Le système F en logique linéaire

Christian RETORÉ

Mémoire de DEA (Diplôme d'Études Approfondies) de mathématiques
Sous la direction de Jean-Yves Girard
Equipe de logique, UFR de mathématiques, Université Paris 7
(Aujourd'hui Equipe de logique, Institut de Mathématiques de Jussieu-Paris Rive Gauche, Université Paris Cité et Université Paris Sorbonne)

Septembre 1987

Warning: *Some colleagues have asked me to deposit this thesis in an archive because certain results can only be found there, but the ChiWriter typesetting is terrible, some parts are handwritten, and the writing is far from ideal. Since I no longer have the source file, producing a better version would require a great deal of time, and I doubt the thesis is worth that effort. So here is a scan of the only version I still have.*

Résumé: Ce mémoire de 1987 étudie la traduction du système F de Girard en logique linéaire du premier ordre. Il est montré que la traduction standard, laquelle repose sur la traduction de l'implication intuitionniste A —> B en !A —o B où —o désigne l'implication linéaire et ! la modalité exponentielle peut être optimisée en omettant certain !, notamment dans les types inductifs dont l'argument d'un constructeur n'est jamais une fonction. Cette économie est cependant impossible pour les types que Girard appelle aussi inductifs dont au moins un constructeur a pour argument une fonction, comme c'est le cas pour les arbres à branchements de type omega.

En étudiant la normalisation des termes du système F en logique linéaire et notamment de l'unicité de la forme normale d'un entier, nous avons été amenés à considérer un cedex qui ne correspond pas à une coupure, mais à la suppression un affaiblissement droit de la formule K suivi d'une contraction ce cette occurrence de K avec une autre occurrence de K déjà présente, réduction qui à la suite de ce mémoire a été nommée Ret dans les travaux de Vincent Danos puis d'autres travaux.

Abstract: This 1987 master thesis examines the translation of Girard's System F into first-order linear logic. It is shown that the standard translation, based on translating the intuitionistic implication A -> B as !A -o B (where -o denotes linear implication and ! the exponential modality), can be optimized by omitting certain occurrences of !, in particular in inductive types whose constructor arguments are never functions. However, such an optimization is impossible for types that Girard also calls inductive where a constructor takes as argument a function, like omega branching trees.

When investigating normalization of System F terms in linear logic, especially the uniqueness of the normal form of a natural number, we are led to consider a reduction step that does not correspond to a cut-elimination step. Instead, it corresponds to eliminating a right weakening on formula K followed by contracting this occurrence of K with another existing occurrence of K. This reduction, following the present dissertation, was named Ret in the work of Vincent Danos and later in subsequent research.

Bibliographie:

1. Jean-Yves Girard Lambda calcul typé, Notes de cours, 1986-1987, Université Paris 7
2. Jean-Yves Girard, Yves Lafont, Paul Taylor, Proofs and Types, Cambridge Tracts in Theoretical Computer Science, 1988, Cambridge University Press (adaptation et traduction de 1)
3. Jean-Yves Girard, Linear Logic, Theoretical Computer Science, 50(1):1–102, 1987
4. Vincent Danos, La logique linéaire appliquée à l'étude de divers processus de normalisation et principalement du lambda-calcul, Doctorat de Mathématiques, Université Paris 7, juin 1990.

LE SYSTEME F EN LOGIQUE LINEAIRE

Notations : étant peu instruit des techniques modernes, j'ai du modifier les symboles usuels :

lambda du 1er ordre : ⌐‖

lambda du second ordre : /\

double flèche : => (le seul qui ait l'air naturel !)

par : ⅋

et, & : ⊓

tenseur : ⊗

plus : +

implication linéaire : ⟜

variables de type : MAJUSCULES et Θ

variables d'individus, termes de F: minuscules

le réseau obtenu en traduisant t : t° autre traduction t°°

parenthèsage implicite à droite pour les " => " ou " ⟜ "

Pour ne pas alourdir le texte j'ai souvent écrit :il (n') existe (pas) un réseau de conclusion A ⟜ B, alors que pour être précis il m'eût fallu ajouter que lorsqu'on le "branche" avec un réseau de conclusion A, on obtient, après élimination des coupures, le réseau de type B qui lui est canoniquement associé (le plus souvent A et B sont deux traduction d'un type de F:
le réseau est alors la fonction qui transforme une traduction d'un F-terme en l'autre.)

LE SYSTEME F EN LOGIQUE LINEAIRE

Notations : étant peu instruit des techniques modernes, j'ai du modifier les symboles usuels :

lambda du 1er ordre : ⌐||

lambda du second ordre : /\

double flêche : => (le seul qui ait l'air naturel !)

par : ⅋

et, & : ⊓

tenseur : ⊗

plus : +

implication linéaire : ⟜

variables de type : MAJUSCULES et θ

variables d'individus, termes de F: minuscules

le réseau obtenu en traduisant t : t° autre traduction t°°

parenthèsage implicite à droite pour les " => " ou " ⟜ "

Pour ne pas alourdir le texte j'ai souvent écrit :il (n') existe (pas) un réseau de conclusion A ⟜ B, alors que pour être précis il m'eût fallu ajouter que lorsqu'on le "branche" avec un réseau de conclusion A, on obtient, après élimination des coupures, le réseau de type B qui lui est canoniquement associé (le plus souvent A et B sont deux traduction d'un type de F:
le réseau est alors la fonction qui transforme une traduction d'un F-terme en l'autre.)

I | Une traduction complète mais couteuse

(que j'ai programmée en méta-vlisp (Emmanuel Saint-James ,LITP ,mars 87).
Voir à la fin le programme ainsi que des exemples de traductions.)

Ia | L'algorithme

Soit t un terme de F :

t = /\ ... ⅋ ...(y αK....α1)

de type T dans le contexte (x1 S1)...(xN SN).

avec _ y : variable de tête (libre ou non)

ou

⅋ ou /\ expression (T possède alors au moins ce redex)

_ αK : terme que l'on appelle dans ce cas aK

ou

type de F que l'on appelle alors ΘK

On peut traduire t par un reseau de conclusions

T°

?(S1°)⊥.....?(SN°)⊥ correspondants une à une aux variables libres de t (pour être précis il faudrait indéxer chaque sortie par la variable en question)

H°⊥ correspondant à la variable libre de tête y s'il y en a une et si elle n'apparait qu'une fois dans t (en abrégé vltu v(ariable) l(ibre) (d'occurrence) u(nique))

où (X)° désigne la traduction du type X de F en logique linéaire définie ainsi :

X est une variable de type

X°= X

X = /\ A (U)

X°=/\ A (U°)

X = (A => B)

X°= !(A°) ⟜ B° = ?(A°)⊥ ⅋ B°

Cette traduction satisfait (T [U/A])°=T°[U°/A].

(par la suite je ne distinguerai plus X° de X pour alléger l'écriture : s'il s'agit d'une conclusion de réseau X devra être lu X°)

On procède évidement par induction sur la structure de t :

A | t est une variable

t° : ⊢⊥ ⊣ (t est alors l'occurence unique d'une vltu)

B | t est une /\ expression

t = /\ A u

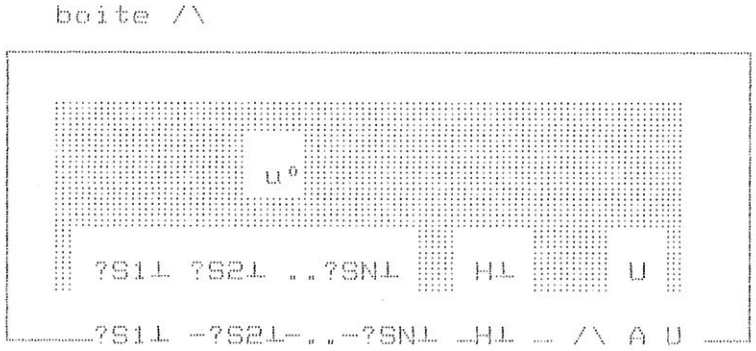
Soit u° la traduction de u , de conclusions

U (type de u)

?S1⊥...?SN⊥ (u et t ont même contexte)

H⊥ (le cas échéant ie u a une vltu)

la variable de type A n'étant pas libre dans S1...SN H on peut construire t° à partir de u° en formant une boite /\ de sortie principale U



C | t est une ⊓‖ expression

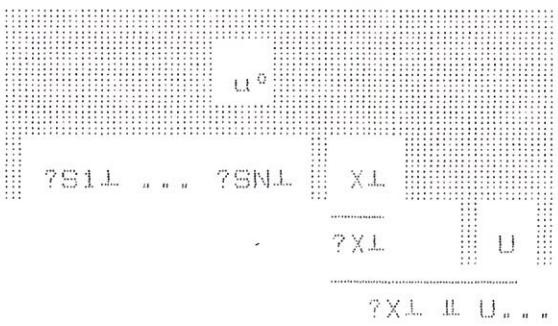
t = ⊓‖ x u x:X u:U t:(X => U)

1_ x est une variable libre de u

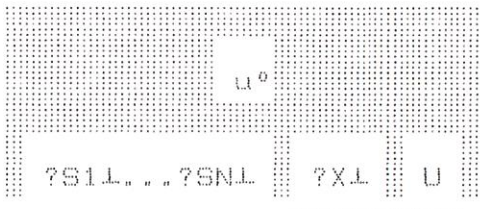
1.1_ x est la variable libre de tête de u et celle ci n'apparait qu' une seule fois (x vltu)

On dispose de u°de conclusions ?S1⊥..?SN⊥ X⊥ U

on forme t° ainsi :



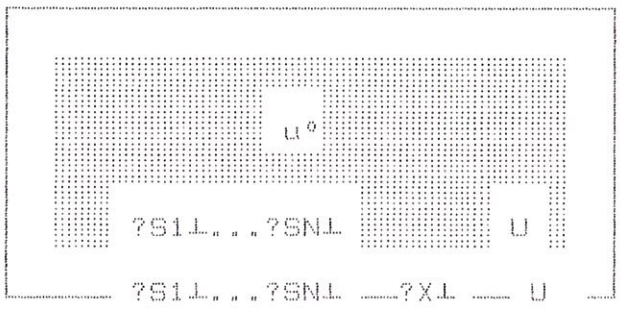
1.2_ x est n'importe quelle autre variable libre de u
On dispose de u° de conclusions ?S1⊥..?SN⊥ ?X⊥ U
on forme t° ainsi :



?X⊥ ⅋ U ..... (ie X => U)

2_ x n'apparait pas dans u
On dispose de u° de conclusions ?S1⊥..?SN⊥ U
il faut affaiblir u° pourfaire apparaitre ?X⊥
puis tout se passe comme en 1.2_ :

boite d'affaiblissement



?X⊥ ⅋ U.....(ie X => U)

D | t est suite d'applications du premier et du second ordre

t = y αK αK-1 ....α1
avec y:Y
αI= aI de type AI s'il s'agit d'un terme
θI s'il s'agit d'un type
Je distinguerai les cas suivants :
1_ y est une variable libre de tête qui peut
1.1_ n'avoir que cette occurence dans t
1.2_ apparaitre dans l'un des aP
2_ y est une abstraction (du 1er ou 2nd ordre)

1_ y est une variable

1.1_qui n'a que cette occurrence dans t (y est une v1tu)

On procède par récurrence sur K de la manière suivante :
p(K) : Quels que soient les termes où types α1...αK et la variable y n'apparaissant pas dans les termes a1...aK on peut construire un réseau qui traduise y αK.....α1 (si toute fois ce terme est bien formé) c'est à dire un réseau rK° de conclusions :

?S1⊥...?SN⊥ où S1...SN est la liste (sans répétition) des types des variables libres des termes a1..aK
Y⊥ Y :type de y
(en effet y est une variable de tête,libre,et nous supposons pour l'instant qu'elle n'apparait qu'ainsi dans t)
T type de t

p(0) t=y deja vu : t° = Y⊥ Y

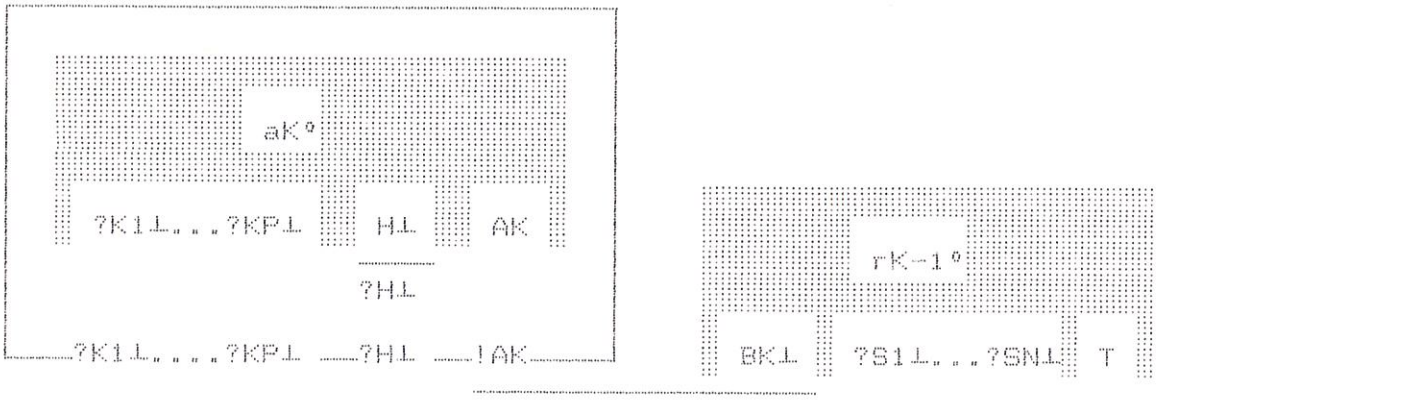
p(K-1)=>p(K)

Soit à traduire le terme y αK αK-1...α1
nous disposons,par hypothèse de récurrence de la traduction
rK-1° (dont la conclusion correspondant à z est BK⊥,sans "?")
du terme z αK-1 ....α1 où z est une variable libre du type de
y αK.Deux cas se présentent :

1.1.a αK est un terme (que l'on nomme aK)

(y est donc de type Y = AK => BK)
Nous disposons alors, par hypothèse d'induction,de aK°
traduction de aK de conclusions
- AK
- H⊥ (si aK possède une vltu)
- ?K1⊥...?KP⊥ (correspondants aux vlib de AK)

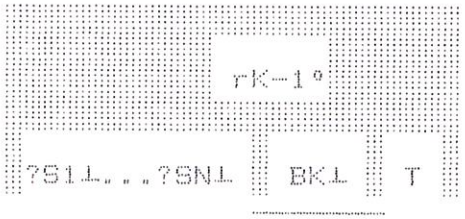
et de rK-1° dont une sortie est BK⊥.
nous pouvons alors construire rK° ainsi :
boite !



Il reste ensuite à contracter les sorties de même types ?(.)⊥
lorsqu'elles sont associées à la même variable.
On a alors p(K).

1.1.b αK est un type (que l'on nomme θK)

(y est donc de type /\ A BK+ avec BK+ [θK \ A] =BK)
(puisque y θK est de type BK)
Dans ce cas on construit rK° à partir de rK-1° par une \/ règle



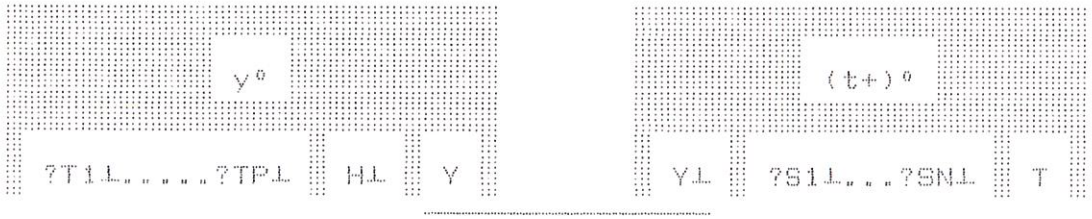
On a alors p(K).

1.2 y a d'autres occurences (dans l'un des αI)

Procéder comme ci-dessus (1.1) en faisant comme si le y de tête était différent des autres occurences de y.Puis une fois le réseau construit "dérélicter" la sortie (correspondant à l'occurence) de tête de y,et alors contracter avec les sorties correspondant aux autres occurences de y.

2 y est une ⅋ ou /\ expression

Soit y+ une variable du type de y (ie Y) n'apparaissant dans aucun des aK ,ni dans y.
D'après 1.1 on peut traduire y+ αK....α1 en un réseau de conclusions :
Y⊥ (sans "?" puisque y est une v1tu)
T type de t = type de y+ αK...α1
?S1⊥.....?SN⊥ correspondant biunivoquement aux variables libres des aK
Par hypothèse d'induction on dispose de y°,traduction de y de conclusions :
Y (type de y)
H⊥ (si y a une v1tu)
?T1⊥...?TP⊥ (autres variables libres de y)
On traduit y αK....α1 à partir de y° et (t+)°=(y+ αK....α1)°
par une coupure associée au redex y αK :
(puis en contractant les sorties correspondant aux mêmes variables libres apparaissant dans y et dans l'un au moins des aK)



CUT

Ib | L'équivalence redex-coupure

Vu la manière dont j'ai écrit les termes de F et la traduction on voit de suite qu'un redex de t equivaut à une utilisation de la coupure dans t°.(Et de ce fait t normal équivaut à t° sans coupure .)
Toutefois on peut aisément démontrer cette équivalence :

t est une variable t n'a pas de redex ,t° n'a pas de coupure

t est une ⅋ expression t = ⅋ x u ou une /\ expression t = /\ A u
t a les mêmes redex que u
t° est obtenu à partir de u° par B ; ses coupures sont donc celles de u° càd par hypothèse d'induction les traductions des redex de u , càd ceux de t.

t est une suite d'applications 1 et 2 , t = y αK...α1
y est une variable
les redex de t sont ceux des aK...a1
t° est obtenu par itération de B 1.1 ou 1.2 qui n'introduisent pas de coupures mais contient aK...a1 .Les aK satisfaisant la propriété (par hypothèse d'induction) t la vérifie aussi.
y est une /\ ou ⅋ expression
les redex de t sont ceux de aK ..a1,ceux de y plus y aK
t° est obtenu à partir de

(t+)°qui comme on vient de le voir contient exclusivement les coupures associées aux redex de aK..a1

y° qui par hypothèse (ne) contient (que) les coupures correspondant aux redex de y

en faisant une coupure entre la sortie positive de y° et l'entrée de tête de (y+)°ce qui correspond au redex y αK

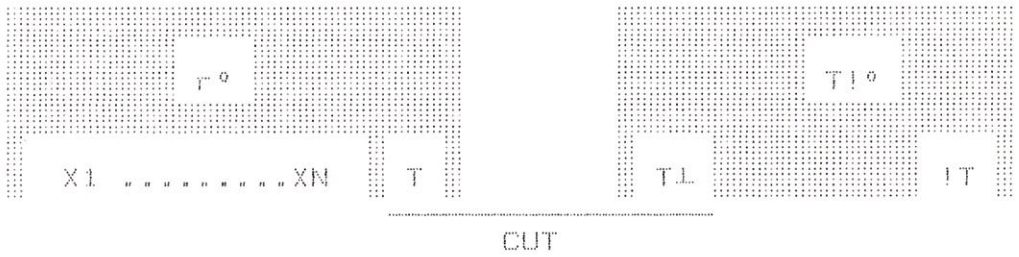
Ic | Le défaut majeur de cette traduction

La traduction de A => B par !A ⟜o B introduit quantité de boite ! imbriquées qui ne correspondent nullement à l'intuition que l'on a du " ! " : une fonction de type A => B n'utilisant qu'une seule fois son argument devrait se traduire par un réseau de conclusion A ⟜o B et, en particulier, un type inductif devrait pouvoir être représenté par :

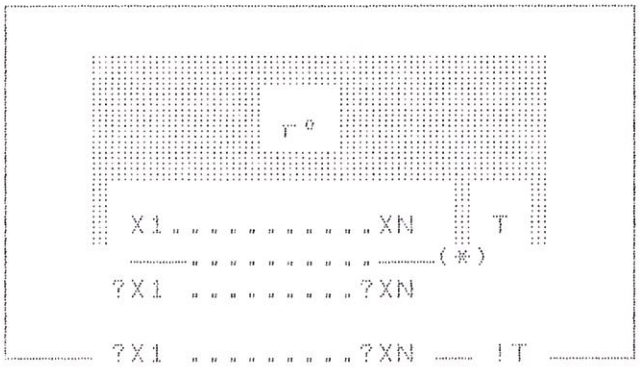
/\ A (...!(Ti1 ⟜o... (Tik ⟜o A))... ⟜o A)

càd avec un " ! " uniquement devant les constructeurs.
C'est pourquoi il convient de chercher d'autres traductions,en veillant cependant à ne pas perdre de pouvoir d'expression:
étant donnée une autre traduction de T ,disons T°°,les mêmes fonctions doivent être représentables de type T°° ⟜o U et T° ⟜o U
c'est notament le cas sil existe deux réseaux de conclusions T°° ⟜o T° et T° ⟜o T°° qui transforment t°° en t°et t° en t°°
(t F terme de type T).

Un type linéaire T sera dit exclamable s'il existe un réseau T!° de conclusions T⊥ et !T tel que le réseau :



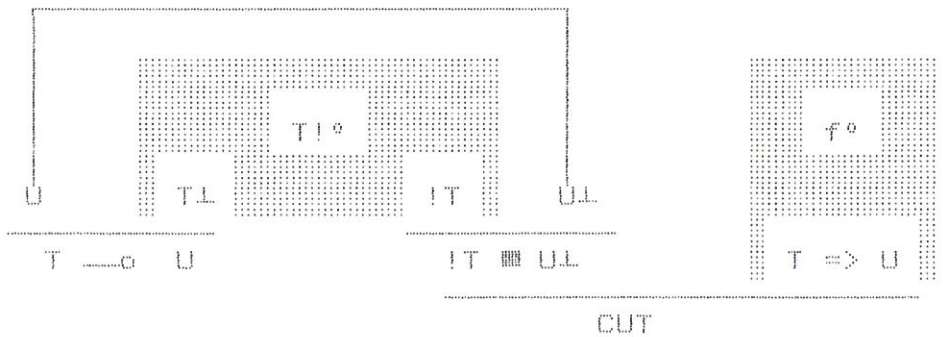
se réduise en un réseau équivalent (modulo l'ordre des contractions et éventuellement le "redex" affaiblissement/contraction voir à la fin) au réseau suivant :



(*) si XI n'est pas un " ? "

En clair, T!° met tout réseau dont une conclusion est T dans une boite ! Cette propriété de T me semble très importante car elle apparait à diverses reprises lorsqu'on cherche à optimiser cette traduction. De plus cette propriété est equivalente à :toute fonction définie sur T est linéarisable (ie étant donné un réseau f° de conclusion T => U il existe un réseau fl° de conclusion T ⊸ U qui appliqué à r° traduction d'un terme de type T retourne un réseau qui une fois réduit soit equivalent à une forme normale de f° appliqué à r°.)

fl° :

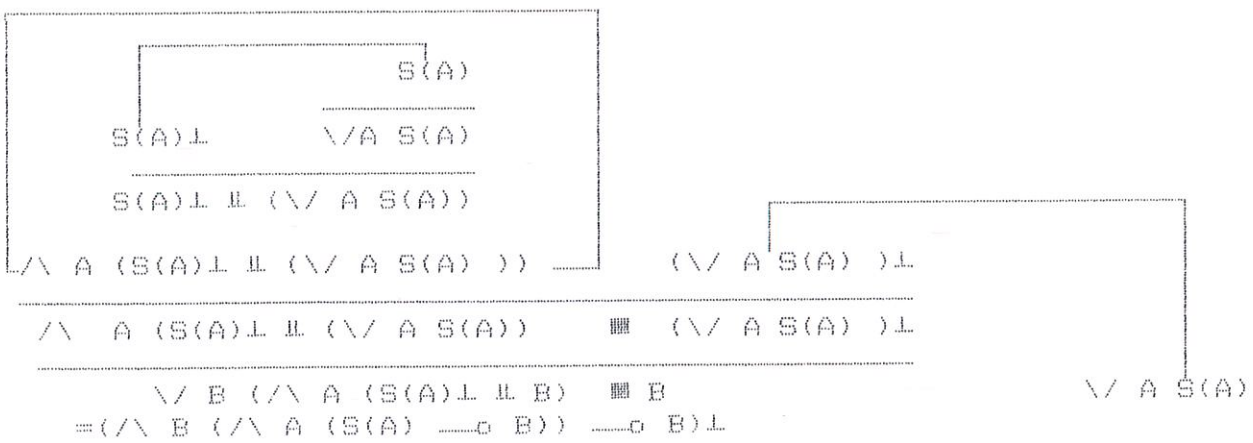


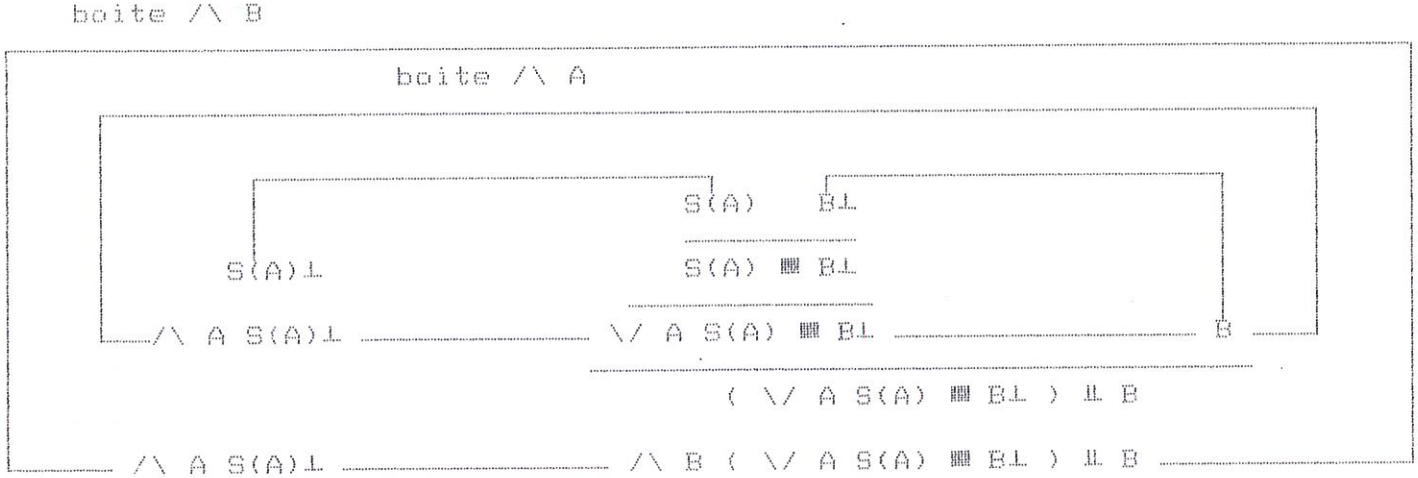
Réciproquement si tout réseau de conclusion T => U (et donc toute fonction de type T => U , si U est un type de F)a un équivalent de type T ⊸ U l'axiome ?T⊥ !T = T => !T a un équivalent de type T ⊸ !T, qui est le réseau cherché.

II | Quelques exemples de traductions

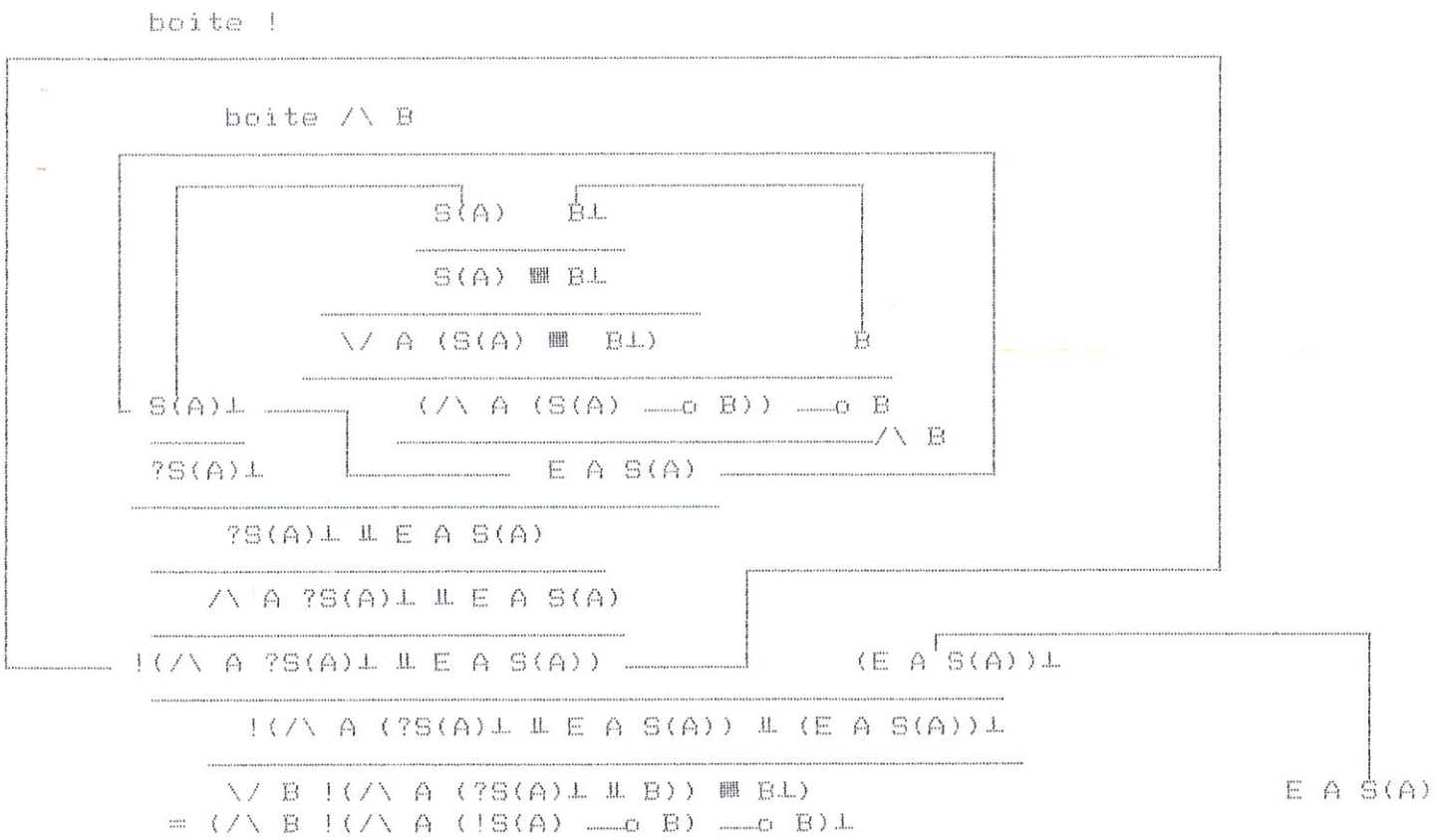
IIa | Le type existentiel

Tout d'abord on vérifie aisément que \/ A S et
/\ B ((/\ A (S(A) ⟜ B)) ⟜ B définissent le même type:

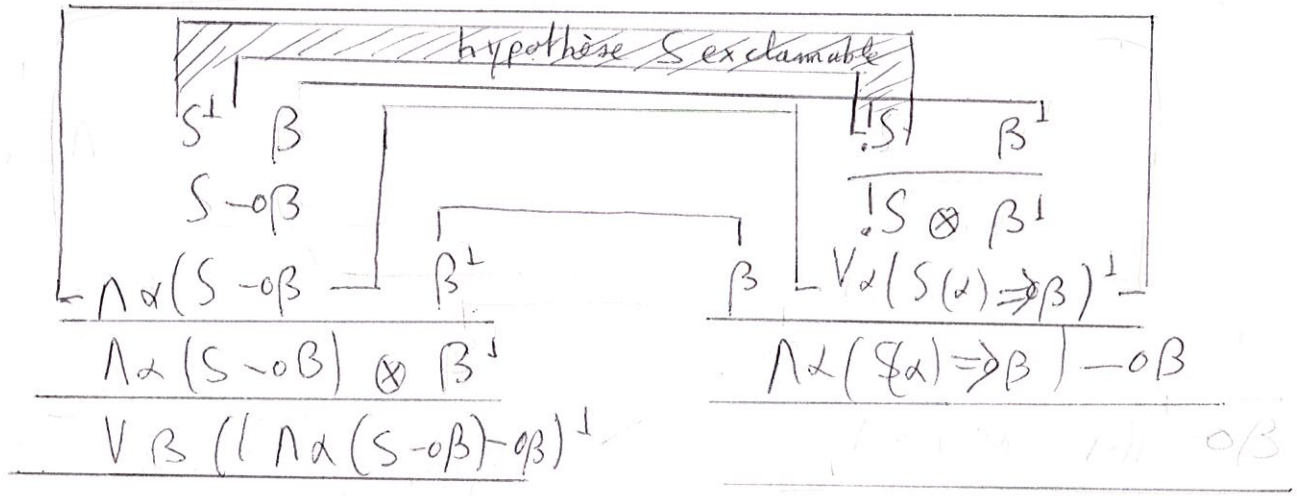




Par contre ,il n'est nullement évident que
\/f A S(A) = \/ B (! (/\ A (! S(A) ⟜ B)) ⟜ B)
définisse le *même* type que l'un des deux précédents .
On a bien sûr \/f A S(A) ⟜ E A S(A) = /\ B (/\ A S(A) ⟜ B) ⟜ B



Par contre E A S(A) ⊸ \/f A S(A) n'est réalisé qu'au cas où S est exclamable et construire le réseau qui,appliqué à \/ A S(A), avec S exclamable ne présente aucune difficulté,et l'on peut même ,par curiosité,regarder ce que deviennent les règles d'introduction et d'élimination du \/f :



l'élimination de $\forall f$ met bien en évidence le fait que l'on a pas, en général, $V\ \alpha\ S(\alpha) \multimap \forall f\ \alpha\ S(\alpha)$

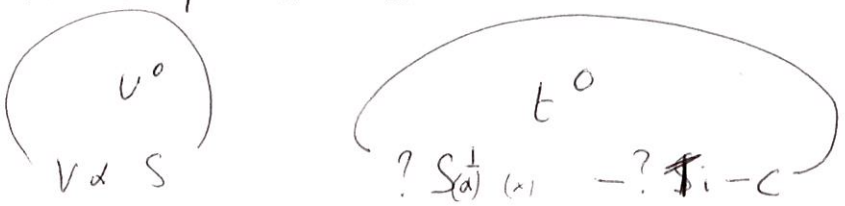
sait t de type C

α non libre dans les types de la base de T sauf S type de x

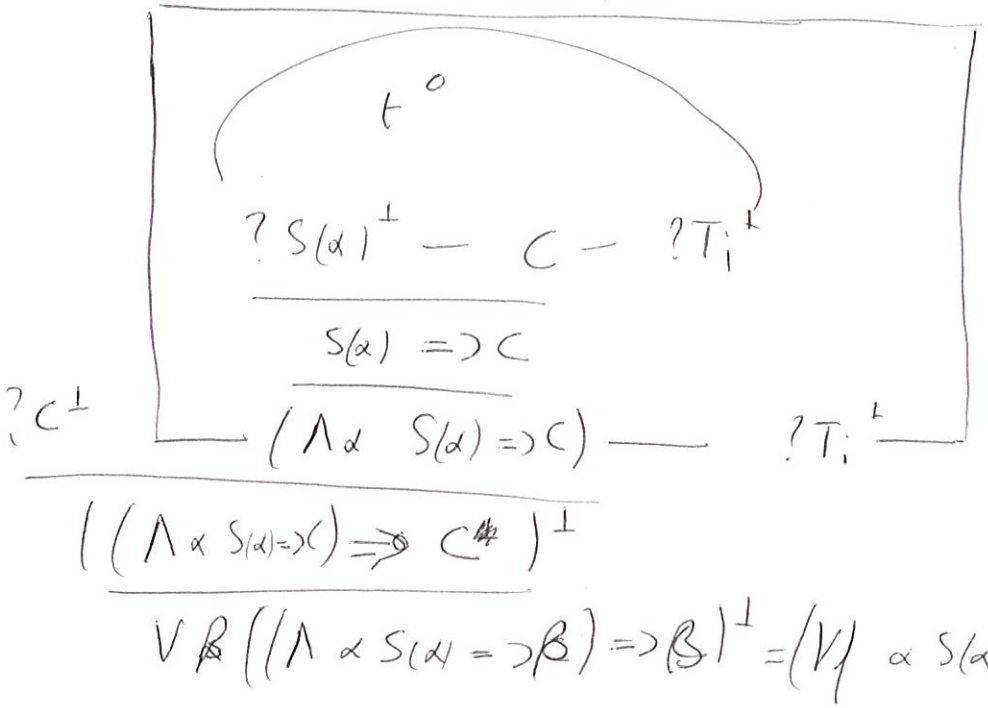
v de type $\forall f\ \alpha\ S$

on peut alors récupérer un objet de type C.

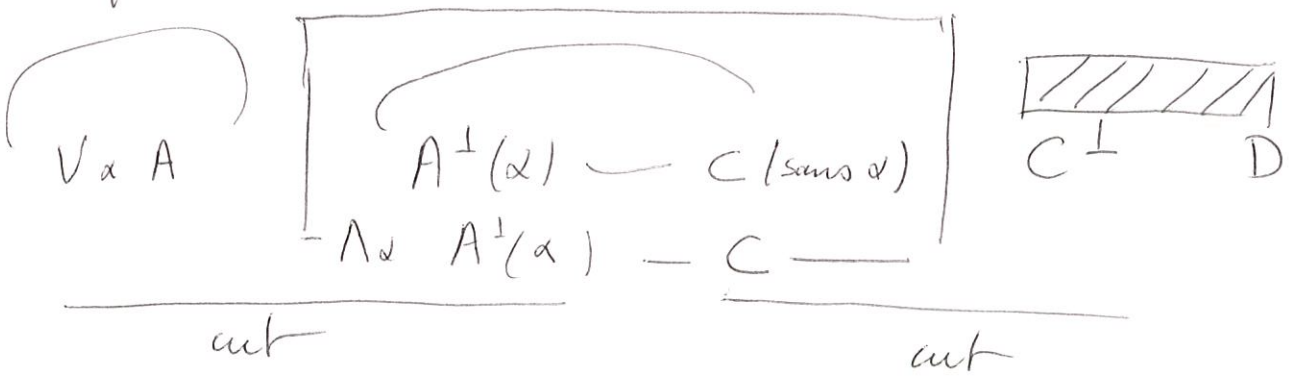
Si on prend comme traduction $V\alpha S$, on voit mal comment faire :



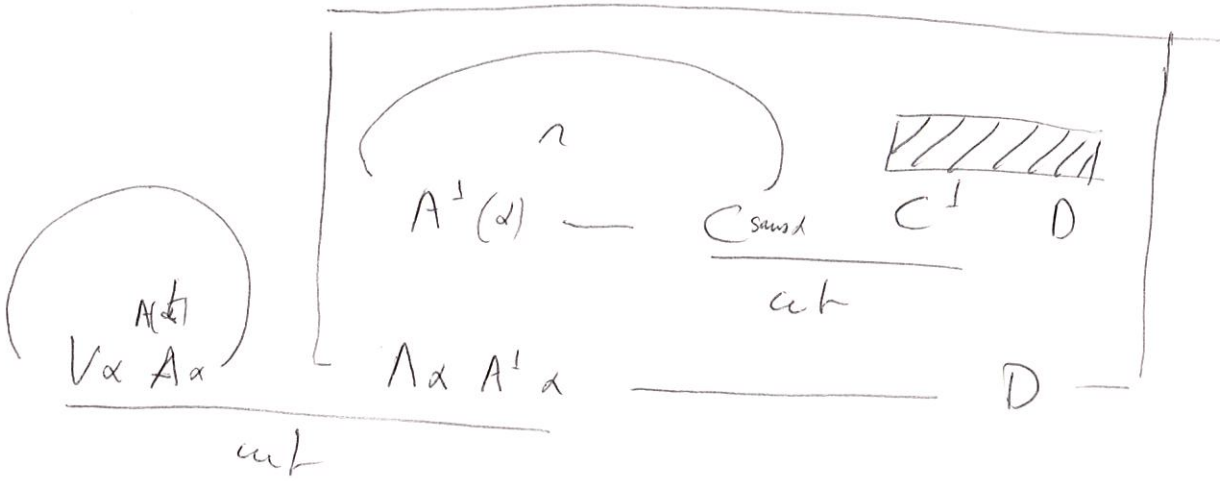
sinon entre v est de type $\forall f\ \alpha\ S(\alpha)$, c'est très simple :



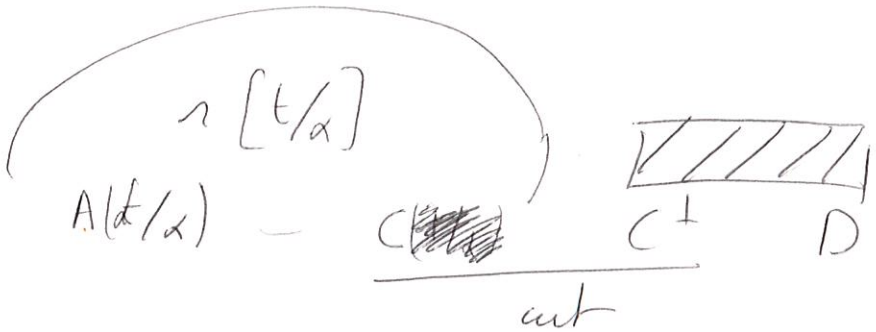
La contraction commutative de $\forall$ en notant $\underline{\quad\cdots\quad}$ $C^\perp$ D l'autre élimination avec laquelle celle du $\forall$ commute



se réduit en



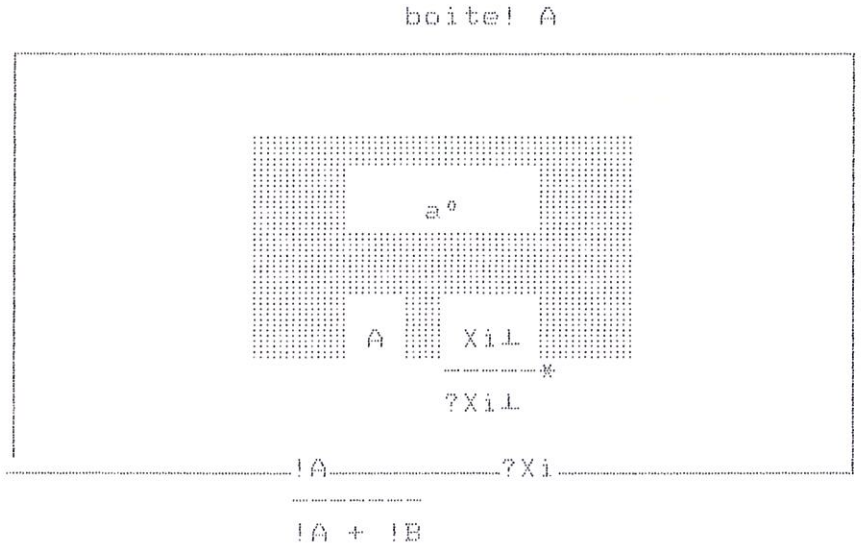
alors que la stratégie de $\vdash$ donne



IIb | La traduction de la somme

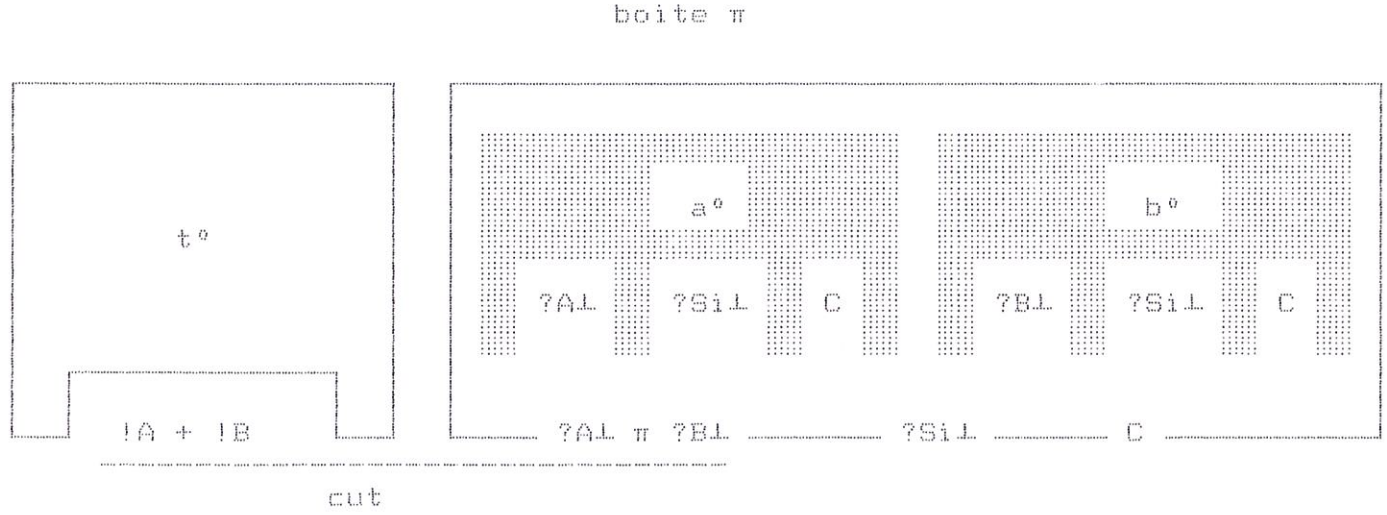
Pour représenter une somme,le plus naturel est-bien sûr-d'utiliser +. Cependant ,pour avoir le même schéma d'élimination que dans F,et surtout la même sémantique cohérente il est préférable de représenter /\ Θ (A => Θ) => (B => Θ) => Θ par !A + !B.Toutefois ,si A et B sont exclamables ,cette traduction est linéairement équivalente à A + B (comment trouver mieux ?).Et comme quelques réseaux valent mieux qu' un long discours :

L'introduction de la somme

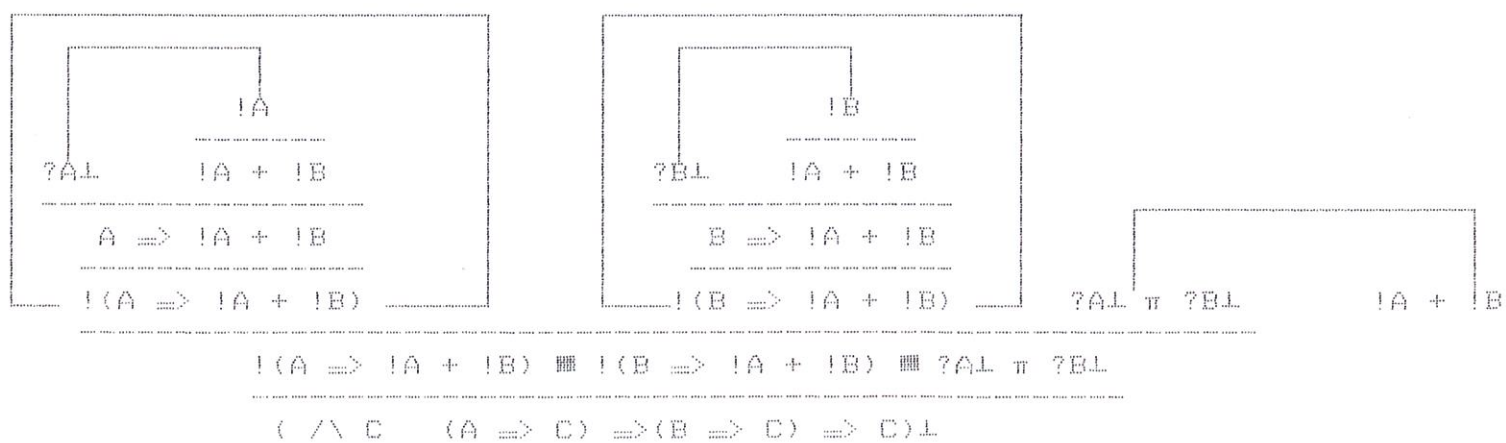


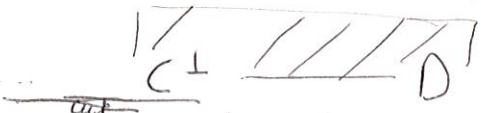
* si Xi n'est pas déja un "?"

L'élimination de la somme

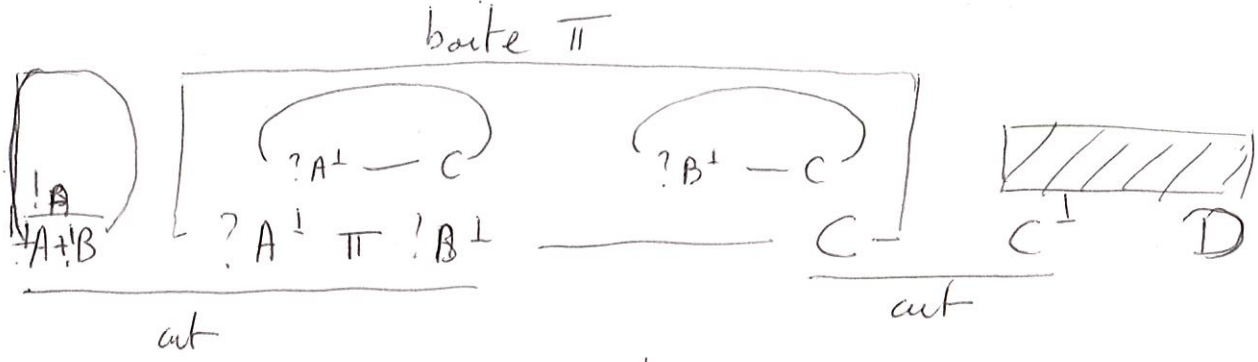


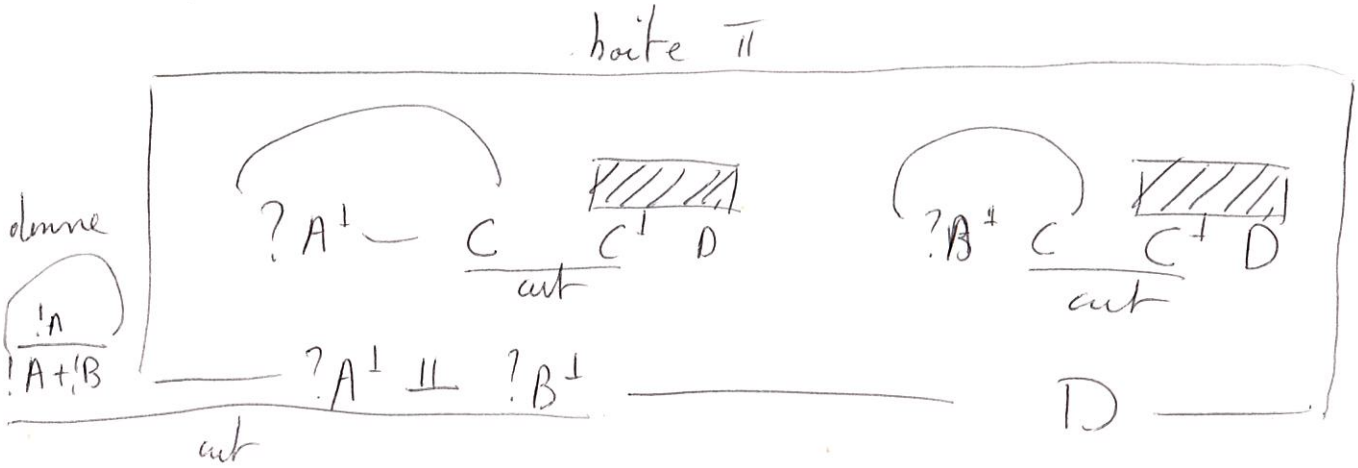
A +f B ⟜ !A + !B



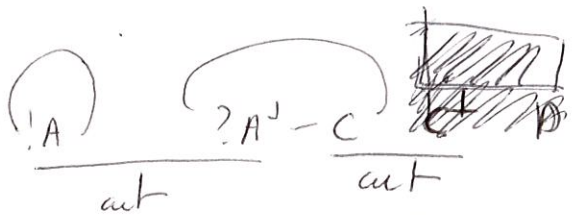
La traduction interprète les règles commutatives:
on peut symboliser l'élimination avec laquelle
commute l'élimination du plus par "l'application"
réseau : 

au quel cas cette contraction s'écrit





alors que la reduction dans ~~ff~~ correspond à calculer ~~d'abord~~ ainsi



$!A + !B \quad \multimap \quad A \parr B$

boite π

boite $\wedge C$

affaiblissement $?(B \Rightarrow C)^\perp$

$!A \quad C^\perp$

$(A \Rightarrow C)^\perp$

$?A^\perp \quad ?(A \Rightarrow C)^\perp \quad ?(B \Rightarrow C)^\perp \quad C$

$?A^\perp \quad \wedge C \quad (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C$

boite $\wedge C$

affaiblissement $?(A \Rightarrow C)^\perp$

$!B \quad C^\perp$

$(B \Rightarrow C)^\perp$

$?B^\perp \quad ?(B \Rightarrow C)^\perp \quad ?(A \Rightarrow C)^\perp \quad C$

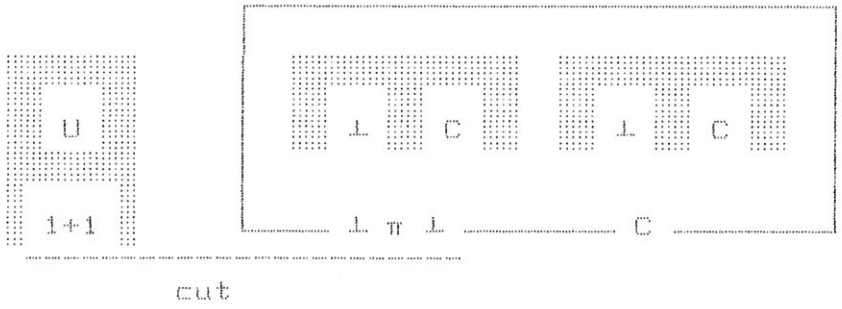
$?B^\perp \quad \wedge C \quad (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C$

$?A^\perp \quad \pi \quad ?B^\perp \quad \wedge C \quad (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C$

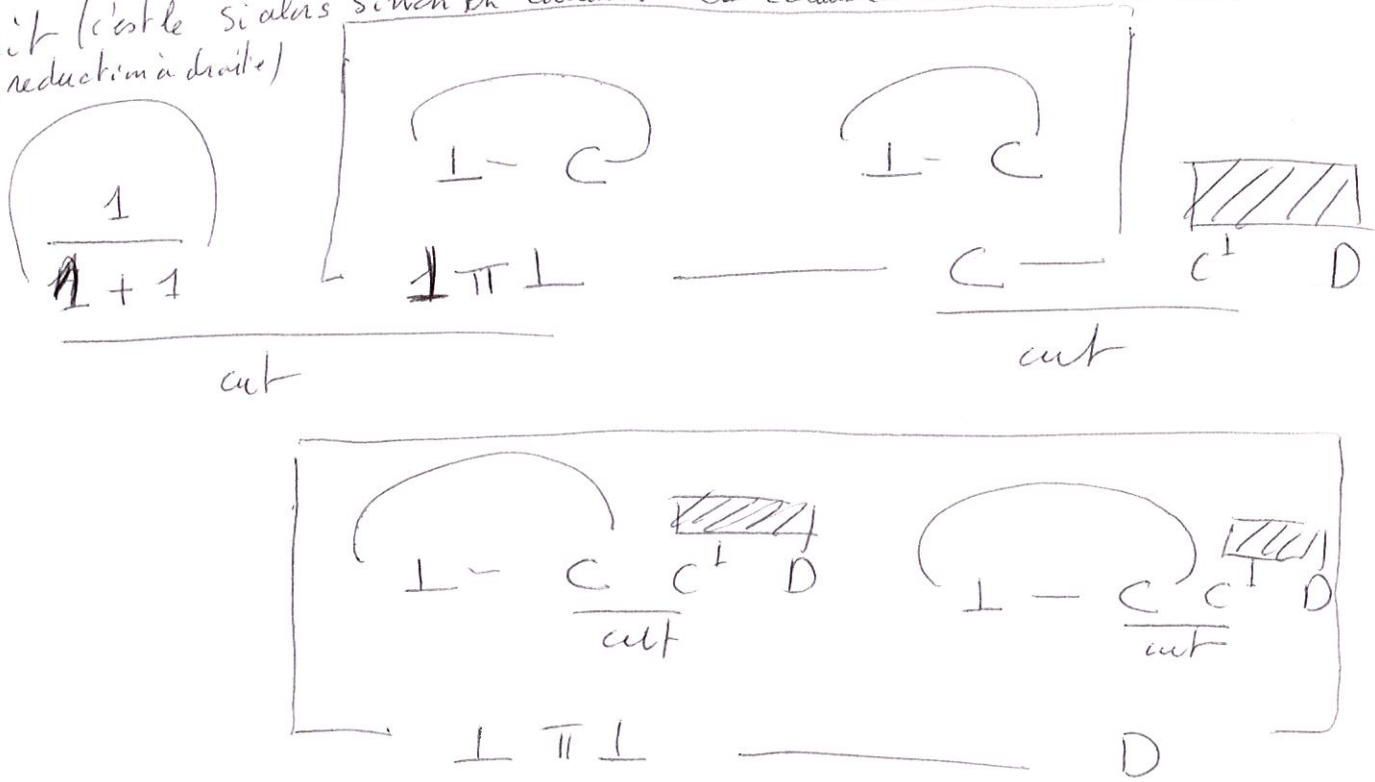
un cas particulier de somme : bool

Comme 1 est exclamable,on prend bool = 1 + 1,et tout va bien :

Vrai : $\dfrac{1}{1+1}\ g+$ Faux : $\dfrac{1}{1+1}\ d+$



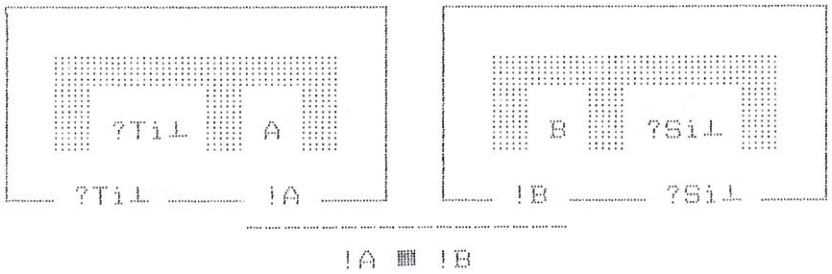
Dans ce cas de "somme" le schema de reduction commutative s'ecrit (c'est le si dans sinon en calculant les clauses avant le booléen! c'est la reduction à droite)



IIc | La traduction du produit

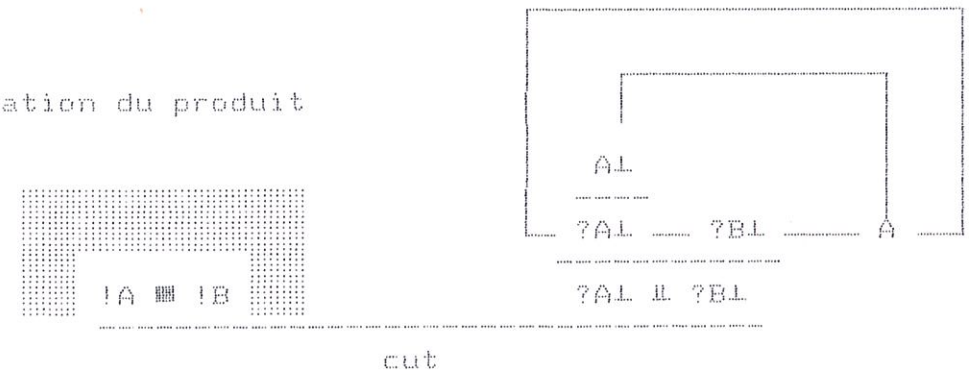
On peut traduire le produit par /\ θ (!A ⟜ !B ⟜ θ) ⟜ θ
en économisant un " ! "(voir plus loin une dicussion plus générale
sur une traduction économique des types inductifs) et même,par
!(A ⅋ B) =!A ⅋ !B,puisque tous ces types sont linéairement équivalents
à /\ θ !(!A ⟜ !B ⟜ θ) ⟜ θ.

Introduction du Produit

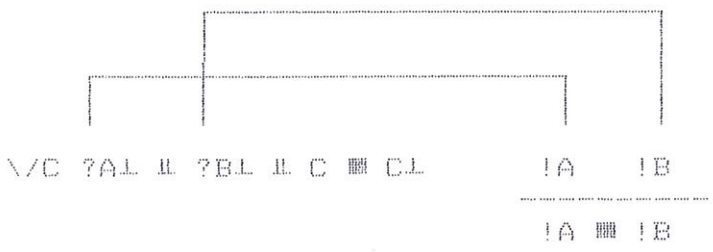


Boite affaiblissement = ?B⊥

Elimination du produit



/\ C (A ⇒B ⇒C) ⟜ C ⟜ !A ⅋ !B



On remarque avec plaisir que si A et B sont exclamables on peut
traduire leur produit par A ⅋ B.

III | Une traduction économique mais incomplète des objets inductifs

Le principal attrait du système F ,à savoir les types inductifs et les définitions par récurrence de fonctions sur leurs termes, est hélas bien mal représenté en logique linéaire par la traduction proposée au début.

L'idéal serait de traduire un type inductif T

T = /\ θ (S1 => S2 =>.....=> Sn => θ)

avec Si = Ti1 => Ti2 =>....Tiki => θ

Ti type ne contenant que des occurrences positives de θ

par

T- =/\ θ (!S'1 ⟜ !S'2 ⟜ ..... ⟜ !S'n ⟜ θ)

avec S'i =Ti1 ⟜ Ti2 ⟜ ....Tiki ⟜ θ

ce qui s'accorde bien avec le sens de " ! " et de son dual " ? ";seuls les constructeurs sont utilisés plusieurs fois et ont besoin d'être identifiés (contractés).

Pour ce qui est de traduire les termes il n'y a ,comme on va le voir, un algorithme,sinon clairement présenté ,du moins trés simple à utiliser une fois construits quelques exemples.Les constructeurs et les fonctions définies par récurrence sont tout aussi simples à traduire et les réseaux qui les représentent plus naturels que les termes de F.

Le seul défaut de cette traduction -et il lui fait perdre beaucoup de son interêt-c'est de ne pas prendre en compte toutes les fonctions définissables par récurrence dans F sur un type inductif.

J'ai toutefois espoir,en étudiant la condition que fait apparaître ce problème,à savoir :

il existe un réseau de conclusions Til[!α] ⟜ !Til[α]

(que j'ai baptisée Til linéaire en α) de le délimiter,et de pouvoir programmer de manière naturelle en logique linéaire.

IIIa | Quelques utilitaires

1 | L'application induite

Soient A un type (linéaire) n'ayant que des occurrences positives de θ et TθP un réseau de conclusions T⊥ , θ et eventuellement d'autres conclusions de types ?Xi.Alors il existe un réseau de conclusions A[T/θ]⊥ et A .

Ce réseau s'obtient par décomposition de l'axiome A(θ)⊥ A(θ).

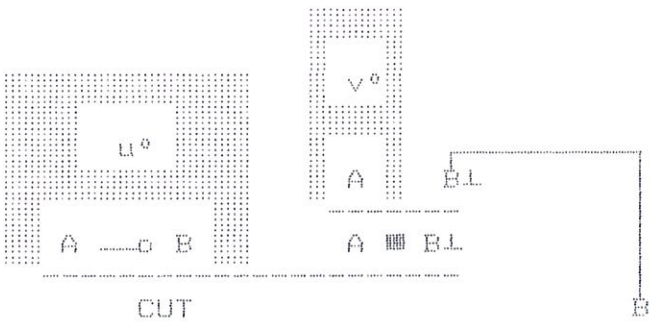
Puisque θ n'a que des occurrences positives dans A ,s'il on remplace cet axiome par le réseau TθP,on ne modifie en rien A(θ) et par contre A(θ)⊥ se trouve changé en A[T/θ]⊥.Pour que le réseau obtenu soit correct il faut que les Xi ne contiennent pas de variables de type abstraites au cours du calcul de A[t/θ]⊥ .Pour les boite ! on a pris la précaution "?"

Si T est un type inductif et A l'un des Til et TθP le réseau canonique de T dans θ obtenu en applicant T à θ (ce réseau est tout simplement obtenu en prenant l'intérieur de la boite /\ θ lorsqu'on décompose l'axiome T T⊥) ce réseau traduit l'application de Til[T/θ] dans Til induite par x:T => x (θ) de T dans θ qui n'est pas trés simple dans F.

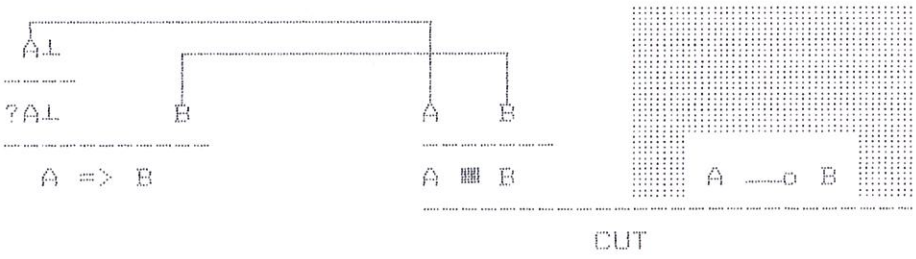
2 | L'utilisation de ⟜

La traduction de l'application et de l'abstraction ne sont presque pas affectées par le remplacement de => par ⟜ :

l'application de u:A ⟜ B à v:A



On n'a jamais à appliquer une fonction à !Z (aucun terme de F n'est traduit ainsi et c'est justement pour éviter de rajouter un ! devant le type des termes en position d'argument que l'on cherche à remplacer => par ⟜)Toutefois cela est très faisable et montre la compatibilité des traductions : il suffit de "dérélicter" le A⊥ de A ⟜ B c'est à dire, plus formellement :



Pour ce qui est de la construction d'un type ⟜ ,on remarque que ce signe n'apparait qu'en position négative et qu'il s'agit donc d'un ⅋ , obtenu comme en Ia ,D1la (c'est le cas où l'on construit le type d'une vltu ,tout comme ici) si ce n'est que la boite ! n'est pas nécessaire.

IIIb |Une traduction sans coupure des objets normaux de type T

Si l'on oublie un peu le formalisme de F pour exclure les bizareries dans le genre /\ Θ ⅋ (f Θ => Θ) f de type entier (mais n'est ce pas là un défaut de F ?) un objet de type T est obtenu par le constructeur ci de type Si[T/Θ] à partir des ki objets ti1...tiki de type Ti1[T/Θ]....Tiki[T/Θ].

On va traduire un objet inductif de type T par un réseau t° de conclusions T⊥,?X1..?Xk⊥correspondant aux éventuelles variables libres de t.

On se propose,dans un premier temps,d'associer au terme t ainsi formé un réseau tΘ° de conclusions Θ

?Si⊥ indéxé par ci (*1)

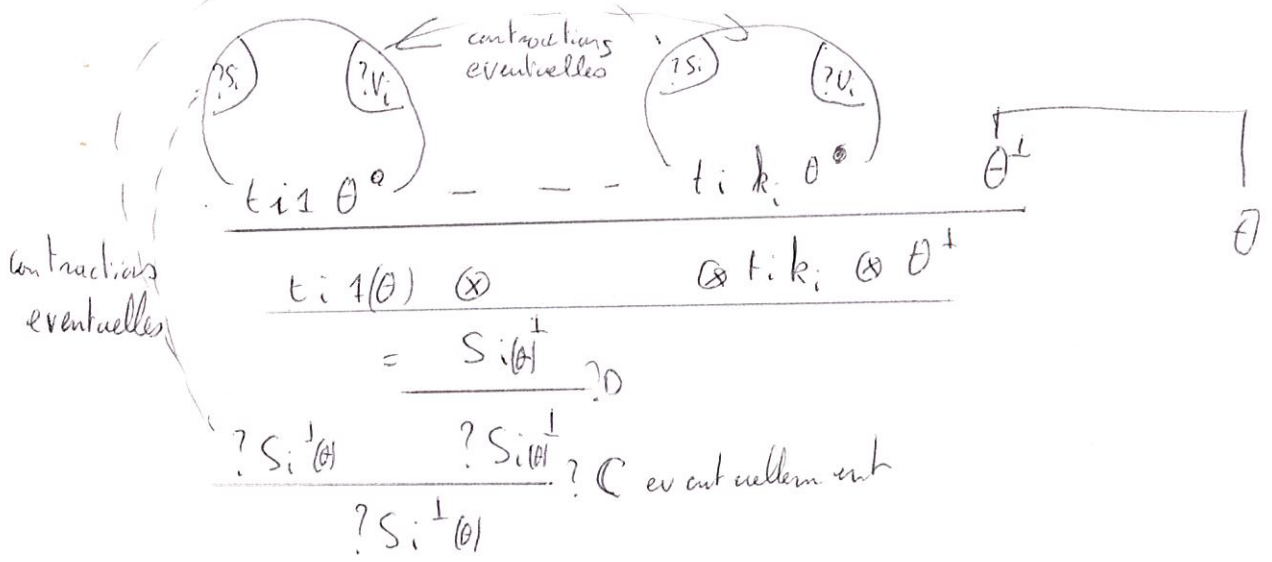
et eventuellement ?S1⊥..?Sn⊥ (indéxées par c1 cn) (*1)

?X1⊥..?Xp⊥ correspondants aux types des variables libres dans les til

construit à partir de réseaux tilΘ° de conclusions Θ obtenus à partir des til° par le procédés décrit au paragraphe précédent.

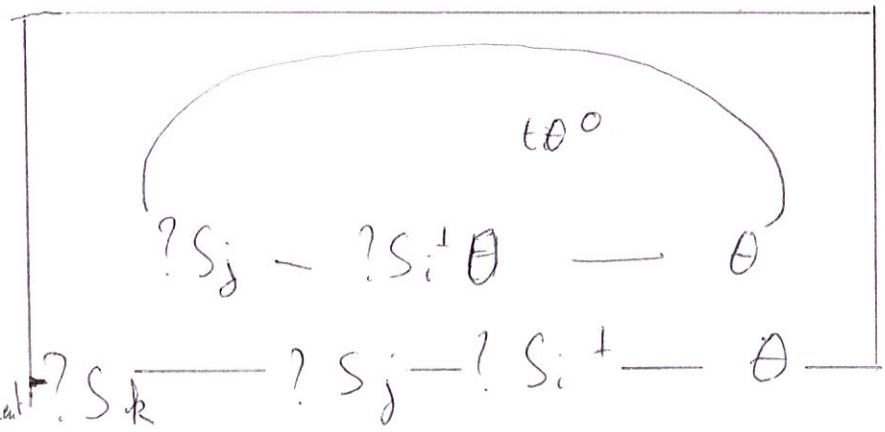
Si Til=T⊥ , tilΘ° coincide avec cette traduction (ce qui justifie la m@me notation)

(*1) Il est nécessaire,pendant que l'on traduit d'indexer les Si par cj car un type inductif peut avoir des Si egaux (la somme de A et A, par exemple)



Ensuite il faut contracter (s'il y a lieu)
— les sorties ?Sk indéxées par ck apparaissant dans les ti1θ°
— la nouvelle sortie ?Si⊥ avec le ?Si⊥ obtenu par l'opération précédente
On a alors tθ°.

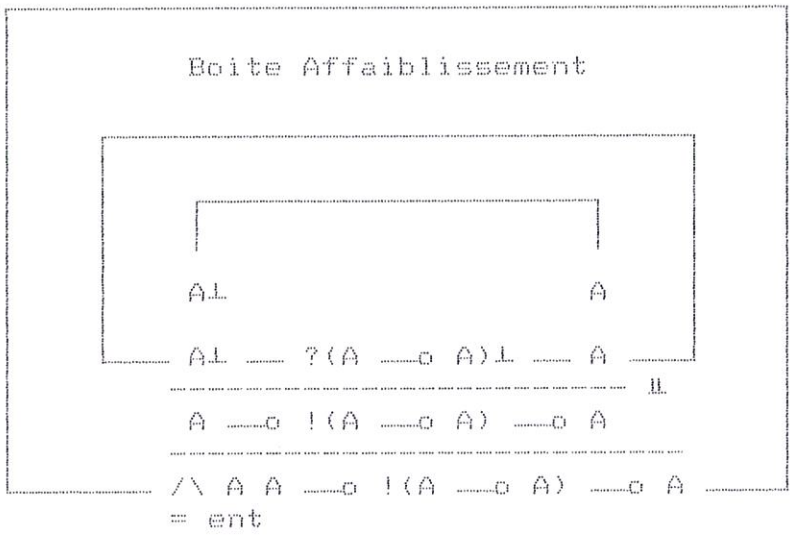
Enfin, pour obtenir t°, il ne reste qu'à
— faire apparaitre les ?Sk⊥ manquants en les indexant par ck
— prendre le par ⅋ des ?Sk⊥ et θ
— mettre le tout dans une boite /\ θ formant ainsi t° de type T-
(θ n'est évidement libre dans aucune des sorties auxiliaires ?Xn⊥ puisqu'on traduit un terme bien formé de F)



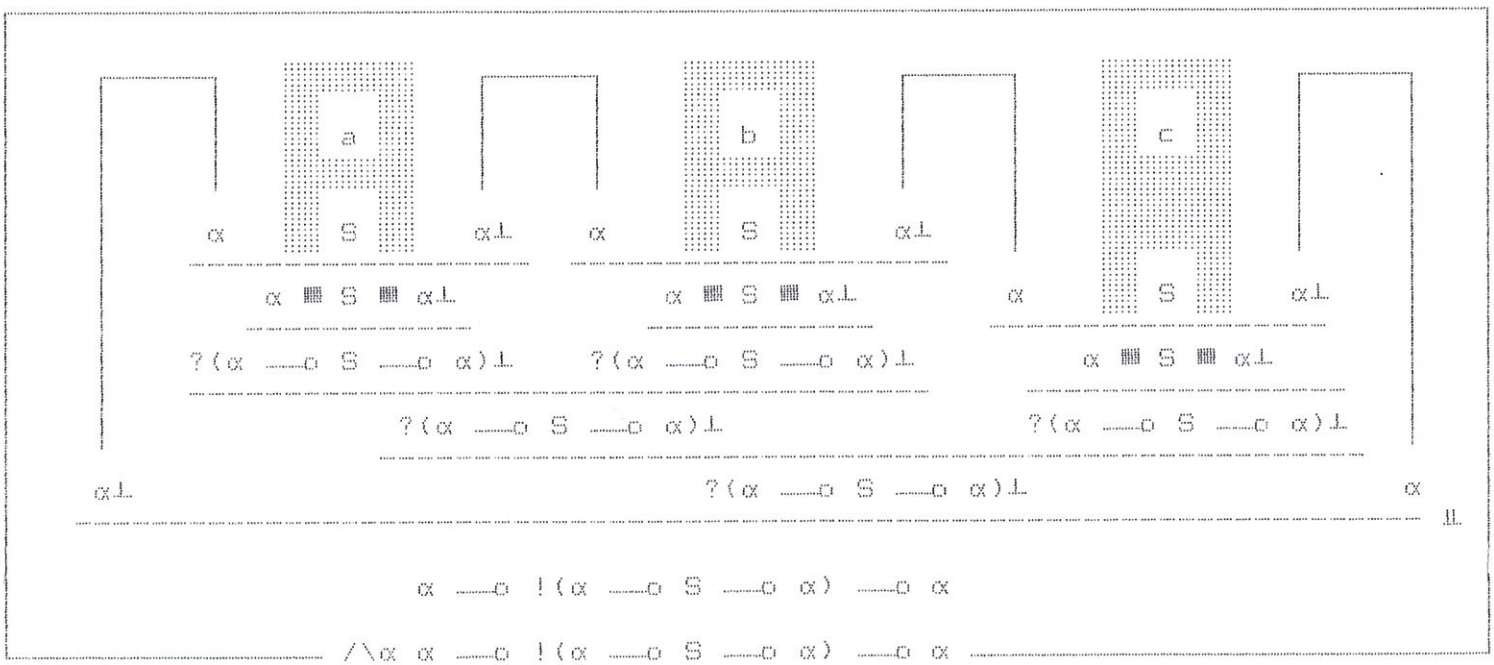
$$!S_k \multimap !S_j \multimap !S_i \multimap \theta$$

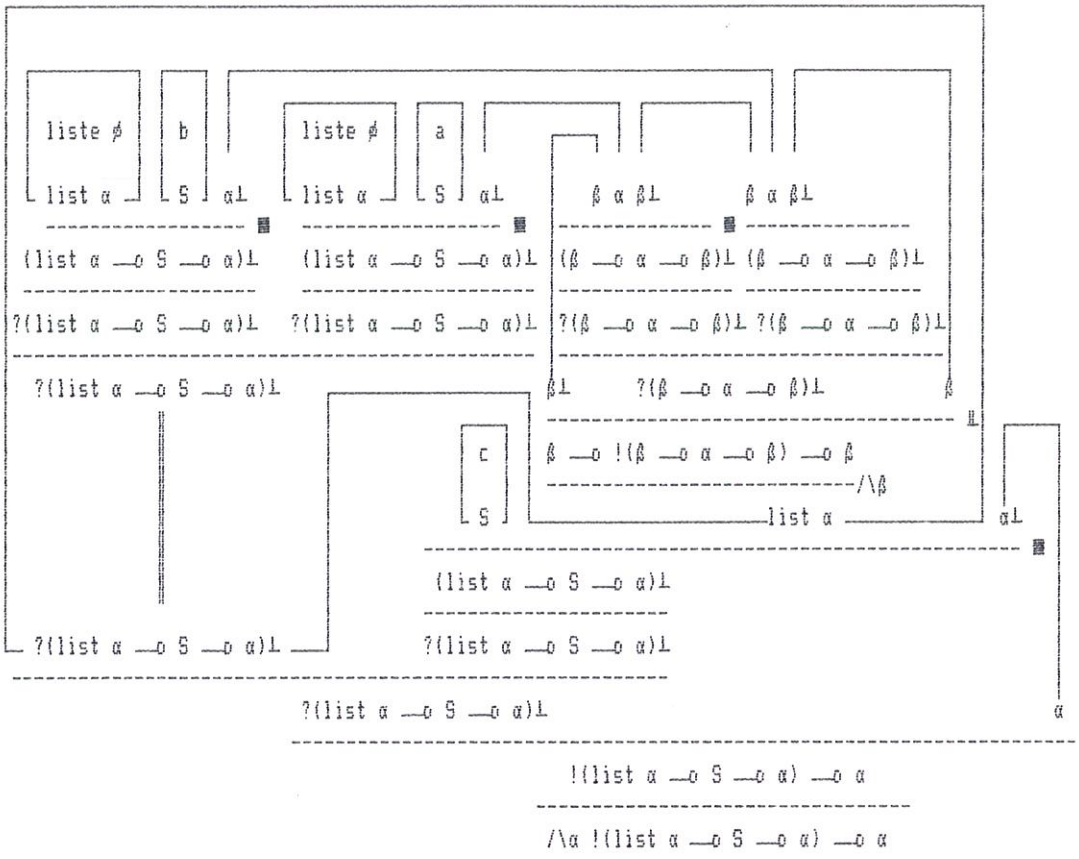
$$\Lambda\theta\ !S_k \multimap !S_j \multimap !S_i \multimap \theta : \theta$$

ZERO

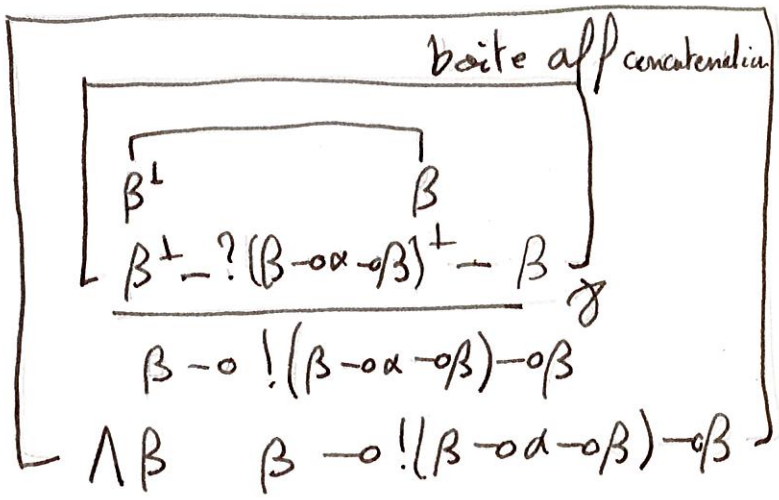


Liste a,b,c de type S





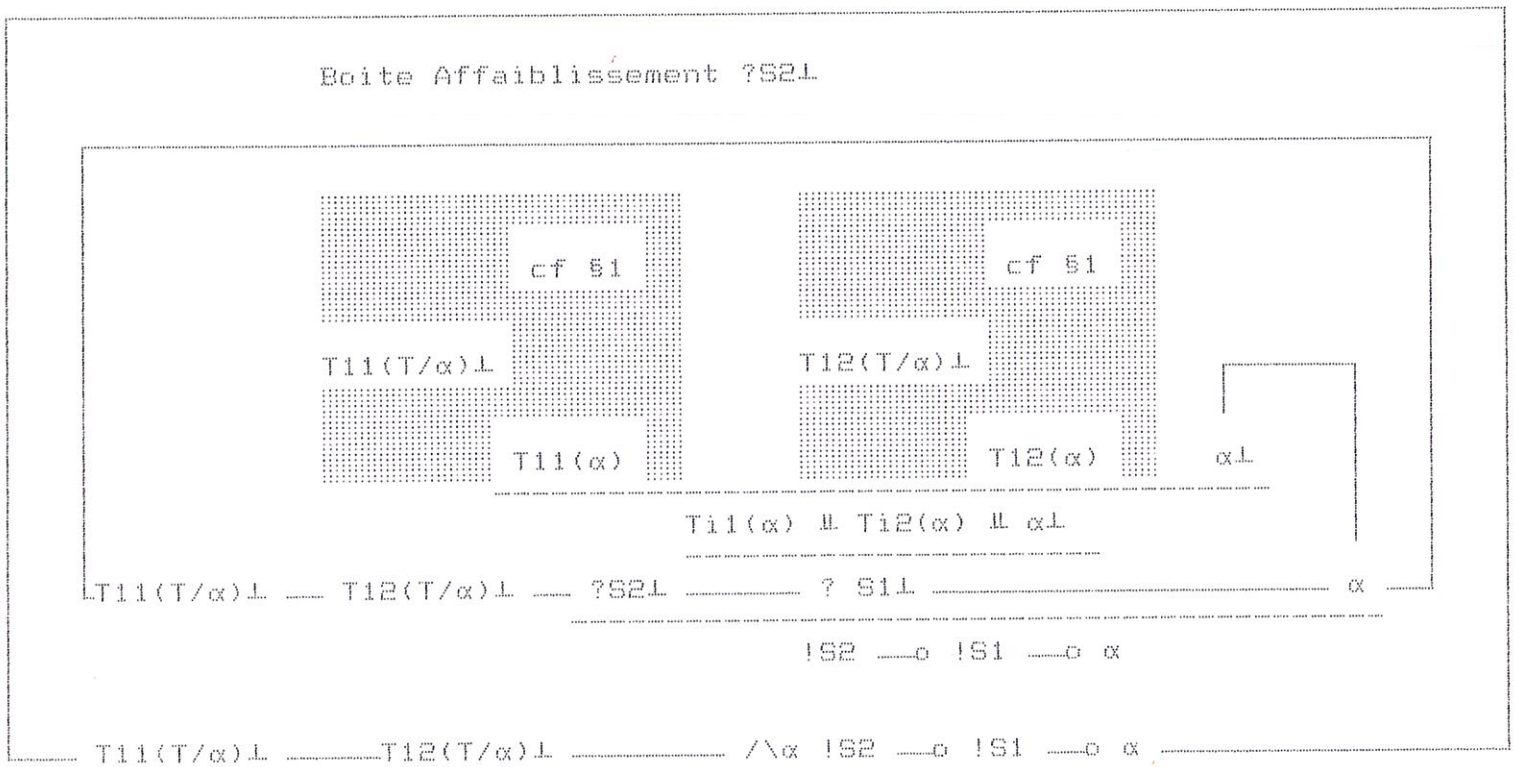
Avec [liste ∅ / list α] =



IIIc | Les constructeurs

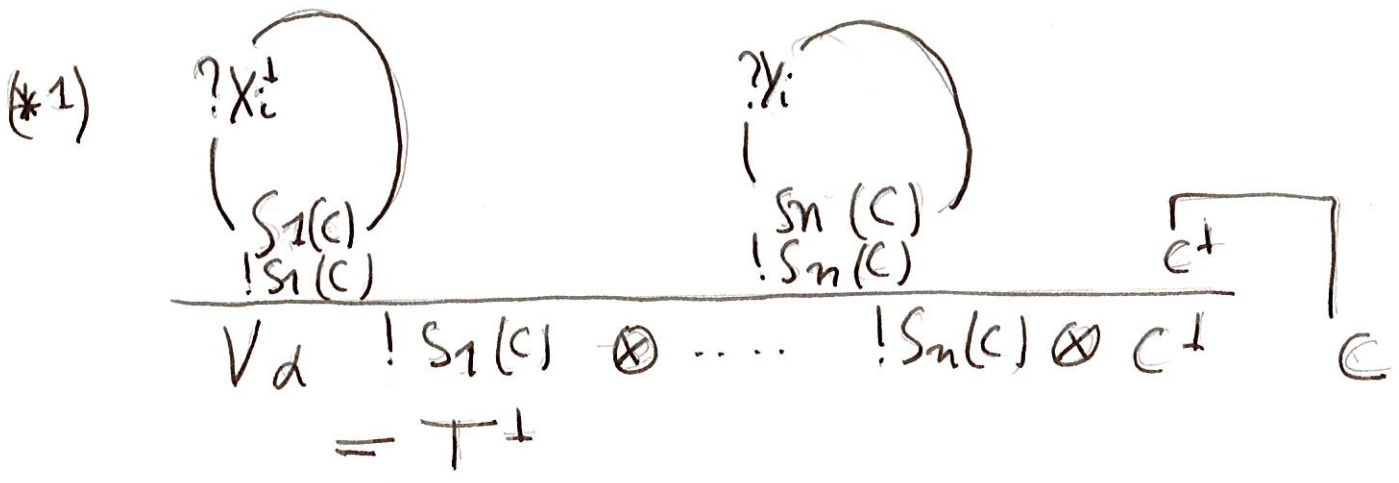
Les constructeurs sont d'une grande simplicité dans cette traduction : étant donné le réseau de conclusion Ti1[T/0]⊥ et Ti1 dont il est question dans "Quelques utilitaires 1",on construit sans peine la traduction du constructeur ainsi :

Constructeur C1 avec S1= T11 ⟜ T12 ⟜ α
T= /\α!S2 ⟜ !S1 ⟜ α



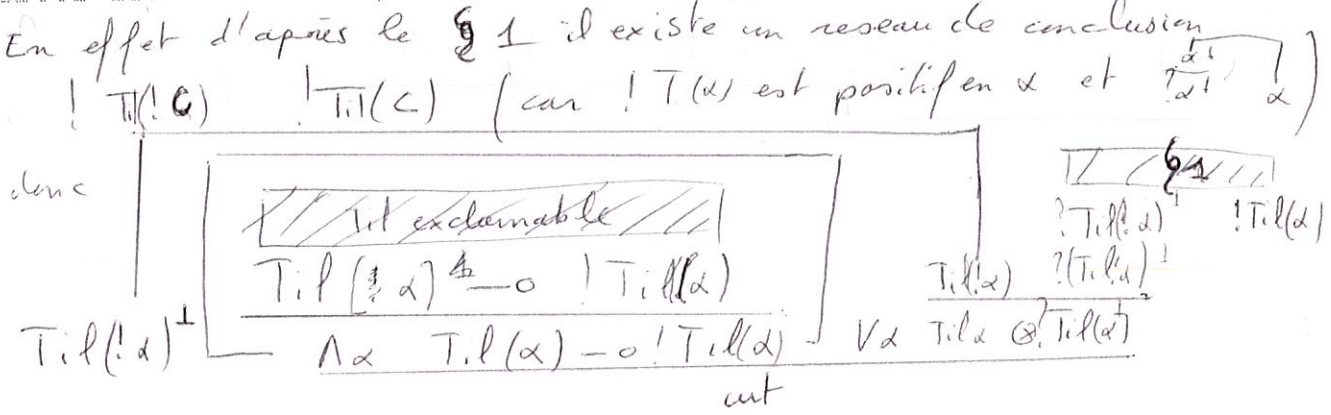
IIId | Les définitions par récurrence sur les types inductifs

Soient des réseaux de conclusions Ti1[C/0]...Tiki[C/0] ⟜ C = Si[C/0] ,on veut définir un réseau de conclusion T ⟜ C càd une fonction linéaire de T dans C.(cela définit a fortiori une fonction de type T => C) Rien de plus simple :



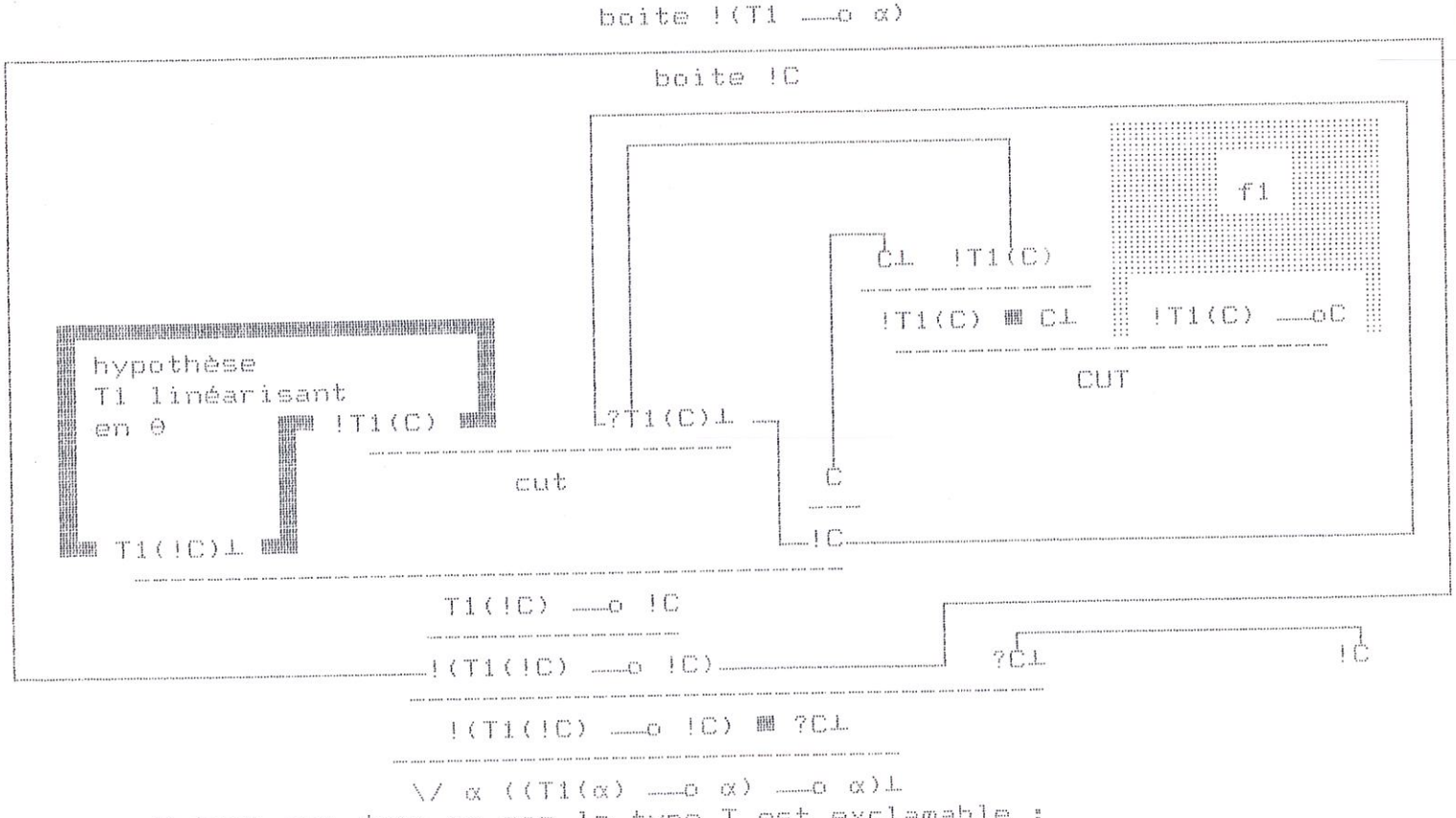
(*1) les types des paramètres doivent être précédés d'un "?" à moins que T soit exclamable (à moins que ce soit un type exclamable)

Le seul problème est, qu'a priori ,lorsque les fonctions sont traduites de F elles sont représentées par un réseau de conclusion Ti1[T/θ] => C càd !Ti1[C/θ].. !Tiki[C/θ] ⟜ C.
On voit de suite que si les Ti1 sont exclamables il n'y a aucun problème
Mais on peut faire mieux : supposons que les Ti1 vérifient:
Ti1[!α] ⟜ !Ti1[α] Ti1 est alors dit "linéarisant en α"
Cette condition est plus faible que Ti1 exclamable :



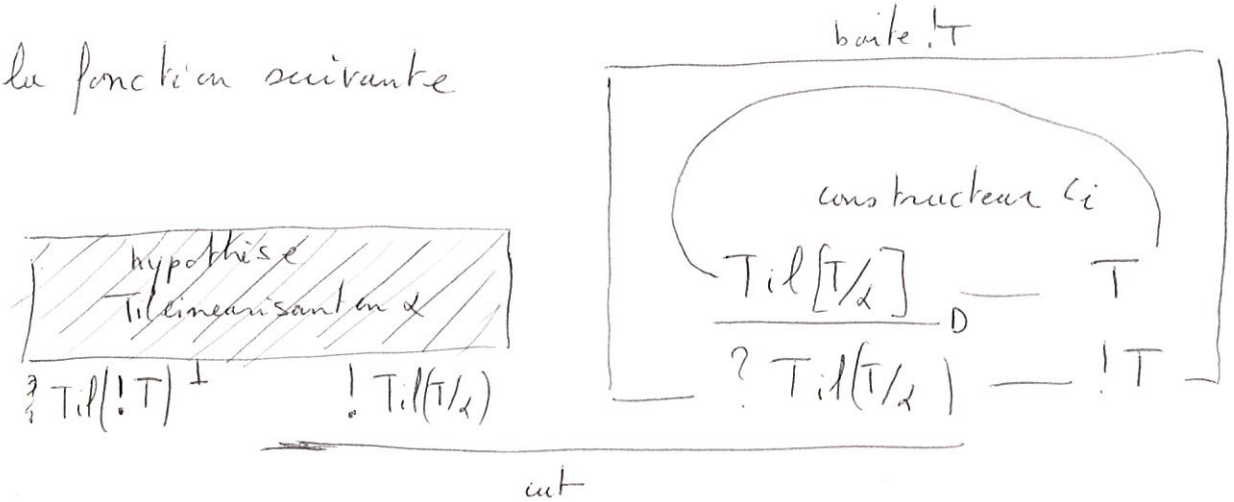
On peut cependant dans ce cas définir les fonctions par récurrence sur T

Pour dessiner le réseau je me suis limité à T =/\ α !(T1(α) ⟜ α) ⟜ α
f1 : T1[C/α] ⟜ C
Mais on devine sans peine ce que cela donne pour le cas général :
Mais on devine sans peine ce que cela donne pour le cas général : il suffit dans la boite !C de faire un CUT entre f de type Ti1 ⟜ Ti2 .. ⟜ C et !Ti1..!Tiki C⊥;ensuite en branchant chacune des sorties ?Ti1⊥ avec l'hypothése Ti1 linearisant en C on obtient un réseau dont le par des sorties est Si; on le met dans une boite !;aprés avoir fait cela pour tous les i, en prenant le " ▪ "de toutes ces boites et de C⊥ on obtient T (C)⊥ d'où T⊥,et le C de l'axiome C C⊥ donne la conclusion C.(très simple, mais pénible à expliquer)



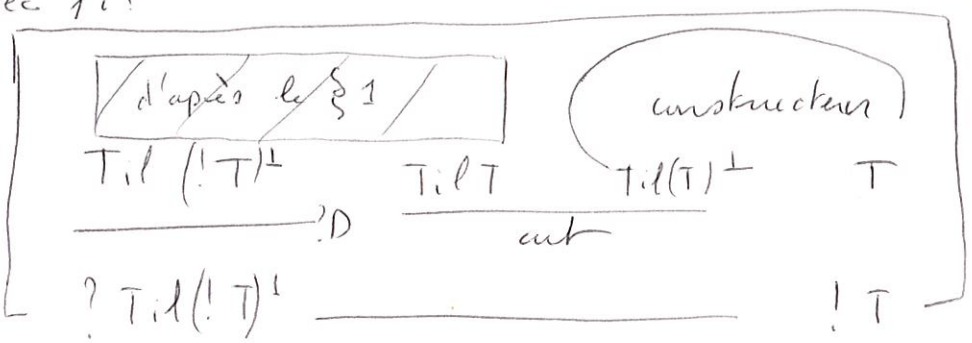
Notons que dans ce cas le type T est exclamable :

En effet il suffit de définir une fonction par récurrence de T dans $!T$ en prenant pour f_i $S_i(!T/\alpha) \Rightarrow !T$ la fonction suivante



(c'est en fait le constructeur d'objet de type $!T$)

Pour ce qui est de la première traduction c'est encore plus simple : on fait la récurrence avec f_i :



où dans cette traduction $S_i = !\, T_i l \multimap \alpha$

J'ai un peu étudié cette condition ,mais sans grand succès :

si U = /\ Θ U'et que U' est linéarisant en α, U l'est aussi (evident

si U = U1 _o U2 et que l'on suppose [illegible] (l'inverse car U2 est négatif en α) et U1 linéarisant en α, alors U l'est aussi.

(mais je doute que ceci soit intéressant, c'est surtout ⇒ qui pose problème)

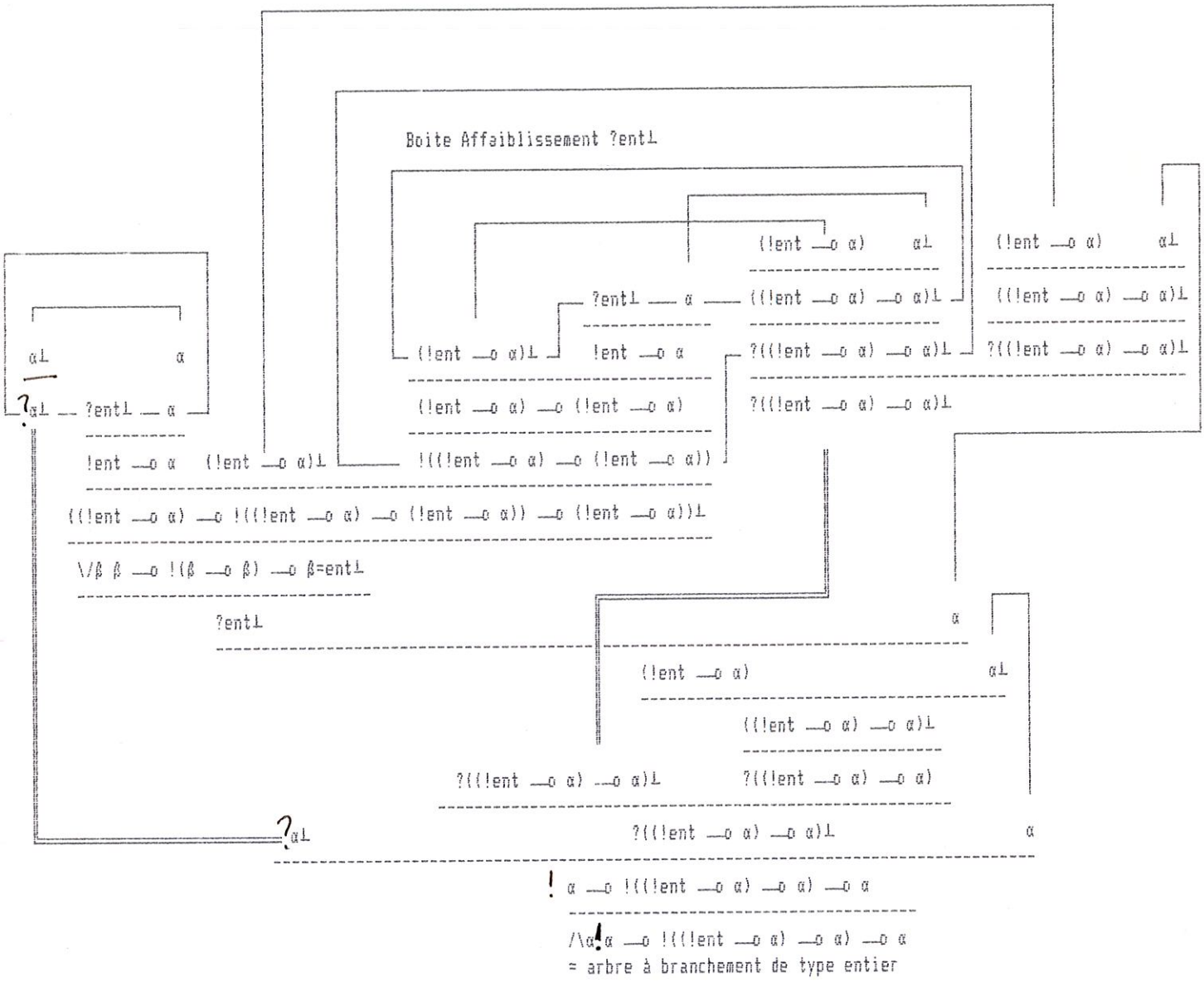
Par contre (ent ⟜ α) ne vérifie pas cette condition ...

Mais cette condition n'étant pas nécessaire on peut quand même esperer pouvoir définir des fonctions par récurrence sur les arbres à branchement entier, et même linéariser les fonctions définies sur ce type il faudra alors utiliser un autre schéma de récurrence.

$$\vdash ent^{\perp}, !(ent \multimap \alpha) \vdash !\alpha$$

$$\vdash ent^{\perp} \otimes !\alpha \;,\; !(ent \multimap \alpha)$$

$$\vdash (ent \multimap !\alpha) \multimap !(ent \multimap \alpha)$$

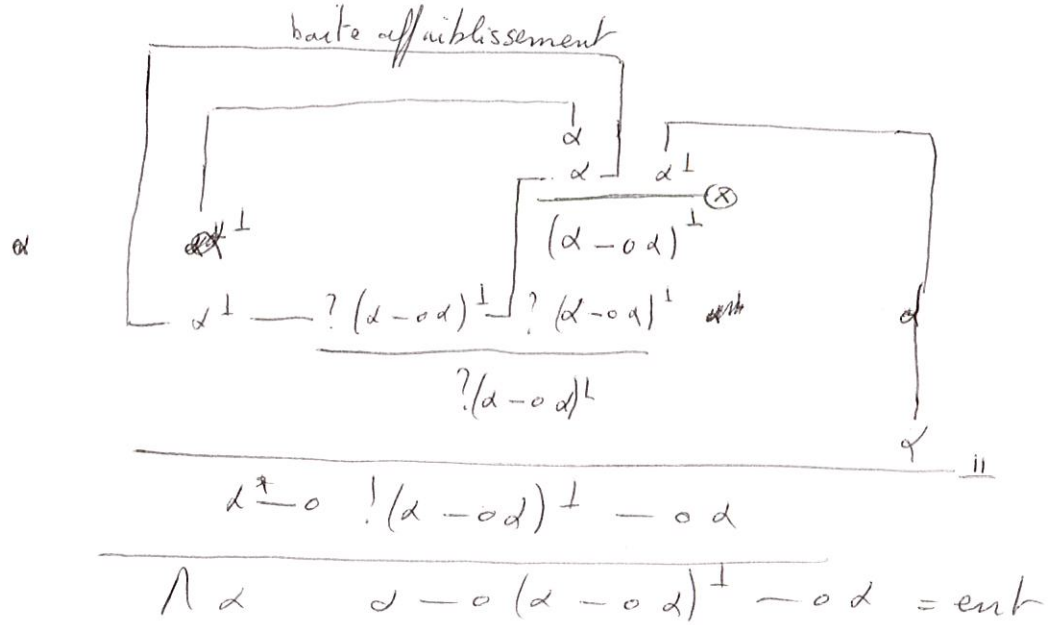


= arbre à branchement de type entier

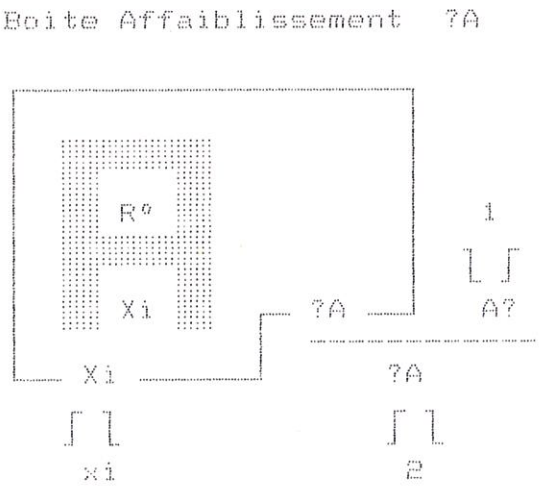
IV | Elimination des coupures et ß-reduction.Une éventuelle forme normale ?

IVa | Le "redex" affaiblissement / contraction

Les traductions font apparaître une autre distinction que l'ordre des contractions ,par exemple entre 1 et successeur de 0 (une fois éliminées les coupures):
successeur de 0 se réduit en (l'autre traduction ne fait que transposer ce problème):

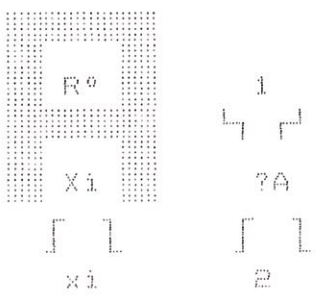


Il va sans dire qu'il en est de même chaque fois que l'on applique une lambda expression dont la variable liée n'apparait pas dans le corps
Je pense qu'il y a lieu de considérer la situation :



(ou même si l'affaiblissement n'est branché nulle part)

comme une sorte de "redex" (d'ailleurs en calcul des sequents,on voit bien qu'un affaiblissement suivi d'une contraction ne change rien) qui se réduit en :



(donc ces R° tout court)

Ce réseau est évidement aussi correct que le précédent:il suffit de placer l'interrupteur de la boite sur la permutation circulaire obtenue en intercalant ?C dans l'ordre des passages successifs par les conclusions de v° (lorsqu'on parcours l'unique grand cycle de v°) et de positionner le par sur sa prémisse de droite et les cycles du second réseau sont exactement ceux du premier : le premier n'ayant qu'un seul cycle le second aussi.(et cela vaut pour toute position des autres interrupteurs qui sont communs).
Je pense qu'en remplaçant le second par le premier il n'y a pas plus de problème (en pensant au calcul des séquents,par exemple) mais je n'ai pas démontré la correction du réseau.

IVb | ß-réduction et élimination des coupures

On se propose de montrer que si t ß t' t° se normalise en un réseau équivalent à t'° modulo :
- l'ordre des contractions
- le "redex" affaiblissement / contraction

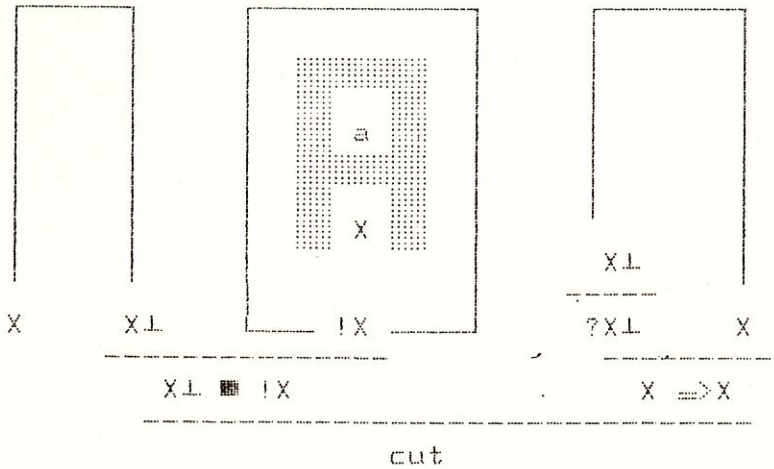
On commence par vérifier cela pour une ß-réduction (*1) (*2)
Il s'agit alors de montrer que ((⊓‖ x u) a)° peut se ramener à un réseau équivalent à u[a/x]°.
On procède par récurrence sur u.

— u est une variable

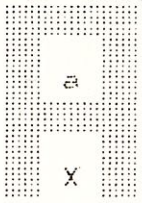
— u = x

(*2) j'ai systématiquement supposé que l'application se traduisait par une coupure - cela n'a en fait aucune importance ici car le cas avec une coupure "en trop" se ramène en une élimination au cas sans -

u=x
(⊓‖ x x) a :



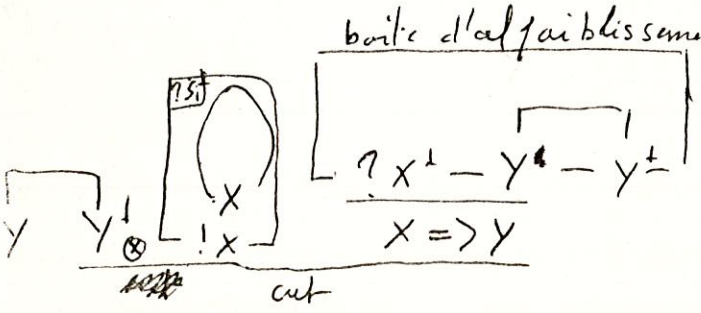
se réduit en



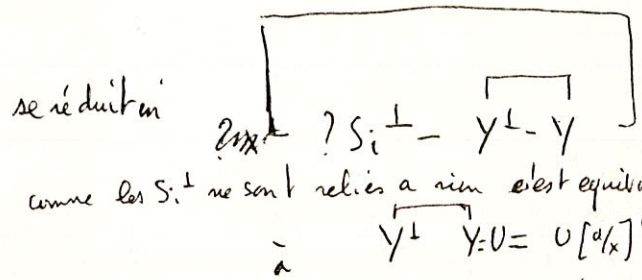
(heureusement!)

— u = y

C'est pour ce cas qu'il faut considérer l'équivalence à affaiblissement / contraction près :



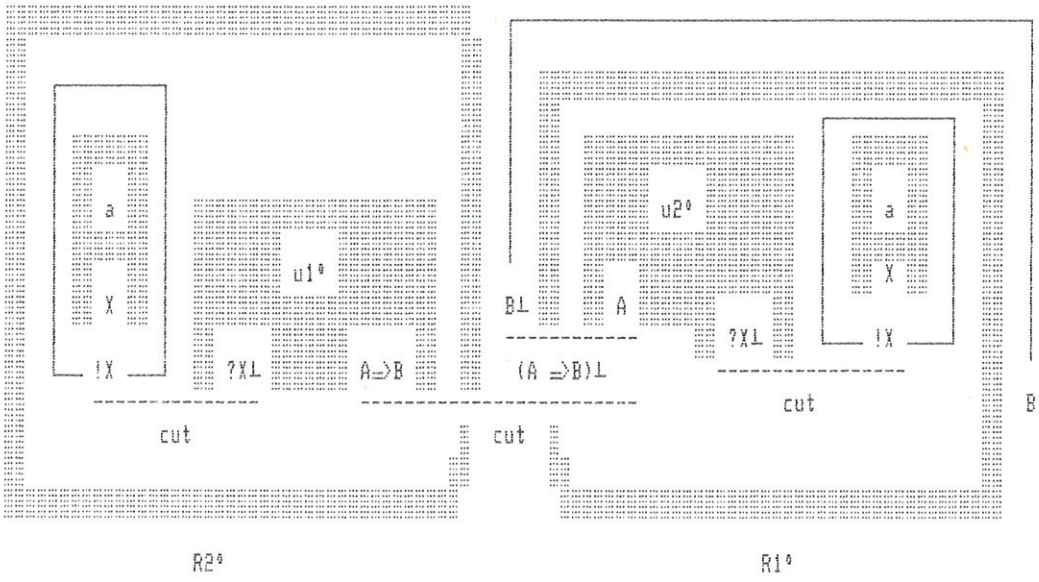
se réduit en



comme les $S_i^\perp$ ne sont reliés à rien c'est equivalent à

(*1) j'ai supposé dans cette preuve fastidieuse que x apparaissait toujours dans le corps de l'abstraction en fait s'il n'apparait pas on tombe précisément sur le "redex" signalé precedement comme dans le cas U=y

u est une application u = u1 u2
On suppose que ((⊓‖ x u1) a)° =\ u1[a/x]°; ((⊓‖ x u2) a)° =\ u2[a/x]°
((⊓‖ x u) a)° se réduit en une étape en :



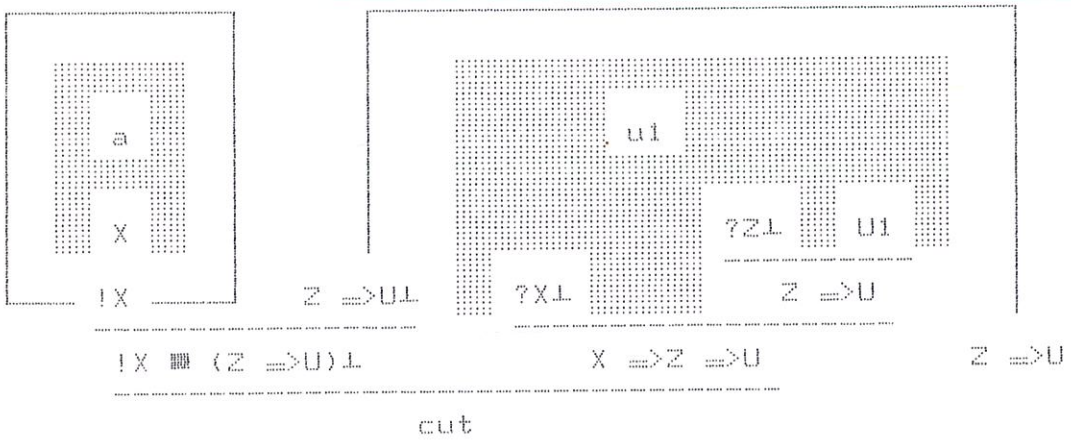
R2° R1°

On remarque que R_1° est obtenu par une reduction à partir de $((\lambda x\ v_1)\ a)^\circ$ et qu'il se reduit donc en $v_1(a/x)$ même chose pour R_2° avec v_2. et dans ce cas le reseau si dessus représente R_2° appliqué à R_1° se reduit en $v_1(a/x)^\circ$ appliqué à $v_2(a/x)^\circ$ c'est à dire $v_1\ v_2\ [a/x]$

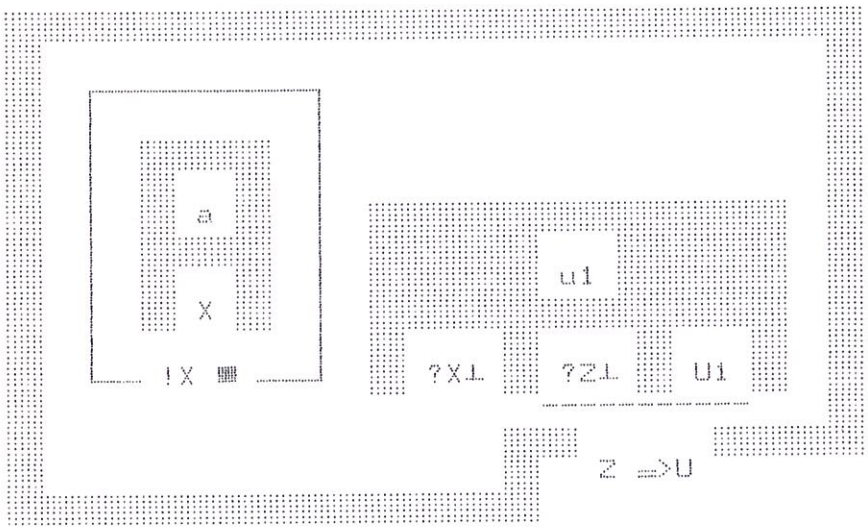
On remarque r1° est obtenu par une élimination à partir de
((⅋ x u1) a)° (idem pour r2°).Ces deux réseaux se réduisent donc en
des réseaux équivalents à u1[a/x]° et u2[a/x]°.On voit donc que le
réseau ci-dessus une fois r1° réduit en u1[a/x]° et r2° en u2[a/x]°
traduit u1[a/x] u2[a/x] ,c'est à dire u[a/x].

— u est une ⅋ -expression u = ⅋ y u1

(⅋ X (⅋ Z u1)) a

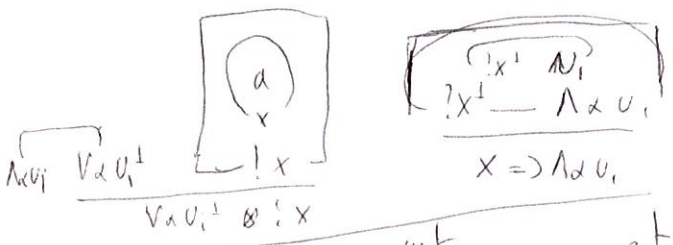


Se réduit en

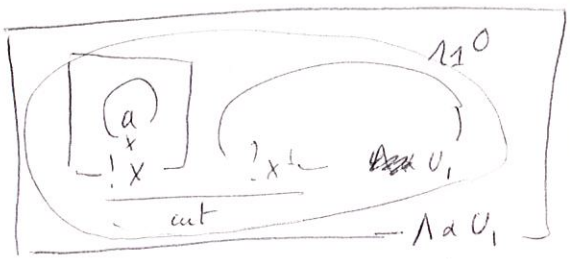


R1°
et R1° est obtenu par une réduction à partir de ((⅋ x u1)a)°qui se réduit par
hypothèse en u1[a/x]°,le tout se réduit donc en(⅋ z u1[a/x])° =(⅋ z u1)[a/x]°

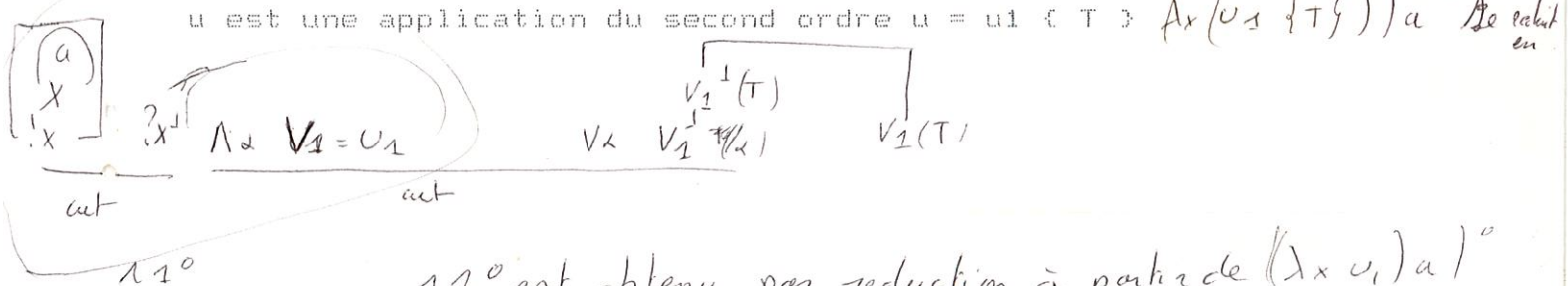
— u est une /\-expression u = /\ α u1



se reduiten



et s'obtient en une réduction à partir
de la traduction de((λx U1) a)° qui se réduit
par hypothèse en U1[a/x] le reseau se reduit
donc en Λα(U1[a/x])° cqfd.



11° est obtenu par reduction à partir de $((\lambda x\, v_1)\, a)^\circ$ et se reduit donc en $(v_1[a/x])^\circ$ on voit alors que le reseau se reduit en $((v1[a/x])\,\{T\})^\circ$ cqfd.

IVc | Une éventuelle forme normale

Pour programmer en logique linéaire d'une manière "confortable" je pense qu'il est préférable de ne pas voir les réseaux correspondants aux programmes, aux données ou aux résultats des évaluations. Il deviennent en effet rapidement illisibles (s'il y en a tant ici c'est pour voir à quoi ils ressemblent, et pour se convaincre que globalement ils sont quasi incompréhensibles -et peut-être aussi par souci esthétique ?). Toujours est-il qu'il faut donc que la machine puisse déterminer l'entier, la liste ou l'arbre dont il est question notamment pour nous retourner en valeur un objet compréhensible.
En clair : il faut un théorème de forme normale.
A mon avis, modulo l'équivalence envisagée plus haut les objets inductifs ont une forme normale (c'est particulièrement visible pour les entiers) qui tient à une espèce de "propriété de sous formules" de ces réseaux.
Je ne l'ai malheureusement pas (encore ?) démontré.
Ce qui en tous cas est sûr, d'après ce qui précède, c'est que si t° (traduction du F-terme t) se réduit en u° et que u° est la traduction suivant l'algorithme proposé, u°est équivalent à fnt° .