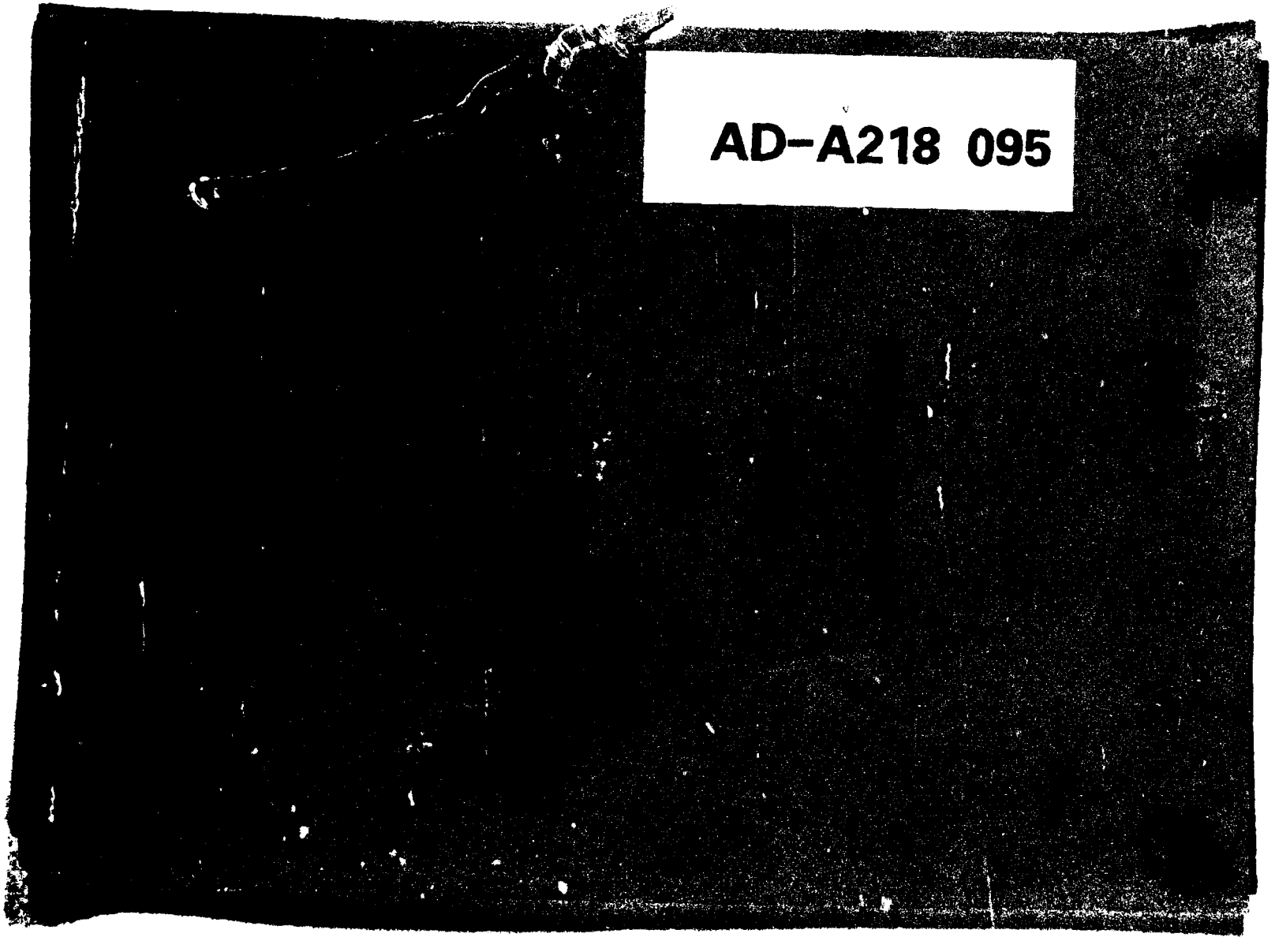

AD-A218 095

Initiation au λ-calcul

Gérard Huet

INRIA

An Initiation into Lambda Calculus

Notes de cours du DEA "Fonctionnalité, Structures de Calcul et Programmation" donné à l'Université Paris VII en 1987-88 et 1988-89.

© G. Huet 1988,1989

15 Juillet 1989

Table des matières

Introduction 3

I Lambda-calcul pur

1 La lambda notation 5
2 Notation abstraite, substitution 6
3 Réduction, conversion 11
4 Le théorème du losange 13
5 Le théorème des développements finis 31
6 Le théorème de standardisation 35
7 Approximations 40
8 Variations sur la notion de calcul 43
9 Séparabilité 46

II Lambda-calcul typé

1 Types conjonctifs 50
2 Interprétation des types 57
3 Typage polymorphe 63

Pour en savoir plus 75

Introduction

Ces notes de cours introduisent la notion de fonction calculable en étudiant le formalisme du λ-calcul.

Ces notes consistent présentement en deux chapitres.

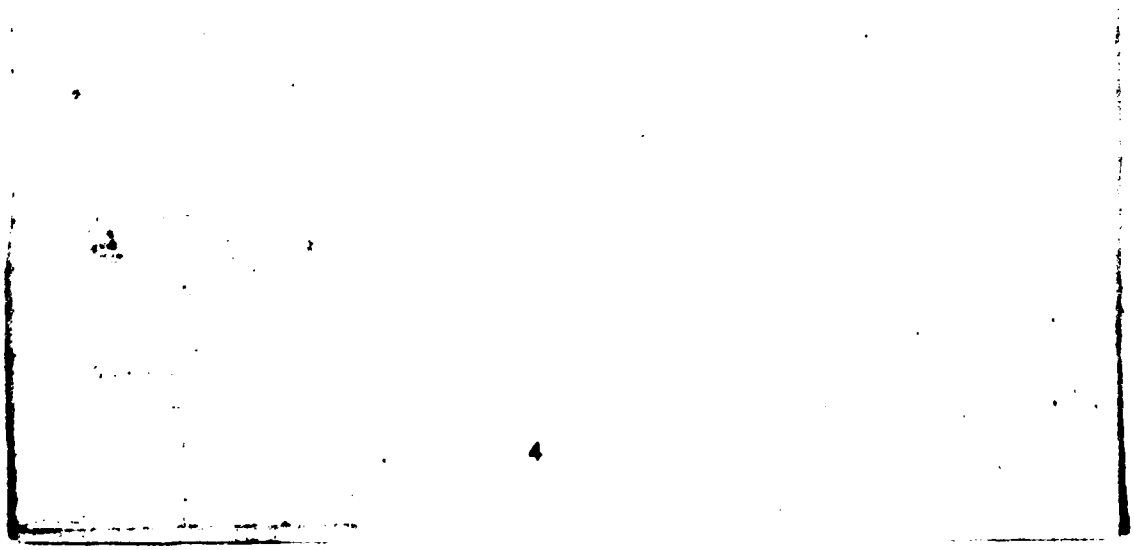
Le premier chapitre traite du formalisme de base : le λ-K-calcul pur avec la β-conversion. Ce formalisme est traité de façon homogène, avec l'usage systématique de la structure abstraite de de Bruijn. Les définitions sont exprimées de manière complètement constructive et même exécutable, dans le langage CAML. Une méthode uniforme de marquage permet de dégager simplement l'idée directrice des preuves.
Les principaux théorèmes présentés sont le théorème du losange et sa conséquence la confluence des calculs, le théorème des développements finis et sa conséquence le théorème de standardisation, enfin le théorème de Böhm et ses conséquences sémantiques.

Le deuxième chapitre traite de différents calculs typés, modélisant des langages fonctionnels divers, ou des systèmes d'inférence en déduction naturelle, suivant la correspondance de Curry-Howard. La caractérisation des termes normalisables (resp. fortement normalisables) correspondant à la discipline des types conjonctifs avec (resp. sans) type universel est démontrée par la construction d'un modèle syntaxique. Différentes disciplines de typage polymorphe sont étudiées.

Ces notes sont encore à l'état d'ébauche. De nombreuses démonstrations ne sont qu'esquissées. Il manque au chapitre des types les sections correspondant aux types dépendants et aux disciplines non prédicatives. Une version plus complète de ces notes traitera également de la récursivité, exprimée dans le λ-calcul, de la théorie des combinateurs, et enfin des modèles du λ-calcul.

DTIC
COPY
INSPECTED
4

By perform 50
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1



I . Lambda-calcul pur

1. La lambda notation

Il n'y a pas de notation généralement admise en mathématiques pour décrire une expression fonctionnelle, c'est à dire dénotant une fonction. A vrai dire, la notion même de fonction est relativement récente en mathématiques. Jusqu'au XIXème siècle, les fonctions n'étaient pas des objets mathématiques à part entière, mais juste une manière de parler pour désigner des procédés permettant de calculer des valeurs mathématiques à partir d'autres. Par exemple, à partir des fonctions sin et cos, on peut fabriquer la fonction qui aux réels x et y, fait correspondre le réel sin(x)+cos(y). On désigne généralement cette fonction par la notation :

$$x,y \mid\!\rightarrow \sin(x)+\cos(y) \qquad (1)$$

qui ne peut pas être employée comme expression, à l'intérieur d'une autre expression par exemple.

Les fonctions sont devenues des objets mathématiques à part entière après l'introduction par Cantor de la théorie des ensembles. Les fonctions de la théorie des ensembles sont confondues avec leur graphe, c'est à dire l'ensemble des paires argument-résultat. Il faut bien remarquer que cette nouvelle définition de fonction, extensionnelle, est fondamentalement différente de la notion intentionnelle d'algorithme, ou règle de calcul de la valeur du résultat à partir de la valeur de l'argument.

Nous allons dans ces notes nous attacher à la notion d'algorithme. Une fonction calculable sera une fonction qui peut être définie par un algorithme. La notion d'algorithme va s'appuyer sur un langage permettant d'écrire des expressions dénotant des valeurs ou des fonctions, et sur des règles de calcul spécifiant comment une expression peut expliciter une valeur, par évaluation progressive. Plusieurs difficultés surgissent.

a) Faut il que la notation distingue une valeur concrète, par exemple un entier, d'une valeur fonctionnelle? On distingue ici les langages typés (par exemple, CAML), et les langages non typés (par exemple, le λ-calcul que nous allons étudier). Certains langages sont intermédiaires, comme LISP, qui distingue les (C-VAL) des fonctions (F-VAL).

b) Le langage et ses règles de calcul garantissent-ils que l'évaluation d'une expression termine toujours? Nous verrons qu'une telle contrainte est nécessairement restrictive, et que les algorithmes ne définissent donc en général que des fonctions partielles, non définies sur certaines valeurs.

c) Comment peut-on désigner de manière non-ambigue l'argument d'un algorithme? Dans la notation ci-dessus, les meta-variables x et y sont utilisées pour cet usage. Cette utilisation traditionnelle de noms pour les variables liées, ou muettes, qui dénotent l'argument d'une fonction, est pratique pour le mathématicien, qui joue sur l'ambiguité entre l'algorithme et la valeur résultat sin(x)+cos(y) pour des valeurs données des arguments x et y. Mais les difficultés surgissent dès qu'on essaye d'étendre cette confusion à des fonctionnelles, c'est à dire des expressions qui prennent en argument ou retournent en résultat des fonctions. Les notations traditionnelles $\int f(x)dx$, $\partial f/\partial x$, $\Sigma_{i=1,10}E(i)$, $\forall x \exists y\ P(x,y)$ montrent qu'une notation fonctionnelle uniforme est désirable.

Nous allons maintenant présenter quelques notations concrètes pour l'expression fonctionnelle (1), dans différents langages.

λx. λy. ((+ (sin x)) (cos y))	λ-calcul de Church
function x → function y → sin(x)+cos(y)	langage CAML
[x][y]<<y>cos><<x>sin>+	AUTOMATH
[x,y] (+ (sin x) (cos y))	

Cette dernière notation sera utilisée dans ces notes de cours.

Toutes les notations concrètes ci-dessus souffrent du défaut d'avoir à utiliser un nom, c'est à dire une chaîne de caractères spécifique, pour noter les variables muettes. Elles ne rendent donc pas compte du concept abstrait d'algorithme, pour lequel le nom de ces variables n'est pas pertinent. Par exemple, [x,y] (+ (sin x) (cos y)) et [u,v] (+ (sin u) (cos v)) dénotent le même algorithme. Pourtant, il n'est pas aisé de spécifier rigoureusement, mais sans lourdeur, la congruence de renommage. Par exemple, il convient de ne pas confondre ci-dessus les noms x et y. La congruence de renommage est appelée traditionnellement α-conversion dans la notation du λ-calcul de Church. Nous éviterons ici ces difficultés en considérant un langage plus abstrait qui contourne cette difficulté.

2. Notation abstraite, substitution

2.1. Quelques propositions de notation abstraite

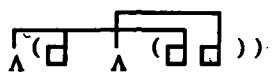
Un certain nombre de propositions de représentations des λ-termes modulo renommage des variables liées ont été faites. Nous en discutons

certaines sur l'exemple:

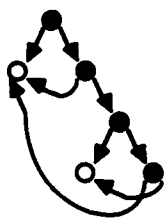
[x](x [y](x y)).

Remarquons que cette syntaxe utilise la notation (f x) pour indiquer l'application de la fonction f à l'argument x.

Une première proposition de syntaxe abstraite, n'utilisant pas de noms de variables, a été évoquée (mais non vraiment utilisée!) par Bourbaki:



Une autre proposition, liée à une implémentation sur ordinateur du λ-calcul, a été faite par C. Wadsworth. Elle utilise un formalisme de graphes:



Les deux propositions ci-dessus sont en fait très proches. Celle de Wadsworth présente l'avantage d'autoriser d'exprimer certains partages de sous-expressions. Elle ont toutes deux un défaut essentiel, celui d'être basées sur un langage bi-dimensionnel. De plus, elles ne permettent pas de parler facilement de sous-expressions, dans lesquelles certaines occurrences de variables peuvent être libres. Enfin, la représentation de Wadsworth n'exprime pas la bonne notion de partage.

La bonne notation abstraite pour les λ-termes est due à N. de Bruijn. Elle utilise l'idée d'indice de liaison: chaque occurrence de variable liée est représentée par un entier, qui est la profondeur relative du lieur de la variable. Sur notre exemple, on obtient:

[] (1 [] (2 1))

Ici les crochets de liaison [] doivent être lus comme un symbole unaire, l'abstraction. Les expressions parenthésées représentent l'application (ici opérateur binaire). Les entiers sont les indices représentant les occurrences de variables.

Remarquons tout de suite que cette représentation abstraite n'est pas une notation lisible, au contraire de la représentation concrète: les deux occurrences de la variable x sont représentées par deux indices différents, alors qu'inversement les deux indices "1" représentent des occurrences de variables distinctes x et y.

2.2. Définitions précises des structures

Nous avons donc besoin des deux langages, correspondant aux deux structures définies ci-dessous en CAML:

```
type lambda =              (* syntaxe abstraite *)
      Ref of num                    (* variable *)
   |  Abs of lambda                 (* abstraction *)
   |  App of lambda * lambda;;      (* application *)

type concrete =            (* syntaxe concrète *)
      Var of string                 (* x *)
   |  Lambda of string * concrete   (* [x] E *)
   |  Apply of concrete * concrete;; (* (E1  E2) *)

let closed = valid 0
    where rec valid n = function
      Ref(m)       → n ≥ m
   |  Abs(e)       → valid (n+1) e
   |  App(e1,e2)   → valid n e1 & valid n e2;;
```

On écrit Λ_n pour l'ensemble des λ-termes M valides dans un contexte de n variables libres, c'est à dire tels que valid n M = true avec la définition ci-dessus. Les termes fermés (dans Λ_0) n'ont pas de variables libres.

Voici maintenant l'algorithme calculant l'indice correspondant à un nom de variable dans un environnement donné, représenté par une liste de chaînes de caractères:

```
let index name = search 1
    where rec search n = function
      []            → Erreur "variable non liée"
   |  first::rest   → if first=name then n else search (n+1) rest;;
```

On donne maintenant l'algorithme traduisant un λ-terme concret fermé en la structure abstraite correspondante :

```
(* parser : concrete →  lambda *)
let parser = parse_env []
     where rec parse_env env = abstract
       where rec abstract = function
       Var(name)          → Ref(index name env).
   |   Lambda(name,c)  → Abs(parse_env (name::env) c)
   |   Apply(c1,c2)        → App(abstract c1, abstract c2);;
```

On peut maintenant définir complètement l'interface entre syntaxe abstraite et syntaxe concrète par une grammaire et un formateur:

```
grammar lambda =
rule entry concrete = parse
         lambda e                          → parser(e)
and lambda = parse
         "("; lambda g; lambda d; ")"      → Apply(g,d)
       |  IDENT x                          → Var(x)
       |  " ["; binder b; "]"; lambda e    → list_it (curry Lambda) b e
       |   "("; lambda e; ")"              → e
and binder = parse
         IDENT x                           → x
       |  IDENT x; binder b                → x::b;;
```

Nous laissons l'écriture du formateur en exercice.

2.3 Principe de récurrence contextuelle

Les définitions récursives sur les λ-termes suivent toutes le même schéma : on appelle une fonction d'un entier n avec la valeur 0 sur une expression fermée; la fonction s'appelle récursivement sur les sous-expressions, l'entier n étant incrémenté à chaque abstraction.

A ce style de définition correspond un style de preuve par récurrence, qui est l'analogue de la récurrence structurelle traditionnelle des structures libres, mais tenant compte du fait que l'opérateur d'abstraction

lie une nouvelle variable.

Principe de récurrence contextuelle.
Soit P_n une propriété des lambda-termes, indicée par un entier naturel n, et vérifiant les conditions de fermeture suivantes.

(a) $P_n(M)$ & $P_n(N) \Rightarrow P_n(App(M,N))$

(b) $P_{n+1}(M) \Rightarrow P_n(Abs(M))$

(c) $0<m\leq n \Rightarrow P_n(Ref(m))$

Alors $P_n(M)$ est vrai pour tout M dans Λ_n, avec n quelconque.

2.4. La substitution

Voici l'algorithme qui recalcule les indices des variables libres d'un terme à travers k niveaux d'abstraction :

```
let lift k = lift_rec 1
where rec lift_rec n = function
      Ref(m)   → if m<n then Ref(m)          (* variable liée   *)
                 else Ref(m+k)               (* variable libre *)
   | Abs(e)    → Abs(lift_rec (n+1) e)
   | App(g,d)  → App(lift_rec n g, lift_rec n d);;
```

Voici maintenant l'algorithme de substitution. Si M est un lambda construit dans un environnement $x::\Gamma$ et N est un lambda construit dans l'environnement Γ, alors $M\{x \leftarrow N\}$ (notation concrète) ou $M\{N\}$ (notation abstraite) est défini comme le terme subst N M, avec subst défini comme suit. On vérifie que ce terme est bien construit dans l'environnement Γ.

```
let subst lam = subst_rec 1
where rec subst_rec n = function
      Ref(m)   → if m=n then (lift (n-1) lam)
                 if m<n then Ref(m)
                 else Ref(m-1)
   | Abs(e)    → Abs(subst_rec (n+1) e)
   | App(g,d)  → App(subst_rec n g, subst_rec n d);;
```

Exemple.
$[y](y\ x)\{x \leftarrow [z]z\}$ = subst (Abs(Ref(1))) (Abs(App(Ref(1),Ref(2)))
= Abs(App(Ref(1),Abs(Ref(1)))) = $[y](y\ \ [z]z)$.

3. Réduction, conversion

On définit maintenant la relation de réduction $\Rightarrow$ entre λ-termes, comme suit.

$$([x]M\ N) \quad \Rightarrow \quad M\{x \leftarrow N\} \qquad (\beta)$$

On dit que le terme $([x]M\ N)$ est un radical, qui se réduit en $M\{x \leftarrow N\}$. On étend la relation $\Rightarrow$ par congruence par rapport à la structure de λ-terme :

$$M \Rightarrow M' \Rightarrow []M \Rightarrow []M' \qquad (\xi)$$
$$M \Rightarrow M' \Rightarrow (M\ N) \Rightarrow (M'\ N)$$
$$M \Rightarrow M' \Rightarrow (N\ M) \Rightarrow (N\ M')$$

Historiquement, la fermeture réflexive-transitive $\Rightarrow^*$ de la relation $\Rightarrow$ s'appelle la β-réduction.

La β-réduction est la règle de calcul du langage algorithmique lambda. Elle est non-déterministe, dans la mesure où un terme possédant plusieurs occurrences de radicaux peut se calculer de manières différentes. Pourtant, le langage est intrinsèquement déterministe, dans la mesure où le résultat final d'un calcul est unique, comme le montrera le théorème du losange ci-dessous. Donnons tout d'abord un exemple de calcul.

Exemple.
On calcule, en soulignant les radicaux à chaque étape :

M = $(([x]\,[y]\,(x\ (y\ x))\ \ [u]\,(u\ u))\ \ [v]\,[w]\,v)$.
$\Rightarrow$ $([y]\,([u]\,(u\ u)\ \ (y\ \ [u]\,(u\ u)))\ \ [v]\,[w]\,v)$
$\Rightarrow$ $([u]\,(u\ u)\ \ ([v]\,[w]\,v\ \ [u]\,(u\ u)))$
$\Rightarrow$ $([u]\,(u\ u)\ \ [w]\,[u]\,(u\ u))$
$\Rightarrow$ $([w]\,[u]\,(u\ u)\ \ [w]\,[u]\,(u\ u))$
$\Rightarrow$ $[u]\,(u\ u) = \Delta$.

Δ est une forme irréductible, car ce terme ne possède pas de radical. On dit qu'un tel terme est en forme normale. On dit aussi que Δ est la forme

normale de M. Cette terminologie est justifiée par la prochaine section. Mais remarquons tout d'abord qu'un terme peut ne pas posséder de forme normale, car ses calculs peuvent ne pas terminer; ainsi :

$\Omega = (\Delta\ \Delta) \Rightarrow (\Delta\ \Delta) \Rightarrow (\Delta\ \Delta) \Rightarrow$ .... boucle!

Remarque. Dans la terminologie anglo-saxonne, on utilise redex, abréviation de "reducible expression", pour radical. Le théorème du losange que nous allons maintenant étudier s'appelle généralement "parallel moves lemma".

Quelques lambdas usuels.

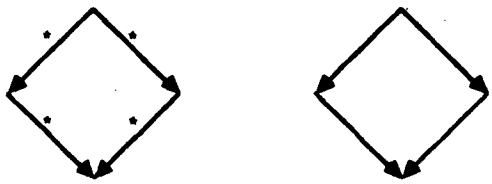
I	=	[x]x	
K	=	[x, y]x	
S	=	[x, y, z](x z (y z))	
A	=	[x, y](x y)	
B	=	[x,y,z](x (y z))	
C	=	[x, y, z](x z y)	
P	=	[x, y, z](z x y)	
T	=	[x, y]x	(= **K**)
F	=	[x, y]y	
0	=	[f, x]x	(= **F**)
1	=	[f, x](f x)	(= **A**)
n	=	[f, x](f (f ... (f x)))	(n fois f)
Y	=	[f]([x](f x x) [x](f x x))	
Θ	=	([x, y](y (x x y)) [x, y](y (x x y)))	
Δ	=	[x](x x)	
Ω	=	([x](x x) [x](x x))	(= (Δ Δ))

4. Le théorème du losange

4.1 Enoncé du théorème et corollaires

Théorème du losange.
Pour tous termes M, M_1 et M_2 tels que $M \Rightarrow^* M_1$ et $M \Rightarrow^* M_2$, il existe un terme N tel que $M_1 \Rightarrow^* N$ et $M_2 \Rightarrow^* N$.

Autrement dit, la relation $\Rightarrow$ est confluente, ou de manière équivalente la β-réduction $\Rightarrow^*$ est fortement confluente, avec les définitions de ces notions exprimées par les diagrammes ci-dessous :



confluence confluence forte

Dans de tels diagrammes, la partie supérieure, en traits pleins, est universellement quantifiée, alors que la partie inférieure, en pointillés, est existentiellement quantifiée.

Définition. On appelle β-conversion l'équivalence $\equiv$ engendrée par $\Rightarrow$.

Enonçons maintenant quelques conséquences importantes du théorème.

Corollaire 1 : Propriété de Church et Rosser. Pour tous termes M et M' convertibles : $M \equiv M'$, il existe un terme commun N résultat de calculs effectués à partir de M et M' respectivement : $M \Rightarrow^* N$ et $M' \Rightarrow^* N$.

Corollaire 2 : Unicité des formes normales. La forme normale de tout terme, si elle existe, est unique.

On appelle normalisable un terme possédant une forme normale, et fortement normalisable un terme tel que tout calcul issu de ce terme se termine. Par exemple, le terme ([w][x]x Ω) est normalisable, de forme normale I = [x]x, mais non fortement normalisable, à cause de son sous-terme non normalisable Ω.

4.2 Occurrences et sous-termes

Avant de faire la preuve du théorème du losange, il convient de développer les notions fondamentales permettant d'exprimer les transformations progressives d'une expression par calcul.

La première notion est celle d'occurrence. Jusqu'ici nous avons parlé de manière informelle d'occurrences de variables. De manière plus formalisée, les occurrences vont désigner l'emplacement des sous-expressions d'une expression. Une étape de calcul sera alors déterminée par l'occurrence à laquelle se trouve le radical en question.

Définition. Soit M une λ-expression. On définit l'ensemble de ses occurrences dom(M) par récurrence comme suit.

```
type sibling = A | B | C
and occurrence == sibling list
and domaine == occurrence list;;

let top = []                    (* l'occurrence principale de tout terme *)
and sons s = map (cons s);;  (* préfixage par le sibling s *)

let rec dom = function      (* dom : lambda → domaine *)
      Ref(_)      → [top]
   |  App(g,d)    → top :: ((sons A (dom g)) @ (sons B (dom d)))
   |  Abs(e)      → top :: (sons C (dom e));;
```

Exemple.
dom([x](x [y](x y))) = [[]; [C]; [C;A]; [C;B]; [C;B;C]; [C;B;C;A]; [C;B;C;B]].

Deux occurrences u et v sont dites cohérentes si elles peuvent

appartenir à un même terme. Dans le cas contraire on écrit u#v.

Les occurrences sont ordonnées naturellement par l'ordre préfixe. On dit que u est au dessus de v, et on écrit u<v si u est un préfixe (sous-liste initiale) de v. On dit que u et v sont disjointes, et on écrit u|v, si u et v sont des occurrences cohérentes non comparables. L'algorithme suivant donne tous les cas de comparaison :

```
let rec compare = function
      ([],[])       → "="
    | ([],_)        → "<"
    | (_,[])        → ">"
    | (C::u,C::v)   → compare(u,v)
    | (C::_,_)      → "#"
    | (_,C::_)      → "#"
    | (A::u,A::v)   → compare(u,v)
    | (B::u,B::v)   → compare(u,v)
    | _             → "|";;
```

Remarque. Un ensemble E fini non vide d'occurrences deux à deux cohérentes peut être ordonné en un domaine d'occurrences de terme si et seulement s'il vérifie les deux conditions suivantes :

a) $u @ [s] \in E \Rightarrow u \in E$ (fermé vers le haut)
b) $u @ [B] \in E \Leftrightarrow u @ [A] \in E$. (complet pour App)

On définit maintenant le sous-terme de M à l'occurrence $u \in dom(M)$, noté M/u, et calculé comme sub(M,u), avec l'algorithme sub ci-dessous :

```
let rec sub = function
    (e , [])          → e
  | (App(g,_) , A::u) → sub(g,u)
  | (App(_,d) , B::u) → sub(d,u)
  | (Abs(e) , C::u)   → sub(e,u)
  | _                 → Erreur "Occurrence hors du domaine";;
```

Remarque. Une occurrence est une représentation d'un terme

"filiforme" dénotant un chemin d'accès à un sous-terme.

On appelle radical de e toute occurrence d'un sous terme de e de la forme App(Abs(_),_). C'est un endroit immédiatement réductible par substitution. Formellement, on définit :

```
let radical u e = match sub(e,u) with
    App(Abs(_),_) → true
  | _             → false;;
```

Autrement dit : radical u e ⇔ u @ [A;C] ∈ dom(e).

Notation. On note R(M) l'ensemble des occurrences de radicaux dans le terme M : R(M) = {u∈ dom(M) | radical u M}.

```
let rec R = function
     Ref(_)          → []
   | Abs(e)          → sons C (R e)
   | App(Abs(e),d)   → top :: (sons A (sons C (R e))) @ (sons B (R d))
   | App(g,d)        → (sons A (R g)) @ (sons B (R d));;
```

Le remplacement du sous-terme de M à l'occurrence u par le terme N, que nous noterons M[u←N], se calcule par remplace N (M,u), avec remplace défini ci dessous :

```
let remplace N = remp 0
where rec remp n = function
   (_ , [])          → lift n N
 | (App(g,d) , A::u) → App(remp n (g,u), d)
 | (App(g,d) , B::u) → App(g, remp n (d,u))
 | (Abs(e) , C::u)   → Abs(remp (n+1) (e,u))
 | _                 → Erreur "Occurrence hors du domaine";;
```

On peut maintenant calculer formellement le terme N obtenu par une étape de réduction M ⇨ N réduisant le radical u de M, comme suit :

```
let réduit M u = match sub(M,u) with
```

```
    App(Abs(body),arg)   → remplace (subst arg body) (M,u)
  | _                    → Erreur "Réduction d'un non-radical";;
```

4.3 Résidus

On définit maintenant une notion fondamentale : celle de résidu. Intuitivement, les sous-termes d'un terme progressivement calculé proviennent de morceaux recombinés du terme d'origine. On trace cet héritage à l'aide de la notion de résidu d'occurrence.

Tout d'abord, considérons un radical R = ([x]M N). L'étape de calcul réduisant R calcule : R ⇨ M', avec M' = M{x ← N}. Les occurrences de M' proviennent de deux sources : les occurrences de N qui remplacent d'éventuelles occurrences de x dans M, et les autres occurrences de M. Calculons tout d'abord l'ensemble X des occurrences de x dans M à l'aide de X = locaux M, avec locaux défini récursivement par :

```
let locaux = loc 1
where rec loc n = function
    Ref(m)    → if m=n then [top] else []
  | App(e,e') → (map (cons A) (loc n e)) @ (map (cons B) (loc n e'))
  | Abs(e)    → map (cons C) (loc (n+1) e);;
```

Maintenant soit T un terme possédant à l'occurrence u le radical T/u = R. On a T ⇨ T', avec T' = T[u←M']. A chaque occurrence v de T on fait correspondre un ensemble d'occurrences V de T', dites résidus de v selon u, comme suit. Tout d'abord, si v<u ou v|u, V={v}. Ensuite, les occurrences du radical proprement dit, u et u@[A], n'ont pas de résidu. Une occurrence u@(A::C::w) dans M a pour résidu l'occurrence u@w, sauf pour les occurrences de la variable substituée x, qui n'ont pas de résidu. Enfin, une occurrence u@(B::w) dans N a pour résidus tous les u@x@w, pour x∈X quelconque.

Formellement, les résidus de v selon u dans le terme M se calculent par résidus M u v, avec l'algorithme résidus ci dessous :

```
let résidus M u = let X = locaux(sub(M,u@[A;C])) in function v → res(u,v)
```

```
where rec res = function
    ([],[])        → []
  | ([],[A])       → []
  | ([],A::C::w)   → if mem w X then [] else [u@w]
  | ([],B::w)      → let wrap x = u@x@w in map wrap X
  | ([],_)         → Erreur "Occurrence hors domaine"
  | (_,[])         → [v]
  | (s::w,s'::w')  → if s=s' then res(w,w')
                     if s=C or s'=C then Erreur "Occurrence hors domaine"
                     else (* (s=A & s'=B) or (s=B & s'=A) *)  [v];;
```

Toute occurrence dans le terme réduit est le résidu d'une occurrence unique. On vérifie que les résidus d'une occurrence de radical sont des occurrences de radicaux, qui intuitivement partagent le même "grain de calcul". On appelle radical résidu une occurrence de radical résidu d'une occurrence de radical. Par contre, certains radicaux du terme réduit proviennent d'un nouvel assemblage App(Abs(e),e'). On dit que ces radicaux sont créés par le calcul.

Remarque. Le radical R = ([x]M N) peut par sa réduction créer un radical de 3 manières différentes :

- vers le bas : à toutes les occurrences dans M de la forme (x P), si N est une abstraction.
- vers le haut : si R apparaît en partie gauche d'une application dans le terme à réduire : (R P), avec :
  - soit M une abstraction
  - soit M=x, et N une abstraction.

Remarquez que les deux premiers cas peuvent se cumuler.

Les résidus d'un ensemble d'occurrences V sont l'union des résidus de chacune des occurrences de V. On les calcule par (Résidus M u V), avec :

```
let Résidus M u = set_extension (résidus M u);;
(* where set_extension f = it_list (fun set x → union set (f x))  *)
```

Finalement, les résidus s'itèrent le long des séquences de réduction.

Ainsi, si on définit une dérivation $M = M_1 \Rightarrow M_2 \Rightarrow \ldots M_n \Rightarrow N$ de longueur n par une suite d'occurrences de radicaux : $D = [u_1 ; \ldots ; u_n]$, alors on calcule les résidus de l'ensemble d'occurrences V de M comme l'ensemble d'occurrences de N :

W = Résidus $M_n\ u_n$ (Résidus $M_{n-1}\ u_{n-1}$ ... (Résidus $M_1\ u_1\ V$) ...)).

Voici l'algorithme trace qui, étant donné M, V et D, calcule N et W :

```
let rec trace (M,V) = function
    []      → (M,V)
  | u::D    → let N = réduit M u and W = Résidus M u V in trace (N,W) D;;
```

Exemple. Voici un exemple de dérivation D = [[] ; []] à partir d'un terme L:

$$L = \quad ([u](u\ u)\quad \underbrace{[v]([x]v\ \ y)}_{0})$$

$$\Rightarrow \quad (\underbrace{[v]([x]v\ \ y)}_{1}\quad \underbrace{[v]([x]v\ \ y)}_{2}) = R$$

$$\Rightarrow \quad \underbrace{([x][v]\underbrace{([x]v\ \ y)}_{4}\ y)}_{3}$$

Figure 1

Après la première étape le radical marqué 0 a pour résidus les deux radicaux marqués 1 et 2. Après la 2ème étape le radical 1 (resp. 2) a pour unique résidu le radical 3 (resp. 4). Les résidus de 0 par D sont donc 3 et 4. Formellement, on calcule : trace (L,[[B; C]]) D = (L',[[] ; [A; C; C]]), avec L'= ([x][v]([x]v y) y). On remarque que le radical R ci-dessus n'est pas résidu d'un radical de L, mais est créé par la dérivation D.

Remarque. Les résidus donnent une idée de "partage" : les sous-termes résidus d'un sous-terme donné peuvent être "partagés" avec ce sous-terme, plutôt que copiés. Toutefois, ce partage ne rend pas compte d'un partage plus complexe de "contextes", ou portions de termes non

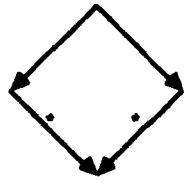
nécessairement sous-termes. De plus, la représentation de de Bruijn ne rend pas compte de ce partage, dans la mesure où les variables globales des résidus doivent être éventuellement recalées (par l'opérateur lift).

4.4 Réductions parallèles.

On cherche à prouver le lemme du losange, c'est à dire à montrer que la relation ⇨ est confluente. Une idée naïve consiste à vouloir prouver que ⇨ est fortement confluente. Ceci échoue, à cause du problème de duplication :

Avec M = (Δ (I I)) , on a M ⇨ N1 = (Δ I) et M ⇨ N2 = ((I I) (I I)), mais la seule manière de fermer le diagramme est N1 ⇨ (I I), alors que N2 ne conduit à (I I) qu'après 2 étapes de réduction.

Par contre, il est vrai que la relation ⇨ est localement confluente. Cette propriété est définie graphiquement par :



confluence locale

Toutefois, ceci ne suffit pas pour prouver directement la confluence, car en général les calculs peuvent ne pas terminer, et on ne peut donc pas employer de récurrence permettant d'utiliser l'idée d'un progrès effectué par la fermeture du diagramme de confluence locale. Remarquez par exemple que le graphe ci-dessous (paradoxe d'Escher-Newman) détermine une relation acyclique localement confluente, mais non confluente :

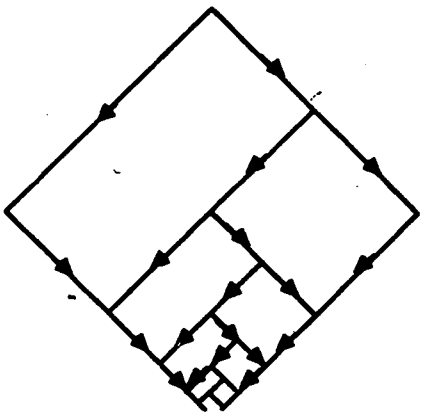


Figure 2

Il y a en gros deux manières pour sortir de cette difficulté. La première consiste à persévérer dans l'idée ci-dessus, en remarquant que la partie inférieure du diagramme de confluence locale n'est pas arbitraire, mais consiste à réduire des résidus des radicaux réduits dans sa partie supérieure. Le progrès peut alors s'exprimer sous la forme du théorème de finitude des développements que nous verrons plus loin. La deuxième manière, due à Tait, consiste à prouver la confluence forte pour une relation $\twoheadrightarrow$ intermédiaire entre $\Rightarrow$ et $\Rightarrow^*$. Nous allons maintenant poursuivre cette idée.

Définition.

La réduction parallèle $\twoheadrightarrow$ est définie comme la plus petite relation entre λ-termes vérifiant :

$$M \twoheadrightarrow M' \,\&\, N \twoheadrightarrow N' \Rightarrow ([x]M\ N) \twoheadrightarrow M'\{x \leftarrow N'\}$$
$$M \twoheadrightarrow M' \Rightarrow []M \twoheadrightarrow []M'$$
$$M \twoheadrightarrow M' \,\&\, N \twoheadrightarrow N' \Rightarrow (M\ N) \twoheadrightarrow (M'\ N')$$
$$x \twoheadrightarrow x$$

Remarque. La relation $\twoheadrightarrow$ correspond à une idée de réduction en parallèle de radicaux dont les occurrences peuvent ne pas être disjointes.

En effet, la notion de réduction en parallèle d'occurrences deux à deux disjointes ne suffirait pas, car il n'est pas vrai que les résidus de radicaux disjoints sont des radicaux disjoints. La figure 1 ci-dessus en donne un contre-exemple.

Il est clair que $\Rightarrow \subseteq \twoheadrightarrow$ et $\twoheadrightarrow \subseteq \Rightarrow^*$, et donc que $\twoheadrightarrow^* = \Rightarrow^*$. La première assertion est évidente. La deuxième s'obtient en constatant que la réduction parallèle d'un ensemble de radicaux peut être séquentialisée en une séquence de réductions simples, pourvu qu'on effectue les réductions de l'intérieur vers l'extérieur. En effet, si u et v sont des occurrences de M, avec v<u ou vlu, on a : résidus M u v = [v].

Nous allons maintenant généraliser la notion d'occurrence à celle d'ensemble d'occurrences. Toutefois, nous ne représenterons pas un ensemble d'occurrences par un domaine (liste d'occurrences) comme précédemment défini, car cette représentation n'a pas la bonne structure pour partager les préfixes communs. On choisit plutôt de généraliser la notion d'occurrence vue comme "terme filiforme" en un terme dont certains des symboles sont marqués. Comme nous nous intéresserons à ces ensembles d'occurrences essentiellement pour marquer certains radicaux, on ne marque que les opérateurs d'application.

4.5 Termes marqués

Voici tout d'abord la structure de λ-terme avec applications marquées.

```
type marqué =
      MRef of num                          (* variable *)
    | MAbs of marqué                       (* abstraction *)
    | MApp of bool * marqué * marqué;;     (* application marquée *)
```

Voici comment traduire un lambda en un terme marqué, étant donné un domaine U :

```
let mark U = markrec []
where rec markrec u = function
      Ref(m)   → MRef(m)
```

```
  | Abs(e)     → MAbs(markrec (C::u) e)
  | App(g,d)   → MApp(mem u U, markrec (A::u) g, markrec (B::u) d);;
```

Inversement, unmark enlève les marques :

```
let rec unmark = function
      MRef(m)      → Ref(m)
  |   MAbs(e)      → Abs(unmark e)
  |   MApp(_,g,d)  → App(unmark g, unmark d);;
```

On suppose partout dans la suite que les marques ne valent true que sur les occurrences de radicaux : M = mark U L ⇒ U ⊆ R(L). On dit qu'un terme marqué M est un marquage du lambda L lorsque L = unmark M. On écrit aussi M ↑ L. On dit que les termes marqués M et M' sont compatibles s'ils sont les marquages d'un même lambda, et on écrit alors M ↑ M'. On écrit de même U ↑ L lorsque U est un ensemble d'occurrences de radicaux du terme L.

Les termes marqués compatibles ont naturellement une structure d'algèbre Booléenne héritée de la structure de l'ensemble des marques. On écrit par exemple M ⊆ M', M ∪ M' etc... avec la signification évidente. Par exemple, M ⊆ M' ssi M = mark U L, M' = mark U' L, et U ⊆ U'. Cette notation s'étend aussi aux termes, en identifiant L à mark [] L.

Donnons maintenant l'algorithme dérivé qui réduit un terme marqué M suivant toutes les marques d'un terme marqué compatible N. On suppose ci-dessous que l'algorithme subst a été étendu aux termes marqués en l'algorithme msubst de la manière naturelle, c'est à dire préservant la valeur des marques.

Définition : Dérivation

On suppose que M et N sont des termes marqués compatibles. Le terme marqué P = dérivé(M,N) représente l'effet de réduire M en tous les radicaux marqués par N, les réductions ayant lieu de l'intérieur vers l'extérieur.

```
let rec dérivé = function
      (MRef(k),_)          → MRef(k)
    | (MAbs(e),MAbs(m))→ MAbs(dérivé(e,m))
    | (MApp(_,MAbs(e),e'),MApp(true,MAbs(m),m'))
                           → let body=dérivé(e,m) and arg=dérivé(e',m')
                             in msubst arg body
    | (MApp(b,e,e'),MApp(false,m,m'))
                           → MApp(b,dérivé(e,m),dérivé(e',m'))
    | _                    → Erreur "Termes non cohérents";;
```

Par la suite, on notera M\N pour dérivé(M,N) :

```
#infix "\\";;
let x\y = dérivé(x,y);;
```

Proposition. L ↠ L' si et seulement si il existe un marquage N de L tel que L' = unmark(M\N), avec M = mark [] L.

Nous laissons au lecteur la preuve de la proposition. En fait l'algorithme dérivé nous donne beaucoup plus d'information que la relation ↠, car les marquages permettent de tracer les résidus de radicaux. Remarquons par exemple que si U ↑ L, alors plus généralement on a L ↠ L'=unmark(P), pour P= M\N, avec M = mark U L. Mais de plus P est la marque des résidus de U par le calcul L ↠ L'. Autrement dit, l'algorithme dérivé joue simultanément les rôles des algorithmes réduit et résidu ci-dessus. Remarquez par exemple que si V ↑ L, et si D_V est une dérivation représentant V de l'intérieur vers l'extérieur, alors pour tout U ↑ L, avec M = mark U L et N = mark V L, on a M\N = mark W L', avec (L',W) = trace (L,U) D_V.

On étend la notation aux λ-termes L et aux ensembles de radicaux U ⊆ R(L) en écrivant L\U pour (mark [] L)\(mark U L). De même on écrit M\U pour M\(mark U (unmark M)). De plus, lorsque le terme L est sous-entendu par le contexte, on écrit V\U pour (mark V L)\(mark U L).

Remarque. A tout terme marqué M correspond de manière unique U et L tels que M = mark U L. Inversement, à L correspond l'ensemble (fini) de tous ses marquages. Mais à U correspond un ensemble infini de termes marqués M compatibles.

4.6. Démonstration du théorème du losange

On montre d'abord un résultat technique, expliquant que la substitution et la dérivation distribuent :

Lemme de substitution.
Pour tous termes marqués M,M',N,N', tels que M ↑ M' et N ↑ N', on a :
msubst (N\N') (M\M') = (msubst N M)\(msubst N' M').

Démonstration. Nous laissons au lecteur la preuve de cette propriété, par récurrence sur M.

Le résultat fondamental s'exprime maintenant comme suit.

Lemme de monotonie des résidus. Soit L un λ-terme, U ↑ L, W ↑ L avec U ⊆ W. Pour tout M ↑ L, on a (M\U)\(W\U) = M\W.

Démonstration.
Récurrence sur M.
Le seul cas intéressant est lorsque M = MApp(b,MAbs(M1), M2). On pose M' = M\U et W' = W\U. Avec U1 = {u | A::C::u∈U}, U2 = {u | B::u∈U}, et notations similaires pour W, on a M1\U1 = M'1, M2\U2= M'2, W1\U1 = W'1 et W2\U2 = W'2 tels que, par récurrence : M'1\W'1 = M1\W1 et M'2\W'2 = M2\W2. Maintenant il y a 2 cas:
a) top ∈ U. Alors top ∈ W aussi, et donc M\W = msubst (M2\W2) (M1\W1), et M' = msubst M'2 M'1. De la même manière W' est un marquage de L\U de la forme msubst W'2 W'1 (avec un abus de notation évident) et donc par le lemme de substitution M'\W' = msubst (M'2\W'2) (M'1\W'1) = M\W.
b) top ∉ U. Alors M' = MApp(b, MAbs(M'1), M'2). Soit W'0 = {A::C::w | w∈W'1} ∪ {B::w | w∈W'2}. Il y a de nouveau 2 cas :
b1) top ∈ W. Alors W' = W'0 ∪ {top}, et de nouveau :
M\W = msubst (M2\W2) (M1\W1) = msubst (M'2\W'2) (M'1\W'1) = M'\W'.
b2) top ∉ W. Alors W' = W'0, et de même :
M\W = MApp(b, MAbs(M1\W1), M2\W2) = M'\W'.

Corollaire : Lemme du cube.
Soit L un λ-terme, U et V des sous-ensembles de R(L). Pour tout M ↑ L, on a (M\U)\(V\U) = (M\V)\(U\V).

Démonstration. On applique le lemme ci-dessus, en prenant W = U ∪ V = V ∪ U.

Corollaire : Théorème du losange.

Démonstration. En particulier le lemme du cube donne la confluence forte de ↠, et donc la confluence de ⇨.

En fait on obtient beaucoup plus. En particulier, on aurait pu faire converger le diagramme de forte confluence plus simplement sur L\R(L). Mais cette preuve n'aurait pas exprimé que ce diagramme peut être fermé d'une manière minimale. Nous allons voir maintenant comment le lemme du cube peut s'exprimer comme propriété catégorique.

4.7. Structure des dérivations

Soit M et N des termes marqués. On définit récursivement la notion de dérivation parallèle D de M vers N, notée D : M ↠* N, comme une paire (M,S), où S est une suite d'ensembles d'occurrences, vérifiant :
a) (M,[]) : M ↠* M, pour tout M.
b) si (N,S) : N ↠* N' et U ↑ M, avec M\U = N, alors (M, U::S) : M ↠* N'.

Si (M,S) est une dérivation parallèle de M vers N, alors N est unique. On écrit N = M\S. Si M=mark U L et N=mark V L, on écrit de même V = U\S.

Remarque. On commet ici un abus de notation : la relation ↠ était définie plus haut pour les λ-termes, et ici la relation ↠ est définie entre termes marqués. Cet abus est justifié en remarquant que M ↠ N ssi M = mark U L, L ↠ L' par une dérivation D comme définie précédemment, et N = mark V L' avec (L',V) = trace (L,U) D.

Réciproquement, on définit une dérivation parallèle de L vers L' comme une paire (L,S), avec (M,S) : M ⇛ * M' par une dérivation parallèle comme défini ci-dessus, avec M = mark [] L et L' = unmark M'. On dit que deux séquences de calcul S et S' sont équivalentes en L ssi M\S = M\S' pour tout marquage M de L. On écrit alors (L,S) ≡ (L,S').

Remarque. L'équivalence entre dérivations ≡, appelée équivalence de permutations, distingue des dérivations aboutissant au même terme. Par exemple, les deux dérivations qui réduisent (I (I M)) en (I M) ne sont pas équivalentes (pourquoi?). Par contre, toutes les dérivations issues d'un terme normalisable et aboutissant à sa forme normale sont équivalentes.

On dit que l'étape de dérivation est vide lorsque l'ensemble de radicaux réduits à cette étape est vide.

On appelle développement de M une dérivation parallèle issue de M telle qu'à chaque étape b) ci-dessus on ait U ⊆ M. Autrement dit, un développement est une dérivation parallèle qui ne fait pas de créations de radicaux, les seuls radicaux contractés étant des résidus des radicaux marqués initialement dans M. Le développement D est dit complet s'il aboutit à un terme n'ayant plus de radicaux marqués.

Nous allons maintenant nous intéresser à la structure des dérivations. La première opération est la concaténation.
Soit D : M ⇛ * N par la suite S et D' : N ⇛ * N' par la suite S', on définit la concaténation D @ D' : M ⇛ * N' comme donnée par la suite S @ S'.

On étend maintenant la notion de résidu à une opération entre dérivations. Tout d'abord, soit S une suite de calcul et U un ensemble d'occurrences. On définit S\U par récurrence sur la longueur de S comme suit.
a) [] \ U = []
b) (V::S) \ U = (V\U) :: (S\(U\V)).

Cette dernière clause suppose bien sûr U ↑ V. On laisse au lecteur le soin de vérifier que si D = (M,S) : M ⇛ * N avec M ↑ U, alors D\U = (M\U, S\U) définit une dérivation parallèle M\U ⇛ * N\U.

Soient maintenant D = (M, S) et D' = (M', S') deux dérivations, avec M ↑ M'. On définit D'\D comme la dérivation (M\S, S'\S), avec l'opération \ définie par récurrence sur la longueur de S' par :

a) [] \ S = []

b) (U::S') \ S = (U\S) :: (S' \(S \U)).

Nous laissons au lecteur le soin de vérifier que cette définition est bien fondée.

Soit D : M ↠* N et D' : M ↠* N' deux dérivations issues du même terme marqué M. On dit que D calcule moins que D', et on écrit D ≤ D', si et seulement si on peut prolonger D en D', c'est à dire s'il existe une dérivation N ↠* N'.

L'ensemble des dérivations issues d'un même terme marqué M forme un sup demi-treillis, avec l'ordre préfixe : (M, S) ≤ (M, S') ssi S ≤ S'. On laisse au lecteur le soin de vérifier que l'opération ∪ est la borne supérieure, avec ∪ définie par D ∪ D' = D @ (D'\D).

Il est clair que la notion de dérivation issue d'un terme marqué n'est qu'un instrument technique pour étudier les dérivations à partir de λ-termes. Pour celà, on remarque que toutes les notions définies ci-dessus s'étendent aux dérivations sur les λ-termes (non marqués), et que les deux opérations @ et \ préservent l'équivalence ≡ ci-dessus.

Définition. D ≤ D' si et seulement si on peut prolonger D en une dérivation équivalente à D', i.e. s'il existe D'' telle que D @ D'' ≡ D'.

Soit Der(M) l'ensemble des dérivations issues d'un lambda M.

Théorème de Lévy. (Der(M)/≡ , ≤) a la structure d'un sup-demi-treillis.

Remarque : ce résultat justifie la généralité de la démonstration que nous avons donnée du théorème du losange, en passant par le lemme du cube. La structure des dérivations du λ-calcul est investigée très

complètement dans la thèse de J.J. Lévy.

Exercices.

1. Donner une version catégorique du théorème exprimant que la catégorie ayant pour objets les λ-termes et pour flèches les dérivations quotientées par permutation admet des sommes amalgamées.

2. (Plotkin) Montrer qu'il n'existe pas de λ-terme se dérivant à la fois sur ([x]((y x) (y z)) z) et sur (([x](x x) (y z)). En déduire que la propriété duale d'inf-demi-treillis n'est pas vraie.

3. Montrer que la structure de sup-demi-treillis des dérivations n'induit pas une structure de sup-demi-treillis de l'ordre de β-réduction sur les termes issus d'un même terme.

Remarque. Les termes marqués peuvent être enrichis d'autres décorations. Par exemple, Barendregt dans son livre considère des termes dont les opérateurs d'abstraction sont marqués par des entiers naturels. J.J. Lévy dans sa thèse considère des termes marqués par des étiquettes, suites mémorisant l'histoire du calcul menant à ce terme. Dans la prochaine section nous allons marquer les variables par des entiers positifs appelés poids.

5. Le théorème des développements finis

Théorème des développements finis. Tout développement est fini, dans le sens où il est ultimement vide.

Autrement dit, toute suite de calculs qui ne réduit à chaque étape que des résidus de radicaux du terme d'origine, termine nécessairement.

La méthode de preuve utilise des termes marqués dont les variables dont pondérées par des entiers strictement positifs :

```
type pondéré =
        PRef of num
    |   PAbs of num * pondéré                    (* 1er arg = poids *)
    |   PApp of bool * pondéré * pondéré;;
```

On étend le poids à un terme pondéré par sommation des poids de ses variables :

```
(* plibres est un environnement de poids *)
let rec poids_env plibres = function
        PRef(n)      → nth plibres n
    |   PAbs(p,e)    → poids_env (p::plibres) e
    |   PApp(_,g,d)  → (poids_env plibres g) + (poids_env plibres d);;

let poids = poids_env [];;   (* poids d'un terme fermé *)
```

Comme au paragraphe précédent nous supposerons que les marques ne valent true que sur des occurrences de radicaux. On suppose qu'on étend de la manière naturelle aux termes pondérés les notions de substitution (algorithme psubst), et donc de calcul, résidu, etc...

On laisse au lecteur la preuve du lemme suivant, par récurrence contextuelle.

Lemme de substitution. Soit plibres un environnement de poids de longueur $n \geq 0$, soit $M \in \Lambda_{n+1}$, $N \in \Lambda_n$, et Q = psubst N M. Alors pour tout entier p≥poids_env plibres N, on a poids_env (p::plibres) M ≥ poids_env plibres Q.

Corollaire. Sous les mêmes conditions, avec P=PApp(b,PAbs(p,M),N), on a poids_env plibres P > poids_env plibres Q.

Il suffit de remarquer pour le corollaire que (poids_env plibres N)>0. On voit donc que la réduction d'un terme bien pondéré en fait décroître le poids, ce qui justifie la définition suivante.

Un terme pondéré est dit <u>décroissant</u> si et seulement si pour toute occurrence de radical marqué PApp(true,PAbs(p,M),N), le poids p est supérieur ou égal au poids de N. Plus précisément, on vérifie :

```
let rec decr_env plibres = function
      PRef(_)        → true
   |  PAbs(p,M)      → decr_env (p::plibres) M
   |  PApp(b,M,N)    → decr_env plibres M & decr_env plibres N &
                       (not b or let (PAbs(p,_)) = M
                       in p≥(poids_env plibres N));;
let décroissant = decr_env [];;
```

Proposition. A tout terme marqué correspond un terme pondéré décroissant de même structure.

Démonstration. On associe au terme marqué M le terme pondéré pond(M) calculé comme suit :

```
let rec pondère plibres = pond_with 1
where rec pond_with p = function
     MRef(n)        → (PRef(n), nth plibres n)
   | MAbs(M)        → let (M',p') = pondère (p::plibres) M
                      in (PAbs(p,M'),p')
   | MApp(b,M,N)    → let (N',p') = pond_with 1 N in
                      let (M',p'') = pond_with (if b then p' else 1) M
                      in (PApp(b,M',N'), p'+p'');;

let pond x = fst(pondère [] x);;
```

On montre que pond(M) est décroissant par récurrence contextuelle sur M. En effet, on vérifie : pondère plibres M = (M',poids_env plibres M').

On laisse au lecteur le soin de vérifier le lemme technique suivant, par récurrence sur M, et en utilisant le lemme de substitution ci-dessus.

Lemme de préservation de la décroissance.

Soit plibres un environnement de poids de longueur $n \geq 0$, soit $M \in \Lambda_{n+1}$, $N \in \Lambda_n$, p≥poids_env plibres N, tels que decr_env (p::plibres) M et decr_env plibres N. On a decr_env plibres (psubst N M).

Lemme de préservation des pondérations décroissantes.

Soit M un terme marqué pondéré décroissant, U un ensemble de radicaux marqués de M : $U \subseteq M$, et $N = M \backslash U$. N est un terme marqué pondéré décroissant, tel que poids(N)≤poids(M), l'inégalité étant stricte si U est non vide.

Démonstration.

L'énoncé concerne un terme fermé. Plus généralement, pour $M \in \Lambda_n$, et plibres un environnement de poids quelconque de longueur $n \geq 0$, avec decr_env plibres M, on montre poids_env plibres N ≤ poids_env plibres M, et decr_env plibres N, par récurrence sur $M \backslash U$.

- Cas M=(M1 M2) ↠ N=(N1 N2), top∉U, avec M1 ↠ N1 et M2 ↠ N2. Par hypothèse de récurrence on a poids_env plibres N1 ≤ poids_env plibres M1, poids_env plibres N2 ≤ poids_env plibres M2, et donc poids_env plibres N ≤ poids_env plibres M. De même on a decr_env plibres M1, et decr_env plibres N. Si top∈N (N marqué), c'est que N1=[x:p]Q1, et alors forcément M1=[x:p]P1, avec top∈M. Par hypothèse decr_env plibres M, on doit avoir p ≥ poids_env plibres M2 ≥ poids_env plibres N2 par hypothèse de récurrence, et on a donc decr_env plibres N.

- Cas M=([x:p]M1 M2) ↠ N=psubst N2 N1, top∉U, avec M1 ↠ N1 et M2 ↠ N2. Par hypothèse de récurrence on a poids_env plibres N1 ≤ poids_env plibres M1, poids_env plibres N2 ≤ poids_env plibres M2, et comme $U \subseteq M$ on a top∉M, d'où p ≥ poids_env plibres M2 par hypothèse decr_env plibres M.

On a également decr_env (p::plibres) M1 et decr_env plibres M2. Par récurrence, on obtient donc aussi decr_env (p::plibres) N1 et decr_env plibres N2. Par le lemme de préservation de la décroissance on en déduit decr_env plibres N. Finalement, on a poids_env plibres M ≥ poids_env plibres ([x:p]N1 N2) > poids_env plibres N par le corollaire du lemme de substitution.

- les autres cas ne présentent pas de difficulté.

Remarque.

En général les radicaux créés ne peuvent être marqués sans violer la condition de décroissance. Considérez par exemple :

$$[x_2]([y_2](y\ x)\ [z_1]z) \Rightarrow [x_2]([z_1]z\ x).$$

Démonstration du théorème.

Soit M un terme marqué. Tout développement de M sans étape vide est de longueur inférieure à poids(pond(M)).

Remarque.

Le théorème a d'abord été prouvé par Curry pour le λ-I-calcul, une restriction qui exige que toutes les variables liées apparaissent au moins une fois. Le théorème n'a été prouvé dans toute sa généralité qu'en 1965 par Schroer. La démonstration ci-dessus est inspirée d'une preuve due à Hyland et Barendregt. Ici nous donnons un poids uniforme à toutes les occurrences d'une variable liée, ce qui simplifie légèrement la preuve, et permet d'assimiler la pondération d'une variable à un type (dans Barendregt, pond est obtenu en pondérant les occurrences de variables de la droite vers la gauche par des puissances de 2 successives).

6. Le théorème de standardisation

Le théorème du losange nous a montré que la notion de calcul était déterministe. En particulier, tout terme M normalisable admet une forme normale Normal(M) unique, et il existe une dérivation $M \Rightarrow^* Normal(M)$. Mais ceci ne nous suffit pas pour construire un interpréteur du λ-calcul. On aimerait que cet interpréteur puisse être défini comme une fonction récursive ϕ associant à tout terme M un ensemble $\phi(M)$ de ses radicaux, avec $\phi(M) = \emptyset$ si et seulement si M est en forme normale. L'interpréteur sera dit correct s'il mène à la forme normale de tout terme normalisable. C'est à dire, pour tout terme M normalisable il existe n, M_1, ... ,M_n, tels que $M=M_1$, M_n=Normal(M), et pour tout i, avec $1 \le i < n$, on a : $M_{i+1}=M_i \setminus \phi(M_i)$.

Par exemple, si on choisit pour $\phi(M)$ l'ensemble des radicaux les plus internes de M (stratégie dite d'appel par valeur), on obtient un interpréteur qui n'est pas correct, comme le montre l'exemple ([x]I Ω). Si on choisit pour $\phi(M)$ l'ensemble de tous les radicaux de M (stratégie dite complète ou de Gross-Knuth), on obtient un interpréteur correct (pourquoi?).

Nous allons maintenant voir qu'il existe des stratégies correctes qui calculent de l'extérieur vers l'intérieur, tout en ne réduisant qu'un radical à chaque étape.

Définition.
Soit u et v deux occurrences. On dit que u est à gauche de v, et on écrit $u \angle v$, si et seulement si soit $u<v$, soit il existe w, u' et v' tels que u=w@(A::u') et v=w@(B::v'). La relation $\angle$ est un ordre total strict.

Lemme de préservation des occurrences à gauche.
Soit u et v des occurrences de radicaux dans un terme quelconque, avec $u \angle v$. On a :
1. $u \setminus v = \{u\}$.

2. $u \angle w$ pour toute occurrence w d'un radical créé par v.

3. $u \angle w$ implique $u \angle w'$ pour tout w' dans w\v.

Mêmes résultats avec V *ensemble* d'occurrences de radicaux, tel que $u \angle v$ pour tout v dans V.

Nous laissons la preuve de ce lemme, par cas sur les positions respectives des occurrences, au lecteur.

Pour tout terme M réductible, on définit gauche(M) comme le radical le plus à gauche de M. On peut le calculer par l'algorithme :

```
exception NormalForm;;
let gauche = searchleft top
  where rec searchleft u = function
     Ref(_)          → raise NormalForm
   | Abs(e)          → searchleft (C::u) e
   | App(Abs(_),_)   → rev(u)
   | App(g,d)        → try searchleft (A::u) g
                       with NormalForm → searchleft (B::u) d;;
```

Définition.

Considérons une dérivation réduisant un radical à chaque étape : (M,S), avec $S=[u_1; \ldots ;u_n]$. Soit $S_{ij} = [u_i; \ldots ;u_j]$, $M_j=M\backslash S_{1i}$. La dérivation est dite standard si et seulement si pour tous $1 \le i < j \le n$, si $u_j \in v\backslash S_{i,j-1}$, avec $v \in R(M_{i-1})$, alors $u_i \angle v$.

C'est à dire, pour toute étape de réduction i, il n'existe pas de radical v à gauche de u_i dans M_{i-1} dont l'un des résidus par la dérivation considérée soit contracté à une étape ultérieure j.

Exemples.

$([x](I\ x)\ (I\ y)) \Rightarrow ([x]x\ (I\ y)) \Rightarrow ([x]x\ y)$ est standard

$([x](I\ x)\ (I\ y)) \Rightarrow ([x]x\ (I\ y)) \Rightarrow (I\ y) \Rightarrow y$ ne l'est pas

Remarques.

1. Si D@D' est standard, D et D' le sont aussi. La réciproque n'est pas vraie en général.

2. Il n'y a pas d'extension évidente de la notion de dérivation standard aux dérivations parallèles. En effet, on a vu qu'une réduction en parallèle de n radicaux peut se séquentialiser en n réductions élémentaires, mais de l'intérieur vers l'extérieur, alors qu'une dérivation standard doit procéder de l'extérieur vers l'intérieur. Il est facile de montrer qu'une étape de réduction parallèle U peut se séquentialiser en réductions élémentaires procédant de l'extérieur vers l'intérieur : on réduit le radical u le plus à gauche de U, et on itère sur U\u; la construction s'arrête par le théorème des développements finis. Plus généralement :

Théorème de standardisation. Toute dérivation est équivalente à une dérivation standard unique.

Avant de montrer ce théorème, on définit récursivement une dérivation st(D) associée à une dérivation parallèle D.

Définition : R(D). Soit D=(M,S) une dérivation parallèle de longueur n issue d'un terme M, avec $S=[U_1; \ldots ;U_n]$. On définit R(D) comme l'ensemble des occurrences de radicaux de M dont au moins un résidu est réduit dans D:

$$R(D) = \{v \in R(M) \mid \exists k \leq n \ u \backslash S_{1,k-1} \cap U_k \neq \emptyset\}.$$

Définition : st(D). Soit D=(M,S) une dérivation parallèle de longueur n issue d'un terme M. On définit récursivement une dérivation st(D), qui réduit un radical à chaque étape. Soit $S=[U_1; \ldots ;U_n]$. Il y a deux cas.

- D est vide, c.a.d. $U_i = \emptyset$ pour tout i. Alors st(D) est la dérivation vide.

- sinon, soit u l'occurrence la plus à gauche de R(D). Par le lemme ci-dessus, on obtient que u est une occurrence de radical dans M, qui est préservée par la dérivation D, jusqu'à être réduite, disons à l'étape k :

$$u \in R(M),\ u \backslash S_{1,k-1} = \{u\},\ u \in U_k.$$

Soit D'= st(D\u). On prend pour st(D) la dérivation issue de M qui réduit u, et se poursuit par D'. C'est-à-dire, avec D'=(M',S') : st(D)=(M,u::S').

La construction est illustrée par la figure 3 ci dessous.

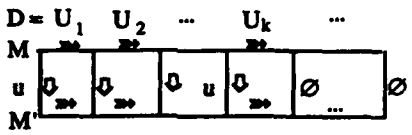


Figure 3

Le seul problème de cette définition est de montrer qu'elle est bien fondée, c'est à dire qu'on ne peut pas boucler indéfiniment dans le 2ème cas. Mais il suffit de remarquer qu'on aurait alors une étape k qui serait sélectionnée un nombre infini de fois. Pour un tel k maximum, à partir d'un certain rang on aurait un développement infini des radicaux de U_k, ce qui est impossible.

Démonstration du théorème de standardisation (Klop). Par construction st(D) est équivalente à D. Il suffit de montrer que st(D) est standard, et qu'une dérivation standard est unique dans sa classe d'équivalence. Nous laissons les détails de ces preuves au lecteur.

Remarque. Il est faux que R(D)=R(D') pour deux dérivations équivalentes D et D'. Par exemple, avec M=([x]I (I I)), D=[{top}] et D'=[{top,[B]}] sont deux dérivations équivalentes menant de M à sa forme normale I.

Définition. Une dérivation <u>normale</u> est une dérivation qui réduit à chaque étape le radical le plus à gauche.

Pour tout M, il existe au plus une dérivation normale de longueur donnée. Elle est calculée par l'interpréteur <u>normal</u>, défini par ϕ(M) = {gauche(M)} si M est réductible.

Théorème. L'interpréteur normal est correct.

Ceci est un corollaire immédiat du théorème de standardisation, car la standard d'une dérivation menant à la forme normale est normale.

Voici la procédure qui itère le calcul de l'interpréteur normal.

```
let onestep e = let u=gauche(e) in réduit e u;;
let rec normal e = try normal (onestep e) with NormalForm → e;;
```

Remarques.

1. Cette spécification n'est pas satisfaisante, car cet interpréteur recalcule à chaque fois le radical à réduire à partir du sommet du terme. Il n'y a pas de moyen direct d'écrire un interpréteur normal calculant récursivement sur la structure d'un lambda, à cause du phénomène de création des radicaux "vers le haut". Nous verrons comment résoudre cette difficulté dans la prochaine section.

2. Cet interpréteur n'est utile que pour les termes normalisables. Il boucle sur les termes non-normalisables, alors qu'intuitivement il est possible en général d'obtenir une information positive sur les approximations successives des termes obtenus le long d'un calcul, même ne se terminant pas sur une forme normale. Nous verrons ci-dessous comment calculer progressivement ces approximations.

3. En général, l'interpréteur normal n'est pas optimal (en nombre de réductions effectuées). Lorsqu'il y a des duplications, il vaut mieux souvent procéder de l'intérieur vers l'extérieur. Pourtant, l'interpréteur en appel par valeur n'est pas correct, car il peut faire des étapes non nécessaires; même lorsqu'il est correct, il n'est donc pas optimal non plus. Pour obtenir un interpréteur optimal, il faut compter pour une étape de calcul la réduction en parallèle de radicaux appartenant à la même famille. Intuitivement, deux radicaux sont dans la même famille s'ils sont résidus, ou créés de la même manière, par des radicaux de la même famille. Cette notion a été étudiée formellement par J.J. Lévy, qui a montré l'optimalité de cette stratégie. Remarquez toutefois qu'une telle stratégie généralise notre notion d'interpréteur, car il doit se rappeler de l'histoire de ses calculs. Cette mémorisation peut s'effectuer par des marques sur les λ-termes (étiquettes).

7. Approximations.

7.1. Forme normale de tête.

Les formes normales de tête sont les lambdas de la forme:

$$[x_1, \ldots, x_n](x\ M_1 \ldots M_p)$$

avec $x, x_1, \ldots, x_n$ variables, $M_1 \ldots M_p$ lambdas quelconques, $n,p \geq 0$.

Une forme normale de tête représente une approximation de forme normale, car elle est invariante (en n,x et p) par β-réduction.

```
type head = Head of num * num * lambda list;; (* n, x, [M1 ; ... ; Mp] *)
```

Voici l'algorithme de mise en forme normale de tête d'un lambda, par réduction normale :

```
let head = hnf 0 []              (* head : lambda → head *)
where rec hnf n stack = function
      Ref(x)     →  Head(n,x,stack)
    | Abs(e)     →  (match stack with
                          []          → hnf (n+1) [] e
                        | arg::args   → hnf n args (subst arg e))
    | App(g,d)   →  hnf n (d::args) g;;
```

On peut maintenant organiser l'interpréteur normal en itérant la fonction head.

```
type normal = Normal of num * num * normal list;;

let rec norm e =          (* norm : lambda → normal *)
    let (Head(n,x,args)) = head(e) in Normal(n,x,map norm args);;
```

7.2. Arbres de Böhm.

Il est aussi possible de définir un interpréteur qui calcule, de l'extérieur vers l'intérieur mais non forcément de la gauche vers la droite, une suite d'approximations décrivant un arbre infini généralisant la notion

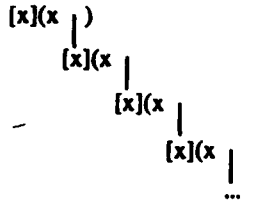
de forme normale. Un tel arbre s'appelle l'arbre de Böhm du lambda.

```
type approximation = BT of num * num * (générateur list)
and générateur == (void → approximation);;
```

L'approximation BT(n,x,[f_1; f_2;...; f_p]) représente l'information qu'on a sur un λ-terme lorsqu'on a reconnu qu'il se réduit sur la forme normale de tête : [x_1,x_2,...., x_n](x e_1 e_2 ... e_p). Le générateur f_i permet de calculer l'approximation de e_i par l'exécution de f_i(). La procédure approx ci-dessous calcule l'approximation d'un λ-terme par réduction normale :

```
(* approx : lambda → approximation *)
let rec approx = apx 0 []
where rec apx n args = function
        Ref(x)      →  BT(n,x,map (fun e () → approx(e)) args)
      | Abs(e)      →  (match args with
                            []          → apx (n+1) [] e
                          | first::rest → apx n rest (subst first e))
      | App(g,d)    →  apx n (d::args) g;;
```

Exemple. Avec D = [x,y](y (x x y)), Θ = (D D) et M = (Θ [x,y](y x)), on obtient approx(M) = BT(1,1,[f]), avec f() calculant récursivement la même approximation BT(1,1,[f]). L'arbre de Böhm de M est donc l'arbre infini :



Exercice. Redéfinir le type générateur en employant une structure de donnée paresseuse, plutôt que des fermetures.

Les lambdas normalisables ont un arbre de Böhm fini. Certains lambdas

ont un arbre de Böhm vide (indiqué par le symbole $\perp$), car la procédure head (ou approx) peut ne pas terminer.

Définition. On dit que le lambda M est <u>défini</u> si et seulement si head(M) termine.

Par exemple, M ci-dessus est défini, bien qu'il n'ait pas de forme normale. Par contre, Ω n'est pas défini, il a pour arbre de Böhm $\perp$. Les termes non définis sont absurdes du point de vue calculatoire : ils n'apportent aucune information.

Proposition. Si (M N) est défini, alors M est défini.
Nous laissons en exercice la preuve de cette proposition, qui utilise le théorème de standardisation.

Nous donnons maintenant une autre caractérisation des termes définis.

Définition. On dit que le lambda M est <u>solvable</u> si et seulement s'il existe n et $N_1, \ldots, N_n$, tels que $(\underline{M}\ N_1 \ldots N_n) \cong I$, où $\underline{M}$ désigne la fermeture $[u_1, u_2, \ldots u_k]M$ de $M \in \Lambda_k$.

Théorème de Wadsworth. Un lambda est défini si et seulement s'il est solvable.
Démonstration. Soit M un lambda, qu'on peut supposer fermé. Si M est solvable, alors il est défini, par la proposition ci-dessus.
Réciproquement, si $M \Rightarrow^* [x_1, x_2, \ldots, x_n](m\ e_1\ e_2 \ldots e_p)$, alors, en définissant $K^p = [x, y_1, y_2, \ldots, y_p]x$, on obtient :
$(M\ (K^p\ I)_1 \ldots (K^p\ I)_n) \Rightarrow^* (K^p\ I\ e'_1\ e'_2 \ldots e'_p) \Rightarrow I$.

8. Variations sur la notion de calcul

La notion de calcul que nous avons étudiée jusqu'à présent s'appelle la β-réduction du λ-K-calcul pur. Il existe de nombreuses variations sur le thème du λ-calcul. Dans le chapitre 3 nous verrons diverses variantes typées. Donnons ici brièvement quelques variantes du calcul pur, c'est à dire sans types.

8.1. Eta-réduction

On considère quelquefois une règle de calcul supplémentaire, appelée η-réduction, qui correspond à une forme faible de l'extensionnalité :

$$[x](M\ x) \twoheadrightarrow M \quad (x \text{ non libre dans } M) \qquad (\eta)$$

ou, en notation abstraite : $[](M^+\ 1) \twoheadrightarrow M$, avec $M^+ = \text{lift } 1\ M$.

Cette règle est justifiée par l'utilisation du λ-calcul pour dénoter des fonctions. En effet, pour tout lambda N, on a $([x](M\ x)\ N) \cong M$ lorsque x n'apparait pas libre dans M.

La théorie de la βη-réduction n'est pas aussi élégante que celle de la β-réduction seule, à cause de l'ambiguité des expressions de la forme $([x](M\ x)\ N)$, qui font se chevaucher des radicaux des deux sortes. Par une sorte de "miracle syntaxique", les deux sortes de réduction mènent au même résultat (M N), mais la notion de résidu n'est plus claire.

La propriété de confluence de la βη-réduction est conséquence des deux lemmes suivants, dont nous laissons la démonstration au lecteur. On utilise le symbole $\twoheadrightarrow$ pour la η-réduction (qui est étendue par congruence sur la structure des termes comme la β-réduction), et $\Rightarrow$ pour la βη-réduction.

Lemme de retard de la règle η.
Si $M \Rightarrow^* N$, alors il existe P tel que $M \Rightarrow^* P \twoheadrightarrow^* N$.

Lemme de commutation des règles β et η.
a. Si $M \Rightarrow N_1$ et $M \rightarrow N_2$ alors il existe P tel que $N_1 \rightarrow^* P$
et $N_2 \Rightarrow P$ ou $N_2 = P$.
b. Si $M \rightarrow N_1$ et $M \rightarrow N_2$ alors il existe P tel que $N_1 \rightarrow P$ et $N_2 \rightarrow P$.

8.2. Réduction faible

La réduction faible s'obtient en supprimant la règle ξ de congruence de la relation ⇨ par rapport à l'abstraction. Autrement dit, la réduction faible ne calcule pas à l'intérieur des abstractions. Ce calcul possède davantage de formes normales, puisque toutes les abstractions sont maintenant irréductibles.

La réduction faible se justifie par l'utilisation du λ-calcul pour décrire des calculs donnant ultimement des données concrètes. La règle ξ peut alors être interprétée comme une règle de manipulation de programmes plutôt qu'une règle de calcul proprement dite.

Nous verrons plus loin un autre formalisme de calcul, appelé logique combinatoire, et nous montrerons son équivalence avec la réduction faible dans le λ-calcul muni des règles β et η.

8.3. λ-I-calcul

Il s'agit ici d'une restriction à la syntaxe du λ-calcul : l'abstraction [x]M n'est autorisée que lorsque l'expression M a au moins une occurrence libre de la variable x. Les expressions autorisées sont appelées <u>strictes</u>, car elles représentent des fonctions strictes, qui ont toujours besoin de calculer leurs arguments.

Voici l'algorithme qui vérifie si un lambda est strict :

```
let rec strict = function
      Ref(_)     → true
    | Abs(e)     → (match locaux(e) with [] → false | _ → strict e)
```

```
  | App(g,d)  → (strict g) & (strict d);;
```

Nous verrons plus tard une interprétation des termes du λ-calcul typé comme les preuves d'un calcul propositionnel intuitioniste. La restriction du λ-I-calcul correspond alors à la logique dite relevante, qui ne reconnait la validité d'une preuve de A⇒B que si la preuve de B utilise effectivement l'hypothèse A.

La théorie syntaxique du λ-I-calcul est plus simple que celle du λ-K-calcul étudiée précédemment. Les termes solvables sont exactement les termes normalisables. Un terme est fortement normalisable si et seulement s'il est normalisable. L'interpréteur par valeur est correct. La dérivation normale est de longueur maximale dans sa classe.

8.4. λ-calcul linéaire

Il s'agit ici d'une restriction supplémentaire : l'abstraction [x]M n'est autorisée que lorsque l'expression M a exactement une occurrence de la variable x. Les expressions autorisées sont appelées linéaires, car elles représentent des fonctions qui utilisent leur argument de façon linéaire.

```
let rec linear = function
      Ref(_)     → true
    | Abs(e)     → (match locaux(e) with [_] → linear e
                                         | _  → false)
    | App(g,d) → (linear g) & (linear d);;
```

La théorie syntaxique devient presque triviale : les dérivations équivalentes ont toutes la même longueur.

9. Séparabilité.

9.1 Enonçé du théorème de Böhm

On rappelle les définitions : $T = [x,y]x$, $F = [x,y]y$.

Définition. On dit que les lambdas M et N sont <u>séparables</u> si et seulement s'il existe des termes fermés $P_1,...,P_n$ tels que $(M\ P_1\ ...\ P_n) \Rightarrow^* T$ et $(N\ P_1\ ...\ P_n) \Rightarrow^* F$.

Théorème de Böhm.
Deux formes β-normales non η-convertibles sont séparables.

Corollaire. Tout modèle identifiant deux formes β-normales non η-convertibles est incohérent.

Ici incohérent veut dire identifiant tous les termes fermés. Ce corollaire est immédiat, en remarquant que par définition de T et F on peut facilement remplacer dans la définition de séparable T et F par des termes fermés P et Q arbitraires.

La preuve du théorème de Böhm nécessite l'introduction de certaines notions sur les arbres de Böhm.

9.2 Chaînes, accessibilité.

Définition. Une <u>chaîne</u> de longueur s est une suite :
$S = [(1, n_1, m_1, p_1); (k_1, n_2, m_2, p_2)\ ...\ ; (k_{s-1}, n_s, m_s, p_s)]$,
vérifiant, pour tout $1 \le i \le s$: $n_i \ge 0$, $p_i \ge 0$, $1 \le m_i \le \Sigma[j=0,i]n_j$,
et pour tout $1 \le i < s$: $1 \le k_i \le p_i$.

Intuitivement, une telle chaîne dénote un chemin de longueur s dans un arbre de Böhm. Les n_i sont les arités des formes normales de tête

successives, les m_i les variables de tête, les p_i leur nombre d'arguments, l'occurrence dans l'arbre de Böhm est la liste des k_i.

Définition. Soit M est un terme fermé, L une liste de générateurs, et Q une liste d'entiers de longueur s. On dit que la chaîne S de longueur s est accessible dans M vers L modulo Q ssi :

- soit s=0, S=[], Q=[], L = [function () → hnf M],
- soit s=t+1, $S=T@[(k, n_s, m_s, p_s)]$, $Q=Q'@[q_s]$, où :

• $T = [(1, n_1, m_1, p_1); (k_1, n_2, m_2, p_2); \dots ; (k_{t-1}, n_t, m_t, p_t)]$ est accessible dans M vers $[f_1; f_2; \dots ; f_\lambda]$ modulo $Q'=[q_1; q_2; \dots ; q_t]$, avec $\lambda = p_t$ si t>0, 1 si t=0,

• $1 \le k \le \lambda$, $f_k() = BT(n,m,[g_1; g_2; \dots ; g_p])$, $p_s=p+q_s$, $n_s=n+q_s$, $m_s=m+\Sigma[j=r,s](q_j)$, avec r minimum tel que $m \ge \Sigma[j=r,s](n_j-q_j)$,

• $L= [g_1; g_2; \dots ; g_p]@[h_q; \dots ; h_2; h_1]$, où $h_j() = BT(0,j,[])$, pour $1 \le j \le q=q_s$.

On suppose dans la définition ci-dessus, et on vérifie par récurrence, les assertions suivantes :

• $1 \le r \le s$.

• $0 \le q_i \le n_i$ pour tout $1 \le i \le t$.

L'intuition est la suivante. S est un chemin dans l'arbre de Böhm d'une η-expansion de M indiquée par le vecteur Q. L'entier r est la hauteur dans cet arbre à laquelle est liée la variable m. A chaque étape i on s'autorise q_i applications de la règle de η-expansion (η-réduction inverse). On suppose que le long de ce chemin le sous-terme correspondant est défini.

Remarques.

1. si M et S sont donnés, et si S est accessible dans M vers L modulo Q, alors L et Q sont calculables de manière unique. Autrement dit, L et Q sont des fonctions récursives partielles de (M,S). On ne peut pas espérer mieux, car nous verrons qu'il est indécidable en général si un terme est défini ou non.

2. Si S est accessible dans M vers une liste L de longueur λ, alors pour

tout $1 \le k \le \lambda$, pour tout $n \ge 0$, si $S@[(k, n, m, p)]$ est accessible dans M, la paire (m, p) est unique. Dans ce cas on écrit :

$$(m, p) = \text{Accès}(M,S,k,n).$$

Remarquons qu'alors pour tout $q \ge 0$, on a : $(m+q, p+q) = \text{Accès}(M,S,k,n+q)$.

9.3 Technique de déböhmification.

Proposition 1. Soient M et N deux termes fermés, tels qu'il existe S, k et n tels que : $\text{Accès}(M,S,k,n) \neq \text{Accès}(N,S,k,n)$. Alors M et N sont séparables.

L'idée de la preuve ("Böhm out technique") est relativement simple, nous laissons les détails au lecteur. Le chemin S représente un chemin compatible dans les arbres de Böhm de M et N, modulo certaines η-expansions, qui mène à une différence irréconciliable, c'est à dire à des sous-termes définis avec des approximations incompatibles. La première étape consiste à projeter M et N le long de S, pour faire apparaître des instances de ces sous-termes par β-réduction de respectivement $(M\ P_1 \ldots P_s)$ et $(N\ P_1 \ldots P_s)$. On se ramène ainsi à l'un des 2 cas :

a) $[u,v](u\ M_1 \ldots M_i)$ à différencier de $[u,v](v\ N_1 \ldots N_j)$, ce qui est facile par application finale de $U = [u_1, u_2 \ldots, u_i]T$ et $V = [v_1, v_2 \ldots, v_j]F$.

b) $[u](u\ M_1 \ldots M_i)$ à différencier de $[u](u\ N_1 \ldots N_j)$, avec $i \neq j$. Dans ce cas, en supposant par exemple $i>j$, avec $k=i-j$, on applique $X = [u_1, u_2 \ldots, u_i]I$, puis $Y = [v_1, v_2 \ldots, v_k]T$, puis k fois F.

La seule difficulté qui se présente est dans la première étape, lorsqu'une variable apparait plusieurs fois le long du chemin S (ou simultanément sur S et en tant que variable de tête de l'un des deux sous-termes accédés par S). Dans ce cas il faut d'abord substituer à cette variable une expression du genre $[u_1, u_2 \ldots, u_n, w](w\ u_1 \ldots u_n)$, avec n suffisamment grand, ce qui permet de se ramener au cas où les variables n'apparaissent qu'une fois. L'exemple qui suit est caractéristique de cette technique.

Exemple. Soit $M = [u](u\ I\ (u\ I\ I))$ et $N = [u](u\ I\ (u\ u\ I))$. On prend :
$P_1 = P$, $P_2 = [u_1, u_2]u_2$, $P_3 = [u_1, u_2]u_1$, $P_4 = I$, $P_5 = [u]T$, $P_6 = [u_1, u_2]F$.

Proposition 2.
Soient M et N deux formes β-normales non η-convertibles. Il existe S, k et n tels que : Accès(M,S,k,n) ≠ Accès(N,S,k,n).

Nous laissons au lecteur la preuve de cette proposition. Le théorème de Böhm est maintenant conséquence directe des propositions 1 et 2.

On remarque qu'il n'est guère possible de faire mieux, dans la mesure où des formes normales η-convertibles ne sont pas séparables. Par exemple, considérez I et A=[u,v](u v).

L'importance du théorème de Böhm est qu'il donne *une condition* très forte sur la construction de modèles. Deux termes normalisables non β-convertibles ne sont identifiables dans un modèle non-trivial que s'ils sont βη-convertibles. Ceci fixe en gros les trois degrés de liberté d'un modèle par rapport au modèle purement syntaxique des arbres de Böhm :
- le modèle peut ou non vérifier η
- le modèle peut identifier plus ou moins les termes non solvables
- le modèle peut être plus ou moins riche en points non définissables.

II - Typage

1. Les types conjonctifs.

1.1 Expressions de types

On considère un ensemble dénombrable V de types atomiques. On définit récursivement l'ensemble T des expressions de types comme le plus petit ensemble contenant la constante ω, les éléments de V, et fermé par les deux opérations binaires $\rightarrow$ et $\wedge$:

$$\begin{array}{lcl} & & \omega \in T \\ \alpha \in V & \Rightarrow & \alpha \in T \\ \alpha, \beta \in T & \Rightarrow & \alpha \rightarrow \beta \in T \\ \alpha, \beta \in T & \Rightarrow & \alpha \wedge \beta \in T \end{array}$$

Nous distinguerons dans la suite trois niveaux de typage, correspondant à des restrictions du système ci-dessus. Le système complet sera désigné par E, le système sans la constante ω par D, et le système n'admettant que le constructeur $\rightarrow$ par C. On écrira donc T_C, T_D et T_E pour les trois ensembles d'expressions de type.

```
type typexpr =      Omega
                |   Atom of num
                |   Arrow of typexpr * typexpr
                |   Conj of typexpr * typexpr;;
```

1.2 Typage des λ-termes

Un contexte est une liste de types :

```
type contexte == typexpr list;;
```

Si Γ est un contexte de longueur n, et x la variable Ref(m), avec $m \leq n$, on note Γ_x pour la m-ième composante de Γ.

Pour faciliter la lecture, on emploie la notation concrète [x]M pour l'abstraction, et on écrit de même $\Gamma[x:\alpha]$ pour $\alpha::\Gamma$.

On donne maintenant récursivement la définition d'une relation entre un contexte Γ de longueur n, un λ-terme $M \in \Lambda_n$ et un type α, écrite $\Gamma \vdash_E M : \alpha$, et dite jugement de typage, comme la plus petite relation $\vdash$ vérifiant :

1. $\Gamma \vdash x : \Gamma_x$
2. $\Gamma[x:\alpha] \vdash M : \beta \Rightarrow \Gamma \vdash [x]M : \alpha \rightarrow \beta$
3. $\Gamma \vdash M : \alpha \rightarrow \beta \wedge \Gamma \vdash N : \alpha \Rightarrow \Gamma \vdash (M\ N) : \beta$
4. $\Gamma \vdash M : \alpha \wedge \beta \Rightarrow \Gamma \vdash M : \alpha$
5. $\Gamma \vdash M : \alpha \wedge \beta \Rightarrow \Gamma \vdash M : \beta$
6. $\Gamma \vdash M : \alpha \wedge \Gamma \vdash M : \beta \Rightarrow \Gamma \vdash M : \alpha \wedge \beta$
7. $\Gamma \vdash M : \omega$

Cet ensemble de règles concerne le système complet E. On vérifie que si $\Gamma \vdash_E M : \alpha$, alors $\alpha \in T_E$.

Expliquons maintenant la restriction au système D. On dit que Γ est un D-contexte de $M \in \Lambda_n$ ssi $\Gamma \in T_D{}^n$. Le jugement $\Gamma \vdash_D M : \alpha$ est alors restreint à Γ D-contexte de M, et est défini par les règles 1 à 6, avec $\alpha, \beta \in T_D$. De même que pour E, on vérifie que si $\Gamma \vdash_D M : \alpha$, alors $\alpha \in T_D$. On définit similairement le jugement $\vdash_C$, avec $\Gamma \in T_C{}^n$, et en se limitant aux règles 1,2 et 3.

On considère également une variante D' de D, qui utilise tous les types de T_E, mais mais dont le jugement de typage est défini par les règles 1 à 6, plus la règle :

7'. $\Gamma \vdash M : \alpha \Rightarrow \Gamma \vdash M : \omega$

Définition. Soit $M,N \in \Lambda_n$. On définit $M \subseteq N$ ssi $\Gamma \vdash M : \tau$ entraîne $\Gamma \vdash N : \tau$.

On écrit au besoin $\subseteq_C$, $\subseteq_D$ et $\subseteq_E$ pour préciser.

La proposition suivante est vraie pour les systèmes C,D, D' et E.

Proposition 1: monotonie du typage.
La relation $\subseteq$ est une Λ-pré-congruence, c'est à dire :

$M \subseteq M' \Rightarrow (M\ N) \subseteq (M'\ N)$
$M \subseteq M' \Rightarrow (N\ M) \subseteq (N\ M')$
$M \subseteq M' \Rightarrow [x]M \subseteq [x]M'$

Démonstration. Immédiate, par inspection de la définition du jugement de typage.

Corollaire. Soit R une Λ-relation, $\Lambda(R)$ la Λ-pré-congruence engendrée par R. Si R est contenue dans $\subseteq$, alors $\Lambda(R)$ l'est aussi.

1.3. Synthèse de type dans D.

Tout terme est typable dans le système E, puisque $\Gamma \vdash_E M : \omega$. Nous verrons plus loin que les termes solvables ont des types plus informatifs que ω. Le terme $(x\ x)$ n'est pas typable dans C, car il n'existe pas d'instance commune à $\alpha \to \beta$ et α. Il est typable dans D, puisque :

$$[x : (\alpha \to \beta) \wedge \alpha] \vdash_D (x\ x) : \beta.$$

Plus généralement, nous allons montrer que tout terme normal est typable dans D.

On montre d'abord un résultat technique. On définit un ordre d'inclusion des contextes, comme suit. On postule que l'opération $\wedge$ est associative, commutative, et idempotente, que ω est un élément neutre pour $\wedge$, et on écrit $\equiv$ pour l'équivalence correspondante dans T. On définit une relation $\leq$ par :

$$\alpha \leq \beta \Leftrightarrow \alpha \wedge \beta \equiv \alpha.$$

Il est facile de vérifier que $\leq$ définit un préordre admettant $\wedge$ pour inf, ω pour élément maximum, et $\equiv$ pour équivalence engendrée. On étend $\leq$ en un ordre d'inclusion des contextes de même longueur: $\Gamma \leq \Delta$ ssi pour tout x on

a : $\Gamma_x \leq \Delta_x$. De même on définit la conjonction $\Gamma \wedge \Delta$ de deux contextes Γ et Δ de même longueur, comme le contexte associant à x le type obtenu à partir de $\Gamma_x \wedge \Delta_x$ en simplifiant par les règles d'idempotence et de neutralité (modulo commutativité et associativité).

Proposition 2. Si $\Gamma \vdash_D M : \alpha$ et $\Delta \leq \Gamma$, alors $\Delta \vdash_D M : \alpha$.

Démonstration. Par récurrence sur la preuve de $\Gamma \vdash_D M : \alpha$. Les détails sont laissés au lecteur.

Remarque. La proposition 2 est également vraie dans les systèmes E, D' et C. On a également la propriété :

Proposition 3. Si $\Gamma \vdash_{D'} M : \alpha$ et $\alpha \leq \beta$, alors $\Gamma \vdash_{D'} M : \beta$.
Cette propriété est vraie aussi dans D avec la condition $\beta \in T_D$.

Lemme 1. Tout terme normal est typable dans D.

Démonstration. Soit $M \in \Lambda_n$ normal. On montre qu'il existe un D-contexte Γ de longueur n et $\alpha \in T_D$ tels que $\Gamma \vdash_D M : \alpha$. De plus α peut être choisi arbitrairement si M n'est pas une abstraction. Par récurrence contextuelle.

- Si M=x, avec α arbitraire, on prend $\Gamma = [x_1:\alpha_1]...[x:\alpha]:... [x_n:\alpha_n]$. Les α_i peuvent aussi être choisis arbitrairement, par exemple comme des éléments de V.
- Si M=[x]N, on a par récurrence $\Gamma[x:\gamma] \vdash_D N : \beta$. On en déduit $\Gamma \vdash_D M : \gamma \rightarrow \beta$.
- Si M=(N P), on a par récurrence $\Gamma \vdash_D P : \beta$. Pour α quelconque, on sait par hypothèse de récurrence que $\Delta \vdash_D N : \beta \rightarrow \alpha$, car M normal entraîne que N n'est pas une abstraction. On en déduit $\Gamma \wedge \Delta \vdash_D M : \alpha$, en utilisant la proposition 2.

Corollaire. Tout terme normal est typable dans D', et dans E, avec un type dans T_D, et un D-contexte.

Remarque. Les règles 1, 2 et 3 sont "dirigées par la syntaxe", alors que les règles 4, 5, 6 et 7 sont "structurelles". Il est possible de restreindre l'utilisation des règles structurelles; par exemple, nous laissons au lecteur la vérification de la proposition suivante.

Proposition 4. Dans une preuve de Γ |- [x]M : $\alpha\rightarrow\beta$, on peut supposer sans perte de généralité que la dernière règle employée est la règle 2.

1.4. Typage et *calcul.*

Nous montrons que le calcul préserve le typage. Le lemme suivant est vrai indifféremment pour les systèmes C, D, D', et E.

Lemme 2. ([x]M N) $\subset$ M[x$\leftarrow$N]

Démonstration. Récurrence sur la preuve du jugement de typage Γ |- ([x]M N) : α. Par cas suivant la dernière règle.

- règle 3. On a donc Γ |- [x]M : $\gamma\rightarrow\alpha$ et Γ |- N : γ. Par la proposition 4, le premier jugement provient de $\Gamma[x:\gamma]$ |- M:α. En remplaçant partout dans la preuve de ce dernier jugement les utilisations des hypothèses x:γ par des preuves de N : γ, on obtient une preuve de Γ |- M[x$\leftarrow$N] : α. (On laisse au lecteur les détails de cette récurrence)
- règles 4,5 et 6 : pas de problème
- de même pour les règles 7 (*pour E*) et 7' (*pour D'*).

Corollaire. M $\Leftrightarrow$ N $\Rightarrow$ M$\subset$N
Par le corollaire de la proposition 1.

Remarquons qu'il en est de même pour la η-réduction $\twoheadrightarrow$.

La réciproque du lemme 2 n'est vraie en général que dans E, à cause des combinateurs constants tels que **K**, qui peuvent faire disparaître par calcul des sous-termes non typables.

Lemme 3. $M[x \leftarrow N] \subseteq_E ([x]M\ N)$.

Démonstration. On donne l'idée. Soit $\Gamma \vdash M[x \leftarrow N] : \alpha$. Dans la preuve de ce jugement, soient $\Gamma \vdash N : \gamma_1$ ... $\Gamma \vdash N : \gamma_n$ tous les jugements employés aux différentes occurrences de x dans M. Soit $\gamma = \gamma_1 \wedge ... \wedge \gamma_n$ si $n>0$, $\gamma = \omega$ si $n=0$. On a $\Gamma[x:\gamma] \vdash M : \alpha$, d'où $\Gamma \vdash [x]M : \gamma \rightarrow \alpha$, et $\Gamma \vdash N : \gamma$, d'où finalement $\Gamma \vdash ([x]M\ N) : \alpha$.

Remarque. Pour expliciter formellement les preuves des lemmes 2 et 3, il faut utiliser $\Gamma\Gamma' \vdash N^{+n} : \gamma_i$, obtenu par itération de la propriété :

$$\Gamma \vdash M : \alpha \Rightarrow \Gamma[x:\tau] \vdash M^+ : \alpha.$$

Corollaire 1. $M \Leftrightarrow N \Rightarrow N \subseteq_E M$

Corollaire 2. $M \Leftrightarrow^* [x]N \Rightarrow$ M a d'autres E-types que ω.

Corollaire 3. Si M est normalisable, alors $\Gamma \vdash_E M : \tau$, avec $\tau \in T_D$ et $\Gamma \in T_D^n$.

Nous allons maintenant montrer que si de plus toutes les réductions issues de M terminent, c'est à dire si M est fortement normalisable, alors M est typable dans D.

1.5. Typage des termes fortement normalisables.

On montre d'abord un lemme qui raffine le lemme 3.

Lemme 4. Si N est D-typable, alors $M[x \leftarrow N] \subseteq_D ([x]M\ N)$.

Démonstration. Comme dans la preuve ci-dessus. Soit $\Gamma \vdash N : \beta$. On prend $\gamma = \gamma_1 \wedge ... \wedge \gamma_n$ si $n>0$, $\gamma = \beta$ si $n=0$.

Remarque. Tout sous-terme d'un terme D-typable l'est aussi. On obtient donc en corollaire le lemme 3 pour le système D, dans le cas du

λ-I-calcul.

Exercice. Le lemme 4 est facile pour D', en prenant $\gamma = \omega$ dans tous les cas. Montrer le lemme 4 dans le cas du système C, en utilisant le lemme du typage principal ci-dessous.

Lemme 5. Si M est fortement normalisable, alors M est D-typable.

Démonstration. Par récurrence sur la paire $(\Xi(M),\theta(M))$, où $\Xi(M)$ est la longueur de la plus longue dérivation issue de M, et $\theta(M)$ est la taille de M.

Si $\theta(M)=0$, M est normal, et est D-typable par le lemme 1. Sinon, M possède un sous-terme réductible, soit ([x]N P). Comme $\Xi(P)\leq\Xi(M)$ et $\theta(P)<\theta(M)$, on a P D-typable par hypothèse de récurrence. Soit M' le terme issu de M en réduisant ce radical. Comme $\Xi(M')<\Xi(M)$, on a M' D-typable par hypothèse de récurrence. M est donc D-typable, par le lemme 4 et le corollaire de la proposition 1.

Remarque. Un terme peut avoir tous ses sous-termes normalisables, sans être pour autant fortement normalisable. Par exemple, considérez

M = ([x] ([y]z (x x)) Δ), normalisable en z, mais non fortement normalisable car réductible à ([y]z Ω).

On a bien $[z : \alpha] \vdash_E M : \alpha$, avec $\alpha \in T_D$ (corollaire du lemme 3), mais M n'est pourtant pas typable dans D.

Problème.

Le lemme 1 n'est pas vrai pour le système C, puisque Δ n'est pas C-typable. Est-il possible d'enrichir C en C', avec la règle 7', pour obtenir le lemme 1, et donc le lemme 5, sans pour autant avoir de types conjonctifs? Montrer que les types ω et $\omega\rightarrow\alpha$ suffisent. Retrouver les types de Wadsworth.

2. Interprétation des types.

2.1. Parties Φ-saturées

Soit Λ l'ensemble des λ-termes. Soit Φ une partie quelconque de Λ.

Une partie A de Λ est dite <u>Φ-saturée</u> ssi pour tous termes M, N_1, ... , N_n de Λ on a, pour tout N dans Φ :

$$(M[x\leftarrow N]\ N_1\ N_2 \dots N_n) \in A \Rightarrow ([x]M\ N\ N_1\ N_2 \dots N_n) \in A.$$

Nous allons maintenant interpréter les types comme des parties Φ-saturées de λ-termes. L'opérateur ∧ sera interprété comme l'intersection. Il est clair que si A et B sont Φ-saturées, alors A∩B l'est aussi. Donnons maintenant l'interprétation de l'opérateur →.

Définition. Soient A et B des parties de Λ. On définit :

$$A\rightarrow B = \{M \mid \forall N \in A\ (M\ N) \in B\}.$$

Lemme 6. Si B est Φ-saturée, A→B l'est aussi pour tout A.

Démonstration. Soit $(M[x\leftarrow N]\ N_1\ N_2 \dots N_n) \in A\rightarrow B$, avec $N \in \Phi$. Soit $P \in A$. On a $(M[x\leftarrow N]\ N_1\ N_2 \dots N_n\ P) \in B$ par définition de A→B, et donc également $([x]M\ N\ N_1\ N_2 \dots N_n\ P) \in B$ puisque B est Φ-saturée. On a donc, par définition de A→B : $([x]M\ N\ N_1\ N_2 \dots N_n) \in A\rightarrow B$.

2.2. Φ-interprétations

Définition. Soit Φ une partie quelconque de Λ. On appelle <u>D-interprétation</u> relative à Φ toute application I qui associe à tout type atomique α une partie Φ-saturée I(α). On étend I à tous les types, par :

$$I(\alpha\wedge\beta) = I(\alpha)\cap I(\beta)$$
$$I(\alpha\rightarrow\beta) = I(\alpha)\rightarrow I(\beta).$$

On définit de même une D'-interprétation, en choisissant pour $I(\omega)$ une partie Φ-saturée telle que $I(\alpha) \subseteq I(\omega)$ pour tout type α.

Enfin, une *E-interprétation* est une D-interprétation relative à $\Phi=\Lambda$, vérifiant de plus : $I(\omega) = \Lambda$.

On obtient, dans les systèmes D, D', et E, pour toute interprétation relative à Φ :

Corollaire du lemme 6. Pour tout type τ, $I(\tau)$ est Φ-saturé.

Lemme de cohérence dans D. Soit I une D-interprétation relative à Φ telle que $I(\tau) \subseteq \Phi$ *pour tout type* $\tau \in T_D$. Soit $M \in \Lambda_n$, et Γ un D-contexte $[x_1:\alpha_1; \ldots ; x_n:\alpha_n]$. Soit $N_1 \in I(\alpha_1), \ldots , N_n \in I(\alpha_n)$. Alors pour tout typage $\Gamma \vdash_D M : \alpha$, on a $M[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n] \in I(\alpha)$.

Démonstration. Par récurrence sur la preuve du typage. Par cas suivant la dernière règle du typage $\Gamma \vdash_D M : \alpha$:

1. $M = x_i$, $\alpha = \alpha_i$ d'où $M[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n] = N_i \in I(\alpha)$.

2. $M = [x]M'$, $\alpha = \beta \rightarrow \gamma$, $\Gamma[x:\beta] \vdash_D M' : \gamma$, pour tout $N \in I(\beta)$ on a par hypothèse de récurrence $M'[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n][x \leftarrow N] \in I(\gamma)$. Comme $I(\gamma)$ est Φ-saturé, et que $N \in I(\beta) \subseteq \Phi$ on a $(M[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n]\ N) \in I(\gamma)$. Par définition de $I(\alpha) = I(\beta) \rightarrow I(\gamma)$, on a donc $M[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n] \in I(\alpha)$.

3. $M = (M_1\ M_2)$, $\Gamma \vdash_D M_2 : \beta$, $\Gamma \vdash_D M_1 : \beta \rightarrow \alpha$, par hypothèse de récurrence on a $M_1[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n] \in I(\beta) \rightarrow I(\alpha)$ et $M_2[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n] \in I(\beta)$, d'où $M[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n] \in I(\alpha)$.

4, 5 et 6. Evident, car $I(\beta \wedge \gamma) = I(\beta) \cap I(\gamma)$.

Lemme de cohérence dans D'. Soit I une D'-interprétation relative à Φ telle que $I(\tau) \subseteq \Phi$ pour tout type $\tau \in T_E$. Soit $M \in \Lambda_n$, et Γ un E-contexte $[x_1:\alpha_1; \ldots ; x_n:\alpha_n]$. Soit $N_1 \in I(\alpha_1), \ldots , N_n \in I(\alpha_n)$. Alors pour tout typage $\Gamma \vdash_D M : \alpha$, on a $M[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n] \in I(\alpha)$.

Démonstration. Comme ci-dessus, avec en plus le cas 7' :

7'. $\alpha = \omega$, $\Gamma \vdash_D M : \beta$, et par hypothèse de récurrence on a $M[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n] \in I(\beta) \subseteq I(\omega)$.

Lemme de cohérence dans E. Soit I une E-interprétation. Soit $M \in \Lambda_n$, et Γ un E-contexte $[x_1:\alpha_1; \ldots ; x_n:\alpha_n]$. Soit $N_1 \in I(\alpha_1), \ldots , N_n \in I(\alpha_n)$. Alors pour tout typage $\Gamma \vdash_E M : \alpha$, on a $M[x_1 \leftarrow N_1] \ldots [x_n \leftarrow N_n] \in I(\alpha)$.

Démonstration. Comme ci-dessus, avec en plus le cas 7, trivial car $I(\omega) = \Lambda$.

2.3. Φ-adéquation

Définition. Soit Ψ et Θ des parties quelconques de Λ.
On dit que le couple (Ψ,Θ) est Φ-adéquat ssi Ψ est Φ-saturé, $\Theta \subseteq \Psi$, et :
A1. Pour tous $M \in \Theta$ et $N \in \Psi$, on a $(M\ N) \in \Theta$
A2. Si pour tout $N \in \Theta$ on a $(M\ N) \in \Psi$, alors $M \in \Psi$.

Soit (Ψ,Θ) un couple Φ-adéquat. Soit $E(\Psi,\Theta)$ l'ensemble des parties Φ-saturées de Ψ qui contiennent Θ. On a $\Psi \in E(\Psi,\Theta)$, et $E(\Psi,\Theta)$ est fermé par intersection. Montrons qu'il est également fermé par l'opération $\rightarrow$.

Lemme 7. Si $X,Y \in E(\Psi,\Theta)$, alors $X \rightarrow Y \in E(\Psi,\Theta)$.

Démonstration. Soit $X,Y \in E(\Psi,\Theta)$, $Z = X \rightarrow Y$. On doit montrer :

- a. $Z \subseteq \Psi$. En effet, soit $M \in Z$. Considérons $N \in \Theta$ quelconque. $X \in E(\Psi,\Theta)$ entraîne $\Theta \subseteq X$, et donc $N \in X$. On obtient donc, par définition de l'opération $\rightarrow$, $(M\ N) \in Y \subseteq \Psi$. Par la condition A2 ci-dessus, on a bien $M \in \Psi$.
- b. Z est Φ-saturé. Par le lemme 6, car Y est Φ-saturé.
- c. $\Theta \subseteq Z$. En effet, soit $M \in \Theta$. Pour tout $N \in X \subseteq \Psi$, on a par la condition A1 ci-dessus $(M\ N) \in \Theta \subseteq Y$, et donc $M \in Z$ par définition de $\rightarrow$.

Corollaire 1. Soit (Ψ,Θ) un couple Φ-adéquat, I une D-interprétation relative à Φ telle que $I(\alpha)\in E(\Psi,\Theta)$ pour tout type atomique α. Alors $I(\alpha)\in E(\Psi,\Theta)$ pour tout $\alpha\in T_D$.

Corollaire 2. Soit (Ψ,Θ) un couple Φ-adéquat, I une D'-interprétation relative à Φ telle que $I(\alpha)\in E(\Psi,\Theta)$ pour tout type atomique α, et $I(\omega)\in E(\Psi,\Theta)$. Alors $I(\alpha)\in E(\Psi,\Theta)$ pour tout $\alpha\in T_E$.

Corollaire 3. Soit (Ψ,Θ) un couple Φ-adéquat, I une E-interprétation telle que $I(\alpha)\in E(\Psi,\Theta)$ pour tout type atomique α. Alors $I(\alpha)\in E(\Psi,\Theta)$ pour tout $\alpha\in T_D$.

Remarque. $\Lambda\in E(\Psi,\Theta)$ n'est vrai que lorsque $\Psi=\Lambda$, cas limite de peu d'intérêt, car on cherche à utiliser Ψ et Θ comme bornes des termes typés, i.e. $\Theta\subset I(\tau)\subset\Psi$.

2.4. Applications

Dans les deux applications qui suivent, on prendra pour Θ le plus petit ensemble $\Theta(\Psi)$ contenant les variables, et fermé par la condition A1 ci-dessus, c'est-à-dire l'ensemble des termes $(x\ N_1\ \ldots\ N_p)$, avec $N_i\in\Psi$, $p\geq 0$.

2.4.1. Dans D ou D' : Fortement normalisables

Supposons que Ψ soit une partie Φ-saturée de Φ, et soit Θ tel que le couple (Ψ,Θ) soit Φ-adéquat. Soit I une D-interprétation relative à Φ, telle que $I(\alpha)=\Psi$ pour tout type atomique α. Par le corollaire 1 ci-dessus, on obtient $\Theta\subset I(\tau)\subset\Psi\subset\Phi$ pour tout type $\tau\in T_D$. On peut utiliser le lemme de D-cohérence. Soit $M\in\Lambda_n$. Si $\Theta=\Theta(\Psi)$, toute variable est dans $I(\tau)$ pour tout type $\tau\in T_D$. On obtient donc que $\Gamma\vdash_D M:\alpha$ entraîne :

$$M=M[x_1\leftarrow x_1]\ \ldots\ [x_n\leftarrow x_n]\in I(\alpha)\subset\Psi.$$

Exemple.
On prend $\Phi=\Psi=$l'ensemble N^* des termes fortement normalisables. On

montre que le couple $(N^*,\Theta(N^*))$ est N^*-adéquat (Exercice). On obtient la réciproque du lemme 5. D'où :

Théorème 1. M est fortement normalisable ssi M est D-typable.

Dans D', la même construction est possible. On prend $I(\omega)=\Psi$, et le même raisonnement, en utilisant le lemme de D'-cohérence, montre que M D'-typable entraîne $M\in\Psi$. Le même exemple montre donc que le typage dans D' est équivalent au typage dans D. On a juste un peu de flexibilité supplémentaire, avec des types tels que $\omega\to\omega$, qui est le type des termes qui donnent un fortement normalisable quand appliqué à un fortement normalisable.

2.4.2. Dans E : normalisables

On prend $\Phi=\Lambda$, et Ψ l'ensemble N des termes normalisables. On montre que le couple $(N,\Theta(N))$ est Λ-adéquat (Exercice). Soit I une E-interprétation, telle que $I(\alpha)=N$ pour tout type atomique α. Par le corollaire 3 ci-dessus, on obtient $\Theta\subseteq I(\tau)\subseteq\Psi$ pour tout $\tau\in T_D$. Soit $M\in\Lambda_n$, tel que :

$[x_1:\alpha_1; \dots ; x_n:\alpha_n] \vdash_E M : \alpha$, avec $\alpha, \alpha_1, \dots, \alpha_n\in T_D$. Puisque $\Theta=\Theta(\Psi)$, toute variable est dans $I(\tau)$ pour tout type $\tau\in T_D$, et donc $x_1\in I(\alpha_1), \dots, x_n\in I(\alpha_n)$. On peut utiliser le lemme de E-cohérence, d'où :

$M = M[x_1\leftarrow x_1] \dots [x_n\leftarrow x_n]\in I(\alpha)\subseteq\Psi$.

On obtient donc la réciproque du corollaire 3 du lemme 3 :

Théorème 2. M est normalisable ssi $\Gamma\vdash_E M : \tau$, avec $\tau\in T_D$ et $\Gamma\in T_D{}^n$.

Corollaire. La propriété d'être E-typable sans ω est indécidable.

Exercice. Utiliser pour Ψ l'ensemble des termes normalisables par dérivation normale. En déduire une autre preuve du fait qu'un terme est normalisable ssi il est normalisable par dérivation normale.

Problème. Trouver d'autres interprétations intéressantes.

Remarque. Cette section est inspirée des notes de cours de J.L. Krivine "Lambda-calcul typé". Notre traitement a l'avantage d'être uniforme pour les deux applications ci-dessus. Le système E est nommé DΩ dans Krivine.

Aide-mémoire. La méthode permet d'encadrer l'interprétation d'un type τ entre les ensembles Θ et Ψ. Dans les deux applications, on obtient :

$$\Theta \subseteq I(\tau) \subseteq \Psi \subseteq \Phi \subseteq \Lambda.$$

Dans D on prend $\Psi = \Phi = \mathbf{N}^*$, dans E on prend $\Psi = \mathbf{N}$ et $\Phi = \Lambda$.

Remarque. Cette méthode impose $I(\tau) \neq \varnothing$ pour tout type τ. En effet, si on avait $I(\tau) = \varnothing$ pour un τ, cela imposerait $\Theta = \varnothing$, et donc $\Psi = \Lambda$ par la condition A2. On n'aurait dans ce cas limite aucune information sur l'interprétation : $\varnothing \subseteq I(\tau) \subseteq \Lambda$ pour tout τ.

3. Typage polymorphe.

3.1. Types principaux.

Définition. Morphismes de types. Une application σ de T dans T est un morphisme de types ssi:

$\sigma(\alpha\wedge\beta) = \sigma(\alpha)\wedge\sigma(\beta)$

$\sigma(\alpha\rightarrow\beta) = \sigma(\alpha)\rightarrow\sigma(\beta)$

$\sigma(\omega) = \omega$.

On étend naturellement une telle application à un contexte.

Proposition 5. Si $\Gamma \vdash M : \alpha$, alors pour tout morphisme de types σ, on a aussi $\sigma(\Gamma) \vdash M : \sigma(\alpha)$.

Démonstration. Facile, par inspection de la relation de typage. Le résultat est vrai dans tous les systèmes considérés.

Définitions. On dit que le typage $\Gamma \vdash M : \alpha$ est plus général que le typage $\Delta \vdash M : \beta$ ssi il existe un morphisme de types σ tel que $\Delta=\sigma(\Gamma)$ et $\beta=\sigma(\alpha)$. On dit que le typage $\Gamma \vdash M : \alpha$ est principal ssi α est plus général que tout type de M dans le contexte Γ.

Un terme ne possède pas en général de D-typage principal. Par exemple, le combinateur I possède dans le contexte vide tous les types $\alpha_1\rightarrow\alpha_1\wedge \alpha_2\rightarrow\alpha_2 \wedge \ldots \wedge \alpha_n\rightarrow\alpha_n$, pour n arbitraire, et ces types ne sont pas en général unifiables en $\alpha\rightarrow\alpha$. Nous allons voir que le résultat est par contre vrai dans la discipline C.

3.2. Synthèse de type dans C.

On procède comme dans la synthèse de type étudiée dans la preuve du lemme 1. La différence principale est que l'opération $\wedge$ lors de la conjonction des contextes est remplacée par l'unification des schémas de type correspondant. Cette opération peut échouer, ou réussir avec un

unificateur principal.

Définition. Soit $M \in \Lambda_n$, et Γ un C-contexte de M. On dit que M est C-typable modulo Γ ssi il existe un morphisme de types σ et un type τ tels que $\sigma(\Gamma) \vdash_C M : \tau$.

Lemme de typage principal dans C.
Soit $M \in \Lambda_n$, et Γ un C-contexte de M. Si M est C-typable modulo Γ, alors il existe un C-typage principal de M modulo Γ.

Démonstration.
On procède comme plus haut, par récurrence contextuelle sur M.
- Si M=x, on a $\Gamma \vdash_C M : \Gamma_x$
- Si M=[x]N, on choisit α nouvelle variable de type. Par récurrence, il y a deux cas. Si $\sigma(\Gamma[x:\alpha]) \vdash_C N : \beta$, on en déduit $\sigma(\Gamma) \vdash_C M : \sigma(\alpha) \to \beta$. Sinon, c'est à dire si N n'est pas C-typable modulo $\Gamma[x:\alpha]$, alors M n'est pas C-typable modulo Γ.
- Si M=(N P), supposons $\sigma(\Gamma) \vdash_C P : \beta$, et $\rho(\sigma(\Gamma)) \vdash_C N : \gamma$. Soit α une nouvelle variable de type. Si $\rho(\beta)$ et $\gamma \to \alpha$ ne sont pas unifiables, M n'est pas typable. Sinon, soit ξ leur unificateur principal. On en déduit $\xi(\rho(\sigma(\Gamma))) \vdash_C M : \xi(\alpha)$. Dans tous les autres cas M n'est pas C-typable modulo Γ.

Nous laissons la preuve de la principalité de la construction au lecteur.

Exemple. [x][y](x (y x)) est C-typable dans le contexte vide, de type principal $(\alpha \to \beta) \to (((\alpha \to \beta) \to \alpha) \to \beta)$, avec α et β deux variables de type.

Remarque. Pour tout terme M, on obtient un typage principal de M, lorsque M est C-typable, en prenant pour Γ le contexte qui associe à toute variable une variable de type distincte.

Problème. Montrer que le calcul du type principal d'un λ-terme M dans C est linéaire en la taille de M.

Remarque. Le langage des C-types n'ayant qu'un seul symbole de

fonction, la seule cause de non-unification de deux schémas de types est le test d'occurrence, qui refuse l'unification d'une variable de type α avec un schéma $\tau \to \tau'$ contenant une occurrence de α. Si on relaxe cette restriction, ce qui revient à autoriser des types récursifs tels que $\alpha = \alpha \to \alpha$, alors tout terme devient typable trivialement.

Remarque. λ-termes typés.

Dans le système C, un typage est entièrement déterminé par une déclaration de type aux variables. Le contexte donne le type des variables libres. Pour les variables liées, il est usuel de déclarer leur type dans un champ supplémentaire de l'opérateur d'abstraction.

Il faut faire attention toutefois à distinguer les λ-termes avec types des λ-termes typés. Par exemple, il est généralement admis que les λ-termes avec types admettent la propriété de Church-Rosser, en utilisant le fait que les termes purs l'admettent, et que le calcul préserve les types. Mais ce raisonnement n'est valable que pour les termes qui ont effectivement un type. Mais il est faux par exemple que le βη-calcul possède la propriété de Church-Rosser sur les termes avec types, comme le montre :

Contre-exemple (Nederpelt). Le terme $[x:\alpha]([y:\beta]y\ x)$ se β-réduit en $[x:\alpha]x$ et se η-réduit en $[y:\beta]y$.

3.3. Polymorphisme simple.

A partir de maintenant, on donne une forme plus lisible au radical : ([x]M N) en l'écrivant let x=N in M.

Considérons le terme let i=[x]x in (i i). Ce terme n'est pas typable dans C. L'algorithme ci-dessus le rejette, car les schémas de types α et $(\alpha \to \beta)$ ne sont pas unifiables. Pourtant la forme réduite ([x]x [x]x) de ce radical est typable. Il est facile de comprendre la plus grande liberté obtenue en dupliquant le sous-terme [x]x : les deux occurrences peuvent être typées avec deux copies $\alpha_1 \to \alpha_1$ et $\alpha_2 \to \alpha_2$ de son type principal $\alpha \to \alpha$. Du point de

vue du typage dans la discipline D, il est facile d'expliquer comment obtenir l'effet de cette duplication, sans avoir à effectuer la conversion. Le typage dans un contexte Γ du radical ([x]M N) aboutit d'une part à :

$\Gamma' \vdash N : \tau'$, avec $\Gamma' = \sigma(\Gamma)$

et d'autre part à :

$\Gamma''[x : \tau_1 \wedge \tau_2 \wedge ... \wedge \tau_n]) \vdash M : \tau'$, avec $\Gamma'' = \rho(\Gamma')$. Plutôt que de chercher à unifier les τ_i comme plus haut, on transforme au contraire τ' en $\tau'_1 \wedge \tau'_2 \wedge ... \wedge \tau'_n$, où les τ'_i sont des copies de τ', mais avec un renommage des variables de type locales à τ' (c'est à dire n'apparaissant pas dans Γ'). On peut maintenant chercher à réconcilier les deux parties de l'application en unifiant $\tau_1 \wedge \tau_2 \wedge ... \wedge \tau_n$ et $\tau'_1 \wedge \tau'_2 \wedge ... \wedge \tau'_n$. Autrement dit, on unifie séparément τ_1 et τ', τ_2 et τ', ..., τ_n et τ', mais en oubliant entre chaque unification les substitutions aux variables locales de τ'. C'est-à-dire, si α_1, α_2, ..., α_k sont les variables locales de τ', on considère que τ' est générique en les α_j, i.e. $\tau' = \forall\alpha_1\,\alpha_2 ... \alpha_k.\tau$. L'unification de τ_i et τ ne garde pas trace des substitutions aux α_j.

Cette méthode est facilement implémentable. Il suffit de garder l'information, pour chaque composante du contexte, de quelles sont ses variables locales. On obtient l'algorithme qui suit. On appelle C$\forall$ la discipline correspondante de typage, intermédiaire entre C et D.

Algorithme de synthèse de type dans C$\forall$.

Soit M un terme à typer dans un contexte Γ. On dit que M est C$\forall$-typable modulo Γ si l'algorithme réussit en retournant une substitution σ et un type τ, ce qu'on écrit $\sigma(\Gamma) \vdash M : \tau$. Sinon l'algorithme échoue.

On procède par récurrence contextuelle sur M.

- Si M=x, avec $\Gamma_x = \forall\alpha_1\,\alpha_2 ... \alpha_k.\tau$, on a $\Gamma \vdash M : \tau'$, avec τ' obtenu à partir de τ en substituant k nouvelles variables aux α_j.

- Si M=[x]N, on choisit α nouvelle variable de type. Par récurrence, il y a deux cas. Si $\sigma(\Gamma[x:\alpha]) \vdash N : \tau$, on en déduit $\sigma(\Gamma) \vdash M : \sigma(\alpha) \to \tau$. Sinon, c'est à dire si N n'est pas typable modulo $\Gamma[x:\alpha]$, alors l'algorithme échoue.

- Si M=(N P), supposons $\sigma(\Gamma) \vdash P : \tau$. Il y a deux cas.

- Si N=[x]N', soient $\alpha_1\ \alpha_2 \ldots \alpha_k$ les variables de τ non libres dans $\sigma(\Gamma)$, et $\Sigma=\forall\alpha_1\ \alpha_2 \ldots \alpha_k.\tau$. Si $\sigma(\Gamma[x:\Sigma]) \vdash N' : \tau'$, on en déduit $\sigma(\Gamma) \vdash M : \tau'$. Sinon l'algorithme échoue.

- Sinon, soit $\rho(\sigma(\Gamma)) \vdash N : \tau'$. Soit α une nouvelle variable de type. Si $\rho(\tau)$ et $\tau' \to \alpha$ ne sont pas unifiables, l'algorithme échoue. Sinon, soit ξ leur unificateur principal. On en déduit $\xi(\rho(\sigma(\Gamma))) \vdash M : \xi(\alpha)$. Sinon, l'algorithme échoue.

Théorème. Si M est typable dans la discipline C∀, alors :
- M est D-typable
- N est C-typable, où N est obtenu à partir de M par réduction parallèle de tous ses radicaux.

Problème (Kanellakis-Mitchell).
1. Montrer que l'algorithme ci-dessus est exponentiel en la profondeur d'imbrication des radicaux dans le terme considéré.
2. Investiger la structure de graphes avec pointeurs sous-jacente au problème du typage dans la discipline C∀.
3. Montrer que ce problème est difficile en espace polynomial ("P-space hard") en lui réduisant le problème QBF (formules booléennes quantifiées).

Problème. Typage polymorphe paresseux. La discipline C∀ correspond à simuler une étape de réduction en parallèle, avec l'appel par valeur, puis de typer dans C. Il est possible d'étendre à une discipline analogue, mais employant l'évaluation normale. La discipline C∀ω s'obtient en ajoutant la constante ω au langage T_C, et en remplaçant les cas d'échec par la réponse ω. Montrer que M est de type principal τ dans la discipline C∀ω, avec τ sans occurrence de ω, ssi N est C-typable, où N est obtenu à partir de M par réduction parallèle de tous ses radicaux. La discipline C∀ω permet ainsi de typer des termes normalisables, mais non fortement normalisables, tels que KIΩ.

Problème. Typage polymorphe itéré. On peut généraliser la discipline C∀ en une discipline C∀2, qui simule deux étapes de réduction

parallèle. On obtient ainsi une hiérarchie de disciplines C∀n, avec n=1,2,... De même pour C∀ωn. La discipline limite C∀ω* permet ainsi de typer tous les termes admettant une forme normale C-typable. Investiger ces différents systèmes, la décidabilité et éventuellement la complexité du typage.

3.4. Extension aux types produits.

Il est facile d'ajouter un produit de types ×, et un constructeur de paires (M,N), avec ses opérateurs de projection fst et snd.

Problème (Klop). Développer le λ-calcul pur avec paires et règles de projection : $\underline{fst}(x,y) \Rightarrow x$ π_1

$\underline{snd}(x,y) \Rightarrow y$ π_2

Montrer Church-Rosser. Montrer que CR échoue avec la règle de paire surjective :

$(\underline{fst}(x), \underline{snd}(x)) \Rightarrow x$ SP.

La discipline C× est obtenue à partir de C en ajoutant les règles :

$\Gamma \vdash M : \alpha \wedge \Gamma \vdash N : \beta \Rightarrow \Gamma \vdash (M,N) : \alpha \times \beta$

$\Gamma \vdash M : \alpha \times \beta \Rightarrow \Gamma \vdash \underline{fst}(M) : \alpha$

$\Gamma \vdash M : \alpha \times \beta \Rightarrow \Gamma \vdash \underline{snd}(M) : \beta$

Finalement, il n'y a pas de difficulté d'étendre le typage C∀ en un typage C×∀.

Problème. Développer le formalisme de λ-calcul typé. Dans C, puis C×, avec les différentes combinaisons de règles de calcul. Montrer CR, etc.

Application. La discipline C×∀ est celle qui est utilisée dans le langage ML. En fait, le noyau de ML peut se décrire à partir de l'ajout au système des opérateurs de conditionnelle et de point-fixe, ainsi que de types pré-définis.

3.5. Extension au langage ML.

On enrichit le calcul C×∀ par des constantes, dont les types, éventuellement polymorphes, contiennent des types atomiques constants tels que bool et int. Par exemple :

$\underline{\text{if}} : \forall\alpha.\ \text{bool} \times \alpha \times \alpha \to \alpha$
$\underline{\text{true}} : \text{bool} \qquad \underline{\text{false}} : \text{bool}$
$\underline{0} : \text{int} \quad \underline{1} : \text{int} \quad \ldots \quad \underline{-1} : \text{int} \quad \ldots$
$= : \text{int} \times \text{int} \to \text{bool}$
$+ : \text{int} \times \text{int} \to \text{int} \qquad - : \text{int} \times \text{int} \to \text{int} \qquad * : \text{int} \times \text{int} \to \text{int}$

Le langage obtenu, appelons-le ML$^-$, est complété par une fonctionnelle de point fixe :

$\underline{Y} : \forall\alpha.\ (\alpha \to \alpha) \to \alpha$

Finalement, on introduit une abréviation pour les définitions récursives par :

$\underline{\text{let rec}}\ x{=}M\ \underline{\text{in}}\ N \twoheadrightarrow \underline{\text{let}}\ x{=}\underline{Y}([x]M)\ \underline{\text{in}}\ N.$

On obtient ainsi un langage fonctionnel polymorphe de base, appelé ML.

Remarque. ML utilise l'évaluation par valeur, ce qui consiste à calculer les radicaux de l'intérieur vers l'extérieur, par opposition à la règle d'évaluation normale. La règle de typage du let est d'ailleurs cohérente avec ce régime d'évaluation. Par contre, la règle ξ n'est pas utilisée, c'est à dire qu'on ne calcule pas à l'intérieur des abstractions. Les termes de la forme [x]M sont considérés comme des formes normales, appelées fermetures.

Les règles de calcul de la conditionnelle sont:

$\underline{\text{if}}(\underline{\text{true}},x,y) \Rightarrow x$
$\underline{\text{if}}(\underline{\text{false}},x,y) \Rightarrow y$

mais là également, x et y ne sont pas évalués à l'intérieur du if. Les

règles de calcul de l'arithmétique correspondent aux tables d'opérations. Finalement, on donne une règle de calcul du combinateur de point-fixe Y, telle que :

$$Y(f) \Rightarrow f(Y(f)).$$

Exemple. Vérifier que let rec fact n = if(n=0,1,n*fact(n-1)) in fact(3) s'évalue bien en 6.

Problème. ML+. Au lieu de définir la récursion à partir de l'opérateur Y, on peut se donner une construction supplémentaire μ, telle que les différentes occurrences d'une variable définie récursivement puissent être typées par des instanciations séparées d'un type polymorphe. Cette discipline C$\mu\forall$ permet de typer par exemple le terme suivant, qui n'est pas typable avec Y dans la discipline C$\forall$:

let rec f = [x](f (K x)).

Milner a démontré l'existence d'un type principal dans cette discipline. Mais le problème de la décidabilité de l'existence d'un type est encore un problème ouvert. Kfoury a montré que cette discipline permet de définir une classe de fonctions récursives plus riche que celles définissables dans la discipline C$\forall$.

3.6. Système T.

On remarque que la règle d'évaluation du point-fixe $\underline{Y}$ invalide la propriété de terminaison du calcul, ce qui montre que $\underline{Y}$ n'est pas définissable dans le calcul typé. Ici nous avons un dilemne : le système ML⁻ n'est pas assez puissant pour programmer une classe suffisante de fonctions arithmétiques, et l'ajout de $\underline{Y}$ donne directement toutes les fonctions partielles récursives, mais au détriment d'une propriété essentielle : la normalisation forte. On peut donc se poser le problème d'enrichir ML⁻ par des procédés de définition récursive plus limités que $\underline{Y}$.

Tout d'abord, on constate que ML⁻ ne permet de programmer que les fonctions polynomiales. Même en s'autorisant de coder les entiers par des itérateurs fonctionnels, on peut montrer que la classe des fonctions définissables est incluse dans la classe des fonctions élémentaires récursives, au sens de Kalmar, c'est à dire des exponentielles itérées. Pour obtenir plus, il faut se donner un opérateur de récursion. C'est l'idée du système T de Gödel.

On enrichit C×∀ par les types bool et nat. On se donne les opérateurs :

$\underline{true}$: bool $\underline{false}$: bool	(bool intro)
$\underline{if} : \forall\alpha.\ bool \times \alpha \times \alpha \to \alpha$	(bool elim)
$\underline{0}$: nat $\underline{S}$: nat → nat	(nat intro)
$\underline{R} : \forall\alpha.\ nat \times (nat \to (\alpha \to \alpha)) \times \alpha \to \alpha$	(nat elim)

Les règles de calcul sont, pour la conditionnelle, celles données plus haut, et pour le récurseur :

$\underline{R}(\underline{0},f,x) \Rightarrow x$

$\underline{R}(\underline{S}(n),f,x) \Rightarrow (f\ n\ \underline{R}(n,f,x))$

Exercices.

1. (Kleene) Programmer la fonction prédecesseur.

2. (Ackerman) Programmer la fonction définissable en ML par :
$\underline{let}\ \underline{rec}$ ack(m,n) =$\underline{if}$(m=$\underline{0}$,n+$\underline{1}$,$\underline{if}$(n=$\underline{0}$,ack(m-$\underline{1}$,n-$\underline{1}$),ack(m-$\underline{1}$,ack(m,n-$\underline{1}$))))

3. Montrer qu'on peut remplacer le récurseur par un itérateur Nat, avec :
Nat : $\forall\alpha.\ \text{nat} \times (\alpha \to \alpha) \times \alpha \to \alpha$
Nat(0,f,x) ⇨ x
Nat(S(n),f,x) ⇨ (f Nat(n,f,x))

L'exercice 2 ci-dessus montre qu'on obtient beaucoup plus que les fonctions primitives récursives, grâce à la fonctionalité. Les fonctionnelles définissables dans T sont appelées fonctionnelles de type fini. On peut montrer que la famille des fonctionnelles de type nat → nat sont exactement les fonctions récursives prouvablement totales dans l'arithmétique de Peano (ou, ce qui est équivalent, dans l'arithmétique de Heyting). On obtient les fonctions primitives récursives en restreignant l'opérateur R au type nat, c'est à dire avec seulement :
Rnat : $\text{nat} \times (\text{nat} \to (\text{nat} \to \text{nat})) \times \text{nat} \to \text{nat}$.

Problème. Etendre la méthode de la partie 2 pour montrer la normalisation forte des termes typables dans le système T.

3.7. Généralisation à des structures de données.

Il est possible d'étendre le système T avec d'autres structures que les booléens et les entiers. Voici un survol des possibilités.

3.7.1. Sommes.

On peut introduire un constructeur + de types sommes, avec les opérations :

inl : $\forall\alpha\beta.\ \alpha \to \alpha+\beta$ inr : $\forall\alpha\beta.\ \beta \to \alpha+\beta$ (somme intro)
match : $\forall\alpha\beta\gamma.\ (\alpha+\beta) \times (\alpha \to \gamma) \times (\beta \to \gamma) \to \gamma$ (somme elim)

et les règles de calcul :

match(inl(x),y,z) ⇨ y(x) ι_1
match(inr(x),y,z) ⇨ z(x). ι_2

On peut alors représenter le type bool par un type somme, avec inl jouant le rôle de false (resp. inr jouant le rôle de true) et match jouant le rôle de if.

L'équivalent de la règle SP pour la somme est :
match(x,inl,inr) ⇨ x SS.

Problème. Etudier le calcul C×+. Etudier les diverses combinaisons de règles de calcul, entre les règles de coupure (β, π_1, π_2, ι_1, ι_2) et les règles d'unicité-initialité (η, SP, SS), dans les cas typé et non-typé.

Problème. Etudier l'axiomatisation catégorique (CCC) du calcul C× typé. Montrer que l'extension C×+ ne correspond pas à postuler un co-produit, qui exige la loi supplémentaire de distributivité du conditionnel:
f(match(x,g,h)) = match(x,f o g,f o h) D.

3.7.2. Types récursifs.

Si on veut généraliser le type nat, il faut maintenant pouvoir exprimer des définitions récursives de type, du style :
type nat = 0 + S of nat.
type α list = Nil + Cons of α × α list.

De telles définitions engendrent automatiquement les opérateurs d'introduction et d'élimination correspondants, ainsi que leurs règles de calcul. Par exemple, pour nat on obtient les opérateurs 0, S, et Nat considérés plus haut. De manière analogue, pour list on obtient :
Nil : ∀α. α list Cons : ∀α. (α × α list) → α list
List : ∀αβ. α list × (α × β → β) × β → β

avec les règles d'itération de liste :

List(Nil,f,x) ⇨ x
List(Cons(a,l),f,x) ⇨ f(a,Nat(l,f,x)).

La règle de formation des définitions récursives de type peut être contrainte de plusieurs façons :

a - Pas de contrainte : on peut écrire
type lambda = Quote of lambda → lambda.

Exercice. Montrer qu'avec ce type on peut simuler le lambda-calcul non-typé, et donc on ne peut espérer un résultat de normalisation forte.
Remarque: CAML se place à ce niveau.

b - Récursions positives.
On peut écrire les types algébriques nat et list ci-dessus, mais aussi des types plus compliqués tels que les ordinaux :

type ord = Zero + Succ of ord + Limit of nat → ord.

Problème. Généraliser la discipline C∀ à une discipline C∀μ, autorisant les types récursifs positifs. Etendre à cette discipline le résultat de normalisation forte.

Pour en savoir plus.

L'ouvrage de base sur le λ-calcul non typé est le livre de Henk Barendregt : "The Lambda Calculus, Its Syntax and Semantics", North-Holland, revised edition, 1984. Une monographie plus courte, du même auteur, est disponible en tant que chapitre du "Handbook of Mathematical Logic", édité par J. Barwise, North-Holland, 1977.

Le meilleur livre général sur le λ-calcul (typé ou non) et la théorie des combinateurs est "Introduction to Combinators and λ-calculus", de Roger Hindley et Jonathan Seldin, Cambridge University Press, 1986.

Pour le λ-calcul typé, et ses connections avec la théorie des catégories, on lira "Introduction to higher order categorical logic", de J. Lambek et P. J. Scott, Cambridge University Press, 1986, ainsi que la monographie de Pierre-Louis Curien : Categorical Combinators, Sequential Algorithms and Functional Programming, Pitman & Wiley, 1986.

A titre historique, on pourra consulter la monographie d'origine d'Alonzo Church : "The calculi of lambda-conversion", Ann. of Math. Studies 6, Princeton University, 1941, ainsi que les œuvres d'Haskell Curry et de ses collaborateurs : Combinatory Logic, chez North-Holland (Vol 1, 1968, et Vol 2, 1972).

Un certain nombre de résultats importants ne sont disponibles que dans des thèses. On citera tout particulièrement la thèse de J.J. Lévy, Université Paris 7, 1977, celle de J.Y. Girard, Université Paris 7, 1972, et celle de J.W. Klop, Université d'Utrecht, 1980, disponible comme la monographie "Combinatory Reduction Systems" du Mathematisch Centrum d'Amsterdam.

Les connections entre λ-calcul typé et théorie de la démonstration sont bien expliquées dans les notes de cours de J. Y. Girard "Types and Proofs", Cambridge University Press, 1989, ainsi que dans celles de J. L. Krivine "λ-calcul typé", à paraître chez Masson. Un ouvrage plus avancé est le

livre de J. Y. Girard "Proof theory and logical complexity", Bibliopolis (Vol 1, 1987, Vol 2, à paraître).

Il n'existe pas de présentation synthétique des modèles du λ-calcul. Il faut lire les travaux de recherche de D. Scott, G. Plotkin, G. Berry, A. Meyer, J. Y. Girard, K. Koymans, P. L. Curien, R. Statman. On lira aussi avec profit les notes de cours de G. Longo (Carnegie-Mellon University, 1988).

Un recueil important de résultats de recherche est le volume anniversaire dédié à H. Curry : "Essays on Combinatory Logic, Lambda Calculus and Formalism", Academic Press, 1980.

La théorie intuitioniste des types de P. Martin-Löf est décrite dans sa monographie "Intuitionistic Type Theory", Bibliopolis, 1984, ainsi que dans les notes "An Introduction to Martin-Löf Type Theory", du groupe Programming Methodology, Göteborg, 1986. Une implémentation en est décrite dans "Implementing Mathematics with the Nuprl Development System", de R. Constable et al., Prentice-Hall, 1986.

Le Calcul des Constructions est étudié dans la thèse de Thierry Coquand, Université Paris 7, 1985, ainsi que dans divers papiers de recherche. Voir aussi mes notes de cours "Formal Structures for Computation and Deduction", Carnegie-Mellon University, 1986.

Le calcul Automath est décrit dans une série de papiers de N. de Bruijn, dont beaucoup sont non publiés, ainsi que dans les thèses de R. Nederpelt "Strong Normalization in a λ-calculus with λ-structured types" (Eindhoven, 1973), D. van Daalen "The language theory of Automath" (Eindhoven, 1980), et R. de Vrijer "Surjective Pairing and Strong Normalization (Eindhoven, 1987).

Enfin, signalons que la théorie des combinateurs est présentée sous forme de récréations mathématiques dans un livre de R. Smullyan : "To mock a mockingbird", Knopf, 1985.