Initiation au λ-calcul

Gérard Huet

INRIA

Notes de cours du DEA "Fonctionnalité, Structures de Calcul et Programmation" donné à l'Université Paris VII de 1987 à 1991.

15 Mars 1991

Table des matières

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, ainsi que des systèmes de types dépendants. Une dernière section traite des systèmes de types généralisés, et de diverses versions du Calcul des Constructions.

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.

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 :

$$x,y \mapsto \sin(x)+\cos(y) \qquad (1)$$

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 valeurs ordinaires (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

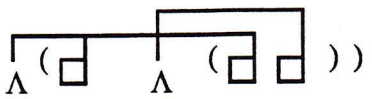
Un certain nombre de propositions de représentations des λ-termes modulo renommage des variables liées ont été faites. Nous en discutons

certaines sur l'exemple:

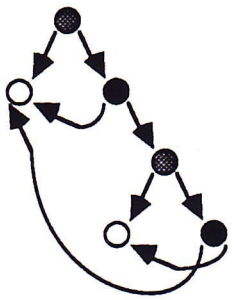
[x](x [y](x y)).

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.

A tout terme M de Λ_n correspond la fermeture $\underline{M}$ = closure M n, avec :

```
let rec closure e = function
     0  →  e
   | n  →  closure (Abs e) (n-1);;
```

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.

Au principe de récurrence contextuelle correspond le procédé de définition par récursion contextuelle. Par exemple, donnons les algorithmes de substitution.

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);;
```

La fonction (lift k) injecte Λ_n dans Λ_{n+k}, avec n quelconque. Si M=lift k N, on écrit $N=lift^{-1}$ k M. On a ($lift^{-1}$ k M) = lift (-k) M, sauf qu'il faut vérifier maintenant m+k>0 dans le cas variable libre, et échouer dans le cas contraire.

Voici maintenant l'algorithme de substitution. Si M est un lambda construit dans un environnement x::Γ et N est un lambda construit dans

l'environnement Γ, alors M{x ← 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 ← [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 <u>réduction</u> ⇨ entre λ-termes, comme suit.

([x]M N) ⇨ M{x ← N} (β)

On dit que le terme ([x]M N) est un <u>radical</u>, qui se <u>réduit</u> en M{x ← N}. On étend la relation ⇨ par congruence par rapport à la structure de λ-terme :

M ⇨ M' ⇒ []M ⇨ []M' (ξ)
M ⇨ M' ⇒ (M N) ⇨ (M' N)
M ⇨ M' ⇒ (N M) ⇨ (N M')

Historiquement, la fermeture réflexive-transitive ⇨* de la relation ⇨ s'appelle la <u>β-réduction</u>.

La β-réduction est la <u>règle de calcul</u> 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 :

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

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

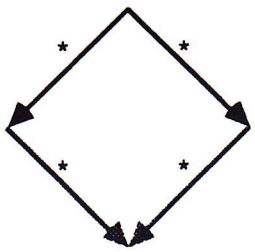
I	=	[x]x	
K	=	[x, y]x	
S	=	[x, y, z](x z (y z))	
A	=	[x, y](x y)	
B	=	[x, y, z](x (y z))	
C	=	[x, y, z](x z y)	
P	=	[x, y, z](z x y)	
T	=	[x, y]x	(= **K**)
F	=	[x, y]y	
0	=	[f, x]x	(= **F**)
1	=	[f, x](f x)	(= **A**)
n	=	[f, x](f (f ... (f x)))	(n fois f)
Y	=	[f]([x](f (x x)) [x](f (x x)))	Turing
Θ	=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

4. Le théorème du losange

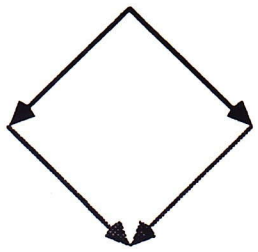
4.1 Enoncé du théorème et corollaires

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

L'algorithme factor calcule le préfixe commun maximal de deux occurrences, avec les suffixes correspondants :

```
let factor = fact_prefix []
where rec fact_prefix w = function
      (A::u,A::v) → fact_prefix (A::w) (u,v)
    | (B::u,B::v) → fact_prefix (B::w) (u,v)
    | (C::u,C::v) → fact_prefix (C::w) (u,v)
    | suffixes    → (rev w, suffixes);;
```

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 le transfert d'une occurrence comme le nombre d'abstractions qu'elle traverse :

```
let transfert u = trans (0,u) where rec trans = function
      (n, C::u)   → trans(n+1,u)
    | (n, _::u )  → trans(n,u)
    | (n, [])     → n;;
```

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 définit l'égalité des deux sous-termes u et v du terme M comme suit:

```
let eq_sub M (u,v) =
      let (_,u',v') = factor(u,v) in
      lift^-1 (transfert u') (sub(M,u)) = lift^-1 (transfert v') (sub(M,v));;
```

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

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

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

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 $M \Rightarrow_u N$ 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";;
```

On écrit simplement $M \Rightarrow N$ pour $\exists\, u \in R(M)\ \ M \Rightarrow_u N$.

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 : $R \Rightarrow M'$, avec $M' = M\{x \leftarrow N\}$. 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(g,d)   → (sons A (loc n g)) @ (sons B (loc n d))
  | Abs(e)     → sons C (loc (n+1) e);;
```

Maintenant soit T un terme possédant à l'occurrence u le radical T/u = R. On a $T \Rightarrow_u T'$, avec $T' = T[u \leftarrow M']$. 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)
(* u∈R(M)   v∈dom(M)  *)
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 du domaine"
    | (_,[])           → [v]
    | (A::w,B::w')     → [v]
    | (B::w,A::w')     → [v]
    | (A::w,A::w')     → res(w,w')
    | (B::w,B::w')     → res(w,w')
    | (C::w,C::w')     → res(w,w')
    | _                → Erreur "Occurrence hors du domaine";;
```

Proposition. Soit $M \Rightarrow_u N$.

1. $\forall v \in dom(N)\ \exists! w \in dom(M) : v \in$ résidus M u w.
   On écrit : w = résidus^{-1} M u v.
2. $\forall w \in dom(M), \forall v_1, v_2 \in$ résidus M u w : eq_sub N (v_1, v_2).
3. $\forall w \in R(M)$: (résidus M u w) $\subseteq R(N)$.

Nous laissons au lecteur la preuve de cette proposition. Toute occurrence dans le terme réduit est donc résidu d'une occurrence unique. Les sous-termes résidus d'une même occurrence sont égaux. Les résidus d'une occurrence de radical sont des occurrences de radicaux, qui intuitivement partagent le même "grain de calcul". On appelle radical résidu dans N une occurrence de R(N) résidu d'une occurrence de R(M). Par

contre, certains radicaux de N proviennent d'un nouvel assemblage App(Abs(g),d). On dit que ces radicaux sont <u>créés</u> par le calcul. Formellement, on définit RR et RC comme suit :

RR(M,u) = {v∈R(N) | résidus^{-1} M u v∈R(M)}, RC(M,u) = R(N) - RR(M,u), avec N=réduit M u.

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 <u>dérivation</u> $M = M_1 \Rightarrow M_2 \Rightarrow \ldots M_n \Rightarrow N$ 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:

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

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

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

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.

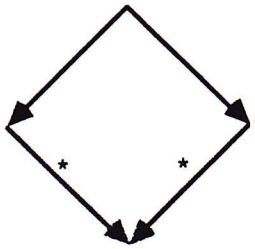
Remarque. Les résidus donnent une idée de "partage" : les sous-termes résidus d'un sous-terme donné peuvent être "partagés" avec ce sous-terme, plutôt que copiés. Toutefois, ce partage ne rend pas compte d'un partage plus complexe de "contextes", ou portions de termes non nécessairement sous-termes.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

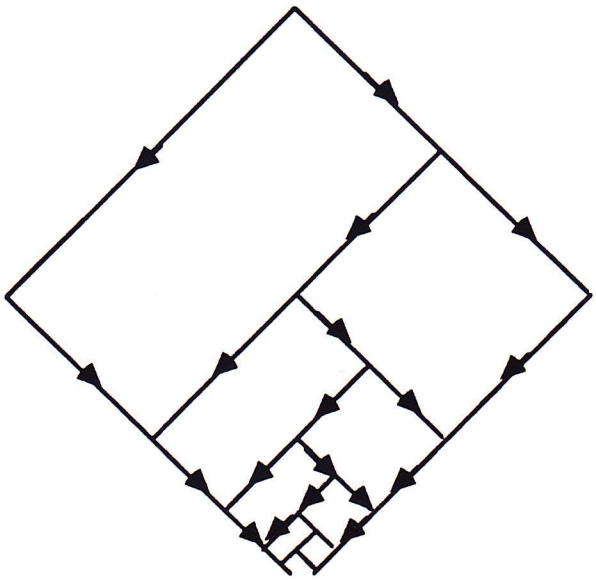


Figure 2

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 $\triangleright \subseteq \twoheadrightarrow$ et $\twoheadrightarrow \subseteq \triangleright^*$, et donc que $\twoheadrightarrow^* = \triangleright^*$. 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. 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 réduit pas de radicaux créés, 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(L) l'ensemble des dérivations issues d'un lambda L.

Théorème de Lévy. (Der(L)/≡ , ≤) 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 abstractions par des entiers positifs appelés poids.

5. Récurrence sur les calculs

5.1 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. Nous développons tout d'abord ce formalisme.

5.2 Termes pondérés

```
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 décroissant 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\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 :

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

5.3 Démonstration du théorème des développements finis.

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).

5.4 Sous-dérivations strictes

Considérons deux dérivations parallèles co-initiales de même longueur : $A = (M, [U_1; \ldots U_n])$ et $B = (M, [V_1; \ldots V_n])$. On dit que B est une sous-dérivation stricte de A, et on écrit B « A, ssi $B = A\backslash u$, avec $u \in R(M)$ tel que $\exists k \leq n$ tel que, avec $U = u\backslash[U_1; \ldots ; U_{k-1}]$, on ait $\varnothing \neq U \subseteq U_k$.

Lemme. La relation « est nœthérienne.

Démonstration.
Par l'absurde. Supposons qu'il existe une suite infinie A_1 » A_2 » A_3 » ... Appelons k la colonne sélectionnée dans la définition ci-dessus. Soit r la colonne sélectionnée infiniment souvent d'indice maximum. Après un certain nombre d'étapes il n'y a plus de colonne sélectionnée d'indice supérieur à r. On obtient alors un développement infini de U_r, ce qui est impossible.

On tire de ce lemme la justification de récurrences sur les dérivations.

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 interprète du λ-calcul. On aimerait que cet interprète puisse être défini comme une fonction récursive ϕ associant à tout terme M un ensemble $\phi(M)$ de ses radicaux, avec $\phi(M) = \varnothing$ si et seulement si M est en forme normale. On appelle un telle fonction stratégie de calcul. La stratégie est dite séquentielle si $\phi(M)$ est singleton pour tout terme M non en forme normale ; elle est dite parallèle sinon.

La stratégie ϕ est dite correcte si elle 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 $M_1=M$, $M_n=Normal(M)$, et pour tout i, avec $1 \le i < n$, on a $M_{i+1}=M_i \backslash \phi(M_i)$.

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 une stratégie qui n'est pas correcte, comme le montre l'exemple $([x]I\ \Omega)$. Si on choisit pour $\phi(M)$ l'ensemble R(M) de tous les radicaux de M, on obtient la stratégie dite complète ou de Gross-Knuth, qui est correcte (pourquoi?).

Nous allons maintenant voir qu'il existe des stratégies séquentielles correctes qui calculent de l'extérieur vers l'intérieur.

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. $u \backslash v = \{u\}$.
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 $w \backslash v$.

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_0,S), avec $S=[u_1; \ldots ;u_n]$. Soit $S_{ij} = [u_i; \ldots ;u_j]$, $M_i=M_0 \backslash S_{1i}$. La dérivation est dite standard si et seulement si pour tous i,j tels que $1 \leq i<j \leq n$, 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 que, 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.

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

$([x](\underline{I\ x})\ (I\ y)) \Rightarrow ([x]x\ \underline{(I\ y)}) \Rightarrow \underline{([x]x\ y)} \Rightarrow y$ 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, on montre le théorème suivant.

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

Avant de démontrer 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:

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

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. $U_i = \varnothing$ 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 de préservation 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), qui est bien définie, car D\u«D. 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.

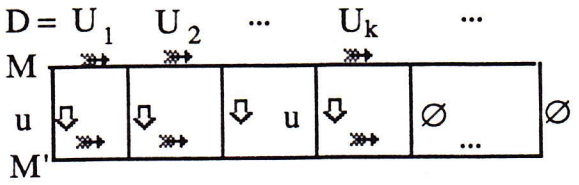


Figure 3

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 en exercice.

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 normale est une dérivation qui réduit à chaque étape le radical le plus à gauche.

Pour tout M, il existe au plus une dérivation normale issue de M de longueur donnée. Elle est calculée par la stratégie normale, définie par ϕ(M) = {gauche(M)} si M est réductible.

Théorème. La stratégie normale est correcte.

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

Voici la procédure qui itère le calcul de la stratégie normale.

```
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 interprète 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 interprète 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 interprète 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 plus loin comment calculer progressivement ces approximations.

3. En général, l'interprète 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, la stratégie d'appel par valeur (radical le plus interne le plus à gauche) n'est pas correcte, car elle peut faire des étapes non nécessaires; même lorsqu'elle est correcte, elle n'est donc pas optimale non plus. Pour obtenir un interprète 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 précédente, car elle 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, [M_1 ; ... ; M_p] *)

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 args = function
      Ref(x)     → Head(n,x,args)
    | Abs(e)     → (match args with
                          []          →hnf (n+1) [] e
                        | first::rest →hnf n rest (subst first 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);;
```

L'objection levée dans la remarque 1 ci-dessous est maintenant levée, mais l'objection 2 demeure. Voyons comment calculer progressivement une approximation de forme normale, pour des termes non forcémént normalisables.

7.2. Arbres de Böhm.

Il est 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.

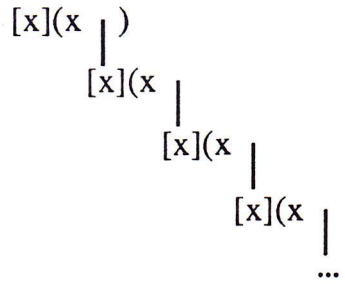
```
type approximation = BT of num * num * (générateur list)
and générateur == (unit → approximation);;
```

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 ⊥), 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 ⊥. 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. On rappelle que $\underline{M}$ désigne la fermeture $[u_1, u_2, \ldots u_k]M$ de $M \in \Lambda_k$.

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 $(\underline{M}\ N_1 \ldots N_n) \equiv \mathbf{I}$,

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 $M \triangleright^* [x_1,x_2,\ldots, x_n](m\ e_1\ e_2 \ldots\ e_p)$, alors, en définissant $\mathbf{K}^p = [x, y_1,y_2,\ldots, y_p]x$, on obtient :
$(M\ (\mathbf{K}^p\ \mathbf{I})_1 \ldots (\mathbf{K}^p\ \mathbf{I})_n) \triangleright^* (\mathbf{K}^p\ \mathbf{I}\ e'_1\ e'_2 \ldots\ e'_p) \triangleright^* \mathbf{I}$.

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'extensionalité :

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

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 $([x](M\ x)\ N) \cong (M\ N)$ lorsque x n'apparait pas libre dans M, et la règle η est donc justifiée par extensionalité.

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 $\twoheadrightarrow$ pour la βη-réduction.

Lemme de retard de la règle η.
Si $M \twoheadrightarrow^* N$, 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.

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 strictes, 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 Enoncé du théorème de Böhm

On rappelle les définitions : $\mathbf{T} = [x,y]x$, $\mathbf{F} = [x,y]y$.

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 $(\underline{M}\ P_1 \ldots P_n) \Rightarrow^* \mathbf{T}$ et $(\underline{N}\ P_1 \ldots P_n) \Rightarrow^* \mathbf{F}$.

On remarque que M et N sont séparables si et seulement si, pour tous termes fermés M' et N', il existe des termes fermés $P_1,...,P_n$ tels que $(\underline{M}\ P_1 \ldots P_n) \Rightarrow^* M'$ et $(\underline{N}\ P_1 \ldots P_n) \Rightarrow^* N'$.

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 :
$S = [(n_1, m_1, p_1, k_1); (n_2, m_2, p_2, k_2) \ldots ; (n_s, m_s, p_s, k_s)]$,
vérifiant, pour tout $1 \le i \le s$: $n_i \ge 0$, $1 \le m_i \le \Sigma[j=1,i]n_j$, $1 \le k_i \le p_i$.

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, B une approximation, et Q une liste d'entiers de longueur s. On dit que la chaîne S de longueur s est accessible dans M vers B modulo Q ssi :

- soit s=0, S=[], Q=[], B = approx M
- soit s=t+1, et :

• $T = [(n_1, m_1, p_1, k_1); (n_2, m_2, p_2, k_2) \ldots ; (n_t, m_t, p_t, k_t)]$ est accessible dans M vers $B' = BT(n,m,[f_1; f_2; \ldots ; f_p])$ modulo $Q'=[q_1; q_2; \ldots ; q_t]$,

• $Q=Q'@[q_s]$, avec $q_s \geq 0$,

• $p_s=p+q_s$, $n_s=n+q_s$,

• $m_s=m+\Sigma[j=r,s]q_j$, avec r minimum tel que $m \geq \Sigma[j=r,s](n_j-q_j)$,

• $S=T@[(n_s, m_s, p_s, k_s)]$, avec $1 \leq k_s \leq p_s$.

• Si $k_s=k \leq p$ on a $B = f_k()$, si $k_s= p_s+1-j$, avec $1 \leq j \leq q_s$, on a $B = BT(0,j,[])$.

L'intuition est la suivante. S est un chemin dans l'arbre de Böhm d'un terme βη–convertible à M. A chaque étape i on approxime le sous-terme à l'occurrence indiquée par la liste des k_i, et on fait q_i η-expansions. L'entier r est la hauteur dans cet arbre à laquelle est liée la variable m, il est bien défini par hypothèse M fermé. On suppose que le long de ce chemin les sous-termes correspondants sont définis. En particulier, M est défini, et à chaque étape où $k \leq p$, $f_k()$ termine.

Remarque.
M et S étant donnés, si S est accessible dans M vers B modulo Q, alors B et Q sont calculables de manière unique. Autrement dit, B 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.

Définitions. Soit S une chaîne, M et M' deux termes fermés tels que S est accessible dans M vers B modulo Q, et accessible dans M' vers B' modulo Q', avec $B = BT(n,m,[f_1; f_2; \ldots ; f_p])$, $B' = BT(n',m',[f_1; f_2; \ldots ; f_{p'}])$. On dit que S distingue M et M' ssi $(m+n',p+n') \neq (m'+n,p'+n)$. Finalement, on dit que les termes fermés M et M' sont distinguables s'il existe une chaîne qui les distingue.

9.3 Technique de déböhmification.

Proposition 1. Soient M et N deux termes fermés distinguables. Alors M et N sont séparables.

Donnons d'abord l'idée de la preuve ("Böhm out technique"). La chaîne S qui les distingue 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 $(M\ P_1\ ...\ P_s)$ et $(N\ P_1\ ...\ P_s)$. On se ramène ainsi à l'un des 2 cas illustrés par les exemples suivants :

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

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

La seule difficulté qui se présente est dans la première étape, lorsqu'une variable apparait plusieurs fois le long de la chaîne 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 $[u_1, u_2\ ...\ , u_n, w](w\ u_1\ ...\ u_n)$, avec n suffisamment grand, ce qui permet de se ramener au cas où les variables n'apparaissent qu'une fois.

Avant de détailler la démonstration, il convient de définir un certain nombre de notions. Tout d'abord, nommons les combinateurs utilisés dans la preuve.

Définitions.

1. Combinateurs de n-uplets.

$$\mathbf{P_n} = [u_1, u_2, \ ...\ , u_{n+1}](u_{n+1}\ u_1\ u_2\ ...\ u_n) \quad (n>0)$$

Par exemple, le combinateur de paire est $\mathbf{P} = \mathbf{P_2}$.

2. Combinateurs de projections.

$$\Pi_{k,n} = [u_1, u_2, \ ...\ , u_n]u_k \qquad (n \geq k > 0)$$

Par exemple, $\mathbf{I} = \Pi_{1,1}$, $\mathbf{T} = \Pi_{1,2}$, et $\mathbf{F} = \Pi_{2,2}$.

On écrit aussi $\mathbf{K_n}$ pour $\Pi_{1,n+1}$. Par exemple, $\mathbf{K} = \mathbf{K_1}$.

On vérifie : $(\mathbf{P_n}\ M_1\ M_2\ \ldots\ M_n\ \Pi_{k,n}) \rightarrow^*\ M_k$.

Maintenant, soient M et N fermés, et distingués par la chaîne : $S = [(n_1, m_1, p_1, k_1); (n_2, m_2, p_2, k_2); \ldots ; (n_s, m_s, p_s, k_s)]$. On suppose donc que S est accessible dans M vers B modulo Q, et accessible dans N vers B' modulo Q', avec $B = BT(n,m,[f_1; f_2; \ldots ; f_p])$, $B' = BT(n',m',[f_1; f_2; \ldots ; f_{p'}])$, tels que $(m+n',p+n') \neq (m'+n,p'+n)$.

Supposons $s>0$, et considérons la variable liée la plus externe. On définit son ensemble d'utilisations U comme l'ensemble des niveaux, entre 1 et $s+1$, où cette variable apparaît comme variable de tête le long du chemin considéré, dans M ou N. Formellement, $i \in U$, avec $1 \leq i \leq s$, ssi $m_i = \Sigma[j=1,i]n_j$, et $s+1 \in U$ ssi $m = n+\Sigma[j=1,s]n_j$ ou $m' = n'+\Sigma[j=1,s]n_j$. On définit aussi son ensemble d'arités comme $\{p_i \mid i \in U, i \leq s\}$, auquel on adjoint p et/ou p' quand $s+1 \in U$. On dit que la variable est schizophrène ssi $|U|>0$. Définitions similaires pour les n_1 variables liées au premier niveau. On appelle schizophrénie le nombre ζ de telles variables schizophrènes.

Démonstration.

On suppose tout d'abord, sans perte de généralité, que M et N ont été convertis par β-réduction en forme normale de tête. Le résultat général suivra par le théorème de Church-Rosser.

On montre la proposition par récurrence lexicographique sur $<s, \zeta, n_1>$.

• $s>0$. Il y a trois cas, suivant le nombre d'utilisations $u=|U|$ de la variable la plus externe.

• $u>1$: la variable est schizophrène. On prend $P=\mathbf{P}_\alpha$, avec α son arité maximum. On vérifie que (M P) et (N P) donnent lieu, après β-réduction, à une situation de même s, mais de moindre schizophrénie. Remarquons toutefois que la chaîne S doit être ajustée, car les n_i peuvent avoir augmenté.

• $u=0$: On choisit pour P un terme clos arbitraire, disons **I**, et on vérifie que (M P) et (N P) donnent lieu à une situation de même s et de même schizophrénie, mais avec un n_1 inférieur.

• $u=1$: Avec $U=\{i\}$, on prend $P=\Pi_{k,p}$, où $k=k_i$, et $p=p_i$, et on vérifie que (M P) et (N P) donnent lieu à une situation de plus petit s.

• $s=0$. Soit $M = [u_1, u_2, \ldots , u_n](m\ M_1\ M_2\ \ldots\ M_p)$,

$N = [u_1, u_2, \ldots, u_{n'}](m'\ N_1\ N_2 \ldots N_{p'})$, avec $(m+n',p+n') \neq (m'+n,p'+n)$.

Soit $i=n+1-m$, et $i'=n+1-m'$. Il y a deux cas :

• $i \neq i'$. Si $n \geq n'$, avec $k=p'+n-n'$, on prend $P_i = (\mathbf{K_p}\ \mathbf{T})$, $P_{i'} = (\mathbf{K_k}\ \mathbf{F})$, et tous les autres P_j, avec $1 \leq j \leq n$, égaux à **I**. Si $n<n'$, avec $k=p+n'-n$, on prend $P_i = (\mathbf{K_k}\ \mathbf{T})$, $P_{i'} = (\mathbf{K_{p'}}\ \mathbf{F})$, et tous les autres P_j, avec $1 \leq j \leq n'$, égaux à **I**.

• $i=i'$. On a donc $p+n' \neq p'+n$. De nouveau deux cas :

• $p+n'>p'+n$. Avec $k=p+n'-(p'+n)$, et $q=p$ si $n \geq n'$, $p+n'-n$ sinon, on prend $P_j = \mathbf{I}$, pour $1 \leq j<i$, $P_i = (\mathbf{K_q}\ \mathbf{I})$, $P_{i+1} = (\mathbf{K_k}\ \mathbf{T})$, et finalement $P_{i+j+1} = \mathbf{F}$, pour $1 \leq j \leq k$.

• $p+n'<p'+n$. Symétrique du précédent. Avec $k=p'+n-(p+n')$, et $q=p'$ si $n \leq n'$, $p'+n-n'$ sinon, on prend $P_j = \mathbf{I}$, pour $1 \leq j<i$, $P_i = (\mathbf{K_q}\ \mathbf{I})$, $P_{i+1} = (\mathbf{K_k}\ \mathbf{F})$, et finalement $P_{i+j+1} = \mathbf{T}$, pour $1 \leq j \leq k$.

Ceci termine l'esquisse de la démonstration de la proposition 1.

Exemple.

Soit $M = [u](u\ \mathbf{I}\ (u\ \mathbf{I}\ \mathbf{I}))$ et $N = [u](u\ \mathbf{I}\ (u\ u\ \mathbf{I}))$. $S = [(1, 1, 2, 2); (0, 1, 2, 2)]$. La variable u étant schizophrène, d'arité maximale 2, on prend $P_1 = \mathbf{P}$. On projette ensuite par $P_2 = [u_1, u_2]u_2$, $P_3 = [u_1, u_2]u_1$, ce qui mène à **I** et **P** respectivement. On sépare finalement par $P_4 = [u_1, u_2]\mathbf{T}$, $P_5 = \mathbf{I}$, $P_6 = [u_1, u_2]\mathbf{F}$.

Proposition 2.

Soient M et N deux lambdas normalisables non distinguables. Alors M et N sont βη-convertibles.

Démonstration. Par récurrence sur la somme des tailles des approximations de M et N. Soit $B=T(n,m,[f_1; f_2; \ldots ; f_p])$=approx M, $B'=BT(n',m',[g_1; g_2; \ldots ; g_{p'}])$=approx N, et supposons par exemple $n'>n$. Soit $q=n'-n$. Par q η-expansions de la forme normale de tête de M, on obtient M', avec approx $M' = BT(n',m+q,[f_1; f_2; \ldots ; f_p; h_1; \ldots ; h_q])$. La chaîne vide est accessible dans M vers B et dans N vers B'. Comme elle ne distingue pas M et N, on doit avoir $m+n'= m'+n$, et $p+n'=p'+n$, d'où $m'=m+q$, $p'=p+q$. Par hypothèse de récurrence, avec N' la forme normale de tête de N, on obtient que M'/j est βη-convertible à N'/j, pour $1 \leq j \leq p'$, et donc que M et N sont βη-convertibles.

Corollaire. Soient M et N deux formes β-normales non η-convertibles. Il existe une chaîne S qui les distingue.

Le théorème de Böhm est maintenant conséquence directe de la proposition 1 et de ce corollaire.

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 → et ∧ :

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

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 → 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≤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. $\Gamma[x:\alpha]\vdash M:\beta \Rightarrow \Gamma\vdash [x]M:\alpha\to\beta$
3. $\Gamma\vdash M:\alpha\to\beta \ \&\ \Gamma\vdash N:\alpha \Rightarrow \Gamma\vdash (M\ N):\beta$
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. $\Gamma\vdash M:\alpha \ \&\ \Gamma\vdash M:\beta \Rightarrow \Gamma\vdash M:\alpha\wedge\beta$
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, Λ(R) la Λ-pré-congruence engendrée par R. Si R est contenue dans $\subseteq$, alors Λ(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 ∧ est associative, commutative, et idempotente, que ω est un élément neutre pour ∧, et on écrit ≡ pour l'équivalence correspondante dans T. On définit une relation ≤ par :

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

Il est facile de vérifier que ≤ définit un préordre admettant ∧ pour inf, ω pour élément maximum, et ≡ pour équivalence engendrée. On étend ≤ 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 $\Gamma \vdash_D M : \gamma \to \beta$.
- Si M=(N P), on a par récurrence $\Gamma \vdash_D P : \beta$. Pour α quelconque, on sait par hypothèse de récurrence que $\Delta \vdash_D N : \beta \to \alpha$, 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 $\Gamma \vdash [x]M : \alpha \to \beta$, 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. $([x]M\ N) \subseteq M[x \leftarrow N]$

Démonstration. Récurrence sur la preuve du jugement de typage $\Gamma \vdash ([x]M\ N) : \alpha$. Par cas suivant la dernière règle.

- règle 3. On a donc $\Gamma \vdash [x]M : \gamma\to\alpha$ et $\Gamma \vdash N : \gamma$. Par la proposition 4, le premier jugement provient de $\Gamma[x:\gamma] \vdash M:\alpha$. En remplaçant partout dans la preuve de ce dernier jugement les utilisations des hypothèses $x:\gamma$ par des preuves de $N : \gamma$, on obtient une preuve de $\Gamma \vdash M[x\leftarrow N] : \alpha$. (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. $M \Rightarrow N \Rightarrow M\subseteq N$

Par le corollaire de la proposition 1.

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

$\vdash [x,y](x\ y) : (\alpha\to\beta)\to(\alpha\wedge\gamma)\to\beta$, mais on n'a pas $\vdash [x]x : (\alpha\to\beta)\to(\alpha\wedge\gamma)\to\beta$.

Il est possible de modifier le système de types pour qu'il devienne cohérent avec la η-réduction, en compliquant le préordre $\leq$ des types, en particulier en postulant la loi :

$$\alpha' \leq \alpha\ \&\ \beta \leq \beta' \Rightarrow \alpha\to\beta \leq \alpha'\to\beta'$$

et en donnant une règle explicite de sous-typage :

$$\Gamma \vdash M : \alpha\ \&\ \alpha \leq \beta \Rightarrow \Gamma \vdash M : \beta.$$

Mais les systèmes deviennent beaucoup plus compliqués.

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 $\gamma = \gamma_1 \wedge \ldots \wedge \gamma_n$ si $n>0$, $\gamma = \omega$ si $n=0$. On a $\Gamma[x:\gamma] \vdash M : \alpha$, d'où $\Gamma \vdash [x]M : \gamma\to\alpha$, 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. $M \Rightarrow N \Rightarrow N \subseteq_E M$

Corollaire 2. $M \Rightarrow^* [x]N \Rightarrow$ 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 $\gamma = \gamma_1 \wedge \ldots \wedge \gamma_n$ 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 longueur $\Xi(M)$ de la plus longue dérivation issue de M.

Si $\Xi(M)=0$, M est normal, et est D-typable par le lemme 1. Sinon, M possède un sous-terme réductible, soit ([x]N P). Comme $\Xi(P)<\Xi(M)$, 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.

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 Φ-saturée ssi pour tous termes M, N_1, ... , N_n de Λ on a, pour tout N dans Φ :

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

Une partie Φ-saturée est fermée par β-réduction de tête inverse, pourvu que l'argument appartienne à Φ.

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

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, $A \rightarrow B$ l'est aussi pour tout A.

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

2.2. Φ-interprétations

Définition. Soit Φ une partie quelconque de Λ. On appelle D-interprétation relative à Φ toute application I qui associe à tout type atomique α une partie Φ-saturée $I(\alpha)$. 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 Θ et Ψ des parties de Λ, avec $\Theta \subseteq \Psi$.
On dit que l'intervalle $[\Theta,\Psi]$ est Φ-adéquat ssi Ψ est Φ-saturé, 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 $[\Theta,\Psi]$ un intervalle Φ-adéquat. Soit $E[\Theta,\Psi]$ l'ensemble des parties Φ-saturées de Ψ qui contiennent Θ. On a $\Psi \in E[\Theta,\Psi]$, et $E[\Theta,\Psi]$ est fermé par intersection. Montrons qu'il est également fermé par l'opération $\rightarrow$.

Lemme 7. Si $X,Y \in E[\Theta,\Psi]$, alors $X \rightarrow Y \in E[\Theta,\Psi]$.

Démonstration. Soit $X,Y \in E[\Theta,\Psi]$, $Z = X \rightarrow Y$. On doit montrer :
- a. $Z \subseteq \Psi$. En effet, soit $M \in Z$. Considérons $N \in \Theta$ quelconque. $X \in E[\Theta,\Psi]$ 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 $[\Theta,\Psi]$ un intervalle Φ-adéquat, I une D-interprétation relative à Φ telle que $I(\alpha) \in E[\Theta,\Psi]$ pour tout type atomique α. Alors $I(\alpha) \in E[\Theta,\Psi]$ pour tout $\alpha \in T_D$.

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

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

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

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 $(x\ N_1\ ...\ N_p)$, 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 l'intervalle $[\Theta,\Psi]$ 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 $\Theta\subseteq I(\tau)\subseteq\Psi\subseteq\Phi$ 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 :

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

Exemple.
On prend $\Phi=\Psi=$l'ensemble $\mathbf{N^*}$ des termes fortement normalisables. On montre que l'intervalle $[\Theta(\mathbf{N^*}),\mathbf{N^*}]$ est $\mathbf{N^*}$-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 $\omega\rightarrow\omega$, 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 l'intervalle $[\Theta(\mathbf{N}), \mathbf{N}]$ est Λ-adéquat (Exercice). Soit I une E-interprétation, telle que $I(\alpha) = \mathbf{N}$ pour tout type atomique α. Par le corollaire 3 ci-dessus, on obtient $\Theta \subseteq I(\tau) \subseteq \Psi$ pour tout type $\tau \in T_D$. Soit $M \in \Lambda_n$, tel que :

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

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

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:

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

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

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

On étend naturellement une telle application à un contexte. On appellera aussi substitution un morphisme de types, en considérant les types atomiques comme des variables.

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 s'il est maximalement plus général..

Un terme ne possède pas en général de D-typage principal. Par exemple, le combinateur I possède dans le contexte vide tous les types $\alpha_1\to\alpha_1\wedge \alpha_2\to\alpha_2 \wedge \ldots \wedge \alpha_n\to\alpha_n$, pour n arbitraire, et ces types ne sont pas en général unifiables en $\alpha\to\alpha$. 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 une substitution 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 C-type principal d'un λ-terme M 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. On passe ainsi du point de vue de typage de Curry (les types désignent des ensembles de λ-termes purs) au point de vue de typage de Church (les λ-termes bien typés définissent une dérivation de leur jugement de typage).

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 $(\alpha\rightarrow\beta)$ 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 $\alpha_1\rightarrow\alpha_1$ et $\alpha_2\rightarrow\alpha_2$ de son type principal $\alpha\rightarrow\alpha$. 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 :

$\Gamma''[x : \tau_1 \wedge \tau_2 \wedge \ldots \wedge \tau_n]) \vdash M : \tau'$, 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∀ la discipline correspondante de typage, intermédiaire entre C et D.

On suppose maintenant qu'une composante de contexte Γ_x associe à la variable x un type <u>générique</u> : $[x:\forall\alpha_1\ \alpha_2 ...\ \alpha_k.\tau]$. Les variables libres de Γ_x sont les variables de τ distinctes des α_i. Les variables libres d'un contexte Γ sont les variables libres dans une composante Γ_x.

Algorithme de synthèse de type dans C∀.

Soit M un terme à typer dans un contexte Γ. On dit que M est C∀-typable modulo Γ si l'algorithme réussit en retournant une substitution σ (pour les variables libres de Γ), 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 $\sigma(\Gamma) \vdash M : \sigma(\alpha) \rightarrow \tau$. 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 $\alpha_1\ \alpha_2 ...\ \alpha_k$ les variables de τ non libres dans $\sigma(\Gamma)$, $\mathfrak{T} = \forall\alpha_1\ \alpha_2 ...\ \alpha_k.\tau$, et $\Gamma' = \sigma(\Gamma)[x:\mathfrak{T}]$ Si $\rho(\Gamma') \vdash N' : \tau'$, on en déduit $\rho(\sigma(\Gamma)) \vdash M :$

τ'. Sinon l'algorithme échoue.

- Sinon, soit $\rho(\sigma(\Gamma)) \vdash N : \tau'$. Soit α une nouvelle variable de type. Si τ' et $\rho(\tau)\rightarrow\alpha$ 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.

Nous n'explicitons pas exactement la notion de nouvelle variable ci-dessus. Une nouvelle variable est a priori une variable non libre dans le contexte courant, mais on veut pouvoir faire cette opération sans pour cela parcourir ce contexte. De même on veut représenter l'information du contexte de manière à localiser au mieux l'opération de substitution aux variables libres du contexte. Différentes formalisations des variables génériques ont été proposées pour résoudre ce problème.

Exercice. Calculer la taille du C∀-type du terme :
let x_0 = [y,z](z y y) in
let x_1 = [y](y (y x_0)) in
...
let x_n = [y](y (y x_{n-1})) in x_n.

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 avec le type principal. 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 a τ

pour C-type principal, 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 par contre CR n'est pas vraie avec la règle supplémentaire exprimant la surjectivité de l'opérateur de paire :

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

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

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

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

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

On peut montrer que sur les termes bien typés, Church-Rosser est vrai, même avec la règle SP.

Finalement, il n'y a pas de difficulté à 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 :

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

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 Cμ∀ permet de typer par exemple le terme suivant, qui n'est pas typable avec Y dans la discipline C∀ :

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

Milner a démontré l'existence d'un type principal dans cette discipline. Malheureusement, Kfoury et al. ont montré que cette discipline de typage est indécidable, ce qui réduit considérablement son intérêt.

3.6. Système T.

On remarque que la règle d'évaluation du point-fixe Y invalide la propriété de terminaison du calcul, ce qui montre que 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 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 Y.

Tout d'abord, on constate que ML⁻ ne permet de programmer que les fonctions polynomiales étendues. 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 :

true : bool false : bool	(bool intro)
if : $\forall\alpha$. bool × α × α → α	(bool elim)
0 : nat S : nat → nat	(nat intro)
R : $\forall\alpha$. nat × (nat→ (α→ α)) × α → α	(nat elim)

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

R(0,f,x) ⇨ x
R(S(n),f,x) ⇨ (f n R(n,f,x))

Exercices.

1. (Kleene) Programmer la fonction prédécesseur.

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

3. Montrer qu'on peut remplacer le récurseur par un itérateur Nat, avec :
Nat : $\forall\alpha$. nat × (α→ α) × α → α
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 : nat × (nat→ (nat→ nat)) × nat → nat.

Problème (Colson). Les résultats de calculabilité classique établissent la définissabilité de fonctions extensionnelles. Une analyse plus fine permet de parler de définissabilité d'algorithmes. Montrer que l'algorithme de calcul du minimum de deux entiers naturels n et m, et faisant un nombre d'opérations min(n,m), n'est pas définissable par l'opérateur de récursion primitive Rnat.

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 :

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

et les règles de calcul :

$\underline{match}(\underline{inl}(x),y,z) \Rightarrow y(x)$ ι_1
$\underline{match}(\underline{inr}(x),y,z) \Rightarrow z(x).$ 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 :
$\underline{match}(x,\underline{inl},\underline{inr}) \Rightarrow x$ 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:
$f(\underline{match}(x,g,h)) = \underline{match}(x,f \circ g,f \circ h)$ 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 $\alpha \times \alpha$ 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 : $\forall\alpha.\ \alpha$ list Cons : $\forall\alpha.\ (\alpha \times \alpha \text{ list}) \rightarrow \alpha$ list

List : $\forall\alpha\beta.\ \alpha \text{ list} \times (\alpha \times \beta \rightarrow \beta) \times \beta \rightarrow \beta$

avec les règles d'itération de liste :

List(Nil,f,x) ⇨ x

List(Cons(a,l),f,x) ⇨ f(a,List(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: Le noyau applicatif de 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$\forall$ à une discipline C$\forall\mu$, autorisant les types récursifs positifs. Etendre à cette discipline le résultat de normalisation forte.

4. Lambda calcul polymorphe.

4.1. Récursivité : les entiers de Church.

Le λ-calcul pur peut être utilisé comme formalisme de calcul universel au sens de Turing. C'est-à-dire qu'on peut coder les fonctions partielles récursives par des λ-termes, et leur calcul par la β-réduction. Le λ-calcul pur est donc un formalisme équivalent pour la définition de fonctions arithmétiques aux systèmes d'équations de Kleene-Gödel, aux machines de Turing, aux systèmes de réécriture de Post, aux algorithmes de Markov, aux grammaires de classe 0 de Thue, etc. Suivant la thèse de Church, le λ-calcul pur permet donc de représenter toutes les fonctions (partielles) calculables d'entiers dans les entiers. Mais il permet naturellement par la même occasion de représenter aussi les fonctionnelles calculables.

Ce résultat remonte à Church, 1936. Church utilisait pour le codage de l'arithmétique les combinateurs donnés p. 13. En particulier, l'entier naturel n est représenté par $\underline{n}$ = [f,x](f (f ... (f x))) avec n occurrences de f. Le booléen Vrai est représenté par **T** = [x,y]x, Faux est représenté par **F** = [x,y]y, l'opérateur de paire est **P** = [x,y,z](z x y), l'expression conditionnelle <u>si</u> p <u>alors</u> x <u>sinon</u> y est représentée par (**P** x y p), l'opérateur de récursion est représenté par un combinateur de point fixe **Y** ou Θ.

On peut considérer ce codage comme la représentation des entiers (en notation unaire) par des itérateurs. Les opérateurs arithmétiques usuels peuvent s'obtenir par les définitions :

$$n+m = [f,x](n\ f\ (m\ f\ x))$$
$$n*m = [f](n\ (m\ f))$$
$$n^m = (m\ n)$$

On laisse en exercice la vérification que $\underline{2}*\underline{2}$ se réduit par β-réduction à la forme normale $\underline{4}$.

On peut se poser le problème de trouver un système de types qui permette de définir un type **N** des entiers de Church autorisant les définitions ci-dessus. Ces définitions suggèrent d'utiliser le polymorphisme, avec **N** = $\forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$. Mais le polymorphisme de

C∀ n'est pas suffisant pour permettre le typage de par exemple [n]n^n. En effet, ce système ne permet pas l'expression d'un type tel que $\mathbf{N}\rightarrow\mathbf{N}\rightarrow\mathbf{N}$, qui nécessite des quantifications à l'intérieur des flèches. Il nous faut pour cela passer à un langage de types plus puissants, permettant la composition arbitraire de flèches et de quantifications. Nous perdrons en passant la propriété de type principal, ce qui nous incite à augmenter les opérations du λ-calcul par un opérateur Λ explicite pour "rendre générique" un algorithme, et son inverse <M τ> qui spécialise un algorithme générique M au type τ. Autrement dit, on abandonne le point de vue de Curry de λ-termes purs typables dans une discipline, pour celui de Church de λ-termes explicitement décorés avec des informations de types, et isomorphes lorsqu'ils sont bien typés à la dérivation de leur jugement de typage.

On obtient ainsi le système F de Girard, redécouvert par Reynolds sous le nom de λ-calcul polymorphe.

4.2. Polymorphisme complet : le système F.

4.2.1. Définition.

Nous allons définir des jugements de typage $\Gamma \vdash M : \tau$, où Γ est un contexte comprenant deux sortes de déclarations : [x:τ], où τ est un type, et [α:type], pour introduire une nouvelle variable de type α. Comme les types ne sont pas définis une fois pour toutes, mais dépendent des déclarations de variables de type, on se donne des règles de construction de contexte, en notant $\vdash \Gamma$ pour "le contexte Γ est bien formé". Le contexte vide est noté [].

Les règles de formation de contexte sont :

$$\vdash []$$

$$\vdash \Gamma \Rightarrow \vdash \Gamma[\alpha : \text{type}]$$

$$\vdash \Gamma \ \& \ \Gamma \vdash \sigma : \text{type} \Rightarrow \vdash \Gamma[x : \sigma]$$

Les règles de formation de types sont :

$$\vdash \Gamma \ \& \ [\alpha : \text{type}] \in \Gamma \ \Rightarrow \ \Gamma \vdash \alpha : \text{type}$$
$$\Gamma \vdash \sigma : \text{type} \ \& \ \Gamma \vdash \tau : \text{type} \ \Rightarrow \ \Gamma \vdash \sigma \to \tau : \text{type}$$
$$\Gamma\,[\alpha : \text{type}] \vdash \tau : \text{type} \ \Rightarrow \ \Gamma \vdash \forall\alpha.\tau : \text{type}$$

et les règles de formation de termes sont :

$$\vdash \Gamma \ \& \ [x : \alpha] \in \Gamma \ \Rightarrow \ \Gamma \vdash x : \alpha$$
$$\Gamma \vdash \sigma : \text{type} \ \& \ \Gamma\,[x : \sigma] \vdash M : \tau \ \Rightarrow \ \Gamma \vdash [x : \sigma]M : \sigma \to \tau$$
$$\Gamma \vdash M : \sigma \to \tau \ \& \ \Gamma \vdash N : \sigma \ \Rightarrow \ \Gamma \vdash (M\,N) : \tau$$
$$\Gamma\,[\alpha : \text{type}] \vdash M : \tau \ \Rightarrow \ \Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau$$
$$\Gamma \vdash M : \forall\alpha.\tau \ \& \ \Gamma \vdash \sigma : \text{type} \ \Rightarrow \ \Gamma \vdash \langle M\,\sigma\rangle : \tau\{\alpha \leftarrow \sigma\}$$

On laisse au lecteur la vérification de la proposition suivante.

Proposition 1. $\Gamma \vdash M : \tau$ entraîne $\Gamma \vdash \tau : \text{type}$.

On munit ce calcul de deux règles de réduction :

$$([x : \tau]M\ N) \Rightarrow M\{x \leftarrow N\} \quad (\beta)$$
$$\langle \Lambda\alpha.M\ \sigma\rangle \Rightarrow M\{\alpha \leftarrow \sigma\} \quad (\beta')$$

On étend la réduction $\Rightarrow$ par congruence comme pour le λ-calcul pur. De même que pour les systèmes précédents, on montre sans difficulté que le calcul préserve le typage :

Proposition 2. $\Gamma \vdash M : \tau$ et $M \Rightarrow N$ entraîne $\Gamma \vdash N : \tau$.

Exemples.

L'identité polymorphe **Id** = $\Lambda\alpha.[x{:}\alpha]x$ a pour type **1** = $\forall\alpha.\ \alpha \to \alpha$ dans le contexte vide. C'est la seule forme normale de ce type.

Le type **B** = $\forall\alpha.\ \alpha \to \alpha \to \alpha$ contient deux formes normales, qui sont **T** = $\Lambda\alpha.[x,y{:}\alpha]x$ et **F** = $\Lambda\alpha.[x,y{:}\alpha]y$.

Le type **N** = $\forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$ contient toutes les formes normales $\underline{\mathbf{n}} = \Lambda\alpha.[f{:}\alpha \to \alpha][x{:}\alpha](f\ (f\ \ldots\ x))$. Il contient aussi $\underline{\mathbf{1}}^* = \Lambda\alpha.[f{:}\alpha \to \alpha]f$, ce qui incite à considérer également une règle de conversion analogue à η, et à se restreindre aux formes normales η-longues.

4.2.2. Normalisation forte.

Théorème (Girard). Le λ-calcul polymorphe muni de la réduction ⇨ est fortement normalisable.

Nous ne donnons pas la preuve de ce théorème. La méthode est la même que celle employée pour le système D. On interprète un terme M du système par le λ-terme pur I(M) obtenu en enlevant les décorations de types :

$$I(x) = x$$
$$I([x : \sigma]M) = [x]I(M)$$
$$I((M\ N)) = (I(M)\ I(N))$$
$$I(\Lambda\alpha.M) = I(M)$$
$$I(<M\ \sigma>) = I(M)$$

Soit Sat la classe des sous-ensembles de $\mathbf{N}^*$ qui sont $\mathbf{N}^*$-saturés. On interprète un type τ défini dans un environnement de variables de types $\alpha_1, \alpha_2, \ldots, \alpha_n$ par un ensemble de λ-termes $I_E(\tau)$, avec $E=I_1 * I_2 * \ldots * I_n$ produit de n ensembles dans Sat. On interprète la quantification de types comme un produit :

$$I_E(\forall\alpha.\tau) = \cap_{S\in Sat}\, I_{E*S}(\tau).$$

On montre que pour tout τ et E on a $I_E(\tau)\in$ Sat. On montre alors, comme pour le système D, que si $\Gamma \vdash M : \tau$, alors $I(M)\in I_E(\tau)\in$ Sat, d'où $I(M)\in \mathbf{N}^*$. Comme β' seul est fortement normalisable, et qu'à une réduction β de M correspond une réduction β de I(M), on en déduit le théorème.

On dit qu'un λ-terme pur L est typable dans la discipline F s'il existe un terme M, un contexte Γ et un type τ du système F, tels que $\Gamma \vdash M : \tau$, avec L = I(M). La preuve précédente montre que si L est typable dans la discipline F, alors L est fortement normalisable au sens de la β-réduction.

Exemples.

Soit distr_pair = [f,x,y,z](z (f x) (f y)). Cet algorithme n'est pas typable dans la discipline C∀, mais il est typable dans la discipline F, avec le type : $(\forall\alpha.\ (\alpha \to \alpha)) \to \alpha \to \beta \to (\alpha \to \beta \to \gamma) \to \gamma$.

Le terme Δ = [x](x x) est typable dans la discipline F, car :

$$[] \vdash [x{:}\mathbf{1}](<x\ \mathbf{1}>\ x) : \mathbf{1}\to\mathbf{1}$$

Corollaire. Le type $\mathbf{0} = \forall\alpha.\alpha$ n'est pas habité : il n'existe pas de terme

fermé M tel que $\vdash M : \mathbf{0}$. En effet, on montre facilement qu'il n'existe pas de forme normale de ce type.

Exercice. Montrer que tout terme normal clos est typable dans la discipline F.

4.2.3. Types de données.

Les types de données algébriques usuels sont définissables comme types du système. Nous avons vu plus haut le cas des booléens **B** et des entiers naturels **N** :

$$\mathbf{B} = \forall\alpha.\ \alpha \to \alpha \to \alpha$$

a deux constructeurs :

$\mathbf{F} = \Lambda\alpha.[f,t:\alpha]f \quad : \mathbf{B}$

et $\mathbf{T} = \Lambda\alpha.[f,t:\alpha]t \quad : \mathbf{B}.$

$$\mathbf{N} = \forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$$

a deux constructeurs :

$\underline{\mathbf{0}} = \Lambda\alpha.[s : \alpha \to \alpha][z : \alpha]z \quad : \mathbf{N}$

et $\underline{\mathbf{S}} = [n : \mathbf{N}]\ \Lambda\alpha.[s : \alpha \to \alpha][z : \alpha](s\ (<n\ \alpha>\ s\ z)) \quad : \mathbf{N} \to \mathbf{N}.$

Dans un environnement où α:type, on peut former le type des listes d'éléments de type α :

$$\mathbf{List} = \forall\beta.\ (\alpha \to \beta \to \beta) \to (\beta \to \beta)$$

On laisse en exercice la définition des constructeurs **Nil** et **Cons**.

Voici une manière de définir des ordinaux, ou plutôt des notations ordinales.

$$\mathbf{Ord} = \forall\alpha.\ ((\mathbf{N} \to \alpha) \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha)$$

Les constructeurs de ce type sont les analogues de $\underline{\mathbf{0}}$ et de $\underline{\mathbf{S}}$ du type **N**, plus un nouveau constructeur qui associe à une suite d'ordinaux une notation pour sa limite :

$\underline{\mathbf{Lim}} = [\sigma : \mathbf{N} \to \mathbf{Ord}]\ \Lambda\alpha.[l:(\mathbf{N}\to\alpha)\to\alpha][s:\alpha \to \alpha][z:\alpha](l\ [n:\mathbf{N}](<(\sigma\ n)\ \alpha>\ l\ s\ z))$

$\quad : (\mathbf{N} \to \mathbf{Ord}) \to \mathbf{Ord}.$

4.2.4. Correspondance logique.

Certains systèmes de type admettent une interprétation logique ; c'est ce qu'on appelle la correspondance de Curry-Howard. Dans cette correspondance, les types deviennent des propositions, les règles de typage des systèmes d'inférence logiques, et les termes des notations pour les preuves.

Par exemple, le système C correspond à la logique minimale présentée sous forme de déduction naturelle. Les seules propositions sont des implications, l'abstraction est la règle $\Rightarrow$-intro, l'application est la règle $\Rightarrow$-élim. La β-réduction correspond à une coupure dans la démonstration, et le résultat de normalisation correspond à l'élimination des coupures.

De même ici, le système F correspond à une logique propositionnelle de 2nd ordre. On appelle d'ailleurs quelquefois ce système le λ-calcul de 2nd ordre. Le théorème de Girard correspond à un résultat d'élimination des coupures. Son corollaire donne la cohérence de la logique, le type **0** correspondant à la proposition absurde.

Les autres connectives de la logique intuitionniste sont définissables par des formules de 2nd ordre. Ainsi, si P et Q sont des propositions, on peut définir leur conjonction $P \wedge Q$ comme $\forall A.\ (P \Rightarrow Q \Rightarrow A) \Rightarrow A$. Ceci correspond à définir le produit de types $\beta \times \gamma = \forall\alpha.(\beta \to \gamma \to \alpha) \to \alpha$. De même la disjonction intuitionniste correspond à la somme de types $\beta + \gamma = \forall\alpha.(\beta \to \alpha) \to (\gamma \to \alpha) \to \alpha$.

4.3. Types d'ordre supérieur : le système Fω.

Nous avons vu comment définir le schéma qui fait correspondre aux types β et γ le type $\beta \times \gamma$. Ceci ne nous permet pas pour autant de définir de manière interne le constructeur de types $\times$. Ni de généraliser l'abstraction de type sur les termes en une abstraction de fonction ou fonctionnelle de type. Le système Fω (aussi appelé λω) permet cette généralisation de F à l'ordre supérieur.

L'idée est simple : on généralise la sorte type à un système d'arités, construites à partir de type et d'une flèche $\to$ exprimant la fonctionnalité des opérateurs de type. On introduit une nouvelle sorte des arités, avec :

$$\text{type} : \text{arité}$$
$$a : \text{arité} \ \& \ b : \text{arité} \Rightarrow a \to b : \text{arité}$$

N.B. le mot "arité" correspond à l'anglais "kind".

Les contextes comprennent maintenant des déclarations $[x : \sigma]$, avec σ un type, et $[\alpha : a]$, avec a une arité. On peut maintenant former des types dépendant d'un opérateur α. Le produit dépendant correspondant $(\alpha : a).\tau$ correspond logiquement à une quantification universelle, et l'élimination correspondante est une opération d'instanciation $\sigma(\tau)$.

Les règles de formation de contexte sont :

$$\vdash []$$
$$\vdash \Gamma \ \& \ a : \text{arité} \Rightarrow \vdash \Gamma[\alpha : a]$$
$$\vdash \Gamma \ \& \ \Gamma \vdash \sigma : \text{type} \Rightarrow \vdash \Gamma[x : \sigma]$$

Les règles de formation d'opérateurs de types sont :

$$\vdash \Gamma \ \& \ [\alpha : a] \in \Gamma \Rightarrow \Gamma \vdash \alpha : a$$
$$\Gamma \vdash \sigma : \text{type} \ \& \ \Gamma \vdash \tau : \text{type} \Rightarrow \Gamma \vdash \sigma \to \tau : \text{type}$$
$$\Gamma[\alpha : a] \vdash \tau : \text{type} \Rightarrow \Gamma \vdash (\alpha : a).\tau : \text{type}$$
$$\Gamma[\alpha : a] \vdash \tau : b \Rightarrow \Gamma \vdash \lambda\alpha : a.\ \tau : a \to b$$
$$\Gamma \vdash \sigma : a \to b \ \& \ \Gamma \vdash \tau : a \Rightarrow \Gamma \vdash \sigma(\tau) : b$$

et les règles de formation de termes sont :

$$\vdash \Gamma \ \& \ [x : \sigma] \in \Gamma \Rightarrow \Gamma \vdash x : \sigma$$
$$\Gamma \vdash \sigma : \text{type} \ \& \ \Gamma[x : \sigma] \vdash M : \tau \Rightarrow \Gamma \vdash [x : \sigma]M : \sigma \to \tau$$
$$\Gamma \vdash M : \sigma \to \tau \ \& \ \Gamma \vdash N : \sigma \Rightarrow \Gamma \vdash (M\ N) : \tau$$
$$\Gamma[\alpha : a] \vdash M : \tau \Rightarrow \Gamma \vdash [\alpha : a]M : (\alpha : a).\tau$$
$$\Gamma \vdash M : (\alpha : a).\tau \ \& \ \Gamma \vdash \sigma : a \Rightarrow \Gamma \vdash \langle M\ \sigma \rangle : \tau\{\alpha \leftarrow \sigma\}$$

On obtient la quantification de types comme cas particulier de produit dépendant, en définissant l'abréviation $\forall\alpha.\tau = (\alpha{:}\text{type}).\tau$. De façon analogue on retrouve la formation de terme polymorphe comme un cas particulier de l'abstraction d'opérateur : $\Lambda\alpha.M = [\alpha : \text{type}]M$.

Example. On peut maintenant définir le produit de types par :

$$\times = \lambda\beta : \text{type}.\ \lambda\gamma : \text{type}.\forall\alpha.(\beta \to \gamma \to \alpha) \to \alpha$$
$$: \text{type} \to \text{type} \to \text{type}$$

On peut maintenant faire des calculs sur les types, avec une règle supplémentaire correspondant à la β-réduction des opérateurs :

$$(\lambda\alpha : a.\ \sigma)(\tau) \supset \sigma\{\alpha \leftarrow \tau\} \quad (\beta o)$$

On étend la relation $\supset$ par congruence sur les opérateurs, ce qui donne une relation d'équivalence $\approx$. On se donne finalement une règle de quotient de cette relation d'équivalence, permettant de faire la coercion d'un terme ayant pour type un opérateur :

$$\Gamma \vdash M : \sigma \ \&\ \Gamma \vdash \sigma : a \ \&\ \sigma \approx \tau \ \Rightarrow\ \Gamma \vdash M : \tau$$

Exemples. Avec le produit de types défini ci-dessus, on peut maintenant programmer l'algorithme de 1ère projection, par :

$$fst = [\alpha:\text{type}][\beta:\text{type}][p:\alpha\times\beta](<p\ \alpha>\ [x:\alpha][y:\beta]x)$$
$$: (\alpha:\text{type}).(\beta:\text{type}).\ \alpha\times\beta \to \alpha$$

Reprenons distr_pair = [f,x,y,z](z (f x) (f y)). Nous avons vu plus haut que ce terme est typable dans la discipline F, avec le type :

$$(\forall\alpha.\ (\alpha \to \alpha)) \to \alpha \to \beta \to (\alpha \to \beta \to \gamma) \to \gamma.$$

De façon plus générale, il est typable dans la discipline Fω, de type :

$$(\tau : \text{type}\to\text{type}).((\alpha:\text{type}).\ \alpha \to \tau(\alpha)) \to \alpha \to \beta \to (\tau(\alpha) \to \tau(\beta) \to \gamma) \to \gamma.$$

Ce type est la forme "paramétrique" du type de D "ad-hoc" :

$$((\alpha \to \alpha') \wedge (\beta \to \beta')) \to \alpha \to \beta \to (\alpha' \to \beta' \to \gamma) \to \gamma.$$

Comme pour le système F, on se donne les règles de calcul de termes :

$$([x : \tau]M\ N) \Rightarrow M\{x \leftarrow N\} \quad (\beta)$$
$$<[\alpha : a]M\ \sigma> \Rightarrow M\{\alpha \leftarrow \sigma\} \quad (\beta')$$

On étend la réduction $\Rightarrow$ par congruence sur les termes.

Le théorème de normalisation forte s'étend à ce calcul :

Théorème. λω muni de $\Rightarrow$ est fortement normalisable.

Exercice (Giannini & Ronchi). Montrer que le terme :
([x,y](y (x **I**)(x **K**)) Δ) est typable dans λω, mais non typable dans F.

5. Disciplines de types dépendants.

Dans la discipline F, nous avons vu comment faire dépendre les termes des types. Dans la discipline P, nous allons maintenant faire dépendre les types des termes.

5.1. Correspondance de Curry-Howard.

Dans cette correspondance, on introduit la quantification universelle au sens de la logique de premier ordre.

5.2. Quantification universelle : produits dépendants.

On décrit un système P, dual du système F vu précédemment : le rôle de type et d'arité sont inversés.

Les types généralisent ceux du système C : le type fonctionnel $\sigma \to \tau$ est généralisé en un produit dépendant $(x : \sigma).\tau$. On doit donc remplacer le jugement σ : type par un jugement indicé sur le contexte $\Gamma \vdash \sigma$: type. On garde la notation $\sigma \to \tau$ pour $(x : \sigma).\tau$ dans le cas où $x \notin \tau$, et de même on note $\sigma \to a$ pour l'arité dépendante $(x : \sigma).a$, dans le cas où $x \notin a$.

Règles de formation de contexte :

$$\vdash []$$
$$\vdash \Gamma \ \& \ \Gamma \vdash \sigma : \text{type} \ \Rightarrow \vdash \Gamma[x : \sigma]$$
$$\vdash \Gamma \ \& \ \Gamma \vdash a : \text{arité} \ \Rightarrow \vdash \Gamma[\alpha : a]$$

Règles de formation de types et d'arités :

$$\vdash \Gamma \Rightarrow \Gamma \vdash \text{type} : \text{arité}$$
$$\vdash \Gamma \ \& \ [\alpha : a] \in \Gamma \Rightarrow \Gamma \vdash \alpha : a$$
$$\Gamma[x : \sigma] \vdash a : \text{arité} \Rightarrow \Gamma \vdash (x : \sigma).a : \text{arité}$$
$$\Gamma[x : \sigma] \vdash \tau : \text{type} \Rightarrow \Gamma \vdash (x : \sigma).\tau : \text{type}$$
$$\Gamma[x : \sigma] \vdash \tau : a \Rightarrow \Gamma \vdash [x : \sigma]\tau : (x : \sigma).a$$
$$\Gamma \vdash \tau : (x : \sigma).a \ \& \ \Gamma \vdash N : \sigma \Rightarrow \Gamma \vdash \langle \tau \, N \rangle : a\{x \leftarrow N\}$$

Les règles de formation de termes sont celles de C :

$$\vdash \Gamma \ \& \ [x:\alpha] \in \Gamma \ \Rightarrow \ \Gamma \vdash x:\alpha$$
$$\Gamma[x:\sigma] \vdash M:\tau \ \Rightarrow \ \Gamma \vdash [x:\sigma]M : (x:\sigma).\tau$$
$$\Gamma \vdash M:(x:\sigma).\tau \ \& \ \Gamma \vdash N:\sigma \ \Rightarrow \ \Gamma \vdash (M\ N):\tau\{x \leftarrow N\}$$

Il y a maintenant deux relations de β-réduction, au niveau des types et au niveau des termes, et on étend cette relation $\Rightarrow$ en une congruence $\approx$ sur les termes, les types, et les arités. On se donne finalement des règles de coercion comme suit :

$$\Gamma \vdash M:\sigma \ \& \ \Gamma \vdash \tau : \text{type} \ \& \ \sigma \approx \tau \ \Rightarrow \ \Gamma \vdash M:\tau$$
$$\Gamma \vdash \sigma : a \ \& \ \Gamma \vdash b : \text{arité} \ \& \ a \approx b \ \Rightarrow \ \Gamma \vdash \sigma : b$$

Exercice. Montrer que la relation de calcul préserve les types.

Exemple.

On peut dans ce système exprimer la signature de l'arithmétique par le contexte Arith_sign = [nat:type][0:nat][S:nat→nat][Eq:nat→nat→type]

et par exemple construire le prédicat d'égalité à zéro par :

Arith_sign ⊢ [x : nat](Eq n 0) : nat→type

On peut également enrichir le contexte Arith_sign en se donnant des axiomes, comme par exemple la réflexivité de l'égalité :

[refl:(x:nat).(Eq x x)]

et on peut vérifier dans ce contexte étendu que (refl 0) : (Eq 0 0).

Mais on n'a pas dans ce cas de moyen direct de représenter dans ce système les connectives logiques, ni les règles d'inférence, car on n'a pas ici d'opérateur de type. On peut toutefois axiomatiser une logique, en déclarant un type des propositions, des connectives logiques, des quantificateurs, et des règles d'inférence.

Par exemple, on peut axiomatiser la logique intuitionniste de 1er ordre par le contexte:

Pos_logic = [D : type][d : D][prop : type][T : prop→type]
[⇒ : prop→prop→prop][∀ : (D→prop)→prop]
[⇒_intro : (A:prop).(B:prop).((T A)→(T B))→ (T (⇒ A B))]
[⇒_elim : (A:prop).(B:prop).(T (⇒ A B))→ (T A)→ (T B)]
[∀_intro : (P:D→prop).((x:D).(T (P x)))→(T (∀ [x:D](P x)))]

[∀_elim : (P:D→prop).(T (∀ [x:D](P x)))→(x:D).(T (P x))]

Le type D représente le domaine d'interprétation, qui est postulé non vide par la déclaration de l'élément d. Le type prop est celui des propositions, construites avec la connective d'implication ⇒ et le quantificateur universel ∀. La proposition usuellement écrite ∀x.P(x) s'écrit ici (∀ [x:D](P x)). Le constructeur de type T correspond au jugement qui associe à une proposition P le type de ses preuves (T P). On utilise ainsi l'analogie jugements=types plutôt que propositions=types. Remarquez comment on utilise les types dépendants pour déclarer naturellement les règles de quantification.

Exercice. Combinez les idées des contextes Arith_sign et Pos_logic pour définir un contexte correspondant à l'arithmétique de Heyting. Contrairement à plus haut, l'égalité doit être une relation (de type nat→nat→prop). Le schéma de récurrence peut-il s'exprimer par une notion générique unique?

Un système voisin de P est le "Logical Framework" LF. Il s'en distingue en séparant les "signatures" des "contextes". Toutes les déclarations d'opérateurs [α : a] dans LF doivent être dans la partie "signature", qui peut être vue comme segment initial d'un contexte de P. De plus, LF admet la règle η. Cette règle sert dans la preuve que le codage d'une logique en LF est fidèle.

D'autres systèmes voisins sont ceux de la famille Automath, qui seront décrits plus loin comme des GTS.

La metathéorie des systèmes P, LF, et Automath est complexe. La normalisation forte de P se prouve par un résultat de conservativité avec le système C, obtenu en considérant la traduction suivante de P dans C.

On traduit tout type et arité de P en un type simple, formé à partir du type atomique ι et de →, par la traduction ρ définie comme suit :

$$\rho(\text{type}) = \iota$$
$$\rho((x : \sigma).a) = \rho(\sigma) \to \rho(a)$$
$$\rho(\alpha) = \iota$$
$$\rho((x : \sigma).\tau) = \rho(\sigma) \to \rho(\tau)$$
$$\rho([x : \sigma]\tau) = \rho(\tau)$$

$$\rho(<\tau\ N>) = \rho(\tau)$$

Maintenant, on se donne une traduction θ des types et des termes de P dans le λ-calcul pur muni d'une constante π, comme suit :

$$\theta(\alpha) = \alpha$$
$$\theta((x : \sigma).\tau) = (\pi\ \theta(\sigma)\ \theta(\tau))$$
$$\theta([x : \sigma]\tau) = ([\alpha][x]\theta(\tau)\ \theta(\sigma))$$
$$\theta(<\tau\ N>) = (\theta(\tau)\ \theta(N))$$
$$\theta(x) = x$$
$$\theta([x : \sigma]M) = ([\alpha][x]\theta(M)\ \theta(\sigma))$$
$$\theta((M\ N)) = (\theta(M)\ \theta(N))$$

Lemme. $\Gamma \vdash_P \sigma : a \Rightarrow \rho(\Gamma) \vdash_C \theta(\sigma) : \rho(a)$
$\Gamma \vdash_P M : \sigma \Rightarrow [\pi : \iota \to \iota \to \iota]\ \rho(\Gamma) \vdash_C \theta(M) : \rho(\sigma)$

Théorème. $\sigma \Rightarrow \tau \Rightarrow \theta(\sigma) \Rightarrow^+ \theta(\tau)$
$M \Rightarrow N \Rightarrow \theta(M) \Rightarrow^+ \theta(N)$

Corollaire. Tout terme bien typé dans la discipline P est fortement normalisable.

Cette méthode de preuve est issue des travaux sur LF de Harper, Honsell et Plotkin.

On voit donc que la discipline P, si elle donne plus d'expressivité que C, ne donne pas pour autant plus de puissance de calcul. En particulier, les fonctions arithmétiques définissables dans P sont les mêmes que dans C.

5.3. Somme dépendante, Théorie intuitionniste des types de Martin-Löf.

On peut augmenter le système P avec une opération de somme dépendante. De même que le cas non-dépendant du produit correspond à la flèche $\to$, le cas non-dépendant de la somme correspondra au produit de types $\times$ (d'où une confusion terminologique regrettable).

On ajoute donc à P les règles :

$$\Gamma[x : \sigma] \vdash \tau : \text{type} \Rightarrow \Gamma \vdash \{x : \sigma\}.\tau : \text{type}$$
$$\Gamma[x : \sigma] \vdash \tau : \text{type} \ \& \ \Gamma \vdash M : \sigma \ \& \ \Gamma \vdash N : \tau\{x \leftarrow M\} \Rightarrow \Gamma \vdash (M,N) : \{x : \sigma\}.\tau$$
$$\Gamma \vdash P : \{x : \sigma\}.\tau \ \& \ \Gamma[x : \sigma][y : \tau] \vdash Q : \rho[(x,y)] \Rightarrow \Gamma \vdash P \ \underline{\text{as}} \ (x,y).Q : \rho[P]$$

La dernière construction "P as (x,y).Q" correspond à peu près au CAML :

$$\underline{\text{match}} \ P \ \underline{\text{with}} \ (x,y) \rightarrow Q$$

en particulier, les variables x et y sont liées dans Q.

L'interprétation logique du type $\{x : \sigma\}.\tau$ est la proposition quantifiée existentielle $\exists x : \sigma.\ \tau$.

On complète le système, appelons-le PS, par la règle de calcul :

$$(M,N) \ \underline{\text{as}} \ (x,y).Q \Rightarrow Q\{x \leftarrow M, y \leftarrow N\}$$

Remarque.
Le système PS est plus expressif que le calcul des prédicats du premier-ordre, avec les quantificateurs usuels. En effet, on peut montrer dans ce système une version de l'axiome du choix, comme le montre l'exercice suivant.

Exercice.
Montrer que dans un contexte Γ tel que $\Gamma \vdash P : (x : \sigma).(y : \tau).\text{type}$, il existe un terme AC de PS tel que :

$$\Gamma \vdash AC : (x : \sigma).\{y : \tau\}.(P\ x\ y) \rightarrow \{f : (x : \sigma)\ \tau\}.(x : \sigma).(P\ x\ (f\ x)).$$

Le système PS, complété par des types union, égalité, et des types inductifs, est la base de la théorie intuitionniste des types de Martin-Löf, qui sort du cadre de ces notes. Signalons seulement que la plupart des résultats meta-théoriques de P s'étendent à PS.

Remarquons que le système PS n'admet pas l'unicité de types dans la présentation ci-dessus. En effet, dans la paire (M,N) : $\{x : \sigma\}.\tau$, le type τ n'est pas explicité, et il n'est pas déterminé de manière unique de $\tau\{x \leftarrow M\}$. Ainsi, avec des types union bool et nat, et ϕ l'opérateur qui envoie 0 sur bool et les entiers positifs sur nat, la paire d'entiers (1,1) admet le type $\{x : \text{nat}\}.\text{nat}$, mais aussi le type $\{x : \text{nat}\}(\phi\ x)$, et ces deux types ne sont pas convertibles. Ce problème suggère une notation plus explicite pour l'opération de paire, telle que $\underline{\text{let}}\ x=M\ \underline{\text{in}}\ N:\tau$. Cette notation suggère que la

variable x pourrait être liante non seulement en τ, mais aussi en N.

Plusieurs extensions de cette dernière idée sont possibles. La première est de remplacer l'idée d'une variable muette x par celle d'un champ nommé par un identificateur x. Ceci aurait l'avantage de rapprocher la notion de valeur de type somme de la notion de "record" informatique, mais aurait l'inconvénient d'exiger une gestion explicite des noms, le type $\{x : \sigma\}.\tau$ étant maintenant différent du type $\{y : \sigma\}.\tau\{x \leftarrow y\}$. Cette proposition est cohérente, par dualité, à donner les noms des sélecteurs avec un type union, distinguant le type bool = true | false du type couleur = rouge | noir. La deuxième extension possible est de donner un statut tout à fait élémentaire à une notion interne de calcul de substitutions, dont le constructeur let x=M in N serait autorisé au niveau des arités également. La meta-notation $N\{x \leftarrow M\}$ deviendrait superflue, et la β-réduction s'exprimerait simplement par $([x : \tau]M\ N) \Rightarrow$ let x=N in M. Le calcul des substitutions serait le mécanisme de base de l'égalité définitionnelle, et non plus la β-réduction. La troisième extension suggérée est de faire jouer un rôle symétrique aux sommes et aux produits vis à vis de la formation des contextes, et donc d'autoriser des déclarations d'égalité dans les contextes. Ceci aurait l'avantage de faire rentrer dans le formalisme la notion de constante définie et de lemme, notions mathématiques essentielles.

Dans les présentations récentes de la théorie de Martin-Löf, le système PS est présenté dans le cadre d'une théorie des arités similaire à LF. La règle η est admise au niveau des arités, mais seule la règle β est admise au niveau des types. Les sortes set et type sont utilisées respectivement à la place de nos sortes type et arité.

Signalons le lien étroit entre la théorie de Martin-Löf enrichie de types inductifs bool et nat et le système T étudié plus haut. On peut considérer la théorie de Martin-Löf comme l'extension naturelle du système T avec des types dépendants. Un opérateur de projection similaire à θ plus haut permet de montrer que les fonctions de type $nat \rightarrow nat$ de la théorie sont les mêmes que celles du système T, c'est à dire les fonctions prouvablement totales dans l'arithmétique de Peano.

6. Systèmes de types généralisés

Les systèmes fonctionnels étudiés jusqu'à présent ont beaucoup de traits en commun. Leur noyau commun, le système C, permet de faire "dépendre un terme M (de type τ) d'un terme x (de type σ)" par la formation de $[x : \sigma]M$ (de type $\sigma \to \tau$). Le système F permet de faire "dépendre un terme M (de type τ) d'un type α" par la formation de $\Lambda\alpha.M$ (de type $\forall\alpha.\tau$). Le système P permet de faire "dépendre un constructeur de type τ (d'arité a) d'un terme x (de type σ)" par la formation de $[x : \sigma]\tau$ (d'arité $(x : \sigma).a$). Finalement, le système $F\omega$ permet de faire "dépendre un constructeur de type τ (d'arité b) d'un constructeur de type α (d'arité a) par la formation de $\lambda\alpha : a.\ \tau$ (d'arité $a \to b$).

On peut reconnaître toutes ces constructions comme des opérations d'abstractions, formant des familles sur une sorte indicée par une autre sorte. Les systèmes ci-dessus se décrivent à l'aide des deux sortes type et arité. L'opération de famille indicée donnera des produits dépendants ou des quantifications, suivant l'interprétation ensembliste ou logique de la sorte considérée. Cette idée de traitement uniforme d'un système fonctionnel défini sur des sortes est due à Berardi et Terlouw, sous le nom de "Generalised Type Systems".

6.1. GTS : Définition

Un système de types généralisé, ou **GTS**, est présenté par les données suivantes:

* Un ensemble **C** de constantes atomiques.
* Un sous-ensemble **S** de C : les sortes.
* Un ensemble **A** d'axiomes de la forme c:s avec $c \in \mathbf{C}$ et $s \in \mathbf{S}$.
* Un ensemble **R** de règles de la forme (s_1, s_2, s_3), avec $s_1, s_2, s_3 \in \mathbf{S}$.

Les termes du **GTS** sont formés des constantes, de variables, d'une opération d'application (M N), et de deux opérations de liaison, l'abstraction [x : M]N et le produit (x : M)N. Les contextes sont des suites de déclarations [x : A], où A est un terme qui doit être typable par une sorte dans le reste du contexte. La règle de β-réduction est admise (comme précédemment, sur les radicaux ([x : A]M N)). La congruence associée est notée $\approx$. Un jugement de typage est de la forme $\Gamma \vdash M : A$, où Γ

est un contexte, M et A sont des termes.

Le **GTS** défini par **A** et **R** est présenté par les règles de jugements de typage suivantes :

$$c{:}S \in \mathbf{A} \Rightarrow [] \vdash c : S$$

$$\Gamma \vdash A : S \Rightarrow \Gamma[x : A] \vdash x : A$$

$$\Gamma \vdash M : B \;\&\; \Gamma \vdash A : S \Rightarrow \Gamma[x : A] \vdash M : B$$

$$\Gamma \vdash M : (x : A)B \;\&\; \Gamma \vdash N : A \Rightarrow \Gamma \vdash (M\,N) : B\{x \leftarrow N\}$$

$$\Gamma \vdash M : A \;\&\; \Gamma \vdash B : S \;\&\; A \approx B \Rightarrow \Gamma \vdash M : B$$

$$\Gamma \vdash A : S_1 \;\&\; \Gamma[x : A] \vdash B : S_2 \Rightarrow \Gamma \vdash (x : A)B : S_3 \;(\dagger)$$

$$\Gamma \vdash A : S_1 \;\&\; \Gamma[x : A] \vdash B : S_2 \;\&\; \Gamma[x : A] \vdash M : B \Rightarrow \Gamma \vdash [x : A]M : (x : A)B \;(\dagger)$$

(†) Dans ces deux dernières règles, on prend toutes les instances de sortes (S_1, S_2, S_3) telles que $(S_1, S_2, S_3) \in \mathbf{R}$.

Le **GTS** est dit <u>fonctionnel</u> si pour tout $c \in \mathbf{C}$ il existe au plus un $s \in \mathbf{S}$ tel que $c{:}s \in \mathbf{A}$, et pour tous $S_1, S_2 \in \mathbf{S}$ il existe au plus un $S_3 \in \mathbf{S}$ tel que $(S_1, S_2, S_3) \in \mathbf{R}$.

Le **GTS** est dit <u>circulaire</u> relativement à la sorte S s'il existe une sorte S' telle que $S{:}S' \in \mathbf{A}$, et $(S', S, S) \in \mathbf{R}$.

Un certain nombre de résultats meta-théoriques se prouvent de manière générique pour tout GTS fonctionnel. Par exemple :

Unicité des types. Dans tout GTS fonctionnel :
$\Gamma \vdash M : A \;\&\; \Gamma \vdash M : B \Rightarrow A \approx B$.

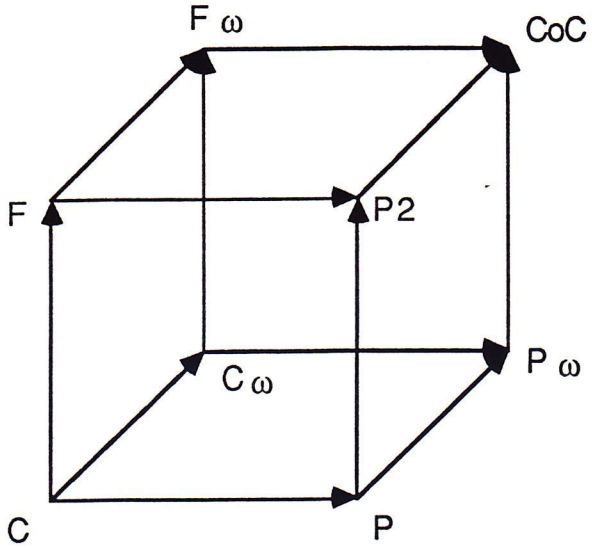
6.2. Le cube de Barendregt

Pour ces premiers exemples de GTS fonctionnels, on prend **C** = **S** = {type, arité}, on a un seul axiome **A** = {type : arité} et les règles sont toutes de la forme (S, S', S'), qu'on abrège en (S, S') :

C. **R** = {(type,type)}
F. **R** = {(type,type), (arité,type)}
P. **R** = {(type,type), (type,arité)}

Cω. **R** = {(type,type), (arité,arité)}
Fω. **R** = {(type,type), (arité,type), (arité,arité)}
P2. **R** = {(type,type), (arité,type), (type,arité)}
Pω. **R** = {(type,type), (type,arité), (arité,arité)}
CoC. **R** = {(type,type), (type,arité), (arité,type), (arité,arité)}

Ces différents systèmes sont résumés dans le "cube de Barendregt", où les flèches dénotent des inclusions :



Le cube de Barendregt

On peut "oublier" la structure le long d'une arête horizontale, ce qui correspond à un principe de réalisabilité. On projette ainsi une preuve dans un système logique (e.g. CoC) sur un terme d'un système fonctionnel (e.g. Fω). Dans cette correspondance, la face de droite des "preuves" enrichit la face de gauche des "programmes". Nous avons vu l'exemple d'une telle projection plus haut, avec la traduction θ de P dans C.

La face inférieure du cube est dite "prédicative", la face supérieure "non-prédicative"; cette notion correspond ici au fait que les systèmes de la face supérieure sont circulaires relativement à la sorte type.

La face antérieure est dite "de premier ordre", la face postérieure est dite "d'ordre supérieur"; cette notion correspond à la possibilité de former des opérateurs de type avec la règle (arité,arité).

Remarques. Afin de ne pas suggérer de connotation inutile, et de

laisser la liberté maximum d'interprétation des sortes, on peut utiliser des symboles moins évocateurs que les noms type et arité. Par exemple, le cube de Barendregt est généralement présenté avec à leur place les sortes plus "neutres" * et ❑. La version "Curry-Howard" de ces GTS tendra à remplacer type et arité par Prop et Type respectivement.

Remarquez que la circularité de la sorte *, pour les systèmes non-prédicatifs, semble dangereuse si on interprète * comme un univers ensembliste, le produit (U:*)V pour V:* devant être "de plus grande cardinalité" que la classe *. Dans la version "Curry-Howard", toutefois, cette circularité semble bénigne, il s'agit juste d'autoriser la formation d'une proposition quantifiée universellement sur une proposition, ce qui est l'essence de la logique d'ordre supérieur.

6.3. Le Calcul des Constructions.

La première version du calcul, définie dans la thèse de T. Coquand (1985), correspond au GTS CoC du cube ci-dessus. On utilise la correspondance de Curry-Howard, et les sortes Prop et Type y remplacent respectivement type et arité. C'est-à-dire :

S = {Prop, Type}
A = {Prop : Type}
R = {(Prop, Prop), (Prop, Type), (Type, Prop), (Type, Type)}

Ce calcul correspond à une version de la logique intuitionniste d'ordre supérieur, présentée comme système de déduction naturelle, avec des termes dénotant les preuves (ce sont les termes M tels que $\Gamma \vdash M : P$, avec $\Gamma \vdash P$: Prop).

Théorème (Coquand). CoC admet la normalisation forte.

Ce théorème permet de prouver la cohérence du système logique, sous la forme suivante. Avec $\bot = (P:Prop)P$, il n'existe pas de preuve de $\bot$ dans le contexte vide, c'est-à-dire de terme Absurde tel que
$\vdash$ Absurde : $\bot$.
Une telle preuve entraînerait bien sûr l'incohérence du système logique, puisque toute proposition P serait prouvable sans hypothèse par le terme de preuve (Absurde P).

Coquand a fait une analyse meta-mathématique du système, où il a montré l'extension à ce calcul des propriétés usuelles (unicité de type, préservation du type par calcul, confluence, etc) et a donné un certain nombre de constructions de modèles. En particulier, un modèle particulièrement simple de "non-pertinence de preuves" interprète Prop comme l'ensemble des booléens {0,1}, et une proposition comme une valeur de vérité. Dans ce modèle $\perp$ vaut 0, ce qui établit la cohérence du système par une méthode élémentaire. Pour mener à bien les détails, toutefois, il convient de distinguer la quantification logique du produit de types, qui a l'interprétation ensembliste usuelle. Ceci suppose que CoC est présenté sous une forme plus explicite, avec un opérateur **T** faisant correspondre à une proposition P le type **T**[P] de ses preuves, dans l'esprit du codage de LF vu en 5.2. Ceci revient à dupliquer les opérations d'abstraction et d'application, au niveau des preuves et au niveau des objets. L'opérateur **T** oublie la structure des preuves, ce qui se traduit dans la sémantique par $\mathbf{T}[0]=\varnothing$ et $\mathbf{T}[1]=\{\varnothing\}$. Ceci donne la liberté d'interpréter différemment les types $(x:A)\mathbf{T}[(P\ x)]$ et $\mathbf{T}[(x:A)(P\ x)]$.

Remarque.

Nous avons vu plus haut que dans le système Fω la somme de types était définissable, par :

$$+ = \lambda\beta : \text{type}.\ \lambda\gamma : \text{type}.\forall\alpha.(\beta \to \alpha) \to (\gamma \to \alpha) \to \alpha$$

Plus généralement, on peut définir dans CoC une quantification existentielle :

$$\exists = \lambda\gamma : \text{Prop}.\ \lambda\beta : (x : \gamma).\text{Prop}.\ (\alpha : \text{Prop}).((x : \gamma)(\beta\ x) \to \alpha) \to \alpha$$

Cet opérateur permet de produire x:X à partir de (∃ X P) (comment?), mais non p:(P x). Cette "somme" est donc intrinsèquement plus faible que la somme du système PS plus haut. On parle de <u>somme faible</u>, par opposition aux <u>sommes fortes</u> des systèmes à la Martin-Löf, où les deux projections sont disponibles. L'équivalent informatique d'une telle somme faible est la notion de type abstrait: un élément de (∃ X P) est la donnée d'un élément x:X vérifiant (P x), mais la justification de cette propriété (le terme p) est encapsulée de manière inaccessible. Coquand a montré que l'extension de CoC avec une somme forte était en fait incohérente. Autrement dit, la circularité et l'existence d'une somme forte sur la même sorte sont incompatibles.

6.4. Le Calcul des Constructions Etendu.

Le Calcul des Constructions Etendu (CCE) possède une hiérarchie d'univers :

S = {Prop, $Type_i$ | i≥0}
A = {Prop : $Type_0$, $Type_i$: $Type_{i+1}$}
R = {(Prop, Prop), (Prop, $Type_i$), ($Type_i$, Prop),($Type_i$, $Type_j$, $Type_{max(i,j)}$)}

Le système est étendu par une règle de cumulativité des univers :

$$\Gamma \vdash T : Type_i \Rightarrow \Gamma \vdash T : Type_{i+1}.$$

Autrement dit, CCE n'est pas un pur GTS, c'est un GTS à sortes ordonnées.

Le théorème de normalisation forte s'étend à CCE. Le système n'a pas la propriété d'unicité de type, à cause de la règle de cumulativité ci-dessus. Mais tout terme typable possède un type minimal, relativement à l'ordre d'inclusion des univers : $Type_i \leq Type_j$ si i≤j.

Il est possible de présenter le système en omettant explicitement les indices d'univers, mais en vérifiant progressivement que les occurrences de la sorte Type utilisées dans une construction vérifient une condition d'acyclicité permettant de garantir qu'une affectation d'indices existe.

Dans CCE on peut développer naturellement un calcul des prédicats d'ordre supérieur. Ainsi, l'idée du quantificateur existentiel ci-dessus s'adapte bien à la définition d'un opérateur de somme faible :

$$\Sigma = [\alpha : Type][P : (x : \alpha).Type]\ (\beta : Type).((x : \alpha).(P\ x) \rightarrow \beta) \rightarrow \beta.$$

Une extension de CCE avec une opération de somme forte au niveau des univers a été présentée, et prouvée cohérente, par Z. Luo.

Le calcul CCE permet l'axiomatisation, et la vérification formelle, d'une grande partie des mathématiques. Mais, avant de discuter de manière plus approfondie de cette question, examinons d'autres exemples, plus simples, de GTS.

6.5. Autres GTS.

6.5.1. Systèmes bien fondés.

On entend par là des systèmes qui admettent la normalisation forte.

Système λ^{τ}

Il s'agit d'une simplification du système C, où l'on interdit les variables de type, mais où on se limite à une seule constante de type ι :

C = {type, ι}
S = {type}
A = {ι : type}
R = {(type,type)}

Les 3 systèmes qui suivent sont issus de la "famille" Automath, avec trois sortes : **S** = {type, arité, sorte}, et un axiome : **A** = {type : arité}.

Système PAL

R = {(type,type,sorte), (type,arité,sorte), (arité,type,sorte),
(arité,arité,sorte), (type,sorte,sorte), (arité,sorte,sorte)}

Il s'agit d'un système très restreint, car l'abstraction n'est possible qu'au niveau le plus externe.

Système AUT-68

R = {(type,type,type), (type,arité,sorte), (arité,type,sorte),
(arité,arité,sorte), (type,sorte,sorte), (arité,sorte,sorte)}

Dans ce système, on peut former par exemple :
[A : type][x : A][P : A→type] ⊢ (P x) : type.

Système AUT-QE

R = {(type,type,type), (type,arité,arité), (arité,type,sorte),
(arité,arité,sorte), (type,sorte,sorte), (arité,sorte,sorte)}

Ce système est assez proche du système P présenté plus haut. On peut maintenant abstraire sur un prédicat :
[A : type][x : A] ⊢ [P : A→type](P x) : (A→type)→type.

Aucun des systèmes PAL, AUT-68 et AUT-QE n'est circulaire en aucune sorte, car ils n'autorisent pas la règle (arité,type,type), mais seulement (arité,type,sorte).

Le systèmes Automath sont diverses extensions de AUT-68 ou AUT-QE avec un système de constantes permettant d'exprimer des définitions et des lemmes. Des extensions plus récentes autorisent les sommes, appelées ici "télescopes".

On donne finalement la présentation sous forme de GTS de la théorie des types de Church :

Système HOL

S = {type, arité, sorte}
A = {type : arité , arité : sorte}.
R = {(type,type), (arité,type), (arité,arité)}

Autrement dit, HOL est Fω enrichi avec l'axiome (arité : sorte).

Remarque. Si l'on désire utiliser ces systèmes de type comme des logiques, et non simplement comme des cadres fonctionnels, on les présentera suivant l'isomorphisme de Curry-Howard, en renommant les sortes (type, arité, sorte) en respectivement (Prop, Type, sorte). On peut alors reconnaître les règles au niveau Prop comme les règles d'inférence d'un calcul des prédicats intuitionniste, et prouver des résultats de conservativité par rapport à des présentations logiques plus traditionnelles. Par exemple, nous reviendrons plus loin sur HOL, présenté sous forme Curry-Howard. Le résultat de normalisation du système fonctionnel est l'analogue d'un résultat d'élimination des coupures dans le système logique, disant qu'il existe une forme canonique, complètement explicitée, de toute preuve. Nous reviendrons sur ces problèmes dans la section 6.6 ci-dessous.

6.5.2. Systèmes mal fondés.

Les systèmes suivants permettent de construire des termes non normalisables. Ils ne peuvent donc servir de support à un système logique.

Système λ*.

S = {type}
A = {type : type}.
R = {(type,type)}

Ce système est fortement circulaire en type, qui est son propre type. Dans ce système, tous les types sont habités ("paradoxe de Girard"). Il n'existe pourtant pas de preuve très simple de ce paradoxe, qui semble intrinsèquement plus compliqué que le paradoxe de Russell. La preuve la plus courte connue à ce jour, due à Coquand, formalise dans ce système le paradoxe de Burali-Forti.

Même si l'on s'interdit d'utiliser l'axiome type:type, il est très facile d'obtenir des systèmes incohérents comme extension des systèmes précédents. Par exemple :

Système U-.

S = {type, arité, sorte}
A = {type : arité , arité : sorte}.
R = {(type,type), (arité,type), (arité,arité), (sorte,arité)}

Remarquez que ce système est juste HOL avec la règle supplémentaire (sorte,arité). Coquand a montré l'existence d'un terme non normalisable dans ce système, inspiré du paradoxe de Reynolds. Ce système est une simplification du système U considéré par Girard, qui admet en supplément la règle (sorte,type). Ce système montre qu'on ne peut pas être "non-prédicatif à deux niveaux" (c'est-à-dire circulaire à la fois relativement à type et à arité) sans perdre la normalisation.

6.6 Logique, paradoxes, contextes cohérents, codages fidèles.

Si l'on essaye d'utiliser un système de typage tels que ceux considérés jusqu'à présent en tant que système logique, il convient de fixer les sortes correspondant aux propositions et aux preuves de la logique considérée. Une théorie, présentée par la signature des opérations axiomatisées, et un ensemble d'axiomes ou de schémas d'axiomes, correspondra à un contexte; on pourra alors se poser le problème de l'adéquation du codage par rapport aux présentations plus traditionnelles

de cette logique.

Il y a en gros un choix entre deux possibilités pour coder les preuves. La première alternative consiste à considérer le système de typage comme un simple "cadre logique" donnant la possibilité d'exprimer la notion de substitution, et la généricité des règles d'inférence et des schémas d'axiomes par l'abstraction du λ-calcul sous-jacent. C'est cette voie qui est suivie plus haut pour les systèmes P ou LF.

La deuxième alternative consiste à utiliser l'isomorphisme de Curry-Howard pour coder directement les preuves comme des λ-termes, dans la tradition de la déduction naturelle. L'abstraction fournit un codage direct des règles d'introduction des connectives $\Rightarrow$ et $\forall$, alors que l'application correspond aux règles d'élimination. C'est cette solution qui a été retenue pour l'axiomatisation des mathématiques dans les systèmes Automath. Dans cette deuxième voie, le problème de l'adéquation du codage est particulièrement aigu, parce que la puissance du système de typage peut éventuellement fournir des preuves ne correspondant pas au codage d'une preuve traditionnelle.

Par exemple, considérons le système HOL, présenté à la Curry-Howard :

Système HOL (version Curry-Howard)

S = {Prop, Type, sorte}
A = {Prop : Type , Type : sorte}.
R = {(Prop,Prop), (Type,Prop), (Type,Type)}

Ce système est assez proche en esprit de la théorie des types de Church ; les différences essentielles sont que les preuves y sont explicites, le système d'inférence correspond à la déduction naturelle et non à un système de Hilbert, et enfin la logique est intuitionniste. Pour obtenir la théorie classique, il faut se placer dans un contexte donnant le tiers-exclu comme axiome.

Considérons l'axiome:

$$EQ = (p,p':Prop).(p\to p')\to(p'\to p)\to(P:Prop\to Prop).P(p)\to P(p')$$

et définissons l'égalité, la disjonction et l'absurdité par:

$$eq = [p:Prop][x,y:p](P:p\to Prop).P(x)\to P(y)$$
$$or = [p,q:Prop](r:Prop)(p\to r)\to(q\to r)\to r$$
$$\bot = (P:Prop)P.$$

Alors on peut montrer que le contexte suivant :
Γ = [e:EQ][bool:Prop][true:bool][false:bool]
[two:(b:bool)(b=true)∨(b=false)][only_two:(true≠false)],
où (b=b') abrège (eq bool b b'), (b≠b') abrège (b=b')→⊥,
et p∨q abrège (or p q),
est incohérent, dans le sens qu'il permet d'habiter ⊥.

Ce paradoxe, dû à Berardi et Geuvers, semble dire que l'axiome EQ a des conséquences inhabituelles. En fait, il s'agit plutôt d'une erreur d'axiomatisation : il est déraisonnable d'axiomatiser un type de données tel que bool par un élément de la sorte Prop, car les "booléens" ainsi axiomatisés héritent des propriétés des structures de preuves, ce qui risque d'amener des conséquences non voulues. Ici, l'axiome EQ n'est pas vraiment une formalisation correcte d'une propriété de l'égalité, mais c'est plutôt un principe de non-pertinence des preuves : les preuves de deux propositions équivalentes sont identifiées. Par l'équivalence des propositions P et P→P, lorsque P est prouvable, on en déduit par EQ un isomorphisme de P et P→P en tant que domaines de leurs preuves. Autrement dit, les preuves d'une proposition habitée telle que bool ont la structure d'une λ-algèbre. Comme la seule λ-algèbre finie est triviale, le paradoxe s'ensuit.

La bonne manière d'axiomatiser un type de données tel que bool dans HOL est de déclarer (remarquez qu'on utilise ici le fait que HOL a l'axiome Type:sorte, permettant des contextes plus généraux que ceux de Fω) :
[bool:Type][égal:bool→bool→Prop]...,
de définir eq_bool = [x,y:bool](P:bool→Prop).P(x)→P(y),
et on peut alors admettre l'axiome (x,y:bool)(égal x y)→(eq_bool x y)
sans paradoxe.

La source de l'axiomatisation erronée était le désir de définir une égalité de Leibniz eq générique. Dans le nouveau codage, nous perdons cette facilité, car le système HOL n'a pas suffisamment de polymorphisme pour permettre de définir :
eq = [T:Type][x,y:T](P:T→Prop).P(x)→P(y).

Si l'on désire axiomatiser des mathématiques génériques, sans pour cela utiliser dangereusement le polymorphisme disponible au niveau des preuves, il faut utiliser un système tel que CCE. On dispose alors d'un

système circulaire relativement à Prop (à cause de la quantification d'ordre supérieur), mais prédicatif au niveau des sortes $Type_i$ qui peuvent être interprétées comme des univers ensemblistes. Les deux niveaux ne se perturbent pas, et on peut prouver que les codages sont fidèles, c'est-à-dire qu'on a une extension conservatrice de la logique d'ordre supérieur traditionnelle. Ce qui est perdu est le codage des types de données à l'ordre supérieur.

Un autre résultat intéressant, dû à Coquand, est qu'il n'est pas possible de "réfléchir" le type Prop dans une proposition, au sens que le contexte suivant est incohérent (on peut y interpréter le système U) :

$\Delta = [\pi : Prop][\rho : \pi \to Prop][\varepsilon : Prop \to \pi][h : (A:Prop).\rho(\varepsilon(A)) \leftrightarrow A]$.

Remarque générale sur la cohérence. (Coquand)

On peut définir généralement la notion de contexte cohérent dans un GTS. On dit que Γ est un contexte cohérent relativement à la sorte S s'il existe un terme Vide tel que $\Gamma \vdash$ Vide : S et il n'existe pas de terme Absurde tel que $\Gamma \vdash$ Absurde : Vide.

La cohérence d'un système logique plongé dans un GTS support dans lequel on a distingué une sorte Prop se ramène alors à montrer que le contexte initial codant les primitives de la logique est cohérent relativement à la sorte Prop. Le rôle de la proposition Vide est celui de l'absurde. Dans un système non prédicatif, c'est-à-dire circulaire relativement à Prop, on peut faire jouer le rôle de Vide à $\bot$ = (P:Prop)P, et ramener la cohérence du système à la cohérence du contexte vide; c'est ce que nous avons fait pour CoC. Dans un système prédicatif, on mettra [Vide:Prop] dans le contexte initial représentant la signature des connectives et des règles d'inférence. Remarquez que dans ce cas on n'a pas en général la règle "ex falso quodlibet", c'est à dire qu'un contexte Γ peut habiter Vide sans pour autant être incohérent au sens ci-dessus.

Pour CoC, on connaît un certain nombre de contextes cohérents. Par exemple, le contexte **Inf** ci dessous est cohérent (Coquand, Seldin). Ce contexte est suffisant pour interpréter l'arithmétique classique d'ordre supérieur. Le dernier axiome est équivalent au tiers-exclu. Les autres sont une forme logique de l'axiome de l'infini.

Inf = $[\infty : Prop][o : \infty][s : \infty \to \infty][> : \infty \to \infty \to Prop]$
$[irr : (x:\infty).x>x \to \bot][trans : (x,y,z:\infty).x>y \to y>z \to x>z][gr : (x:\infty).(s\ x)>x]$
$[classic : (P:Prop).((P \to \bot) \to \bot) \to P]$.

7. Le Calcul des Constructions Inductives.

7.1 Introduction.

Revenons un instant au Calcul des Constructions CoC. Dans ce calcul, qui contient le système F, on peut utiliser un codage imprédicatif pour définir les types de données tels que **B** et **N**. Comme la section précédente l'a montré, ceci présente un certain danger, car les éléments de type **N** possèdent, en tant que preuves de la proposition de second-ordre

$$\mathbf{N} = \forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha),$$

des propriétés peut-être indésirables. Mais en revanche il semble y avoir un intérêt à suivre cette voie : les entiers sont définis de fait, il n'y a pas besoin d'axiomatiser l'existence de 0 et de Succ, ces constructeurs étant définissables comme nous l'avons vu. De plus, cette définition est dans un sens correcte, car on peut montrer que les éléments en forme normale (η-longue) de type **N** sont isomorphes aux entiers. On a donc d'une certaine manière le modèle standard de l'arithmétique "pré-défini". Mais cette meta-constatation est illusoire. En effet, cette connaissance ne nous sert à rien pour faire des preuves sur les entiers, car par exemple nous n'avons aucun moyen de dériver 0≠Succ(0). On peut d'ailleurs se demander quel est précisément le statut de l'égalité, et si l'égalité de Leibniz eq = [p:Prop][x,y:p](P:p→Prop).P(x)→P(y) est vraiment appropriée. De même, l'axiome de récurrence

Peano = (P:**N**→Prop).P(0)→((x:**N**)P(x)→P(Succ(x)))→(x:**N**)P(x)

n'est pas non plus démontrable, et doit donc être postulé comme hypothèse.

Toutes ces raisons militent en faveur du retour à une axiomatisation plus traditionnelle de l'arithmétique, en postulant les axiomes de Peano par le contexte :

[nat:Type][=:nat→nat→Prop][0:nat][Succ:nat→nat]

[A1:(x:nat)(= 0 (Succ x))→⊥]

[A2:(x,y:nat)(= (Succ x)(Succ y))→(= x y)]

[Peano:(P:nat→Prop).P(0)→((x:nat)P(x)→P(Succ(x)))→(x:nat)P(x)]

plus les axiomes de l'égalité. Mais les preuves dans ce système sont très lourdes, car une preuve de n=m ne permet pas directement de prouver P(n) à partir de P(m), il faut faire explicitement appel à l'axiome de substitutivité de l'égalité.

Le Calcul des Constructions Inductives répond à ces critiques en donnant un moyen primitif de définir des types, des propositions et des prédicats de manière récursive. Un tel type définit des constructeurs et des destructeurs, correspondant à des principes de récurrence/récursion. Il étend strictement la puissance du calcul précédent, car de nouvelles relations de réduction sont introduites pour chaque nouvelle paire destructeur-constructeur. Autrement dit, le calcul CCI n'est plus un GTS pur, car d'autres égalités définitionnelles sont admises que la β-conversion.

Ce calcul est bien adapté à la formalisation de concepts récursifs, et à l'extraction de programmes à partir de preuves de leurs spécifications.

7.2 Définition de CCI (suivant Coquand & Paulin).

Le système est une extension du système CCE, dont il partage les sortes, les axiomes et les règles. En oubliant les indices d'univers, on peut présenter le système comme une sorte d'union de CoC et de λ* :

S = {Prop, Type}
A = {Prop : Type , Type : Type}
R = {(Prop, Prop), (Prop, Type), (Type, Prop), (Type, Type)}

On se donne trois schémas de constructions de termes supplémentaires: les minimisateurs, les constructeurs, et les éliminateurs. Intuitivement, un minimisateur sera une définition de type ou prédicat inductif, par cas suivant une union disjointe. Les constructeurs discriminent les différents cas, ce sont donc toutes les manières de construire un élément du type. Les éliminateurs correspondent au niveau Prop, à un schéma de récurrence, permettant le raisonnement par cas, et au niveau Type à un schéma de récursion, permettant la définition de valeurs par récurrence sur le type. Pour chaque paire (éliminateur,constructeur), on a une règle de calcul supplémentaire. Donnons tout d'abord quelques définitions auxiliaires.

On dit qu'un terme T est un <u>type</u> dans le contexte Γ si $\Gamma \vdash T : S$, avec $S \in \mathbf{S}$. Dans la suite, on emploiera les meta-variables S et T respectivement pour sorte et type, le contexte pertinent étant sous-entendu.

Une arité est un terme qui est une sorte S, ou de la forme (x:T).A, avec T un type, et A une arité :

$$A := S \mid (x:T).A$$

C'est-à-dire, une arité est de la forme $A = (x:T_1).(x:T_2) \ldots (x:T_k).S$. On dit que A est de longueur k, et on écrit $|A| = k$. L'entier k désigne le nombre de paramètres du type inductif.

On se place dans un contexte contenant la déclaration [X:A], avec A une arité. Dans la suite, on soulignera un terme tel que $\underline{M}$ pour indiquer qu'il ne contient pas d'occurrence de la variable X.

On dit que le terme P contient X positivement s'il est de la forme :

$$P := X \mid (P\ \underline{M}) \mid (x : \underline{T}).P$$

On dit que le terme C a une forme de constructeur s'il est de la forme :

$$C := X \mid (C\ \underline{M}) \mid P \to C \mid (x : \underline{T}).C$$

On remarque que les cas d'abstraction dépendante et non-dépendante sont distingués (mais disjoints).

On dit finalement qu'un type de constructeur est un type qui a une forme de constructeur.

Exemple. Dans la suite, on considérera l'exemple A=Type, et les deux formes de constructeur $C_1 = X$ et $C_2 = X \to X$, afin de définir le type **nat**.

Minimisateur.

Soit Γ un contexte, A une arité construite sur la sorte S, $[C_1 \mid \ldots \mid C_n]$ une liste de n types de constructeurs, avec $\Gamma\,[X:A] \vdash C_i : S$. On donne le procédé de construction de minimisation $\mu X:A.[C_1 \mid \ldots \mid C_n]$, avec X lié dans les C_i, et la règle de jugement de typage :

$$\Gamma\,[X:A] \vdash C_i : S \ (i \leq n) \ \Rightarrow \ \Gamma \vdash \mu X:A.[C_1 \mid \ldots \mid C_n] : A$$

Dans notre exemple, on définit **nat** = μX:Type.[X | X → X].

Constructeurs. On se place sous les hypothèses $\Gamma\,[X:A] \vdash C_i : S \ (i \leq n)$, permettant de former $T = \mu X:A.[C_1 \mid \ldots \mid C_n]$. Pour $i \leq n$, on autorise de former le

i-ème constructeur de ce type, noté **C**(i,T). On se donne les n règles de jugement de typage :

$$\Gamma\,[X{:}A] \vdash C_i : S \;\; (i \leq n) \;\Rightarrow\; \Gamma \vdash \mathbf{C}(i,T) : C_i\{X \leftarrow T\} \quad \text{avec } T = \mu X{:}A.[C_1 | \ldots | C_n]$$

Dans notre exemple, on trouve les deux constructeurs :
$\underline{\mathbf{0}}$ = **C**(1,**nat**) : **nat** et $\underline{\mathbf{S}}$ = **C**(2,**nat**) : **nat**→**nat**.

Remarque. Le type concret CAML :

type $\alpha_1 \ldots \alpha_k$ t = c_1 of T_1 | ... | c_n of T_n

dans le cas particulier où toutes les occurrences de t dans les T_i sont de la forme $\alpha_1 \ldots \alpha_k$ t, correspond (approximativement) à définir :

$[\alpha_1, \ldots, \alpha_k : \text{Type}]\ \mu X{:}\text{Type}.[C_1 | \ldots | C_n]$, avec $C_i = T_i\{(\alpha_1 \ldots \alpha_k\ t) \leftarrow X\} \rightarrow X$,

à nommer ce type t et ses constructeurs $c_1, \ldots, c_n$.

Avant de poursuivre avec la définition des éliminateurs, un certain nombre de définitions auxiliaires additionnelles sont nécessaires.

Soit A une arité. Si X:A, C est un type de constructeur de X, et N un terme quelconque, on définit le terme SubX(C,N) par :

```
let rec SubX(C,N) = SubX_rec C
        where rec SubX_rec = function
            X          →  N
          | (C M)      →  SubX(C,(N M))
          | P → C      →  P → P{X←N} → SubX_rec(C)
          | (x : T).C  →  (x : T).SubX_rec(C);;
```

Soit M un terme de type (X N_1 ... N_k), avec [X:A] dans le contexte courant, avec |A| = k. Soit CL une liste de type de constructeurs de X. On définit le type Elim([N_1 ... N_k],CL,A) comme :

```
let Elim([N1 ... Nk],CL,A) = (P:A). Elim_rec CL
        where rec Elim_rec = function
              []     →  (P N1 ... Nk)
            | C::L   →  SubX(C,P) → Elim_rec(L);;
```

Par exemple, avec CL = [X | X → X], on obtient :
Elim([],CL,Prop) = (P:Prop). P → (X → P → P) → P.

Eliminateurs.

Les termes d'élimination sont de la forme **E**(M), ou $\mathbf{R}_{S'}$(M), avec S' une sorte.

a. S = Prop.

On se place sous les hypothèses $\Gamma\,[X:A]\vdash C_i : S$ $(i\leq n)$, permettant de former T= μX:A.CL, avec CL = $[C_1 \mid ... \mid C_n]$. Le terme **E**(M) est un principe de minimisation. La règle de typage en est :

$$\Gamma \vdash M : (T\ N_1\ ...\ N_k) \Rightarrow \Gamma \vdash \mathbf{E}(M) : Elim([N_1\ ...\ N_k],CL,A)\{X\leftarrow T\}$$

avec T = μX:A.CL, k = |A|.

Par exemple, avec :
conj = [A,B:Prop] μX:Prop.[A→B→X],
et en notant P∧Q pour (**conj** P Q), on obtient un opérateur :
Γ [A,B:Prop] ⊢ c : A∧B
⇒ Γ [A,B:Prop] ⊢ **E**(c) : (P : Prop).(A→B→P)→P,
et en particulier :
elim_∧_left = [A,B:Prop][c : A∧B](**E**(c) A [x:A][y:B]x) : (A,B:Prop)A∧B→A.

b. S = Type.

Sous les mêmes hypothèses, on peut autoriser S' = Prop, et alors $\mathbf{R}_{S'}$(M) est un principe de récurrence, ou S' = Type, et alors $\mathbf{R}_{S'}$(M) est un principe de récursion.

Le principe d'élimination peut ici dépendre des constructeurs du type, et nous devons donner des définitions plus complexes des notions précédentes. Ainsi, avec le type **nat** ci-dessus, on veut pouvoir obtenir le principe de récurrence de Peano.

Soit A une arité, S' une sorte, et R un terme de type A. On définit récursivement l'arité Ar(A,R,S') par :

```
let Ar(A,R,S') = Ar_rec (A,R)
                 where rec Ar_rec = function
```

```
                (S,R)        → (R → S')
            |   ((x:T).A,R)→  (x:T).Ar_rec(A,(R x));;
```

Avec A=Type et R=**nat**, on a Ar(A,R,S') = **nat** → S'. Remarquez qu'on a toujours |Ar(A,R,S')| = |A|+1.

Dans un contexte Γ[X:A], avec C un type de constructeur de X, c:C, et N:Ar(A,X,S'), on définit le terme SubXd(C,c,N) par :

```
let rec SubXd(C,c,N) = match (C,c) with
                         (X,N')          → (N N')
                       | ((C M),N')      → SubXd(C,N',(N M))
                       | ((x : I).C,N')  → (x : I).SubXd(C,(N' x),N)
                       | (P → C,N')      → (p:P).φ(P,N,p) → SubXd(C,(N' p),N)
                         where rec φ(P,N,Q) = match P with
                                                X          → (N Q)
                                              | (P M)      → φ(P,(N M),Q)
                                              | (x : I).P  → (x : I). φ(P,N,(Q x));;
```

Exercice. Vérifier que, sous les hypothèses données, SubXd(C,c,N) est bien typé.

Par exemple, on calcule :
SubXd(X,0,P) = (P 0)
et SubXd(X→X,S,P) = (i:X)(P i) → (P (S i)).

Soit M un terme de type (X N_1 ... N_k), avec [X:A] dans le contexte courant, |A| = k. Soit CL = [C_1 | ... | C_n] une liste de type de constructeurs de X, A'=Ar(A,X,S'), et N'_1, ... , N'_n une suite de termes tels que N'_i:C_i. On définit le type Elimd([N_1 ... N_k],CL,A',[N'_1 ... N'_n],M) comme :

```
let Elimd([N1 ... Nk],CL,A',[N'1 ... N'n],M) = (P:A'). Elim_rec(CL,[N'1 ... N'n])
        where rec Elim_rec = function
                               ([],[])       → (P N1 ... Nk M)
                             | (C::L,N'::L') → (SubXd(C,N',P) → Elim_rec(L,L'));;
```

Par exemple, avec CL = [X | X → X], A' = X → Prop, 0 : X, et S : X → X, on a :

```
Elimd([],CL,A',[0,S],n) = (P:X→Prop).SubXd(X,0,P)→SubXd(X→X,S,P)→(P n)
                        = (P:X→Prop).(P 0) → ((i:X)(P i)→(P (S i))) → (P n).
```

Finalement, dans les conditions de définition d'un type inductif $T=\mu X{:}A.CL$, avec $|A| = k$ et $|CL| = n$, on définit :

$\varepsilon(T,[N_1 \ldots N_k],S',M) = \text{Elimd}([N_1 \ldots N_k],CL,A',[X_1 \ldots X_n],M)\{X \leftarrow T,\ X_i \leftarrow \mathbf{C}(i,T),\ i \leq n\}$,

où $A'=Ar(A,X,S')$.

Nous laissons au lecteur le soin de vérifier les conditions de type.

La règle de typage devient ici :

$$\Gamma \vdash M : (T\ N_1 \ldots N_k) \Rightarrow \Gamma \vdash \mathbf{R}_{S'}(M) : \varepsilon(T,[N_1 \ldots N_k],S',M)$$

(S' = Prop ou S' = Type).

Par exemple, avec T = **nat**, S' = Prop, on obtient :

$\Gamma \vdash n :$ **nat** $\Rightarrow$

$\Gamma \vdash \mathbf{R}_{Prop}(n) : (P{:}\ \mathbf{nat} \rightarrow Prop).\ (P\ \underline{\mathbf{0}}) \rightarrow ((i{:}\mathbf{nat}).(P\ i) \rightarrow (P\ (\underline{\mathbf{S}}\ i))) \rightarrow (P\ n).$

On peut donc définir :

Peano = $[n{:}\mathbf{nat}].\ \mathbf{R}_{Prop}(n)$

$: (n{:}\mathbf{nat}).(P{:}\ \mathbf{nat} \rightarrow Prop).\ (P\ \underline{\mathbf{0}}) \rightarrow ((i{:}\mathbf{nat}).(P\ i) \rightarrow (P\ (\underline{\mathbf{S}}\ i))) \rightarrow (P\ n)$,

qui correspond au principe de récurrence usuel, à permutation d'arguments près. De même, si S'=Type, on obtient un principe de récursion.

Règles de calcul.

Si X:A, C est un type de constructeur de X, N, Q des termes quelconques, et L une suite de termes, on définit le terme Transform(C,Q,N,L) par :

```
let Transform (C,Q,N,L) = Trans_rec Q (C,L)
    where rec Trans_rec Q = function
        (X,[])              → Q
      | ((C M),[])          → Q
      | ((x : T).C,N'::L')  → Trans_rec (Q N') (C,L')
      | (P → C,N'::L')      → Trans_rec (Q N' (ψ(P) N')) (C,L')
                              where rec ψ = function
                                    X       → N
                                  | (P M)   → (ψ(P) M)
                                  | (x : T).P → [h:(x : T)P][x : T].(ψ(P) (h x));;
```

On se place sous les hypothèses $\Gamma\ [X:A] \vdash C_i : S$ $(i \leq n)$, permettant de

former T= μX:A.CL, avec CL = $[C_1 \mid ... \mid C_n]$. Soit $N_0, ..., N_n$ une liste de n+1 termes. On définit $\xi(T,[N_0, ..., N_n])$ par :

let $\xi(T,[N_0, ..., N_n])$ = xi T A
where rec xi U = function
S → [h:U](**E**(h) N_0 ... N_n)
| (x:U').A' → [x:U']xi (U x) A';;

La règle de calcul peut maintenant se décrire comme le remplacement du radical (E N N_1 ... N_n), avec E= **E**((**C**(i,T) N'_1 ... N'_k)) par le terme :
Transform(C_i, N_i, $\xi(T, [N, N_1, ..., N_n])$, $[N'_1, ..., N'_k]$) (i≤n).

Idem en remplaçant **E** par $\mathbf{R}_S$.

Exercice. Donner les conditions pour que les termes construits dans les règles ci-dessus soient bien typés.

Remarque. On a maintenant une égalité définitionnelle qui varie avec le contexte, puisque la règle ci-dessus n'est active que dans la portée de la définition de T. Les règles de réduction créent en général des radicaux supplémentaires, au sens de la β-réduction.

Par exemple, avec T = **nat**, P : **nat** → Type, et **R** = $\mathbf{R}_{Type}$, on obtient les deux règles :

(**R**(**0**) P z s) ⇨ z

et (**R**((**S** n)) P z s) ⇨ Transform(X→X, s, [m:**nat**](**R**(m) P z s), [n])
= (s n ([m:**nat**](**R**(m) P z s) n))
⇨ (s n (**R**(n) P z s)) par β-réduction.

On peut par exemple définir l'addition par :
add = [n,m:**nat**](**R**(n) ([i:**nat**]**nat**) m [i:**nat**]**S**),
et avoir (**add** **0** m) ≅ m,
et (**add** (**S** n) m) ≅ (**S** (**add** n m)).

De même, le prédécesseur se définit par :
pred = [n:**nat**](**R**(n) ([i:**nat**]**nat**) **0** [p,m:**nat**]p),
et (**pred** (**S** n)) se réduit en n en temps constant.

7.2 Problèmes ouverts.

La meta-théorie du système CCI reste à faire. On conjecture que la relation de calcul est confluente, et qu'elle admet la normalisation forte.

La notion de positivité choisie ici est très forte. Elle interdit par exemple de former $\mu X{:}Type.[(X \to Prop) \to Prop]$. En fait, Coquand a montré que l'ajout de ce type inductif mène à un paradoxe. Ceci est similaire à la preuve de Reynolds de non-existence d'une sémantique ensembliste pour F. Par contre, il serait possible d'autoriser la formation de types tels que : $[P : Prop] \vdash \mu X{:}Prop.[(X \to P) \to P] : Prop$.

De même la restriction à S=Type dans le cas **b** dépendant est nécessaire, car si on l'autorisait pour S=Prop on pourrait définir une somme forte incohérente avec la non-prédicativité.

La règle η est utile dans la pratique, et devra peut-être être ajoutée à une version ultérieure du système. De même, il sera sans doute utile de généraliser la construction de types inductifs, pour autoriser des récursions croisées.

Les types inductifs tels qu'ils ont été définis ci-dessus ne sont pas munis des noms de leurs constructeurs et destructeurs. Dans l'exemple **nat** ci-dessus, les noms **0**, **S** et **add** sont des abréviations, ou des constantes gérées par un mécanisme de constantes nommées. Mais il n'y a pas de moyen direct de reconnaître que (**add** (**S** n) m) ≅ (**S** (**add** n m)).

7.3 Autres exemples.

Nous avons vu plus haut que la connective de conjonction était définissable par le type :

conj = $[A,B{:}Prop]\ \mu X{:}Prop.[A \to B \to X]$.

De la même manière on peut définir le produit de types :

prod = $[A,B{:}Type]\ \mu X{:}Type.[A \to B \to X]$.

La généralisation dépendante correspond respectivement à la quantification existentielle et à la somme dépendante :

exists = $[A{:}Type][P{:}A \to Prop]\ \mu X{:}Prop.[(x{:}A).(P\ x) \to X]$,

sigma = [A:Type][P:A→Type] μX:Type.[(x:A).(P x)→X].

La disjonction (resp. somme) se définit de manière analogue :
disj = [A,B:Prop] μX:Prop.[A→X | B→X],
sum = [A,B:Type] μX:Type.[A→X | B→X].

On distingue les propositions **False** = μX:Prop.[] et **True** = μX:Prop.[X].

Finalement, voici la définition inductive de l'égalité, dûe à C. Paulin :
equal = [A:Type][x:A] μP: A→Prop.[(P x)].

Notons $M=_A N$ pour (**equal** A M N). Plaçons nous dans un contexte [A:Type][x:A]. Le prédicat inductif (**equal** A x) a un constructeur **refl** : $x=_A x$. Et si M : $x=_A y$, on a **E**(M) : (P : A→ Prop).(P x) → (P y). Il est remarquable qu'on retrouve l'égalité de Leibniz.

Les types de données algébriques se définissent très naturellement. Nous avons vu l'exemple :
nat = μX:Type.[X | X → X].
De la même façon :
void = μX:Type.[], sans constructeurs,
unit = μX:Type.[X], avec un constructeur **one**,
bool = μX:Type.[X | X], avec deux constructeurs **false** et **true**,
list = [A:Type] μX:Type.[X | A → X → X], de constructeurs :
nil : [A:Type](**list** A) et **cons** : [A:Type]A → (**list** A) → (**list** A).

Si Φ est un opérateur positif en X, on peut définir généralement :
μ(Φ) = μX:Type.[Φ(X) → X].

Donnons pour finir le type des listes de longueur donnée :
list_n = [A:Type] μX: **nat**→Type.[(X **0**) | (n: **nat**).(X n) → (X (**S** n))].

Remarque. On peut définir une coercion de **bool** dans Prop par :
choice = [b:bool]($\mathbf{R}_{Type}$(b) (**bool**→Prop) **False True**). La proposition **choice**(**true**) est prouvable (par **C**(1,**True**)), alors que **choice**(**false**) est incohérente, car contr : **choice**(**false**) ≅ **False** ⊢ **E**(contr) : (P:Prop).P = ⊥.
Ceci montre que $\mathbf{true}=_{\mathbf{bool}}\mathbf{false}$ est contradictoire, prouvant que CCI est strictement plus puissant que CoC.

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", Masson, 1990. Un ouvrage plus avançé 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 intuitionniste des types de P. Martin-Löf est décrite dans sa monographie "Intuitionistic Type Theory", Bibliopolis, 1984. Une implémentation en est décrite dans "Implementing Mathematics with the Nuprl Development System", de R. Constable et al., Prentice-Hall, 1986. Voir également "Programming in Martin-Löf's Type Theory", par B. Nordström, K. Petersson & J. Smith, Oxford University Press, 1990.

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.