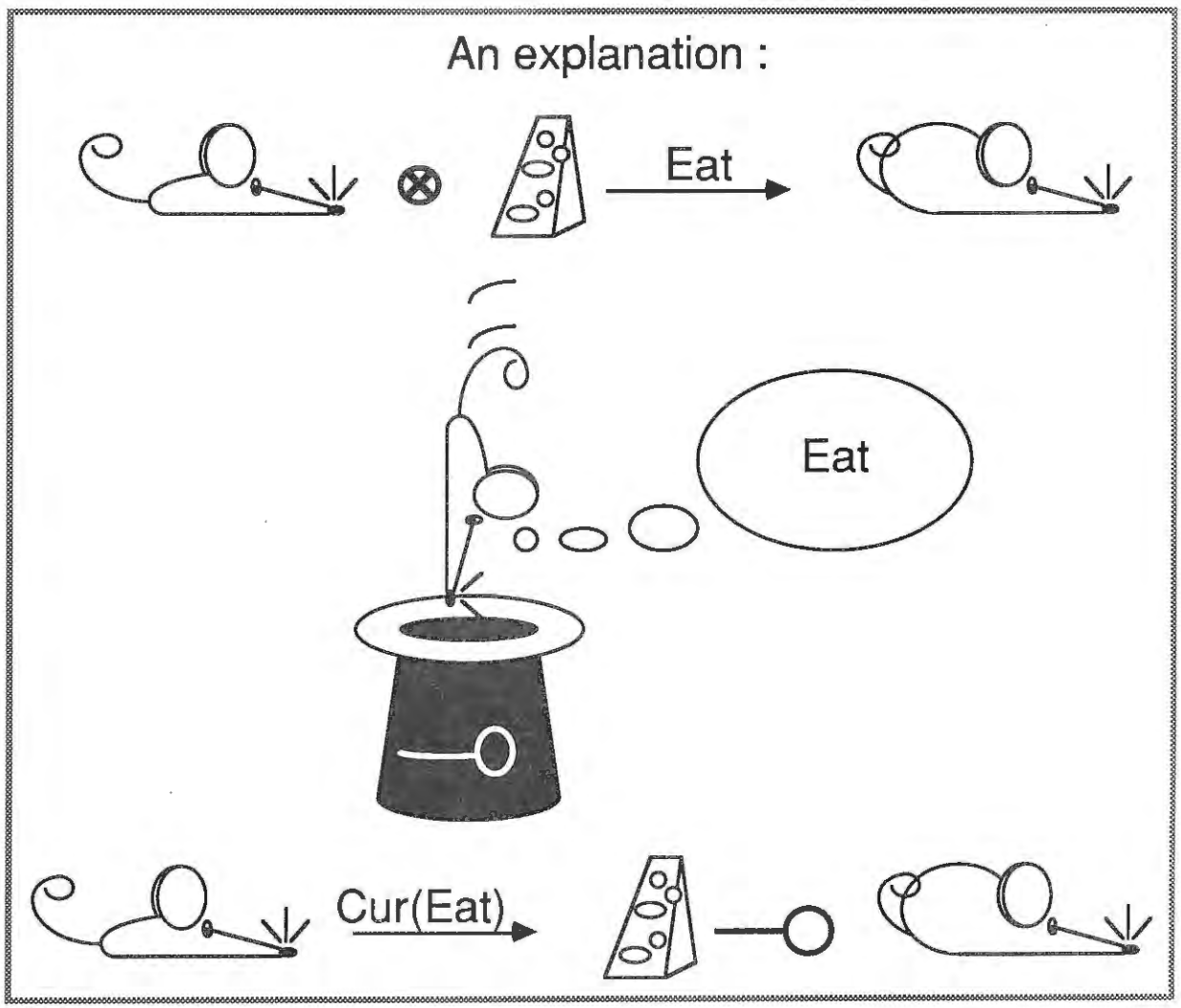


Figure 2.5: implication linéaire

2.2.2 Combinateurs Linéaires

Les *combinateurs linéaires* sont les opérateurs de base pour la théorie des *catégories monoïdales symétriques fermées* avec *produits* et *coproduits* finis (annexe D).

Compositeurs séquentiels:

$$\frac{\varphi : A \to B \quad \psi : B \to C}{\psi \circ \varphi : A \to C} \qquad \frac{}{\mathbf{Id} : A \to A}$$

Compositeurs parallèles:

$$\frac{\varphi : A \to B \quad \psi : C \to D}{\varphi \otimes \psi : A \otimes C \to B \otimes D} \qquad \frac{}{\mathbf{1} : \mathbf{1} \to \mathbf{1}}$$

Combinateurs d'arrangement:

$$\frac{}{\mathbf{Assl} : A \otimes (B \otimes C) \leftrightarrow (A \otimes B) \otimes C : \mathbf{Assr}}$$

$$\frac{}{\mathbf{Insl} : A \leftrightarrow \mathbf{1} \otimes A : \mathbf{Dell}} \qquad \frac{}{\mathbf{Exch} : A \otimes B \leftrightarrow B \otimes A : \mathbf{Exch}}$$

Ces combinateurs sont *reversibles* (on écrit $\varphi : A \leftrightarrow B : \psi$ pour $\varphi : A \to B$ et $\psi : B \to A$). Les noms sont des sigles: **Assl** (**Ass**ociate **l**eft), **Assr** (**Ass**ociate **r**ight), **Insl** (**In**sert **l**eft), **Dell** (**Del**ete left) et **Exch** (**Exch**ange).

Combinateurs logiques:

$$\frac{\varphi : A \otimes B \to C}{\mathbf{Cur}(\varphi) : A \to B \multimap C} \qquad \frac{}{\mathbf{App} : (A \multimap B) \otimes A \to B}$$

$$\frac{\varphi : A \to B \quad \psi : A \to C}{\mathbf{Pair}(\varphi, \psi) : A \to B \,\&\, C} \qquad \frac{}{\mathbf{Fst} : A \,\&\, B \to A} \qquad \frac{}{\mathbf{Snd} : A \,\&\, B \to A} \qquad \frac{}{\mathbf{Void} : A \to \mathbf{t}}$$

$$\frac{}{\mathbf{Inl} : A \to A \oplus B} \qquad \frac{}{\mathbf{Inr} : B \to A \oplus B} \qquad \frac{\varphi : A \to C \quad \psi : B \to C}{\mathbf{Case}(\varphi, \psi) : A \oplus B \to C} \qquad \frac{}{\mathbf{Can} : \mathbf{0} \to A}$$

Proposition 4 *Les deux formalismes (Calcul des Séquents et Logique Catégorique Combinatoire) sont équivalents:*

Pour tout combinateur $A \to B$ on a une preuve de $A \vdash B$, et pour toute preuve de $A_1, \ldots, A_n \vdash B$ on a un combinateur $A_1 \otimes \ldots \otimes A_n \to B$.

La démonstration est presque immédiate, les seuls cas délicats étant:

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C} \qquad \frac{}{\Gamma, \mathbf{0} \vdash A}$$

On utilise l'*implication linéaire* pour *envoyer* Γ à droite.

Par exemple, la preuve de $A \otimes (B \oplus C) \vdash (A \otimes B) \oplus (A \otimes C)$ correspond au combinateur suivant:

$$\mathbf{App} \circ (\mathbf{Case}(\mathbf{Cur}(\mathbf{Inl} \circ \mathbf{Exch}), \mathbf{Cur}(\mathbf{Inr} \circ \mathbf{Exch})) \otimes \mathbf{Id}) \circ \mathbf{Exch} : A \otimes (B \oplus C) \to (A \otimes B) \oplus (A \otimes C)$$

2.2.3 Les produits

Un *produit tensoriel* est évidemment plus *faible* qu'un *produit cartésien*. Le point fondamental est que $\otimes$ a une fermeture $(\multimap)$, mais pas le *produit cartésien* (ou *direct*) $\&$. En fait, $A \otimes B$ peut être considéré comme un type des couples *stricts*, et $A \& B$ comme un type des couples *paresseux*. En effet, **Fst** et **Snd** ne sont pas stricts par rapport à leurs arguments (seul l'un d'entre eux est utilisé), mais l'application linéaire **App** a besoin à la fois de la fonction et de son argument.

2.3 La Modalité

Comme dans le cas intuitionniste (par exemple, les *types de donnée concrets* de ML), on peut définir des *types de donnée récursifs*.

Toute définition récursive n'est pas raisonnable. Ainsi, l'*infinité actuelle* (figure 2.6) n'est pas réaliste, car elle suppose une *mémoire infinie à priori*.

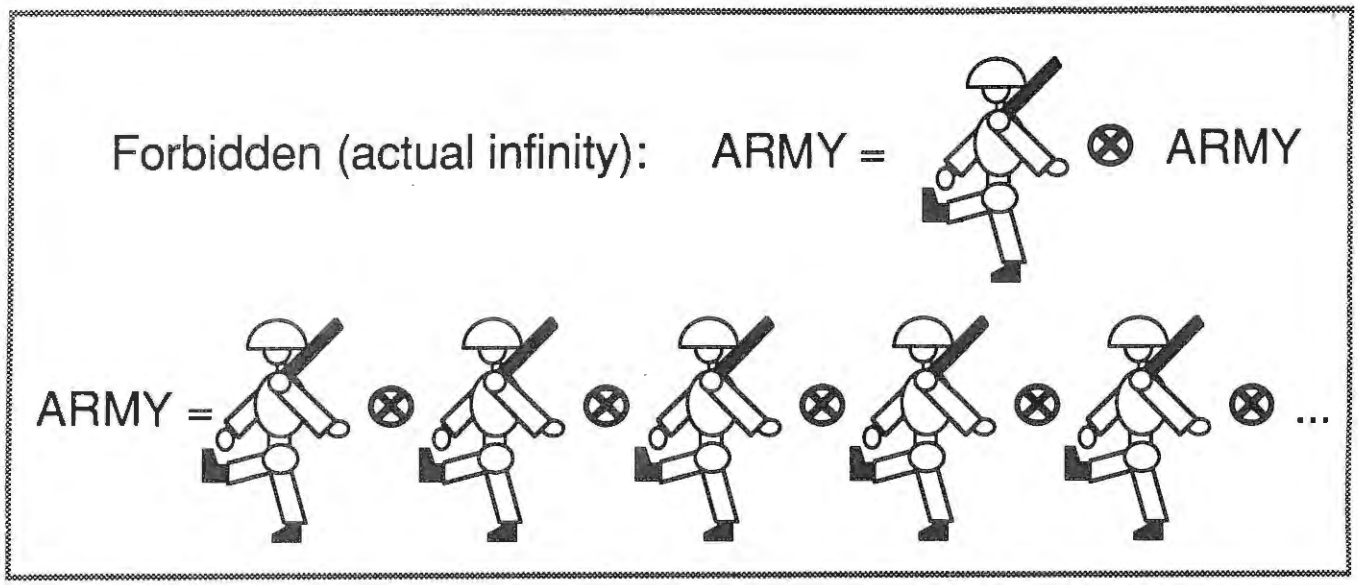


Figure 2.6: infinité actuelle

2.3.1 Types de donnée inductifs

On a l'équation habituelle définissant le type des entiers naturels:

$$\mathbf{Nat} = \mathbf{1} \oplus \mathbf{Nat}$$

C'est tout à fait suffisant pour un langage de programmation avec *récursion générale*, mais pour un système logique, on a besoin d'un opérateur explicite d'*itération primitive*. **Nat** est alors caractérisé par les règles suivantes:

$$\frac{}{\mathbf{Zero} : \mathbf{1} \to \mathbf{Nat}} \qquad \frac{}{\mathbf{Succ} : \mathbf{Nat} \to \mathbf{Nat}} \qquad \frac{\varphi : \mathbf{1} \to A \quad \psi : A \to A}{\mathbf{Iter}(\varphi, \psi) : \mathbf{Nat} \to A}$$

Le lecteur peut retrouver lui-même les règles explicites pour les *types de donnée inductifs* suivants:

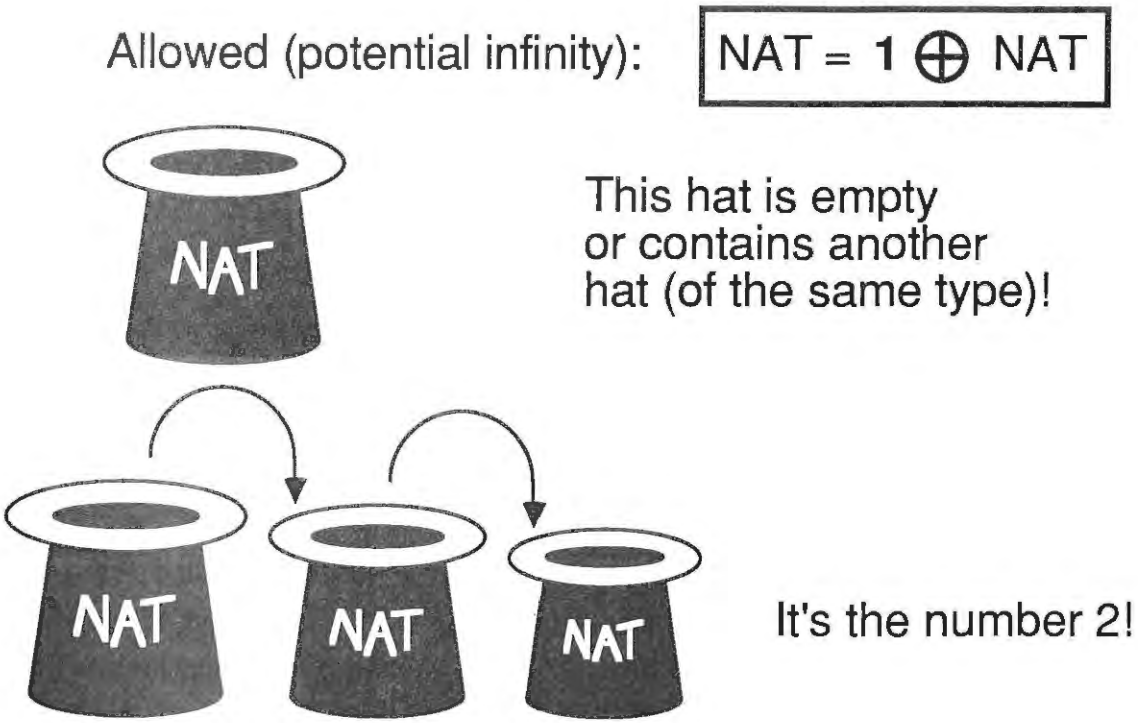


Figure 2.7: entiers naturels

$$\mathbf{List}\ A = 1 \oplus (A \otimes \mathbf{List}\ A)$$

$$\mathbf{Tree}\ A = A \oplus 1 \oplus (\mathbf{Tree}\ A \otimes \mathbf{Tree}\ A)$$

2.3.2 Bien sûr!

Si on remplace $\oplus$ par & dans les équations précédentes, on obtient la notion (duale) de *type de donnée projectif*. La *modalité* $!A$ (lire "bien sûr A") peut être caractérisée par l'équation suivante (figure 2.8):

$$!A = A \,\&\, 1 \,\&\, (!A \otimes !A)$$

Les règles correspondantes sont:

$$\frac{\varphi : A \to B \quad \varepsilon : A \to 1 \quad \delta : A \to A \otimes A}{\mathbf{Make}(\varphi, \varepsilon, \delta) : A \to !B}$$

$$\frac{}{\mathbf{Read} : !A \to A} \qquad \frac{}{\mathbf{Kill} : !A \to 1} \qquad \frac{}{\mathbf{Dupl} : !A \to !A \otimes !A}$$

La modalité est un *artifice* pour retrouver ce qui est perdu avec la *Logique Linéaire*, à savoir la *contraction* et l'*affaiblissement*. La similitude avec **Tree** A est évidente, mais cette définition n'est pas complète (voir annexe E). Toutefois, cela suffira pour notre propos.

$!A$ est une *co-algèbre libre* sur A:

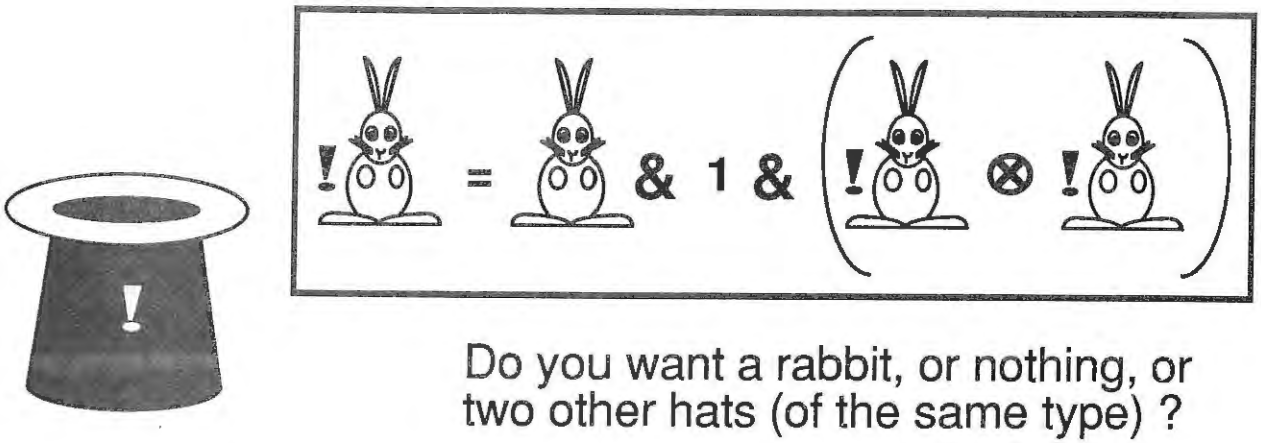


Figure 2.8: modalité

Proposition 5 *Tout* $\varphi : !A \to B$ *peut être relevé en un combinateur* $\mathbf{Lift}(\varphi) : !A \to !B$.

De plus, on a:

Proposition 6 *La modalité transforme le produit direct en produit tensoriel:*

$$!\mathbf{t} \equiv \mathbf{1} \qquad\qquad !(A \,\&\, B) \equiv !A \otimes !B$$

Pour démontrer cette proposition, il faut construire les combinateurs:

$$\mathbf{Subl} : !\mathbf{t} \leftrightarrow \mathbf{1} : \mathbf{Crys} \qquad\qquad \mathbf{Crac} : !(A \,\&\, B) \leftrightarrow !A \otimes !B : \mathbf{Glue}$$

Toutes ces constructions sont détaillées en annexe F.

2.3.3 La Logique Intuitionniste retrouvée

Avec la modalité *bien sûr*, on peut retrouver le pouvoir expressif de la *Logique Intuitionniste*. Plus précisément, on a un *plongement* de la *Logique Intuitionniste* dans la *Logique Linéaire*. Ainsi, les connecteurs *linéaires* apparaissent comme plus primitifs que les connecteurs *intuitionnistes*.

Le *plongement* envoie une formule *intuitionniste* A sur une formule *linéaire* $|A|$:

- $|A| = A$ lorsque A est une formule atomique.
- $|A \wedge B| = |A| \,\&\, |B|$ et $|\top| = \mathbf{t}$.
- $|A \Rightarrow B| = !|A| \multimap |B|$.
- $|A \vee B| = !|A| \oplus !|B|$ et $|\bot| = \mathbf{0}$.

Théorème 8 *Une formule intuitionniste* A *est prouvable (en Logique Catégorique) si et seulement si sa traduction* $|A|$ *est prouvable (en Logique Linéaire Catégorique).*

Pour montrer que A est prouvable quand $|A|$ l'est, il suffit d'étendre la traduction de 2.1.3 ($!A$ est traduite exactement comme A).

Pour la réciproque, on a besoin du lemme suivant:

Lemme 3 *Pour tout combinateur catégorique $\varphi : A \to B$, il existe un combinateur linéaire $|\varphi| : !|A| \to |B|$.*

La démonstration par récurrence utilise les propositions 5 et 6. Par exemple, si $\varphi : A \to B$ donne $|\varphi| :!|A| \to |B|$, et $\psi : B \to C$ donne $|\psi| :!|B| \to |C|$, alors $|\psi \circ \varphi| = \psi \circ \mathbf{Lift}(\varphi) :!|A| \to |C|$.

Le théorème précédent signifie que la *Logique Linéaire* avec modalité a la puissance de la *Logique Intuitionniste*. Par suite, les langages fonctionnels peuvent être implantés sur la *Machine Linéaire Abstraite* (section 2.5). Toutefois, cette implantation n'est pas réaliste, à cause de la traduction des *combinateurs catégoriques* dans les *combinateurs linéaires*, qui génère un code énorme.

2.3.4 Règles du Calcul des Séquents pour la modalité

En fait, Girard considère la modalité comme un *nouveau connecteur* caractérisé par les règles suivantes:

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B} \qquad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}$$

$!\Gamma$ représente une suite $!A_1, \ldots, !A_n$ dans la dernière règle qui, apparemment, n'a pas de traduction simple en *Combinateurs Catégoriques*. D'ailleurs, nos règles ne s'expriment pas simplemement en *Calcul des Séquents*, mais notre modalité est plus forte (les règles de Girard sont dérivables). De plus, nous n'introduisons aucun nouveau concept (modulo récursion, la modalité est définissable à partir du *produit direct* et du *produit tensoriel*). On peut dire que notre modalité est une *implantation* de celle de Girard.

2.4 Théorie de l'Exécution

2.4.1 Combinateurs atomiques et primitifs

On part d'un graphe donné, dont les points sont appelés *types atomiques* et les flêches *combinateurs atomiques*. Par exemple, les *types atomiques* sont les *états*, et les *combinateurs atomiques* les *actions de base* de robots commandés par notre ordinateur (voir section 1.6).

Un produit tensoriel de types atomiques est un *type primitif*, et la composition (sequentielle et parallèle) de combinateurs atomiques et d'arrangement (voir section 2.2) est un *combinateur primitif*. Naturellement, les domaines et co-domaines des *combinateurs primitifs* sont des *types primitifs*.

Les effets des programmes seront des *combinateurs primitifs*.

2.4.2 Combinateur canoniques

Pour tout type A, on distingue une classe de combinateurs $\mu : X \to A$ (où X varie parmi les *types primitifs*) que l'on appelle *combinateurs canoniques* de type A[1].

- Si A est atomique, le seul *combinateur canonique* de type A est $\mathbf{Id} : A \to A$.
- Les *combinateurs canoniques* de type $A \otimes B$ sont les $\mu \otimes \nu : X \otimes Y \to A \otimes B$ où $\mu : X \to A$ et $\nu : Y \to B$ sont des *combinateurs canoniques*.
- Le seul *combinateur canonique* de type $\mathbf{1}$ est $\mathbf{1}$.

[1] Notez la similitude avec les *termes canoniques* de [Martinloef].

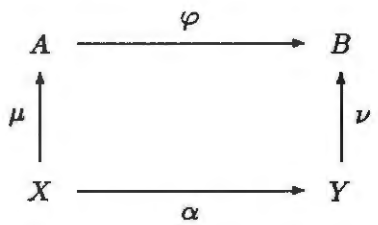
- Pour tout autre type A, les *combinateurs canoniques* de type A sont les $\gamma \circ \mu : X \to A$ où $\gamma : Y \to A$ est un *constructeur* (c'est à dire un combinateur de la forme $\mathbf{Cur}(\varphi)$, $\mathbf{Pair}(\varphi, \psi)$, **Void**, **Inl**, **Inr**) et $\mu : X \to Y$ est *canonique*[2].

  Par exemple, les *combinateurs canoniques* de type $A \multimap B$ sont les $\mathbf{Cur}(\varphi) \circ \mu : X \to A \multimap B$ où $\varphi : Y \otimes A \to B$, et $\mu : X \to Y$ est *canonique*.

Les données manipulées par les programmes seront des *combinateurs canoniques*.

2.4.3 Règles d'Evaluation

On définit une *relation d'exécution* $\mu \overset{\varphi}{\underset{\alpha}{\rightsquigarrow}} \nu$ lorsque $\varphi : A \to B$, $\mu : X \to A$, $\nu : Y \to B$ et $\alpha : X \to Y$, où μ, ν sont *canoniques* et α est *primitif*.



Cette relation va avoir la signification suivante: "Le code φ appliqué à la donnée μ rend la donnée ν, avec l'effet α". En fait, cette notion d'*effet* n'a pas été introduite délibérément: Elle vient naturellement lorsqu'on cherche à définir une sémantique opérationnelle des *Combinateurs Linéaires*.

La *relation d'exécution* est définie récursivement:

$$\frac{(\alpha \text{ est un combinateur atomique})}{\mathbf{Id} \overset{\alpha}{\rightsquigarrow} \mathbf{Id}} \qquad \frac{\mu \overset{\varphi}{\rightsquigarrow} \mu' \quad \mu' \overset{\psi}{\rightsquigarrow} \mu''}{\mu \overset{\psi \circ \varphi}{\rightsquigarrow} \mu''} \qquad \frac{}{\mu \overset{\mathbf{Id}}{\rightsquigarrow} \mu}$$

$$\frac{\mu \overset{\varphi}{\rightsquigarrow} \mu' \quad \nu \overset{\psi}{\rightsquigarrow} \nu'}{\mu \otimes \nu \overset{\varphi \otimes \psi}{\rightsquigarrow} \mu' \otimes \nu'} \qquad \frac{}{1 \overset{1}{\rightsquigarrow} 1}$$

$$\frac{}{\lambda \otimes (\mu \otimes \nu) \overset{\mathbf{Assl}}{\rightsquigarrow} (\lambda \otimes \mu) \otimes \nu} \qquad \frac{}{(\lambda \otimes \mu) \otimes \nu \overset{\mathbf{Assr}}{\rightsquigarrow} \lambda \otimes (\mu \otimes \nu)}$$

[2]Cette définition n'est pas bien fondée puisque Y n'est pas, en général, une sous-formule de A. Peu importe: Ce qui est défini récursivement, c'est la *relation* "μ est un *combinateur canonique* de type A".

$$\frac{}{\mu \overset{\mathbf{Insl}}{\rightsquigarrow} 1 \otimes \mu} \qquad \frac{}{1 \otimes \mu \overset{\mathbf{Dell}}{\rightsquigarrow} \mu} \qquad \frac{}{\mu \otimes \nu \overset{\mathbf{Exch}}{\rightsquigarrow} \nu \otimes \mu}$$

$$\frac{(\gamma \text{ est un constructeur})}{\mu \overset{\gamma}{\underset{\mathbf{Id}}{\rightsquigarrow}} \gamma \circ \mu} \qquad \frac{\lambda \otimes \mu \overset{\varphi}{\underset{\alpha}{\rightsquigarrow}} \nu}{(\mathbf{Cur}(\varphi) \circ \lambda) \otimes \mu \overset{\mathbf{App}}{\underset{\alpha}{\rightsquigarrow}} \nu}$$

$$\frac{\mu \overset{\varphi}{\underset{\alpha}{\rightsquigarrow}} \nu}{\mathbf{Pair}(\varphi,\psi) \circ \mu \overset{\mathbf{Fst}}{\underset{\alpha}{\rightsquigarrow}} \nu} \qquad \frac{\mu \overset{\psi}{\underset{\alpha}{\rightsquigarrow}} \nu}{\mathbf{Pair}(\varphi,\psi) \circ \mu \overset{\mathbf{Snd}}{\underset{\alpha}{\rightsquigarrow}} \nu}$$

$$\frac{\mu \overset{\varphi}{\underset{\alpha}{\rightsquigarrow}} \nu}{\mathbf{Inl} \circ \mu \overset{\mathbf{Case}(\varphi,\psi)}{\underset{\alpha}{\rightsquigarrow}} \nu} \qquad \frac{\mu \overset{\psi}{\underset{\alpha}{\rightsquigarrow}} \nu}{\mathbf{Inr} \circ \mu \overset{\mathbf{Case}(\varphi,\psi)}{\underset{\alpha}{\rightsquigarrow}} \nu}$$

On vérifie aisément:

Proposition 7 *Si* $\mu \overset{\varphi}{\underset{\alpha}{\rightsquigarrow}} \nu$ *alors le carré correspondant commute (voir annexe D.3):*

$$\begin{array}{ccc} A & \xrightarrow{\ \varphi\ } & B \\ {\scriptstyle\mu}\uparrow & = & \uparrow{\scriptstyle\nu} \\ X & \xrightarrow[\ \alpha\ ]{} & Y \end{array}$$

Pour évaluer un *programme* φ sur une *donnée* μ, on applique les règles d'*exécution* en remontant, jusqu'à ce que l'on trouve une *donnée* ν et un *effet* α tels que $\mu \overset{\varphi}{\underset{\alpha}{\rightsquigarrow}} \nu$. Visiblement, ce mécanisme est déterministe, mais il n'est pas clair qu'il termine toujours. Le résultat fondamental est:

Théorème 9 *(terminaison)*

Pour tout $\varphi : A \to B$ *et* μ *combinateur canonique de type* A, *il existe un combinateur canonique* ν *de type* B *et un combinateur primitif* α *tels que* $\mu \overset{\varphi}{\underset{\alpha}{\rightsquigarrow}} \nu$.

La démonstration (annexe G) est analogue à celle du théorème 6.

ν et α sont uniquement déterminés par φ et μ. Comme ν est le *résultat* et α l'*effet*, nous faisons un abus de notation[3] en notant $\nu = \varphi\ \mu$ et $\alpha = \partial\ \varphi\ \mu$.

Il est possible de reformuler les *règles d'exécution* avec ces notations. Par exemple:

$$(\varphi \circ \psi)\ \mu = \varphi\ (\psi\ \mu) \qquad\qquad \partial\ (\varphi \circ \psi)\ \mu = \partial\ \varphi\ (\psi\ \mu) \circ \partial\ \psi\ \mu$$

2.5 La Machine Linéaire Abstraite

La *Machine Linéaire Abstraite* est une cousine de la *Machine Catégorique Abstraite* (section 1.4). En particulier elle utilise les mêmes ingrédients: *code*, *environnement* et *pile*. C'est une *implantation séquentielle* des *règles d'exécution* (section 2.4).

2.5.1 Environnement et code

L'environnement est la représentation d'un *combinateur canonique*:

- Le *combinateur canonique* $\mu \otimes \nu$ est représenté par un couple (u, v) (couple strict) et 1 par $()$.
- Le *combinateur canonique* $\gamma \circ \mu$ est représenté par une paire $\gamma \cdot u$ où γ est un morceau de code.

Le code est une liste d'instructions primitives. La *composition séquentielle* devient la concaténation (à l'envers), mais la *composition parallèle* doit être séquentialisée:

- $\varphi \circ \psi$ devient $\psi \mathbin{@} \varphi$ (concaténation) et **Id** devient [] (liste vide).
- $\varphi \otimes \psi$ $(= (\mathbf{Id} \otimes \psi) \circ (\varphi \otimes \mathbf{Id}))$ devient $[\mathbf{Split}] \mathbin{@} \varphi \mathbin{@} [\mathbf{Cons}; \mathbf{Xsplit}] \mathbin{@} \psi \mathbin{@} [\mathbf{Xcons}]$ et 1 $(= \mathbf{Id})$ devient [].

Split pousse la seconde composante de l'environnement sur la pile, et accède à la première, alors que **Cons** reconstruit le couple. **Xsplit** et **Xcons** sont les instructions symétriques. Bien sûr, la séquence [**Cons**; **Xsplit**] peut être abrégée en une instruction unique **Swap** qui échange l'environnement avec le sommet de la pile.

Pour les autres combinateurs, la traduction est immédiate. Par exemple, **Assl** devient [**Assl**].

La fin d'une liste d'instructions est marquée par **Return**.

2.5.2 Exécution

Maintenant, nous pouvons décrire le fonctionnement de la machine:

[3] Ici, le *combinateur* φ est considéré comme un *programme* acceptant une donnée de type A et retournant une donnée de type B. Par contre, les *combinateurs canoniques* μ et ν sont considérés comme des données de types A et B respectivement.

Machine Linéaire Abstraite					
Avant			Après		
code	environnement	pile	code	environnement	pile
Split :: C	(u, v)	S	C	u	$v :: S$
Cons :: C	u	$v :: S$	C	(u, v)	S
Xsplit :: C	(u, v)	S	C	v	$u :: S$
Xcons :: C	v	$u :: S$	C	(u, v)	S
Assl :: C	$(u, (v, w))$	S	C	$((u, v), w)$	S
Assr :: C	$((u, v), w)$	S	C	$(u, (v, w))$	S
Insl :: C	u	S	C	$((), u)$	S
Dell :: C	$((), u)$	S	C	u	S
Exch :: C	(u, v)	S	C	(v, u)	S
$\gamma :: C$	u	S	C	$\gamma \cdot u$	S
App :: C	$(\mathbf{Cur}(C') \cdot u, v)$	S	C'	(u, v)	$C :: S$
Fst :: C	$\mathbf{Pair}(C', C'') \cdot u$	S	C'	u	$C :: S$
Snd :: C	$\mathbf{Pair}(C', C'') \cdot u$	S	C''	u	$C :: S$
$\mathbf{Case}(C', C'')$ :: C	$\mathbf{Inl} \cdot u$	S	C'	u	$C :: S$
$\mathbf{Case}(C', C'')$ :: C	$\mathbf{Inr} \cdot u$	S	C''	u	$C :: S$
Return :: C	u	$C' :: S$	C'	u	S

2.5.3 Extensions possibles

Il est bien sûr possible d'ajouter les *entiers machine*, avec des *instructions primitives* pour l'arithmétique, par exemple:

Avant		Après	
code	environnement	code	environnement
$\mathbf{Num}(p)$ :: C	$()$	C	p
$+ :: C$	(p, q)	C	$p + q$

En fait, à cause des *constraintes linéaires*, il faudra aussi des instructions primitives pour *dupliquer* ou *effacer* ces entiers (voir 3.1.5 et 3.3.1).

On peut aussi ajouter des instructions primitives pour les *effets* externes (*entrées/sorties*):

Avant		Après		
code	environnement	code	environnement	effet
Input :: C	$()$	C	c	c est envoyé par le clavier
Output :: C	c	C	$()$	c est envoyé à l'écran

Ces *effets* (en particulier les *sorties*) sont semblables aux *combinateurs atomiques* considérés pour le modèle théorique de la section 2.4.

2.5.4 Exemples d'exécution

Si α et β sont des instructions avec de l'*effet* (par exemple, des sorties) le programme $\mathbf{Fst} \circ \mathbf{Pair}(\alpha, \beta)$ est exécuté sans l'effet de β:

instruction	environnement	pile	effet
Pair([α; **Return**], [β; **Return**])	**Id**		
Fst	**Pair**([α; **Return**], [β; **Return**]]) · **Id**		
α	**Id**	[**Return**]	α
	Id	[**Return**]	

Que se passe-t-il maintenant si on exécute le programme $\alpha \otimes \beta$?

instruction	environnement	pile	effet
Split	(**Id**, **Id**)		
α	**Id**	**Id**	α
Cons	**Id**	**Id**	
Xsplit	(**Id**, **Id**)		
β	**Id**	**Id**	β
Xcons	**Id**	**Id**	
	(**Id**, **Id**)		

Puisque nous avons choisi d'implanter la composition parallèle dans un certain ordre, α est exécuté avant β. Mais cet ordre n'est pas spécifié par le programme: On a une *alternance arbitraire*!

En fait, si T est le type (atomique) représentant un *terminal*[4], $\alpha \otimes \beta$ est de type $T \otimes T \to T \otimes T$. Ce programme a besoin de *deux* terminaux!

En pratique, on remplace la donnée **Id** (qui ne contient aucune information) par l'adresse (ou port) du terminal:

instruction	environnement	pile	effet
Split	$(port_1, port_2)$		
α	$port_1$	$port_2$	α sur le $port_1$
Cons	$port_1$	$port_2$	
Xsplit	$(port_1, port_2)$		
β	$port_2$	$port_1$	β sur le $port_2$
Xcons	$port_2$	$port_1$	
	$(port_1, port_2)$		

Il n'y a pas de problème puisque les effets sont exécutés sur des terminaux indépendants.

2.5.5 Code bouclé

Pour la modalité, il n'est pas besoin d'instruction spécifique. En effet, au niveau de l'implantation, on n'a aucune raison d'interdire le *code bouclé*. On utilise donc les définitions récursives:

$$!A = A \,\&\, (\mathbb{1} \,\&\, (!A \otimes !A))$$

$$\mathbf{Read} = \mathbf{Fst} : !A \to A \qquad \mathbf{Kill} = \mathbf{Fst} \circ \mathbf{Snd} : !A \to \mathbf{1} \qquad \mathbf{Dupl} = \mathbf{Snd} \circ \mathbf{Snd} : !A \to !A \otimes !A$$

[4]Une *console*, pas un *object terminal* dans une catégorie!

$$\mathbf{Make}(\varphi, \varepsilon, \delta) = \pi \text{ où } \pi = \mathbf{Pair}(\varphi, \mathbf{Pair}(\varepsilon, (\pi \otimes \pi) \circ \delta))$$

Ainsi, le combinateur $\mathbf{Make}(-,-,-)$ est compilé en *code bouclé* (figure 2.9).

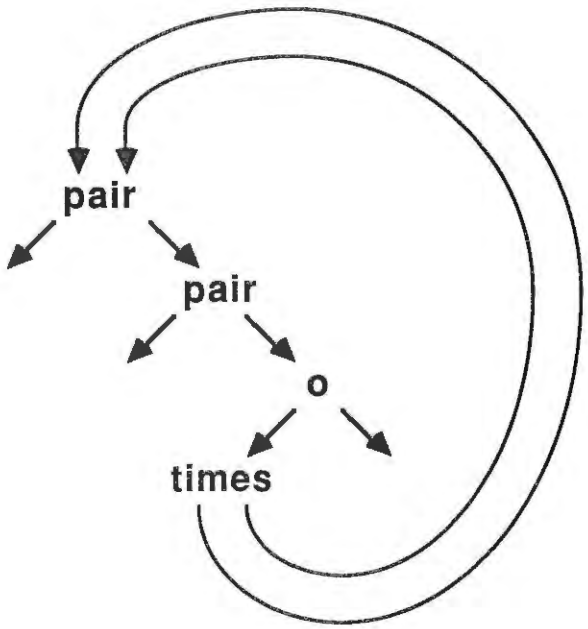


Figure 2.9: code bouclé

2.6 Avantages de la Machine Linéaire

2.6.1 Effets de bord

Rien n'empêche d'ajouter des instructions pour les *effets de bord*, c'est à dire la modification interne des données.

Par exemple, vous pouvez introduire un type prédéfini des *differences de listes* dont les objets sont représentés par un couple de pointeurs (la tête et la fin de la liste), et un instruction primitive pour la concaténation physique (comme *nconc*).

Cela, bien sûr, vous pouvez le faire aussi en *programmation fonctionnelle*, mais vous vous exposez alors aux *effets de bord pervers*, puisque le remplacement physique peut affecter d'autres parties de l'environnement. Rien de la sorte ne se produira ici, pour une raison très simple: Il n'y a *pas de partage*!

2.6.2 Récupération de mémoire

Considérons la *gestion de la mémoire* de la *Machine Catégorique Abstraite* (figure 2.10). L'environnement est un *graphe* dont les noeuds peuvent être *partagés* ou *orphelins*. On peut même avoir des structures isolées.

Parce qu'il y a des noeuds *partagés*, il est impossible de récupérer une cellule après un accès.

Parce qu'il y a des noeuds *orphelins*, il est utile de récupérer toutes les cellules inaccessibles lorsqu'il n'y en a plus de disponible (récupération de mémoire).

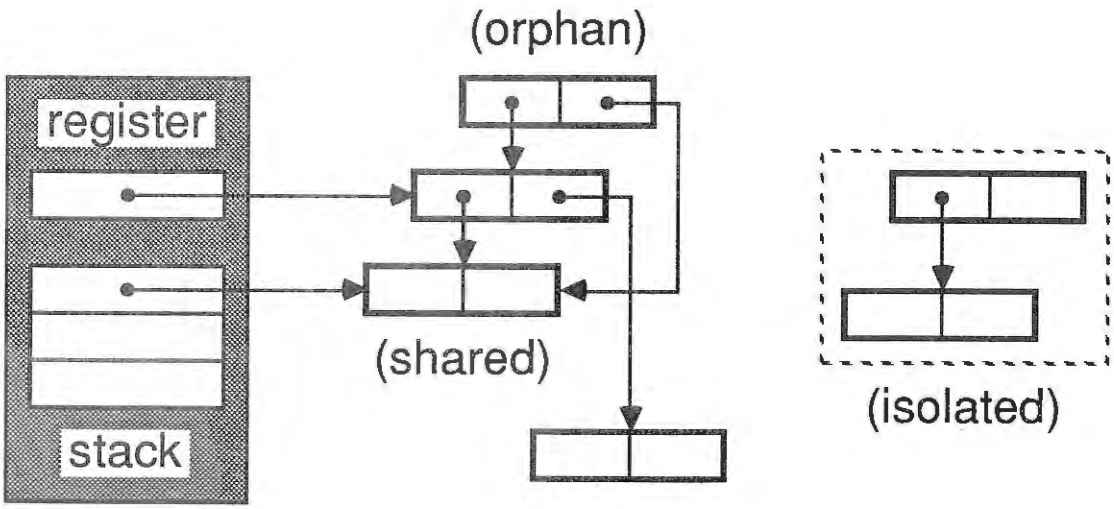


Figure 2.10: Machine Catégorique Abstraite

Avec la *Machine Linéaire Abstraite*, l'environnement est un *arbre* (graphe sans *partage* et sans *orphelin*). Plus précisément, il y a un *arbre* pour le registre et pour chaque composante de la pile (figure 2.11).

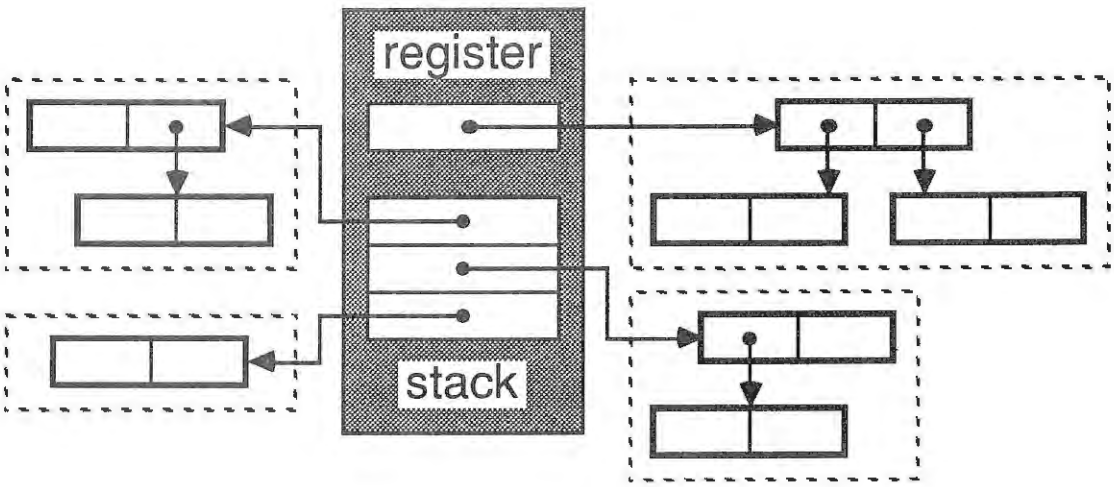


Figure 2.11: Machine Linéaire Abstraite

Comme il n'y a pas de *partage*, il est possible de récupérer une cellule après un accès. La figure 2.12 montre l'exemple typique de l'instruction **Xsplit**. Après l'accès, la cellule *retourne au paradis des cellules libres*! D'ailleurs, l'accès n'est pas le seul cas: Quand on exécute un destructeur $(\mathbf{App}, \mathbf{Fst}, \mathbf{Snd}, \mathbf{Case}(C', C''))$, on doit récupérer la cellule de la fermeture.

Comme il n'y a pas d'*orphelin*, la mémoire n'est pleine que dans les situations désespérées.

Finalement, la *Machine Linéaire Abstraite* est une machine *sans récupérateur de mémoire*. Plus précisément, la récupération de la mémoire fait partie du travail des instructions de base.

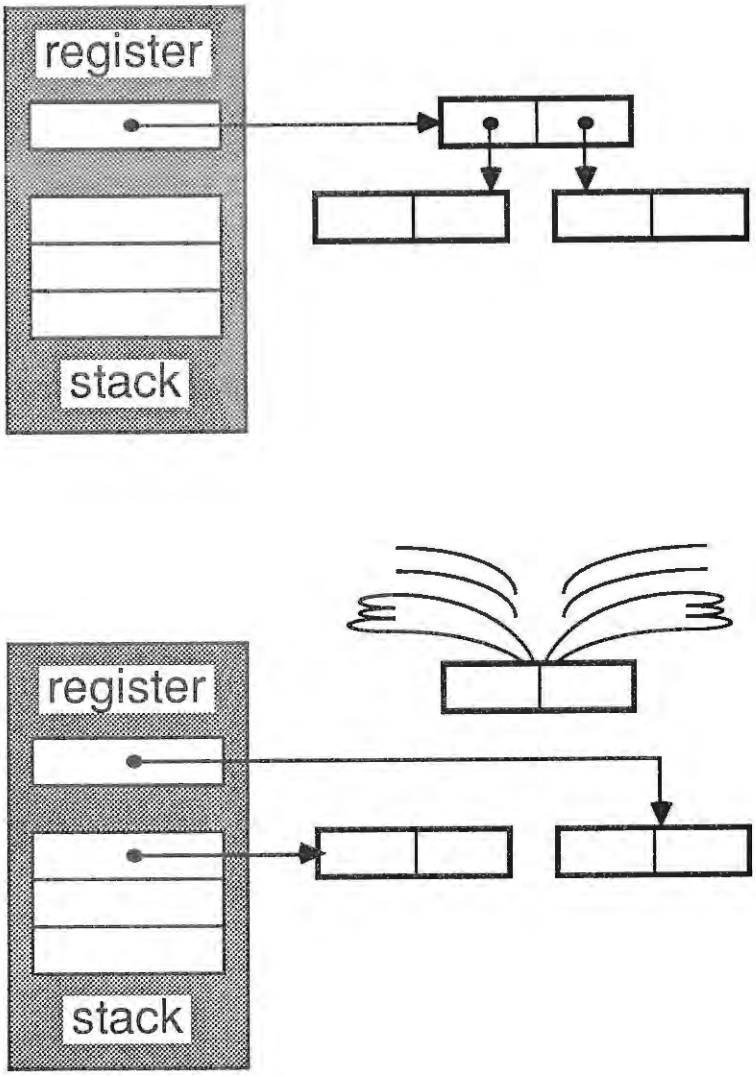


Figure 2.12: Machine Linéaire au travail

2.6.3 Paresse et effets de bord

Les connecteurs $\multimap$, $\&$, $\oplus$ peuvent être qualifiés de *paresseux*, car les données correspondantes *ne sont pas évaluées*: Un constructeur ($\mathbf{Cur}(C), \mathbf{Pair}(C', C''), \mathbf{Inl}$ ou $\mathbf{Inr}$) fabrique une *fermeture* qui est évaluée par un destructeur ($\mathbf{App}, \mathbf{Fst}, \mathbf{Snd}$ ou $\mathbf{Case}(C', C'')$).

- Une donnée de type $A \multimap B$ est un $\mathbf{Cur}(C) \cdot u$ où C correspond à un combinateur $X \otimes A \to B$ et u est l'environnement de type X.
- Une donnée de type $A \,\&\, B$ est $\mathbf{Pair}(C', C'') \cdot u$ où C' et C'' correspondent à des combinateurs $X \to A$ et $X \to B$ et u est l'environnement de type X.
- Une donnée de type $A \oplus B$ est un *constructeur* (**Inl** ou **Inr**) avec une donnée de type A ou B: c'est la représentation usuelle des coproduits.

$A \otimes B$ est le type des paires *strictes*, et $A \,\&\, B$ celui des paires *paresseuses*. Bien sûr, ils correspondent à des utilisations différentes: Les structures *strictes* sont bien adaptées à la *permutation* (par exemple le tri), tandis que les *paresseuses* sont bien adaptées à l'*accès* (menus).

En bref, l'évaluation stricte et l'évaluation paresseuse cohabitent harmonieusement dans un même système de types! Notre stratégie d'évaluation assure qu'aucun calcul n'est effectué s'il n'est pas nécessaire.

2.7 λ-Calcul

2.7.1 Contraintes Linéaires

L'absence de *contraction* et d'*affaiblissement* en *Calcul des Séquents* correspond à des *contraintes linéaires* en *λ-Calcul*.

Nous utilisons le *λ-Calcul* avec *paires* explicites et *conditionnelle*:

$$
\begin{array}{ll}
\textit{term}: & \textit{variable} \\
& \textit{constant} \\
& \textit{term}\,,\,\textit{term} \\
& \lambda\ \textit{pattern}\ .\ \textit{term} \\
& \textit{term}\ \textit{term} \\
& \textbf{if}\ \textit{term}\ \textbf{then}\ \textit{term}\ \textbf{else}\ \textit{term} \\
\\
\textit{pattern}: & \textit{variable} \\
& () \\
& \textit{pattern}\,,\,\textit{pattern}
\end{array}
$$

Dans un terme clos, une variable doit apparaître exactement *deux fois*: une fois dans un *motif* (pattern), où elle est liée, et une fois dans la portée de ce motif. La conditionnelle suit une règle spéciale, car les deux termes de l'alternative doivent *partager* l'environnement.

- $\mathcal{F}ree\,(x) = \{x\}$ (x variable)
- $\mathcal{F}ree\,(c) = \emptyset$ (c constante)
- $\mathcal{F}ree\,(M, N) = \mathcal{F}ree\,(M) \cup \mathcal{F}ree\,(N)$ ($\mathcal{F}ree\,(M)$ et $\mathcal{F}ree\,(N)$ doivent être disjoints)
- $\mathcal{F}ree\,(\lambda\, P\,.\,M) = \mathcal{F}ree\,(M) - \mathcal{F}ree\,(P)$ ($\mathcal{F}ree\,(P)$ doit être inclus dans $\mathcal{F}ree\,(M)$)
- $\mathcal{F}ree\,(M\ N) = \mathcal{F}ree\,(M) \cup \mathcal{F}ree\,(N)$ ($\mathcal{F}ree\,(M)$ et $\mathcal{F}ree\,(N)$ doivent être disjoints)

- $\mathcal{F}ree\,(\textbf{if}\,M\,\textbf{then}\,N\,\textbf{else}\,N') = \mathcal{F}ree\,(M) \cup \mathcal{F}ree\,(N)$ ($\mathcal{F}ree\,(M)$ et $\mathcal{F}ree\,(N)$ doivent être disjoints, $\mathcal{F}ree\,(N)$ et $\mathcal{F}ree\,(N')$ doivent être égaux)

Par exemple $\lambda\, x\,.\,x$ est correct, mais $\lambda\, x\,.\,xx$ ne l'est pas (indépendamment des règles de typage).

2.7.2 Règles de typage

Les règles de typage sont essentiellement les mêmes que celles du *λ-Calcul* traditionnel. On n'a qu'à remplacer le produit cartésien par $\otimes$ et la flêche par $\multimap$. Le type booléen est $\mathbf{Bool} = \mathbf{1} \oplus \mathbf{1}$.

$$\frac{}{x : A \vdash x : A} \qquad \frac{x_i : A_i \vdash M : A \quad y_j : B_j \vdash N : B}{x_i : A_i, y_j : B_j \vdash \langle M, N\rangle : A \otimes B} \qquad \frac{}{\vdash () : \mathbf{1}}$$

$$\frac{x_i : A_i \vdash P : A \quad x_i : A_i, y_j : B_j \vdash M : B}{y_j : B_j \vdash \lambda P\,.\,M : A \multimap B} \qquad \frac{x_i : A_i \vdash M : A \multimap B \quad y_j : B_j \vdash N : A}{x_i : A_i, y_j : B_j \vdash M\,N : B}$$

$$\frac{}{\vdash \mathbf{true} : \mathbf{Bool}} \qquad \frac{}{\vdash \mathbf{false} : \mathbf{Bool}} \qquad \frac{x_i : A_i \vdash M : \mathbf{Bool} \quad y_j : B_j \vdash N : A \quad y_j : B_j \vdash N' : A}{x_i : A_i, y_j : B_j \vdash \textbf{if}\,M\,\textbf{then}\,N\,\textbf{else}\,N' : A}$$

On peut ajouter des règles pour le *produit direct* et la *somme directe*, mais cela n'enrichit pas fondamentalement le calcul, vu que les constructions correspondantes sont codables en *λ-Calcul Linéaire* pur (avec conditionnelle et récursion).

2.7.3 Un langage de programmation

Le *λ-Calcul Linéaire* pur est beaucoup plus simple que le *λ-Calcul* pur: la réduction termine toujours, en temps linéaire! Néanmoins, on peut ajouter des constructions recursives pour obtenir un *véritable* langage de programmation.

A titre d'exemple, nous donnons deux versions de *quicksort* (voir également [Lafont87]).

Le premier est un programme *fonctionnel* écrit en CAML (figure 2.13).

Le second est écrit dans la *version linéaire* de CAML (décrit dans la troisième partie de la thèse). On doit adapter l'algorithme aux contraintes linéaires: La fonction *compare* (comparaison non destructive) appliquée à un couple `(x,y)` retourne `(c,x,y)` où `c` est vrai lorsque `x < y`. Pour les *entiers*, on peut considérer que c'est une *primitive*.

2.8 Conclusion

Nous résumons ici notre vision de la *Logique Linéaire Intuitionniste* comparée à la *Logique Intuitionniste*. Le chapitre suivant est une mise en oeuvre de ces idées.

logique:	Logique Intuitionniste	Logique Linéaire Intuitionniste
règles structurelles:	échange, identité, coupure contraction, affaiblissement	échange, identité, coupure (pas de contraction, pas d'affaiblissement)
connecteurs:	$\wedge, \top, \Rightarrow, \vee, \bot$	$\otimes, \mathbf{1}, \multimap, \&, \mathbf{t}, \oplus, \mathbf{0}$ (modalité)
modèle catégorique:	catégorie cartésienne fermée (avec coproduits finis)	catégorie monoïdale symétrique fermée (avec produits et coproduits finis)
machine:	Machine Catégorique Abstraite	Machine Linéaire Abstraite
calcul:	λ-Calcul	λ-Calcul Linéaire

```
let rec append =

 function x::X,Y        -> x::append(X,Y)

 |        [],Y          -> Y ;;


let rec partition =

 function x::X,y        -> let (U,V) = partition(X,y) in
                           if x < y then (x::U,V) else (U,x::V)

 |        [],y          -> [],[] ;;


let rec quicksort =

 function x::X          -> let (U,V) = partition(X,x) in
                           append(quicksort U,x::quicksort V)

 |        []            -> [] ;;
```

Figure 2.13: quicksort fonctionnel

```
function partition y,[]   -> y,[],[]

                | y,x::X -> let c,x,y = compare(x,y) in
                            let y,U,V = partition(y,X) in
                            y,(if c then (x::U,V) else (U,x::V));;

function quicksort []   -> []

                | x::X -> let y,U,V = partition(x,X) in
                          append(quicksort U,y::quicksort V);;
```

Figure 2.14: quicksort linéaire

Chapitre 3

Un Langage Linéaire

LIVE est au *λ-Calcul Linéaire*, ce que CAML est au *λ-Calcul Ordinaire*.

Rappelons qu'en *λ-Calcul Linéaire*, on exige que chaque variable soit ***utilisée exactement une fois***. De ce fait, tous les termes (non typés) sont normalisables ... en temps linéaire! Ce calcul est donc particulièrement ***inexpressif***, et c'est pourquoi il n'a guère suscité l'intérêt des théoriciens. En fait, on peut retrouver l'expressivité d'un véritable langage de programmation si on introduit la *conditionnelle* et la *récursion*[1].

Imposer de telles contraintes à un langage de programmation releverait du pur *masochisme*[2] si on n'en tirait quelque avantage. Les motivations sont exposées dans le deuxième chapitre: La *fonctionnalité pure* est inconciliable avec l'*efficacité* et l'*interactivité*. Plutôt que d'abandonner les principes de la programmation *déclarative*, on *change les règles du jeu*:

Il n'est plus permis de partager, ou d'abandonner les données. Par contre, on a le droit de les modifier physiquement (ce qui permet une gestion *dynamique* de l'environnement, sans *récupérateur de mémoire*).

En appliquant ces nouvelles règles, on obtient un *langage* qui ressemble encore beaucoup à CAML, mais qui n'est plus un *langage fonctionnel*, car il possède des *caractéristiques procédurales*, tout en restant *purement déclaratif*.

Pour la syntaxe, on est resté le plus proche possible de CAML[3]. Comme il serait prétentieux de vouloir décrire complétement et formellement un *langage* qui est encore à l'état d'*ébauche*, on a deux niveaux de description:

- un langage restreint, le *noyau*, qui est complétement spécifié (section 3.2)
- un langage complet, avec tout le *sucre syntaxique*, pour avoir des exemples de programmes lisibles.

En annexe, on trouvera un *maquette d'implantation* (du *noyau*) en CAML, avec une traduction du *code linéaire* dans l'assembleur LLM3 [LELISP]. On peut également envisager:

- une implantation complète
- le *bootstrap*

[1] Il est bien connu que la *conditionnelle* et la *récursion* peuvent être *codées* en *λ-Calcul Pur*, ce qui n'est plus vrai dans le cas *linéaire*.

[2] Il y a même une certaine régression dans le passage de CAML à LIVE.

[3] Une exception toutefois: la notation unique pour les termes et les types. Ainsi (A, B) dénote le *produit tensoriel* (qui joue un rôle similaire au *produit cartésien* de CAML), et **scheme** $A \rightarrow B$ dénote l'*implication linéaire*.

- une implantation directe de la LAM
- la LAM microcodée
- des architectures spécialisées

3.1 Fonctions et variables

Les principales différences entre CAML et LIVE concernent les *fonctions* et les *variables*.

3.1.1 Les fonctions en CAML: routines et schémas

Jusqu'aux fondements de l'analyse moderne à la fin du siècle dernier, les *fonctions* n'étaient pas considérées comme des objets mathématiques ordinaires. De même, en programmation traditionnelle (non fonctionnelle), les fonctions ne sont pas des données, mais des programmes, autrement dit des *routines*.

En programmation fonctionnelle, la plupart des fonctions jouent en fait le rôle de *routine*. Ainsi on définit une fois pour toutes la fonction append:

```
#let rec append = function x::X,Y -> x::append(X,Y)
                         | [],Y   -> Y;;
Val append = - : ('a list & 'a list -> 'a list)

#append(["Twas"; "brillig"],["and"; "the"; "slithy"; "toves"]);;
["Twas"; "brillig"; "and"; "the"; "slithy"; "toves"] : string list

#append(["Did"; "gyre"; "and"; "gimble"],["in"; "the"; "wabe"]);;
["Did"; "gyre"; "and"; "gimble"; "in"; "the"; "wabe"] : string list
```

Toutefois, pour alléger l'environnement, on peut aussi utiliser des *fonctions locales*:

```
#let append(X,Y) = apprec X where rec apprec = function x::X -> x::apprec X
                                                      | []   -> Y;;
Val append = - : ('a list & 'a list -> 'a list)
```

Dans cet exemple, la fonction locale `apprec` est essentiellement une routine avec un paramètre *invisible* `Y`.

On utilise les routines pour construire de nouvelles routines:

```
#let rec reverse = function x::X -> append(reverse X,[x])
                          | []   -> [];;
Val reverse = - : ('a list -> 'a list)

#reverse ["L"; "I"; "V"; "E"];;
["E"; "V"; "I"; "L"] : string list
```

Comme le temps de calcul de `append(X,Y)` est proportionnel à la longueur de `X`, celui de `reverse X` est proportionnel au carré de la longueur de `X`.

Voici une définition de `reverse` qui utilise vraiment la *fonctionnalité* de CAML[4], avec un temps de calcul proportionnel à la longueur de `X`:

[4]Dans ce cas particulièrement simple, on a évidemment des algorithmes similaires (en fait plus efficaces) qui n'utilisent pas la *fonctionnalité*.

```
#let cons x X = x::X;;
Val cons = - : ('a -> 'a list -> 'a list)

#let reverse = let rec revrec = function x::X -> (revrec X) o (cons x)
                                       | []   -> I
  in function X -> revrec X [];;
Val reverse : ('a list -> 'a list)
```

Dans cet exemple, la fonction `revrec X` n'est pas une routine[5] (elle est utilisée une seule fois dans le programme): elle représente une *liste incomplète* qui ne sera instanciée que lors de l'application finale `revrec X []`.

On voit donc apparaître une nouvelle race de fonctions, les *schémas*, qui permettent de coder une forme d'évaluation retardée.

3.1.2 Les fonctions en LIVE

Avec les contraintes linéaires, la séparation entre *routines* et *schémas* est légalisée.

Comme les *routines* sont, de loin, les plus importantes pour un langage de programmation, on continue à les appeler *fonctions*:

```
#function append x::X,Y -> x::append(X,Y)
              | [],Y    -> Y;;
Function append ('a list,'a list) : 'a list

#append(["All"; "Mimsy"],["were"; "the"; "borogoves"]);;
["All"; "Mimsy"; "were"; "the"; "borogoves"] : string list
```

Il n'y a pas de *fonction locale*.

On peut aussi définir des *opérations* et des *constantes*. Une opération est simplement une fonction dont l'argument est un couple, et une constante est une fonction dont l'argument est (), mais on a une notation plus agréable:

```
#operation (x,y) scal (x',y') -> x*x' + y*y';;
Operation (int,int) scal (int,int) : int

#constant one -> 1,0;;
Constant one : int,int

#one scal one;;
1 : int
```

Notez que la constante one est définie une fois pour toutes, et utilisable à volonté, ce qui n'est pas possible avec une variable:

```
#let i = 0,1;;
Val i = 0,1 : int,int

#i scal i;;
unbound variable i
```

Dans ce dernier exemple, l'une des occurrences de la variable *i* est liée par la définition globale qui précède, mais l'autre ne peut plus être liée vu que la valeur a déjà été *attribuée* (pas de partage!).

[5] Ne pas confondre avec la fonction revrec, qui est une routine.

3.1.3 Les schémas en LIVE

Les schémas sont des données, avec une syntaxe analogue à celle des fonctions de CAML:

```
#scheme X -> "And"::X;;
- : scheme (string list) -> string list

#it on ["the"; "mome"; "raths"; "outgrabe"];;
["And"; "the"; "mome"; "raths"; "outgrabe"] : string list
```

Contrairement aux fonctions, on peut les définir localement. Par contre *on ne peut les utiliser qu'une fois.*

Il faut donc distinguer syntaxiquement l'*appel* d'une fonction (qui est toujours un identificateur) et l'*instanciation* d'un schéma.

On peut définir la composition et l'identité pour les schémas:

```
#operation f o g -> scheme x -> f on (g on x);;
Operation (scheme 'a -> 'b) o (scheme 'c -> 'a) : scheme 'c -> 'b

#constant I -> scheme x -> x;;
Constant I : scheme 'a -> 'a
```

Et voici la version rapide de reverse:

```
#function cons x -> scheme X -> x::X;;
Function cons 'a : scheme ('a list) -> 'a list

#function reverse X -> revrec X on []

    where revrec x::X -> (revrec X) o (cons x)
                | []   -> I;;
Function reverse ('a list) : 'a list
```

3.1.4 Variables et identificateurs

Pour avoir une compilation efficace en programmation fonctionnelle, on doit faire la distinction entre *variables globales* et *variables locales.* L'accès aux *variables globales* se fait à la compilation, au moyen d'une table de symboles, alors que l'accès aux *variables locales* se fait à l'exécution.

Souvent, on utilise des *mots* (par exemple `reverse`) pour les variables globales et des *lettres* (par exemple `X`) pour les variables locales (sauf les fonctions locales qui, on l'a vu, sont essentiellement des routines avec paramètres invisibles), mais il n'y a pas de distinction syntaxique entre variables globales et locales: on peut les utiliser exactement dans les mêmes contextes[6].

Avec les contraintes linéaires, la séparation syntaxique entre variables globales et locales est légalisée: On a les *identificateurs* (de fonction) d'une part, et les *variables* (proprement dites) d'autre part. Une fonction toute seule n'est pas un terme: Elle doit être *appliquée à un argument.*

Naturellement, pour vérifier qu'un terme satisfait les contraintes linéaires, on ne considère pas les occurrences d'*identificateurs.* C'est ce qui permet d'utiliser les *fonctions* plusieurs fois.

[6] C'est le *compilateur*, et non l'*analyseur syntaxique*, qui reconnaît les variables globales.

3.1.5 Variables arithmétiques

Les contraintes linéaires sont très génantes si l'on veut définir les fonctions arithmétiques les plus élémentaires, par exemple le *carré* ou la *factorielle*.

Rappelons qu'avec la *Machine Linéaire*, le partage des structures est interdit. Mais, si les entiers sont représentés par des *mots machine* (petits entiers), leur duplication n'entraîne aucun partage. Une remarque similaire s'applique à l'effacement. On aura donc des *instructions primitives* pour la *duplication* et l'*effacement* d'entier.

On introduit la duplication et l'effacement explicitement, au niveau des *motifs* (*patterns*):

```
#function square(x as y) -> x*y;;
Function square int : int

#function zero _ -> 0;;
Function zero int : int
```

Notez qu'une expression de la forme `x as y` ou `_` est nécessairement de type entier, sinon on pourrait dupliquer ou effacer n'importe quoi.

Dans la suite, on utilisera le même nom pour les variables dupliquées:

```
#function square(x as x) -> x*x;;
```

Nous pouvons maintenant définir la fonction factorielle:

```
#function fact(x as x) -> if x=0 then let _ = x in 1
                                 else let y as y = x in y*fact(y-1);;
Function fact int : int
```

En utilisant le filtrage sur les entiers, on a un programme plus lisible:

```
#function fact 0      -> 1
            | x as x -> x*fact(x-1);;
```

De toute façon, cette notation reste assez lourde. On peut également envisager un système d'indices de multiplicité:

```
#function fact 0   -> 1
            | x^2 -> x*fact(x-1);;
```

Le compilateur pourrait aussi calculer lui-même la multiplicité des variables. Dans ce cas on retrouverait une syntaxe similaire à CAML:

```
#function fact 0 -> 1
            | x -> x*fact(x-1);;
```

Rappelons toutefois que cette *entorse aux contraintes linéaires* n'est qu'apparente, puisqu'elle est fondamentalement liée au type de la variable.

3.1.6 Exemple

Nous reprenons ici l'exemple de la section 2.7, mais avec des *listes incomplètes*:

```
#function partition y,[]    -> y,[],[]

                  | y,x::X -> let c,x,y = compare(x,y) in
                             let y,U,V = partition(y,X) in
                             y,(if c then (x::U,V) else (U,x::V))

 and quicksort []    -> I

             | x::X -> scheme Y -> let y,U,V = partition(x,X) in
                                   quicksort U on (y::(quicksort V on Y))

 and compare(p as p,q as q) -> p<q,p,q;;
Function partition (int,(int list)) : int,(int list),(int list)
Function quicksort (int list) : scheme (int list) -> int list
Function compare (int,int) : bool,int,int

#quicksort [1;9;8;7] on [];;
[1;7;8;9] : int list
```

3.2 Syntaxe de LIVE

Nous donnons ici la syntaxe du *noyau* de LIVE. Les *types concrets* et les *menus* seront introduits dans la section 3.5.

3.2.1 Les types

Les types sont ceux du fragment *multiplicatif* de la *Logique Linéaire Intuitioniste* (section 2.1), auxquels on ajoute les types primitifs **Int** (entiers) et **Bool** (booléens)[7]. On a de plus le polymorphisme implicite de CAML, c'est à dire que les types ne sont pas nécessairement clos:

type : *variable*
type , *type* (produit tensoriel)
() (unité tensorielle)
Int
Bool
scheme *type* → *type* (implication linéaire)

3.2.2 Syntaxe abstraite des termes

Comme on l'a vu dans la section 3.1, la syntaxe des termes est proche de celle de CAML:

[7] L'introduction d'un type **String** (chaîne de caractères) est problématique. On peut le considérer comme un type atomique, en utilisant la zone statique de la mémoire (dans ce cas, on a le droit de *dupliquer* ou d'*effacer* les chaînes, mais pas de les *concaténer*). Mais on peut aussi représenter les chaînes comme des listes de caractères ...

term : *variable*
term , *term*
()
integer
boolean
term operation term (opération arithmétique)
term comparison term (comparaison arithmétique)
if *term* **then** *term* **else** *term*
let *pattern* = *term* **in** *term*
scheme *pattern* → *term* (abstraction d'un schéma)
term **on** *term* (instanciation d'un schéma)
identifier term (appel d'une fonction)

pattern : *variable*
pattern , *pattern*
()
-
variable **as** ... **as** *variable*

3.2.3 Contraintes linéaires et typage

Pour être correct, un terme doit satisfaire deux types de contraintes:

- les *contraintes linéaires*: Dans un terme clos, chaque variable doit apparaître essentiellement deux fois, un fois dans le motif (pattern) où elle est liée, et une fois dans la portée (scope) de ce motif. Dans la conditionnelle, les deux termes de l'alternative partagent l'environnement.
- les *contraintes de typage* (analogues à celles de CAML).

Les *contraintes linéaires* font que le terme est *compilable*, alors que les *contraintes de typage* assurent le bon comportement du code compilé[8].

On peut exprimer ces contraintes au moyen d'un système de règles d'inférence. Le séquent $x_1 : A_1, \ldots x_n : A_n \vdash u : B$ signifie que u est un terme de type B avec les variables libres x_1 de type A_1, ..., x_n de type A_n. On a une notation analogue $x_1 : A_1, \ldots x_n : A_n \Vdash p : B$ pour les motifs (patterns):

$$\frac{\Gamma, x : A, y : B, \Delta \vdash u : C}{\Gamma, y : B, x : A, \Delta \vdash u : C}\ (\text{échange}) \qquad \frac{}{x : A \vdash x : A}\ (\text{identité})$$

$$\frac{\Gamma \vdash u : A \quad \Delta \vdash v : B}{\Gamma, \Delta \vdash u, v : A, B}\ (\text{produit tensoriel}) \qquad \frac{}{\vdash () : ()}\ (\text{unité tensorielle})$$

$$\frac{}{\vdash 0 : \mathbf{Int}} \quad \ldots \quad \frac{}{\vdash \mathbf{true} : \mathbf{Bool}} \qquad \frac{}{\vdash \mathbf{false} : \mathbf{Bool}}$$

$$\frac{\Gamma \vdash u : \mathbf{Int} \quad \Delta \vdash v : \mathbf{Int}}{\Gamma, \Delta \vdash u + v : \mathbf{Int}} \quad \ldots \quad \frac{\Gamma \vdash u : \mathbf{Int} \quad \Delta \vdash v : \mathbf{Int}}{\Gamma, \Delta \vdash u < v : \mathbf{Bool}} \quad \ldots$$

[8] Il se peut fort bien qu'un terme *non typable* ait un bon comportement à l'exécution. Ainsi, il est possible de *coder* les *types concrets* dans le noyau de LIVE (non typé), en utilisant implicitement des types dépendants (annexe I).

$$\frac{\Gamma \vdash u : \mathbf{Bool} \quad \Delta \vdash v : A \quad \Delta \vdash w : A}{\Gamma, \Delta \vdash \mathbf{if}\ u\ \mathbf{then}\ v\ \mathbf{else}\ w : A}$$

$$\frac{\Gamma \Vdash p : A \quad \Delta \vdash u : A \quad \Gamma, \Theta \vdash v : B}{\Delta, \Theta \vdash \mathbf{let}\ p = u\ \mathbf{in}\ v : B}$$

$$\frac{\Gamma \Vdash p : A \quad \Gamma, \Delta \vdash u : B}{\Delta \vdash \mathbf{scheme}\ p \to u : \mathbf{scheme}\ A \to B} \qquad \frac{\Gamma \vdash u : \mathbf{scheme}\ A \to B \quad \Delta \vdash v : A}{\Gamma, \Delta \vdash u\ \mathbf{on}\ v : B}$$

$$\frac{\Gamma \vdash u : A \quad (f \text{ fonction de } A \text{ dans } B)}{\Gamma \vdash f\ u : B}$$

$$\frac{}{x : A \Vdash x : A} \qquad \frac{\Gamma \Vdash u : A \quad \Delta \Vdash v : B}{\Gamma, \Delta \Vdash u, v : A, B} \qquad \frac{}{\Vdash () : ()}$$

$$\frac{}{\Vdash \_ : \mathbf{Int}} \qquad \frac{\Gamma \Vdash u : \mathbf{Int} \quad \Delta \Vdash v : \mathbf{Int}}{\Gamma, \Delta \Vdash u\ \mathbf{as}\ v : \mathbf{Int}} \qquad \ldots$$

Pour éviter toute ambiguïté, il faut interdire la répétition d'un nom de variable dans le même environnement ($x : A, x : B \vdash x, x : A, B$), sauf dans le cas de la duplication ($x : \mathbf{Int}, x : \mathbf{Int} \Vdash x\ \mathbf{as}\ x : \mathbf{Int}$).

3.2.4 Fonctions

La définition d'une suite de fonctions (mutuellement récursives) est de la forme:

function *identifier pattern* $\to$ *term* **and** ... **and** *identifier pattern* $\to$ *term*

3.2.5 Synthèse des types

Comme pour CAML, il est possible de construire le type le plus général d'un terme (ou d'une fonction) en utilisant un algorithme d'unification.

3.3 Les Principes de la Compilation

3.3.1 La Machine Linéaire

Les principes de la *Machine Linéaire Abstraite* étant découverts (section 2.5), il fallait trouver un jeu d'instructions simple et élégant, adapté à la compilation de notre langage.

Rappelons que la machine possède un *registre* et une *pile*. Etant donnés un motif p et un terme u construits dans le même contexte Γ ($\Gamma \Vdash p$ et $\Gamma \vdash u$), il faut générer le code $[\![u]\!]_p$ qui *calcule la valeur de u dans l'environnement p*. Initialement, l'environnement est dans le registre. Après exécution du code, le registre contient la valeur, et la pile a retrouvé son état initial.

Parmi les instructions, on distingue les transpositions **Swap**, **Swaap** et **Sswap**, la *construction* et la *destruction* de *couple* **Cons** et **Split**, la *construction* et la *destruction* de *nil* **Push** et **Pop**, la

duplication et l'*effacement* d'*entier* **Copy** et **Erase**. Toutes ces instructions sont utilisées pour la *gestion* de l'environnement.

On a aussi des instructions pour *l'introduction* d'*entier* et de *booléen*, les *opérations et comparaisons arithmétiques*, le *branchement conditionnel* **Branch**$(-,-)$, la *construction* et l'*instanciation* de *schéma* **Cur**$(-)$ et **App**, l'*appel* de *fonction* **Call**$(-)$:

Machine Linéaire					
Avant			Après		
code	environnement	pile	code	environnement	pile
Swap $:: C$	u	$v :: S$	C	v	$u :: S$
Swaap $:: C$	u	$v :: w :: S$	C	w	$v :: u :: S$
Sswap $:: C$	u	$v :: w :: S$	C	u	$w :: v :: S$
Cons $:: C$	u	$v :: S$	C	(u, v)	S
Split $:: C$	(u, v)	S	C	u	$v :: S$
Push $:: C$	u	S	C	$()$	$u :: S$
Pop $:: C$	$()$	$u :: S$	C	u	S
$0 :: C$	u	S	C	0	$u :: S$
...					
true $:: C$	u	S	C	**true**	$u :: S$
false $:: C$	u	S	C	**false**	$u :: S$
Copy $:: C$	i	S	C	i	$i :: S$
Erase $:: C$	i	$u :: S$	C	u	S
$+ :: C$	u	$v :: S$	C	$u + v$	S
...					
$< :: C$	u	$v :: S$	C	$u < v$	S
...					
Branch(C', C'') $:: C$	**true**	$u :: S$	C'	u	$C :: S$
Branch(C', C'') $:: C$	**false**	$u :: S$	C''	u	$C :: S$
Cur(C') $:: C$	u	S	C	$(C' \cdot u)$	S
App $:: C$	$(C' \cdot u)$	$v :: S$	C'	(u, v)	$C :: S$
Call(C') $:: C$	u	S	C'	u	$C :: S$
Return $:: C$	u	$C' :: S$	C'	u	S

Il faut noter que **Split** est l'exact inverse de **Cons**: La cellule est immédiatement récupérée, et il n'y a pas d'encombrement de la mémoire. De même, **App** récupère la cellule de la fermeture $(C' \cdot u)$[9].

3.3.2 Le code de distribution

La principale difficulté est la compilation de la couple u, v, et plus généralement, de toutes les constructions binaires telles que les *opérations* et les *comparaisons* arithmétiques, l'*instanciation* et une partie de la *conditionnelle*.

A la différence de la *Machine Catégorique*, il n'y a pas de partage de l'environnement. Par contre, les *contraintes linéaires* font que le motif p peut se décomposer en deux parties: p_u pour u et p_v pour v.

On commence donc par le *code de distribution* $(split_v)$[10] qui pousse p_u sur la pile et garde p_v dans le registre. On se trouve alors dans un cas de figure similaire à la *Machine Catégorique* (on a seulement *décomposé* l'environnement au lieu de le *partager*):

[9] Dans l'implantation, on *recycle* la cellule de $(C' \cdot u)$ pour construire le couple (u, v).
[10] Ne pas confondre avec l'*instruction* **Split**.

- $[\![u, v]\!]_p = split_v \mathbin{@} [\![v]\!]_{p_v} \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![u]\!]_{p_u} \mathbin{@} [\mathbf{Cons}]$

Le code $split_v$ peut être assez complexe, car p_u et p_v risquent d'être mélangés à l'intérieur de p, comme dans $[\![x + z, y]\!]_{(x,y),z}$. On construit donc $split_v$ par récurrence sur p:

- Lorsque p est un couple, on commence par le décomposer avec l'*instruction* **Split**, on traite récursivement les deux composantes et on réassemble les morceaux avec **Cons**. On a besoin de la transposition **Swaap** (figure 3.1).
- Pour (), c'est **Push** (figure 3.2).
- Pour une variable x, il y a deux cas: **Push** si x est utilisée par u, et [**Push**; **Swap**] si x est utilisée par v (figure 3.3).
- Lorsque p est de type entier (par exemple x **as** y), on a le droit de le dupliquer avec **Copy**.

Naturellement, la distribution d'une variable est très inéquitable, vu que l'un reçoit tout, et l'autre ne reçoit rien!

Ainsi on a:

$[\![x + z, y]\!]_{(x,y)z} =$ [**Split**; **Split**; **Push**; **Swaap**; **Push**; **Swap**; **Swaap**; **Cons**; **Swaap**; **Cons**; **Swaap**; **Push**; **Swaap**; **Cons**; **Swaap**; **Cons**] @ $[\![y]\!]_{((),y),()}$ @ [**Swap**] @ $[\![x + z]\!]_{(z,()),z}$ @ [**Cons**].

Si on obtient un code si long, c'est qu'on a utilisé une méthode trop systématique, sans tenir compte des cas particuliers. Avec les *optimisations* de la section 3.4, on a une traduction nettement plus raisonnable:

$[\![x + z, y]\!]_{(z,y),z} =$ [**Split**; **Split**; **Sswap**; **Cons**] @ $[\![x + z]\!]_{x,z}$ @ [**Swap**] @ $[\![y]\!]_y$ @ [**Swap**; **Cons**].

Le résultat se réduira finalement à:

[**Split**; **Split**; **Sswap**; +; **Cons**][11].

3.3.3 Le code de récupération et l'accès

De même, pour compiler () (ou n'importe quelle constante) dans un environnement p qui, forcément, ne contient aucune variable, on utilise le *code de récupération* (*pop*):

- $[\![()]\!]_p = pop \mathbin{@} [\mathbf{Push}]$

Cette fois, la définition par récurrence ne pose aucune difficulté:

- Lorsque p est un couple, on commence par le décomposer avec l'*instruction* **Split** et on récupère récursivement les deux composantes (figure 3.4).
- Pour () , on a l'instruction **Pop**.
- Et pour _ , on a **Erase**.

Quand on compile une variable, l'environnement ne contient plus que cette variable. Il suffit donc d'*élaguer* l'environnement, c'est à dire de récupèrer tous les () et _ :

- $[\![x]\!]_p = prune$

La définition de *prune* utilise *pop*. En fait, les optimisations font que, dans la plupart des cas, l'environnement est déjà élagué, ce qui permet d'affirmer que le temps d'accès à une variable est optimal (0)!

[11] C'est mieux que la CAM!

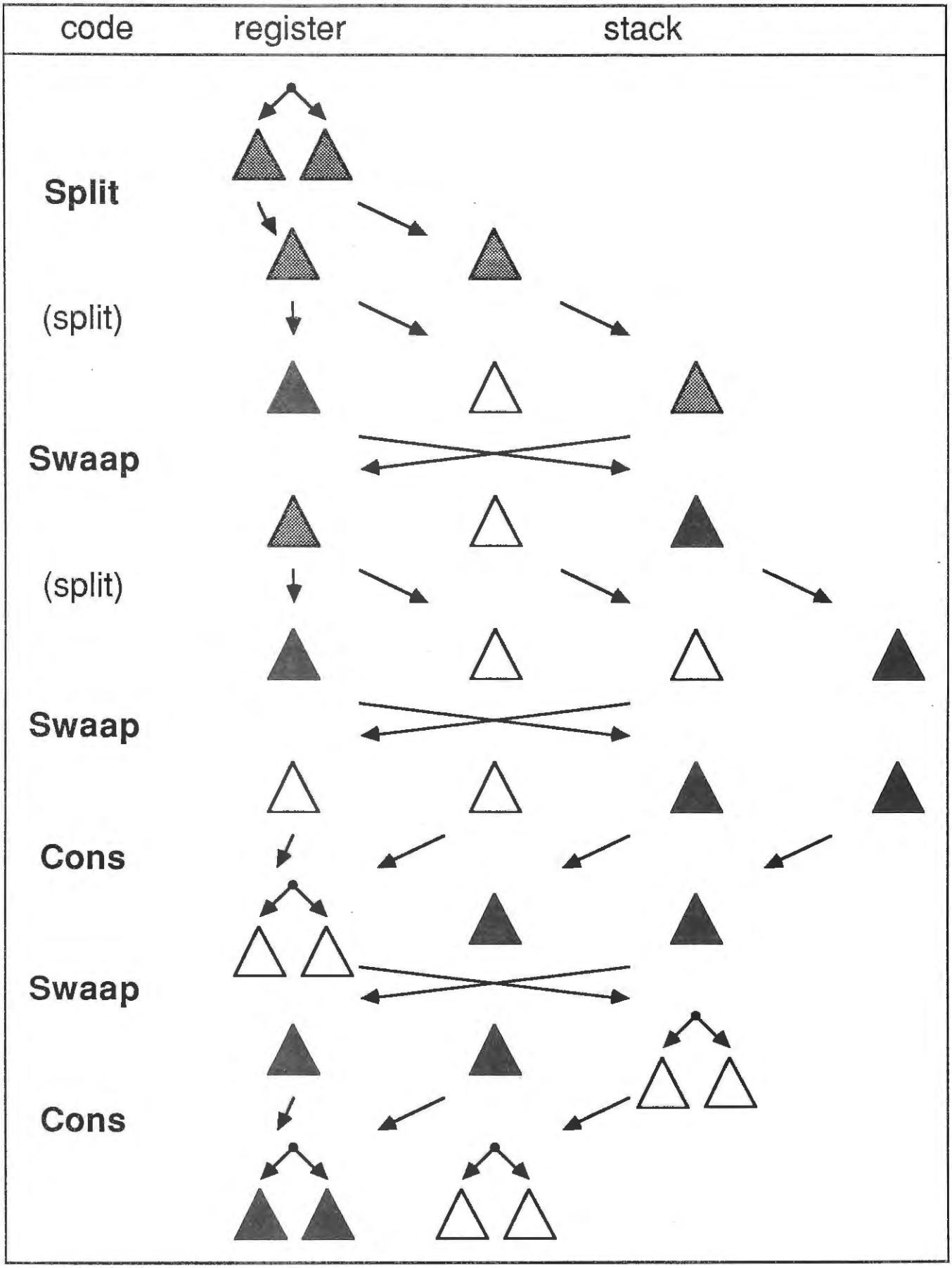


Figure 3.1: distribution d'un couple

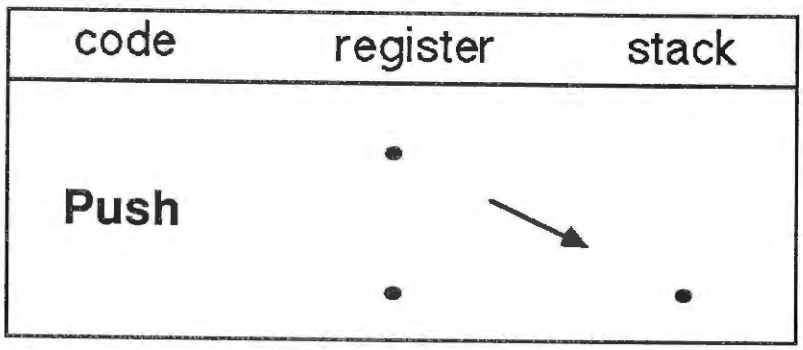
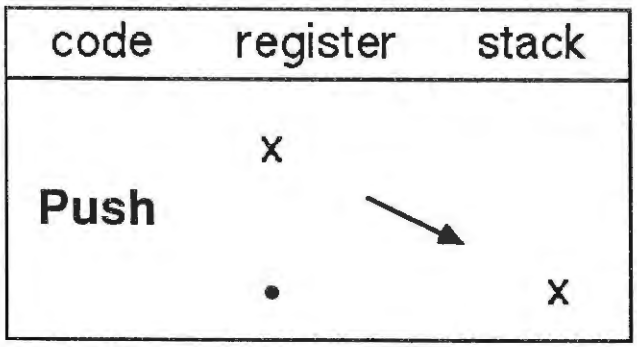


Figure 3.2: distribution de nil



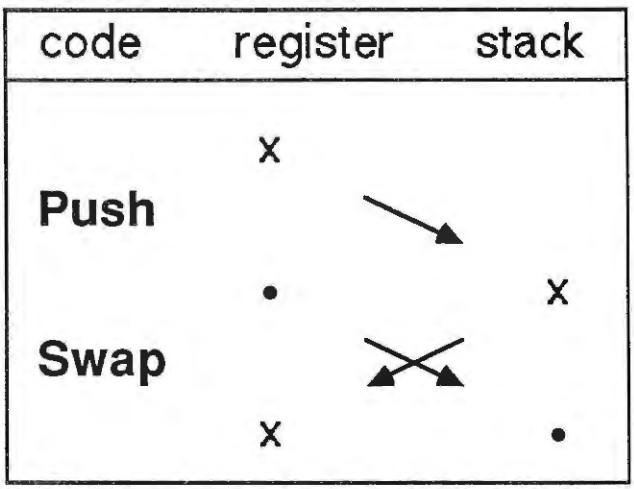


Figure 3.3: distribution d'une variable

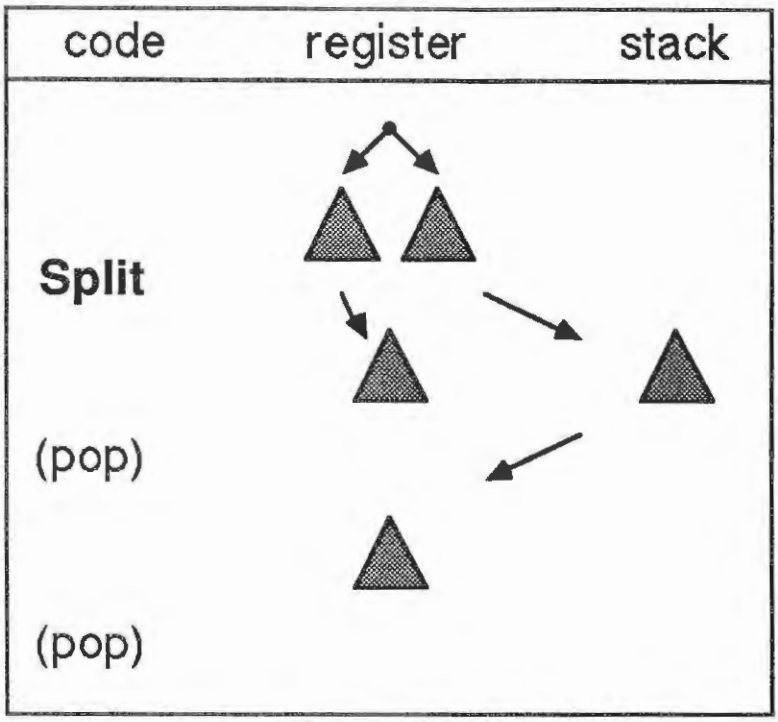


Figure 3.4: récupération d'un couple

3.3.4 Compilation des autres expressions

Une fois les difficultés de gestion résolues, la compilation est analogue à celle de CAML (section 1.4):

- $[\![0]\!]_p = pop \mathbin{@} [0]$
- ...
- $[\![\mathbf{true}]\!]_p = pop \mathbin{@} [\mathbf{true}]$
- $[\![\mathbf{false}]\!]_p = pop \mathbin{@} [\mathbf{false}]$
- $[\![u + v]\!]_p = split_v \mathbin{@} [\![v]\!]_{p_v} \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![u]\!]_{p_u} \mathbin{@} [+]$
- ...
- $[\![u < v]\!]_p = split_v \mathbin{@} [\![v]\!]_{p_v} \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![u]\!]_{p_u} \mathbin{@} [<]$
- ...
- $[\![\mathbf{if}\ u\ \mathbf{then}\ v\ \mathbf{else}\ w]\!]_p = split_u \mathbin{@} [\![u]\!]_{p_u} \mathbin{@} [\mathbf{Branch}([\![v]\!]_{p_v}, [\![w]\!]_{p_v})]$
- $[\![\mathbf{let}\ q\ =\ u\ \mathbf{in}\ v]\!]_p = split_u \mathbin{@} [\![u]\!]_{p_u} \mathbin{@} [\mathbf{Swap}; \mathbf{Cons}] \mathbin{@} [\![v]\!]_{p',q}$ (p' est le complément de p_u)
- $[\![\mathbf{scheme}\, q \rightarrow u]\!]_p = [\mathbf{Cur}([\![u]\!]_{p,q} \mathbin{@} [\mathbf{Return}])]$
- $[\![u\ \mathbf{on}\ v]\!]_p = split_v \mathbin{@} [\![v]\!]_{p_v} \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![u]\!]_{p_u} \mathbin{@} [\mathbf{App}]$
- $[\![f\ u]\!]_p = [\![u]\!]_p \mathbin{@} [\mathbf{Call}(C)]$ (C est le code de la fonction f).

3.4 Les Optimisations

Si on appliquait mécaniquement les principes de la section 3.3, le compilateur générerait beaucoup trop de *code de gestion*, notamment pour la *distribution* de l'environnement.

C'est pourquoi il est absolument nécessaire d'*optimiser* le code généré, *pendant* ou *après* la compilation.

Dans le premier cas, il s'agit de détecter, en cours de compilation, des *cas particuliers* qui se traduisent plus simplement (optimisations *globales*). Dans le second, il s'agit de simplifier le code au moyen d'un système de *règles de reécriture* (optimisations *locales*).

3.4.1 Optimisations locales

Nous prétendons que le *code catégorique linéaire* se prête particulièrement bien aux optimisations *locales*. Par exemple, on a les instructions réversibles:

- [**Swap**; **Swap**] → []
- [**Swaap**; **Swaap**] → []
- [**Sswap**; **Sswap**] → []
- [**Cons**; **Split**] → []
- [**Split**; **Cons**] → []
- [**Push**; **Pop**] → []
- [**Pop**; **Push**] → []

Les instructions **Swap** et **Swaap** (ou **Swap** et **Sswap**, ou **Swaap** et **Sswap**) engendrent le *groupe des permutations de 3 éléments*. On peut représenter les éléments par leurs formes normales [], [**Swap**], [**Swaap**], [**Sswap**], [**Swaap**; **Swap**] et [**Sswap**; **Swap**], si on ajoute les règles:

- [**Swap**; **Swaap**] → [**Sswap**; **Swap**]
- [**Swap**; **Sswap**] → [**Swaap**; **Swap**]
- [**Swaap**; **Sswap**] → [**Sswap**; **Swap**]
- [**Sswap**; **Swaap**] → [**Swaap**; **Swap**]

On a aussi des simplifications qui utilisent la commutativité de certaines instructions, par exemple:

- [**Swap**; +] → [+]
- [**Copy**; **Swap**] → [**Copy**]

3.4.2 Macro-instructions

Nous avons choisi un jeu d'instructions pratiquement minimal pour notre *Machine Linéaire*[12]. Avec les architectures traditionnelles, on a intérêt à définir des *macro-instructions* qui sont exécutées plus rapidement:

- [**Swap**; **Cons**] → [**Xcons**]
- [**Split**; **Swap**] → [**Xsplit**]
- [**Split**; **Swap**; **Cons**] → [**Exch**]
- [**Push**; **Swap**] → [**Xpush**]
- [**Swap**; **Pop**] → [**Xpop**]
- [i; **Cons**] → [**Cons** i]
- [i; **Xcons**] → [**Xcons** i]
- [i; +] → [+i]
- [=; **Branch**(C', C'')] → [**Breq**(C', C'')]
- ...

D'autre part, lorsque **Call**(C) est *terminale*, on peut la remplacer par **Jump**(C) qui n'empile pas l'adresse de retour. La même remarque s'applique à **Branch**(−, −) et **App**.

3.4.3 Optimisations globales

Les optimisations locales ne suffisent pas. Certaines optimisations *essentielles* doivent être réalisées au moment de la compilation. C'est le cas du *code de distribution* d'un couple (voir section 3.3):

Lorsque toutes les variables de p sont utilisées par u (ou par v), il est inutile d'essayer de décomposer p (on ne ferait que fabriquer des () superflus). Dans ce cas, on peut traiter p comme un bloc (figures 3.5 et 3.6).

Ce genre d'optimisation apparemment anodine complique sérieusement le programme de compilation (voir annexe H).

3.4.4 Un problème ouvert

Du point de vue de l'implantation, il serait agréable de séparer clairement la phase de compilation de la phase d'optimisation. Or l'*optimisation de la distribution* que nous venons d'évoquer ne correspond pas à des règles de simplification du *code*, mais à des équations au niveau des *combinateurs linéaires* (annexe D).

Il serait intéressant d'orienter les équations de la *théorie des catégories monoïdales symétriques* (qui correspond au code de gestion de l'environnement), mais nous ne sommes jamais parvenus à exhiber un *système confluent et terminant* pour cette théorie.

Une réponse positive à ce problème pourrait conduire à un *optimiseur optimal* du code de gestion de la *Machine Linéaire*!

[12] On aurait pu se passer de **Sswap**, qui est équivalente à [**Swap**;**Swaap**;**Swap**].

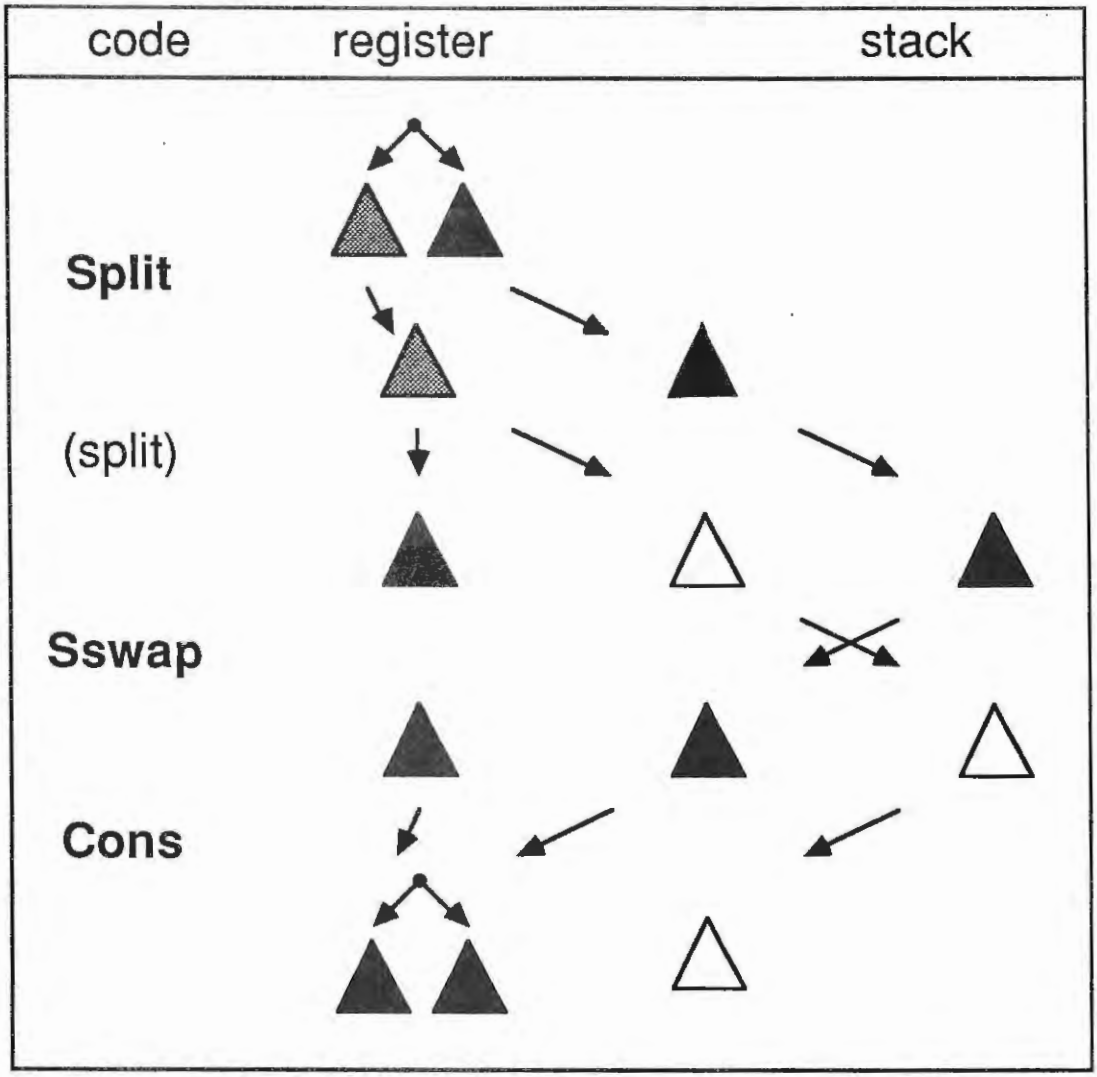


Figure 3.5: un cas d'optimisation

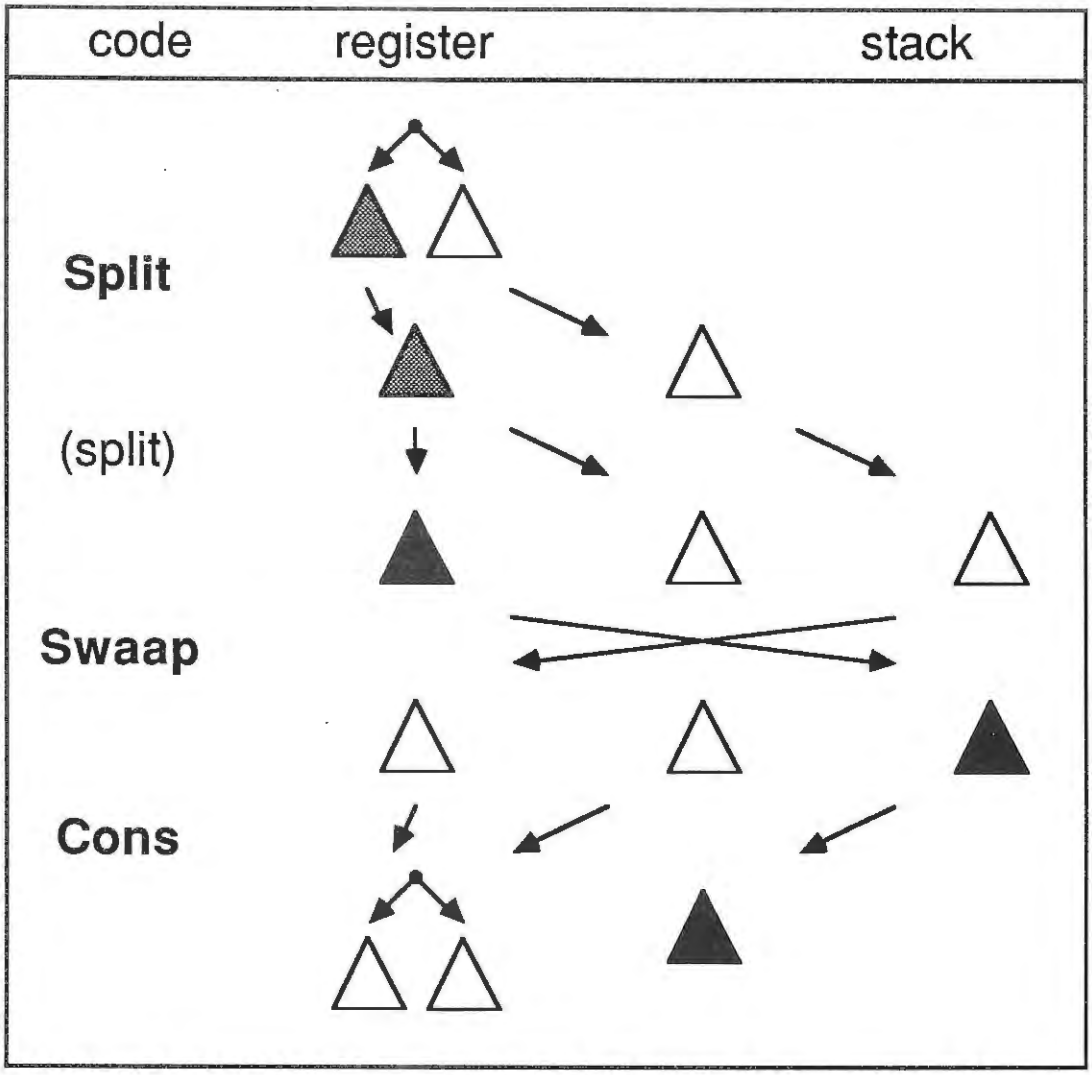


Figure 3.6: un autre cas d'optimisation

3.5 Types Concrets et Menus

Pour avoir un véritable langage de programmation, on permet à l'utilisateur de définir ses propres types de donnée sous forme de *types concrets* et de *menus*.

3.5.1 Types concrets

Les types concrets de LIVE sont tout à fait similaires à ceux de CAML. Par exemple, on peut définir la *somme directe*:

```
#type 'a 'b sum = inl 'a | inr 'b;;
Type sum defined
Constructor inl 'a : 'a 'b sum
          | inr 'a : 'b 'a sum
```

On retrouve ainsi les types concrets habituels: *listes*, *termes*, ..., avec la définition par *filtrage* (*pattern-matching*).

3.5.2 Menus

Les *menus*, par contre, constituent une nouveauté, même si on peut les assimiler à des *enregistrements* (*records*).

Un *menu* est une suite finie d'objets (de types différents). Un seul de ces objets sera finalement choisi, mais le choix est retardé ... De ce point de vue, les *menus* sont de même nature que les *schémas*.

L'exemple de base est le *produit direct*:

```
#type 'a 'b product = menu fst -> 'a | snd -> 'b;;
Type product defined
Destructor fst ('a 'b product) : 'a
         | snd ('a 'b product) : 'b

#menu fst -> 1987 | snd -> "mille neuf cent quatre-vingt sept";;
- : int string product

#fst it;;
1987 : int
```

Voici un exemple très simple de programme utilisant des menus. Il s'agit de calcul arithmétique sur des valeurs approchées. Si on veut calculer une valeur par défaut de x-y, il faut une valeur par défaut de x, et une valeur par excès de y ...

```
#type approx = menu lower -> int | higher -> int;;
Type approx defined
Destructor lower approx : int
         | higher approx : int

#operation x plus y -> menu lower  -> lower x + lower y
                         | higher -> higher x + higher y;;
Operation approx plus approx : approx;;
```

```
#operation x minus y -> menu lower  -> lower x - higher y
                          | higher -> higher x - lower y;;
Operation approx minus approx : approx;;

#let x = menu lower -> 39 | higher -> 45;;
Val x = - : approx

#let y = menu lower -> 14 | higher -> 18;;
Val y = - : approx

#lower(x minus y);;
21 : int
```

Notez que les contraintes linéaires interdisent le calcul de x minus x. D'autre part, si on utilisait des couples *valeur par défaut, valeur par excès*, on dupliquerait tous les calculs[13].

Avec les *menus récursifs*, on peut définir la modalité de la *Logique Linéaire* (section 2.3):

```
#type 'a of_course = menu read -> 'a
                       | kill -> ()
                       | dupl -> 'a of_course,'a of_course;;
Type of_course defined
Destructor read ('a of_course) : 'a
         | kill ('a of_course) :  ()
         | dupl ('a of_course) : 'a of_course,'a of_course
```

3.5.3 Implantation

On introduit des instructions spécialisées pour les types concrets et les menus:

Avant			Après		
code	environnement	pile	code	environnement	pile
$\mathbf{Mark}(i) :: C$	u	S	C	$(i \cdot u)$	$C :: S$
$\mathbf{Case}(C_1, \ldots, C_n) :: C$	$(i \cdot u)$	$v :: S$	C_i	(u, v)	$C :: S$
$\mathbf{Freeze}(C_1, \ldots, C_n) :: C$	u	S	C	$((C_1, \ldots, C_n) \cdot u)$	$C :: S$
$\mathbf{Choose}(i) :: C$	$((C_1, \ldots, C_n) \cdot u)$	S	C_i	u	$C :: S$

Dans ce tableau, i est un entier, et $(C_1, \ldots, C_n)$ est une *table* d'adresses de code, ce qui permet un accès en *temps constant*.

A chaque *constructeur* c, on associe un entier i, qui est son rang dans la déclaration du *type concret*. De même, à chaque *destructeur* d, on associe son rang dans la déclaration du *type menu*.

Les nouvelles instructions sont parfaitement adaptées à la compilation:

- $[\![c\, u]\!]_p = [\![u]\!]_p \mathbin{@} [\mathbf{Mark}(i)]$ (i est le rang du constructeur c)
- $[\![\mathbf{match}\, u\, \mathbf{with}\, c_1\, p_1 \to u_1 \mid \ldots \mid c_n\, p_n \to u_n]\!]_p = split_u \mathbin{@} [\![u]\!]_{p_u} \mathbin{@}$ $[\mathbf{Case}([\![u_1]\!]_{p_1,p'}, \ldots, [\![u_n]\!]_{p_n,p'})]$ (p' est le complément de p_u, et on suppose que les constructeurs $c_1, \ldots, c_n$ apparaissent selon leur rang)
- $[\![\mathbf{menu}\, d_1 \to u_1 \mid \ldots \mid d_n \to u_n]\!]_p = [\mathbf{Freeze}([\![u_1]\!]_p, \ldots, [\![u_n]\!]_p)]$ (on suppose que les destructeurs $d_1, \ldots, d_n$ apparaissent selon leur rang)

[13] Le *produit tensoriel* est *strict*, alors que le *produit direct* est *paresseux* (section 2.5).

- $[\![d\, u]\!]_p = [\![u]\!]_p \mathbin{@} [\mathrm{Choose}(i)]$ (i est le rang du destructeur d)

Pour les types concrets, on s'est limité au *filtrage* (*pattern matching*) *externe* (pas de constructeur en profondeur), *exhaustif* et *non ambigu*. On peut évidemment admettre des formes plus générales de *filtrage*, à condition de rester exhaustif (il n'y a pas d'exception en LIVE) et de résoudre les ambiguïtés (comme en CAML).

3.6 La fonctionnalité (partiellement) retrouvée

Dans la section 3.1, nous avons expliqué que les *fonctions* ne sont pas des *données*. C'est dommage, car on ne peut plus définir les fonctionnelles de CAML:

```
#let map f = maprec where rec maprec = function x::X -> f x::maprec X
                                                | []   -> [];;
Val map = - : (('a -> 'b) -> 'a list -> 'b list)

#map succ [0;8;7;6];;
[1; 9; 8; 7] : num list
```

3.6.1 Adresses de fonctions

Il n'est pas question de définir une fonction `map` sur les schémas, à cause des contraintes linéaires. Il faudrait un type qui, comme les entiers, échappe à ces contraintes.

Or les fonctions, une fois compilées, sont représentées par des adresses de code dans la zone *statique* de la mémoire. On peut donc les considérer comme des atomes: on les duplique ou les efface comme des entiers.

On introduit donc un type de donnée **function** $A \to B$, des *adresses de fonctions* de A dans B (ne pas confondre avec le type **scheme** $A \to B$ des schémas de A dans B).

On manipule ces adresses de la façon suivante: Si F est l'identificateur d'une fonction A dans B, 'F est un terme clos de type **function** $A \to B$. Inversement, si f est un terme de type **function** $A \to B$, $\{f\}$ est une fonction de A dans B.

Avec ces conventions, on peut définir une fonction `map`[14]:

```
#function map f -> scheme x::X -> let f as f = f in {f}x::(map f on X)
                        | []   -> let _ = f in [];;
Function map (function 'a -> 'b) : scheme ('a list) -> ('b list)

#map 'succ on [0;8;7;6];;
[1; 9; 8; 7] : int list
```

On peut aussi définir l'opérateur `make` de la *Logique Linéaire* (section 2.5):

```
#function make u ->
  scheme x -> menu read -> let r,_,_ = u in {r}x
                | kill -> let _,k,_ = u in {k}x
                | dupl -> let r as r,k as k,d as d as d = u in
                          let y,z = {d}x in
                          make(r,k,d) on y, make(r,k,d) on z;;
Function make (function 'a -> 'b),(function 'a -> ()),(function 'a -> 'a,'a) :
 scheme 'a -> ('b of_course)
```

[14] Pour la duplication et l'effacement d'adresse de fonction, il faudrait utiliser des notations distinctes de la duplication et l'effacement d'entier (à cause du *polymorphisme*).

3.6.2 Implantation

Pour compiler ces nouvelles constructions, il suffit d'introduire les instructions **Quote**$(-)$ et **Call**:

Avant			Après		
code	environnement	pile	code	environnement	pile
$\mathbf{Quote}(C') :: C$	u	S	C	C'	$u :: S$
$\mathbf{Call} :: C$	C'	$u :: S$	C'	u	$C :: S$

Remarquez que $[\mathbf{Quote}(C); \mathbf{Call}]$ équivaut à $[\mathbf{Call}(C)]$.

- $[\![\text{`}f \,]\!]_p = pop \mathbin{@} [\mathbf{Quote}(C)]$
- $[\![\{u\}v \,]\!]_p = split_v \mathbin{@} [\![v \,]\!]_{p_v} \mathbin{@} [\mathbf{Swap}] \mathbin{@} [\![u \,]\!]_{p_u} \mathbin{@} [\mathbf{Call}]$

3.7 Conclusion

Dans ce chapitre, on a montré qu'il est possible de développer un véritable langage de programmation à partir des principes de la *Logique Linéaire Intuitionniste*.

Quels peuvent être les avantages de LIVE sur un langage fonctionnel comme CAML?

- Pas de *garbage collector*, d'où une implantation plus simple et potentiellement plus efficace.
- Un calcul d'optimisations assez sophistiqué (section 3.4) correspondant à une analyse fine de l'environnement, simplifiée par la forme particulière des combinateurs.
- Les *contraintes linéaires* conditionnent le style des programmes. Typiquement, on décrit les structures de donnée une seule fois, ce qui assure un certaine optimalité.
- Le *type checker* détecte des erreurs qui sont admises par CAML, par exemple une variable qui n'est pas utilisée.
- ...

Par contre il est évident que les *contraintes linéaires* compliquent les programmes, et la syntaxe n'est guère agréable.

De plus, il reste des problèmes fondamentaux non résolus:

- La notion d'*entrée/sortie* reste essentiellement étrangère au langage.
- Les *structures de donnée incomplètes* sont représentées par des *schémas*, c'est à dire des *fermetures*, ce qui est inefficace[15]: Il y a trop de paresse!
- Le langage est essentiellement séquentiel[16].

C'est pourquoi nous étudions maintenant la *Logique Linéaire Classique* [Girard87] qui répond à ces questions de façon très élégante ...

[15] Les variables logiques de PROLOG permettent une représentation directe de ces structures.

[16] On peut exécuter en parallèle les deux composantes d'un programme $\varphi \otimes \psi$, mais il s'agit d'une forme de parallèlisme très limité.

Annexe A

Le Calcul des Séquents Intuitionniste

On manipule des *séquents* $A_1, \ldots, A_n \vdash B$ (sous les hypothèses $A_1, \ldots, A_n$, on montre B).

Les *règles structurelles* ne font pas intervenir de connecteur:

(Γ, Δ désignent des suites de formules $A_1, \ldots, A_n$)

$$\frac{}{A \vdash A}\,(\text{identité}) \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\,(\text{coupure}) \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,(\text{échange})$$

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\,(\text{affaiblissement}) \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\,(\text{contraction})$$

Les *règles logiques* font intervenir une seule occurrence de connecteur, à gauche ou à droite du symbole $\vdash$:

$$\frac{}{\vdash \top} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \wedge B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \wedge B \vdash C}$$

$$\frac{}{\bot \vdash A} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma, A \vdash C \quad \Delta, B \vdash C}{\Gamma, \Delta, A \vee B \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \Rightarrow B \vdash C}$$

On a une *propriété de sous-formule* évidente:

Proposition 8 *Dans une preuve sans coupure d'un séquent $A_1, \ldots, A_n \vdash B$, il n'y a que des sous-formules de $A_1, \ldots, A_n$ et B.*

On a un *Théorème de Normalisation* (*élimination de la règle de coupure*). Par conséquent:

Corollaire 5 *Le Calcul des Séquents sans la règle de coupure est équivalent au Calcul des Séquents avec la règle de coupure.*

La *règle de coupure* peut être éliminée, mais elle n'est pas "superflue":

- Elle permet d'écrire des preuves plus courtes et plus élégantes.
- Du point de vue théorique, elle permet de démontrer les résultats de complétude et d'équivalence avec les autres systèmes de preuves (*Déduction Naturelle*, *Combinateurs Catégoriques*).
- Si on ajoute d'autres règles (par exemple le *principe de récurrence* de l'*Arithmétique de Peano*), le *Théorème de Normalisation* ne s'applique plus.

Annexe B

Les Coupures Commutatives

Une *coupure commutative* est une règle d'*élimination* dont la *formule principale* est la *conclusion* d'une règle d'*élimination* de $\bot$ ou de $\vee$:

$$\dfrac{\dfrac{\vdots}{\bot}}{\dfrac{A \wedge B}{A}} \qquad \dfrac{\dfrac{\vdots}{\bot}}{\dfrac{A \wedge B}{B}} \qquad \dfrac{\dfrac{\vdots}{\bot}}{\dfrac{\bot}{A}} \qquad \dfrac{\dfrac{\vdots}{\bot} \quad \overset{[A]}{\vdots} \quad \overset{[B]}{\vdots}}{\dfrac{A \vee B \quad C \quad C}{C}} \qquad \dfrac{\dfrac{\vdots}{\bot} \quad \vdots}{\dfrac{A \Rightarrow B \quad A}{B}}$$

$$\dfrac{\vdots \quad \overset{[A]}{\vdots} \quad \overset{[B]}{\vdots}}{\dfrac{A \vee B \quad C \wedge D \quad C \wedge D}{\dfrac{C \wedge D}{C}}} \qquad \dfrac{\vdots \quad \overset{[A]}{\vdots} \quad \overset{[B]}{\vdots}}{\dfrac{A \vee B \quad C \wedge D \quad C \wedge D}{\dfrac{C \wedge D}{D}}} \qquad \dfrac{\vdots \quad \overset{[A]}{\vdots} \quad \overset{[B]}{\vdots}}{\dfrac{A \vee B \quad \bot \quad \bot}{\dfrac{\bot}{C}}}$$

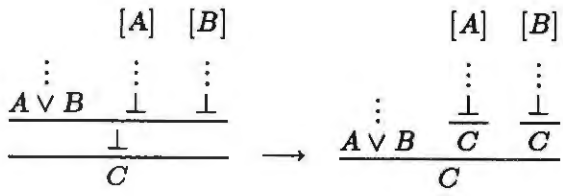
$$\dfrac{\vdots \quad \overset{[A]}{\vdots} \quad \overset{[B]}{\vdots} \quad \overset{[C]}{\vdots} \quad \overset{[D]}{\vdots}}{\dfrac{A \vee B \quad C \vee D \quad C \vee D \quad E \quad E}{\dfrac{C \vee D}{E}}} \qquad \dfrac{\vdots \quad \overset{[A]}{\vdots} \quad \overset{[B]}{\vdots} \quad \vdots}{\dfrac{A \vee B \quad C \Rightarrow D \quad C \Rightarrow D \quad C}{\dfrac{C \Rightarrow D}{D}}}$$

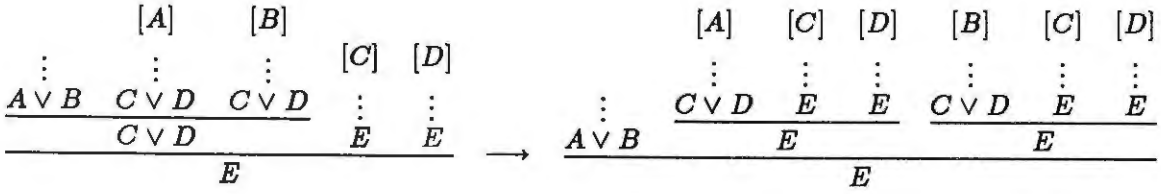
Les coupures du $\bot$ s'éliminent de façon évidente:

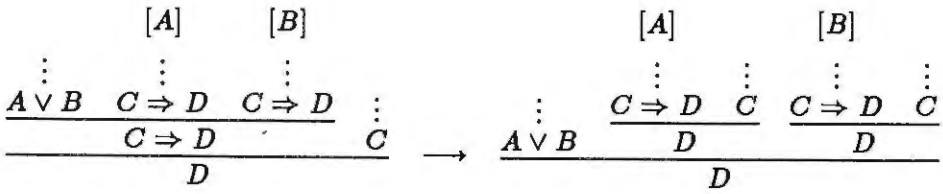
$$\dfrac{\dfrac{\vdots}{\bot}}{\dfrac{A \wedge B}{A}} \longrightarrow \dfrac{\vdots}{\dfrac{\bot}{A}}$$

Les coupures du $\vee$ s'éliminent par "commutation" des règles:

$$\dfrac{\vdots \quad \overset{[A]}{\vdots} \quad \overset{[B]}{\vdots}}{\dfrac{A \vee B \quad C \wedge D \quad C \wedge D}{\dfrac{C \wedge D}{C}}} \longrightarrow \dfrac{\vdots \quad \overset{[A]}{\vdots} \quad \overset{[B]}{\vdots}}{\dfrac{A \vee B \quad \dfrac{C \wedge D}{C} \quad \dfrac{C \wedge D}{C}}{C}}$$







Annexe C

Démonstration du Théorème de Cohérence Hiérarchique

On démontre ici le *Théorème de Cohérence Hiérarchique*:

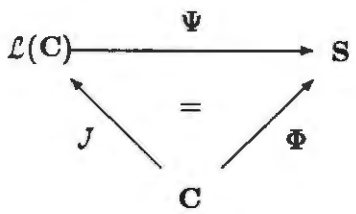
Pour toute catégorie $\mathbf{C}$*, le foncteur canonique* $J : \mathbf{C} \to \mathcal{L}(\mathbf{C})$ *est plein et fidèle. Autrement dit, si* A *et* B *sont des objets de* $\mathbf{C}$*,* J *induit une bijection:*

$$\mathbf{Hom}_{\mathbf{C}}(A, B) \xrightarrow{\sim} \mathbf{Hom}_{\mathcal{L}(\mathbf{C})}(A, B)$$

C.1 J est fidèle

Pour montrer que J est fidèle, il suffit de trouver *une catégorie cartésienne fermée* $\mathbf{S}$ avec un foncteur fidèle $\Phi : \mathbf{C} \to \mathbf{S}$.

En effet, par la propriété universelle de $\mathcal{L}(\mathbf{C})$, on a alors un foncteur $\Psi : \mathcal{L}(\mathbf{C}) \to \mathbf{S}$ tel que $\Psi J = \Phi$. Comme Φ est fidèle, J l'est aussi[1].



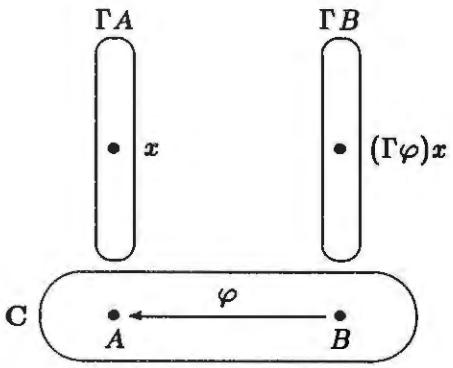
On considère la catégorie $\hat{\mathbf{C}}$ des *foncteurs contravariants* $\mathbf{C} \to \mathbf{Set}$ (*préfaisceaux*), et le foncteur de *Yoneda* $\mathcal{Y} : \mathbf{C} \to \hat{\mathbf{C}}$ défini par:

$$(\mathcal{Y}A)B = \mathbf{Hom}_{\mathbf{C}}(B, A)$$

Lemme 4 *(Yoneda)*
Soit $\Gamma \in \hat{\mathbf{C}}$*. Pour tout objet* A *de* $\mathbf{C}$*, on a un isomorphisme canonique* $\Gamma A \simeq \mathbf{Hom}_{\hat{\mathbf{C}}}(\mathcal{Y}A, \Gamma)$.

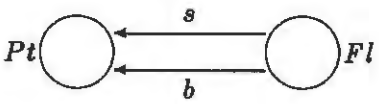
Démonstration:
A tout $x \in \Gamma A$, on associe la transformation naturelle τ définie par $\tau_B\, \varphi = (\Gamma\varphi)x$ pour tout $\varphi \in (\mathcal{Y}A)B$.

[1] Notez que si Φ est plein, on ne peut en déduire que J l'est aussi.

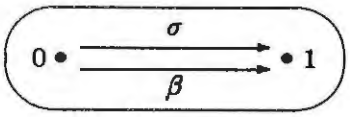


Réciproquement, à toute transformation naturelle $\tau : \mathcal{Y}A \to \Gamma$, on associe $\tau_A\, id_A \in \Gamma A$. □

Par exemple, un *graphe* est la donnée de deux ensembles Pt (*points*) et Fl (*flêches*) avec deux fonctions $s, b : Pt \to Fl$ (*source* et *but*):



Autrement dit, un graphe est un objet de $\hat{\mathbf{C}}$ où $\mathbf{C}$ est la catégorie:



$\mathcal{Y}0$ est le graphe réduit à un point, et $\mathcal{Y}1$ est constitué d'une flêche reliant deux points:

Si Γ est un graphe, un point de Γ (c'est à dire un élément de $\Gamma 0$) peut être considéré comme un morphisme $\mathcal{Y}0 \to \Gamma$, et une flêche de Γ (c'est à dire un élément de $\Gamma 1$) comme un morphisme $\mathcal{Y}1 \to \Gamma$.

Le lemme de *Yoneda* a pour conséquence:

Corollaire 6 *Le foncteur de* Yoneda $\mathcal{Y} : \mathbf{C} \to \hat{\mathbf{C}}$ *est plein et fidèle.*

Démonstration: On applique le lemme de *Yoneda* à $\Gamma = \mathcal{Y}B$:

$$\mathbf{Hom}_{\mathbf{C}}(A, B) = (\mathcal{Y}B)A \simeq \mathbf{Hom}_{\hat{\mathbf{C}}}(\mathcal{Y}A, \mathcal{Y}B)$$

Donc $\mathcal{Y}$ est bijectif sur les morphismes. □

Pour montrer que J est fidèle, il suffit de vérifier:

Lemme 5 $\hat{\mathbf{C}}$ *est cartésienne fermée.*

Démonstration: La construction du produit cartésien est évidente: $(\Gamma \wedge \Delta)A = \Gamma A \times \Delta A$. Mais on n'a sûrement pas $(\Gamma \Rightarrow \Delta)A = \Delta A^{\Gamma A}$ (ce qui ne définit même pas un foncteur).

Par contre, si $\Gamma \Rightarrow \Delta$ existe, le lemme de *Yoneda* et la caractérisation catégorique de l'exponentielle donnent:

$$(\Gamma \Rightarrow \Delta)A \simeq \mathbf{Hom}_{\hat{\mathbf{C}}}(yA, \Gamma \Rightarrow \Delta) \simeq \mathbf{Hom}_{\hat{\mathbf{C}}}(yA \wedge \Gamma, \Delta)$$

On pose alors $(\Gamma \Rightarrow \Delta)A = \mathbf{Hom}_{\hat{\mathbf{C}}}(yA \wedge \Gamma, \Delta)$, et on vérifie que c'est une exponentielle. □

Ainsi, la catégorie des graphes est cartésienne fermée. La définition de l'exponentielle n'était nullement évidente à priori (ni les points ni les flêches de $\Gamma \Rightarrow \Delta$ ne sont des morphismes de graphe):

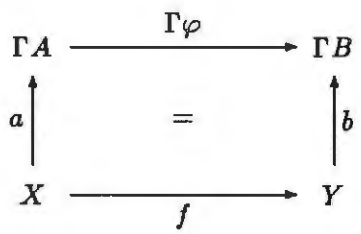
- Un *point* de $\Gamma \Rightarrow \Delta$ est une *fonction* F des points de Γ dans les points de Δ.
- Une flêche $F \to G$ dans $\Gamma \Rightarrow \Delta$ est la donnée d'une flêche $F\,A \to G\,B$ dans Δ pour toute flêche $A \to B$ dans Γ.

C.2 J est plein

Pour montrer que J est plein, la catégorie $\hat{\mathbf{C}}$ ne suffit plus. On a besoin d'une nouvelle construction.

Si $\Gamma : \mathbf{L} \to \mathbf{S}$ est un foncteur, on définit une catégorie $\mathcal{G}l(\Gamma)$ (*glueing*):

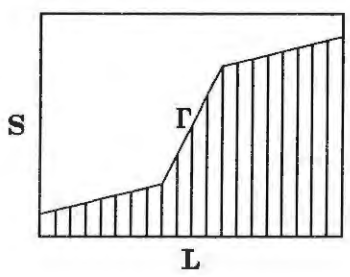
- Un objet de $\mathcal{G}l(\Gamma)$ est un couple d'objets A dans $\mathbf{L}$ et X dans $\mathbf{S}$, avec un morphisme $a : X \to \Gamma A$ dans $\mathbf{S}$.
- Un morphisme $A, X, a \to B, Y, b$ est un couple de morphismes $\varphi : A \to B$ dans $\mathbf{L}$ et $f : X \to Y$ dans $\mathbf{S}$ tels que $\Gamma\varphi \circ a = b \circ f$.



On a évidemment deux foncteurs d'oubli $\mathcal{F}st : \mathcal{G}l(\Gamma) \to \mathbf{L}$ et $\mathcal{S}nd : \mathcal{G}l(\Gamma) \to \mathbf{S}$.

Donnons deux exemples:

Si $\mathbf{L}$ et $\mathbf{S}$ sont des ensembles ordonnés, et Γ une fonction croissante $\mathbf{L} \to \mathbf{S}$, $\mathcal{G}l(\Gamma)$ est l'*hypographe* de Γ:



Si $\mathbf{L} = \mathbf{S} = \mathbf{Set}$ et Γ est le foncteur $A \mapsto A \times A$, $\mathcal{G}l(\Gamma)$ est la catégorie des graphes!

Lemme 6 *(Bénabou)*

Si $\mathbf{L}$ *et* $\mathbf{S}$ *sont des catégories cartésiennes fermées,* $\mathbf{S}$ *a des produits fibrés (ou, ce qui revient au même, des égalisateurs), et* Γ *préserve les produits cartésiens, alors* $\mathcal{G}l(\Gamma)$ *est une catégorie cartésienne fermée et* $\mathcal{F}st$ *préserve les produits cartésiens et les exponentielles (mais* $\mathcal{S}nd$ *préserve seulement les produits cartésiens).*

Démonstration: La construction du produit cartésien est immédiate. Celle de l'exponentielle est plus délicate:

Soient A, X, a et B, Y, b dans $\mathcal{G}l(\Gamma)$. On veut construire l' exponentielle C, Z, c. Comme il faut que $\mathcal{F}st$ préserve les exponentielles, on pose $C = A \Rightarrow B$. Naturellement, Z n'est pas $X \Rightarrow Y$, mais le produit fibré de deux morphismes $a' : \Gamma(A \Rightarrow B) \to X \Rightarrow \Gamma B$ et $b' : X \Rightarrow Y \to X \Rightarrow \Gamma B$ que l'on construit à partir de a et b.

$$\begin{array}{ccc} Z & \xrightarrow{\quad c \quad} & \Gamma(A \Rightarrow B) \\ \downarrow & = & \downarrow a' \\ X \Rightarrow Y & \xrightarrow[b']{} & X \Rightarrow \Gamma B \end{array}$$

Nous ne détaillons pas les vérifications qui nécessitent quelques diagrammes indigestes. □

Si $\mathbf{L} = \mathbf{S} = \mathbf{Set}$ et Γ est le foncteur $A \mapsto A \times A$, on retrouve le fait que la catégorie des graphes est cartésienne fermée.

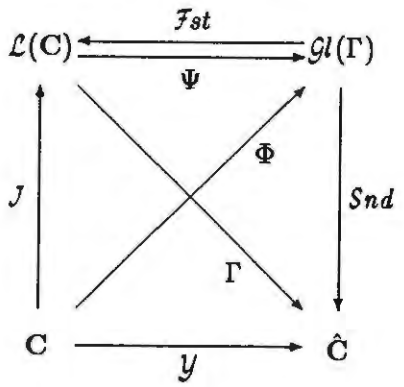
On va appliquer le lemme 6 au foncteur $\Gamma : \mathcal{L}(\mathbf{C}) \to \hat{\mathbf{C}}$ défini par $\Gamma A = \mathbf{Hom}_{\mathcal{L}(\mathbf{C})}(J-, A)$. Il est clair que Γ préserve les produits cartésiens.

On considère le foncteur $\Phi : \mathbf{C} \to \mathcal{G}l(\Gamma)$ défini par:

- ΦA est le couple JA, yA muni de la transformation naturelle $\iota : yA \to \Gamma(JA)$ telle que $\iota_B \varphi = J\varphi$ pour tout $\varphi \in (yA)B = \mathbf{Hom}_{\mathbf{C}}(B, A)$.

On a $\mathcal{F}st\Phi = J$.

Par la propriété universelle de $\mathcal{L}(\mathbf{C})$, on a un foncteur (unique) $\Psi : \mathcal{L}(\mathbf{C}) \to \mathcal{G}l(\Gamma)$ qui préserve les produits cartésiens et les exponentielles et tel que $\Psi J = \Phi$.



Ainsi, $\mathcal{F}st\,\Psi$ préserve les produits cartésiens et les exponentielles, et $(\mathcal{F}st\,\Psi)J = \mathcal{F}st(\Psi J) = \mathcal{F}st\,\Phi = J$. Donc, par la propriété d'unicité, $\mathcal{F}st\,\Psi = \mathcal{I}d_{\mathcal{L}(\mathbf{C})}$.

Donc, pour tout $\varphi \in \mathbf{Hom}_{\mathcal{L}(\mathbf{C})}(A, B)$, $\Psi\varphi$ est de la forme φ, τ avec $\tau \in \mathbf{Hom}_{\hat{\mathbf{C}}}(A, B)$ tel que le diagramme suivant commute:

$$\begin{array}{ccc} \Gamma(JA) & \xrightarrow{\Gamma\varphi} & \Gamma(JB) \\ {\scriptstyle\iota}\uparrow & = & \uparrow{\scriptstyle\iota} \\ yA & \xrightarrow[\tau]{} & yB \end{array}$$

En particulier, pour $id_A \in (yA)A$, on obtient:

$$\varphi = \varphi \circ Jid_A = J(\tau_A id_A)$$

Ceci montre que J est plein.

Annexe D

Logique Linéaire Catégorique

D.1 Terminologie

Une *catégorie monoïdale symétrique* est une catégorie $\mathcal{C}$ munie d'un bifoncteur $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ et d'un objet $\mathbf{1} \in \mathcal{C}$ tels que:

$$X \otimes (Y \otimes Z) \cong (X \otimes Y) \otimes Z \qquad X \cong \mathbf{1} \otimes X \qquad X \otimes Y \cong Y \otimes X$$

$\cong$ dénote un *isomorphisme naturel.* Ces *isomorphismes naturels* doivent vérifier les équations de MacLane-Kelly (voir D.3).

Une *catégorie monoïdale symétrique* $\mathcal{C}$ est *fermée* si le foncteur $X \mapsto X \otimes A$ admet un adjoint à droite $Y \mapsto A \multimap Y$ pour tout $A \in \mathcal{C}$:

$$\mathbf{Hom}(X \otimes A, Y) \cong \mathbf{Hom}(X, A \multimap Y)$$

Un *modèle catégorique* de la *Logique Linéaire* est tout simplement une *catégorie monoïdale symétrique fermée* $(\mathcal{C}, \otimes, \mathbf{1}, \multimap)$ avec *produits finis* $(\&, \mathbf{t})$ et *coproduits finis* $(\oplus, \mathbf{0})$.

D.2 Quelques modèles catégoriques

Une catégorie avec *produits* finis est *monoïdale symétrique*, donc une *catégorie cartésienne fermée* avec *coproduits* finis est un *modèle catégorique* de la *Logique Linéaire*[1]. Nous qualifions ces modèles de *dégénés* parce qu'il n'y a pas de différence entre le *produit tensoriel* et le *produit direct.* Dans **Set**, par exemple, $\otimes$ (ainsi que $\&$) est le produit cartésien, $\oplus$ est l'union disjointe, et $X \multimap Y = Y^X$.

Un exemple plus intéressant est la catégorie des modules sur un anneau: $\otimes$ est le produit tensoriel et $X \multimap Y = \mathbf{Hom}(X, Y)$. Dans ce cas, il n'y a pas de différence entre le *produit direct* $(\&)$ et la *somme directe* $(\oplus)$.

La catégorie des espaces topologiques n'est pas cartésienne fermée, mais c'est un *modèle catégorique* de la *Logique Linéaire*: $X \otimes Y$ est l'ensemble $X \times Y$ muni de la topologie la plus fine qui rende les *sections* $x \mapsto (x, y)$ et $y \mapsto (x, y)$ continues. $\mathbf{1}$ est le singleton. $X \multimap Y$ est l'espace des fonctions continues $X \to Y$, muni de la topologie de la *convergence simple.* $\&$ est le produit cartésien ordinaire et $\oplus$ est l'union disjointe. Le produit *tensoriel* et le produit *direct* sont différents, mais $\mathbf{1}$ sont $\mathbf{t}$ identiques.

Un autre exemple est la catégorie des *ensembles pointés*: Un objet est un ensemble muni d'un élément distingué $\perp$, et un morphisme est une fonction préservant $\perp$. Le *produit tensoriel* de X par Y est $X \times Y$ où l'on identifie toutes les paires contenant un $\perp$, et l'*unité tensorielle* est $\{\perp, \top\}$.

[1] voilà qui justifie la traduction de 2.1.3.

$X \multimap Y$ est l'ensemble $\mathbf{Hom}(X, Y)$ avec la fonction constante $\perp$ comme objet $\perp$. Le *produit direct* est le produit cartésien, et la *somme* directe est l'union disjointe où l'on identifie les deux $\perp$. $\mathbf{t}$ et $\mathbf{0}$ sont identiques.

D.3 Equations

Les définitions de D.1 donnent la *théorie équationnelle*:

Axiomes catégoriques:

$$(\varphi \circ \psi) \circ \chi = \varphi \circ (\psi \circ \chi) \qquad \mathbf{Id} \circ \varphi = \varphi \qquad \varphi \circ \mathbf{Id} = \varphi$$

Fonctorialité du produit tensoriel:

$$(\varphi \circ \varphi') \otimes (\psi \circ \psi') = (\varphi \otimes \psi) \circ (\varphi' \otimes \psi') \qquad \mathbf{Id} \otimes \mathbf{Id} = \mathbf{Id} : A \otimes B \to A \otimes B$$

$$\mathbf{1} = \mathbf{Id} : \mathbf{1} \to \mathbf{1}$$

Naturalité des combinateurs d'arrangement:

$$\begin{array}{ccc} A \otimes (B \otimes C) & \xrightarrow{\mathbf{Assl}} & (A \otimes B) \otimes C \\ \varphi \otimes (\psi \otimes \chi) \downarrow & = & \downarrow (\varphi \otimes \psi) \otimes \chi \\ A' \otimes (B' \otimes C') & \xrightarrow[\mathbf{Assl}]{} & (A' \otimes B') \otimes C' \end{array} \qquad \cdots$$

Inverses:

$$\mathbf{Assl} \circ \mathbf{Assr} = \mathbf{Id} : (A \otimes B) \otimes C \to (A \otimes B) \otimes C$$

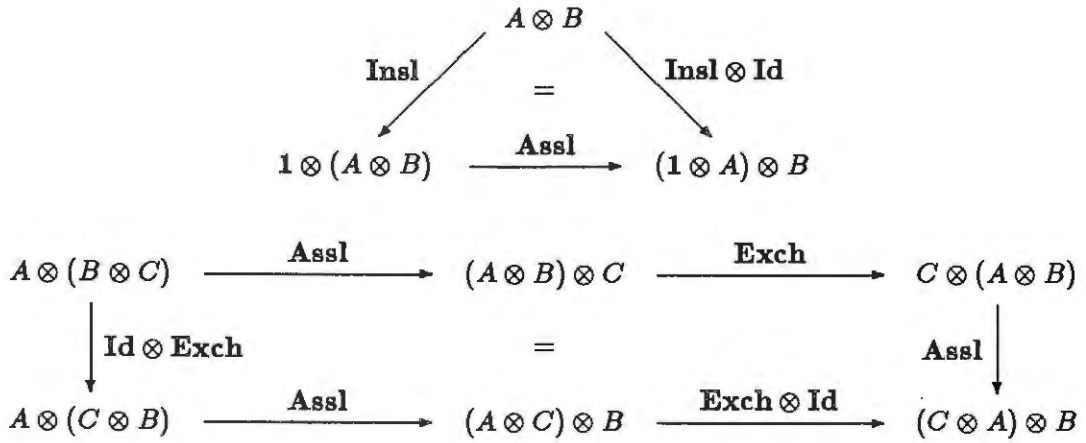
$$\mathbf{Assr} \circ \mathbf{Assl} = \mathbf{Id} : A \otimes (B \otimes C) \to A \otimes (B \otimes C)$$

$$\mathbf{Insl} \circ \mathbf{Dell} = \mathbf{Id} : \mathbf{1} \otimes A \to \mathbf{1} \otimes A \qquad \mathbf{Dell} \circ \mathbf{Insl} = \mathbf{Id} : A \to A$$

$$\mathbf{Exch} \circ \mathbf{Exch} = \mathbf{Id} : A \otimes B \to A \otimes B$$

Equations de MacLane-Kelly:

$$\begin{array}{ccccc} A \otimes (B \otimes (C \otimes D)) & \xrightarrow{\mathbf{Assl}} & (A \otimes B) \otimes (C \otimes D) & \xrightarrow{\mathbf{Assl}} & ((A \otimes B) \otimes C) \otimes D \\ \downarrow \mathbf{Id} \otimes \mathbf{Assl} & & = & & \mathbf{Assl} \otimes \mathbf{Id} \uparrow \\ A \otimes ((B \otimes C) \otimes D) & & \xrightarrow{\mathbf{Assl}} & & (A \otimes (B \otimes C)) \otimes D \end{array}$$



Fermeture:

$$\mathbf{App} \circ (\mathbf{Cur}(\varphi) \otimes \psi) = \varphi \circ (\mathbf{Id} \otimes \psi) \qquad \mathbf{Cur}(\varphi) \circ \psi = \mathbf{Cur}(\varphi \circ (\psi \otimes \mathbf{Id}))$$

$$\mathbf{Cur}(\mathbf{App}) = \mathbf{Id} : A \multimap B \to A \multimap B$$

Produit:

$$\mathbf{Fst} \circ \mathbf{Pair}(\varphi, \psi) = \varphi \qquad \mathbf{Snd} \circ \mathbf{Pair}(\varphi, \psi) = \psi \qquad \mathbf{Pair}(\varphi, \psi) \circ \chi = \mathbf{Pair}(\varphi \circ \chi, \psi \circ \chi)$$

$$\mathbf{Pair}(\mathbf{Fst}, \mathbf{Snd}) = \mathbf{Id} : A \,\&\, B \to A \,\&\, B \qquad \mathbf{Void} \circ \varphi = \varphi \qquad \mathbf{Void} = \mathbf{Id} : \mathbf{t} \to \mathbf{t}$$

Coproduit:

$$\mathbf{Case}(\varphi, \psi) \circ \mathbf{Inl} = \varphi \qquad \mathbf{Case}(\varphi, \psi) \circ \mathbf{Inr} = \psi \qquad \chi \circ \mathbf{Case}(\varphi, \psi) = \mathbf{Case}(\chi \circ \varphi, \chi \circ \psi)$$

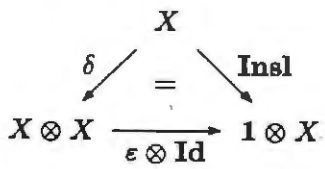
$$\mathbf{Case}(\mathbf{Inl}, \mathbf{Inr}) = \mathbf{Id} : A \oplus B \to A \oplus B \qquad \varphi \circ \mathbf{Can} = \varphi \qquad \mathbf{Can} = \mathbf{Id} : \mathbf{0} \to \mathbf{0}$$

Annexe E

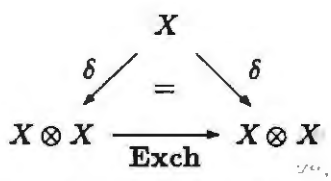
La Co-monade Bien Sûr!

On utilise la modalité $!A$ lorsqu'on a besoin d'un ***nombre arbitraire de copies*** d'une donnée de type A. Mais les règles de la section 2.3 n'expriment pas que ce sont des copies de la ***même*** donnée. **Vu que $\otimes$ *n'est pas*** un produit cartésien, on ne peut exiger, par exemple, que $\mathbf{Dupl} : !A \to !A \otimes !A$ soit la diagonale. Mais on peut exprimer cela de façon contournée.

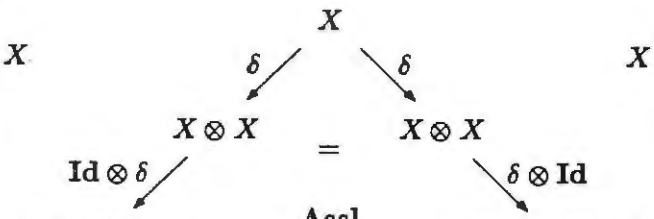
Un type X muni de deux morphismes $\varepsilon : X \to \mathbf{1}$ et $\delta : X \to X \otimes X$ est un ***co-monoïde*** (commutatif) s'il vérifie les axiomes suivants:



(neutre à gauche)



(commutativité)



(associativité)

Maintenant, nous pouvons donner une définition correcte de la modalité:
$!A$ est le ***co-monoïde co-libre*** co-engendré par A.

D'où les règles correctes:

$$\frac{\varphi : A \to B \quad \varepsilon : A \to 1 \quad \delta : A \to A \otimes A \quad (A, \delta, \varepsilon \text{ est un co-monoïde})}{\mathbf{Make}(\varphi, \varepsilon, \delta) : A \to !B}$$

$$\frac{}{\mathbf{Read} : !A \to A} \quad \frac{}{\mathbf{Kill} : !A \to 1} \quad \frac{}{\mathbf{Dupl} : !A \to !A \otimes !A} \quad (!A, \mathbf{Dupl}, \mathbf{Kill} \text{ est un co-monoïde})$$

On ajoute le fait que $\mathbf{Make}(\varphi, \varepsilon, \delta)$ est le seul morphisme $\pi : A \to !B$ tel que:

$$\mathbf{Read} \circ \pi = \varphi \qquad \mathbf{Kill} \circ \pi = \varepsilon \qquad \mathbf{Dupl} \circ \pi = (\pi \otimes \pi) \circ \delta$$

Dans le cas degénéré (voir D.2), les seuls co-monoïdes sont les X, ε, δ où ε est la flêche canonique, et δ est la diagonale. Cela signifie que les ***modèles degénérés*** sont des ***modèles*** de la modalité, avec $!A = A$.

Mais le résultat le plus intéressant est cette version catégorique du théorème 8 (voir aussi [Fox]):

Théorème 10 *Si $\mathcal{C}$ est une catégorie monoïdale symétrique fermée avec produits finis et modalité, alors la catégorie de co-Kleisli associée à la co-monade ! est cartésienne fermée*[1].

La démonstration utilise le résultat suivant, qui est une version catégorique de la proposition 6:

Proposition 9 *Il y a un isomophisme canonique:* $!(A \& B) \cong !A \otimes !B$.

[1]Pour ces notions de *monade* et de *catégorie de Kleisli*, voir [Maclane] par exemple.

Annexe F

Quelques Combinateurs Linéaires

$$\mathbf{Insr} = \mathbf{Exch} \circ \mathbf{Insl} : A \to A \otimes \mathbf{1} \qquad \mathbf{Delr} = \mathbf{Dell} \circ \mathbf{Exch} : A \otimes \mathbf{1} \to A$$

$$\mathbf{Mix} = \mathbf{Assr} \circ ((\mathbf{Assl} \circ (\mathbf{Id} \otimes \mathbf{Exch}) \circ \mathbf{Assr}) \otimes \mathbf{Id}) \circ \mathbf{Assl} : (A \otimes B) \otimes (C \otimes D) \to (A \otimes C) \otimes (B \otimes D)$$

$$\frac{x : !A \to B}{\mathbf{Lift}(x) = \mathbf{Make}(x, \mathbf{Kill}, \mathbf{Dupl}) : !A \to !B} \qquad \frac{x : A \to B}{!x = \mathbf{Lift}(x \circ \mathbf{Read}) : !A \to !B}$$

$$\mathbf{Subl} = \mathbf{Kill} : !\mathbf{t} \to \mathbf{1} \qquad \mathbf{Crys} = \mathbf{Make}(\mathbf{Void}, \mathbf{Id}, \mathbf{Insl}) : \mathbf{1} \to !\mathbf{t}$$

$$\mathbf{Crac} = (!\mathbf{Fst} \otimes !\mathbf{Snd}) \circ \mathbf{Dupl} : !(A \,\&\, B) \to !A \otimes !B$$

$$\mathbf{Glue} = \mathbf{Make}(\mathbf{Pair}(\mathbf{Delr} \circ (\mathbf{Read} \otimes \mathbf{Kill}), \mathbf{Dell} \circ (\mathbf{Kill} \otimes \mathbf{Read})), \mathbf{Dell} \circ (\mathbf{Kill} \otimes \mathbf{Kill}), \mathbf{Mix} \circ (\mathbf{Dupl} \otimes \mathbf{Dupl})) : !A \otimes !B \to !(A \,\&\, B)$$

Annexe G

Démonstration du Théorème de Terminaison (cas linéaire)

On démontre le théorème 9.

Si $\varphi : A \to B$ est un combinateur, et $\mathcal{C}$ un ensemble de *combinateurs canoniques* de type B, on écrit $\varphi\ \mu \downarrow \mathcal{C}$ lorsque μ est un *combinateur canonique* de type A et il existe $\nu \in \mathcal{C}$ et un *combinateur primitif* α tels que:

$$\mu \overset{\varphi}{\underset{\alpha}{\rightsquigarrow}} \nu$$

Pour toute fomule A, on construit par récurrence un ensemble $\mathbf{Conv}(A)$ de combinateurs canoniques de type A:

- $\mathbf{Conv}(A) = \{\mathbf{Id}\}$ (A est atomique)
- $\mathbf{Conv}(\mathbf{1}) = \{1\}$
- $\mathbf{Conv}(A \otimes B) = \{\mu \otimes \nu;\ \mu \in \mathbf{Conv}(A),\ \nu \in \mathbf{Conv}(B)\}$
- $\mathbf{Conv}(A \multimap B) = \{\mathbf{Cur}(\varphi) \circ \mu;\ \varphi : X \otimes A \to B,\ (\forall \nu \in \mathbf{Conv}(A))\ \varphi\ (\mu \otimes \nu) \downarrow \mathbf{Conv}(B)\}$
- $\mathbf{Conv}(A \,\&\, B) = \{\mathbf{Pair}(\varphi, \psi) \circ \mu;\ \varphi : X \to A,\ \psi : X \to B,\ \varphi\ \mu \downarrow \mathbf{Conv}(A),\ \psi\ \mu \downarrow \mathbf{Conv}(B)\}$
- $\mathbf{Conv}(\mathbf{t}) = \{\mathbf{Can} \circ \mu\}$
- $\mathbf{Conv}(A \oplus B) = \{\mathbf{Inl} \circ \mu;\ \mu \in \mathbf{Conv}(A)\} \cup \{\mathbf{Inr} \circ \mu;\ \mu \in \mathbf{Conv}(B)\}$
- $\mathbf{Conv}(\mathbf{0}) = \emptyset$

Par récurrence sur les *combinateurs*, et sur les *combinateurs canoniques*, on vérifie successivement:

Lemme 7 *Pour tout combinateur $\varphi : A \to B$ et $\mu \in \mathbf{Conv}(A)$, $\varphi\ \mu \downarrow \mathbf{Conv}(B)$.*

Lemme 8 *Pour tout A, $\mathbf{Conv}(A)$ est l'ensemble de* tous *les* combinateurs canoniques *de type A.*

Le *théorème de terminaison* est une conséquence de ces deux lemmes.

Annexe H

Une Maquette d'Implantation en CAML

Cette maquette comprend:

- un compilateur du *noyau* de LIVE (section 3.2)
- une syntaxe concrète YACC pour l'interface
- une simulation de la *Machine Linéaire* en CAML
- un traducteur du code LAM en assembleur LLM3.

Il n'y a pas de *boucle interactive*, ni de *contrôle des types.*

Pour la *démonstration*, on dispose de 4 fonctions:

- `EVAL` qui compile et exécute le code sur la machine simulée (avec détection des erreurs d'exécution)
- `LAM` qui imprime le *code linéaire*
- `LLM3` qui imprime l'assembleur LLM3
- `EXEC` qui compile et exécute le code sur la LLM3.

H.1 Syntaxe Abstraite des Termes

Elle est décrite dans la section 3.2.

Le fichier `term.ml`:

```
type term = Pair of term & term | Void | Var of string
          | Integer of num | Boolean of bool     (* constants *)
          | Operate of operation & term & term   (* arithmetic operation *)
          | Compare of comparison & term & term  (* arithmetic comparison *)
          | Cond of term & term & term           (* conditional *)
          | Local of pattern & term & term       (* local binding *)
          | Scheme of pattern & term             (* scheme abstraction *)
          | On of term & term                    (* scheme application *)
          | Funcall of string & term             (* function call *)
```

```
 and operation = Plus | Times | Diff

 and comparison = Eq | Neq | Lt | Gt | Le | Ge

 and pattern = pair of pattern & pattern | void | var of string list;;


(* Free variables (according to the linear constraints) *)

let remove x = removerec where rec removerec =

 function y::Y  -> if x=y then Y else y::removerec Y
        | []    -> failwith "unused " ^ x;;


let rec free = function Pair(u,v)       -> free u @ free v
                      | Var x           -> [x]
                      | Operate(_,u,v)  -> free u @ free v
                      | Compare(_,u,v)  -> free u @ free v
                      | Cond(u,v,_)     -> free u @ free v
                      | Local(p,u,v)    -> free u @ bind p (free v)
                      | Scheme(p,u)     -> bind p (free u)
                      | On(u,v)         -> free u @ free v
                      | Funcall(_,u)    -> free u
                      | _               -> []

 and bind = function pair(p,q)  -> bind p o bind q
                   | void       -> I
                   | var X      -> list_it remove X;;
```

H.2 Syntaxe Concrète

Comme il n'y a pas de boucle interactive, les programmes sont de la forme:

function *identifier pattern* → *term* **and** ... **and** *identifier pattern* → *term* **in** *term*

Le fichier `live.mly`:

```
/* function in and () on <> <= >= if then else let scheme -> _ as */
%token FUNCTION IN AND VOID ON NEQ LE GE IF THEN ELSE LET SCHEME ARROW US AS
%token IDENT NUM BOOL

%%

program : FUNCTION defs IN term         {$2,$4}
        | term                          {[],$1}
        ;

def     : IDENT pat ARROW term          {$1,$2,$4}
        ;
```

```
defs    : defs AND def                  {$3::$1}
        | def                           {[$1]}
        ;

term0   : '(' term ')'                  {$2}
        | VOID                          {Void}
        | IDENT                         {Var $1}
        | NUM                           {Integer $1}
        | BOOL                          {Boolean $1}
        ;

term1   : IDENT term1                   {Funcall($1,$2)}
        | term0
        ;

term2   : term2 ON term1                {On($1,$3)}
        | term1
        ;

term3   : '-' term3                     {Operate(Diff,Integer 0,$2)}
        | term2
        ;

term4   : term4 '*' term3               {Operate(Times,$1,$3)}
        | term3
        ;

term5   : term5 '+' term4               {Operate(Plus,$1,$3)}
        | term5 '-' term4               {Operate(Diff,$1,$3)}
        | term4
        ;

term6   : term5 '=' term5               {Compare(Eq,$1,$3)}
        | term5 NEQ term5               {Compare(Neq,$1,$3)}
        | term5 '<' term5               {Compare(Lt,$1,$3)}
        | term5 '>' term5               {Compare(Gt,$1,$3)}
        | term5 LE term5                {Compare(Le,$1,$3)}
        | term5 GE term5                {Compare(Ge,$1,$3)}
        | term5
        ;

term7   : term6 ',' term                {Pair($1,$3)}
        | term6
        ;

term    : IF term THEN term ELSE term   {Cond($2,$4,$6)}
        | LET pat '=' term IN term      {Local($2,$4,$6)}
        | SCHEME pat ARROW term         {Scheme($2,$4)}
        | term7
        ;

pat0    : '(' pat ')'                   {$2}
        | VOID                          {void}
        | US                            {var[]}
        | names                         {var $1}
        ;
```

```
pat     : pat0 ',' pat                  {pair($1,$3)}
        | pat0
        ;

names   : names AS IDENT                {$3::$1}
        | IDENT                         {[$1]}
        ;

%%
```

H.3 La Machine Linéaire

Elle est décrite dans la section 3.3.

Le fichier `lam.ml`:

```
type instruction =

   Swap | Swaap | Sswap          (* transposition *)
 | Cons | Split                  (* pair construction and destruction *)
 | Push | Pop                    (* void construction and destruction *)
 | Int of num | Bool of bool     (* atomic introduction *)
 | Copy | Erase                  (* arithmetic duplication and deletion *)
 | Op of operation               (* arithmetic operation *)
 | Cmp of comparison             (* arithmetic comparison *)
 | Branch of code & code         (* conditional branch *)
 | Cur of code | App             (* scheme construction and destruction *)
 | Call of code

 and code == instruction list;;


(* Simulation *)

type word = cons of word & word | nil
          | int of num | bool of bool
          | clos of code & word          (* closure *)
          | ret of code;;                (* return address *)

let lam_op = function Plus -> op + | Times -> op * | Diff -> op -;;

let lam_cmp = function Eq  -> op =  | Lt  -> op < | Le  -> op <=
                     | Neq -> op <> | Gt  -> op > | Ge  -> op >=;;


let rec lam =

 function c::C,uS       -> lam(match c,uS with

             Swap,u,v::S                -> C,v,u::S
           | Swaap,u,v::w::S             -> C,w,v::u::S
```

```
        | Sswap,u,v::w::S             -> C,u,w::v::S
        | Cons,u,v::S                 -> C,cons(u,v),S
        | Split,cons(u,v),S           -> C,u,v::S
        | Push,u,S                    -> C,nil,u::S
        | Pop,nil,u::S                -> C,u,S
        | Int i,u,S                   -> C,int i,u::S
        | Bool b,u,S                  -> C,bool b,u::S
        | Copy,(int _ as u),S         -> C,u,u::S
        | Erase,int _,u::S            -> C,u,S
        | Op f,int i,int j::S         -> C,int(lam_op f (i,j)),S
        | Cmp f,int i,int j::S        -> C,bool(lam_cmp f (i,j)),S
        | Branch(C',C''),bool b,u::S -> (if b then C' else C''),u,ret C::S
        | Cur C',u,S                  -> C,clos(C',u),S
        | App,clos(C',u),v::S         -> C',cons(u,v),ret C::S
        | Call C',u,S                 -> C',u,ret C::S
        | _                           -> failwith "execution error")

    | [],u,ret C::S -> lam(C,u,S)

    | [],u,[]       -> u;;          (* to return the final result *)
```

H.4 Génération du Code et Optimisations

Pour éviter de concaténer du code, on utilise des *listes incomplètes* représentées par des fonctions de code dans code (voir section 3.1). En prime, on a les *optimisations locales* (section 3.4). Notez l'élégance de cette méthode.

Le fichier code.ml:

```
let swap_op = function Plus -> Plus | Times -> Times;;

let swap_cmp = function Eq      -> Eq   | Lt    -> Gt   | Le    -> Ge
                      | Neq     -> Neq  | Gt    -> Lt   | Ge    -> Le;;



let rec SWAP = function Swap::C          -> C
                      | Swaap::C         -> Sswap::SWAP C
                      | Sswap::C         -> Swaap::SWAP C
                      | Op f::C          -> Op(swap_op f)::C ? Swap::Op f::C
                      | Cmp f::C         -> Cmp(swap_cmp f)::C
                      | C                -> Swap::C

 and SWAAP = function Swaap::C  -> C
                    | Sswap::C  -> SSWAP(SWAP C)
                    | C         -> Swaap::C

 and SSWAP = function Sswap::C  -> C
                    | Swaap::C  -> SWAAP(SWAP C)
                    | C         -> Sswap::C;;
```

```
let CONS = function Split::C    -> C
                  | C           -> Cons::C;;

let SPLIT = function Cons::C    -> C
                   | C          -> Split::C;;

let XCONS = SWAP o CONS and XSPLIT = SPLIT o SWAP;;

let PUSH = function Pop::C      -> C
                  | C           -> Push::C;;

let POP = function Push::C      -> C
                 | C            -> Pop::C;;

let INT i C = Int i::C;;

let BOOL b C = Bool b::C;;

let COPY = function Swap::C             -> Copy::C
                  | Swaap::Swap::C      -> Copy::Sswap::C
                  | Sswap::Swap::C      -> Copy::Swaap::C
                  | C                   -> Copy::C;;

let ERASE C = Erase::C;;

let OP f C = Op f::C;;

let CMP f C = Cmp f::C;;

let BRANCH(F',F'') C = Branch(F'[],F''[])::C;;

let CUR F' C = Cur(F'[])::C;;

let APP C = App::C;;

let CALL C' C = Call C'::C;;
```

H.5 Code de Gestion

Le calcul du *code de distribution* `split` est assez compliqué, à cause des *optimisations globales* (section 3.4). A partir d'une liste de variables `X` et d'un motif `p`, on construit un triplet `Y,F,U`, où `Y` est la liste des variables qui ne sont pas dans `p`, `F` est le *code de distribution*, et `U` est la décomposition obtenue (de type `splitting`).

Le fichier `arrange.ml`:

```
let rec pop = function pair(p,q)        -> SPLIT o (pop p) o (pop q)
                     | void             -> POP
                     | var[]            -> ERASE
                     | var(x::_)        -> failwith "unused variable " ^ x;;
```

```
let rec prune p = cut p ? I,p and cut(pair(p,q)) =

 let F,G,r = SPLIT o pop p,prune q ? XSPLIT o pop q,prune p in F o G,r;;

type splitting = AR of pattern & pattern        (* the first part accepted *)
               | RA of pattern & pattern        (* the first part rejected *)
               | Accept of pattern              (* the whole accepted *)
               | Reject of pattern              (* the whole rejected *);;

let rec split(X,p) = let F,p0 = prune p in match p0 with

   pair(q,r)     -> let Y,G,U = split(X,q) in
                    let Z,H,V = split(Y,r) in Z,(match U,V with

    AR(q1,q2),AR(r1,r2) -> F o SPLIT o G o SWAAP o H o SWAAP o CONS o SWAAP o
                           CONS, AR(pair(q1,r1),pair(q2,r2))

  | RA(q2,q1),RA(r2,r1) -> F o SPLIT o G o SWAAP o H o SWAAP o CONS o SWAAP o
                           CONS, RA(pair(q2,r2),pair(q1,r1))

  | AR(q1,q2),RA(r2,r1) -> F o SPLIT o G o SWAAP o H o SSWAP o CONS o SWAAP o
                           CONS, AR(pair(q1,r1),pair(r2,q2))

  | RA(q2,q1),AR(r1,r2) -> F o SPLIT o G o SWAAP o H o SSWAP o CONS o SWAAP o
                           CONS, RA(pair(q2,r2),pair(r1,q1))

  | AR(q1,q2),Accept _  -> F o SPLIT o G o SSWAP o CONS, AR(pair(q1,r),q2)
  | AR(q1,q2),Reject _  -> F o SPLIT o G o SWAAP o CONS, RA(pair(r,q2),q1)
  | RA(q2,q1),Accept _  -> F o SPLIT o G o SWAAP o CONS, AR(pair(r,q1),q2)
  | RA(q2,q1),Reject _  -> F o SPLIT o G o SSWAP o CONS, RA(pair(q2,r),q1)

  | Accept _,AR(r1,r2)  -> F o XSPLIT o H o SSWAP o CONS, AR(pair(r1,q),r2)
  | Reject _,AR(r1,r2)  -> F o XSPLIT o H o SWAAP o CONS, RA(pair(q,r2),r1)
  | Accept _,RA(r2,r1)  -> F o XSPLIT o H o SWAAP o CONS, AR(pair(q,r1),r2)
  | Reject _,RA(r2,r1)  -> F o XSPLIT o H o SSWAP o CONS, RA(pair(r2,q),r1)

  | Accept _,Reject _   -> F o SPLIT, AR(q,r)
  | Reject _,Accept _   -> F o SPLIT, RA(q,r)

  | Accept _,Accept _   -> F, Accept p0
  | Reject _,Reject _   -> F, Reject p0)

 | void          -> X,F,Accept p0

 | var Y         -> (match partition Y with

                       X',_,[]       -> X',F,Accept p0
                     | X',[],_       -> X',F,Reject p0
                     | X',Y1,Y2      -> X',F o COPY,AR(var Y1,var Y2))

    where rec partition = function y::Y -> let X',Y1,Y2 = partition Y in
                                           remove y X',y::Y1,Y2 ? X',Y1,y::Y2

                                  | []  -> X,[],[];;
```

H.6 Compilation

On applique les principes de la section3.3, en utilisant des *listes incomplètes.*

Remarquez l'utilisation de la récursion dans le programme (très astucieux) qui construit le code bouclé à partir de définitions récursives.

Le fichier `compile.ml`:

```
(* Compilation of a term according to a static and a dynamic environment *)

let compile_term(E,p,u) = comp p u [] where rec comp p =

 (function Pair uv              -> (binary uv) o CONS

         | Void                 -> (pop p) o PUSH

         | Var x                -> let F,q = prune p in (match bind q [x] with

                                          [] -> F | _ -> failwith "unbound " ^ x)

         | Integer i            -> (pop p) o INT i

         | Boolean b            -> (pop p) o BOOL b

         | Operate(f,uv)        -> (binary uv) o OP f

         | Compare(f,uv)        -> (binary uv) o CMP f

         | Cond(u,v,v')         -> let F,q = unary u in

                                   F o BRANCH(comp q v,comp q v')

         | Local(q,u,v)         -> let F,r = unary u in

                                   F o XCONS o (comp(pair(r,q)) v)

         | Scheme(q,u)          -> let F,r = prune p in

                                   F o CUR(comp(pair(r,q)) u)

         | On uv                -> (binary uv) o APP

         | Funcall(x,u)         -> (comp p u) o

                                   CALL(assoc x E ? failwith "undefined " ^ x))

where binary(u,v) = let _,F,V = split(free v,p) in match V with

          AR(p1,p2)      -> F o (comp p1 v) o SWAP o (comp p2 u)
        | RA(p2,p1)      -> F o (comp p2 u) o SWAP o (comp p1 v) o SWAP
        | Accept p0      -> F o (comp p0 v) o PUSH o (comp void u)
        | Reject p0      -> F o (comp p0 u) o PUSH o (comp void v) o SWAP
```

```
and unary u = let _,F,U = split(free u,p) in match U with

          AR(p1,p2)      -> F o (comp p1 u), p2
        | RA(p2,p1)      -> F o SWAP o (comp p1 u), p2
        | Accept p0      -> F o (comp p0 u) o PUSH o SWAP, void
        | Reject p0      -> F o PUSH o (comp void u), p0;;

(* Compilation of a program with recursive definitions *)

let compile_program(U,u) = compilerec U ([],void,u) where rec compilerec =

 function []             -> compile_term

        | (x,qv)::V      -> let compile = compilerec V in

  function E,pu -> let rec C = compile((x,C)::E,qv) in compile((x,C)::E,pu);;

                         (* here we make looping code *)
```

H.7 Impression

Pour imprimer le code LAM, il faut "départager" une structure bouclée et générer des étiquettes.

Le fichier `print.ml`:

```
(* Printing values *)

let print_word = message o stringrec where rec stringrec =

 function cons(u,v)      -> "(" ^ stringrec u ^ "," ^ stringrec v ^ ")"
        | nil            -> "()"
        | int i          -> string_of_num i
        | bool b         -> string_of_bool b
        | clos _         -> "-";;                (* closures are not printed *)


(* Printing LAM code *)

let string_op = function Plus -> "+" | Diff -> "-" | Times -> "*";;

let string_cmp = function Eq    -> "="  | Lt    -> "<"  | Le    -> "<="
                        | Neq   -> "<>" | Gt    -> ">"  | Ge    -> ">=";;

let string_ins =

 function Swap  -> "Swap"       | Int i  -> string_of_num i
        | Swaap -> "Swaap"      | Bool b -> string_of_bool b
        | Sswap -> "Sswap"      | Copy   -> "Copy"
        | Cons  -> "Cons"       | Erase  -> "Erase"
        | Split -> "Split"      | Op f   -> string_op f
        | Push  -> "Push"       | Cmp f  -> string_cmp f
        | Pop   -> "Pop"        | App    -> "App";;
```

```
let print_code C = let r = ref 0 and L = ref([] : (code & string)list) in
                   let new() = " @" ^ string_of_num(incr r) in

                   print_newline(); printrec C where rec printrec =

 function Branch(C',C'')::C      -> let n',n'' = new(),new() in

                                    print_string("Branch" ^ n' ^ n'' ^ "; ");
                                    printrec C; labrec(n',C'); labrec(n'',C'')

       | Cur C'::C               -> let n' = new() in

                                    print_string("Cur" ^ n' ^ "; ");
                                    printrec C; labrec(n',C')

       | Call C'::C              -> (let n' = assq C' !L in

                                     print_string("Call" ^ n' ^ "; ");
                                     printrec C) ?

                                    let n' = new() in L := (C',n')::!L;

                                    print_string("Call" ^ n' ^ "; ");
                                    printrec C; labrec(n',C')

       | c::C                    -> print_string(string_ins c ^ "; ");
                                    printrec C

       | []                      -> message "Return"; print_newline()

 and labrec(n,C) = print_string(n ^ ": "); printrec C;;
```

H.8 Expansion

La machine LLM3 [LELISP] possède 4 registres a1,...,a4. Ici, a1 est utilisé comme registre principal et a4 contient la *liste des cellules libres* (*free list*).

Notez surtout les traductions de **Cons** et **Split** (qui sont parfaitement symétriques) ainsi que **App** qui récupère un couple pour en faire une fermeture.

Ici encore, on utilise des *listes incomplètes*, ce qui justifie l'introduction d'une nouvelle syntaxe pour l'assembleur LLM3.

On a un fichier de *macro-instructions LLM3*, avec, en particulier, une instruction d'initialisation de la machine, qui construit la *free list*.

Le fichier expand.ml:

```
directive use_syntax "obj";;

let cons_to_obj x y = obj_cons(x,y);;

let obj_of_num = obj_int o int_of_num;;

let obj_of_bool = function true -> <<t>> | false -> <<()>>;;
```

```
let lap_op = function Plus -> <<plus>> | Diff -> <<diff>> | Times -> <<times>>;;

let lap_cmp f = obj_of_string("lam_" ^ string_cmp f);;


directive use_syntax "lap";;

let lap_ins = function Swap     -> <<(xtopst a1)>>

                     | Swaap    -> <<(pop a2) (xtopst a1) (push a2)>>

                     | Sswap    -> <<(pop a2) (xtopst a2) (push a2)>>

                     | Cons     -> <<(mov a1 (car a4)) (mov a4 a1)
                                     (mov (cdr a1) a4) (pop (cdr a1))>>

                     | Split    -> <<(push (cdr a1)) (mov a4 (cdr a1))
                                     (mov a1 a4) (mov (car a4) a1)>>

                     | Push     -> <<(push a1) (mov '() a1)>>

                     | Pop      -> <<(pop a1)>>

                     | Int i    -> <<(push a1) (mov '{^ obj_of_num i ^} a1)>>

                     | Bool b   -> <<(push a1) (mov '{^ obj_of_bool b ^} a1)>>

                     | Copy     -> <<(push a1)>>

                     | Erase    -> <<(pop a1)>>

                     | Op f     -> <<(pop a2) ({^ lap_op f ^} a2 a1)>>

                     | Cmp f    -> <<(pop a2) (jcall {^ lap_cmp f ^})>>

                     | App      -> <<(mov (cdr a1) a2) (pop (cdr a1))
                                     (jcall lam_app)>>;;


let lap_code C = let r = ref 0 and L = ref([] : (code & obj) list) in
                 let new() = obj_of_num(incr r) in

       <<(fentry mleval subr0) (jcall lam_init)>> (laprec C <:obj<((end))>>)

 where rec laprec =

 function Branch(C',C''')::C     -> let n,n'' = new(),new() in

                      <<(mov a1 a2) (pop a1) (push (@ ^n)) (btnil a2 ^n'')>>
                           o (laprec C') o lab(n'',C''') o lab(n,C)
```

```
        | Cur C'::C              -> let n' = new() in

                                <<(mov a1 (car a4)) (mov a4 a1) (mov (cdr a1) a4)
                                  (mov (@ ^n') (cdr a1))>> o (laprec C) o lab(n',C')

        | Call C'::C             -> (let n' = assq C' !L in

                                      <<(call ^n')>> o (laprec C)) ?

                                     let n' = new() in L := (C',n')::!L;

                                     <<(call ^n')>> o (laprec C) o lab(n',C')

        | c::C                   -> lap_ins c o (laprec C)

        | []                     -> <<(return)>>

 and lab(n,C) = (cons_to_obj n) o (laprec C);;
```

Le fichier `lap.mly`:

```
%mlescape
/* () ' */
%token NIL QUOTE
%token IDENT

%%

lap     : lap obj        {$1 o cons_to_obj $2}
        | obj            {cons_to_obj $1}
        ;

obj     : '(' list ')'   {$2}
        | NIL            {obj_nil}
        | IDENT          {obj_of_string $1}
        | '@'            {obj_of_string "@"}
        | QUOTE obj      {obj_cons(obj_of_string "quote",obj_cons($2,obj_nil))}
        | mlescape
        ;

list    : obj list       {obj_cons($1,$2)}
        | obj            {obj_cons($1,obj_nil)}
        ;

%%
```

Le fichier `macro.lo`:

```
(loader '((fentry lam_= subr2) (cnbeq a1 a2 1) (mov '() a1) (return)
          (fentry lam_<> subr2) (cnbne a1 a2 1) (mov '() a1) (return)
          (fentry lam_< subr2) (cnblt a1 a2 1) (mov '() a1) (return)
```

```
(fentry lam_> subr2) (cnbgt a1 a2 1) (mov '() a1) (return)
(fentry lam_<= subr2) (cnble a1 a2 1) (mov '() a1) (return)
(fentry lam_>= subr2) (cnbge a1 a2 1) (mov '() a1) (return)

1 (mov 't a1) (return)

(fentry lam_app subr2) (bri a2)

(fentry lam_init subr0) (mov '1000 a1) (mov (mlval "replicate") a2)
(jcall ml_appl) (mov '() a2) (jcall ml_iappl) (mov a1 a4)
(mov '() a1) (return) (end)) ())
```

H.9 Le Système

Le fichier `live.ml`:

```
directive map use ["term";"lam";"code";"arrange";"compile";"print";"expand"];;

let EVAL P = print_word(lam(compile_program P,nil,[ret[]]));;

let LAM = print_code o compile_program;;

let LLM3 = lap_code o compile_program;;

let EXEC P = ml_loader(LLM3 P,false); run();;

use "macro"; use_syntax "live";;
```

Annexe I

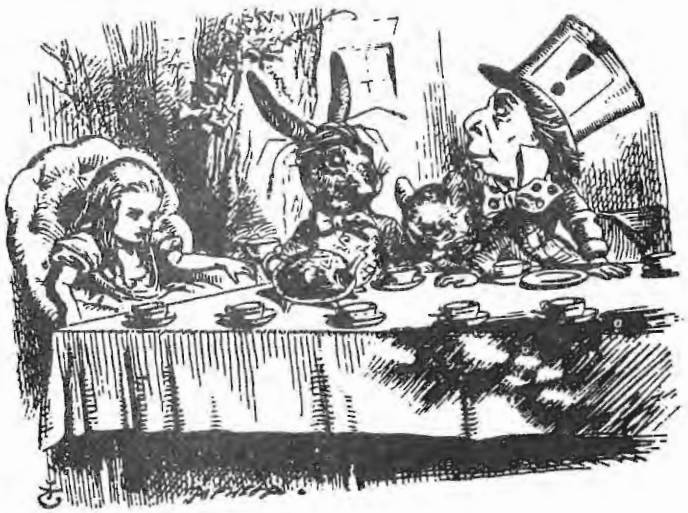
Exemples

On a repris les exemples *factorielle* et *quicksort* de la section 3.1. En l'absence des types concrets, on utilise un *codage* (`(true,())` pour [], et `(false,(x,X))` pour `x::X`).

```
(* let FACT =

<<function fact n as n -> if n = 0 then let _ = n in 1
                                   else let n as n = n in
                                        n * (fact (n - 1)) in
  fact 5>>;; *)

#EVAL FACT;;
120
() : void

#LAM FACT;;

Pop; 5; Call @1; Return

 @1: Copy; 0; =; Branch @2 @3; Return

 @2: Erase; 1; Return

 @3: Copy; 1; Swap; -; Call @1; *; Return

() : void

#LLM3 FACT;;
<<((fentry mleval subr0) (jcall lam_init) (pop a1) (push a1)
   (mov (quote 5) a1) (call 1) (return) 1 (push a1) (push a1)
   (mov (quote 0) a1) (pop a2) (jcall lam_=) (mov a1 a2) (pop a1)
   (push (@ 2)) (btnil a2 3) (pop a1) (push a1) (mov (quote 1) a1)
   (return) 3 (push a1) (push a1) (mov (quote 1) a1) (xtopst a1)
   (pop a2) (diff a2 a1) (call 1) (pop a2) (times a2 a1) (return) 2
   (return) (end))>>
 : obj

#EXEC FACT;;
<<120>> : obj
```

```
(* let QUICKSORT =

<<function nil u -> true,u and cons u -> false,u

 and partition y,n,u -> if n then let () = u in y,nil(),nil()

                              else let x,X = u in
                                   let c,x,y = compare(x,y) in
                                   let y,U,V = partition(y,X) in
                                   y,(if c then cons(x,U),V else U,cons(x,V))

 and quicksort n,u ->

  scheme Y -> if n then let () = u in Y

                   else let x,U,V = partition u in
                        quicksort U on cons(x,quicksort V on Y)

 and compare(p as p,q as q) -> p<q,p,q in

  quicksort(cons(1,cons(9,cons(8,cons(7,nil())))))) on nil()>>;; *)

#EVAL QUICKSORT;;
(false,(1,(false,(7,(false,(8,(false,(9,(true,()))))))))))
() : void

#LAM QUICKSORT;;

Call @1; Push; Call @1; 7; Cons; Call @2; 8; Cons; Call @2; 9; Cons; Call @2;
1; Cons; Call @2; Call @3; App; Return

 @3: Cur @4; Return

 @4: Split; Split; Swaap; Cons; Swap; Branch @5 @6; Return

 @5: Split; Swap; Pop; Return

 @6: Split; Swap; Call @7; Split; Swap; Split; Swaap; Cons; Sswap; Split;
Swaap; Swap; Call @3; App; Swap; Cons; Call @8; Swap; Call @3; App; Return

 @8: false; Cons; Return

 @7: Split; Swap; Split; Swaap; Cons; Swap; Branch @9 @10; Return

 @9: Split; Swap; Call @11; Push; Call @11; Cons; Swap; Cons; Return

 @11: true; Cons; Return

 @10: Split; Swap; Split; Sswap; Cons; Call @12; Split; Swap; Split; Sswap;
Cons; Swaap; Swap; Cons; Call @7; Split; Swaap; Split; Sswap; Cons; Swap;
Branch @13 @14; Swap; Cons; Return

 @13: Split; Swap; Split; Sswap; Swap; Cons; Call @15; Cons; Return

 @15: false; Cons; Return
```

```
 @14: Split; Swap; Split; Swaap; Cons; Call @15; Swap; Cons; Return

 @12: Split; Copy; Swaap; Copy; Swaap; Cons; Swaap; Cons; Swap; Split; <;
Cons; Return

 @2: false; Cons; Return

 @1: true; Cons; Return

() : void

#EXEC QUICKSORT;;
<<(() 1 () 7 () 8 () 9 t)>> : obj
```



Bibliographie

[BreaCoq] V. Breazu-Tannen & T. Coquand, Extensional Models for Polymorphism, in: *TAPSOFT '87, Volume* **2**, LNCS **250** (Springer-Verlag, Pisa) 291-307.

[CAML] The CAML Primer & CAML, The Reference Manual, Projet Formel, INRIA-ENS (1987).

[Cartmell] J. Cartmell, Generalised Algebraic Theorie and Contextual Categories, Thesis (Oxford,1978).

[CoqEhr] T. Coquand & T. Ehrhard, An equational presentation of higher order logic, in: D.H. Pitt, A. Poigné & D.E. Rydeheard, eds., *Category Theory and Computer Science* LNCS **283** (Springer Verlag, 1987) 40-56.

[CouCurMau] G. Cousineau, P.L. Curien & M. Mauny, The Categorical Abstract Machine, in: J. P. Jouannaud, ed., *Functional Programming Languages and Computer Architecture*, LNCS **201** (Springer-Verlag, 1985) 50-64.

[CouCurMauSu] G. Cousineau, P.L. Curien, M. Mauny & A. Suárez, Combinateurs Catégoriques et Implémentation des Langages Fonctionnels (1986).

[Church40] A. Church, A formulation of the simple theory of types, *Journal of Symbolic Logic* **5,1** (1940) 56–68.

[Church41] A. Church, *The Calculi of Lambda-Conversion* (Princeton U. Press, 1941).

[Curien83] P.L. Curien. Combinateurs catégoriques, algorithmes séquentiels et programmation applicative, Thèse de Doctorat d'Etat, Université de Paris VII (1983).

[Curien85] P. L. Curien, Categorical Combinatory Logic, in: W. Brauer, ed., *ICALP 85*, LNCS **194** (Springer-Verlag, Nafplion) 130-139.

[Curien86] P. L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming* (Pitman, 1986).

[CurFeys] H. B. Curry & R. Feys, *Combinatory Logic Vol. I* (North-Holland, Amsterdam,1958).

[Fox] T. Fox, The Coalgebra Enrichment of Algebraic Categories, *Comm. in Algebra* **9** 3 (1981) 223-234.

[Gentzen] *The collected Papers of Gerhard Gentzen*, E. Szabo, ed. (North-Holland, Amsterdam, 1969).

[Girard71] J.Y. Girard, Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types, in: J.E. Fenstad, ed., *Proc. 2nd Scandinavian Logic Symposium* (North-Holland, 1971) 63-92.

[Girard86] J.Y. Girard, Linear logic and Parallelism, *Proceedings of the School on semantics of Parallelism held in IAC* (CNR, Roma, 1986).

[Girard87] J.Y. Girard, Linear logic, *Theoretical Computer Science* **50** (1987) 1-102.

[Girard87a] J.Y. Girard, Lambda calcul typé, Cours de D.E.A. 86-87, Université de Paris VII.

[GirLaf] J.Y. Girard & Y. Lafont, Linear Logic and Lazy Computation, in: *TAPSOFT '87, Volume* **2**, LNCS **250** (Springer-Verlag, Pisa) 52-66.

[Huet] G. Huet, Initiation à la Théorie des Catégories, INRIA (1985).

[HuPoi] H. Huwig & A. Poigné, A note on inconsistencies caused by fixpoints in a cartesian closed category.

[Kelly] G.M. Kelly, On Maclane's conditions for coherence of natural associativities, *J. Algebra* **1** (1964) 397-402.

[Kleene] S.C. Kleene, *Introduction to Meta-mathematics* (North Holland, 1952).

[Lafont87] Y. Lafont, Linear Logic Programming, in: P. Dybjer, B. Nordström, K. Petersson, Jan M. Smith eds., *Proceedings of the Workshop on Programming Logic* Report **37** (Göteborg, 1987).

[Lambek68] J. Lambek, Deductive systems and categories, *Math. Systems Theory* (1968).

[Lambek80] J. Lambek, From Lambda-calculus to Cartesian Closed Categories, in: J.P. Seldin & J.R. Hindley eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism* (Academic Press, 1980).

[LamSco] J. Lambek & P.J. Scott, *Introduction to higher order categorical logic*, Cambridge studies in advanced mathematics (Cambridge University Press, 1986).

[LELISP] J. Chailloux, Le_Lisp Version 15.2, Le Manuel de Référence, INRIA (1986).

[Maclane] S. MacLane, *Categories for the working mathematician*, Graduate Texts in Mathematics **5** (Springer-Verlag, 1971).

[Martinloef] P. MartinLöf, *Intuitionistic Type Theory*, Studies in Proof Theory, Lecture Notes (Bibliopolis, Napoli, 1984).

[MauSu] M. Mauny & A. Suarez, Implementing Functional Languages in the Categorical Abstract Machine, in : *Proceedings of the Lisp and Functional Programming Conference* (ACM, Boston, 1986).

[Mauny] M. Mauny, Compilation des langages fonctionnels dans les combinateurs catégoriques, Application au langage ML, Thèse de Troisième Cycle, Université de Paris VII (1985).

[Prawitz] D. Prawitz, *Natural Deduction* (Almqist and Wiskell, Stockolm, 1965).

[Seely84] R.A.G. Seely, Locally cartesian closed categories and type theory, *Math. Proc. Camb. Philos. Soc.* **95** (1984) 33-48.

[Seely86] R.A.G. Seely, Categorical Semantics for Higher Order Polymorphic Lambda Calculus (1986).

[Suarez] A. Suarez, Une implémentation de ML en ML, Thèse (1988).

[Szabo] M.E. Szabo, *Algebra of proofs*, Studies in Logic and the Foundations of Mathematics **88** (North-Holland, Amsterdam, 1978).

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

- Résumé -

Cette thèse (doctorat d'université) comporte trois chapitres

De la Déduction Naturelle à La Machine Catégorique

CAML est un langage fonctionnel interactif compilé particulièrement adapté au développement de logiciels complexes, tels que :

. langages de programmation
. systèmes de preuve
. maquettes de systèmes experts ...

L'implantation de CAML est fondée sur la Machine Catégorique Abstraite (CouCurMau), elle-même issue des combinateurs catégoriques de J. Lambek.
Nous donnons les fondements mathématiques de ce type d'implantation :

. Le mécanisme d'évaluation catégorique est obtenu à partir de la preuve d'un résultat mathématique.

. L'exécution d'un programme n'est pas un processus de normalisation.

Ce premier chapitre constitue une vue d'ensemble sur les rapports entre logique, théorie des catégories et informatique dans le cas de la programmation fonctionnelle.

La Machine Linéaire Abstraite

Dès 1986, Jean-Yves Girard commence à populariser sa Logique Linéaire en participant à plusieurs colloques d'informatique théorique. Son travail (Girard87) a suscité beaucoup d'intérêt dans la communauté informatique, car il s'agit d'une approche radicalement nouvelle dans le domaine des fondements de la programmation.

A la lumière de nos travaux sur la Machine Catégorique, nous y trouvons la solution de problèmes qui accompagnent la programmation fonctionnelle depuis son origine :

. Les traits impurs (effets de bord, affectations,...)
. La récupération de mémoire (garbage collection).
. Les deux modes d'évaluation (strict et paresseux)

Le deuxième chapitre contient une introduction à cette nouvelle logique, ainsi que les principes de la Machine Linéaire.

Un Langage Linéaire

LIVE (la version linéaire de CAML) est un nouveau langage qui intègre des caractéristiques non fonctionnelles, tout en conservant la nature déclarative des langages fonctionnels purs. Il est implanté sur la Machine Linéaire (pas besoin de garbage collector!).

La description de cette implantation constitue le troisième et dernier chapitre qui présuppose une certaine familiarité avec CAML.

Mots-Clefs : Calcul des Séquents - Catégories Cartésienne Fermée - Catégorie Monoïdale - Combinateur Catégorique - Déduction Naturelle - Effet de Bord - Evaluation - Lambda-Calcul - Logique Intuitionniste - Logique Linéaire - Machine Abstraite - Programmation Fonctionnelle - Récupération de la Mémoire.

ISBN 2 - 7261 - 0515 - 7