De la deduction naturelle a la machine categorique

Y. Lafont

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 670

DE LA DÉDUCTION NATURELLE À LA MACHINE CATÉGORIQUE

Yves LAFONT

Mai 1987

De la Déduction Naturelle à la Machine Catégorique*

Y. Lafont
INRIA, Projet Formel & Imperial College, Londres

Résumé

On connaît depuis longtemps les liens étroits entre *Logique Intuitionniste*, *λ-Calcul Typé* et *Catégories Cartésiennes Fermées*. Mais on présente souvent l'exécution des programmes de façon *inadéquate*, comme un mécanisme de *reécriture* (*β-réduction* ou *élimination des coupures*).

Le *mécanisme d'évaluation* de la *Machine Catégorique Abstraite* [CouCurMau] est obtenu ici à partir de la preuve d'un *Théorème de Cohérence Hiérarchique*. C'est le prototype d'une implantation *élégante* et *efficace* des langages fonctionnels.

Abstract

The close links between *Intuitionistic Logic*, *Typed λ-Calculus* and *Cartesian Closed Categories* are well-known. But the execution of programs is often inadequately described as a rewriting mechanism (*β-reduction* or *cut elimination*).

The idea of the *evaluation mechanism* of the *Categorical Abstract Machine* [CouCurMau] comes here from the proof of a *Hierarchical Coherence Theorem*. It is the prototype of an *elegant* and *efficient* implementation of functional languages.

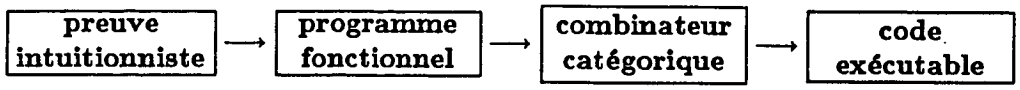
*Première partie de la thèse de doctorat de l'auteur, intitulée "Logiques, Catégories & Machines".

Introduction

Le titre de cet article 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].

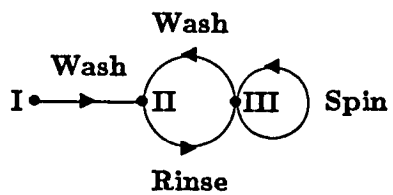
On réalise ainsi le *Programme de Constructivisation des Mathématiques*:



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



Pour nous, un *programme fonctionnel* est un *combinateur catégorique* $\varphi : A \to B$, où A et B sont des formules du calcul propositionnel (section 2).

Pour que φ 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 une suite d'actions primitives $\downarrow\varphi : A \to B$ et une seule telle que $\downarrow\varphi \equiv \varphi$. C'est ce qu'exprime le *Théorème de Cohérence Hiérarchique* (section 3).

C'est pour *démontrer* ce théorème[2] que nous introduisons les *sémantiques* des *formules* et des *combinateurs* qui expliquent le *mécanisme d'évaluation* de la *Machine Catégorique* (section 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* [GirLaf] a été introduite.

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 4 présente la *Machine Catégorique*.

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

[2] Si on traduit les *Combinateurs Catégoriques* dans le *λ-Calcul Typé*, le *Théorème de Cohérence Hiérarchique* se déduit aisément du *Théorème de Normalisation* [Szabo]. Mais la preuve directe est plus instructive.

Pour comprendre les sections suivantes, il est utile d'avoir une certaine familiarité avec les langages fonctionnels [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 5). Quant aux *effets de bords* et aux *échappements*, nous renonçons tout simplement à les intégrer dans notre formalisme (section 6). La dernière section contient quelques remarques sur les *optimisations*.

L'auteur a été fortement influencé par les articles de référence sur la *Machine Catégorique Abstraite* [CouCurMau,CouCurMauSu,MauSu] ainsi que par des discussions fructueuses avec plusieurs membres du *Projet Formel*.

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 (B n'est plus considérée comme une hypothèse):

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

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.)

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

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

$$\frac{\begin{array}{cc}\vdots & \vdots \\ A \Rightarrow B & A\end{array}}{B} \qquad\qquad \frac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \Rightarrow B}$$

- 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).

1.2 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:

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

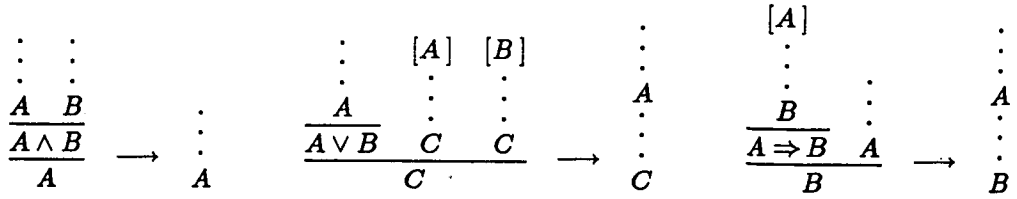
Cette déduction contient une *coupure*[3], c'est à dire une règle d'*élimination* dont la *formule principale* est *conclusion* d'une règle d'*introduction*.

Il y a cinq types de *coupure*:

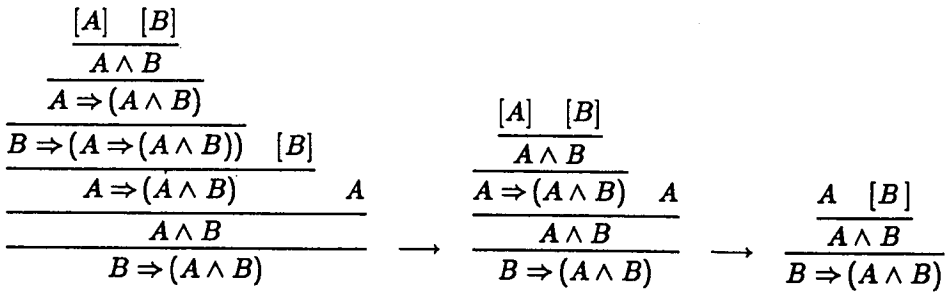
$$\frac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{\dfrac{A \wedge B}{A}} \qquad \frac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{\dfrac{A \wedge B}{B}} \qquad \frac{\begin{array}{ccc}\vdots & [A] & [B] \\ A & \vdots & \vdots \\ A \vee B & C & C\end{array}}{C} \qquad \frac{\begin{array}{ccc}\vdots & [A] & [B] \\ B & \vdots & \vdots \\ A \vee B & C & C\end{array}}{C} \qquad \frac{\begin{array}{cc}[A] & \\ \vdots & \\ B & \vdots \\ A \Rightarrow B & A\end{array}}{B}$$

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

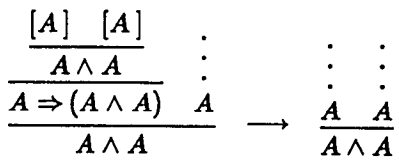
Comme l'une des règles élimine ce que l'autre introduit, la déduction peut être "simplifiée" (*élimination* de la coupure):



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



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'un déduction en *dupliquant* des branches:



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

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[4]. □

En particulier, on a:

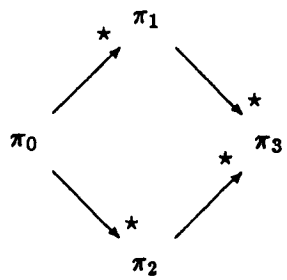
[4]Ce n'est pas vrai pour les éliminations de $\bot$ et $\vee$. C'est pourquoi on doit introduire la notion de *coupure commutative*.

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 clôture 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:

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.*

2 λ-Calcul Typé et Combinateurs Catégoriques

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[5] 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).

Les règles de la *Déduction Naturelle* s'expriment facilement dans ce cadre[6]. Pour simplifier, on se limite aux connecteurs $\wedge$ et $\Rightarrow$ (la disjonction est traitée dans la section 5):

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

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

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

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

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 $\wedge$ 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*[7].

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 3 et 4).

---

[5] 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.

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

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

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 $A_1, \ldots A_n$ pour simplifier la forme des jugements $(A \vdash B)$.

On obtient un système dont les preuves sont des *Combinateurs Catégoriques*:

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

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

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

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

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

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

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

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

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

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

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

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

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.

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* $\varphi : (\cdots((K \wedge A_1) \wedge A_2) \ldots) \wedge A_n \to B$[8].

La traduction des termes en combinateurs sera utilisée pour la compilation des langages fonctionnels (section 4):

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

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

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

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

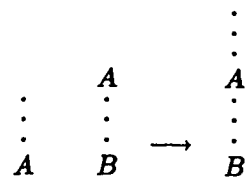
$$\frac{\Gamma \vdash \mu : A \Rightarrow B \longmapsto \varphi : \hat{\Gamma} \to A \Rightarrow B \qquad \Gamma \vdash \nu : A \longmapsto \psi : \hat{\Gamma} \to A}{\Gamma \vdash \mu\, \nu : B \longmapsto \mathbf{App} \circ \langle \varphi, \psi \rangle : \hat{\Gamma} \to B}$$

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

[8] On utilise une formule quelconque K (pas nécessairement $\top$) lorsque l'environnement est vide. Notez que les deux traductions ne sont pas inverses l'une de l'autre.

2.4 Déclarations locales

Dans la section 1, nous avons utilisé la *substitution*:



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:

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

3 Evaluation

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* (appendice 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[9]:

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

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*[10]*. 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.*

Nous donnons en appendice 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).*

3.2 Sémantique fonctionnelle

On donne ici une preuve "élémentaire" du théorème 5, directement issue de la preuve "catégorique" (appendice C).

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[11] $\varphi : A \to B$.

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.

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:

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

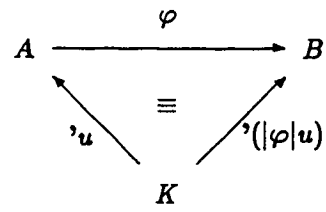
---

[9] 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.

[10] *Surjectif* et *injectif* sur les morphismes.

[11] On distingue les *combinateurs typés* des *morphismes* (qui sont des classes d'équivalence de combinateurs).

($\equiv$ 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 **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*)).

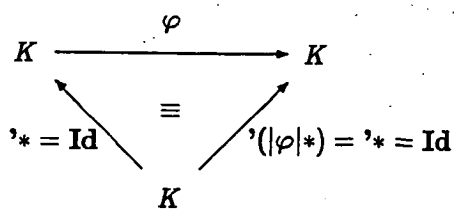
On définit inductivement la ***sémantique*** des combinateurs:

- $|\mathbf{Id}|\,u = u$. 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.

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". 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*.

On a une nouvelle notion de *valeur*:

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

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

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

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

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

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')* [12].

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 $\varphi\, u \triangleright v$.

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

4 La Machine Catégorique

La *relation d'évaluation* de la section 3 peut être considérée comme une *machine abstraite* dont le code est un *combinateur catégorique*.

4.1 Description de la Machine

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

Le code est une liste d'intructions (contigües en mémoire) executées 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 **Cur**(φ) construit une clôture 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:

- 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 $[x_0; x_1; \cdots; x_n]$)

état courant			état suivant		
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

4.2 Supplément à la Machine Catégorique

Le *valeurs* introduites dans la section 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**$(-)$[13]:

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:

code	environnement	pile	code	environnement	pile
Negate :: C	u	S	C	$-u$	S
Plus :: C	(u, v)	S	C	$u + v$	S
Times :: C	(u, v)	S	C	$u \times v$	S

4.3 Compilation des λ-termes.

La compilation est donnée par la *traduction des λ-termes en combinateurs catégoriques* (voir section 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 $\mu[v_1/x_1, \ldots, v_n/x_n]$):

- $[\![x]\!]_{(X,x)} = [\mathbf{Snd}]$
- $[\![x]\!]_{(X,y)} = [\mathbf{Fst}] \mathbin{@} [\![x]\!]_X$
- $[\![\mathbf{fst}\ \mu]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Fst}]$
- $[\![\mathbf{snd}\ \mu]\!]_X = [\![\mu]\!]_X \mathbin{@} [\mathbf{Snd}]$

[13] Notez la similitude entre l'instruction **Quote**(u) et le combinateur 'u de la section 3.

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

Remarque: 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 7.

Voici un exemple emprunté à [CouCurMauSu]:

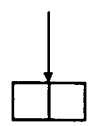
$[\![\mathbf{let}\, f = \mathbf{fun}\, x \to x * x\, \mathbf{in}\, \mathbf{let}\, x = 3\, \mathbf{in}\, f\,(f\,x)]\!]_{\_} = [\mathbf{Push}; \mathbf{Cur}(C); \mathbf{Cons}; \mathbf{Push}; \mathbf{Quote}(3); \mathbf{Cons}; \mathbf{Push}; \mathbf{Fst}; \mathbf{Snd}; \mathbf{Swap}; \mathbf{Push}; \mathbf{Fst};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

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

Le code exécuté (figure 1) donne le résultat escompté (81).

4.4 Le récupérateur de mémoire

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



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

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 *clôture* (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*[14].

[14] 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).

Figure 1: **Exemple d'exécution**

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**]	$(C \cdot \_)$
Times	$(3, 3)$	[**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	[]	
Times	$(9, 9)$	[]		
Return	81	[]		
	81			

5 Conditionnelle et Types Concrets

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:

$$\frac{u \mathrel{\circ}_{\circ} A}{(inl\ u) \mathrel{\circ}_{\circ} A \vee B} \qquad \frac{u \mathrel{\circ}_{\circ} B}{(inr\ u) \mathrel{\circ}_{\circ} A \vee B}$$

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

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

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

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

Mais la traduction de $\mathbf{case}\ \lambda\ \mathbf{of}\ x \to \mu \mid y \to \nu$ en *combinateur catégorique* est compliquée[15]. 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.

[15] $\mathbf{App} \circ \langle \mathbf{Case}(\mathbf{Cur}(\varphi \circ \langle \mathbf{Snd}, \mathbf{Fst} \rangle), \mathbf{Cur}(\psi \circ \langle \mathbf{Snd}, \mathbf{Fst} \rangle)) \circ \chi, \mathbf{Id} \rangle$.

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 $\mathbf{Val}(\mathbf{Bool}) = \{\mathit{true}, \mathit{false}\}$ et:

$$\frac{}{\mathit{true} \mathbin{:\!\circ} \mathbf{Bool}} \qquad \frac{}{\mathit{false} \mathbin{:\!\circ} \mathbf{Bool}}$$

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

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

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

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

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

Cette fois, la traduction ne pose pas de difficulté:

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

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

On en déduit la compilation de la conditionnelle:

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

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

5.3 Types énumérés

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

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

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

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

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).

5.4 Le coproduit comme type concret

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

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

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* $\{\textbf{inl}\} \wedge A \cup \{\textbf{inr}\} \wedge B$. Mais comment donner un sens "catégorique" à l'union de deux types quelconques?
- Comme la somme dépendante $\sum_{m \in \textbf{inl} \mid \textbf{inr}} \textbf{case } m \textbf{ of inl} \to A \mid \textbf{inr} \to B$. 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[16].

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

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

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

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

5.5 Autres types concrets

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

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

Ainsi, on a :
$[\![\mathbf{fun}\ \mathbf{zero} \to \mathbf{zero} \mid (\mathbf{succ}\ x) \to x]\!]_- = [\mathbf{Cur}([\mathbf{Push}; \mathbf{Snd}; \mathbf{Fst}; \mathbf{Cons}; \mathbf{Branch}_2(C', C''); \mathbf{Return}])]$
où:

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

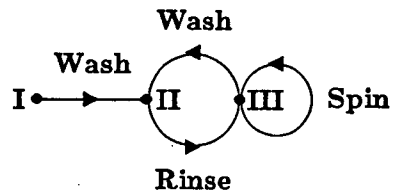
Les *types récursifs* sont utilisés avec des *fonctions récursives*. La récursion ne nécessite pas d'instruction spécifique, mais le code d'une fonction récursive est *bouclé* (section 7), si bien que la *terminaison* de l'évaluation (*Théorème de Convergence* de la section 3) n'est plus assurée[17].

[17] 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*).

6 Effets de Bord et Echappements

6.1 Actions primitives

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



Par exemple [**Wash**; **Rinse**] et [**Wash**; **Rinse**; **Wash**; **Rinse**; **Spin**] sont des *actions* de type $\mathbf{I} \to \mathbf{III}$, 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 3) tout combinateur catégorique $\varphi : \mathbf{I} \to \mathbf{III}$ est congru à une *action* (unique) $\downarrow\varphi : \mathbf{I} \to \mathbf{III}$.

On va donc étendre le langage:

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 $\varphi : (\cdots((\mathbf{I} \wedge A_1) \wedge A_2)\ldots) \wedge A_n \to B$:

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

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:

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

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

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

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[18] (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).

6.2 Actions calculées

Suivant la preuve du *Théorème de Cohérence Hiérarchique* (appendice C), on pose $\mathbf{Val(A)} = \mathbf{Hom}_{\Sigma^\bullet}(\mathbf{I}, \mathbf{A})$ pour A = **I**, **II**, **III**. Une valeur d'un type de base est donc simplement une liste d'actions primitives:

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

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

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:

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

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 (par exemple des *entrées/sorties*). 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* avec un langage fonctionnel.

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

[18] Le combinateur catégorique correspondant à ce programme est **Fst** ∘ (**Id**, **Rinse** ∘ **Wash**) qui est congru à **Id**.

- *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.

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**:

code	environnement	pile	code	environnement	pile
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:

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

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]:

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

fail provoque un échec qui peut être ratrappé par ? . Plus précisément, pour évaluer $\mu \,?\, \nu$, on évalue μ, et si on échoue au cours de cette évaluation, on évalue ν à la place.

Il est nécessaire d'ajouter une *pile d'échappement*[19] pour sauvegarder les états de la machine:

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

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

[19] 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].

7 Quelques Optimisations

7.1 Amélioration de la Machine Catégorique

Dans le code compilé, certaines instructions (**App**, **Branch**$(-,-)$, et les primitives binaires **Plus**, **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:

code	environnement	pile	code	environnement	pile
App :: C	u	$(C' \cdot v) :: S$	C'	(v,u)	$C :: S$
Branch(C',C'') :: C	*true*	$u :: S$	C'	u	$C :: S$
Branch(C',C'') :: C	*false*	$u :: S$	C''	u	$C :: S$
Plus :: C	u	$v :: S$	C	$u+v$	S
Times :: C	u	$v :: S$	C	$u \times v$	S

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

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 clôture $(C \cdot u)$, 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*[20]. 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[21]:

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

- $C = [\mathbf{Push}; \mathbf{Snd}; \mathbf{Fst}; \mathbf{Branch}_2(C', C''); \mathbf{Return}]$

[20] L'environnement qui apparaît *à l'exécution*. Les annotations n'apparaissent qu'*à la compilation*.
[21] On utilise ici les *nouvelles* instructions primitives (7.1).

- $C' = [\![0]\!]_{((\_,x)\{f \to C\},(\_,\_))} @ [\mathbf{Return}] = [\mathbf{Quote}(0); \mathbf{Return}]$
- $C'' = [\![f\ y + 1]\!]_{((\_,x)\{f \to C\},(\_,y))} @ [\mathbf{Return}] =$
  $[\mathbf{Push}; \mathbf{Fst}; \mathbf{Cur}(C); \mathbf{Swap}; \mathbf{Snd}; \mathbf{Snd}; \mathbf{App}; \mathbf{Push}; \mathbf{Quote}(1); \mathbf{Plus}; \mathbf{Return}]$[22]

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

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* $x_{n-1} : A_{n-1}, \ldots, x_0 : A_0 \vdash \mu : B$, on associe $[\![\mu]\!]_{[x_0;\cdots;x_{n-1}]}$. Si on exécute ce code avec une pile $u_0 :: \cdots :: u_n :: S$, la valeur obtenue est $\mu[u_0/x_0, \ldots, u_{n-1}/x_{n-1}]$, et *la pile est inchangée*[23].

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

code	valeur	pile	code	valeur	pile
$\mathbf{Acces}_n :: C$	u	$u_0 :: \cdots :: u_n :: S$	C	u_n	$u_0 :: \cdots :: u_n :: S$

- $[\![x_n]\!]_{x_0::\cdots::x_n::X} = \mathbf{Acces}_n\ (x_n \neq x_0, \ldots, x_{n-1})$
- $[\![\mathbf{let}\ x = \mu\ \mathbf{in}\ \nu]\!]_X = [\![\mu]\!]_X @ [\mathbf{Push}] @ [\![\nu]\!]_{x::X} @ [\mathbf{Pop}]$
- $[\![\mathbf{fst}\ \mu]\!]_X = [\![\mu]\!]_X @ [\mathbf{Fst}]$
- $[\![\mathbf{snd}\ \mu]\!]_X = [\![\mu]\!]_X @ [\mathbf{Snd}]$
- $[\![(\mu, \nu)]\!]_X = [\![\mu]\!]_X @ [\mathbf{Push}] @ [\![\nu]\!]_{\_::X} @ [\mathbf{Cons}]$
- $[\![c]\!]_X = [\mathbf{Quote}(u)]$ (c est une constante ou une variable globale de valeur u)
- $[\![\mathbf{if}\ \lambda\ \mathbf{then}\ \mu\ \mathbf{else}\ \nu]\!]_X = [\![\lambda]\!]_X @ [\mathbf{Branch}([\![\mu]\!]_X @ [\mathbf{Return}], [\![\nu]\!]_X @ [\mathbf{Return}])]$
- $[\![-\mu]\!]_X = [\![\mu]\!]_X @ [\mathbf{Negate}]$
- $[\![\mu + \nu]\!]_X = [\![\mu]\!]_X @ [\mathbf{Push}] @ [\![\nu]\!]_{\_::X} @ [\mathbf{Plus}]$
- $\ldots$

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

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

[22] On a utilisé ici une autre optimisation:

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

[23] 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.*

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

$$C = [\![\, \mu \,]\!]_{[\_;x_1;\cdots;x_n]} @ [\mathbf{Return}]^{24}$$

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

Naturellement, ce type de compilation ne s'étend pas aux *termes fonctionnels*[25]. Par exemple, pour évaluer **let** $x = 3$ **in fun** $y \to x + y$, 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.

7.4 Autres optimisations

Il y a beaucoup d'autres optimisations dans [Suarez], notamment la reconnaissance les *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 5):

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

Comme l'ensemble des adresses de paires est *disjoint* de l'ensemble des atomes (remarque 14, section 4), on peut représenter **zero** par un atome plutôt que par une paire. On utilise alors une instruction qui reconnaît les atomes au lieu de $\mathbf{Branch}(-,-)$.

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

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

Autrement dit, on peut éliminer le *constructeur superflu* **atom**.

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 **let** $(x, y) = \mu$ **in** ν.

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:

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

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

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

De ce fait, $\mathbf{fst}\ \mu$ est équivalent à $\mathbf{let}\ (x, \_) = \mu\ \mathbf{in}\ x$ (et $\mathbf{snd}\ \mu$ à $\mathbf{let}\ (\_, x) = \mu\ \mathbf{in}\ x$). On n'a donc pas besoin de $\mathbf{fst}\ -$ et $\mathbf{snd}\ -$ 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 5).

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*[26].

Nous proposons une méthode générale pour implanter des langages de toute sorte:

- 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[27].

[26]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. C'est pourquoi nous étudions maintenant une *logique plus fine* introduite par J.Y. Girard [Girard86,GirLaf].

[27]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.

Appendices

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 4 *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.

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 $\vee$:

$$\dfrac{\dfrac{\vdots}{\bot}}{\dfrac{A \wedge B}{A}} \qquad \dfrac{\dfrac{\vdots}{\bot}}{\dfrac{A \wedge B}{B}} \qquad \dfrac{\dfrac{\vdots}{\bot}}{\dfrac{\bot}{A}} \qquad \dfrac{\dfrac{\vdots}{\bot} \quad \begin{matrix}[A] \\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B] \\ \vdots \\ C\end{matrix}}{\dfrac{A \vee B \quad C \quad C}{C}} \qquad \dfrac{\dfrac{\vdots}{\bot} \quad \vdots}{\dfrac{A \Rightarrow B \quad A}{B}}$$

$$\dfrac{\begin{matrix}\vdots \\ A \vee B\end{matrix} \quad \begin{matrix}[A] \\ \vdots \\ C \wedge D\end{matrix} \quad \begin{matrix}[B] \\ \vdots \\ C \wedge D\end{matrix}}{\dfrac{C \wedge D}{C}} \qquad \dfrac{\begin{matrix}\vdots \\ A \vee B\end{matrix} \quad \begin{matrix}[A] \\ \vdots \\ C \wedge D\end{matrix} \quad \begin{matrix}[B] \\ \vdots \\ C \wedge D\end{matrix}}{\dfrac{C \wedge D}{D}} \qquad \dfrac{\begin{matrix}\vdots \\ A \vee B\end{matrix} \quad \begin{matrix}[A] \\ \vdots \\ \bot\end{matrix} \quad \begin{matrix}[B] \\ \vdots \\ \bot\end{matrix}}{\dfrac{\bot}{C}}$$

$$\dfrac{\begin{matrix}\vdots \\ A \vee B\end{matrix} \quad \begin{matrix}[A] \\ \vdots \\ C \vee D\end{matrix} \quad \begin{matrix}[B] \\ \vdots \\ C \vee D\end{matrix} \quad \begin{matrix}[C] \\ \vdots \\ E\end{matrix} \quad \begin{matrix}[D] \\ \vdots \\ E\end{matrix}}{E} \qquad \dfrac{\begin{matrix}\vdots \\ A \vee B\end{matrix} \quad \begin{matrix}[A] \\ \vdots \\ C \Rightarrow D\end{matrix} \quad \begin{matrix}[B] \\ \vdots \\ C \Rightarrow D\end{matrix} \quad \vdots}{\dfrac{C \Rightarrow D \qquad C}{D}}$$

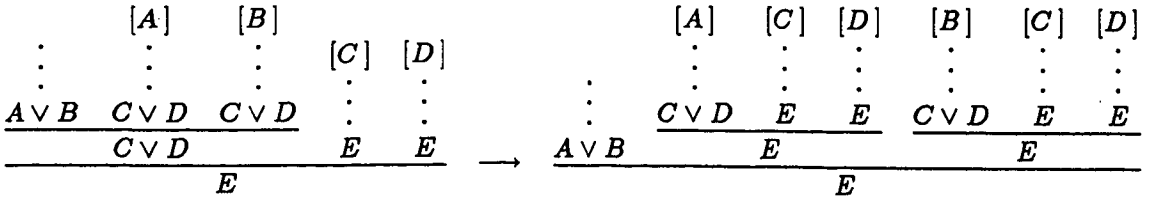
Les coupures du $\bot$ s'éliminent de façon évidente:

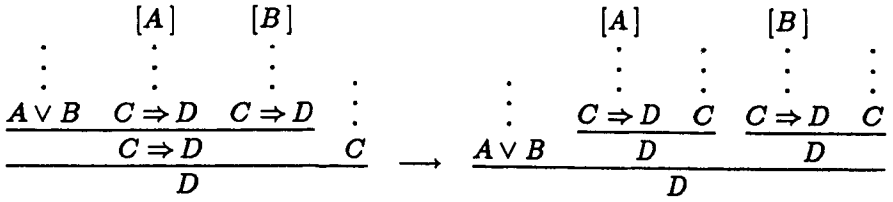
$$\dfrac{\dfrac{\vdots}{\bot}}{\dfrac{A \wedge B}{A}} \longrightarrow \dfrac{\vdots}{\dfrac{\bot}{A}}$$

Les coupures du $\vee$ s'éliminent par "commutation" des règles:

$$\dfrac{\begin{matrix}\vdots \\ A \vee B\end{matrix} \quad \begin{matrix}[A] \\ \vdots \\ C \wedge D\end{matrix} \quad \begin{matrix}[B] \\ \vdots \\ C \wedge D\end{matrix}}{\dfrac{C \wedge D}{C}} \longrightarrow \dfrac{\begin{matrix}\vdots \\ A \vee B\end{matrix} \quad \dfrac{\begin{matrix}[A] \\ \vdots \\ C \wedge D\end{matrix}}{C} \quad \dfrac{\begin{matrix}[B] \\ \vdots \\ C \wedge D\end{matrix}}{C}}{C}$$

$$\dfrac{\begin{matrix}\vdots \\ A \vee B\end{matrix} \quad \begin{matrix}[A] \\ \vdots \\ \bot\end{matrix} \quad \begin{matrix}[B] \\ \vdots \\ \bot\end{matrix}}{\dfrac{\bot}{C}} \longrightarrow \dfrac{\begin{matrix}\vdots \\ A \vee B\end{matrix} \quad \dfrac{\begin{matrix}[A] \\ \vdots \\ \bot\end{matrix}}{C} \quad \dfrac{\begin{matrix}[B] \\ \vdots \\ \bot\end{matrix}}{C}}{C}$$


[A] [B] [C] [D]
A∨B C∨D C∨D
C∨D E E
E
→
[A] [C] [D] [B] [C] [D]
C∨D E E C∨D E E
A∨B E E
E


[A] [B]
A∨B C⇒D C⇒D
C⇒D C
D
→
[A] [B]
C⇒D C C⇒D C
A∨B D D
D

C Démonstration du Théorème de Cohérence Hiérarchique

On démontre ici le *Théorème de Cohérence Hiérarchique*:

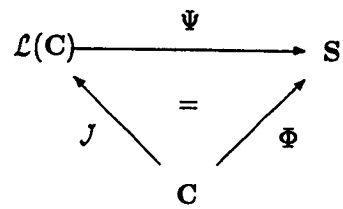
Pour toute catégorie **C**, *le foncteur canonique* $J : \mathbf{C} \to \mathcal{L}(\mathbf{C})$ *est plein et fidèle. Autrement dit, si* A *et* B *sont des objets de* **C**, J *induit une bijection:*

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

C.1 J est fidèle

Pour montrer que J est fidèle, il suffit de trouver *une catégorie cartésienne fermée* **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[28].

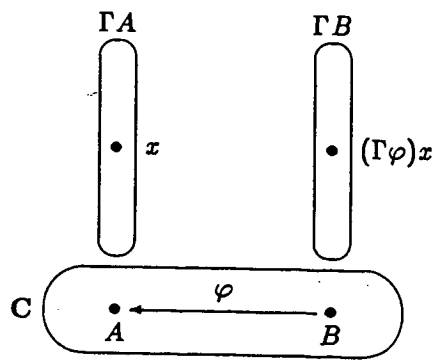


On considère la catégorie $\hat{\mathbf{C}}$ des *foncteurs contravariants* $\mathbf{C} \to \mathbf{Set}$ (*préfaisceaux*), et le foncteur de *Yoneda* $\mathcal{Y} : \mathbf{C} \to \hat{\mathbf{C}}$ défini par:

$$(\mathcal{Y}A)B = \mathbf{Hom}_{\mathbf{C}}(B, A)$$

Lemme 3 *(Yoneda)*
Soit $\Gamma \in \hat{\mathbf{C}}$. *Pour tout objet* A *de* **C**, *on a un isomorphisme canonique* $\Gamma A \simeq \mathbf{Hom}_{\hat{\mathbf{C}}}(\mathcal{Y}A, \Gamma)$.

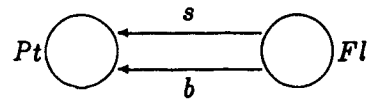
Démonstration:
A tout $x \in \Gamma A$, on associe la transformation naturelle τ définie par $\tau_B\, \varphi = (\Gamma\varphi)x$ pour tout $\varphi \in (\mathcal{Y}A)B$.



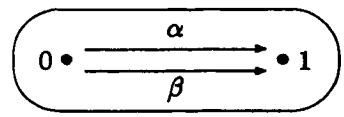
[28] Notez que si Φ est plein, on ne peut en déduire que J l'est aussi.

Réciproquement, à toute transformation naturelle $\tau : \mathcal{Y}A \to \Gamma$, on associe $\tau_A\, id_A \in \Gamma A$. □

Exemple: Un *graphe* est la donnée de deux ensembles Pt (*points*) et Fl (*flèches*) 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 flèche 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 flèche 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 4 $\hat{\mathbf{C}}$ *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:

$$(\Gamma \Rightarrow \Delta)A \simeq \mathbf{Hom}_{\hat{\mathbf{C}}}(\mathcal{Y}A, \Gamma \Rightarrow \Delta) \simeq \mathbf{Hom}_{\hat{\mathbf{C}}}(\mathcal{Y}A \wedge \Gamma, \Delta)$$

On pose alors $(\Gamma \Rightarrow \Delta)A = \mathbf{Hom}_{\hat{\mathbf{C}}}(\mathcal{Y}A \wedge \Gamma, \Delta)$, et on vérifie que c'est une exponentielle. □

Exemple: Ainsi, la catégorie des graphes est cartésienne fermée. La définition de l'exponentielle n'était nullement évidente à priori (ni les points ni les flèches 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 flèche $F \to G$ dans $\Gamma \Rightarrow \Delta$ est la donnée d'une flèche $F\,A \to G\,B$ dans Δ pour toute flèche $A \to B$ 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*):

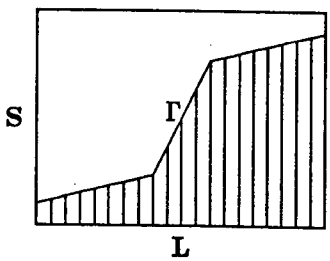
- Un objet de $\mathcal{G}l(\Gamma)$ est un couple d'objets A dans $\mathbf{L}$ et X dans $\mathbf{S}$, avec un morphisme $a : X \to \Gamma A$ dans $\mathbf{S}$.
- Un morphisme $A, X, a \to B, Y, b$ est un couple de morphismes $\varphi : A \to B$ dans $\mathbf{L}$ et $f : X \to Y$ dans $\mathbf{S}$ tels que $\Gamma\varphi \circ a = b \circ f$.

$$\begin{array}{ccc} \Gamma A & \xrightarrow{\Gamma\varphi} & \Gamma B \\ {\scriptstyle a}\uparrow & = & \uparrow{\scriptstyle b} \\ X & \xrightarrow[f]{} & Y \end{array}$$

On a évidemment deux foncteurs d'oubli $\mathcal{F}st : \mathcal{G}l(\Gamma) \to \mathbf{L}$ et $\mathcal{S}nd : \mathcal{G}l(\Gamma) \to \mathbf{S}$.

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:

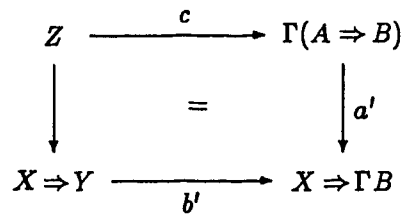


Si $\mathbf{L} = \mathbf{S} = \mathbf{Set}$ et Γ est le foncteur $A \mapsto A \times A$, $\mathcal{G}l(\Gamma)$ est la catégorie des graphes!

Lemme 5 *(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.



Nous ne détaillons pas les vérifications qui nécessitent quelques diagrammes indigestes. □

Exemple: 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.

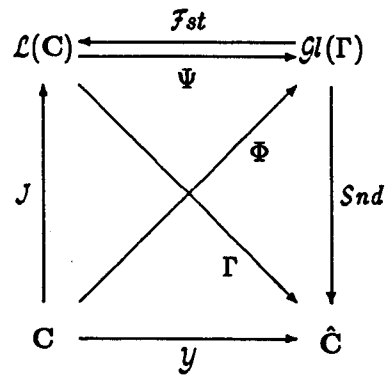
On va appliquer le lemme 5 au foncteur $\Gamma : \mathcal{L}(\mathbf{C}) \to \hat{\mathbf{C}}$ défini par $\Gamma A = \mathbf{Hom}_{\mathcal{L}(\mathbf{C})}(J\_, A)$. Il est clair que Γ préserve les produits cartésiens.

On considère le foncteur $\Phi : \mathbf{C} \to \mathcal{G}l(\Gamma)$ défini par:

- ΦA est le couple $JA, \mathcal{Y}A$ muni de la transformation naturelle $\iota : \mathcal{Y}A \to \Gamma(JA)$ telle que $\iota_B \varphi = J\varphi$ pour tout $\varphi \in (\mathcal{Y}A)B = \mathbf{Hom}_{\mathbf{C}}(B, A)$.

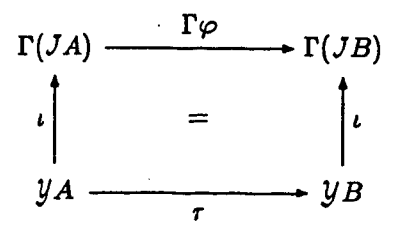
On a $\mathcal{F}st\,\Phi = J$.

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:



En particulier, pour $id_A \in (\mathcal{Y}A)A$, on obtient:

$$\varphi = \varphi \circ \mathcal{J}id_A = J(\tau_A id_A)$$

Ceci montre que J est plein.

Références

[CAML] "The CAML Primer" & "CAML, The Reference Manual." (le manuel d'utilisation de CAML) Projet Formel, INRIA-ENS (1987).

[CouCurMau] G. Cousineau, P.L. Curien & M. Mauny. "The Categorical Abstract Machine." Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 50–64 (1985).

[CouCurMauSu] G. Cousineau, P.L. Curien, M. Mauny & A. Suarez. "Combinateurs Catégoriques et Implémentation des Langages Fonctionnels."

[Church40] A. Church. "A formulation of the simple theory of types." Journal of Symbolic Logic **5,1** 56–68.

[Church41] A. Church. "The Calculi of Lambda-Conversion." Princeton U. Press, Princeton N.J.

[Curien83] P.L. Curien. "Combinateurs catégoriques, algorithmes séquentiels et programmation applicative." Thèse de Doctorat d'Etat, Université Paris VII.

[Curien85] P. L. Curien. "Categorical Combinatory Logic." ICALP 85, Nafplion, Springer-Verlag LNCS 194.

[Curien86] P. L. Curien. "Categorical Combinators, Sequential Algorithms and Functional Programming." Pitman.

[CurFeys] H. B. Curry & R. Feys. "Combinatory Logic Vol. I." North-Holland, Amsterdam (1958).

[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." Fenstad, 63-92.

[Girard86] J.Y. Girard. "Linear logic." A paraître dans T.C.S.

[GirLaf] J.Y. Girard & Y. Lafont. "Linear Logic and Lazy Computation." Rapport INRIA (1986) et CFLP 87, Pisa, Springer-Verlag.

[Huet] G. Huet. "Initiation à la Théorie des Catégories." INRIA (1985).

[Kleene] S.C. Kleene. "Introduction to Meta-mathematics." North Holland (1952).

[Lambek68] J. Lambek. "Deductive systems and categories." Math. Systems Theory.

[Lambek80] J. Lambek. "From Lambda-calculus to Cartesian Closed Categories." To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press.

[LamSco] J. Lambek & P.J. Scott. "Introduction to higher order categorical logic." Cambridge studies in advanced mathematics. Cambridge University Press (1986).

[Maclane] S. MacLane. "Categories for the working mathematician." Graduate Texts in Mathematics 5. Springer-Verlag (1971).

[Martinlöf] P. Martin-Löf. "An intuitionistic theory of types: predicative part." Rose and Shepherdson, 73-118 (1974).

[MauSu] M. Mauny & A. Suarez. "Implementing Functional Languages in the Categorical Abstract Machine." Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.

[Mauny] M. Mauny. "Compilation des langages fonctionnels dans les combinateurs catégoriques. Application au langage ML." Thèse de Troisième Cycle. Université 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** 33-48.

[Seely86] R.A.G. Seely "Categorical Semantics for Higher Order Polymorphic Lambda Calculus." non publié.

[Suarez] A. Suarez. "Une implémentation de ML en ML." Thèse à paraître (1987).

[Szabo] M.E. Szabo "Algebra of proofs" Studies in Logic and the Foundations of Mathematics vol 88. North-Holland Publishing Co., Amsterdam (1978).

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique