Axiomatisation d'un Lambda-Calcul avec Substitutions Explicites dans Coq

Amokrane Saibi

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Axiomatisation d'un λ-Calcul avec Substitutions Explicites dans Coq

Amokrane SAÏBI

N° 2345

Juillet 1994

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Rapport de recherche

1994

ISSN 0249-6399

INRIA
ROCQUENCOURT

Axiomatisation d'un λ-Calcul avec Substitutions Explicites dans Coq

Amokrane SAÏBI

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Coq

Rapport de recherche n ° 2345 — Juillet 1994 — 40 pages

Résumé : Nous présentons l'axiomatisation de la théorie du $\lambda\sigma_{\Uparrow}$-calcul[6] dans le système Coq[7]. Le principal résultat axiomatisé est la confluence de ce calcul. Par ailleurs, nous proposons un codage uniforme des systèmes de réécriture à une ou plusieurs sortes, sur lequel nous étudions la confluence locale.

(Abstract: pto)

This research was partially supported by ESPRIT Basic Research Action "TYPES."

Unité de recherche INRIA Rocquencourt

Axiomatization of a λ-Calculus with Explicit Substitutions in Coq

Abstract: We present the axiomatization of the $\lambda\sigma_{\Uparrow}$-calculus in the system Coq[7]. The principal result axiomatized is the confluence of this calculus. Furthermore we propose a uniform encoding for many-sorted rewriting systems, on which we study the local confluence.

1 Introduction

Le λ-calcul est la base théorique des langages de programmation fonctionnelle, sa principale règle est la règle β :

$$(\lambda x.a)b \longrightarrow a\{b/x\}$$

où a et b sont des termes et $a\{b/x\}$ représente a où toutes les occurrences libres de la variable x sont remplacées par b. L'opération de substitution, qui décrit le passage des paramètres des fonctions, n'apparaît dans le λ-calcul qu'au niveau du méta-langage. En pratique, cette opération n'est pas simple à effectuer car on doit éviter les conflits de noms et faire attention à la portée des arguments des fonctions. De plus pour des raisons d'efficacité, les substitutions sont souvent retardées et explicitement enregistrées. Pour combler la distance entre le λ-calcul et ses implémentations, il devient nécessaire de trouver des variantes fines du λ-calcul, qui puissent manipuler explicitement des substitutions.

Les λσ-calculs [1, 6, 10] (appelés aussi λ-calculs avec substitutions explicites) sont des formalismes permettant de rendre compte de manière explicite du calcul de la substitution. Ils servent de cadre théorique à la conception, à la compréhension et à la vérification des machines abstraites.

Dans les λσ-calculs, les substitutions ont une représentation syntaxique. Ainsi, si a est un terme et s une substitution alors, $a[s]$ représente le terme a sur lequel s doit être effectuée. La règle de base des λσ-calculs est la règle $Beta$, elle se contente de déclencher la substitution :

$$(\lambda x.a)b \longrightarrow a[(b/x) \cdot id]$$

$[(b/x) \cdot id]$ est une construction syntaxique dénotant la substitution qui remplace x par b sans affecter les autres variables ($\cdot$ est appelé *cons* et id est la substitution identité). En simplifiant, on peut voir une substitution comme une liste de couples "terme substitué/variable".

Les difficultés liées à la capture des variables ont conduit les auteurs des λσ-calculs à opter pour une notation sans variables, c'est la notation de de Bruijn pour le λ-calcul :

$$a ::= n \mid aa \mid \lambda a$$

où n est un entier naturel. L'idée est que toute variable liée peut-être codée par sa hauteur de liaison (le nombre de λ que l'on rencontre avant d'atteindre le λ lieur), par exemple :

$$\lambda x.\lambda y.xy \;\; \mathrm{d}evient \;\; \lambda\lambda 10$$

Dans les λσ-calculs, ces indices correspondent aussi à une position dans un environnement, la règle $Beta$ devenant alors :

$$(\lambda a)b \longrightarrow a[b \cdot id]$$

Bien entendu d'autres règles sont nécessaires pour gérer explicitement les substitutions. Ces règles sont appelées *règles de substitution*. Elles devront en particulier être capables de "pousser" la substitution à l'intérieur de a. Ainsi, dans le cas de $(\lambda c)[b \cdot id]$, on doit éviter de remplacer les occurrences de 0 dans c puisque celles-ci correspondent à des variables liées ; la solution est de décaler (*shift*) la substitution. De plus, nous devons incrémenter (*lift*) tous les indices dans b afin d'éviter les captures.

Le $\lambda\sigma_{\Uparrow}$-calcul (initialement appelé λEnv-calcul) défini par T.Hardin et J.-J.Lévy est un λσ-calcul particulier possédant la propriété de confluence. Dans cet article, nous décrivons le codage de cette preuve de confluence dans le système d'aide à la démonstration Coq. L'avantage d'un codage dans un tel système est la sûreté : c'est la machine qui contrôle l'exactitude des preuves.

Seules les principales étapes de la preuve sont décrites dans ce papier. Pour plus de détails, le lecteur pourra se référer au script complet disponible en ftp sur **ftp.inria.fr**.

À travers ce développement, nous proposons une démarche générale de codage dans Coq des systèmes de réécriture à une ou plusieurs sortes. Nous insistons par ailleurs sur la démonstration de la confluence locale de $\sigma_{\Uparrow}$ et montrons comment adapter cette preuve à un système de réécriture quelconque.

La première partie de cet article est consacrée à la description du système Coq. Par contre nous supposons une certaine familiarité avec la réécriture (voir [11]) et les $\lambda\sigma$-calculs (voir [1, 6, 8, 9, 10]).

2 Utilisation du système Coq

Coq est une implémentation du Calcul des Constructions Inductives (CCI). C'est à la fois un langage de programmation et un système d'aide à la démonstration. Il permet d'une part la spécification et la synthèse de programmes, et d'autre part, la représentation et la vérification de preuves mathématiques. CCI est le résultat de l'extension du calcul des constructions par un mécanisme primitif de définitions inductives [17].

Le calcul des constructions est un λ-calcul typé d'ordre supérieur conçu par T.Coquand et G.Huet en 1985. Formellement, un λ-calcul typé [2, 13] définit un langage de termes et de types, ainsi qu'un ensemble de règles décrivant comment dériver des jugements de typage de la forme $\Gamma \vdash M : \alpha$ (qu'on lit "M est un terme de type α dans le contexte Γ"). On pourra se référer à [4] pour avoir une présentation détaillée des règles d'inférence du calcul des constructions.

Dans ce calcul, les types sont aussi des termes ; leur grande puissance d'expression nous permet de définir aussi bien des structures de données que des propositions logiques (via l'isomorphisme de Curry-Howard).

Le but de cette section n'est pas de présenter Coq dans tous ses détails. On se contentera d'en donner une description suffisante pour la compréhension des fragments de code présents dans la suite. Une présentation complète du système Coq se trouve dans [7]. On pourra aussi, avec profit, consulter les articles décrivant des développements réalisés dans Coq [3, 14, 15, 18, 19].

2.1 Les termes

Coq possède deux sortes de types : *Prop* et *Set*. *Prop* est le type des propositions logiques alors que *Set* est le type des spécifications (ensembles). Du point de vue de l'extraction de programmes, *Set* est le type des propositions ayant un contenu constructif (qu'on retrouve dans le programme extrait). Les propositions sont habitées par leurs preuves, i.e. si M est une proposition (M : *Prop*) alors les termes de type M sont des preuves de M.

On distingue trois constructions principales dans la syntaxe des termes :

- `(M N)` est l'application du terme fonctionnel `M` à l'argument `N`. L'application est associative à gauche : `(((M N1) N2) ... Nn)` s'écrit aussi `(M N1 N2 ... Nn)`.
- `[x :N]M` est l'abstraction de la variable `x` (de type `N`) dans le terme `M`. Le terme construit est un terme fonctionnel ($\lambda x \in N.M$).
- `(x :N)M` a deux interprétations possibles.

  Dans le cas d'une spécification, il s'agit d'un type dépendant ($\Pi x \in N.M$), i.e. le type d'une fonction dont le type du résultat dépend de l'argument. La syntaxe `N -> M` pour le type des fonctions ($N \longrightarrow M$) est une abréviation de `(x :N)M` lorsque `x` n'apparaît pas dans `M`.

  Dans le cas d'une proposition, `(x :N)M` correspond à la quantification universelle $\forall x \in N.M$. L'implication logique $N \Rightarrow M$ n'en est qu'une abréviation lorsque x n'apparaît pas dans M. Sa syntaxe dans Coq

est encore `N -> M`. L'implication est associative à droite : `M1 -> M2 -> ... Mn -> N` est équivalent à `(M1 -> (M2 -> ... (Mn -> N)))`.

Une autre construction importante de Coq est le mécanisme des définitions inductives. La prochaine section lui est consacrée.

Les autres connecteurs et quantificateurs, ainsi que l'égalité sont définis dans le paquetage *Prelude*. Nous en donnons la syntaxe :

- `M /\ N` est la conjonction des propositions `M` et `N`
- `M \/ N` est la disjonction des propositions `M` et `N`
- `M <-> N` : `M` si et seulement si `N`
- `~M` est la négation de la proposition `M`
- `False` est la proposition absurde
- `True` est la proposition tautologie
- `<M>Ex([x:M]N)` : il existe `x` de type `M` tel que `N`
- `<M>t = u` est l'égalité de `t` et `u` de type `M`

Remarques.

1. Les abréviations sont utilisées en mathématiques pour augmenter la lisibilité de formules complexes. Dans Coq, ce rôle est tenu par le mot clé `Definition` qui associe un nom à un terme. Ainsi par exemple, `(not A)` n'est qu'une abréviation de `A -> False` :

   ```
   Definition not [A:Prop] A -> False.
   ```

   `~A` est une syntaxe équivalente à `(not A)` :

   ```
   Syntax not "~_".
   ```

2. Les commentaires sont encadrés par `(*` et `*)`.

2.2 Les définitions inductives

Comme la théorie sous-jacente aux définitions inductives [17] est quelque peu complexe, nous avons préféré n'en donner qu'une présentation informelle. Cette présentation s'appuie sur de nombreux exemples dont la majorité provient de [7].

2.2.1 Types de données

La syntaxe des définitions de types de données dans Coq est proche de celle des types concrets de CAML. On définit un type inductif en donnant ses constructeurs avec leur type. Ainsi, l'ensemble des entiers est inductivement défini par :

```
Inductive Set nat = O : nat | S : nat -> nat.
```

ce type inductif a pour nom `nat` et possède deux constructeurs `O` (la constante zéro) de type `nat` et `S` (la fonction successeur) de type `nat -> nat`.

Le principal avantage des types inductifs par rapport aux types concrets de CAML est que Coq associe automatiquement à chaque type inductif un *principe d'induction* pour le type des propositions et un *principe de récursion* pour le type des spécifications.

Les principes d'induction et de récursion pour `nat` sont :

```
(P:nat -> Prop)(P O) -> ((y:nat)(P y) -> (P (S y))) -> (n:nat)(P n).
```

```
(P:nat -> Set)(P O) -> ((y:nat)(P y) -> (P (S y))) -> (n:nat)(P n).
```

La différence entre les deux principes réside dans le type du prédicat `P` (`nat -> Prop` ou `nat -> Set`).

Le principe d'induction de `nat` signifie : pour prouver `(P n)` pour tout `n`, il suffit de prouver `(P O)` et prouver que `(P y)` implique `(P (S y))` (pour tout `y`). Il s'agit d'un raisonnement par récurrence sur la structure des entiers. Un raisonnement analogue est possible pour tout ensemble construit de manière inductive.

On peut aussi définir le type des booléens. Sa définition dans *Prelude* est :

```
Inductive Set bool = true : bool | false : bool.
```

Son principe d'induction :

```
(P:bool -> Prop)(P true) -> (P false) -> (b:bool)(P b).
```

dit que tout élément de type `bool` est soit `true` soit `false`.

La définition de types polymorphes est aussi possible, ainsi :

```
Inductive Set list [A:Set] = nil : (list A) | cons : A -> (list A) -> (list A).
```

est une définition qui associe à chaque ensemble `A` le type `(list A)` des listes d'éléments de type `A`.

Remarque. Ce mécanisme ne permet pas la définition d'ensembles mutuellement récursifs.

2.2.2 Prédicats inductifs

Toujours grâce aux définitions inductives, on peut définir des prédicats sur les objets qu'on manipule (dans un style proche de celui de Prolog). Ainsi la propriété "n est pair" se traduit dans Coq par `(even n)` où `even` est le prédicat sur les entiers défini comme suit :

```
Inductive Definition even : nat -> Prop
 =  even_O : (even O)
  | even_S : (n:nat)(even n) -> (even (S(S n))).
```

Cette définition donne le nom `even` au plus petit (condition de minimalité) prédicat sur les entiers vérifiant les deux clauses `even_O` et `even_S`. Ces deux clauses ne sont que la traduction dans le langage Coq des axiomes mathématiques :

$$\begin{cases} 0 \ est\ pair \\ \forall n.\ n\ pair \Rightarrow (S\ (S\ n))\ pair \end{cases}$$

La condition de minimalité est exprimée dans le système par un axiome appelé *principe d'induction*. Le principe fourni pour `even` est :

```
          (P:nat -> Prop)
           (P 0) ->
            ((n:nat)(even n) -> (P n) -> (P (S (S n)))) ->
             (n:nat)(even n) -> (P n)
```

Il nous assure que `even` est contenu dans tout prédicat `P` vérifiant les clauses `even_0` et `even_S`. Un principe analogue est associé à chaque définition inductive.

L'utilisation des définitions inductives ne se limite pas au codage de prédicats unaires. Elles peuvent aussi représenter des relations. Prenons l'exemple de la relation ternaire `Plus`, `(Plus n m p)` représente "$p = n + m$" :

```
Inductive Definition Plus : nat -> nat -> nat -> Prop
=  Plus_0 : (m:nat)(Plus 0 m m)
 | Plus_S : (n,m,p:nat)(Plus n m p) -> (Plus (S n) m (S p)).
```

Le dernier exemple de cette section est la définition de la relation $\leq$. Elle comporte deux axiomes :

$$\begin{cases} \forall n.\, n \leq n \\ \forall n, m.\, n \leq m \Rightarrow n \leq (S\ m) \end{cases}$$

Son codage dans Coq est le suivant :

```
Induction Definition le [n:nat] : nat -> Prop
=  le_0 : (le n n)
 | le_S : (m:nat)(le n m) -> (le n (S m)).
```

D'autres relations sur les entiers s'en déduisent. Ainsi `lt` représente $<$:

```
Definition lt [n,m:nat] (le (S n) m).
```

et `gt` est la traduction de $>$:

```
Definition gt [n,m:nat] (lt m n).
```

Remarques.

1. La syntaxe `Inductive Set` n'est qu'un cas particulier de `Inductive Definition`. La définition des entiers (`nat`) qu'on a précédemment donnée est équivalente à :

   ```
   Inductive Definition nat : Set = 0 : nat | S : nat -> nat.
   ```

2. L'utilisateur n'a pas à utiliser explicitement les principes d'induction des définitions inductives. Leur utilisation se fait à travers des tactiques dites d'élimination (qu'on verra plus loin).

3. Même les connecteurs logiques sont définis de manière inductive. Nous prenons l'exemple de la disjonction (`A\/B` n'est qu'une abréviation de `(or A B)`) :

   ```
   Inductive Definition or [A,B:Prop] : Prop
    =  or_introl : A -> (or A B)
     | or_intror : B -> (or A B).
   ```

4. Nous donnons quelques précisions sur l'égalité dans Coq. Nous avons déjà vu que sa syntaxe était `<A>t = u` où `A` est de type `Set`. Elle est en fait définie par `eq` :

```
Inductive Definition eq [A:Set;x:A] : A -> Prop
    = refl_equal : <A>x = x.
```

C'est la plus petite relation binaire réflexive dans `A`. Son principe d'induction est :

```
(A:Set)(x:A)(P:A -> Prop)(P x) -> (a:A)(<A>x = a) -> (P a)
```

Il signifie : si `x` et `a` sont égaux alors pour tout prédicat `P` sur `A`, `(P x)` $\Rightarrow$ `(P a)`. Il s'agit donc d'une égalité au sens de Leibniz.

5. Le mécanisme des définitions inductives, bien que très puissant, souffre de quelques restrictions concernant le type des constructeurs (voir [17, 3, 19]).

2.3 Les définitions récursives

À chaque définition inductive I correspond un *récurseur* R_I. Il définit une fonction par récurrence structurelle sur les termes de type I. R_{nat}, par exemple, est défini par :

$$\begin{cases} R_{nat}(C,x,f,0) & \equiv & x \\ R_{nat}(C,x,f,(S\ n)) & \equiv & (f\ n\ R_{nat}(C,x,f,n)) \end{cases}$$

où :
- C est un type,
- x est de type C,
- et f est une fonction de type $nat \longrightarrow C \longrightarrow C$.

$R_{nat}(C,x,f,n)$ est alors de type C. Il est représenté dans Coq par $(< C > \ Match\ n\ with\ x\ f)$.

`Match` nous permet d'écrire des fonctions récursives par filtrage sur des termes inductivement définis. Considérons l'exemple de `plus` :

```
Definition plus = [n,m:nat](<nat>Match n with
    (* O *)     m
    (* (S p) *) [p,plus_p_m:nat] (S plus_p_m)).
```

On reconnaît derrière cette définition (dont la syntaxe est d'un abord assez difficile), les équations suivantes :

$$\begin{cases} (plus\ 0\ m) & \equiv & m \\ (plus\ (S\ n)\ m) & \equiv & (S\ (plus\ n\ m)) \end{cases}$$

D'une manière générale, si t est un terme inductif de type I, alors $(< C > \ Match\ t\ with\ f_1 \ldots f_n)$ est une fonction de type $I \longrightarrow C$, où f_i correspond au traitement du $i^{\text{ème}}$ constructeur de I. Il implémente le récurseur associé à I.

2.4 Le développement de preuves dans Coq

L'utilisateur a le choix entre plusieurs syntaxes pour formuler l'énoncé à prouver. En voici quelques-unes :

$Theorem < nom\_du\_théorème > : < énoncé > .$
$Lemma < nom\_du\_lemme > : < énoncé > .$
$Remark < nom\_de\_la\_remarque > : < énoncé > .$

Cet énoncé est alors appelé *but* initial. D'une manière générale, un but est la donnée d'une formule à prouver sous certaines hypothèses. Celles-ci forment le *contexte local d'hypothèses* du but. Dans le cas d'un

but initial, le contexte local d'hypothèses est vide.

Le développement d'une preuve se fait de manière interactive, par l'usage de commandes appelées *tactiques*. L'application d'une tactique à un but peut soit le résoudre, soit le transformer en de nouveaux buts (chacun avec son contexte local d'hypothèses). Le système affiche, à chaque étape, les buts restant à prouver. La preuve s'achève lorsqu'aucun but ne subsiste.

Il existe des tactiques d'introduction (`Intro` ...), de résolution (`Apply`, `Assumption`, `Split`, `Left`, `Right`, `Exists` ...), d'élimination (`Elim` ...) et de conversion (`Simpl`, `Unfold` ...) :

- `Intro` : appliquée à un but de la forme `(x :N)M` (ou `N -> M`), introduit l'hypothèse `x :N` (resp. `N`) dans le contexte local et transforme le but en `M`.
- `Apply` : réduit le but à prouver en d'autres plus élémentaires par l'application d'un axiome ou un théorème déjà prouvé.
- `Assumption.` cherche dans le contexte local une hypothèse identique au but courant.
- `Split` : transforme le but `N /\ M` en les deux buts `N` et `M`.
- `Left` : transforme le but `N \/ M` en `N`.
- `Right` : transforme le but `N \/ M` en `M`.
- `Exists u` : transforme le but `<M>Ex([x :M]N)` en `N{u/x}` (i.e. `N` dans lequel toutes les occurrences libres de `x` sont remplacées par `u`), `u` étant un terme de type `M`.
- `Elim H` : permet de faire des preuves par induction. Cette tactique applique le principe d'élimination associé à la définition inductive de `H` au but courant. `H` doit correspondre à un terme inductivement défini (exemples : `H :nat` ou `H :(even x)`).
- `Simpl` : applique tous les schémas de récursion présents dans le but.
- `Unfold nom` : remplace dans le but les occurrences de `nom` par sa définition.

Nous allons maintenant voir l'effet de toutes ces tactiques sur un exemple concret. L'exemple est tiré du paquetage `Arith`. On veut prouver :

$$\forall n, m, p, q.\, n \leq m \,\&\, p \leq q \Rightarrow n + p \leq m + q$$

On suppose avoir déjà prouvé le résultat suivant :

$$\forall n, m, p.\, n \leq m \Rightarrow p + n \leq p + m$$

dont l'énoncé dans Coq est :

```
Lemma le_reg_l : (n,m,p:nat)(le n m) -> (le (plus p n) (plus p m)).
```

On commence par énoncer la formule qu'on veut prouver :

```
Lemma le_plus_plus :
       (n,m,p,q:nat) (le n m) -> (le p q) -> (le (plus n p) (plus m q)).
```

On présente, à chaque étape, la tactique utilisée suivie d'un commentaire.

- `Goal.` La preuve commence.

- `Intros n m p q H H0.` `Intros` itère la tactique `Intro`. Le contexte local, initialement vide, contient maintenant 6 hypothèses.

```
1 subgoal
  (le (plus n p) (plus m q))
  ============================
    H0 : (le p q)
    H : (le n m)
    q : nat
    p : nat
    m : nat
    n : nat
```

- `Elim H.` Application du principe d'induction associé à `le`.

```
2 subgoals
  (le (plus n p) (plus n q))
  ============================
    H0 : (le p q)
    H : (le n m)
    q : nat
    p : nat
    m : nat
    n : nat
subgoal 2 is:
  (m0:nat)
   (le n m0) ->
    (le (plus n p) (plus m0 q)) -> (le (plus n p) (plus (S m0) q))
```

- `Apply le_reg_l.` D'après le lemme `le_reg_l`, pour prouver $n + p \leq n + q$, il suffit de prouver $p \leq q$.

```
2 subgoals
  (le p q)
  ============================
    H0 : (le p q)
    H : (le n m)
    q : nat
    p : nat
    m : nat
    n : nat
subgoal 2 is:
  (m0:nat)
   (le n m0) ->
    (le (plus n p) (plus m0 q)) -> (le (plus n p) (plus (S m0) q))
```

- `Assumption.` Le but est vrai par hypothèse.

```
1 subgoal
  (m0:nat)
   (le n m0) ->
    (le (plus n p) (plus m0 q)) -> (le (plus n p) (plus (S m0) q))
```

```
============================
  H0 : (le p q)
  H : (le n m)
  q : nat
  p : nat
  m : nat
  n : nat
```

- `Intros m0 H1 H2.`

```
1 subgoal
  (le (plus n p) (plus (S m0) q))
  ============================
    H2 : (le (plus n p) (plus m0 q))
    H1 : (le n m0)
    m0 : nat
    H0 : (le p q)
    H : (le n m)
    q : nat
    p : nat
    m : nat
    n : nat
```

- `Simpl.` Applique le schéma de récursion $(S\ m0) + q \equiv (S\ (m0 + q))$.

```
1 subgoal
  (le (plus n p) (S (plus m0 q)))
  ============================
    H2 : (le (plus n p) (plus m0 q))
    H1 : (le n m0)
    m0 : nat
    H0 : (le p q)
    H : (le n m)
    q : nat
    p : nat
    m : nat
    n : nat
```

- `Apply le_S.` `le_S` est une clause dans la définition (inductive) de `le`. Elle stipule que : $\forall n, m.n \leq m \Rightarrow n \leq (S\ m)$.

```
1 subgoal
  (le (plus n p) (plus m0 q))
  ============================
    H2 : (le (plus n p) (plus m0 q))
    H1 : (le n m0)
    m0 : nat
    H0 : (le p q)
    H : (le n m)
    q : nat
```

```
p : nat
m : nat
n : nat
```

- `Assumption.`

  ```
  Goal proved!
  ```

- `Save.` On sauve la preuve dans le système pour pouvoir l'utiliser dans d'autres développements de preuves.

  ```
  le_plus_plus is defined
  ```

3 Les relations dans Coq

Les relations modélisant des processus de calcul sont appelées *relations de réduction*. $a\ R\ b$, souvent noté $a \xrightarrow{R} b$, se lit "a se réduit en b", et b est dit *réduit* de a.

Soit A un ensemble quelconque. Nous donnons dans cette section quelques propriétés fondamentales des relations binaires R sur A. Nous verrons aussi leur codage dans Coq.

3.1 Définitions générales

Commençons tout d'abord par déclarer A et R. R est défini par sa fonction caractéristique, i.e. la fonction qui prend deux éléments de A et décide si le second est un réduit du premier.

```
Variable A : Set.

Variable R : A -> A -> Prop.
```

Nous aurons souvent besoin des définitions de base suivantes :

- la composition de relations $R_1\ R_2 = \{(x, y) \mid \exists z\ x\ R_1\ z\ \&\ z\ R_2\ y\}$ se traduit dans Coq par `(comp_rel A R1 R2)` :

  ```
  Inductive Definition comp_rel [R1,R2:A -> A -> Prop] : A -> A -> Prop
  = comp_2rel : (x,y,z:A)(R1 x y) -> (R2 y z) -> (comp_rel R1 R2 x z).
  ```

  N.B. La paramétrisation de cette notion, comme des suivantes, relativement à l'ensemble A est assurée automatiquement par le mécanisme des sections[7].

- $R_1 \subseteq R_2$, où $\subseteq$ dénote l'inclusion de relations, est codé en `(inclus A R1 R2)` :

  ```
  Definition inclus [A:Set][R1,R2:A -> A -> Prop]
        (x,y:A)(R1 x y) -> (R2 x y).
  ```

- R^+ est la clôture transitive de R, elle est définie dans Coq par `(rel_plus A R)` :

  ```
  Inductive Definition rel_plus [R:A -> A -> Prop] : A -> A -> Prop
  =  relplus_1step  : (x,y:A)(R x y) -> (rel_plus R x y)
   | relplus_trans1 : (x,y,z:A)(R x y) -> (rel_plus R y z) -> (rel_plus R x z).
  ```

- R^* est la clôture réflexive transitive de R, elle est définie dans Coq par `(star A R)` :

```
Inductive Definition star [R:A -> A -> Prop] : A -> A -> Prop
=  star_refl   : (x:A)(star R x x)
 | star_trans1 : (x,y,z:A)(R x y) -> (star R y z) -> (star R x z).
```

  R^* est définie comme étant la plus petite relation vérifiant :

$$\frac{}{x\ R^*\ x}\,(star\_refl) \qquad\qquad \frac{x\ R\ y \quad y\ R^*z}{x\ R^*\ z}\,(star\_trans1)$$

  `star_trans1` est un cas particulier de la transitivité, suffisant toutefois pour prouver que `(star A R)` est transitive :

```
Lemma star_trans:(A:Set)(R:A->A->Prop)(x,y,z:A)
                 (star A R x y)->(star A R y z)->(star A R x z).
```

3.2 La confluence

Une des propriétés fondamentales des relations est la confluence. Elle exprime une propriété de déterminisme du calcul vis-à-vis du chemin de réduction suivi. Nous suivons [11] pour le codage de cette propriété dans Coq.

Soit $x \in A$. La relation binaire R sur A est :

- *confluente* en x, si pour tout y, z, si $x\ R^*\ y$, et $x\ R^*\ z$, il existe u tel que $y\ R^*\ u$, et $z\ R^*\ u$.

```
Definition confluence_en [x:A] (y,z:A)(star A R x y) -> (star A R x z) ->
                         <A>Ex([u:A] (star A R y u) /\ (star A R z u)).
```

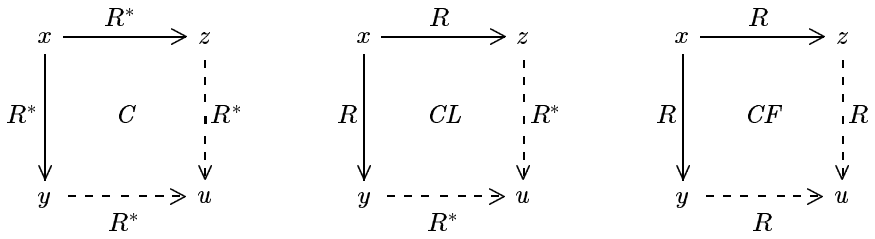
- *localement confluente* en x, si pour tout y, z, si $x\ R\ y$, et $x\ R\ z$, il existe u tel que $y\ R^*\ u$, et $z\ R^*\ u$.

```
Definition local_confluence_en [x:A] (y,z:A)(R x y) -> (R x z) ->
                         <A>Ex([u:A] (star A R y u) /\ (star A R z u)).
```

- *fortement confluente* en x, si pour tout y, z, si $x\ R\ y$, et $x\ R\ z$, il existe u tel que $y\ R\ u$, et $z\ R\ u$.

```
Definition strong_confluence_en [x:A] (y,z:A)(R x y) -> (R x z) ->
                              <A>Ex([u:A] (R y u) /\ (R z u)).
```

Ces propriétés s'expriment par les diagrammes suivants, où un trait plein indique une hypothèse ("si ...") et un trait pointillé une conclusion ("il existe ..."):



Une relation binaire est confluente (resp. localement confluente, fortement confluente) si elle l'est en tout point de A :

```
Definition confluence (x:A)(confluence_en x).

Definition local_confluence (x:A)(local_confluence_en x).

Definition strong_confluence (x:A)(strong_confluence_en x).
```

3.3 La nœthérianité

Une autre propriété importante des relations est la *nœthérianité*. Elle nous garantit la terminaison des calculs quelle que soit la stratégie de réduction adoptée. Une relation R est nœthérienne s'il n'existe pas de suite infinie de la forme $a_0\ R\ a_1\ R\ \ldots$. Il s'avère assez difficile de donner une version constructive de cette notion. La traduction que nous en donnons est due à Huet [12]. Elle nécessite l'introduction de plusieurs notions intermédiaires.

Un sous-ensemble de A est défini par sa fonction caractéristique :

```
Definition a_set A -> Prop.
```

Nous définissons l'inclusion $\subseteq_A$ sur ces sous-ensembles par :

```
Definition sub [Y,Z:a_set] (x:A)(Y x) -> (Z x).
```

On dit qu'un sous-ensemble est *universel* s'il contient tous les éléments de A :

```
Definition universal [Y:a_set] (x:A)(Y x).
```

`(R x)` représente l'ensemble des réduits de x par R, $R_x = \{y \in A |\ x\ R\ y\}$. L'*adjoint* d'un sous-ensemble Y est alors $\{x \in A |\ R_x \subseteq_A Y\}$:

```
Definition adjoint : a_set -> a_set = [Y:a_set][x:A] (sub (R x) Y).
```

Y est *héréditaire* s'il contient son adjoint :

```
Definition hereditary  [Y:a_set] (sub (adjoint Y) Y).
```

On dit alors que R est nœthériennne si tout sous-ensemble héréditaire est universel :

```
Definition noetherian (Y:a_set)(hereditary Y) -> (universal Y).
```

On peut ainsi prouver dans le système le principe de récurrence nœthérienne. C'est ce principe qui est, comme on le verra plus loin, à la base des preuves du lemme de Newman et du lemme de Yokouchi.

```
Lemma  noetherian_induction: (A:Set)
                             (R:A -> A -> Prop)
                             (noetherian A R) ->
                             (x:A)
                             (P:A -> Prop)
                             ((y:A)((z:A)(rel_plus A R y z) -> (P z)) -> (P y)) ->
                             (P x).
```

qu'on lit :

Si R est nœthérienne et si P est un prédicat sur A tel que :

$$\forall y \in A.(\forall z \in A.\, y\, R^+\, z \Rightarrow P(z)) \Rightarrow P(y)$$

Alors, $\forall x \in A.\, P(x)$.

Remarquez que `noetherian_induction` est un lemme prouvable, et non pas un axiome supplémentaire. Autrement dit, R nœthérienne implique logiquement que la récurrence nœthérienne est valide.

Une autre définition possible pour la nœthérianité est : R est nœthérienne si et seulement si R^{-1} est bien fondée, où R^{-1} est la relation symétrique de R ($a\, R^{-1}\, b \Leftrightarrow b\, R\, a$).

```
Definition symmetric [a,b:A] (R b a).
```

Une relation binaire R sur A est bien fondée si toute partie A' non vide de A contient un élément a_0 minimal dans A, i.e. tel que $\neg(a\, R\, a_0)$ pour tout $a \in A$. Elle est définie dans Coq grâce à la notion d'accessibilité :

```
Inductive Definition Acc [A:Set][R:A -> A -> Prop] : A -> Prop
   = Acc_intro : (x:A)((y:A)(R y x) -> (Acc A R y)) -> (Acc A R x).
```

R est bien fondée si tous les éléments de A lui sont accessibles :

```
Definition well_founded [A:Set][R:A -> A -> Prop] (a:A)(Acc A R a).
```

On définit alors la nœthérianité par :

```
Definition noetherian' [A:Set][R:A -> A -> Prop]
                       (well_founded A (symmetric A R)).
```

L'équivalence des deux définitions de la nœthérianité a été prouvée dans le système Coq par Gérard Huet :

```
Lemma equivalence_noetherian_noetherian'
               [A:Set][R:A -> A -> Prop]
               (noetherian A R) <-> (noetherian' A R).
```

Le résultat `(noetherian A R) -> (noetherian' A R)` ne doit pas nous étonner car la définition de `noetherian` peut s'écrire :

```
(P:A -> Prop)((y:A)((z:A)(R y z) -> (P z)) -> (P x)) -> (x:A)(P x)
```

en remplaçant dans la définition de `noetherian` toutes les constantes qui y apparaissent par leur définition (successivement `universal`, `hereditary`, `adjoint`, `sub` et `a_set`). Or cette formule n'est autre que le principe de récurrence bien fondée pour R^{-1}.

4 Codage de la réécriture dans Coq

Dans cette section, nous proposons un codage général pour les systèmes de réécriture de termes du premier ordre. Les définitions inductives utilisées sont écrites dans une pseudo-syntaxe Coq.

Notation. Pour éviter l'utilisation de points de suspension, nous utilisons la notation vectorielle suivante :

$\vec{t}$	dénote la séquence de termes $t_1 \ldots t_n$,
$(\vec{x} : \vec{A})$	dénote $(x_1 : A_1) \ldots (x_n : A_n)$.

4.1 Algèbre

Une signature est définie par la donnée d'un ensemble fini $\{s_1, \ldots, s_n\}$ de sortes et d'un ensemble fini Σ d'opérateurs (ou constructeurs). Chaque opérateur C est muni d'une arité (type) $s'_1 \longrightarrow \cdots \longrightarrow s'_m$ $(m \geq 1)$, où les s'_j sont des sortes $(1 \leq j \leq m)$. On dit alors que C est de sorte s'_m. On dira aussi que C est une constante lorsque $m = 1$. L'algèbre (initiale) des termes de sorte s_i est (inductivement) définie par :

- si C est une constante de sorte s_i, alors $C \in s_i$
- si C est un opérateur d'arité $s_{i_1} \longrightarrow \cdots \longrightarrow s_{i_m} \longrightarrow s_i$ et si $t_j \in s_{i_j}$ $(1 \leq j \leq m)$, alors $(C\ t_1 \ldots t_m) \in s_i$

Par exemple, $(C_1\ C_2\ C_5)$ est un terme de l'algèbre — en supposant qu'il soit bien typé. On peut voir ce terme comme une instance du terme $(C_1\ C_2\ a)$ ou encore $(C_1\ b\ a)$, i.e. un terme contenant des variables universellement quantifiées (a, b, ...).

Si M est un terme, on notera l'ensemble de ses variables par $var(M)$.

Notons que les définitions des différentes sortes s_i sont mutuellement récursives. Or, on ne dispose pas aujourd'hui d'un tel mécanisme dans Coq. Il est toutefois possible de simuler ce mécanisme à l'aide des types dépendants. On va coder l'algèbre de termes par une définition inductive ALG de type $W \longrightarrow Set$ où W est un type inductif à n constantes :

$Inductive\ Set\ W = W_1 : W | \ldots | W_n : W.$

Chaque clause CC_i de ALG correspond à un opérateur C_i de Σ. Son type $s_{i,1} \longrightarrow \cdots \longrightarrow s_{i,m_i}$ est traduit en $(ALG\ W_{i,1}) \longrightarrow \cdots \longrightarrow (ALG\ W_{i,m_i})$.

$$
\begin{array}{lcl}
Inductive\ Definition\ ALG :\ W \longrightarrow Set & & \\
= CC_1 & : & (ALG\ W_{1,1}) \longrightarrow \cdots \longrightarrow (ALG\ W_{1,m_1}) \\
 & & \vdots \\
| CC_i & : & (ALG\ W_{i,1}) \longrightarrow \cdots \longrightarrow (ALG\ W_{i,m_i}) \\
 & & \vdots \\
| CC_r & : & (ALG\ W_{r,1}) \longrightarrow \cdots \longrightarrow (ALG\ W_{r,m_r}).
\end{array}
$$

Dans la définition ci-dessus, $(ALG\ W_i)$ représente s_i :

$Definition\ s_1\ (ALG\ W_1).$
$\vdots$
$Definition\ s_n\ (ALG\ W_n).$

4.2 Système de réécriture

Un système de réécriture de termes du premier ordre est la donnée d'une signature et d'un ensemble R de couples de termes (α_i, β_i) tels que $var(\beta_i) \subseteq var(\alpha_i)$ et où α_i n'est pas une variable. De plus α_i et β_i doivent être de même sorte.

On code ce système de réécriture par une définition inductive de type $(w : W)(ALG\ w) \longrightarrow (ALG\ w) \longrightarrow Prop$, où w sert à indiquer la sorte des arguments :

$$\begin{array}{lll} \textit{Inductive Definition } systemR: & (w:W)(ALG\ w) \longrightarrow (ALG\ w) \longrightarrow Prop & \\ = rule_1 & : & \forall \vec{a}_1 \in \vec{s}_1.(systemR\ w_1\ \alpha_1\ \beta_1) \\ & & \vdots \\ \mid rule_i & : & \forall \vec{a}_i \in \vec{s}_i.(systemR\ w_i\ \alpha_i\ \beta_i) \\ & & \vdots \\ \mid rule_m & : & \forall \vec{a}_m \in \vec{s}_m.(systemR\ w_m\ \alpha_m\ \beta_m). \end{array}$$

Chaque règle (α_i, β_i) est représentée par une clause $rule_i$, où les quantifications portent sur les variables de $var(\alpha_i)$. $w_i \in \{W_1, \ldots W_n\}$ indique la sorte de la règle.

4.3 Relation de réécriture

Nous considérons maintenant la relation engendrée par le système R (notée $\xrightarrow{R}$). On la construit en rajoutant les règles de congruence (passage au contexte) :

$$a_i \xrightarrow{R} b_i \Rightarrow (C\ a_1 \cdots a_i \cdots a_n) \xrightarrow{R} (C\ a_1 \cdots b_i \cdots a_n)$$

où C est un opérateur (qui ne soit pas une constante). Dans Coq, ces règles correspondent aux clauses dont les noms commencent par *R_context_*.

La forme générale d'une clause $R\_context\_C_{i,j}$ est la suivante :

$$\begin{array}{lll} R\_context\_C_{i,j} & : & \forall \vec{a}_i \in \vec{s}_i.\forall b_{i,j} \in s_{i,j}. \\ & & (relR\ w_{i,j}\ a_{i,j}\ b_{i,j}) \Rightarrow (relR\ w_i\ (C_i\ a_{i,1} \ldots a_{i,j} \ldots a_{i,m_i})\ (C_i\ a_{i,1} \ldots b_{i,j} \ldots a_{i,m_i})) \end{array}$$

avec :

- $1 \leq i \leq r$, r étant le nombre d'opérateurs de Σ
- $1 \leq j \leq m_i$, m_i étant le nombre d'argument de C_i (dont l'arité est $s_{i,1} \longrightarrow \cdots s_{i,m_i} \longrightarrow s_i$)
- on a $s_{i,j} \equiv (ALG\ w_{i,j})$ et $s_i \equiv (ALG\ w_i)$

on a ainsi $(\Sigma_{i=1}^{r} m_i)$ clauses $R\_context\_$:

$$\begin{array}{lll} \textit{Inductive Definition } relR: & (w:W)(ALG\ w) \longrightarrow (ALG\ w) \longrightarrow Prop & \\ = rule & : & \forall w \in W.\forall a, b \in (ALG\ w).\ (systemR\ w\ a\ b) \Rightarrow (relR\ w\ a\ b) \\ \mid R\_context\_C_{1,1} & : & \forall \vec{a}_1 \in \vec{s}_1.\forall b_{1,1} \in s_{1,1}. \\ & & (relR\ w_{1,1}\ a_{1,1}\ b_{1,1}) \Rightarrow (relR\ w_1\ (C_1\ a_{1,1} \ldots a_{1,m_1})\ (C_1\ b_{1,1} \ldots a_{1,m_1})) \\ & & \vdots \\ \mid R\_context\_C_{r,m_r} & : & \forall \vec{a}_r \in \vec{s}_r.\forall b_{r,m_r} \in s_{r,m_r}. \\ & & (relR\ w_{r,m_r}\ a_{r,m_r}\ b_{r,m_r}) \Rightarrow (relR\ w_r\ (C_r\ a_{r,1} \ldots a_{r,m_r})\ (C_r\ a_{r,1} \ldots b_{r,m_r})). \end{array}$$

Remarques.

1. Le type de $relR$ est $(w : W)(ALG\ w) \longrightarrow (ALG\ w) \longrightarrow Prop$. Il ne s'agit manifestement pas d'une relation puisque le type des relations est `A -> A -> Prop` où `A:Set`. En fait, $relR$ correspond à autant de relations qu'il y a de sortes : $(ALG\ w_i) \longrightarrow (ALG\ w_i) \longrightarrow Prop$ est la relation sur les termes de sorte s_i.

2. Formellement, la relation $\stackrel{R}{\longrightarrow}$ engendrée par un système de réécriture est la plus petite relation stable par substitution et compatible avec les opérateurs de Σ. On peut aussi définir cette relation de manière axiomatique par :

   (a) pour toute substitution θ et pour toute règle (α, β), on a $\theta(\alpha) \stackrel{R}{\longrightarrow} \theta(\beta)$

   (b) pour tout $C \in \Sigma$, $a_i \stackrel{R}{\longrightarrow} b_i \Rightarrow (C\ a_1 \cdots a_i \cdots a_n) \stackrel{R}{\longrightarrow} (C\ a_1 \cdots b_i \cdots a_n)$

   Une substitution θ est un homomorphisme qui associe des termes à des variables. $\theta(\alpha)$ est alors le résultat de la substitution des variables contenues dans α, disons v_i, par leur image $\theta(v_i)$.
   Le deuxième point est reporté tel quel dans la définition de $relR$. Quant au premier, au lieu de quantifier sur les substitutions θ, on quantifie sur les variables elles-mêmes. Toute règle (α, β) du système R est codée dans $systemR$ par une clause $rule_i$:

   $$\forall \vec{a} \in \vec{s}.(systemR\ w\ \alpha\ \beta)$$

   où w sert à indiquer la sorte de la règle, $\vec{a} = var(\alpha)$ et $\vec{s}$ sont des sortes. Or dans Coq, si on sait prouver $\forall \vec{a} \in \vec{s}.P$, on peut en déduire une preuve de $P\{\vec{t}/\vec{a}\}$ (avec $\vec{t} \in \vec{s}$) grâce à la règle d'élimination du $\forall$:

   $$\frac{\forall a \in A.P}{P\{t/a\}}\,(t \in A)$$

   où $\{t/a\}$ est l'opération de substitution de la variable a par un terme t.

   En prenant $(systemR\ w\ \alpha\ \beta)$ à la place de P, on voit bien qu'on pourra prouver :

   $$(systemR\ w\ \alpha\{\vec{t}/\vec{a}\}\ \beta\{\vec{t}/\vec{a}\})$$

   qui n'est rien d'autre que $(systemR\ w\ \theta(\alpha)\ \theta(\beta))$, θ étant la substitution $\{\vec{t}/\vec{a}\}$.

   Ainsi on n'a pas à coder la notion de substitution puisqu'elle nous est gracieusement fournie par le système.

5 Axiomatisation du $\lambda\sigma_{\Uparrow}$-calcul : Syntaxe et Règles

Le $\lambda\sigma_{\Uparrow}$-calcul est un système de réécriture de termes. Il possède deux sortes (les termes et les substitutions explicites) inductivement définies par :

Termes	$a ::= \mathtt{n} \mid aa \mid \lambda a \mid a[s] \mid X_i$
Substitutions Explicites	$s ::= id \mid \uparrow \mid a \cdot s \mid s \circ s \mid \Uparrow(s) \mid x_i$

Les termes et les substitutions explicites sont appelés $\lambda\sigma_{\Uparrow}$-termes. L'application est une opération binaire notée par juxtaposition (aa). L'opération $\circ$ dans $s \circ s$ dénote la composition des substitutions explicites. Quant aux symboles $\uparrow$ et $\Uparrow$, ils sont respectivement appelés *shift* et *lift*. X_i et x_i sont des métavariables, il ne faut pas confondre métavariables et variables : une métavariable est un élément de l'algèbre, alors qu'une variable est un nom représentant un terme quelconque de l'algèbre.

Dans tous les $\lambda\sigma$-calculs, et en particulier dans le $\lambda\sigma_{\Uparrow}$-calcul, les indices de de Bruijn utilisés sont toujours strictement positifs. Cependant ce choix n'est pas primordial et rien dans la théorie ne nous empêche de commencer à compter à partir de 0. Dans notre codage, on a choisi de prendre $n \in \mathbb{N}$ pour pouvoir bénéficier de l'axiomatisation de $\mathbb{N}$ déjà disponible dans Coq. Si on avait pris $n \in \mathbb{N}^*$, on aurait été obligé d'indiquer à chaque utilisation de n que $n > 0$.

Les définitions des termes et substitutions explicites sont mutuellement récursives. Pour les coder, on va utiliser le mécanisme décrit en 4.

```
Inductive Set wsort = ws : wsort | wt : wsort.

Inductive Definition TS : wsort -> Set
=   var    : nat -> (TS wt)
  | app    : (TS wt) -> (TS wt) -> (TS wt)
  | lambda : (TS wt) -> (TS wt)
  | env    : (TS wt) -> (TS ws) -> (TS wt)
  | id     : (TS ws)
  | shift  : (TS ws)
  | cons   : (TS wt) -> (TS ws) -> (TS ws)
  | comp   : (TS ws) -> (TS ws) -> (TS ws)
  | lift   : (TS ws) -> (TS ws)
  | meta_X : nat -> (TS wt)
  | meta_x : nat -> (TS ws).
```

`(TS wt)` et `(TS ws)` correspondent alors respectivement aux termes et aux substitutions explicites :

```
Definition terms (TS wt).

Definition sub_explicits (TS ws).
```

Le $\lambda\sigma_{\Uparrow}$-calcul est défini par les règles de la figure 1.

Dans Coq, nous avons choisi de représenter chaque règle par une définition inductive. Nous gardons ainsi la possibilité de former des sous-systèmes du $\lambda\sigma_{\Uparrow}$-calcul en vue de les étudier. Comme exemple, nous donnons le codage des trois premières règles :

```
Inductive Definition reg_beta : (b:wsort)(TS b) -> (TS b) -> Prop
= reg1_beta : (a,b:terms)
        (reg_beta wt (app (lambda a) b) (env a (cons b id))).

Inductive Definition reg_app : (b:wsort)(TS b) -> (TS b) -> Prop
=  reg1_app : (a,b:terms)(s:sub_explicits)
        (reg_app wt (env (app a b) s) (app (env a s) (env b s))).

Inductive Definition reg_lambda : (b:wsort)(TS b) -> (TS b) -> Prop
=  reg1_lambda : (a:terms)(s:sub_explicits)
        (reg_lambda wt (env (lambda a) s) (lambda (env a (lift s)))).
```

Le sous-système contenant toutes les règles sauf *Beta* est appelé $\sigma_{\Uparrow}$. Il s'agit du système qui implémente l'opération de substitution :

(Beta)	$(\lambda a)b$	$\longrightarrow$	$a[b \cdot id]$
(App)	$(ab)[s]$	$\longrightarrow$	$(a[s])\ (b[s])$
(Lambda)	$(\lambda a)[s]$	$\longrightarrow$	$\lambda(a[\Uparrow s])$
(Clos)	$(a[s])[t]$	$\longrightarrow$	$a[s \circ t]$
(VarShift1)	$\mathtt{n}[\uparrow]$	$\longrightarrow$	$\mathtt{n+1}$
(VarShift2)	$\mathtt{n}[\uparrow \circ s]$	$\longrightarrow$	$\mathtt{n+1}[s]$
(FVarCons)	$\mathtt{0}[a \cdot s]$	$\longrightarrow$	a
(FVarLift1)	$\mathtt{0}[\Uparrow(s)]$	$\longrightarrow$	$\mathtt{0}$
(FVarLift2)	$\mathtt{0}[\Uparrow(s) \circ t]$	$\longrightarrow$	$\mathtt{0}[t]$
(RVarCons)	$\mathtt{n+1}[a \cdot s]$	$\longrightarrow$	$\mathtt{n}[s]$
(RVarLift1)	$\mathtt{n+1}[\Uparrow(s)]$	$\longrightarrow$	$\mathtt{n}[s \circ \uparrow]$
(RVarLift2)	$\mathtt{n+1}[\Uparrow(s) \circ t]$	$\longrightarrow$	$\mathtt{n}[s \circ (\uparrow \circ t)]$
(AssEnv)	$(s \circ t) \circ u$	$\longrightarrow$	$s \circ (t \circ u)$
(MapEnv)	$(a \cdot s) \circ t$	$\longrightarrow$	$a[t] \cdot (s \circ t)$
(ShiftCons)	$\uparrow \circ (a \cdot s)$	$\longrightarrow$	s
(ShiftLift1)	$\uparrow \circ \Uparrow(s)$	$\longrightarrow$	$s \circ \uparrow$
(ShiftLift2)	$\uparrow \circ (\Uparrow(s) \circ t)$	$\longrightarrow$	$s \circ (\uparrow \circ t)$
(Lift1)	$\Uparrow(s) \circ \Uparrow(t)$	$\longrightarrow$	$\Uparrow(s \circ t)$
(Lift2)	$\Uparrow(s) \circ (\Uparrow(t) \circ u)$	$\longrightarrow$	$\Uparrow(s \circ t) \circ u$
(LiftEnv)	$\Uparrow(s) \circ (a \cdot t)$	$\longrightarrow$	$a \cdot (s \circ t)$
(IdL)	$id \circ s$	$\longrightarrow$	s
(IdR)	$s \circ id$	$\longrightarrow$	s
(LiftId)	$\Uparrow(id)$	$\longrightarrow$	id
(Id)	$a[id]$	$\longrightarrow$	a

Figure 1 : Le système de réécriture $\lambda\sigma_{\Uparrow}$

```
Inductive Definition systemSL : (b:wsort)(TS b) -> (TS b) -> Prop
=  regle_app        : (a,b:terms)(reg_app wt a b) -> (systemSL wt a b)
 | regle_lambda     : (a,b:terms)(reg_lambda wt a b) -> (systemSL wt a b)
 | regle_clos       : (a,b:terms)(reg_clos wt a b) -> (systemSL wt a b)
 | regle_varshift1  : (a,b:terms)(reg_varshift1 wt a b) -> (systemSL wt a b)
 | regle_varshift2  : (a,b:terms)(reg_varshift2 wt a b) -> (systemSL wt a b)
 | regle_fvarcons   : (a,b:terms)(reg_fvarcons wt a b) -> (systemSL wt a b)
 | regle_fvarlift1  : (a,b:terms)(reg_fvarlift1 wt a b) -> (systemSL wt a b)
 | regle_fvarlift2  : (a,b:terms)(reg_fvarlift2 wt a b) -> (systemSL wt a b)
 | regle_rvarcons   : (a,b:terms)(reg_rvarcons wt a b) -> (systemSL wt a b)
 | regle_rvarlift1  : (a,b:terms)(reg_rvarlift1 wt a b) -> (systemSL wt a b)
 | regle_rvarlift2  : (a,b:terms)(reg_rvarlift2 wt a b) -> (systemSL wt a b)
 | regle_assenv     : (s,t:sub_explicits)(reg_assenv ws s t) -> (systemSL ws s t)
 | regle_mapenv     : (s,t:sub_explicits)(reg_mapenv ws s t) -> (systemSL ws s t)
 | regle_shiftcons  : (s,t:sub_explicits)(reg_shiftcons ws s t) -> (systemSL ws s t)
 | regle_shiftlift1 : (s,t:sub_explicits)(reg_shiftlift1 ws s t) ->
                      (systemSL ws s t)
 | regle_shiftlift2 : (s,t:sub_explicits)(reg_shiftlift2 ws s t) ->
                      (systemSL ws s t)
 | regle_lift1      : (s,t:sub_explicits)(reg_lift1 ws s t) -> (systemSL ws s t)
 | regle_lift2      : (s,t:sub_explicits)(reg_lift2 ws s t) -> (systemSL ws s t)
 | regle_liftenv    : (s,t:sub_explicits)(reg_liftenv ws s t) -> (systemSL ws s t)
 | regle_idl        : (s,t:sub_explicits)(reg_idl ws s t) -> (systemSL ws s t)
 | regle_idr        : (s,t:sub_explicits)(reg_idr ws s t) -> (systemSL ws s t)
 | regle_liftid     : (s,t:sub_explicits)(reg_liftid  ws s t) -> (systemSL ws s t)
 | regle_id         : (a,b:terms)(reg_id wt a b) -> (systemSL wt a b).
```

Nous définissons enfin le système $\lambda\sigma_{\Uparrow}$:

```
Inductive Definition systemLSL : (b:wsort)(TS b) -> (TS b) -> Prop
=  beta1 : (M,N:terms)(reg_beta wt M N) -> (systemLSL wt M N)
 | SL1   : (b:wsort)(M,N:(TS b))(systemSL b M N) -> (systemLSL b M N).
```

Nous considérons maintenant la relation engendrée par $\sigma_{\Uparrow}$. `(relSL b M N)` signifie $M \xrightarrow{\sigma_{\Uparrow}} N$, `b` (de type `wsort`) indiquant la sorte (termes ou substitutions explicites) des arguments.

```
Inductive Definition relSL : (b:wsort)(TS b) -> (TS b) -> Prop
=  SL_one_regle      : (b:wsort)(M,N:(TS b))(systemSL b M N) -> (relSL b M N)
 | SL_context_app_l  : (a,a',b:terms)(relSL wt a a') ->
                       (relSL wt (app a b) (app a' b))
 | SL_context_app_r  : (a,b,b':terms)(relSL wt b b') ->
                       (relSL wt (app a b) (app a b'))
 | SL_context_lambda : (a,a':terms)(relSL wt a a') ->
                       (relSL wt (lambda a) (lambda a'))
 | SL_context_env_t  : (a,a':terms)(s:sub_explicits)(relSL wt a a') ->
                       (relSL wt (env a s) (env a' s))
 | SL_context_env_s  : (a:terms)(s,s':sub_explicits)(relSL ws s s') ->
                       (relSL wt (env a s) (env a s'))
 | SL_context_cons_t : (a,a':terms)(s:sub_explicits)(relSL wt a a') ->
                       (relSL ws (cons a s) (cons a' s))
 | SL_context_cons_s : (a:terms)(s,s':sub_explicits)(relSL ws s s') ->
                       (relSL ws (cons a s) (cons a s'))
```

```
 | SL_context_comp_l : (s,s',t:sub_explicits)(relSL ws s s') ->
                       (relSL ws (comp s t) (comp s' t))
 | SL_context_comp_r : (s,t,t':sub_explicits)(relSL ws t t') ->
                       (relSL ws (comp s t) (comp s t'))
 | SL_context_lift   : (s,s':sub_explicits)(relSL ws s s') ->
                       (relSL ws (lift s) (lift s')).
```

De même pour la relation engendrée par $\lambda\sigma_{\Uparrow}$, elle est traduite par `relLSL` :

```
Inductive Definition relLSL : (b:wsort)(TS b) -> (TS b) -> Prop
=  LSL_one_regle      : (b:wsort)(M,N:(TS b))(systemLSL b M N) -> (relLSL b M N)
 | LSL_context_app_l  : (a,a',b:terms)(relLSL wt a a') ->
                        (relLSL wt (app a b) (app a' b))
 | LSL_context_app_r  : (a,b,b':terms)(relLSL wt b b') ->
                        (relLSL wt (app a b) (app a b'))
 | LSL_context_lambda : (a,a':terms)(relLSL wt a a') ->
                        (relLSL wt (lambda a) (lambda a'))
 | LSL_context_env_t  : (a,a':terms)(s:sub_explicits)(relLSL wt a a') ->
                        (relLSL wt (env a s) (env a' s))
 | LSL_context_env_s  : (a:terms)(s,s':sub_explicits)(relLSL ws s s') ->
                        (relLSL wt (env a s) (env a s'))
 | LSL_context_cons_t : (a,a':terms)(s:sub_explicits)(relLSL wt a a') ->
                        (relLSL ws (cons a s) (cons a' s))
 | LSL_context_cons_s : (a:terms)(s,s':sub_explicits)(relLSL ws s s') ->
                        (relLSL ws (cons a s) (cons a s'))
 | LSL_context_comp_l : (s,s',t:sub_explicits)(relLSL ws s s') ->
                        (relLSL ws (comp s t) (comp s' t))
 | LSL_context_comp_r : (s,t,t':sub_explicits)(relLSL ws t t') ->
                        (relLSL ws (comp s t) (comp s t'))
 | LSL_context_lift   : (s,s':sub_explicits)(relLSL ws s s') ->
                        (relLSL ws (lift s) (lift s')).
```

Pour des raisons de commodité, nous donnons des noms à $\sigma^*_{\Uparrow}$ et $\lambda\sigma^*_{\Uparrow}$, respectivement :

```
Definition relSLstar [b:wsort] (star (TS b) (relSL b)).

Definition relLSLstar [b:wsort] (star (TS b) (relLSL b)).
```

6 $\sigma_{\Uparrow}$ est nœthérien

Pour montrer qu'une relation (de réécriture) $\xrightarrow{R}$ est nœthérienne, il suffit de trouver une relation nœthérienne $\succ$ sur l'ensemble des termes telle que :

$$a \succ b \;\; si \;\; a \xrightarrow{R} b$$

Établissons ce résultat dans Coq :

```
Lemma noether_inclus : (A:Set)(R,R':A -> A -> Prop)
                       (noetherian A R) ->
                       ((x,y:A)(R' x y) -> (R x y)) ->
                       (noetherian A R').
```

Pour le système $\sigma_{\Uparrow}$, on utilise un ordre lexicographique avec deux composantes polynomiales :

$$a \succ b \;\; ssi \;\; (P_1(a), P_2(a)) \succ_{lex} (P_1(b), P_2(b))$$

où P_1 et P_2 sont deux polynômes définis par :

$$\begin{array}{ll}
P_1(\mathtt{n}) = 2^{n+1} & P_2(\mathtt{n}) = 1 \\
P_1(ab) = P_1(a) + P_1(b) & P_2(ab) = P_2(a) + P_2(b) + 1 \\
P_1(\lambda a)) = P_1(a) + 2 & P_2(\lambda a) = 2 * P_2(a) \\
P_1(a[s]) = P_1(a) * P_1(s) & P_2(a[s]) = P_2(a) * (1 + P_2(s)) \\
P_1(a \cdot s) = P_1(a) + P_1(s) & P_2(a \cdot s) = P_2(a) + P_2(s) + 1 \\
P_1(\uparrow) = 2 & P_2(\uparrow) = 1 \\
P_1(id) = 2 & P_2(id) = 1 \\
P_1(s \circ t) = P_1(s) * P_1(t) & P_2(s \circ t) = P_2(s) * (1 + P_2(t)) \\
P_1(\Uparrow(s)) = P_1(s) & P_2(\Uparrow(s)) = 4 * P_2(s) \\
P_1(X_i) = 2 & P_2(X_i) = 1 \\
P_1(x_i) = 2 & P_2(x_i) = 1
\end{array}$$

Quant à $\succ_{lex}$, il correspond à l'ordre lexicographique sur les couples d'entiers naturels :

$$(n_1, n_2) \succ_{lex} (m_1, m_2) \;\; ssi \;\; \left\{ \begin{array}{l} n_1 > m_1 \\ ou \\ n_1 = m_1 \;\; \& \;\; n_2 > m_2 \end{array} \right.$$

Les opérations arithmétiques usuelles ainsi que leurs propriétés sont déjà définies dans Coq, dans le paquetage *Arith*. On le charge grâce à la commande `Require Arith`.

La définition de P_1 et P_2 dans Coq est la suivante :

```
Definition P1 : (b:wsort)(TS b) -> nat =
[b:wsort][U:(TS b)](<[b:wsort]nat>Match U with
  (* var *)    [n:nat] (power2 (S n))
  (* app *)    [a:terms][P1a:nat][b:terms][P1b:nat] (plus P1a P1b)
  (* lambda *) [a:terms][P1a:nat] (plus P1a (S(S O)))
  (* env *)    [a:terms][P1a:nat][s:sub_explicits][P1s:nat] (mult P1a P1s)
  (* id *)     (S (S O))
  (* shift *)  (S (S O))
  (* cons *)   [a:terms][P1a:nat][s:sub_explicits][P1s:nat] (plus P1a P1s)
  (* comp *)   [s:sub_explicits][P1s:nat][t:sub_explicits][P1t:nat]
               (mult P1s P1t)
  (* lift *)   [s:sub_explicits][P1s:nat]P1s
  (*  X  *)    [i:nat] (S(S O))
  (*  x  *)    [i:nat] (S(S O)) ).

Definition P2 : (b:wsort)(TS b) -> nat =
[b:wsort][U:(TS b)](<[b:wsort]nat>Match U with
  (* var *)    [n:nat] (S O)
  (* app *)    [a:terms][P2a:nat][b:terms][P2b:nat] (S (plus P2a P2b))
  (* lambda *) [a:terms][P2a:nat] (mult (S(S O)) P2a)
  (* env *)    [a:terms][P2a:nat][s:sub_explicits][P2s:nat] (mult P2a (S P2s))
  (*  id *)    (S O)
  (* shift *)  (S O)
  (* cons *)   [a:terms][P2a:nat][s:sub_explicits][P2s:nat] (S (plus P2a P2s))
  (* comp *)   [s:sub_explicits][P2s:nat][t:sub_explicits][P2t:nat]
```

```
              (mult P2s (S P2t))
  (* lift *)   [s:sub_explicits][P2s:nat] (mult (S(S(S(S 0)))) P2s)
  (*  X  *)    [i:nat] (S 0)
  (*  x  *)    [i:nat] (S 0)  ).
```

L'ordre $\succ$ est traduit dans Coq par `[b :wsort](lexfg (TS b) (P1 b) (P2 b))`. Sachant que `gt` représente l'ordre supérieur sur les entiers, on code `lexfg` par :

```
Definition lexfg [A:Set][f,g:A -> nat][a,b:A]
                 (gt (f a) (f b)) \/
                 (<nat>(f a) = (f b) /\ (gt (g a) (g b))).
```

Il nous faut alors prouver que $\succ$ est une relation nœthérienne :

```
Lemma lexfg_notherian : (A:Set)(f,g:A-A -> Prop)(noetherian A (lexfg A f g)).
```

Et que $\sigma_{\Uparrow}$ est contenu dans $\succ$:

```
Lemma lexfg_relSL : (b:wsort)(M,N:(TS b))(relSL b M N) ->
                                      (lexfg (TS b) (P1 b) (P2 b) M N).
```

Le lemme `lexfg_relSL` se prouve par récurrence sur `(relSL b M N)`. On est ainsi amené à prouver que :

1. $\alpha \succ \beta$ pour toute règle (α, β) de $\sigma_{\Uparrow}$.
2. $a_i \succ b_i$ entraîne $(C\ a_1 \cdots a_i \cdots a_n) \succ (C\ a_1 \cdots b_i \cdots a_n)$, où C est un opérateur (qui ne soit pas une constante).

On est alors en mesure de prouver que $\sigma_{\Uparrow}$ est nœthérien en appliquant le lemme `noether_inclus` :

```
Theorem relSL_noetherian : (b:wsort)(noetherian (TS b) (relSL b)).
```

7 $\sigma_{\Uparrow}$ est localement confluent

7.1 Présentation du problème

Posons le but à prouver :

```
Theorem conf_local_SL : (b:wsort)(local_confluence (TS b) (relSL b)).
```

Explicitons ce but en revenant à la définition de `local_confluence` :

```
   (z:(TS b))(relSL b x z) ->
       <(TS b)>Ex([u:(TS b)] (relSLstar b y u) /\ (relSLstar b z u)).
  ============================
    H : (relSL b x y)
    y : (TS b)
    x : (TS b)
    b : wsort
```

Puis éliminons `(relSL b x y)` (avec la tactique `Elim H`), nous obtenons ainsi un but pour chaque clause dans `relSL`. Considérons, par exemple, le but relatif à la règle *Id* :

```
<(TS wt)>Ex([u:(TS wt)]
                  (relSLstar wt a u) /\ (relSLstar wt M u))
============================
  H2 : (relSL wt (env a id) M)
  M : (TS wt)
  a : terms
```

L'idée maintenant est de trouver pour quelles "formes" de `M`, l'hypothèse `H2` est "vraie" i.e. quels sont les réduits possibles de $a[id]$ (dans la syntaxe de Coq : `(env a id)`) par la relation $\sigma_{\Uparrow}$. Ceci revient (en partie) à chercher toutes les règles qui produisent des *paires critiques* avec *Id*. Il faudra faire le même raisonnement pour chaque règle de $\sigma_{\Uparrow}$.

7.2 Vérification du théorème de Knuth-Bendix

Rappelons brièvement la définition de paire critique[11, 16]. On dit que deux règles (α, β) et (α', β') déterminent une paire critique (P, Q) s'il existe un sous-terme γ (qui ne soit pas une variable) de α unifiable avec α', i.e. tel que $\theta(\gamma) \equiv \theta(\alpha')$ où θ est une substitution. La paire critique correspond aux deux manières de réduire l'instance $\theta(\alpha)$: $P \equiv \theta(\beta)$ et $Q \equiv \theta(\alpha)\{\theta(\beta')/\theta(\gamma)\}$ (ce terme représente $\theta(\alpha)$ dans lequel on a réécrit $\theta(\gamma)$ en $\theta(\beta')$).

Coq ne permet pas dans sa version actuelle de déterminer automatiquement ces paires critiques, on doit donc le faire à la main (ou en utilisant le système KB de l'INRIA).

Formulons notre problème de confluence locale dans un cadre plus général : Soit R un système de réécriture à plusieurs sortes, on cherche pour chaque règle $r = (\alpha, \beta)$ à résoudre le but :

Br

$$\exists u.\ \beta \xrightarrow{R^*} u \ \wedge\ M \xrightarrow{R^*} u$$
$$==========$$
$$H : \alpha \xrightarrow{R} M$$
$$\vdots$$

Avec une syntaxe mathématique usuelle, cela donnerait : "chercher pour chaque règle (α, β) et pour toute substitution τ, un terme u tel que :

$$\tau(\beta) \xrightarrow{R^*} u \ \&\ M \xrightarrow{R^*} u$$

sachant que M est un terme quelconque vérifiant $\tau(\alpha) \xrightarrow{R} M$". L'introduction de τ correspond au fait que les variables de α sont toutes libres.

On va commencer par déduire de l'hypothèse H des conditions sur M et $\vec{a}$ $(= var(\alpha))$, i.e. chercher tous les réduits possibles des instances de α. On distingue trois cas :

Cas 1. $\tau(\alpha)$ est réduit en $\tau(\beta)$ en utilisant la règle (α, β), i.e. $M = \beta$.

Cas 2. $\tau(\alpha)$ est réduit avec une règle (α_i, β_i). (α, β) et (α_i, β_i) déterminent alors une paire critique (P_i, Q_i) avec une substitution θ_i.

$$M = Q_i$$

En regroupant tous les cas de paires critiques dans une disjonction et en rajoutant les contraintes concernant $\vec{a}$, on obtient :

$$(\bigvee_{i=1}^{m} (\exists \vec{b_i}.(\bigwedge_{j=1}^{n} a_j = \theta_i(a_j)) \wedge M = Q_i))$$

Les égalités $a_j = \theta_i(a_j)$ représentent la substitution θ_i et les $\vec{b_i}$ sont les nouvelles variables introduites par θ_i.

Cas 3. C'est la $k^{\text{ème}}$ occurrence de la variable a_j, notée a_j^k, qu'on réécrit :

$$M = \alpha_j^k$$

où $\alpha_j^k \equiv \alpha\{c_j/a_j^k\}$, i.e. α où la $k^{\text{ème}}$ occurrence de a_j a été substituée par une nouvelle variable c_j. En considérant le cas de toutes les occurrences de toutes les variables, on aboutit à :

$$(\bigvee_{j=1}^{n}(\exists c_j.a_j \xrightarrow{R} c_j \wedge (\bigvee_{k=1}^{p_j} M = \alpha_j^k)))$$

En résumé, on doit prouver pour chaque règle $r = (\alpha, \beta)$ un lemme *case_R_r* (R est le nom du système de réécriture), de la forme (écrit dans une pseudo-syntaxe Coq) :

$$\begin{gathered}\forall \vec{a}.\forall M.\\ \alpha \xrightarrow{R} M \Rightarrow\\ (M = \beta) \vee\\ (\bigvee_{i=1}^{m}(\exists \vec{b_i}.(\bigwedge_{j=1}^{n} a_j = \theta_i(a_j)) \wedge M = Q_i)) \vee\\ (\bigvee_{j=1}^{n}(\exists c_j.a_j \xrightarrow{R} c_j \wedge (\bigvee_{k=1}^{p_j} M = \alpha_j^k)))\end{gathered}$$

Nous allons maintenant vérifier le théorème de Knuth-Bendix[11, 16] : une relation de réécriture $\xrightarrow{R}$ est localement confluente si et seulement si pour toute paire critique (Q_1, Q_2), il existe u tel que $Q_1 \xrightarrow{R^*} u$ et $Q_2 \xrightarrow{R^*} u$. On dit que la paire converge.

En appliquant le lemme *case_R_r* au but **Br**, on se retrouve avec trois types de buts :

B1
$\exists u.\ \beta \xrightarrow{R^*} u \ \wedge\ \beta \xrightarrow{R^*} u$
==========
$\vdots$

B2
$\exists u.\ \theta_i(\beta) \xrightarrow{R^*} u \ \wedge\ Q_i \xrightarrow{R^*} u$
==========
$\vdots$

B3
$\exists u.\ \beta \xrightarrow{R^*} u \ \wedge\ \alpha_j^k \xrightarrow{R^*} u$
==========
$H' : a_j \xrightarrow{R} c_j$
$\vdots$

Il nous reste donc à résoudre ces buts. On constate qu'on peut résoudre de manière mécanique les buts **B1** et **B3** et non **B2** :

Cas de B1 Il suffit de prendre $u \equiv \beta$.

Cas de B2 C'est le cas correspondant aux paires critiques. En effet $\theta_i(\beta)$ n'est rien d'autre que P_i. Il faut étudier la convergence des paires critiques du système R.

Cas de B3 $u \equiv \beta\{c_j/a_j\}$ (β où toutes les occurrences de a_j sont substituées par c_j) convient car :

- $\beta \xrightarrow{R^*} u$, en réduisant toutes les occurrences (par passage au contexte) de a_j en c_j.
- $\alpha_j^k \xrightarrow{R^*} u$, en commençant par réduire toutes les occurrences de a_j en c_j puis en appliquant la règle (α, β).

Optimisation. La construction du but **B2** n'est pas immédiate. En effet β est transformé en $\theta_i(\beta)$ par utilisation des égalités $a_j = \theta_i(a_j)$. Coq peut faire ces substitutions à notre place en changeant l'énoncé de *case_R_r* de telle façon à tenir compte du contexte (représenté ici par le prédicat P) :

$$\forall P.\forall \vec{a}.$$
$$(P\ \vec{a}\ \beta) \Rightarrow$$
$$(\forall \vec{b_1}.(P\ \theta_1(a_1) \ldots \theta_1(a_n)\ Q_1)) \Rightarrow$$
$$\vdots$$
$$(\forall \vec{b_m}.(P\ \theta_m(a_1) \ldots \theta_m(a_n)\ Q_m)) \Rightarrow$$
$$(\forall c_1.a_1 \xrightarrow{R} c_1 \Rightarrow (P\ \vec{a}\ \alpha_1^1)) \Rightarrow$$
$$\vdots$$
$$(\forall c_1.a_1 \xrightarrow{R} c_1 \Rightarrow (P\ \vec{a}\ \alpha_1^{p_1})) \Rightarrow$$
$$\vdots$$
$$(\forall c_n.a_n \xrightarrow{R} c_n \Rightarrow (P\ \vec{a}\ \alpha_n^1)) \Rightarrow$$
$$\vdots$$
$$(\forall c_n.a_n \xrightarrow{R} c_n \Rightarrow (P\ \vec{a}\ \alpha_n^{p_n})) \Rightarrow$$
$$\forall M \in A.\alpha \xrightarrow{R} M \Rightarrow (P\ \vec{a}\ M)$$

Remarque. Un objectif futur est d'automatiser complètement la preuve de confluence locale, i.e. concevoir une tactique (appelons la TacKB) telle que :

```
Theorem confluence_local_R : (w:W)(local_confluence (ALG w) (relR w)).
  TacKB.
Goal proved!
```

Pour cela trois problèmes restent à résoudre :

1. Déterminer les paires critiques pour énoncer les lemmes *case_R_r*,
2. Générer la preuve des lemmes *case_R_r*,
3. Prouver la convergence des paires critiques (buts **B2**).

Les étapes 1 et 3 sont facilement résolubles. Il suffit d'utiliser un algorithme de Knuth-Bendix (tel que le système KB de l'INRIA) pour la détermination et la vérification de convergence des paires critiques. Il est alors facile de reporter ces informations dans Coq.

La preuve à la main des lemmes *case_R_r* dans le cas de $\sigma_{\Uparrow}$ est principalement basée sur l'*inversion* du prédicat `relSL`. Pour engendrer automatiquement la preuve des lemmes *case_R_r* dans le cas général, il faut disposer d'une tactique d'inversion[5]. Dans notre cas, cette tactique servirait à inverser $relR$, c'est-à-dire à engendrer une définition récursive $relR\_inv$ et à prouver la propriété suivante :

$$\forall w.\forall a,b.(relR\ w\ a\ b) \Leftrightarrow (relR\_inv\ w\ a\ b)$$

7.3 $\sigma_{\Uparrow}$ est localement confluent

On va utiliser la méthode qu'on vient de décrire pour prouver la confluence locale de $\sigma_{\Uparrow}$. Remarquons que $\sigma_{\Uparrow}$ est linéaire gauche (dans chaque règle (α,β), les variables ont une occurrence unique dans $\alpha : p_j = 1$) ; la taille des lemmes *case_SL_r* (SL pour $\sigma_{\Uparrow}$ et r indique un nom de règle) s'en trouve ainsi réduite.

Étudions, par exemple, le lemme associé à *Id* (*case_SL_Id*) :

$$a[id] \xrightarrow{\sigma_{\Uparrow}} a$$

On vérifie que les règles *App*, *Lambda* et *Clos* sont les seules à produire des paires critiques avec *Id* :

- $(a_1 b_1)[id] \xrightarrow{App} (a_1[id])(b_1[id])$, avec $\theta(a) = a_1 b_1$.
- $(\lambda\, a_1)[id] \xrightarrow{Lambda} \lambda(a_1[\Uparrow(id)])$, avec $\theta(a) = (\lambda\, a_1)$.
- $(a_1[s])[id] \xrightarrow{Clos} a_1[s \circ id]$; avec $\theta(a) = a_1[s]$.

On en déduit directement le lemme *case_SL_Id*, écrit dans une pseudo-syntaxe Coq :

$$\forall P \in terms \longrightarrow terms \longrightarrow Prop.\forall a \in terms.$$
$$(P\ a\ a) \Rightarrow$$
$$(\forall a_1, b_1 \in terms.(P\ (a_1 b_1)\ ((a_1[id])(b_1[id])))) \Rightarrow$$
$$(\forall a_1 \in terms.(P\ (\lambda\, a_1)\ \lambda(a_1[\Uparrow(id)]))) \Rightarrow$$
$$(\forall a_1 \in terms.\forall s \in sub\_explicits.(P\ (a_1[s])\ (a_1[s \circ id]))) \Rightarrow$$
$$(\forall a' \in terms.a \xrightarrow{\sigma_{\Uparrow}} a' \Rightarrow (P\ a\ (a'[id]))) \Rightarrow$$
$$\forall M \in terms.a[id] \xrightarrow{\sigma_{\Uparrow}} M \Rightarrow (P\ a\ M)$$

Et dans Coq :

```
(P:terms -> terms -> Prop)(a:terms)
(*Id    *) (P a a) ->
(*App   *) ((a1,b1:terms)(P (app a1 b1) (app (env a1 id) (env b1 id)))) ->
(*Lambda*) ((a1:terms)(P (lambda a1) (lambda (env a1 (lift id))))) ->
(*Clos  *) ((a1:terms)(s:sub_explicits)(P (env a1 s) (env a1 (comp s id)))) ->
           ((a':terms)(relSL wt a a') -> (P a (env a' id))) ->
           (M:terms)(relSL wt (env a id) M) -> (P a M).
```

On sait résoudre les sous-buts relatifs aux paires critiques car on vérifie que toutes les paires convergent :

- pour *App*, $a_1 b_1 \xrightarrow{\sigma_{\Uparrow}^*} a_1 b_1$ et $(a_1[id])(b_1[id]) \xrightarrow{Id} a_1(b_1[id]) \xrightarrow{Id} a_1 b_1$
- pour *Lambda*, $(\lambda\, a_1) \xrightarrow{\sigma_{\Uparrow}^*} (\lambda\, a_1)$ et $\lambda(a_1[\Uparrow(id)]) \xrightarrow{LiftId} \lambda(a_1[id]) \xrightarrow{Id} (\lambda\, a_1)$
- pour *Clos*, $a_1[s] \xrightarrow{\sigma_{\Uparrow}^*} a_1[s]$ et $a_1[s \circ id] \xrightarrow{IdR} a_1[s]$

L'application de ce procédé pour toutes les règles nous permet de conclure à la confluence locale de $\sigma_{\Uparrow}$ (car toutes les paires critiques convergent).

8 $\sigma_{\Uparrow}$ est confluent

Pour prouver la confluence de $\sigma_{\Uparrow}$, on va utiliser le lemme de Newman. Dans la section suivante, nous donnons la preuve mathématique dont s'inspire notre preuve dans Coq. Cette preuve est tirée de [11].

8.1 Lemme de Newman

Énoncé. Une relation nœthérienne est confluente si elle est localement confluente.

C'est un résultat intéressant, puisque la confluence locale est une propriété nettement plus facile à prouver que la confluence.

```
Theorem Newman : (A:Set)(R:A -> A -> Prop)
                 (noetherian A R) ->
                 (local_confluence A R) -> (confluence A R).
```

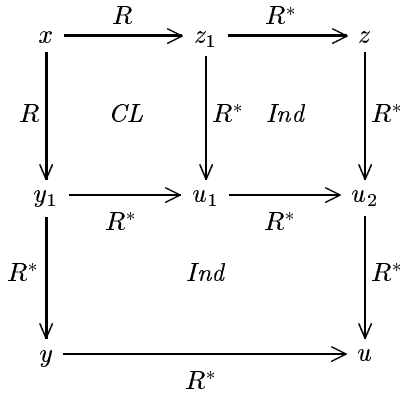
Preuve. Soit R une relation définie sur un ensemble A. Supposons que R est nœthérienne et localement confluente et montrons qu'elle est confluente.

La preuve dans Coq se fait par récurrence nœthérienne avec la propriété :

$$P(x) \equiv \forall y, z \in A.\, x\, R^*\, y\, \&\, x\, R^*\, z \Rightarrow \exists u \in A.\, (y\, R^*\, u\, \&\, z\, R^*\, u)$$

On distingue deux cas :

1. Si $x \equiv y$ (resp. $x \equiv z$), alors prendre $u \equiv z$ (resp. $u \equiv y$).
2. Sinon, on construit le diagramme suivant (*Ind* indique l'utilisation d'une hypothèse de récurrence) :



8.2 $\sigma_{\Uparrow}$ est confluent

On va maintenant montrer comment utiliser le lemme de Newman dans Coq pour prouver la confluence de $\sigma_{\Uparrow}$. On veut montrer :

```
Theorem confluence_SL : (b:wsort)(confluence (TS b) (relSL b)).
```

- `Goal.` La preuve commence.
- `Intro b.`

```
(confluence (TS b) (relSL b))
============================
  b : wsort
```

- `Apply Newman.` On utilise le lemme de Newman, le système nous demande alors de prouver que `relSL` est nœthérien et localement confluent.

```
2 subgoals
  (noetherian (TS b) (relSL b))
  ============================
    b : wsort
subgoal 2 is:
  (local_confluence (TS b) (relSL b))
```

- `Apply relSL_noetherian.` On sait que `relSL` est nœthérien.

```
1 subgoal
  (local_confluence (TS b) (relSL b))
  ============================
    b : wsort
```

- `Apply conf_local_SL.` On sait que `relSL` est localement confluent.

```
Goal proved!
```

9 Lemme de Yokouchi

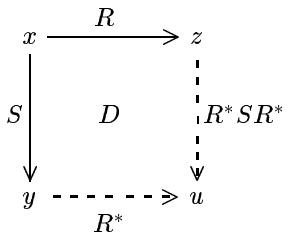
Le lemme de Yokouchi [20] est le point clé de la preuve de confluence du $\lambda\sigma_{\Uparrow}$-calcul. On va en donner un énoncé tiré de [6] et expliquer sa preuve dans Coq en utilisant des diagrammes.

Une autre preuve de la confluence du $\lambda\sigma_{\Uparrow}$-calcul est décrite dans [8], elle est basée sur un autre résultat : la méthode d'interprétation.

Énoncé. Soient R et S deux relations définies sur un même ensemble A. Si les conditions suivantes sont vérifiées :

1. R est nœthérienne
2. R est confluente
3. S est fortement confluente

4. le diagramme ci-dessous commute :

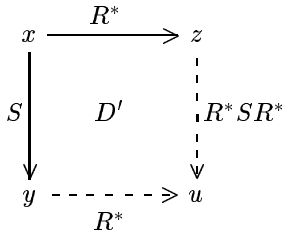


Alors R^*SR^* est fortement confluente.

L'énoncé équivalent dans Coq est :

```
Variable A : Set.

Variable R,S : A -> A -> Prop.

Hypothesis C : (confluence A R).

Hypothesis N : (noetherian A R).

Hypothesis SC : (strong_confluence A S).

Definition Rstar_S_Rstar (comp_rel A (star A R)(comp_rel A S (star A R))).

Hypothesis commut1 : (x,y,z:A)(R x z) -> (S x y) ->
                     <A>Ex([u:A] (star A R y u) /\ (Rstar_S_Rstar z u)).

Theorem Yokouchi : (strong_confluence A Rstar_S_Rstar).
```

Preuve. On commence par généraliser le diagramme D :



Soit dans Coq :

```
Lemma commut : (x,y,z:A)(star A R x z) -> (S x y) ->
                              <A>Ex([u:A] (star A R y u)/\(Rstar_S_Rstar z u)).
```

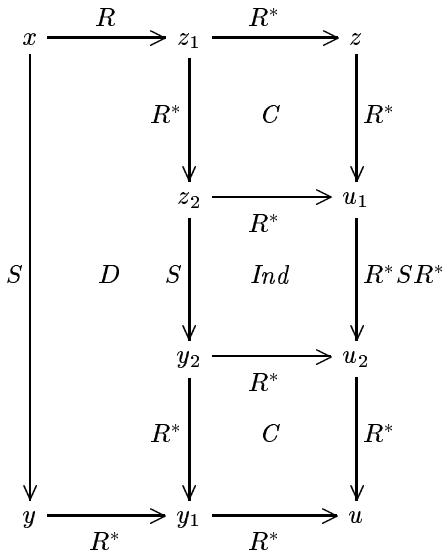
Nous procédons par récurrence nœthérienne en prenant :

$$(P\ x) \equiv \forall y, z \in A.\, x\, R^*\, z\ \&\ x\, S\, y \Rightarrow \exists u \in A.\, (y\, R^*\, u\ \&\ z\, R^*SR^*\, u)$$

On a alors deux cas :

1. Si $x \equiv z$ alors prendre $u \equiv y$.

2. Sinon, on construit le diagramme :

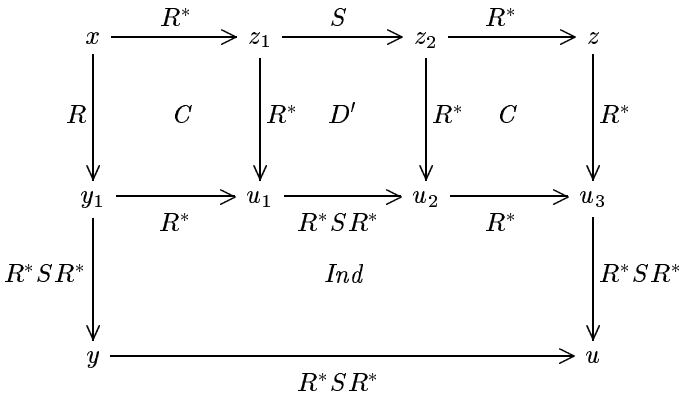


Pour prouver la confluence forte de R^*SR^*, on procède aussi par récurrence nœthérienne mais cette fois-ci avec :

$$P(x) \equiv \forall y, z \in A.\, x\, R^*SR^*\, y \,\&\, x\, R^*SR^*\, z \Rightarrow \exists u \in A.\, (y\, R^*SR^*\, u \,\&\, z\, R^*SR^*\, u)$$

On étudie la longueur de la dérivation du premier R^* dans $x\, R^*SR^*\, y$:

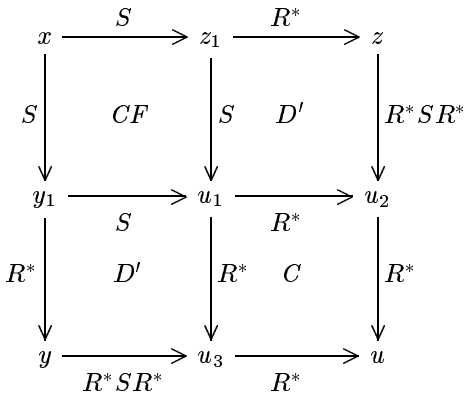
1. Si la longueur de la dérivation n'est pas nulle, i.e. $x\, RR^*SR^*\, y$, alors :



2. Si $x\, SR^*\, y$, alors on utilise le même raisonnement pour $x\, R^*SR^*\, z$:

   (a) $x\, RR^*SR^*\, z$, le symétrique du diagramme précédent convient.

   (b) $x\, SR^*\, z$:



10 Définition de $Beta\|$

On appelle $Beta\|$ la parallélisation de la règle *Beta*. Elle est définie par les règles d'inférence suivantes :

$$\frac{}{n \xrightarrow{Beta\|} n} \qquad \frac{}{id \xrightarrow{Beta\|} id}$$

$$\frac{a \xrightarrow{Beta\|} a'}{\lambda a \xrightarrow{Beta\|} \lambda a'} \qquad \frac{}{\uparrow \xrightarrow{Beta\|} \uparrow}$$

$$\frac{a \xrightarrow{Beta\|} a' \quad b \xrightarrow{Beta\|} b'}{ab \xrightarrow{Beta\|} a'b'} \qquad \frac{a \xrightarrow{Beta\|} a' \quad s \xrightarrow{Beta\|} s'}{a \cdot s \xrightarrow{Beta\|} a' \cdot s'}$$

$$\frac{a \xrightarrow{Beta\|} a' \quad s \xrightarrow{Beta\|} s'}{a[s] \xrightarrow{Beta\|} a'[s']} \qquad \frac{s \xrightarrow{Beta\|} s' \quad t \xrightarrow{Beta\|} t'}{s \circ t \xrightarrow{Beta\|} s' \circ t'}$$

$$\frac{a \xrightarrow{Beta\|} a' \quad b \xrightarrow{Beta\|} b'}{(\lambda a)b \xrightarrow{Beta\|} a'[b' \cdot id]} \qquad \frac{s \xrightarrow{Beta\|} s'}{\Uparrow(s) \xrightarrow{Beta\|} \Uparrow(s')}$$

$$\frac{}{X_i \xrightarrow{Beta\|} X_i} \qquad \frac{}{x_i \xrightarrow{Beta\|} x_i}$$

Elle se traduit directement dans Coq par :

```
Inductive Definition beta_par : (b:wsort)(TS b) -> (TS b) -> Prop
=  var_bpar    : (n:nat)(beta_par wt (var n)(var n))
 | id_bpar     : (beta_par ws id id)
 | shift_bpar  : (beta_par ws shift shift)
 | app_bpar    : (M,N,M',N':terms)(beta_par wt M M') -> (beta_par wt N N') ->
                 (beta_par wt (app M N) (app M' N'))
 | lambda_bpar : (M,M':terms)(beta_par wt M M') ->
                 (beta_par wt (lambda M) (lambda M'))
 | env_bpar    : (M,M':terms)(s,s':sub_explicits)(beta_par wt M M') ->
                 (beta_par ws s s') -> (beta_par wt (env M s) (env M' s'))
 | beta_bpar   : (M,N,M',N':terms)(beta_par wt M M') -> (beta_par wt N N') ->
                 (beta_par wt (app (lambda M) N) (env M' (cons N' id)))
 | cons_bpar   : (M,M':terms)(s,s':sub_explicits)(beta_par wt M M') ->
```

```
                (beta_par ws s s') -> (beta_par ws (cons M s) (cons M' s'))
 | lift_bpar    : (s,s':sub_explicits)(beta_par ws s s') ->
                (beta_par ws (lift s) (lift s'))
 | comp_bpar    : (s,s',t,t':sub_explicits)(beta_par ws s s') -> (beta_par ws t t') ->
                (beta_par ws (comp s t) (comp s' t'))
 | metaX_bpar   : (n:nat)(beta_par wt (meta_X n)(meta_X n))
 | metax_bpar   : (n:nat)(beta_par ws (meta_x n)(meta_x n)).
```

Remarque. Dans la définition de $Beta\|$ de [10], les règles :

$$\frac{}{n \overset{Beta\|}{\longrightarrow} n} \qquad \frac{}{id \overset{Beta\|}{\longrightarrow} id} \qquad \frac{}{\uparrow \overset{Beta\|}{\longrightarrow} \uparrow} \qquad \frac{}{X_i \overset{Beta\|}{\longrightarrow} X_i} \qquad \frac{}{x_i \overset{Beta\|}{\longrightarrow} x_i}$$

sont remplacées au profit d'une seule règle : $\dfrac{}{M \overset{Beta\|}{\longrightarrow} M}$ (pour tout $\lambda\sigma_{\Uparrow}$-terme M). Ces deux définitions sont bien entendu équivalentes. Toutefois du point de vue du codage dans Coq, la première est plus intéressante pour deux raisons :

- la deuxième définition introduit des paires critiques inutiles.
- la première définition conduit à des preuves plus élégantes car elle nous donne directement les cas de base d'un raisonnement par récurrence, alors que dans le cas de la deuxième définition, ceux-ci ne sont obtenus qu'au prix d'une nouvelle récurrence sur la structure de M.

11 Application du Lemme de Yokouchi à $\sigma_{\Uparrow}$ et $Beta\|$

Dans cette section, on veut prouver que $\sigma_{\Uparrow}^* Beta\| \sigma_{\Uparrow}^*$ est fortement confluent. Pour cela, on va utiliser le lemme de Yokouchi sur les relations $\sigma_{\Uparrow}$ et $Beta\|$. Les deux premières conditions du lemme, en l'occurrence la nœthérianité et la confluence de $\sigma_{\Uparrow}$, ont déjà été vérifiées.

Il s'agit donc maintenant de prouver la confluence forte de $Beta\|$ et la commutation du diagramme D. Ces deux preuves se font selon le même principe que la preuve de confluence locale de $\sigma_{\Uparrow}$.

Pour prouver la confluence forte de $Beta\|$:

```
Theorem sconf_betapar : (b:wsort)(strong_confluence (TS b) (beta_par b)).
```

on étudie ses paires critiques, i.e. on cherche à déterminer quelles règles de $Beta\|$ sont applicables pour chaque membre gauche des règles de $Beta\|$. On ne trouve qu'une seule paire critique :

$$\frac{\dfrac{a \overset{Beta\|}{\longrightarrow} a'}{\lambda a \overset{Beta\|}{\longrightarrow} \lambda a'} \quad b \overset{Beta\|}{\longrightarrow} b'}{(\lambda a) b \overset{Beta\|}{\longrightarrow} (\lambda a') b'} \qquad\qquad \frac{a \overset{Beta\|}{\longrightarrow} a' \quad b \overset{Beta\|}{\longrightarrow} b'}{(\lambda a) b \overset{Beta\|}{\longrightarrow} a'[b' \cdot id]}$$

La preuve se fait par récurrence sur `(beta_par b x y)` avec le but suivant :

```
  (z:(TS b))(beta_par b x z) ->
            (<(TS b)>Ex([u:(TS b)] (beta_par b y u) /\ (beta_par b z u)))
  ============================
    H : (beta_par b x y)
    y : (TS b)
    x : (TS b)
    b : wsort
```

Seul le sous-but relatif à la paire critique pose problème :

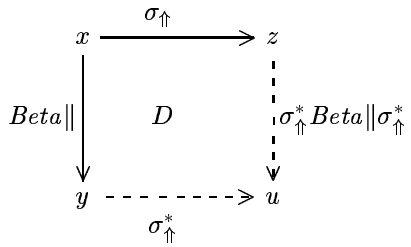
```
<(TS wt)>Ex([u:(TS wt)]
          (beta_par wt (app (lambda a') b') u)
       /\ (beta_par wt (env a'' (cons b'' id)) u))
============================
  H16 : (beta_par wt b'' b0)
  H15 : (beta_par wt b' b0)
  b0 : (TS wt)
  H13 : (beta_par wt a'' a0)
  H12 : (beta_par wt a' a0)
  a0 : (TS wt)
  H9 : (beta_par wt a a')
  a' : terms
  H7 : (beta_par wt b b'')
  H6 : (beta_par wt a a'')
  b'' : terms
  a'' : terms
  a : terms
  H2 : (beta_par wt b b')
  b' : terms
  b : term
```

on cherche u tel que : $(\lambda a')b' \stackrel{Beta\|}{\longrightarrow} u$ et $a''[b''\cdot id] \stackrel{Beta\|}{\longrightarrow} u$, sachant que par hypothèse de récurrence : $a' \stackrel{Beta\|}{\longrightarrow} a_0$, $a'' \stackrel{Beta\|}{\longrightarrow} a_0$, $b' \stackrel{Beta\|}{\longrightarrow} b_0$ et $b'' \stackrel{Beta\|}{\longrightarrow} b_0$. On vérifie que $a_0[b_0 \cdot id]$ convient :

$$\frac{a' \stackrel{Beta\|}{\longrightarrow} a_0 \qquad b' \stackrel{Beta\|}{\longrightarrow} b_0}{(\lambda a')b' \stackrel{Beta\|}{\longrightarrow} a_0[b_0 \cdot id]}$$

$$\frac{a'' \stackrel{Beta\|}{\longrightarrow} a_0 \qquad \dfrac{b'' \stackrel{Beta\|}{\longrightarrow} b_0 \qquad \overline{id \stackrel{Beta\|}{\longrightarrow} id}}{b''\cdot id \stackrel{Beta\|}{\longrightarrow} b_0 \cdot id}}{a''[b''\cdot id] \stackrel{Beta\|}{\longrightarrow} a_0[b_0 \cdot id]}$$

Attaquons-nous maintenant à la vérification de la quatrième condition de Yokouchi :



En appelant `slstar_bp_slstar` la traduction de $\sigma^*_{\Uparrow} Beta\| \sigma^*_{\Uparrow}$ dans Coq :

```
Definition slstar_bp_slstar [b:wsort]
(comp_rel (TS b) (relSLstar b) (comp_rel (TS b) (beta_par b) (relSLstar b))).
```

l'énoncé à démontrer s'écrit :

```
Theorem commutation : (b:wsort)(x,y,z:(TS b))
          (relSL b x z) ->
          (beta_par b x y) ->
          <(TS b)>Ex([u:(TS b)] (relSLstar b y u) /\ (slstar_bp_slstar b z u)).
```

La preuve se fait par récurrence sur la réduction $x \stackrel{\sigma_{\Uparrow}}{\longrightarrow} z$. Pour chaque membre gauche des règles de $\sigma_{\Uparrow}$, on cherche les règles de $Beta\|$ qui lui sont applicables. Étudions le cas de la règle App $((ab)[s] \stackrel{\sigma_{\Uparrow}}{\longrightarrow} (a[s])(b[s]))$. Deux règles de $Beta\|$ sont applicables à $(ab)[s]$:

1. $(ab)[s] \stackrel{Beta\|}{\longrightarrow} (a'b')[s']$ avec $a \stackrel{Beta\|}{\longrightarrow} a'$, $b \stackrel{Beta\|}{\longrightarrow} b'$ et $s \stackrel{Beta\|}{\longrightarrow} s'$. On prend $u \equiv (a'[s'])(b'[s'])$, en effet $(a'b')[s'] \stackrel{App}{\longrightarrow} u$ et $(a[s])(b[s]) \stackrel{Beta\|}{\longrightarrow} u$.

2. $((\lambda a_1)b)[s] \stackrel{Beta\|}{\longrightarrow} (a'_1[b' \cdot id])[s']$, avec $a_1 \stackrel{Beta\|}{\longrightarrow} a'_1$, $b \stackrel{Beta\|}{\longrightarrow} b'$ et $s \stackrel{Beta\|}{\longrightarrow} s'$. On prend $u \equiv a'_1[b'[s'] \cdot s']$, en effet :

   - $((\lambda a_1)[s])(b[s]) \stackrel{Lambda}{\longrightarrow} (\lambda(a_1[\Uparrow s]))(b[s]) \stackrel{Beta\|}{\longrightarrow} (a'_1[\Uparrow s'])[b'[s'] \cdot id] \stackrel{Clos}{\longrightarrow} a'_1[\Uparrow(s') \circ (b'[s'] \cdot id)] \stackrel{LiftEnv}{\longrightarrow} a'_1[b'[s'] \cdot (s' \circ id)] \stackrel{Idr}{\longrightarrow} u$ et,

   - $(a'_1[b' \cdot id])[s'] \stackrel{Clos}{\longrightarrow} a'_1[(b' \cdot id) \circ s'] \stackrel{MapEnv}{\longrightarrow} a'_1[b'[s'] \cdot (id \circ s')] \stackrel{Idl}{\longrightarrow} u$.

12 Le $\lambda\sigma_{\Uparrow}$-calcul est confluent

Le lemme de Yokouchi nous a permis de prouver que $\sigma^*_{\Uparrow} Beta\| \sigma^*_{\Uparrow}$ est fortement confluent. Ce résultat n'est en fait qu'un jalon dans la preuve de confluence du $\lambda\sigma_{\Uparrow}$-calcul. Celle-ci est obtenue en utilisant le critère de confluence suivant :

Énoncé. Soient R et S deux relations sur un même ensemble A. Si S est confluent et $R \subseteq S \subseteq R^*$, Alors R est aussi confluent.

Preuve. La confluence (voir 3.2) d'une relation quelconque R ne dépend que du comportement de sa clôture réflexive transitive R^*. Or de l'hypothèse $R \subseteq S \subseteq R^*$, on déduit que $R^* = S^*$ (i.e. $R^* \subseteq S^*$ et $S^* \subseteq R^*$). Donc R est confluent si et seulement si S est confluent.

La traduction de ce résultat dans Coq est la suivante :

```
Lemma inclus_conf : (A:Set)(R,S:A -> A -> Prop)
                    (inclus A R S) ->
                    (inclus A S (star A R)) ->
                    (confluence A S) ->
                    (confluence A R).
```

La confluence du $\lambda\sigma_{\Uparrow}$-calcul est obtenue en trouvant une relation confluente S telle que :

$$\lambda\sigma_{\Uparrow} \subseteq S \subseteq \lambda\sigma^*_{\Uparrow}$$

Le candidat pour jouer le rôle de S n'est autre que $\sigma^*_{\Uparrow} Beta\| \sigma^*_{\Uparrow}$. En effet $\sigma^*_{\Uparrow} Beta\| \sigma^*_{\Uparrow}$ est confluent :

```
Goal (b:wsort)(confluence (TS b) (slstar_bp_slstar b)).
```

car la confluence forte implique la confluence :

```
Lemma strong_conf_conf : (A:Set)(R:A -> A -> Prop)
                         (strong_confluence A R) -> (confluence A R).
```

Il ne nous reste plus qu'à vérifier que $\lambda\sigma_{\Uparrow} \subseteq \sigma^*_{\Uparrow} Beta\| \sigma^*_{\Uparrow}$:

```
Lemma relLSL_inclus_slbpsl :
          (b:wsort)(inclus (TS b) (relLSL b) (slstar_bp_slstar b)).
```

et que $\sigma^*_{\Uparrow} Beta \| \sigma^*_{\Uparrow} \subseteq \lambda\sigma^*_{\Uparrow}$:

```
Lemma slbpsl_inclus_relLSLstar :
          (b:wsort)(inclus (TS b) (slstar_bp_slstar b) (relLSLstar b)).
```

On est alors capable de prouver la confluence du $\lambda\sigma_{\Uparrow}$-calcul :

```
Theorem confluence_LSL: (b:wsort)(confluence (TS b) (relLSL b)).
```

une fois vérifiées toutes les conditions du lemme `inclus_conf`.

13 Conclusions

La preuve que nous venons de décrire a nécessité six mois de travail dont une partie consacrée à l'apprentissage de Coq. Le script de la preuve (plus de 250000 octets) est vérifié en 20' sur une Sparc10.

Ce travail nous a permis de développer une bibliothèque de résultats "classiques" portant sur les relations (notamment **Newman**, **inclus_conf**, **noether_inclus**, **Yokouchi** et **strong_conf_conf**). Nous avons tenu à faire ce développement dans le cadre le plus général possible, en utilisant systématiquement le polymorphisme sur les ensembles et les relations. Cette bibliothèque pourra ainsi être utilisée dans les développements futurs portant sur les relations ou la réécriture.

On remarque le mélange harmonieux entre des raisonnements abstraits et des raisonnements par récurrence, notamment dans les preuves du lemme de Newman et du lemme de Yokouchi. Une autre remarque est la quasi absence de codage : la plupart des notions mathématiques utilisés sont codées de manière naturelle dans Coq.

Une perspective intéressante à ce travail est de prouver (dans Coq) que le $\lambda\sigma_{\Uparrow}$-calcul simule le λ-calcul, i.e. une β-réduction correspond à une étape de *Beta* suivie d'une normalisation par $\sigma_{\Uparrow}$. Cette preuve utiliserait le codage du λ-calcul décrit dans [15].

Remerciements

Je tiens à remercier Thérèse Hardin pour sa disponibilité, les conseils qu'elle n'a cessé de me prodiguer et l'attention avec laquelle elle a suivi ce travail. Je voudrais aussi remercier Gilles Dowek, Benjamin Werner et Gérard Huet pour toutes nos fructueuses discussions. Un grand merci à Valérie Ménissier-Morain pour ses compétences TEXniques.

Références

[1] M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy, *Explicit Substitutions.* ACM Conference on Principle of Programming Languages, San Francisco, 1990.

[2] H.P.Barendregt, *Lambda calculi with types.* In Handbook of Logic in Computer Science, Vol II, Ed. S.Abramsky, D.Gabby and T.Marbaum, Oxford University Press, 1993.

[3] S. Boutin, *Vérification en Coq d'un compilateur Mini-ML en CAM.* Rapport de D.E.A. Université Paris 7, 1992.

[4] T. Coquand, *Une théorie des constructions.* Thèse de troisième cycle. Université Paris 7, 1985.

[5] C. Cornes *Inversion des prédicats inductifs.* Rapport de stage de DEA d'Informatique Fondamentale, Université Paris VII, septembre 93.

[6] P.-L. Curien, T. Hardin, J.-J. Lévy, *Confluence properties of Weak and Strong Calculi of Explicit Substitutions.* Rapport de recherche CEDRIC (CNAM) 92–1, 56 pages. To appear in J.A.C.M.

[7] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, B. Werner, *The Coq Proof Assistant User's Guide, version 5.8.* Rapport technique INRIA 154, Mai 1993.

[8] T. Hardin, *Eta-conversion for the languages of explicit substitutions.* Applicable Algebra in Engineering, Communication and Computing, 1993.

[9] T. Hardin, *From Categorical Combinators to $\lambda\sigma$-calculi, a quest for confluence.* Term graph Rewriting, Ed Sleep M.R, Plasmeijer M.J and al. Wiley 1993.

[10] T. Hardin, J.-J. Lévy, *A Confluent Calculus of Substitutions.* France-Japan Artificial Intelligence and Computer Science Symposium, Izu, 1989. Rapport de recherche CEDRIC 90/11.

[11] G. Huet, *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems.* J.A.C.M., vol 27(4), pp 797–821, October 1980.

[12] G. Huet, *Inductive Principles Formalized in the Calculus of Constructions.* in Programming of Future Generation Computers, Eds. K. Fuchi and M. Nivat, North-Holland 1988.

[13] G. Huet, *A Uniform Approach to Type Theory.* Rapport de recherche INRIA 795, February 88.

[14] G. Huet, *The Gilbreath Trick: A case study.* Procedings of 12th FST/TCS Conference, New Delhi, Dec. 1992. Ed. R. Shyamasundar, Springer Verlag LNCS 652, pp. 229–240.

[15] G. Huet, *Residual Theory in λ-calculus: A complete Gallina Development.* Rapport de recherche INRIA 2002, 1993. To appear in J. of Functional Programming.

[16] D. Knuth, P. Bendix, *Simple word problems in universal algebras.* In Computationel Problems in Abstract Algebra, J. Leech, Ed., Pergamon Press. Elmsford, N.Y., 1970, pp. 263–297.

[17] C. Paulin-Mohring, *Inductive Definitions in the System Coq - Rules and Properties.* Proceedings TLCA 93, Ultrecht, March 93. LNCS 664, pp. 328–345.

[18] C. Paulin-Mohring, B.Werner, *Synthesis of ML programs in the system Coq.* Proceedings of the first workshop on Logical Frameworks, 1992.

[19] J. Rouyer, *Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs.* Rapport de recherche INRIA 1795.

[20] H. Yokouchi, *Church-Rosser theorem for Categorical Combinators.* Theoretical Computer Sc. 65, 1989, pp. 271–290.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399