INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Coq en Coq

Bruno Barras

N° 3026

Octobre 1996

THÈME 2

Rapport de recherche

ISSN 0249-6399

Coq en Coq

Bruno Barras*

Thème 2 — Génie logiciel
et calcul symbolique

Projet Coq

Rapport de recherche n° 3026 — Octobre 1996

Résumé : L'étape essentielle de la certification d'un système de preuves tel que Coq serait la vérification de son noyau: un vérificateur de types d'un petit système de vérification de preuves basé sur le *Calcul des Constructions Inductives* (CCI). Dans ce papier, nous formalisons dans Coq la définition et la métathéorie du *Calcul des Constructions* (CC), qui est un fragment de CCI. En particulier, nous démontrons la normalisation forte et la décidabilité du typage pour ce système. De ce dernier résultat, un programme en *Caml Light* testant la validité d'un jugement de typage dans le *Calcul des Constructions* a été extrait. Ce programme intégré dans un système comprenant un analyseur syntaxique et un *pretty-printer* est un système de vérification de preuve autonome et performant pour le *Calcul des Constructions* baptisé *Coc*. La preuve du lemme de Newman produite avec Coq a pu être revérifiée dans *Coc* avec des performances raisonables.

Mots-clé : Théorie des Types, métathéorie, Calcul des Constructions, extraction de programmes.

(Abstract: pto)

*Bruno.Barras@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Coq in Coq

Abstract: The essential step of the formal verification of a proof-checker such as Coq is the verification of its kernel: a type-checker for the *Calculus of Inductive Constructions* (CIC) which is its underlying formalism. The present work is a first small-scale attempt on a significative fragment of CIC: the Calculus of Constructions (CC). We formalize the definition and the metatheory of (CC) in Coq. In particular, we prove strong normalization and decidability of type inference. From the latter proof, we extract a certified *Caml Light* program, which performs type inference (or type-checking) for an arbitrary typing judgement in CC. Integrating this program in a larger system, including a parser and pretty-printer, we obtain a stand-alone proof-checker, called *CoC*, for the *Calculus of Constructions*. As an example, the formal proof of Newman's lemma, build with Coq, can be re-verified by *CoC* with reasonable performance.

Key-words: Type Theory, metatheory, Calculus of Constructions, program extraction.

Table des matières

Introduction

Le système Coq est un système d'aide à la preuve qui peut être utilisé pour développer des applications critiques. L'intérêt de ce système est la coexistence de deux aspects intimement liés par l'isomorphisme de Curry-Howard:

- Un aspect mathématique: ce système permet d'élaborer des preuves compliquées. On peut espérer formaliser une grande partie des mathématiques.
- Un aspect programmation: les preuves en logique intuitionniste permettent de déduire des algorithmes correspondant aux propositions prouvées.

Cette dualité se traduit par la possibilité de transformer des preuves en programmes. C'est ce qu'on appelle l'extraction (voir [15]).

Pourquoi vérifier Coq ?

Le système est assez gros (plusieurs milliers de lignes de Caml Light). La probabilité pour que des erreurs se dissimulent n'est donc pas négligeable.

Comme le système est censé certifier la validité de théories, une erreur de programmation pourrait être très dommageable: on peut imaginer que Coq pourrait accepter une preuve dans laquelle se serait glissée une erreur de raisonnement. Il est donc essentiel de chercher à augmenter la confiance que l'on peut avoir en ce système.

Il y a deux conditions à vérifier pour pouvoir considérer le système comme validé:

- Le système logique qu'implémente Coq est cohérent, i.e. la proposition absurde n'a pas de preuve.
- L'implémentation est correcte vis-à-vis du système logique: le programme n'accepte que des démonstrations valides. On peut aussi vouloir montrer qu'il ne rejette aucune preuve correcte (complétude).

Nous allons maintenant étudier comment vérifier ces deux conditions. Une des premières idées qui vient à l'esprit est de faire les preuves à l'aide de Coq.

Pourquoi en Coq ?

C'est la possibilité de pouvoir extraire un programme exécutable qui est séduisante: si nous arrivons à prouver les résultats métathéoriques du système logique de Coq, y compris la correction de l'extraction, nous serons en mesure d'extraire un programme ayant la même puissance que Coq.

Nous pourrions nous arrêter là en nous disant que notre programme est vérifié. Mais il est intéressant de faire remarquer la possibilité de faire ce qu'on pourrait appeler le *bootstrap* de Coq: il suffit de re-vérifier la preuve avec notre programme extrait, ce qui nous permet d'extraire à nouveau un vérificateur de preuves, qui devrait être identique à celui qui l'a engendré (résultat de la première extraction).

Mise à part la confiance supplémentaire que le système gagnerait, le développement d'une théorie mathématique récente et assez complexe comme le Calcul des Constructions serait la démonstration que les outils mis à notre disposition par le système sont suffisamment puissants pour s'adapter à notre façon de raisonner sans trop freiner notre intuition.

Quelques objections

La vision idyllique qui vient d'être décrite ne doit pas nous faire oublier les problèmes qui nous attendent:

Le programme est trop long pour être entièrement vérifié

Certifier Coq en entier est beaucoup trop ambitieux à cause de la taille du système. Heureusement, le système est conçu de sorte que la sûreté ne porte que sur une faible portion de code, le "noyau". La seule opération que sait faire le noyau est la vérification de type, ce qui le rend nettement plus petit.

Il y a une couche au-dessus de ce noyau qui permet de lire des commandes du langage Gallina et les traduit en opérations élémentaires. Pour qu'un système soit effectivement utilisable, il faut que cette couche fasse le lien entre les opérations à faire et la manière de raisonner de l'utilisateur. On peut donner quelques exemples de fonctionnalités du système sur lesquelles ne repose pas la cohérence du système:

- La possibilité de démontrer les résultats sous la forme de sous-buts à prouver, que des commandes (les tactiques) permettent de simplifier ou résoudre. Ceci est essentiel: il n'est pas raisonnable de demander la preuve d'un bloc.
- Pouvoir modifier l'apparence des objets manipulés pour une écriture plus naturelle des développements grâce à un interpréteur de grammaires extensible.
- Un mécanisme de sections qui permet de paramétrer un développement, cela procède un peu à la manière du déchargement d'hypothèse en déduction naturelle.

Avec cette architecture à deux niveaux (décrite par Boyer et Dowek dans [4]), on peut se contenter de vérifier le noyau, puisque tout raisonnement erroné sera rejeté lors de la deuxième passe. L'inconvénient de ne pas tout vérifier est que le code intermédiaire doit être compréhensible car c'est le seul qui a valeur de preuve, et il faut s'assurer que la couche supérieure n'a pas modifié le sens du développement. Ce risque n'est pas négligeable si l'on abuse des facilités de syntaxe: il est possible de cacher le sens effectif de ce qu'on fait. Par exemple, le statut des axiomes à l'intérieur d'une section est assez sensible: la différence entre une hypothèse locale à la section et un véritable axiome qui survivra à la fermeture de la section ne se fait que par le choix d'un mot-clé plutôt qu'un autre, alors que ces deux notions sont très éloignées.

En général, les développements sont stockés sous forme de texte. Il faut donc rajouter au système des procédures d'analyse grammaticale et d'affichage des objets. Ces fonctions sont particulièrement difficiles à vérifier car leur spécification ne s'exprime pas facilement. Ce qu'on demande à un analyseur grammatical est qu'il traduise une chaîne de caractère en l'objet qu'elle dénote; la spécification d'une telle fonction est en général assez fastidieuse et en tout cas pas très explicite; or, la qualité essentielle d'un bon analyseur grammatical est précisément d'avoir une correspondance intuitive simple entre la chaîne de caractères et l'arbre de syntaxe abstraite. Le mieux semble être de vérifier une fonction qui prend en argument un arbre de syntaxe des commandes et renvoie un arbre de syntaxe abstraite correspondant au résultat de la commande. Les fonctions d'interface avec l'utilisateur seraient réduites au minimum et pourraient être admises sans grand risque.

La première condition n'est pas démontrable en Coq

Cette affirmation découle du second théorème d'incomplétude de Gödel. Informellement, ce théorème dit qu'il n'est pas possible de prouver la cohérence d'un système logique dans un autre de "complexité logique" équivalente, sauf bien sûr si ce dernier est incohérent. Ceci est bien entendu fatal à l'idée de *bootstrap* intégral du système[1]: si on arrivait à prouver la cohérence à l'intérieur du système, tout ce qu'on aurait prouvé serait son incohérence.

Le *bootstrap* n'étant pas possible, deux possibilités s'offrent à nous:

- admettre un axiome traduisant la cohérence, en trouvant une formulation non calculatoire de cette hypothèse afin de ne pas perdre le caractère constructif de la preuve.
- ou bien prouver la cohérence du système grâce à un système plus fort. Cela semble satisfaisant à première vue, mais cela ne résoud pas tout: un système plus fort est un système qui permet de démontrer plus de choses, donc il a encore plus de chances d'être incohérent que le système concerné. La solution pragmatique la plus couramment utilisée consiste à utiliser un système suffisamment puissant, qui soit largement répandu et en lequel on ait confiance, comme par exemple la Théorie des Ensembles de Zermelo ou une de ses extensions (Zermelo-Fraenkel, ZFC, etc...). Il faut bien attirer l'attention sur le fait qu'on ne peut guère faire mieux, la certitude absolue n'étant pas de ce monde.

Dans les λ-calculs, la partie logiquement complexe de la preuve de cohérence est la preuve de normalisation forte du calcul, ce qui correspond au résultat d'élimination des coupures dans les systèmes logiques. Intuitivement, on peut se rendre compte que poser cette propriété comme axiome ne nous empêchera pas d'extraire un programme: la normalisation forte ne sert qu'à justifier que l'algorithme de normalisation termine, mais cela n'empêche pas de prouver que si la fonction termine, alors le résultat sera correct.

La deuxième solution sort un peu du cadre du sujet: c'est plus la correction du programme vis-à-vis de la logique que la cohérence de la logique elle-même que nous voulons prouver.

[1] Même sans connaître le second théorème d'incomplétude de Gödel, on se rend bien compte qu'une preuve d'auto-cohérence ne serait pas très convaincante: pour parodier une phrase désormais célèbre, doit-on croire quelqu'un qui dirait "Je ne mens pas" ?

Nous nous restreignons ici au Calcul des Constructions (c'est le système des premières versions de Coq), ce qui fait que le second théorème d'incomplétude ne s'applique pas. Nous en profiterons pour employer la deuxième possibilité. Mais pour le système dans toute sa puissance (i.e. avec les types inductifs et les univers), nous admettrions la normalisation forte du calcul.

La preuve faite avec un outil non vérifié pourrait être erronée

Celle qui saute aux yeux lorsque l'on *bootstrape* un système logique, c'est que si le programme que l'on utilise est susceptible de contenir une erreur, tout ce qu'il nous affirme peut être mis en question, et la preuve perd toute valeur.

Nous avons déjà vu que la première condition devrait être admise pour le *bootstrap* (et ce serait la seule hypothèse à admettre), ce qui revient à dire que nous sommes bien d'accord que ce formalisme déduit des propositions “vraies”. Tout le problème est de se convaincre que la conclusion de ce développement est bien un théorème dans ce formalisme.

Il y a plusieurs moyens de s'en convaincre, suivant le degré de confiance de chacun:

- On peut avoir confiance en la capacité d'implémenter correctement quelques règles d'inférence par des programmeurs chevronnés. C'est ce que tout le monde a fait jusqu'ici.
- On peut lire ce rapport et nous croire sur parole quand nous disons ne pas nous en être trop remis au système [2]. Cela revient à considérer Coq comme un programme qui met en forme les directives de preuves qu'on lui donne, et qui décèle la plupart des erreurs. La preuve aura toujours plus de valeur que si elle n'avait été vérifiée que par des humains, ce qui est le cas de beaucoup d'articles ou de thèses qui font office de référence. Mais il n'est pas vain de chercher d'autres moyens de vérification.
- On peut demander l'avis à des mathématiciens compétents leur avis sur la preuve. Reste à régler le problème de la représentation de la preuve. La forme naturelle des preuves en Coq est le λ-terme; celui-ci est très long et difficile à comprendre. On peut aussi chercher à le traduire dans un langage naturel. Mais pour cela, il faudrait vérifier le traducteur de preuves. Une autre idée (évoquée par Pollack dans [17]) serait que toute personne désirant se convaincre de l'exactitude de la preuve pourrait écrire elle-même un vérificateur de preuves pour la logique correspondante.

[2] On voit ici un avantage des systèmes d'aide à la preuve interactifs: on n'utilise que des arguments intuitivement valides. C'est plus sûr que de laisser la machine faire tout le travail sans aucun moyen de contrôle.

De plus, même si le système se révèle incohérent, la preuve n'est pas perdue car elle correspond à des arguments de bon sens que l'on cherchera à pouvoir reproduire. Historiquement, à chaque fois qu'un système formel s'est révélé incohérent, on ne l'a pas jeté en bloc; on a restreint les mécanismes de déduction afin (1) d'éviter les paradoxes connus et (2) de pouvoir encore démontrer ce qui parait sensé.

Plan

Nous supposerons que le lecteur est assez familiarisé avec Coq pour comprendre l'énoncé des lemmes sous la forme Gallina. Une lecture assez rapide du manuel de référence [8] devrait permettre de comprendre le développement. Les résultats seront systématiquement énoncés en langage informel mathématique, sauf dans les deux derniers chapitres.

Le chapitre 1 présente le Calcul des Constructions. Nous avons choisi une formulation qui diffère de la présentation initiale (notations de de Bruijn [5]), afin de simplifier les résultats métathéoriques.

Nous montrerons ensuite les principaux résultats métathéoriques de ce λ-calcul. Ceux-ci nous servirons à démontrer les résultats calculatoires (en particulier le résultat majeur: la décidabilité du typage) dans le chapitre 3.

Le chapitre suivant sera réservé à la description de la procédure d'extraction de programmes, aux améliorations envisageables et aux développements que l'on peut faire à partir des fonctions extraites.

Les deux derniers chapitres présenteront les résultats permettant de prouver la normalisation forte du Calcul des Constructions en Coq. Cette preuve est assez complexe, ne suit pas exactement la démonstration informelle, et pourrait probablement être améliorée (tant au niveau de la présentation que dans l'enchaînement des lemmes), mais elle présente l'avantage d'être (à notre connaissance) la première formalisation complète de cette preuve.

Nous conclurons ce rapport avec des statistiques concernant le développement: temps de mise au point des preuves, tailles respectives des preuves et des programmes extraits, etc... Nous en profiterons pour discuter de la faisabilité de preuves importantes par des moyens formels. Enfin, nous suggérerons des extensions envisageables (et envisagées!) afin d'aboutir au *bootstrap* de Coq.

1 Formalisation du Calcul des Constructions

Le Calcul des Constructions est un λ-calcul assez évolué puisqu'il réunit la logique d'Automath (de Bruijn), et le système $\mathcal{F}\omega$. Il a été introduit par Coquand et Huet dans [6] et [7].

Nous différons de la présentation habituelle en adoptant les notations de de Bruijn pour les variables. Celles-ci ont l'avantage de ne pas poser les problèmes d'α-conversion qui caractérise la présentation avec variables nommées. Cela se paie car il y aura plus de résultats à démontrer.

L'intérêt de cette présentation n'est pas tant d'introduire une nouvelle formulation de ce calcul, que de formaliser une théorie mathématique récente. En parallèle de cette présentation figurera la traduction en Coq de ces définitions, et le cas échéant, l'équivalent Caml Light sous une forme légèrement modifiée pour être plus lisible.

Ce chapitre ne comprend que les définitions de la syntaxe, des règles de calcul et de typage. Les propriétés métathéoriques seront démontrées dans les chapitres suivants. Ces définitions forment le vocabulaire de base pour exprimer les résultats. On pourra constater tout le long de ce chapitre la forte ressemblance des formulations mathématique et formelle. Cette absence de codage est importante pour se convaincre plus facilement que l'on prouve bien ce que l'on croit. Une fois admise la validité de ces définitions (ce qui fait appel au bon sens commun), les résultats prouvés ne seront plus contestables, à moins de remettre en cause le formalisme employé.

1.1 Syntaxe

1.1.1 Les termes et les types

Nous considérons un λ-calcul qui comporte des types dépendants, qui correspondent en logique aux prédicats. Les prédicats sont ici des fonctions qui produisent une proposition, quand on leur applique des termes ou des types. Il est nécessaire d'avoir pour les types une syntaxe plus complexe que celle du λ-calcul simplement typé: les prédicats introduisent la même notion que l'abstraction du λ-calcul simplement typé, mais au niveau des types.

Il est apparu naturellement l'idée d'utiliser la même syntaxe pour les termes et les types, et les deux notions ne diffèrent que dans la position du terme dans le jugement de typage.

Avant de décrire l'algèbre des termes, nous introduisons la classe syntaxique des sortes.

Définition 1.1 *Dans le Calcul des Constructions, l'ensemble des sortes se compose de deux éléments:* Kind *et* Prop*.*

$$Sort := \mathrm{Kind} \mid \mathrm{Prop}$$

Traduction en Coq:

```
Inductive Set sort:=
   kind: sort
 | prop: sort.
```

En Caml Light, nous aurions écrit:

dénomination informelle	notation de de Bruijn	notation avec noms de variables	notation formelle
Environnement	Γ	Γ	`e : env`
Index	$\Gamma(n) = t$	$\Gamma(n) = (x, t)$	`(item t e n)`
Index avec relocation	$\uparrow^{n+1}\Gamma(n) = t$	$\Gamma(n) = (x, t)$	`(item_lift t e n)`
Sortes	Prop, Kind	Prop, Kind	`prop`, `kind`
Variables	n	x si $\Gamma(n) = (x, T)$	`(Ref n)`
Abstraction	$\lambda T.M$	$\lambda x{:}T.M$	`(Abs T M)`
Application	$(u\ v)$	$(u\ v)$	`(App u v)`
Produit dépendant	$\Pi T.U$	$\Pi x{:}T.U$	`(Prod T U)`
Produit non dépendant	$A \rightarrow B$	$A \rightarrow B$	`(Prod A (lift (S O) B))`
Relocation	$\uparrow^n_k T$	T	`(lift_rec n T k)`
Substitution	$M[k := N]$	$M[x\backslash N]$ si $\Gamma(k) = (x, T)$	`(subst_rec N M k)`

Figure 1 : Correspondance entre les styles de notations

```
type sort = kind
          | prop
;;
```

Les sortes sont des constantes particulières. On les distingue des termes pour pouvoir se placer plus facilement dans le cadre des systèmes de types purs (PTS [3]), dans lequel s'inscrivent le Calcul des Constructions (CC) et le Calcul des Constructions Etendu (ECC). De cette manière, tous les PTS ont la même définition des termes, paramétrée par l'ensemble des sortes.

La correspondance entre les notations informelles, mathématiques et formelles est décrite dans la figure 1.

Définition 1.2 *L'ensemble des termes est le plus petit ensemble tel que:*

(Srt) *si s est une sorte, alors s est un terme,*

(Ref) *si n est un entier, alors n est un terme,*

(Abs) *pour toute paire de termes (T, t), $\lambda T.t$ est un terme,*

(App) *pour toute paire de termes (u, v), $(u\ v)$ est un terme,*

(Prod) *pour toute paire de termes (T, U), $\Pi T.U$ est un terme.*

La classe syntaxique $Term$ peut être définie de la façon suivante:

$$Term := s \in Sort \mid n \in \mathbb{N} \mid \lambda t.t' \mid (t\ t') \mid \Pi t.t'$$

où t et t' désignent des éléments de $Term$.

Traduction en Coq:

```
Inductive Set term:=
    Srt: sort->term
  | Ref: nat->term
  | Abs: term->term->term
  | App: term->term->term
  | Prod: term->term->term.
```

```
type term = Srt of sort
          | Ref of int
          | Abs of term * term
          | App of term * term
          | Prod of term * term
;;
```

Remarque

- On utilise les notations à la Church pour l'abstraction et le produit. Si on ne précise pas le type de la variable introduite, on perd l'unicité du typage. Bien que cela soit possible, on n'introduira pas la notion de type principal, car cela serait redondant avec la possibilité de faire des abstractions sur des types (on n'utilise pas la généricité comme dans Caml Light; ici, le polymorphisme est explicite, et les fonctions dites polymorphes prennent en argument le type sous lequel elle sont utilisées).

- Le produit est une généralisation de la flèche du λ-calcul simplement typé quand il y a des types dépendants: le type du résultat d'une fonction peut dépendre de la valeur de l'argument. On peut constater la différence entre produit dépendant et non dépendant en figure 1.

- Afin d'éviter tous les problèmes d'α-conversion, on utilise la notation de de Bruijn pour les variables. Celles-ci sont représentées par des entiers qui indiquent le nombre de λ ou de Π à franchir pour tomber sur celui qui introduit cette variable (on dit que l'abstraction et le produit lient une variable dans leur sous-terme de droite). Cette notation n'est prévue que pour les variables liées. Mais il est possible d'utiliser cette notation pour des variables libres, dans la mesure où on rajoute un environnement qui caractérise celles-ci. De plus, cela évite d'avoir à renommer les variables de de Bruijn en variables nommées lors d'une descente récursive dans le terme.

- Il n'y a pas de constantes, mais cela ne réduit pas la puissance du calcul. Tous ce qui peut être démontré dans un système avec constantes peut l'être dans le système correspondant sans constantes: il suffit de remplacer toutes les occurrences de la constante par sa définition. On peut justifier cela en disant que le calcul sans constantes ne parle que des termes en forme δ-normale (sans constantes) et donc qu'il est nécessaire de décrire la δ-normalisation au niveau métathéorique.

Définition 1.3 *On définit la relation de sous-terme* direct*, notée* $\subset_{st}$*:*

$$x \subset_{st} y \overset{def}{\Longleftrightarrow} \begin{cases} \forall T, t, y = \lambda T.t \Rightarrow x = T \vee x = t \\ \forall u, v, y = (u\ v) \Rightarrow x = u \vee x = v \\ \forall T, U, y = \Pi T.U \Rightarrow x = T \vee x = U \end{cases}$$

TRADUCTION EN Coq:

```
Inductive subterm: term->term->Prop :=
    sbtrm_abs_l: (A,B:term)(subterm A (Abs A B))
  | sbtrm_abs_r: (A,B:term)(subterm B (Abs A B))
  | sbtrm_app_l: (A,B:term)(subterm A (App A B))
  | sbtrm_app_r: (A,B:term)(subterm B (App A B))
  | sbtrm_prod_l: (A,B:term)(subterm A (Prod A B))
  | sbtrm_prod_r: (A,B:term)(subterm B (Prod A B)).
```

1.1.2 L'environnement

L'environnement sert à caractériser les types des variables libres. Dans les λ-calculs avec variables nommées, il s'agit d'une liste de paire de noms et de types. Avec les notations de de Bruijn, ce n'est plus qu'une liste de types, puisque le rang dans la liste joue le rôle de nom de variable.

Définition 1.4 *Les environnements sont des listes de termes.*
TRADUCTION EN Coq:

```
Definition env := (list term).
```

On note $|\Gamma|$ la longueur de la liste. Pour désigner le n-ième élément d'une liste Γ (la tête de la liste ayant le rang 0), on utilisera la notation $\Gamma(n)$, qui suppose implicitement que $n < |\Gamma|$.

1.2 Les règles de calcul

Une fois la syntaxe des termes donnée, il reste à définir la notion de calcul sur ces termes. La règle de calcul à la base de tous les λ-calculs est la β-réduction. Cette règle permet de remplacer les variables symbolisant les arguments d'une fonction par les termes passés en argument à cette fonction. Elle fait donc appel à la notion de substitution (section 1.2.2). Contrairement au λ-calcul avec substitutions explicites, la fonction qui effectue la substitution est définie au niveau métathéorique.

Avec les notations de de Bruijn, la substitution n'est pas primitive. Il faut encore une notion intermédiaire: la relocation des variables (traduction de "lift" ou "weak" suivant les auteurs). Cette fonction est elle aussi métathéorique, puisque la substitution l'est déjà.

1.2.1 Relocation des variables de de Bruijn

Le sens d'une variable dépend de l'environnement dans lequel elle est considérée: la variable 0 code toujours la dernière variable introduite. Lorsque nous introduisons une nouvelle variable dans l'environnement, toutes les variables ayant un rang supérieur ou égal doivent être relogées afin d'éviter le phénomène de capture de variable (qui provoquerait un changement de sens pour cette variable). Nous allons définir une fonction qui permet de "créer" n variables numérotées de k à $k+n-1$, en augmentant de n toutes les variables supérieures ou égales à k.

Définition 1.5 *On définit la relocation d'amplitude n au niveau k sur un terme par récurrence sur la structure du terme grâce aux équations suivantes:*

$$\begin{array}{rcl} \uparrow_k^n s & = & s \\ \uparrow_p^k n & = & \begin{cases} k+n & si\ n \geq p \\ n & si\ n < p \end{cases} \end{array}$$

$$\begin{array}{rcl} \uparrow^n_k \lambda T.t & = & \lambda \uparrow^n_k T.\ \uparrow^n_{k+1} t \\ \uparrow^n_k (u\ v) & = & (\uparrow^n_k u\ \ \uparrow^n_k v) \\ \uparrow^n_k \Pi T.U & = & \Pi \uparrow^n_k T.\ \uparrow^n_{k+1} U \end{array}$$

On notera $\uparrow^n M$ la relocation au niveau 0.

TRADUCTION EN Coq:

```
Fixpoint lift_rec [n:nat;t:term]: nat->term :=
   [k:nat]Cases t of
      (Srt s) => (Srt s)
    | (Ref i) => Case (le_gt_dec k i) of
                   [_:(le k i)](Ref (plus n i))
                   [_:(gt k i)](Ref i)
                 end
    | (Abs T M)  => (Abs (lift_rec n T k) (lift_rec n M (S k)))
    | (App u v)  => (App (lift_rec n u k) (lift_rec n v k))
    | (Prod A B) => (Prod (lift_rec n A k) (lift_rec n B (S k)))
    end.

Definition lift:=[n:nat][t:term](lift_rec n t O).
```

```
let rec lift_rec n t k =
  match t with
    Srt s → Srt s
  | Ref i → if k ≤ i then Ref (n+i) else Ref i
  | Abs(t,m) → Abs((lift_rec n t k),(lift_rec n m (S k)))
  | App(u,v) → App((lift_rec n u k),(lift_rec n v k))
  | Prod(a,b) → Prod((lift_rec n a k),(lift_rec n b (S k)))
;;

let lift n t =
  lift_rec n t O
;;
```

Un terme de l'environnement est cohérent avec le plus grand préfixe de l'environnement qui ne le contient pas. La définition qui suit permet de reloger les variables d'un terme d'un environnement de façon à ce qu'il soit cohérent avec l'environnement tout entier.

Définition 1.6 *Le terme t est le n-ième élément de Γ relogé s'il existe un terme u qui soit le n-ième élément de Γ et tel que $t = \uparrow^{n+1} u$.*

$$(\texttt{item\_lift}\ \ t\ \Gamma\ n) \stackrel{def}{\Longleftrightarrow} \exists u, t = \uparrow^{n+1} u \wedge \Gamma(n) = u$$

TRADUCTION EN Coq:

```
Definition item_lift := [t:term][e:env][n:nat]
    (Ex2 ([u:term]t=(lift (S n) u)) [u:term](item ? u e n)).
```

On utilisera la notation $\uparrow^{n+1}\Gamma(n)$ pour désigner le n-ième élément de Γ relogé.

1.2.2 La substitution

L'opération précédente permet de définir la substitution: entre l'endroit où l'on considère un terme, et l'endroit où il doit être inséré, des variables ont pu apparaître à cause d'abstractions ou de produits. Il est nécessaire de reloger les variables du terme par lequel on substitue.

Exemple: Pour réduire le radical de $(\lambda a.\Pi b.\lambda c.2\ T)$, il faut remplacer le 2 par T, mais la variable 0 de T doit devenir la variable 2 (non pas 3, car l'abstraction du radical disparaît); ainsi, le terme précédent se réduit en $\Pi b.\lambda c.\ \uparrow^2 T$.

Définition 1.7 *On définit la substitution de la variable k du terme M par un terme N par récurrence sur la structure de M grâce aux équations suivantes:*

$$
\begin{array}{rcl}
s[k := N] & = & s \\
n[k := N] & = & \left\{ \begin{array}{ll} n-1 & si\ n > k \\ \uparrow^n N & si\ n = k \\ n & si\ n < k \end{array} \right. \\
(\lambda T.t)[k := N] & = & \lambda T[k := N].t[k+1 := N] \\
(u\ v)[k := N] & = & (u[k := N]\ v[k := N]) \\
(\Pi T.U)[k := N] & = & \Pi T[k := N].U[k+1 := N]
\end{array}
$$

Traduction en Coq:

```
[k:nat]Cases M of
     (Srt s) => (Srt s)
   | (Ref i) => Case (lt_eq_lt_dec k i) of
                  [C:{(lt k i)}+{k=i}]
                    Case C of
                      [_:(gt i k)](Ref (pred i))
                      [_:k=i](lift k N)
                    end
                  [_:(lt i k)](Ref i)
                end
   | (Abs A B)  => (Abs (subst_rec N A k) (subst_rec N B (S k)))
   | (App u v)  => (App (subst_rec N u k) (subst_rec N v k))
   | (Prod T U) => (Prod (subst_rec N T k) (subst_rec N U (S k)))
   end.

 Definition  subst:=[N,M:term](subst_rec N M O).
```

```
let rec subst_rec n m k =
  match m with
    Srt s → Srt s
  | Ref i → if k<i then Ref (pred i)
```

```
            else if k=i then lift k n
                 else Ref i
  | Abs(a,b) → Abs((subst_rec n a k),(subst_rec n b (S k)))
  | App(u,v) → App((subst_rec n u k),(subst_rec n v k))
  | Prod(t,u) → Prod((subst_rec n t k),(subst_rec n u (S k)))
;;

let subst n m =
 subst_rec n m 0
;;
```

Jusqu'ici, nous avons défini les objets que les programmes que nous voudrons extraire doivent manipuler, ainsi que les fonctions élémentaires sur les termes.

A partir de maintenant et jusqu'au chapitre 3, toutes les définitions et tous les résultats donnés n'auront aucun contenu calculatoire, ce qui signifie que lorsque l'on fera l'extraction, aucune des prochaines définitions n'aura d'influence sur le code extrait.

1.2.3 La β-réduction

Dans ce calcul, on ne considère qu'une seule règle de calcul: la β-réduction.

Définition 1.8 *On définit la relation de β-réduction de la façon usuelle. Pour cela on donne les règles d'inférence de la figure 2.*

TRADUCTION EN Coq:

```
Inductive red1: term->term->Prop :=
  beta:   (M,N,T:term)(red1 (App (Abs T M) N) (subst N M))
| abs_red_l: (M,M':term)(red1 M M') -> (N:term)(red1 (Abs M N) (Abs M' N))
| abs_red_r: (M,M':term)(red1 M M') -> (N:term)(red1 (Abs N M) (Abs N M'))
| app_red_l:(M1,N1:term)(red1 M1 N1)
              ->(M2:term)(red1 (App M1 M2)(App N1 M2))
| app_red_r:(M2,N2:term)(red1 M2 N2)
              ->(M1:term)(red1 (App M1 M2)(App M1 N2))
| prod_red_l:(M1,N1:term)(red1 M1 N1)
              ->(M2:term)(red1 (Prod M1 M2)(Prod N1 M2))
| prod_red_r:(M2,N2:term)(red1 M2 N2)
              ->(M1:term)(red1 (Prod M1 M2)(Prod M1 N2)).
```

Définition 1.9 *On définit la fermeture réflexive, transitive de $\triangleright_\beta$, et elle sera notée $\triangleright_\beta^*$.*

TRADUCTION EN Coq:

```
Inductive red [M:term]: term->Prop :=
  refl_red:     (red M M)
| trans_red:    (P,N:term)(red M P)->(red1 P N)->(red M N).
```

D'une manière générale, pour toute relation R, on notera R^* sa fermeture réflexive transitive et R^+ sa fermeture transitive.

$$(\text{BETA})\ \frac{}{(\lambda T.M\ N) \rhd_\beta M[0 := N]}$$

$$(\text{ABS-L})\ \frac{M \rhd_\beta M'}{\lambda M.N \rhd_\beta \lambda M'.N} \qquad (\text{ABS-R})\ \frac{M \rhd_\beta M'}{\lambda N.M \rhd_\beta \lambda N.M'}$$

$$(\text{APP-L})\ \frac{M_1 \rhd_\beta N_1}{(M_1\ M_2) \rhd_\beta (N_1\ M_2)} \qquad (\text{APP-R})\ \frac{M_2 \rhd_\beta N_2}{(M_1\ M_2) \rhd_\beta (M_1\ N_2)}$$

$$(\text{PROD-L})\ \frac{M_1 \rhd_\beta N_1}{\Pi M_1.M_2 \rhd_\beta \Pi N_1.M_2} \qquad (\text{PROD-R})\ \frac{M_2 \rhd_\beta N_2}{\Pi M_1.M_2 \rhd_\beta \Pi M_1.N_2}$$

Figure 2 : La relation de β-réduction

Définition 1.10 *La β-conversion est la fermeture réflexive, symétrique, transitive de $\rhd_\beta$. Elle sera notée $\approx_\beta$.*

TRADUCTION EN Coq:

```
Inductive conv [M:term]: term->Prop :=
  refl_conv: (conv M M)
| trans_conv_red: (P,N:term)(conv M P)->(red1 P N)->(conv M N)
| trans_conv_exp: (P,N:term)(conv M P)->(red1 N P)->(conv M N).
```

1.2.4 Termes fortement normalisables

La formulation usuelle de la normalisation forte est qu'il n'existe pas de suite infinie de termes réduits à partir de t. Il est équivalent de dire que t est fortement normalisable s'il est accessible (le prédicat `Acc` défini dans le prélude de Coq, mais nous rappelons sa définition) pour le symétrique de la relation de β-réduction.

Définition 1.11 *L'ensemble $\mathcal{SN}$ des termes fortement normalisables est défini inductivement comme le plus petit ensemble de termes vérifiant la propriété*

$$\forall t, (\forall u, t \rhd_\beta u \Rightarrow u \in \mathcal{SN}) \Rightarrow t \in \mathcal{SN}$$

TRADUCTION EN Coq:

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

Definition sn: term->Prop := (Acc ? (transp ? red1)).
```

1.2.5 Termes normaux

Il y a deux possibilités équivalentes de définir les termes normaux:

- soit aucun de ses sous-termes n'est un radical,
- soit il n'existe pas de terme vers lequel il se réduit.

La deuxième formulation semble la plus pratique, si on la compare à la façon dont on a défini les termes fortement normalisables: les termes normaux apparaissent comme des termes fortement normalisables particuliers.

Définition 1.12 *Un terme t est dit normal si pour tout terme u, t ne se réduit pas en u.*
TRADUCTION EN Coq:

```
Definition normal: term->Prop := [t:term](u:term)~(red1 t u).
```

Remarque Cette définition est équivalente à celle donnée informellement, même en logique intuitionniste: la négation de l'existentielle est équivalente à la quantification universelle de la négation d'une formule.

1.3 Le typage des termes

Jusqu'ici, le système décrit était un λ-calcul quelconque muni de la β-réduction. Nous nous attaquons maintenant à la relation de typage sur ces termes, qui caractérise le Calcul des Constructions.

Nous allons définir simultanément par récurrence deux sortes de jugements:

- un jugement de validité des environnements
- un jugement de typage

Définition 1.13 *On définit simultanément les deux ensembles de jugements dérivables comme la plus petite paire d'ensembles de jugements qui soit close par application des règles d'inférence de la figure 3.*
TRADUCTION EN Coq:

```
Mutual Inductive wf: env->Prop :=
  wf_nil: (wf (nil ?))
| wf_var: (e:env)(T:term)(s:sort)(typ e T (Srt s))->(wf (cons ? T e))

with typ: env->term->term->Prop :=
  type_prop: (e:env)(wf e)->(typ e (Srt prop) (Srt kind))
| type_var: (e:env)(wf e)->(v:nat)(t:term)(item_lift t e v)
               ->(typ e (Ref v) t)
| type_abs: (e:env)(T:term)(s1:sort)(typ e T (Srt s1))
```

$$(\text{WF-[]})\ \frac{}{[]\vdash} \qquad (\text{WF-VAR})\ \frac{\Gamma\vdash T : s}{\Gamma;T\vdash}(s\in\text{SORT})$$

$$(\text{PROP})\ \frac{\Gamma\vdash}{\Gamma\vdash \text{Prop} : \text{Kind}} \qquad (\text{VAR})\ \frac{\Gamma\vdash \qquad \uparrow^{n+1}\Gamma(n)=T}{\Gamma\vdash n : T}$$

$$(\text{ABS})\ \frac{\Gamma\vdash T : s_1 \qquad \Gamma;T\vdash M : U \qquad \Gamma;T\vdash U : s_2}{\Gamma\vdash \lambda T.M : \Pi T.U}(s_1,s_2\in\text{SORT})$$

$$(\text{APP})\ \frac{\Gamma\vdash v : V \qquad \Gamma\vdash u : \Pi V.U_r}{\Gamma\vdash (u\ v) : U_r[0:=v]}$$

$$(\text{PROD})\ \frac{\Gamma\vdash T : s_1 \qquad \Gamma;T\vdash U : s_2}{\Gamma\vdash (\Pi T.U) : s_2}\ (s_1,s_2\in\text{SORT})$$

$$(\text{CONV})\ \frac{\Gamma\vdash t : U \qquad \Gamma\vdash V : s \qquad U\approx_\beta V}{\Gamma\vdash t : V}\ (s\in\text{SORT})$$

Figure 3 : Règles de typage du Calcul des Constructions

```
                ->(M,U:term)(s2:sort)(typ (cons ? T e) U (Srt s2))
                  ->(typ (cons ? T e) M U)
                  ->(typ e (Abs T M) (Prod T U))
| type_app: (e:env)(v,V:term)(typ e v V)
              ->(u,Ur:term)(typ e u (Prod V Ur))
                 ->(typ e (App u v) (subst v Ur))
| type_prod: (e:env)(T:term)(s1:sort)(typ e T (Srt s1))
                 ->(U:term)(s2:sort)(typ (cons ? T e) U (Srt s2))
                   ->(typ e (Prod T U) (Srt s2))
| type_conv: (e:env)(t,U,V:term)(typ e t U)->(conv U V)
      ->(s:sort)(typ e V (Srt s))->(typ e t V).
```

Remarque

- Avec la règle `wf_var`, on peut introduire dans l'environnement deux genres de variables: des variables de terme (quand s = Prop) ou de type (quand s = Kind).
- Pour typer une variable, il ne faut pas oublier de reloger les variables du terme introduit dans l'environnement: pour un terme en n-ième position, on a introduit $n+1$ variables depuis que ce terme a été typé.

- Les règles de typage de l'abstraction et de l'application est à comparer avec celle du λ-calcul simplement typé, ce qui permet de se rendre compte en quoi le produit généralise la flèche.

- La règle de typage du produit peut se simplifier en:

$$(\textsc{Prod'})\ \frac{\Gamma; T \vdash U : s_2}{\Gamma \vdash (\Pi T.U) : s_2}(s_1, s_2 \in \textsc{Sort})$$

  puisque l'on verra que l'on ne peut déduire un jugement de typage que dans un environnement bien formé, ce qui signifie que T est typable par une sorte. On a gardé cette règle sous cette forme pour mieux coller au formalisme des PTS où les produits que l'on s'autorise à former sont contraints par une relation sur les sortes s_1 et s_2. On aurait pu simplifier la règle de l'abstraction de la même manière.

- La règle de conversion (`typ_conv`) dit que les types β-convertibles sont habités par les mêmes termes, à condition d'être bien formés. En anticipant un peu, on pourrait se dire que la prémisse $\Gamma \vdash V : s$ serait inutile si l'on demandait $U \rhd_\beta^* V$ au lieu de $U \approx_\beta V$ (en effet, nous montrerons le lemme d'auto-réduction, qui dit qu'un terme garde son type par β-réduction). Mais cela poserait des problèmes, car le lemme d'auto-réduction ne pourrait plus être montré sous une forme aussi forte (voir [17]).

2 Résultats non calculatoires

Dans ce chapitre sont énoncés les résultats de la métathéorie usuelle du Calcul des Constructions. La plupart des lemmes démontrés sont des traductions en indices de de Bruijn de ceux de Geuvers et Nederhof dans [11]. Les règles de typage n'étant pas identiques, ils sont démontrés dans un ordre différent.

Ces résultats n'ont aucun contenu calculatoire, ce qui signifie que les termes de preuve n'ont aucune influence sur les algorithmes que nous démontrerons dans le chapitre 3, bien qu'étant utilisés dans leur preuve de correction.

2.1 Propriétés algébriques de `lift` et `subst`

Les résultats énoncent des propriétés sur les fonctions **`lift`** et **`subst`**. Ils ont déjà étés démontrés par Huet dans [12] pour le λ-calcul pur. La traduction pour des termes avec annotations de type se fait très facilement.

2.1.1 Simplifications de `lift_rec` et `subst_rec`

Nous commençons par démontrer un lemme qui décrit l'effet des opérations **`lift_rec`** et **`subst_rec`** sur les variables.

Lemme 2.1 *Pour tout entier k, n, p et tout terme M,*

$$
\begin{aligned}
\uparrow_p^k n &= \begin{cases} k+n & si\ n \geq p \\ n & si\ n < p \end{cases} \\
n[k := u] &= \begin{cases} n-1 & si\ n > k \\ \uparrow^n u & si\ n = k \\ n & si\ n < k \end{cases}
\end{aligned}
$$

Traduction en Coq:

```
Lemma lift_ref_ge: (k,n,p:nat)(le p n)
          ->(lift_rec k (Ref n) p)=(Ref (plus k n)).
Lemma lift_ref_lt: (k,n,p:nat)(gt p n)
          ->(lift_rec k (Ref n) p)=(Ref n).
Lemma subst_ref_gt: (u:term)(n,k:nat)(gt n k)
          ->(subst_rec u (Ref n) k)=(Ref (pred n)).
Lemma subst_ref_eq: (u:term)(n:nat)(subst_rec u (Ref n) n)=(lift n u).
Lemma subst_ref_lt: (u:term)(n,k:nat)(gt k n)
          ->(subst_rec u (Ref n) k)=(Ref n).
```

Preuve Les preuves découlent immédiatement de la définition des fonctions **`lift_rec`** et **`subst_rec`**. Bien que la formulation mathématique de ce lemme soit identique à la définition de la relocation et de la substitution de la variable, il y a quand même quelque chose à démontrer: les conditions doivent être disjointes pour assurer que la définition associe une seule valeur par argument. Il s'agit d'un résultat d'arithmétique trivial. ■

Ces lemmes permettent de simplifier les expressions de relocation et de substitution sur les variables. Ainsi, lorsque des contraintes sur les indices de de Bruijn permettent de savoir de quelle manière on peut simplifier une formule, il suffit d'appliquer le lemme correspondant, au lieu d'avoir à raisonner par cas et de devoir traiter des cas absurdes.

2.1.2 Propriétés de simplification, commutativité et d'associativité

Chacun des sept résultats généraux démontrés est le reformulé sous une forme moins générale, mais plus agréable à utiliser. Dans tous les cas, la version forte doit nécessairement être démontrée avant la version affaiblie.

Tous se démontrent assez facilement (la principale difficulté est de trouver la formulation suffisamment générale qui puisse être démontrée directement) et de façons similaires. On ne détaillera pas ces preuves. Seules quelques indications seront données à la fin.

Lemme 2.2 *La relocation d'amplitude nulle est l'identité sur les termes.*

$$\uparrow_k^0 M = M$$

Traduction en Coq:

```
Lemma lift_rec0: (M:term)(k:nat)(lift_rec O M k)=M.
Lemma lift0: (M:term)(lift O M)=M.
```

Remarque On pourra vérifier que les équations suivantes se simplifient bien au vu de ce résultat.

Lemme 2.3 *La composée d'une relocation avec avec une autre relocation au niveau d'une des variables créées par la première se simplifie en une seule relocation. Formellement,*

$$k \leq i \leq k+n \Rightarrow \uparrow_i^p (\uparrow_k^n M) = \uparrow_k^{p+n} M$$

Traduction en Coq:

```
Lemma simpl_lift_rec: (M:term)(n,k,p,i:nat)(le i (plus k n))->(le k i)
        ->(lift_rec p (lift_rec n M k) i)
            =(lift_rec (plus p n) M k).
Lemma simpl_lift: (M:term)(n:nat)(lift (S n) M)=(lift (S O) (lift n M)).
```

Lemme 2.4 *Dans le cas où la simplification précédente n'a pas lieu, il est possible de permuter les deux relocations:*

$$i \leq k \Rightarrow \uparrow_i^p (\uparrow_k^n M) = \uparrow_{p+k}^n (\uparrow_i^p M)$$

Traduction en Coq:

```
Lemma permute_lift_rec: (M:term)(n,k,p,i:nat)(le i k)
     ->(lift_rec p (lift_rec n M k) i)
         =(lift_rec n (lift_rec p M i) (plus p k)).
Lemma permute_lift: (M:term)(k:nat)
    (lift (S O) (lift_rec (S O) M k))=(lift_rec (S O) (lift (S O) M) (S k)).
```

Lemme 2.5 *La composée d'une relocation d'amplitude non nulle avec une substitution sur une des variables créées par la relocation est équivalente à une relocation. Formellement,*

$$k \leq p \leq n+k \Rightarrow (\uparrow_k^{n+1} M)[p := N] = \uparrow_k^n M$$

Traduction en Coq:

```
Lemma simpl_subst_rec: (N,M:term)(n,p,k:nat)(le p (plus n k))->(le k p)
       ->(subst_rec N (lift_rec (S n) M k) p)=(lift_rec n M k).
Lemma simpl_subst: (N,M:term)(n,p:nat)(le p n)
              ->(subst_rec N (lift (S n) M) p)=(lift n M).
```

Lemme 2.6 *La composée d'une relocation avec une substitution d'une variable plus grande que le niveau de la relocation est équivalente à la composée d'une substitution avec une relocation:*

$$k \leq p \Rightarrow \uparrow_k^n (M[p := N]) = (\uparrow_k^n M)[n+p := N]$$

Traduction en Coq:

```
Lemma commut_lift_subst_rec: (M,N:term)(n,p,k:nat)(le k p)
          ->(lift_rec n (subst_rec N M p) k)
             =(subst_rec N (lift_rec n M k) (plus n p)).
Lemma commut_lift_subst: (M,N:term)(k:nat)
  (subst_rec N (lift (S O) M) (S k))=(lift (S O) (subst_rec N M k)).
```

Lemme 2.7 *La relocation se distribue sur toute substitution sur une variable plus petite que le niveau de la relocation:*

$$\uparrow_{p+k}^n (M[p := N]) = (\uparrow_{p+k+1}^n M)[p := \uparrow_k^n N]$$

Traduction en Coq:

```
Lemma distr_lift_subst_rec: (M,N:term)(n,p,k:nat)
           (lift_rec n (subst_rec N M p) (plus p k))
              =(subst_rec (lift_rec n N k) (lift_rec n M (S (plus p k))) p).
Lemma distr_lift_subst: (M,N:term)(n,k:nat)
           (lift_rec n (subst N M) k)
              =(subst (lift_rec n N k) (lift_rec n M (S k))).
```

Lemme 2.8 *La substitution est distributive sur elle-même.*

$$(M[p := N])[p + n := P] = (M[p + n + 1 := P])[p := N[n := P])]$$

TRADUCTION EN Coq:

```
Lemma distr_subst_rec: (M,N,P:term)(n,p:nat)
    (subst_rec P (subst_rec N M p) (plus p n))
    =  (subst_rec (subst_rec P N n) (subst_rec P M (S (plus p n))) p).
Lemma distr_subst: (P,N,M:term)(k:nat)
          (subst_rec P (subst N M) k)
             =(subst (subst_rec P N k) (subst_rec P M (S k))).
```

Remarque Cette propriété s'énonce apparemment beaucoup plus simplement en variables nommées:

$$M[x := N][y := P] = M[y := P][x := N[y := P]] \text{ si } x \neq y$$

Mais les apparences sont trompeuses: la condition annexe sur les noms des variables substituées pose autant de problèmes que la condition annexe de la définition de la substitution sur l'abstraction.

PREUVE

Toutes les preuves se font par récurrence sur M, puisque connaître le constructeur qui forme M permet de simplifier le but à résoudre en utilisant les définitions de **lift_rec** et de **subst_rec**. A chaque fois, le cas de base correspondant aux sortes est trivial, et les cas récursifs (abstraction, application et produit) se résolvent en appliquant les deux hypothèses de récurrence avec des arguments faciles à trouver. Seul le cas des variables n'est pas immédiat, mais il suffit de se laisser guider par les définitions de **subst_rec** et **lift_rec**, en utilisant les lemmes de la section précédente dès que les contraintes sur les paramètres entiers le permettent. ■

2.2 Résultats concernant la β-réduction

Dans les deux premiers paragraphes de cette section, nous abordons une partie du développement peu intéressante, puisqu'il s'agit de prouver les propriétés élémentaires de la β-réduction. Ces lemmes sont assez triviaux, mais ne peuvent pas être déduits automatiquement par le système.

Il est cependant très utile de les démontrer, car ils simplifieront de beaucoup les preuves ultérieures. En effet, un certain nombre d'entre eux peuvent être déclarés comme “indices” pour Coq, c'est-à-dire qu'à la demande de l'utilisateur, le système va essayer de résoudre le but courant en appliquant ces lemmes. Ce genre d'outils est indispensable dans un système d'aide à la preuve puisqu'il libère l'utilisateur de longues recherches susceptibles de lui faire perdre le fil du raisonnement.

Par exemple, si l'on voit que le but courant peut être résolu en appliquant la réflexivité de l'égalité, il n'est pas nécessaire d'aller rechercher dans l'environnement le nom exact du théorème: cela peut se faire automatiquement.

Evidemment, tous les lemmes ne peuvent pas être donnés comme indices: ceux pour lesquelles des variables figurent dans les prémisses mais pas dans la conclusion ne sont pas utilisables facilement. Il faudrait pour cela faire une recherche de preuves par unification d'ordre supérieur. Des tactiques de ce type sont en cours de développement.

Afin de ne pas alourdir inutilement le discours, la plupart des lemmes de cette section seront formulés en français, sans preuve. La formulation en Coq sera toutefois donnée en petits caractères afin que le lecteur puisse se rendre compte du degré de précision nécessaire lors de la formalisation. Celle-ci demande parfois d'énoncer des trivialités, mais ce n'est quand même pas excessif.

2.2.1 Propriétés de `red`

La β-réduction a été définie grâce à un type inductif, donc le système engendre automatiquement un théorème permettant de raisonner par récurrence sur le nombre de pas de réduction. Pour chaque terme M, ce théorème (`red_ind`) donne une condition suffisante pour qu'un ensemble de termes P contienne tous les termes vers lesquels se réduit M:

$$\left.\begin{array}{r} M \in P \\ \forall R, N \in Term, \quad \left.\begin{array}{r} M \rhd_\beta^* R \\ R \rhd_\beta N \\ R \in P \end{array}\right\} \Rightarrow N \in P \end{array}\right\} \Rightarrow \forall t, M \rhd_\beta^* t \Rightarrow t \in P$$

Dans une prochaine démonstration, nous aurons besoin d'un autre schéma de récurrence: celui qui pour chaque terme N, caractérise l'ensemble des termes se réduisant vers N. Ces deux schémas de récurrence sont équivalents, et on peut démontrer le second à partir du premier:

Pour pour tout terme N, et tout ensemble de termes P:

$$\left.\begin{array}{r} N \in P \\ \forall M, R \in Term, \quad \left.\begin{array}{r} M \rhd_\beta R \\ R \rhd_\beta^* N \\ R \in P \end{array}\right\} \Rightarrow M \in P \end{array}\right\} \Rightarrow \forall M, M \rhd_\beta^* N \Rightarrow M \in P$$

```
Lemma red1_red_ind: (N:term)(P:term->Prop)(P N)
                    ->((M,R:term)(red1 M R)->(red R N)->(P R)->(P M))
                     ->(M:term)(red M N)->(P M).
```

La β-réduction contient la β-réduction d'un pas et elle est transitive:

$$\frac{M \rhd_\beta N}{M \rhd_\beta^* N} \qquad \frac{M \rhd_\beta^* N \qquad N \rhd_\beta^* P}{M \rhd_\beta^* P}$$

```
Lemma one_step_red: (M,N:term)(red1 M N)->(red M N).
Lemma trans_red_red: (M,N,P:term)(red M N)->(red N P)->(red M P).
```

On montre des propriétés plus fortes de transitivité de la β-réduction:

$$\frac{u \rhd_\beta^* u_0 \qquad v \rhd_\beta^* v_0}{(u\ v) \rhd_\beta^* (u_0\ v_0) \qquad \lambda u.v \rhd_\beta^* \lambda u_0.v_0 \qquad \Pi u.v \rhd_\beta^* \Pi u_0.v_0}$$

```
Lemma red_red_app: (u,u0,v,v0:term)(red u u0)->(red v v0)->(red (App u v)(App u0 v0)).
Lemma red_red_abs: (u,u0,v,v0:term)(red u u0)->(red v v0)->(red (Abs u v)(Abs u0 v0)).
Lemma red_red_prod: (u,u0,v,v0:term)(red u u0)->(red v v0)->(red (Prod u v)(Prod u0 v0)).
```

La β-réduction est conservée par relocation et par substitution:

$$\frac{u \rhd_\beta v}{\uparrow_k^n u \rhd_\beta \uparrow_k^n v} \qquad \frac{t \rhd_\beta u}{t[k:=a] \rhd_\beta u[k:=a] \qquad a[k:=t] \rhd_\beta^* a[k:=u]}$$

```
Lemma red1_lift: (u,v:term)(red1 u v)->(n,k:nat)(red1 (lift_rec n u k) (lift_rec n v k)).
Lemma red1_subst_r: (t,u:term)(red1 t u)
                 ->(a:term)(k:nat)(red1 (subst_rec a t k) (subst_rec a u k)).
Lemma red1_subst_l: (a,t,u:term)(k:nat)(red1 t u)
                 ->(red (subst_rec t a k) (subst_rec u a k)).
```

Un terme normal ne se réduit que vers lui-même:

```
Lemma red_normal: (u,v:term)(red u v)->(normal u)->(u=v).
```

Un produit se réduit en un produit (formellement, on utilise le codage imprédicatif de l'existentielle):

$$\forall u, v, t \in Term, \Pi u.v \rhd_\beta^* t \Rightarrow \exists a, b \in Term, \begin{cases} t = \Pi a.b \\ u \rhd_\beta^* a \\ v \rhd_\beta^* b \end{cases}$$

```
Lemma red_prod_prod: (u,v,t:term)(red (Prod u v) t)
              ->(P:Prop)((a,b:term)t=(Prod a b)->(red u a)->(red v b)->P)
                     ->P.
```

2.2.2 Propriétés de conv

La relation $\approx_\beta$ contient $\rhd_\beta^*$ et le symétrique de $\rhd_\beta$:

```
Lemma one_step_conv_exp: (M,N:term)(red1 M N)->(conv N M).
Lemma red_conv: (M,N:term)(red M N)->(conv M N).
```

Elle est symétrique et transitive:

```
Lemma sym_conv: (M,N:term)(conv M N)->(conv N M).
Lemma trans_conv_conv: (M,N,P:term)(conv M N)->(conv N P)->(conv M P).
```

Elle est préservée par produit, relocation et substitution:

```
Lemma conv_conv_prod: (a,b,c,d:term)(conv a b)->(conv c d)
                ->(conv (Prod a c)(Prod b d)).
Lemma conv_conv_lift: (a,b:term)(n,k:nat)(conv a b)
                ->(conv (lift_rec n a k) (lift_rec n b k)).
Lemma conv_conv_subst: (a,b,c,d:term)(k:nat)(conv a b)->(conv c d)
                ->(conv (subst_rec a c k)(subst_rec b d k)).
```

2.2.3 Propriétés de clôture de $\mathcal{SN}$

Un terme fortement normalisable ne se réduit que vers des termes fortement normalisables:

Lemme 2.9 *Pour tous termes A et B,*

$$A \in \mathcal{SN} \wedge A \rhd^*_\beta B \Rightarrow B \in \mathcal{SN}$$

Traduction en Coq:

```
Lemma sn_red_sn: (a,b:term)(sn a)->(red a b)->(sn b).
```

Preuve

Par récurrence sur $a \rhd^*_\beta b$, en inversant le prédicat `sn`. ■

Lemme 2.10 *Un produit est fortement normalisable à partir du moment où ses deux sous-termes le sont:*

Pour tous termes A et B,

$$A \in \mathcal{SN} \wedge B \in \mathcal{SN} \Rightarrow \Pi A.B \in \mathcal{SN}$$

Traduction en Coq:

```
Lemma sn_prod: (A:term)(sn A)->(B:term)(sn B)->(sn (Prod A B)).
```

Lemme 2.11 *Si un terme M dans lequel on a effectué une substitution est fortement normalisable, alors M l'est aussi:*

$$(M[0 := T]) \in \mathcal{SN} \Rightarrow M \in \mathcal{SN}$$

Traduction en Coq:

```
Lemma sn_subst: (T,M:term)(sn (subst T M))->(sn M).
```

Preuve

Cela est dû au fait qu'à toute réduction à partir de M donne lieu à une réduction dans $M[0 := T]$. Donc s'il n'existe pas de chemin de réduction infini à partir de ce dernier terme, il n'en existe pas à partir de M. Cela se fait formellement par récurrence sur l'hypothèse. ■

Nous prouvons que la β-expansion peut être anticipée par rapport à l'ordre de sous-terme direct:

Lemme 2.12 *L'ordre de sous-terme direct et le symétrique de la β-réduction vérifient la condition de commutation suivante avec $R_1 = \subset_{st}$ et $R_2 = \lhd_\beta$ (voir figure 4):*

$$\forall x, y, z, y\ R_1\ x \wedge z\ R_2\ y \Rightarrow \exists y', \left\{ \begin{array}{l} y'\ R_2\ x \\ z\ R_1\ y' \end{array} \right.$$

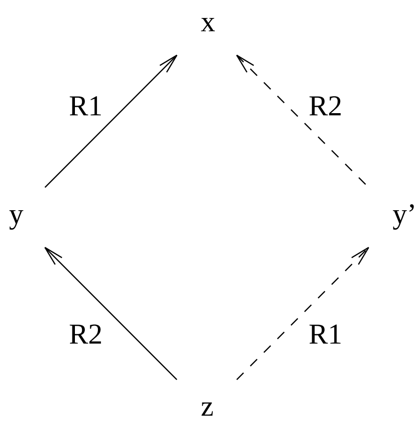

Figure 4 : Propriété `commut R1 R2` $\equiv (R2; R1 \subseteq R1; R2)$

Traduction en Coq:

```
Lemma commut_red1_subterm: (commut ? subterm (transp ? red1)).
```

Preuve

Par cas sur x. Les cas des sortes et des variables est trivial puisque les sortes et les variables n'ont aucun sous-terme. Pour les trois autres constructeurs, il est facile de trouver le bon terme car la β-réduction passe au contexte. ■

Lemme 2.13 *Tout sous-terme d'un terme fortement normalisable est fortement normalisable:*

$$A \in \mathcal{SN} \wedge B \subset_{st} A \Rightarrow B \in \mathcal{SN}$$

Traduction en Coq:

```
Lemma subterm_sn: (a,b:term)(sn a)->(subterm b a)->(sn b).
```

Preuve

Par récurrence sur la première hypothèse. Pour prouver que b est fortement normalisable, il suffit d'appliquer l'hypothèse de récurrence sur le terme que le lemme précédent fait intervenir. ■

2.2.4 Conservation des sortes par β-expansion

Définition 2.1 *Pour toute sorte s, on définit $\mathsf{MemSort}_s$ l'ensemble des termes qui ont la sorte s comme sous-terme. C'est le plus petit ensemble de termes vérifiant les propriétés:*

$$s \in \mathsf{MemSort}_s$$

$$\begin{array}{rcl}\forall u,v \in Term, u \in \mathsf{MemSort}_s \vee v \in \mathsf{MemSort}_s & \Rightarrow & \Pi u.v \in \mathsf{MemSort}_s \\ \forall u,v \in Term, u \in \mathsf{MemSort}_s \vee v \in \mathsf{MemSort}_s & \Rightarrow & \lambda u.v \in \mathsf{MemSort}_s \\ \forall u,v \in Term, u \in \mathsf{MemSort}_s \vee v \in \mathsf{MemSort}_s & \Rightarrow & (u\ v) \in \mathsf{MemSort}_s\end{array}$$

TRADUCTION EN Coq:

```
Inductive mem_sort [s:sort]: term->Prop :=
   mem_eq: (mem_sort s (Srt s))
 | mem_prod_l: (u,v:term)(mem_sort s u)->(mem_sort s (Prod u v))
 | mem_prod_r: (u,v:term)(mem_sort s v)->(mem_sort s (Prod u v))
 | mem_abs_l: (u,v:term)(mem_sort s u)->(mem_sort s (Abs u v))
 | mem_abs_r: (u,v:term)(mem_sort s v)->(mem_sort s (Abs u v))
 | mem_app_l: (u,v:term)(mem_sort s u)->(mem_sort s (App u v))
 | mem_app_r: (u,v:term)(mem_sort s v)->(mem_sort s (App u v)).
```

Lemme 2.14 *Tout terme se réduisant vers une sorte admet cette sorte comme sous-terme:*

$$t \rhd_\beta s \Rightarrow t \in \mathsf{MemSort}_s$$

TRADUCTION EN Coq:

```
Lemma red_sort_mem: (t:term)(s:sort)(red t (Srt s))->(mem_sort s t).
```

PREUVE On démontre d'abord ce résultat pour les opérations de relocation et de substitution:

```
Lemma mem_sort_lift: (t:term)(n,k:nat)(s:sort)(mem_sort s (lift_rec n t k))
                       ->(mem_sort s t).
Lemma mem_sort_subst: (b,a:term)(n:nat)(s:sort)(mem_sort s (subst_rec a b n))
                        ->(mem_sort s a)\/(mem_sort s b).
```

Grâce au schéma de récurrence défini dans la section précédente, on se ramène facilement au cas de la β-réduction d'un pas, puis par récurrence sur cette hypothèse, la démonstration est facile: pour la règle β, on utilise le lemme **mem_sort_subst** qu'on vient de démontrer. ■

2.3 Confluence du Calcul des Constructions

Dans cette section, nous démontrons la confluence de la β-réduction en utilisant l'argument de Tait et Martin-Löf qui consiste à trouver une relation localement confluente dont la fermeture réflexive transitive est équivalente à la β-réduction: il s'agit de la β-réduction parallèle.

$$(\textsc{Beta'})\ \frac{M \rhd_{/\!/} M' \qquad N \rhd_{/\!/} N'}{(\lambda T.M\ N) \rhd_{/\!/} M'[0 := N']}$$

$$(\textsc{Srt})\ \frac{}{s \rhd_{/\!/} s} \qquad (\textsc{Ref})\ \frac{}{n \rhd_{/\!/} n}$$

$$(\textsc{Abs})\ \frac{M \rhd_{/\!/} M' \qquad T \rhd_{/\!/} T'}{\lambda T.M \rhd_{/\!/} \lambda T'.M'} \qquad (\textsc{App})\ \frac{M \rhd_{/\!/} M' \qquad N \rhd_{/\!/} N'}{(M\ N) \rhd_{/\!/} (M'\ N')}$$

$$(\textsc{Prod})\ \frac{M \rhd_{/\!/} M' \qquad N \rhd_{/\!/} N'}{\Pi M.N \rhd_{/\!/} \Pi M'.N'}$$

Figure 5 : La relation de β-réduction parallèle

2.3.1 La β-réduction parallèle

Définition 2.2 *La relation de β-réduction parallèle est la plus petite relation sur les termes stable par application des règles de la figure 5. Elle sera notée $\rhd_{/\!/}$.*

Traduction en Coq:

```
Inductive par_red1 : term -> term -> Prop :=
  par_beta    : (M,M':term)(par_red1 M M') -> (N,N':term)(par_red1 N N')
                    ->(T:term)(par_red1 (App (Abs T M) N) (subst N' M'))
| sort_par_red: (s:sort)(par_red1 (Srt s) (Srt s))
| ref_par_red : (n:nat)(par_red1 (Ref n) (Ref n))
| abs_par_red : (M,M':term)(par_red1 M M') ->(T,T':term)(par_red1 T T')
                -> (par_red1 (Abs T M) (Abs T' M'))
| app_par_red : (M,M':term)(par_red1 M M') -> (N,N':term)(par_red1 N N') ->
                            (par_red1 (App M N) (App M' N'))
| prod_par_red : (M,M':term)(par_red1 M M') -> (N,N':term)(par_red1 N N') ->
                            (par_red1 (Prod M N) (Prod M' N')).

Definition par_red := (clos_trans term par_red1).
```

La fermeture transitive de cette relation sera notée $\rhd_{/\!/}^{+}$.

2.3.2 Résultats élémentaires sur la β-réduction parallèle

- La β-réduction parallèle est réflexive.

  ```
  Lemma refl_par_red1 : (M:term)(par_red1 M M).
  ```

- Elle contient la relation de β-réduction d'un pas.

```
Lemma red1_par_red1 : (M,N:term)(red1 M N) -> (par_red1 M N).
```

- Sa fermeture transitive est équivalente à la β-réduction.

```
Lemma red_par_red : (M,N:term)(red M N) -> (par_red M N).
Lemma par_red_red : (M,N:term) (par_red M N) -> (red M N).
```

- Elle est invariante par relocation et substitution.

```
Lemma par_red1_lift: (n:nat)(a,b:term)(par_red1 a b)
       ->(k:nat)(par_red1 (lift_rec n a k) (lift_rec n b k)).
Lemma par_red1_subst: (c,d:term)(par_red1 c d)->(a,b:term)(par_red1 a b)
       ->(k:nat)(par_red1 (subst_rec a c k) (subst_rec b d k)).
```

- Une abstraction se réduit toujours en une abstraction.

```
Lemma inv_par_red_abs: (P:Prop)(T,U,x:term)(par_red1 (Abs T U) x)
       ->((T',U':term)(x=(Abs T' U'))->(par_red1 U U')->P)->P.
```

2.3.3 Résultats de confluence

Nous commençons par définir la notion de confluence pour une relation:

Définition 2.3 *Une relation R est dite fortement confluente si et seulement si elle vérifie*

$$\left.\begin{array}{l} x\ R\ y \\ x\ R\ y' \end{array}\right\} \Rightarrow \exists z. \left\{\begin{array}{l} y\ R\ z \\ y'\ R\ z \end{array}\right.$$

pour tous termes x, y et y' (voir figure 6).

TRADUCTION EN Coq:

```
Definition confluent :=
     [R:term->term->Prop](x,y:term)(R x y)->(y':term)(R x y')
       ->(Ex2 [z:term](R y z) [z:term](R y' z)).
```

Remarque La notion de confluence forte peut se définir à partir de celle de commutation de la section précédente. R est fortement confluente si et seulement si elle commute avec son symétrique:

(confluent R) $\Leftrightarrow$ (commut R [x,y:term](R y x))

Lemme 2.15 *La relation de β-réduction parallèle $\rhd_{/\!/}$ est fortement confluente.*

TRADUCTION EN Coq:

```
Lemma confluence_par_red1: (confluent par_red1).
```

PREUVE

On raisonne par récurrence sur l'hypothèse $x \rhd_{/\!/} y$ (notations de la figure 6), puis inversion de l'hypothèse $x \rhd_{/\!/} y'$. Cela nous génère les huit cas (quatre pour le cas de l'application, et un pour chaque autre constructeur) suivant la règle utilisée. Pour chacun de ces cas, il est très facile de trouver le terme commun vers lequel se réduisent y et y'. ■

$$
\begin{tikzcd}
 & x \arrow[dl, "R"'] \arrow[dr, "R"] & \\
y \arrow[dr, dashed, "R"'] & & y' \arrow[dl, dashed, "R"] \\
 & z &
\end{tikzcd}
$$

Figure 6 : Propriété de confluence forte

Lemme 2.16 *La relation de β-réduction est confluente, ou de manière équivalente, sa fermeture réflexive transitive est fortement confluente.*

TRADUCTION EN Coq:

```
Lemma confluence_red: (confluent red).
```

PREUVE

Il suffit de démontrer la confluence de $\rhd_{/\!/}^{+}$, car celle-ci est équivalente à $\rhd_{\beta}^{*}$.

```
Lemma confluence_par_red: (confluent par_red).
```

Cette propriété est une conséquence de la confluence de $\rhd_{/\!/}$: une relation fortement confluente est confluente, ce qui se démontre avec le résultat intermédiaire:

```
Lemma strip_lemma: (x,y:term)(par_red x y)->(y':term)(par_red1 x y')
   ->(Ex2 [z:term](par_red1 y z) [z:term](par_red y' z)).
```

$\blacksquare$

De ce lemme, nous pouvons déduire assez facilement le théorème suivant:

Théorème 1 *La relation de β-réduction vérifie la propriété de Church-Rosser.*

TRADUCTION EN Coq:

```
Theorem church_rosser: (u,v:term)(conv u v)
              ->(Ex2 [t:term](red u t) [t:term](red v t)).
```

Remarque On a démontré la version non calculatoire de ce théorème. La preuve de la version calculatoire serait beaucoup plus difficile, car elle exclut une démonstration par récurrence sur l'hypothèse $u \approx_\beta v$, qui n'a aucun contenu calculatoire, mais elle fournirait un algorithme qui calcule un terme vers lequel se réduisent deux termes convertibles, ce que nous ferons dans la section 3.3.

Corollaire 2.1 *Si deux produits sont convertibles, alors leurs sous-termes gauches (resp. droits) sont convertibles:*

$$\Pi a.c \approx_\beta \Pi b.d \Rightarrow \begin{cases} a \approx_\beta b \\ c \approx_\beta d \end{cases}$$

TRADUCTION EN Coq:

```
Lemma inv_conv_prod_r: (a,b,c,d:term)(conv (Prod a c)(Prod b d))->(conv c d).
Lemma inv_conv_prod_l: (a,b,c,d:term)(conv (Prod a c)(Prod b d))->(conv a b).
```

PREUVE
On applique `church_rosser`. Il existe donc un terme t vers lequel se réduisent $\Pi a.c$ et $\Pi b.d$. En appliquant deux fois `red_prod_prod`, on en déduit que a et b d'une part, et c et d d'autre part, se réduisent vers un même terme. ■

On démontre encore quelques conséquences simples de la propriété de Church-Rosser:

- Deux sortes convertibles sont égales.

```
Lemma conv_sort: (s1,s2:sort)(conv (Srt s1) (Srt s2))->s1=s2.
```

- Kind et Prop ne sont pas convertibles.

```
Lemma conv_kind_prop: ~(conv (Srt kind) (Srt prop)).
```

- Une sorte et un produit ne sont jamais convertibles.

```
Lemma conv_sort_prod: (s:sort)(t,u:term)~(conv (Srt s) (Prod t u)).
```

2.4 Métathéorie du Calcul des Constructions

2.4.1 Lemmes d'inversion

Lemme 2.17 *Les termes d'un environnement bien formé sont bien formés:*

$$\Gamma; t; \Delta \vdash \Rightarrow \exists s \in Sort, \Gamma \vdash t : s$$

TRADUCTION EN Coq: (La proposition (`trunc` n e f) est vraie si et seulement si f est la liste e privée de ses n éléments de tête)

```
Lemma wf_sort: (n:nat)(e,f:env)(trunc ? (S n) e f)->(wf e)
      ->(t:term)(item ? t e n)
       ->(Ex [s:sort](typ f t (Srt s))).
```

PREUVE
Par récurrence sur la longueur de Δ. ■

Pour tout jugement dérivable, l'environnement de ce jugement est bien formé. Autrement dit:

Lemme 2.18 *La règle*

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash}$$

est admissible.

TRADUCTION EN Coq:

```
Lemma typ_wf: (e:env)(t,T:term)(typ e t T)->(wf e).
```

PREUVE

La démonstration de ce résultat est très courte (deux tactiques suffisent): il s'agit d'une récurrence sur la dérivation du jugement de typage, et pour chaque cas, la bonne formation de l'environnement se déduit soit des prémisses de la règle employée, soit de l'hypothèse de récurrence. ∎

Les lemmes d'inversion sont utilisés à chaque fois que l'on désire tirer des informations d'une hypothèse consistant en un jugement de typage dont le constructeur du terme de gauche est connu. Ils permettent de déduire que les sous-termes d'un terme bien typé sont bien typés, mais aussi de raisonner par récurrence sur le terme au lieu de la dérivation du jugement. Il y en a un par constructeur de terme ou de sorte:

Théorème 2

$$\begin{array}{rcl}
\Gamma \vdash \mathrm{Kind} : T & \Rightarrow & \bot \\
\Gamma \vdash \mathrm{Prop} : T & \Rightarrow & T \approx_\beta \mathrm{Kind} \\
\Gamma \vdash n : T & \Rightarrow & T \approx_\beta \uparrow^{n+1}\Gamma(n) \\
\Gamma \vdash \lambda A.M : U & \Rightarrow & \exists T \in Term, \exists s_1, s_2 \in Sort, \left\{ \begin{array}{l} \Gamma \vdash A : s_1 \\ \Gamma; A \vdash M : T \\ \Gamma; A \vdash T : s_2 \\ U \approx_\beta \Pi A.T \end{array} \right. \\
\Gamma \vdash (u\ v) : T & \Rightarrow & \exists V, U_r \in Term, \left\{ \begin{array}{l} \Gamma \vdash u : \Pi V.U_r \\ \Gamma \vdash v : V \\ T \approx_\beta U_r[0 := v] \end{array} \right. \\
\Gamma \vdash \Pi u.v : T & \Rightarrow & \exists s_1, s_2 \in Sort, \left\{ \begin{array}{l} \Gamma \vdash T : s_1 \\ \Gamma; T \vdash U : s_2 \\ T \approx_\beta s_2 \end{array} \right.
\end{array}$$

TRADUCTION EN Coq:

```
Lemma inv_typ_kind: (e:env)(t:term)~(typ e (Srt kind) t).
Lemma inv_typ_prop: (e:env)(T:term)(typ e (Srt prop) T)->(conv T (Srt kind)).
Lemma inv_typ_ref: (P:Prop)(e:env)(T:term)(n:nat)(typ e (Ref n) T)
  ->((U:term)(item ? U e n)->(conv T (lift (S n) U))->P)
   ->P.
Lemma inv_typ_abs: (P:Prop)(e:env)(A,M,U:term)(typ e (Abs A M) U)->
```

```
          ((s1,s2:sort)(T:term)(typ e A (Srt s1))->(typ (cons ? A e) M T)
                  ->(typ (cons ? A e) T (Srt s2))->(conv (Prod A T) U)->P)
          ->P.
Lemma inv_typ_app: (P:Prop)(e:env)(u,v,T:term)(typ e (App u v) T)->
    ((V,Ur:term)(typ e u (Prod V Ur))->(typ e v V)
       ->(conv T (subst v Ur))->P)
     ->P.
Lemma inv_typ_prod: (P:Prop)(e:env)(T,U,s:term)(typ e (Prod T U) s)->
     ((s1,s2:sort)(typ e T (Srt s1))->(typ (cons ? T e) U (Srt s2))
                   ->(conv (Srt s2) s)->P)
       ->P.
```

Preuve Les six propositions se démontrent simultanément par récurrence sur la dérivation du jugement de typage.

Formellement, on utilise un prédicat d'inversion:

```
Definition inv_type: Prop->env->term->term->Prop :=
  [P:Prop][e:env][t:term][T:term]<Prop>Case t of
     [s:sort]<Prop>Case s of
        True
        (conv T (Srt kind))->P
        end
     [n:nat](x:term)(item ? x e n)->(conv T (lift (S n) x))->P
     [A,M:term](s1,s2:sort)(U:term)(typ e A (Srt s1))->(typ (cons ? A e) M U)
                ->(typ (cons ? A e) U (Srt s2))->(conv T (Prod A U))->P
     [u,v:term](Ur,V:term)(typ e v V)->(typ e u (Prod V Ur))
                ->(conv T (subst v Ur))->P
     [A,B:term](s1,s2:sort)(typ e A (Srt s1))->(typ (cons ? A e) B (Srt s2))
                ->(conv T (Srt s2))->P
     end.
```

dont on montre qu'il est invariant par β-conversion du deuxième terme par cas sur t (on utilise à chaque fois la transitivité de la β-conversion):

```
Lemma inv_type_conv: (P:Prop)(e:env)(t,U,V:term)(conv U V)
    ->(inv_type P e t U)->(inv_type P e t V).
```

Il est alors très facile de démontrer le théorème suivant par récurrence sur l'hypothèse $\Gamma \vdash t : T$:

```
Theorem typ_inversion: (P:Prop)(e:env)(t,T:term)(typ e t T)
                          ->(inv_type P e t T)->P.
```

Les six lemmes proposés sont des cas particuliers directs de ce dernier théorème. ∎

Remarque

Malgré les apparences, ces lemmes sont démontrés de la manière usuelle. La formulation n'est pas la même car on a recours

- au codage imprédicatif de l'existentielle car celui-ci s'inverse plus simplement qu'une cascade d'existentielle,
- à une définition d'une proposition par cas sur le terme plutôt qu'à la conjonction des six lemmes, encore pour des raisons de commodité.

Corollaire 2.2 *Aucun des termes convertibles avec* Kind *n'est typable.*

$$t \approx_\beta \mathrm{Kind} \Rightarrow \forall T, \neg(\Gamma \vdash t : T)$$

TRADUCTION EN Coq:

```
Lemma inv_typ_conv_kind: (e:env)(t,T:term)(conv t (Srt kind))->~(typ e t T).
```

PREUVE
On démontre tout d'abord:

```
Lemma typ_mem_kind: (e:env)(t,T:term)(mem_sort kind t)->~(typ e t T).
```

par récurrence sur t. Les termes atomiques ne contiennent Kind que s'ils lui sont égal, et donc non typable (`inv_typ_kind`). Pour les trois autres constructeurs, on utilise le fait que si une abstraction, une application ou un produit est typable, ses sous-terme le sont aussi (grâce aux lemmes d'inversion).

Il ne reste plus qu'à voir que tout terme convertible avec Kind le contient. Il suffit d'appliquer le théorème de Church-Rosser et le lemme sur la réduction des termes en forme normale pour déduire qu'il se réduit vers Kind. En appliquant le lemme `red_mem_sort`, on en déduit qu'il contient Kind.. ■

2.4.2 Lemme d'affaiblissement

Le lemme d'affaiblissement ("weakening" ou "thinning" suivant les auteurs) correspond à l'idée que la typabilité d'un terme n'est pas perdue si l'on rajoute une variable dans l'environnement. Avec les indices de de Bruijn, la formulation de ce lemme est un peu plus compliquée car le fait d'introduire une nouvelle variable bouleverse la numérotation même à l'intérieur de l'environnement.

Nous définissons l'opération qui consiste à insérer un terme dans un environnement tout en préservant la cohérence des indices de de Bruijn. Informellement, on sépare cette opération en deux: relocation des termes définis postérieurement, et insertion du terme dans la liste.

Définition 2.4 *La relocation des termes de l'environnement est définie par les deux règles de réécriture suivantes:*

$$\begin{array}{rcl} []^+ & = & [] \\ (\Delta; T)^+ & = & \Delta^+; \uparrow^1_{|\Delta|} T \end{array}$$

Définition 2.5 *La relation d'insertion d'un élément dans un environnement* `ins_in_env` *se définit par l'équation:*

$$(\texttt{ins\_in\_env}\ T\ \ |\Delta|\ \ (\Gamma;\Delta)\ \ \Gamma') \stackrel{def}{\Longleftrightarrow} \Gamma' = (\Gamma; T; \Delta^+)$$

TRADUCTION EN Coq:

```
Inductive ins_in_env [A:term]: nat->env->env->Prop :=
   ins_O: (e:env)(ins_in_env A O e (cons ? A e))
 | ins_S: (n:nat)(e,f:env)(t:term)(ins_in_env A n e f)
  ->(ins_in_env A (S n) (cons ? t e) (cons ? (lift_rec (S O) t n) f)).
```

Remarque

Formellement, on définit cette relation par un prédicat inductif, à la Prolog.

On montre l'effet de l'insertion d'un terme dans l'environnement:

Lemme 2.19

$$\begin{array}{rcl} (\Gamma; A; (\Delta^+))(k+1) & = & (\Gamma;\Delta)(k)\ si\ |\Delta| \leq k \\ \uparrow^{k+1}(\Gamma; A; (\Delta^+))(k) & = & \uparrow^1_{|\Delta|}\uparrow^{k+1}(\Gamma;\Delta)(k)\ \ si\ |\Delta| > k \end{array}$$

TRADUCTION EN Coq:

```
Lemma ins_item_ge: (A:term)(n:nat)(e,f:env)(ins_in_env A n e f)
         ->(v:nat)(le n v)
                    ->(t:term)(item ? t e v)->(item ? t f (S v)).
Lemma ins_item_lt: (A:term)(n:nat)(e,f:env)(ins_in_env A n e f)
         ->(v:nat)(gt n v)
          ->(t:term)(item_lift t e v)->(item_lift (lift_rec (S O) t n) f v).
```

Théorème 3 *La règle*

$$\frac{\Gamma;\Delta \vdash t : T \qquad \Gamma \vdash A : s}{\Gamma; A; \Delta^+ \vdash \uparrow^1_{|\Delta|} t : \uparrow^1_{|\Delta|} T}(s \in \text{SORT})$$

est admissible.

TRADUCTION EN Coq: (version affaiblie)

```
Theorem thinning: (e:env)(t,T:term)(typ e t T)
             ->(A:term)(wf (cons ? A e))
              ->(typ (cons ? A e) (lift (S O) t) (lift (S O) T)).
```

PREUVE

Pour démontrer ce théorème, on procède en trois étapes.

- Démontrer une forme apparemment affaiblie de ce théorème, en ajoutant une hypothèse supposant que $\Gamma; A; \Delta^+$ est bien formé:

```
Lemma typ_weak_weak: (g:env)(A:term)(s:sort)(typ g A (Srt s))
                      ->(e:env)(t,T:term)(typ e t T)
                       ->(n:nat)(trunc ? n e g)
                        ->(f:env)(ins_in_env A n e f)
                         ->(wf f)
                     ->(typ f (lift_rec (S O) t n) (lift_rec (S O) T n)).
```

  Par récurrence sur l'hypothèse $\Gamma \vdash t : T$. Le cas des variables utilise les lemmes du début de cette section. Pour le cas de l'application on utilise la propriété algébrique `distr_lift_subst`.

- En utilisant ce dernier lemme, on montre par récurrence sur Δ que l'hypothèse rajoutée est la conséquence des deux hypothèses originales:

$$\frac{\Gamma; \Delta \vdash \qquad \Gamma \vdash A : s}{\Gamma; A; \Delta^+ \vdash}(s \in \textsc{Sort})$$

- On conclut en composant les deux derniers théorèmes.

La forme affaiblie du théorème correspond au cas $\Delta = [\,]$. Il est donc très facile de prouver $\Gamma; A; \Delta \vdash$ juste après la première étape. ■

Le premier corollaire concerne l'utilisation répétée de `thinnning`.

```
Lemma thinning_n: (n:nat)(e,f:env)(trunc ? n e f)->(t,T:term)(typ f t T)
                  ->(wf e)->(typ e (lift n t) (lift n T)).
```

Le deuxième est la version relogée de `wf_sort` de la section 2.4.1.

```
Lemma wf_sort_lift: (n:nat)(e:env)(t:term)(wf e)->(item_lift t e n)
      ->(Ex [s:sort](typ e t (Srt s))).
```

2.4.3 Le lemme de substitution

L'idée du lemme de substitution est qu'un jugement de typage reste dérivable si l'on substitue une variable par un terme du type de la variable substituée. Ici encore, nous introduisons une opération qui substitue une variable dans les environnements.

Nous définissons informellement l'opération en deux fois: effectuer la substitution pour les termes définis postérieurement, et enlever cette variable de l'environnement, car la variable substituée disparaît.

Définition 2.6 *La substitution dans un environnement est définie informellement avec les deux règles de réécriture suivantes:*

$$\begin{array}{rcl} [\,][t] & = & [\,] \\ (\Delta; T)[t] & = & \Delta[t]; T[|\Delta| := t] \end{array}$$

Définition 2.7 *La relation de substitution dans l'environnement avec relocation est définie par la formule:*

$$(\texttt{sub\_in\_env}\ \ t\ \ T\ \ |\Delta|\ \ (\Gamma;T;\Delta)\ \ \Gamma') \stackrel{def}{\Longleftrightarrow} \Gamma' = (\Gamma;\Delta[t])$$

TRADUCTION EN Coq:

```
Inductive sub_in_env [t,T:term]: nat->env->env->Prop :=
   sub_O: (e:env)(sub_in_env t T O (cons ? T e) e)
 | sub_S: (e,f:env)(n:nat)(u:term)(sub_in_env t T n e f)
       ->(sub_in_env t T (S n) (cons ? u e) (cons ? (subst_rec t u n) f)).
```

Lemme 2.20

$$\begin{array}{rcl} (\Gamma;(\Delta[t]))(k) & = & (\Gamma;T;\Delta)(k+1)\ \ si\ |\Delta| \leq k \\ (\Gamma;T;\Delta)(|\Delta|) & = & T \\ \uparrow^{k+1}(\Gamma;(\Delta[t]))(k) & = & \uparrow^{k+1}(\Gamma;T;\Delta)(k)[|\Delta| := t]\ \ si\ |\Delta| > k \end{array}$$

TRADUCTION EN Coq:

```
Lemma nth_sub_sup: (t,T:term)(n:nat)(e,f:env)(sub_in_env t T n e f)
         ->(v:nat)(le n v)
                    ->(u:term)(item ? u e (S v))->(item ? u f v).
Lemma nth_sub_eq: (t,T:term)(n:nat)(e,f:env)(sub_in_env t T n e f)
                      ->(item ? T e n).
Lemma nth_sub_inf: (t,T:term)(n:nat)(e,f:env)(sub_in_env t T n e f)
         ->(v:nat)(gt n v)
          ->(u:term)(item_lift u e v)->(item_lift (subst_rec t u n) f v).
```

Théorème 4 *La règle*

$$\frac{\Gamma;t;\Delta \vdash u : U \qquad \Gamma \vdash d : t}{\Gamma;(\Delta[d]) \vdash u[|\Delta| := d] : U[|\Delta| := d]}$$

est admissible.

TRADUCTION EN Coq: (version affaiblie)

```
Theorem substitution: (e:env)(t,u,U:term)(typ (cons ? t e) u U)
            ->(d:term)(typ e d t)
              ->(typ e (subst d u) (subst d U)).
```

PREUVE

La démonstration du lemme de substitution se fait de manière analogue à celle du lemme d'affaiblissement. Le résultat apparemment affaibli à démontrer est:

```
Lemma typ_sub_weak: (g:env)(d:term)(t:term)(typ g d t)
                     ->(e:env)(u,U:term)(typ e u U)
                      ->(f:env)(n:nat)(sub_in_env d t n e f)
                       ->(wf f)
                        ->(trunc ? n f g)
                  ->(typ f (subst_rec d u n) (subst_rec d U n)).
```

On démontre par récurrence sur Δ que l'hypothèse supplémentaire est une conséquence des autres hypothèses.

Le théorème est la composition de ces deux derniers résultats. ■

2.4.4 Unicité du type modulo β-conversion

On établit ici que dans un environnement donné, tous les types d'un terme donné sont β-convertibles. Ceci est assez clair au vu des lemmes d'inversion.

Théorème 5 *Les types d'un même terme sont convertibles:*

$$\left.\begin{array}{l}\Gamma \vdash t : T \\ \Gamma \vdash t : U\end{array}\right\} \Rightarrow T \approx_\beta U$$

Traduction en Coq:

```
Theorem typ_unique: (e:env)(t,T:term)(typ e t T)->(U:term)(typ e t U)->(conv T U).
```

Preuve

La démonstration ne pose pas de problème. Elle se fait par récurrence sur la dérivation du jugement de typage. Il ne reste plus qu'à appliquer les lemmes d'inversion sur la deuxième hypothèse. On utilise aussi le fait que le terme qui figure en n-ième position dans une liste est unique. ■

2.4.5 Classification des termes

Nous pouvons maintenant prouver que les types habités sont bien formés (ou égaux à Kind).

Théorème 6 *Seuls les types bien formés et* Kind *sont habités: pour tout environnement* Γ *et tout terme* t, T*,*

$$\Gamma \vdash t : T \Rightarrow (\exists s \in Sort, \Gamma \vdash T : s) \vee (T = \mathrm{Kind})$$

Traduction en Coq:

```
Theorem type_case: (e:env)(t,T:term)(typ e t T)->
              (Ex [s:sort](typ e T (Srt s)))\/(T=(Srt kind)).
```

Preuve

Par récurrence sur le jugement $\Gamma \vdash t : T$:

Prop T=Kind est trivial.

Variable Appliquer le lemme `wf_sort_lift`.

Abstraction Il suffit d'appliquer la règle de typage du produit.

Application On utilise l'hypothèse de récurrence sur le sous-terme de gauche. Soit son type est typable par une sorte et le lemme de substitution permet de conclure, soit son type est Kind, ce qui est absurde car ce doit être un produit.

Produit On sait que le produit admet une sorte comme type. Soit cette sorte est Prop, qui admet la sorte Kind comme type, soit c'est Kind elle-même.

Conversion Pour pouvoir utiliser cette règle, il fallait justement que le type admette une sorte comme type.

■

Remarque La formulation de ce lemme met en évidence une classification des termes bien formés en trois niveaux:

- les termes qui ont pour type la sorte Kind: ils servent à former les types admissibles pour les prédicats, les propositions et les types de donnée; on les appelle "ordres".
- les termes dont le type a pour type la sorte Kind: ce sont les prédicats, les propositions et les types de données.
- les termes dont le type a pour type la sorte Prop: ce sont les preuves de propositions et les objets.

Cette classification sera étudiée plus en détail avant la preuve de normalisation forte. Dès à présent, on peut toutefois faire remarquer le rôle des sortes: ce sont des termes qui assurent la bonne formation d'une certaine classe de types.

On peut renforcer la conclusion du théorème d'unicité du typage dans le cas des ordres:

Corollaire 2.3 Kind *est le seul type des ordres.*

$$\left.\begin{array}{l}\Gamma \vdash t : T\\ \Gamma \vdash t : \mathrm{Kind}\end{array}\right\} \Rightarrow T = \mathrm{Kind}$$

Traduction en Coq:

```
Lemma type_kind_not_conv: (e:env)(t,T:term)(typ e t T)->(typ e t (Srt kind))
                          ->(T=(Srt kind)).
```

Preuve

On raisonne par cas sur le jugement $\Gamma \vdash t : T$ grâce au théorème précédent. Dans le premier cas, l'unicité du typage nous prouverait que T est typable, ce qui contredit le corollaire `inv_typ_conv_kind`, car T est convertible avec Kind par unicité du typage. Le deuxième cas correspond à la conclusion du corollaire. ■

2.4.6 Le théorème d'auto-réduction

Le théorème d'auto-réduction énonce la correction de la β-réduction par rapport au typage: un terme ne perd pas son type au cours du calcul, c'est-à-dire par β-réduction.

Nous généralisons la notion de β-réduction aux environnements:

Définition 2.8 *C'est la plus petite relation sur les environnements qui soit stable par application des règles suivantes:*

$$\frac{t \rhd_\beta u}{\Gamma; t \rhd_\beta \Gamma; u} \qquad \frac{\Gamma \rhd_\beta \Gamma'}{\Gamma; t \rhd_\beta \Gamma'; t}$$

TRADUCTION EN Coq:

```
Inductive red1_in_env: env->env->Prop :=
   red1_env_hd: (e:env)(t,u:term)(red1 t u)
                 ->(red1_in_env (cons ? t e) (cons ? u e))
 | red1_env_tl: (e,f:env)(t:term)(red1_in_env e f)
                 ->(red1_in_env (cons ? t e) (cons ? t f)).
```

Effet de la réduction sur les termes de l'environnement:

Lemme 2.21 *Pour tout environnement Γ, Γ', et pour tout entier n,*

$$\Gamma \rhd_\beta \Gamma' \Rightarrow \begin{cases} \Gamma(n) = \Gamma'(n) \\ \vee \\ \Gamma(n) \rhd_\beta \Gamma'(n) \wedge \forall k > n, \Gamma(k) = \Gamma'(k) \end{cases}$$

TRADUCTION EN Coq:

```
Lemma red_item: (n:nat)(t:term)(e:env)(item_lift t e n)
    ->(f:env)(red1_in_env e f)
     ->(item_lift t f n)
        \/( ((g:env)((trunc ? (S n) e g)->(trunc ? (S n) f g)))
            /\ (Ex2 [u:term](red1 t u) [u:term](item_lift u f n)) ).
```

PREUVE

Par récurrence sur n, sans difficulté particulière. ■

Théorème 7 *La règle*

$$\frac{\Gamma \vdash t : T \qquad \Gamma \rhd_\beta^* \Gamma' \qquad t \rhd_\beta^* u}{\Gamma' \vdash u : T}$$

est admissible.

TRADUCTION EN Coq: (version affaiblie)

```
Theorem subject_reduction: (e:env)(t,u:term)(red t u)
         ->(T:term)(typ e t T)->(typ e u T).
```

PREUVE

Ici encore, il y a trois étapes:

- démontrer le résultat apparemment affaibli

$$\Gamma \vdash t : T \Rightarrow \left\{ \begin{array}{l} \forall \Gamma', \Gamma \rhd_\beta \Gamma' \wedge \Gamma' \vdash \Rightarrow \Gamma' \vdash t : T \\ \forall u, t \rhd_\beta u \Rightarrow \Gamma \vdash u : T \end{array} \right.$$

```
Lemma subj_red: (e:env)(t,T:term)(typ e t T)->
                  ((f:env)(red1_in_env e f)->(wf f)->(typ f t T))
                     /\((u:term)(red1 t u)->(typ e u T)).
```

  par récurrence sur la dérivation de $\Gamma \vdash t : T$. La preuve est assez longue à cause des nombreux cas à traiter. Le cas de l'application utilise le lemme de substitution.

  A ce niveau, on peut en déduire la version affaiblie.

- prouver

$$\Gamma \vdash \wedge \Gamma \rhd_\beta \Gamma' \Rightarrow \Gamma' \vdash$$

  par récurrence sur Γ en utilisant le lemme qui vient d'être démontré.

- en combinant ces deux résultats, on déduit:

$$\Gamma \vdash t : T \Rightarrow \left\{ \begin{array}{l} \forall \Gamma', \Gamma \rhd_\beta \Gamma' \Rightarrow \Gamma' \vdash t : T \\ \forall u, t \rhd_\beta u \Rightarrow \Gamma \vdash u : T \end{array} \right.$$

- on prouve le résultat général par récurrence sur les deux hypothèses $t \rhd_\beta^* u$ et $\Gamma \rhd_\beta^* \Gamma'$.

■

Corollaire 2.4 *Deux termes typables β-convertibles ont des types β-convertibles:*

$$\left. \begin{array}{r} \Gamma \vdash u : U \\ \Gamma \vdash v : V \\ u \approx_\beta v \end{array} \right\} \Rightarrow U \approx_\beta V$$

TRADUCTION EN Coq:

```
Lemma typ_conv_conv:(e:env)(u,U,v,V:term)(typ e u U)->(typ e v V)->(conv u v)
                ->(conv U V).
```

PREUVE

De Church-Rosser, on déduit l'existence d'un terme t vers lequel se réduisent u et v. En appliquant le théorème d'auto-réduction deux fois, on prouve que t admet U et V comme types, et donc U et V sont β-convertibles. ■

2.5 Théorèmes de normalisation forte

Le résultat de normalisation forte du Calcul des Constructions est non calculatoire (puisque `sn` est un prédicat non calculatoire). Les preuves de normalisations fortes sont réputées difficiles, ce qui se confirme dans notre cas, puisque la preuve représente la moitié du développement. C'est pourquoi, afin d'aller à l'essentiel, celle-ci sera expliquée dans le chapitre 6. On se contentera ici de donner les deux résultats utilisés par la suite.

Théorème 8 *Tout terme bien typé est fortement normalisable:*

$$\Gamma \vdash t : T \Rightarrow t \in \mathcal{SN}$$

TRADUCTION EN Coq:

```
Theorem fort_norm: (e:env)(t,T:term)(typ e t T)->(sn t).
```

et le deuxième, que tout type (au sens: terme qui figure à droite d'un jugement de typage dérivable) l'est aussi:

Corollaire 2.5 *Tout type habité est fortement normalisable:*

$$\Gamma \vdash t : T \Rightarrow T \in \mathcal{SN}$$

TRADUCTION EN Coq:

```
Lemma type_sn: (e:env)(t,T:term)(typ e t T)->(sn T).
```

PREUVE

Ce dernier lemme est un corollaire immédiat du premier grâce au théorème `type_case`.
■

3 Résultats calculatoires

Nous démontrons les résultats calculatoires, c'est-à-dire ceux qui produiront effectivement le programme. Le but est de montrer le théorème principal: la décidabilité du typage. La première section rappelle les moyens de spécifier des programmes en Coq.

Comme nous avons défini un calcul avec types dépendants, et que nous avons la règle de conversion, il faut être capable de déterminer si deux types bien formés sont β-convertibles. Nous aurons besoin d'un algorithme de décision pour la β-conversion. Mais, dans le λ-calcul pur, la conversion de deux termes quelconques est un problème indécidable. Cependant, la β-conversion de deux termes fortement normalisables est décidable, grâce à un algorithme simple faisant appel à la normalisation. C'est pourquoi nous prouvons dans la section 3.2 l'algorithme de normalisation.

3.1 La spécification de programmes

Trois types de Coq sont dédiés à la spécification de programmes:

(sig T [x:T]P) noté $\{x : T|P\}$ spécifie les fonctions qui retournent un élément x de type T qui vérifie la propriété P. Ce type est défini comme l'existentielle, mais il est dans la sorte Set. Comme la logique de Coq est constructive, les objets de ce type se prouvent en donnant un élément de T et une preuve qu'il vérifie P. Ce programme s'extrait vers un objet de type T vérifiant P.

(sumor [x:T]P Q) noté $\{x : T|P\} + \{Q\}$ spécifie un objet de type (T `sumor`), qui est la somme disjointe de T et d'un type singleton. On spécifie soit un terme x de type T vérifiant la propriété P, soit `inright` signifiant que Q est vérifiée. Il sert typiquement dans les fonctions de recherche qui peuvent échouer (comme `list_item`).

(sumbool P Q) noté $\{P\} + \{Q\}$ spécifie `true` si P est vérifiée, et `false` si Q est vraie (si P et Q sont vraies, le résultat n'est pas spécifié et peut dépendre de l'implémentation). Pour prouver une telle spécification, il suffit de donner une preuve de P ou une preuve de Q. $\{P\} + \{\neg P\}$ sert à spécifier la décidabilité de la propriété P puisque la preuve comporte un algorithme qui décide P (sauf si on pose des axiomes calculatoires).

La preuve d'un lemme ayant un contenu calculatoire a la même structure que l'algorithme qui réalise ce lemme. Une conséquence est qu'il est possible de s'aider d'un programme pour démontrer des lemmes (voir [14]). Il existe une tactique qui permet d'associer un programme à un lemme; il est alors possible d'automatiser les grands traits de la preuve en suivant ce programme. Une fois que toutes les informations contenues dans le programme ont été utilisées, il ne reste plus qu'à prouver des buts non calculatoires, qui n'ont donc aucune influence sur l'algorithme. Ce moyen sera largement utilisé dans la suite pour deux raisons:

- la plupart des spécifications sont suffisamment simples pour qu'on imagine tout de suite un programme efficace qui la réalise. Cela nous permet de nous concentrer sur la justification de la correction du programme (i.e. les résultats non calculatoires qui restent),

- on est assuré que le programme que l'on extraira suivra l'algorithme donné: on a un meilleur contrôle de l'efficacité du programme.

Notre objectif principal est de spécifier un vérificateur de preuves: c'est un programme qui répond `true` si le jugement donné en argument est dérivable, et `false` s'il ne l'est pas; cela revient à prouver la décidabilité du typage:

```
(e:env)(t,T:term){(typ e t T)}+{~(typ e t T)}
```

Remarque
Si l'on ne voulait prouver que la correction de l'implémentation, on aurait pu spécifier cette fonction avec `(e:env)(t,T:term){(typ e t T)}+{True}`, qui permet d'être sûr qu'un jugement est dérivable quand le programme répond favorablement, mais qui peut rejeter des jugements corrects.

3.2 L'algorithme de normalisation

Nous devons spécifier la fonction de normalisation des termes et en prouver l'algorithme. Comme celui-ci se définit par récurrence, nous devons donner un ordre bien fondé correspondant aux appels récursifs de cette fonction, afin d'en prouver la terminaison. Après l'avoir défini, nous montrerons qu'il est bien fondé. L'algorithme sera prouvé correct dans la dernière partie de cette section.

3.2.1 L'ordre relatif à la normalisation

Il nous faut définir un ordre qui caractérise les appels récursifs de l'algorithme de normalisation, i.e. une relation R telle que $(t\ R\ u)$ soit vrai si la normalisation de u peut faire un appel récursif avec t.

Cet ordre doit contenir l'ordre de sous-terme direct. Par exemple, pour normaliser un produit, il suffit d'appeler récursivement la fonction de normalisation pour les deux sous-termes et retourner le produit de deux résultats, car le produit de deux termes normaux est normal.

Il doit aussi contenir l'ordre de β-expansion à cause du cas de l'application: si celle-ci correspond à un radical, il faut normaliser le terme obtenu par réduction de ce radical.

Nous prenons la réunion de ces deux ordres, ce qui permet de faire tous les appels récursifs souhaités. La contrepartie de ce gain est que la preuve de bonne fondation de cet ordre n'est pas si facile, la réunion de deux ordres bien fondés ne l'étant pas toujours. Heureusement, une propriété de commutation entre ces deux ordres nous assurera le résultat.

Définition 3.1 *On définit l'ordre $\prec$ de la façon suivante:*

$$x \prec y \overset{def}{\Longleftrightarrow} (x \subset_{st} y) \vee (y \rhd_{\beta} x)$$

On notera $\prec^+$ sa fermeture transitive, que l'on appellera ordre de normalisation.

TRADUCTION EN Coq:

```
Definition ord_norm1:= (union ? subterm (transp ? red1)).
Definition ord_norm:= (clos_trans ? ord_norm1).
```

Lemme 3.1 *L'ordre de normalisation contient le symétrique de $\rhd_\beta^+$ et la relation de sous-terme direct.*

TRADUCTION EN Coq:

```
Lemma subterm_ord_norm: (a,b:term)(subterm a b)->(ord_norm a b).
Lemma red_red1_ord_norm: (a,b:term)(red a b)->(c:term)(red1 b c)->(ord_norm c a).
```

PREUVE

Le premier lemme répète la définition de $\prec^+$. Le deuxième se démontre par récurrence sur la dérivation de $a \rhd_\beta^* b$. ■

3.2.2 Bonne fondation de l'ordre de normalisation

Théorème 9 *L'ordre de normalisation $\prec^+$ restreint aux termes fortement normalisables est bien fondé.*

TRADUCTION EN Coq:

```
Theorem wf_ord_norm: (t:term)(sn t)->(Acc ? ord_norm t).
```

PREUVE

On utilise un résultat classique sur les ordres bien fondés: la fermeture transitive d'une relation bien fondée est une relation bien fondée. Il ne reste plus qu'à montrer que $\prec$ restreint aux termes fortement normalisables est bien fondé:

```
Lemma wf_ord_norm1: (t:term)(sn t)->(Acc ? ord_norm1 t).
```

Or, $\prec$ est la réunion de deux ordres vérifiant la propriété `commut` (il s'agit du lemme `commut_red1_subterm`) qui assure la bonne fondation de la réunion à partir du moment où ils sont tous les deux bien fondés (lemme `Acc_union` de la théorie `RELATIONS/WELLFOUNDED` de Coq).

L'ordre de sous-terme direct est bien fondé (par récurrence sur la structure du terme):

```
Lemma wf_subterm: (well_founded ? subterm).
```

L'ordre de la β-expansion restreint à $\mathcal{SN}$ est bien fondé, ce qui est vrai par définition de l'ensemble des termes fortement normalisables. ■

La conséquence de ce résultat est que l'algorithme de normalisation termine pour tout terme fortement normalisable, ce qui n'est pas une grande surprise.

3.2.3 L'algorithme

Lemme 3.2 *Tout terme fortement normalisable admet une forme normale.*
TRADUCTION EN Coq:

```
Theorem compute_nf: (t:term)(sn t)->{u:term|(red t u)&(normal u)}.
```

PREUVE Pour faciliter la démonstration, nous utilisons l'algorithme suivant:

```
let compute_nf t =
  acc_rec (fun a norm → match a with
                         Srt s0 → Srt s0
                       | Ref n0 → Ref n0
                       | Abs(t0,t1) → Abs((norm t0),(norm t1))
                       | App(u0,v0) →
                           (match norm u0 with
                              Srt s → App((Srt s),(norm v0))
                            | Ref n → App((Ref n),(norm v0))
                            | Abs(a0,b) →
                                norm (subst_rec (norm v0) b O)
                            | App(a0,b) → App((App(a0,b)),(norm v0))
                            | Prod(a0,b) →
                                App((Prod(a0,b)),(norm v0)))
                       | Prod(t0,u0) → Prod((norm t0),(norm u0))) t
;;
```

Il s'agit ici d'une récurrence nœthérienne sur le terme à normaliser, l'ordre de normalisation garantissant la terminaison. Une fois que l'on a utilisé toutes les informations que l'on peut tirer de cet algorithme, il ne reste à prouver que des résultats non calculatoires simples:

- les appels récursifs se font bien selon l'ordre de normalisation,
- on appelle toujours la fonction avec des termes fortement normalisables,
- le résultat est un réduit de l'argument, et il est normal. Par exemple, pour le cas du produit, il faut vérifier que le produit de deux termes normaux est normal.

■

Remarque Dans cet algorithme, on ne fait pas les améliorations proposées par Huet dans [13] consistant à éviter de dupliquer les structures non modifiées. Cela nécessiterait l'intervention d'exceptions, qui ne sont pas utilisables dans des programmes purement fonctionnels.

3.3 L'algorithme de conversion

L'algorithme de conversion se déduit très rapidement du précédent: comme nous sommes dans un calcul ayant la propriété de Church-Rosser, l'algorithme le plus simple consiste à mettre les 2 termes en forme normale, puis comparer les 2 termes obtenus. En effet, la propriété de Church-Rosser a pour conséquence l'unicité des formes normales.

Lemme 3.3 *L'égalité des sortes et des termes est décidable.*

Traduction en Coq:

```
Lemma eqsort: (s,t:sort){s=t}+{~s=t}.
Lemma eqterm: (u,v:term){u=v}+{~u=v}.
```

Preuve

La preuve de ces lemmes est très simple, car nous avons affaire à des types inductifs, ce qui permet de faire l'équivalent Caml Light du filtrage. Il suffit de faire une récurrence sur le premier argument, et dans chacun des cas, faire une récurrence sur le deuxième argument. Enfin, les tactiques de discrimination entre les différents constructeurs de termes, permettent d'aboutir rapidement au résultat. ■

Lemme 3.4 *La relation de β-conversion restreinte aux termes fortement normalisable est décidable.*

Traduction en Coq:

```
Theorem is_conv: (u,v:term)(sn u)->(sn v)->{conv u v}+{~(conv u v)}.
```

Preuve

Ceci correspond au programme suivant:

```
let is_conv u v =
  match eqterm (compute_nf u) (compute_nf v) with
    Left → Left
  | Right → Right
;;
```

■

3.4 Les procédures de typage

Nous distinguons deux opérations de typage:

la vérification de type: décider si un jugement est dérivable ou non

l'inférence de type: étant donné un environnement et un terme, donner un type tel que le jugement formé soit dérivable, ou bien signaler qu'aucun jugement de cette sorte n'est dérivable.

Le résultat théorique le plus souvent énoncé est la décidabilité du typage. Cependant, pour pouvoir décider de la validité d'un jugement, il est nécessaire de pouvoir inférer les types, à cause du cas de l'application: étant donné le type du résultat, il n'est pas possible de deviner le type de l'argument.

Nous commençons par démontrer quelques fonctions annexes aux procédures de typage. Puis, la décidabilité de l'inférence et de la vérification de type dans un environnement bien formé sera prouvée. La décidabilité du jugement de typage sera alors très facile à montrer, mais nous verrons que ce dernier résultat n'a pas d'intérêt dans l'optique de réaliser un système de vérification de preuves efficace.

3.4.1 Test de réduction vers un produit

Dans la règle de typage de l'application, il faut pouvoir décider si le sous-terme de gauche a un type sous la forme d'un produit:

Lemme 3.5 *La réduction d'un terme fortement normalisable vers un produit est décidable.*

TRADUCTION EN Coq:

```
Lemma red_to_prod: (t:term)(sn t)->
                     {p:term*term|(u,v:term)(p=(u,v))->(red t (Prod u v))}
                    +{(u,v:term)~(conv t (Prod u v))}.
```

PREUVE

La démonstration suit l'algorithme:

```
let red_to_prod t =
  match compute_nf t with
    Srt q → Inright
  | Ref r → Inright
  | Abs(s,t0) → Inright
  | App(u,v) → Inright
  | Prod(u,v) → Inleft (Pair(u,v))
;;
```

qui met le terme sous forme normale, ce qui nécessite la précondition de normalisation forte, et discrimine suivant les constructeurs. Si la forme normale n'est pas un produit, il ne peut se réduire vers un produit à cause de la propriété de Church-Rosser et le fait qu'un produit se réduit toujours en un produit. ∎

Remarque Ce programme n'est vraiment pas optimisé: il n'est pas nécessaire de mettre le terme sous forme normale; il aurait suffi de mettre le terme sous forme normale de tête faible.

3.4.2 Test de réduction vers une sorte

Dans la plupart des règles, il faut que les types soient bien formés (i.e. typables par une sorte). On écrit donc une spécification qui décide si un type habité est convertible avec une sorte. Dans le cas de Kind, il faut même qu'il lui soit égal, car la règle de conversion ne s'applique pas à Kind (lemme `inv_typ_conv_kind`).

Lemme 3.6 *Pour tout jugement $\Gamma \vdash t : T$ avec T fortement normalisable, la propriété*

$$\exists s \in Sort, \Gamma \vdash t : s$$

est décidable.

TRADUCTION EN Coq:

```
Lemma red_to_sort: (T:term)(sn T)
        ->{(e:env)(t:term)(typ e t T)->(Ex [s:sort](typ e t (Srt s)))}
          +{(e:env)(t:term)(typ e t T)->(s:sort)~(typ e t (Srt s))}.
```

PREUVE

Il suffit de vérifier si T est Kind ou s'il est convertible avec Prop:

```
let red_to_sort t =
 match eqterm (Srt Kind) t with
   Left → Left
 | Right →
      (match is_conv (Srt Prop) t with
         Left → Left
       | Right → Right)
;;
```

■

Remarque La spécification en Coq n'est pas tout à fait celle énoncée par le lemme: on aurait dû spécifier une fonction prenant un environnement et deux termes retournant un booléen:

```
Lemma red_to_sort': (e:env)(t,T:term)(typ e t T)->(sn T)
      ->{(Ex [s:sort](typ e t (Srt s)))}+{(s:sort)~(typ e t (Srt s))}.
```

Mais comme le résultat ne dépend que du paramètre T, on distribue les arguments inutiles dans la propriété prouvée, ce qui cache ces deux arguments. Il faut cependant garder la précondition T fortement normalisable (qui est une conséquence de T est un type habité) pour pouvoir appeler l'algorithme de conversion.

3.4.3 L'inférence de type

Théorème 10 *L'inférence de type dans un environnement bien formé est décidable.*

Traduction en Coq:

```
Theorem infer: (e:env)(t:term)
                (wf e)->{T:term|(typ e t T)}+{(T:term)~(typ e t T)}.
```

Preuve

Comme cette fonction sera au cœur de toute implémentation, nous devons prendre grand soin dans l'algorithme employé. Le programme d'inférence de type qui suit est assez performant puisqu'il fait une simple récurrence structurelle sur le type à inférer:

```
let infer e t =
 let rec f = function
  Srt s →
    (fun f → match s with
             Kind → Inright
           | Prop → Inleft (Srt Kind))
 | Ref n →
    (fun e0 → match list_item e0 n with
               Inleft t → Inleft (lift_rec (S n) t O)
             | Inright → Inright)
 | Abs(t1,t2) →
    (fun e0 → match f t1 e0 with
               Inleft t →
                 (match red_to_sort t with
                   Left →
                     (match f t2 (Cons(t1,e0)) with
                       Inleft b →
                         (match eqterm (Srt Kind) b with
                           Left → Inright
                         | Right → Inleft (Prod(t1,b)))
                     | Inright → Inright)
                 | Right → Inright)
             | Inright → Inright)
 | App(t1,t2) →
    (fun e0 → match f t1 e0 with
               Inleft t →
                 (match red_to_prod t with
                   Inleft p →
                     (match p with
                       Pair(v,ur) →
                         (match f t2 e0 with
```

```
                                        Inleft b →
                                          (match is_conv v b with
                                             Left →
                                               Inleft (subst_rec t2 ur O)
                                            | Right → Inright)
                                        | Inright → Inright))
                                | Inright → Inright)
                            | Inright → Inright)
 | Prod(t1,t2) →
    (fun e0 → match f t1 e0 with
                            Inleft t →
                              (match red_to_sort t with
                                 Left →
                                   (match f t2 (Cons(t1,e0)) with
                                      Inleft b →
                                        (match eqterm (Srt Kind) b with
                                           Left → Inleft (Srt Kind)
                                         | Right →
                                             (match is_conv (Srt Prop) b with
                                                Left → Inleft (Srt Prop)
                                              | Right → Inright))
                                    | Inright → Inright)
                                | Right → Inright)
                            | Inright → Inright)
 in f t e
;;
```

Il faut produire un programme pour chacun des cinq constructeurs de termes:

Sortes L'inférence des sortes est très simple: soit c'est Prop et donc le type est Kind (l'environnement étant bien formé), soit c'est Kind, alors le terme n'est pas typable.

Variables Le cas des variables est lui aussi assez simple: il suffit de rechercher le n-ième terme dans l'environnement et le reloger. Si la recherche échoue, c'est que la variable n'a pas été déclarée, donc elle n'est pas typable.

Abstraction Le cas de l'abstraction est assez intéressant: on vérifie d'abord que le sous-terme de gauche est correct en inférant son type par appel récursif, et qu'il est bien formé grâce à `red_to_sort`. Si l'inférence échoue, alors le terme n'est pas typable par contraposée du lemme d'inversion de l'abstraction. Pour le sous-terme de droite, on utilise le résultat `type_case`. Il suffit de tester si le type inféré n'est pas Kind, ce qui assure alors qu'il est bien formé.

Application On commence par typer le terme de gauche. Celui-ci doit se réduire vers un produit. S'il ne l'est pas, l'application n'est pas typable. Ensuite, on infère le type du terme de droite, qui doit être convertible avec le sous-terme gauche du produit. Le type de l'application est alors le sous-terme droit du produit dans lequel on a substitué la variable 0 par l'argument.

Produit On vérifie que le sous-terme de gauche est bien formé comme pour l'abstraction (appel récursif puis `red_to_sort`). Pour le sous-terme droit, il ne suffit pas d'utiliser la même procédure et retourner le résultat de l'appel récursif, car celui-ci peut ne pas être en forme normale, et s'il contient des variables libres, il a toutes les chances de ne plus être bien formé dans l'environnement Γ (il l'est dans $\Gamma; t_1$). Il faut tester s'il est égal à Kind ou convertible avec Prop, et retourner la sorte correspondante. Ce "détail" illustre bien la nécessité de la formalisation.

■

3.4.4 La vérification du type

Théorème 11 *La dérivabilité d'un jugement de typage dont l'environnement est bien formé est décidable.*

Traduction en Coq:

```
Lemma check_typ: (e:env)(t,tp:term)(wf e)->{(typ e t tp)}+{~(typ e t tp)}.
```

Preuve

```
let check_typ e t tp =
 match infer e t with
   Inleft tp' →
      (match eqterm (Srt Kind) tp' with
         Left →
            (match eqterm (Srt Kind) tp with
               Left → Left
             | Right → Right)
       | Right →
            (match infer e tp with
               Inleft s →
                  (match is_conv tp tp' with
                     Left → Left
                   | Right → Right)
             | Inright → Right))
 | Inright → Right
;;
```

On commence par inférer le type de t. Si t n'est pas typable, alors la vérification échoue. Sinon, il se présente deux cas, correspondant à la disjonction de `type_case`:

- t de type Kind: il faut que tp soit lui aussi Kind (lemme `inv_typ_conv_kind`).
- t de type tp': il faut que
  - tp soit typable,
  - tp et tp' soient convertibles.

  La première condition sert uniquement à s'assurer que le terme est fortement normalisable (via le théorème de normalisation forte), car la conversion n'est assurée de terminer que pour les termes fortement normalisable.

■

Théorème 12 *Pour tout environnement* Γ *bien formé, et tout terme* t*, la validité de l'environnement* $\Gamma; t$ *est décidable.*

TRADUCTION EN Coq:

```
Lemma add_typ: (e:env)(t:term)(wf e)
                    ->{(wf (cons ? t e))}+{~(wf (cons ? t e))}.
```

PREUVE

Il n'y a qu'à voir si t est typable par une sorte. On utilise le programme:

```
let add_typ e t =
  match infer e t with
    Inleft tp →
      (match red_to_sort tp with
         Left → Left
       | Right → Right)
  | Inright → Right
;;
```

On infère le type de t, et on appelle `red_to_sort`. ■

3.4.5 La décidabilité des jugements

Théorème 13 *Le jugement de validité de l'environnement est décidable.*

TRADUCTION EN Coq:

```
Lemma decide_wf: (e:env){(wf e)}+{~(wf e)}.
```

PREUVE

Par récurrence sur l'environnement:

Nil: L'environnement vide est bien formé.

Cons t l: Par appel récursif sur l, on sait s'il est bien formé: s'il ne l'est pas, alors (Cons t l) ne peut pas l'être; s'il l'est, il suffit d'utiliser `add_typ` pour décider de la validité de l'environnement (Cons t l).

■

Théorème 14 *Le jugement de typage du Calcul des Constructions est décidable.*
TRADUCTION EN Coq:

```
Lemma decide_typ: (e:env)(t,T:term){(typ e t T)}+{~(typ e t T)}.
```

PREUVE
Il suffit de combiner `check_typ` avec `decide_wf`. On commence par décider si l'environnement est bien formé: s'il ne l'est pas, le jugement n'est pas dérivable par contraposée du lemme `typwf`; s'il l'est on peut utiliser `check_typ` puisque la précondition est remplie. ■

Remarque
Les deux dernières spécifications démontrées ne sont que d'un intérêt théorique. Elles énoncent la décidabilité des deux jugements mutuellement dépendants `wf` et `typ`.

Elles n'ont aucun intérêt en pratique puisque les préconditions de `infer` et `add_typ` sont des invariants dans pratiquement toute implémentation rationnelle d'un système logique. Il serait particulièrement inefficace de revérifier la validité du contexte à chaque fois que l'on veut typer un terme.

4 Extraction

4.1 Procédure d'extraction vers Caml Light

Pour une présentation plus détaillée, nous conseillons la lecture de [16], où Paulin-Mohring et Werner expliquent avec un exemple les principes de l'extraction.

L'extraction consiste à interpréter un jugement dérivable par un programme exécutable. Cette interprétation se fait en trois étapes:

- enlever tous les termes non calculatoires des preuves. Le système de types est conçu de façon à préserver la validité d'un jugement de typage par cette opération: il est ainsi interdit de prouver un résultat calculatoire par récurrence sur une hypothèse logique.
- éliminer les types dépendants: cela ne change pas le comportement calculatoire. On obtient un jugement qui est dérivable dans le système $\mathcal{F}\omega^{idt}$ (système $\mathcal{F}\omega$ muni de types inductifs).
- l'interprétation proprement dite. Celle-ci se fait assez facilement du fait de la similitude du système $\mathcal{F}\omega^{idt}$ et des langages de programmations comme ML ou Caml Light. Les types inductifs s'implémentent en des types concrets de Caml Light, et le Case de Coq s'interprète par le filtrage. Il faut se donner l'interprétation des variables libres, i.e. les axiomes. Il existe une commande pour donner un programme à chaque axiome.

La commande **Write Caml File** *fichier* $[t_1 \ldots t_n]$**.** sert à produire un fichier de code Caml Light contenant les programmes extraits des termes $t_1 \ldots t_n$, ainsi que tous les termes de l'environnement dont ils dépendent. Ce fichier peut être compilé, puis inclus dans d'autres modules (voir section 4.3 pour un exemple de développement dont la base est le code extrait).

On extrait les fonctions du chapitre 3 qui nous intéressent: `infer`, `add_typ` et `check_typ` (le résultat de cette extraction se trouve en annexe B). Ils correspondent aux opérations élémentaires d'un vérificateur de preuves.

On n'extrait pas `decide_wf` ou `decide_typ` car elles ne sont pas utilisables: la validité de l'environnement est un invariant du système de preuve que l'on veut réaliser. On préfère vérifier au fur et à mesure que l'on insère des termes que l'environnement reste valide.

En théorie, le code produit n'a pas à être relu, puisqu'on est assuré qu'il obéit à sa spécification. Mais, il peut être intéressant de l'analyser tout de même, pour se rendre compte des améliorations à apporter à l'extraction, en termes d'efficacité des programmes.

4.2 Optimisation du code extrait

4.2.1 Améliorations facilement automatisables

En lisant le code on voit tout de suite des améliorations simples, qui pourraient être mises en œuvre dans les toutes prochaines versions de Coq. Elles concernent principalement la façon dont sont interprétés les types inductifs: ceux-ci sont systématiquement traduits en des types concrets, ce qui fait que les types prédéfinis de Caml Light sont dupliqués. On perd en même temps toutes les optimisations dont peuvent bénéficier ces types.

Il est très facile de modifier l'implémentation d'un type inductif: il suffit de donner:

- le type de Caml Light lui correspondant,
- l'équivalent des constructeurs,
- un destructeur correspondant au Case,
- et eventuellement, les fonctions optimisées dans le langage cible peuvent se substituer à l'interprétation d'une définition maladroite en Coq.

Nous proposons maintenant quelques améliorations qui rendraient le code un peu plus lisible, mais surtout plus efficace. Les faire à la main n'est pas une bonne solution à long terme, puisqu'à chaque extraction, elles seraient à refaire; cependant, cela permet de se rendre compte des points à améliorer dans l'extraction.

- Implémenter le type inductif `sumbool` par le type `bool`. Cela permet aussi d'utiliser le `if/then/else` qui est très intuitif quand on décide d'une propriété. Les traductions à effectuer sont les suivantes:

$$
\begin{array}{rcl}
\texttt{inleft} & \mapsto & \texttt{true} \\
\texttt{inright} & \mapsto & \texttt{false} \\
\begin{array}{l} <T> \texttt{Case } e_1 \texttt{ of} \\ \quad \texttt{inleft} -> \ e_2 \\ \quad \texttt{inright} -> \ e_3 \\ \texttt{end} \end{array} & \mapsto & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3
\end{array}
$$

- Remplacer les fonctions qui font un filtrage, puis reconstruisent leur argument par la fonction identité. Par exemple:

$$
\begin{array}{l} \texttt{if } x \ \ \texttt{then true} \\ \qquad\ \ \texttt{else false} \end{array} \ \mapsto \ x
$$

- Implémenter les `nat` par les `int` de Caml Light. On fait ici une économie de place car on représente les entiers en base 2 au lieu de les représenter en notation unaire. Mais on prend le risque d'introduire des erreurs de calcul en cas de débordement non prévus (les `int` de Caml Light sont limités à $\pm 2^{30}$). Une meilleure solution semble être l'utilisation des `bignum` qui cumulent les deux avantages: ils sont compacts et ne risquent pas d'introduire d'erreur de calcul.

$$
\begin{array}{rcl}
\texttt{O} & \mapsto & 0 \\
\texttt{S} & \mapsto & succ \\
<T>\texttt{Case}\ e_1\ \texttt{of}\ e_2\ e_3\ \texttt{end} & \mapsto & \begin{array}{l}(\texttt{match}\ e_1\ \texttt{with} \\ \quad \texttt{0} -> \ e_2 \\ \quad |\ \texttt{n} -> \ (e_3\ (\texttt{pred n}))) \end{array}
\end{array}
$$

La première simplification décrite nous permet d'aller plus loin: les fonctions comme `eq_nat_dec` ou `le_gt_dec` peuvent s'implémenter en `eq_int` et `le_int`. Mais il faut se méfier de `pred`: en Coq, le prédécesseur de `O` est `O`; pas en Caml Light.

- Remplacer `eqsort` et `eqterm` par l'égalité polymorphe de Caml Light.

4.2.2 Autres améliorations envisageables

Simplifier les η-redexes Ceci ne devrait pas poser de problème pour des programmes fonctionnels. Mais pour l'extraction, il est spécifié que les axiomes calculatoires non réalisés (c'est-à-dire ceux pour lesquels on n'a pas donné de programme correspondant) sont convertis en exceptions, qui n'est pas une construction fonctionnelle de Caml Light. Il faut donc faire attention aux simplifications, certaines η-réductions peuvent précipiter l'évaluation d'arguments, ce qui pourrait déclencher des exceptions.

Remplacer `sumor` par des exceptions La plupart du temps, une spécification utilisant `sumor` correspond à une fonction que l'on aurait codé avec une exception. Cette amélioration n'est pas simple à mettre en œuvre.

Améliorations algorithmiques Les algorithmes employés sont très peu optimisés. Nous proposons ici des améliorations que l'on pourrait faire sans grande difficulté:

- A chaque fois que l'on doit réduire un terme pour mieux connaître sa structure, on le normalise, ce qui n'est pas nécessaire en général: souvent, il suffirait de mettre en forme normale de tête faible.

  Ceci est flagrant pour l'algorithme de réduction vers un produit. On peut chercher à améliorer le test de conversion en testant l'égalité structurelle au lieu de réduire systématiquement.

- La relocation et la substitution sont implémentés par leur définition métathéorique. Or, celles-ci ont été données avec le souci de paraître intuitives, et non pas efficaces. Pour pouvoir proposer un autre algorithme, il suffirait d'introduire deux fonctions dont la spécification correspondrait exactement à la notion métathéorique:

```
Lemma lift_implemente: (n:nat)(t:term){u:term|u=(lift n t)}.
Lemma subst_implemente: (N,M:term){u:term|u=(subst N M)}.
```

4.3 Réalisation d'un vérificateur de preuves (Coc)

4.3.1 Utilisation du code extrait

Le code extrait peut servir de base à la réalisation d'un noyau assez simple. Il faut programmer une interface utilisateur à la main. Il serait bon de la vérifier. Pour faciliter ce travail, on a programmé cette interface de façon modulaire: on peut distinguer (au moins) trois niveaux de représentation des termes:

1. syntaxe concrète (sous forme de chaîne de caractères)
2. arbre de syntaxe abstraite (terme en variables nommées)
3. terme en indices de de Bruijn

Le premier est le plus concret, et le dernier est le seul sur lequel nous avons des résultats vérifiés formellement.

Pour produire une interface, il suffit de produire quatre sortes de fonctions:

- un analyseur syntaxique qui permet de lire des commandes dans la syntaxe habituelle du λ-calcul.($1 \Rightarrow 2$)
- une fonction d'affichage des termes.($2 \Rightarrow 1$)
- un convertisseur de termes en notation de de Bruijn en variables nommées. ($2 \Leftrightarrow 3$)
- une boucle permettant d'entrer des commandes au clavier, de les exécuter, puis d'afficher leur résultat, pour lier le tout.

Le passage de la deuxième à la troisième représentation est assuré par le convertisseur variables nommées-indices de de Bruijn; il ne devrait pas poser de problème particulier à formaliser, une fois qu'il est convenu de la façon de faire la conversion.

Le passage de la première à la deuxième représentation est plus compliqué. Il pourrait être intéressant de prouver que la composée de l'affichage et de l'analyse syntaxique est l'identité, mais cela n'assure pas que ces deux fonctions sont correctes. Un analyseur syntaxique est très difficile à spécifier car ce qu'on lui demande, c'est de reconnaître le terme auquel on avait pensé en entrant la ligne. Cela donne lieu en général à des spécifications aussi longues que les programmes. Il est alors plus simple de donner le programme comme spécification.

Spécifier la boucle de *toplevel* n'est pas évident car l'idée de boucle est typiquement impérative. On préfèrera présenter le programme sous la forme d'une machine abstraite, mieux adaptée aux programmes impératifs.

4.3.2 La machine abstraite de Coc

Etat Il y a deux deux variables globales modifiables: ENV et CTX.

- ENV contient les types des variables libres des termes.
- CTX contient les noms (pour l'affichage) des variables libres. Cette liste doit être de même longueur que ENV.

term	:=	*sort*	(sorte)
	|	*ident*	(variables)
	|	$[ident\{,ident\}^* : term]term$	(abstraction)
	|	$(term^+)$	(application)
	|	let *ident* : *term* := *term* in *term*	(radical)
	|	$(ident\{,ident\}^* : term)term$	(produit)
	|	*term*`->`*term*	(produit non dépendant)
sort	:=	Kind	
	|	Prop	

Levée des ambiguïtés:
- `->` est associatif à droite
- `->` est prioritaire sur [ ] et ( )

Figure 7 : Grammaire des termes Coc

Invariants du système

- ENV est bien formé,
- ENV et CTX sont de même longueur,
- CTX ne contient pas deux éléments identiques.

Les deux dernières conditions ne servent qu'à assurer l'existence et l'unicité de la représentation des termes de de Bruijn. Afin d'éviter toute ambiguïté au moment de l'affichage, il n'est pas admis de déclarer un axiome sous un nom déjà employé. En effet, en variables nommées, une variable redéfinie provoque la capture de cette variable; en de Bruijn, la variable cachée reste accessible, mais serait représentée de la même façon.

Opérations Cette machine comporte six commandes, qu'une boucle intéractive permet de mettre en œuvre. Nous décrivons ici la syntaxe et l'effet de ces commandes (voir grammaire des termes, figure 7):

`Infer` *term*`.` infère le type de l'argument et l'affiche ou répond `mal type` si le terme n'est pas typable. On fait directement appel à la fonction `infer` extraite; il n'y a qu'à interpréter le résultat: soit c'est `(inleft` T`)`, soit c'est `inright`.

`Check` *term*`:`*term*`.` vérifie si le jugement donné en argument est dérivable dans l'environnement courant. On utilise `check_typ` qui retourne justement un booléen. Il suffit d'afficher `Correct` ou `Echec` suivant le résultat.

`Axiom` *ident*`:`*term*`.` permet d'ajouter un terme dans l'environnement en vérifiant que celui-ci reste bien formé. C'est ce que vérifie `add_typ`. On donne un nom (*ident*) à cet axiome pour le réutiliser sans avoir à connaître son indice. Il faut vérifier que cet identificateur n'est pas déjà utilisé, sinon on risque de briser le troisième invariant.

`Delete.` enlève le dernier terme entré dans l'environnement. Il n'y a rien à vérifier puisque l'environnement reste bien formé. Un message indique si l'on a tenté de supprimer un axiome dans l'environnement vide.

`List.` affiche l'ensemble des noms d'axiomes définis.

`Quit.` termine la session.

Les trois dernières commandes n'appellent pas de commentaire: elle ne sont là que pour rendre possible l'utilisation du système en mode interactif.

Il faut attirer l'attention sur le fait que les trois premières collent parfaitement aux spécifications des trois fonctions extraites. On leur donne juste un goût impératif car cela correspond mieux à l'apparence interactive du système.

4.3.3 Résultat

On peut considérer que Coc implémente un vérificateur de preuves utilisant un langage mathématique élémentaire, rendant impossible son utilisation en tant qu'aide au développement de preuves.

Il faut pour cela des utilitaires permettant de traduire des commandes d'un langage mathématique de haut niveau (comme par exemple Gallina). Nous allons voir que l'on peut déjà utiliser le système Coq comme un compilateur de Gallina vers Coc.

4.4 Exemple de développement en Coc

Nous allons maintenant simuler la procédure décrite par Boyer et Dowek dans [4] où l'on re-vérifie les preuves engendrées par un système complexe à l'aide d'un vérificateur très rudimentaire.

Enoncé du lemme de Newman: Si R est une relation nœtherienne et localement confluente, alors R est confluente.

Il a été possible de traduire la démonstration du lemme de Newman donnée par Huet dans [13] dans cette syntaxe avec peu d'efforts. Nous disposons de la preuve de ce lemme sous forme de tactiques. Comme notre noyau ne comprend que les termes-preuve, il faut compiler le langage Gallina vers celui de Coc. L'utilitaire le mieux adapté est évidemment Coq lui-même.

Il y a tout de même quelques petites modifications à apporter, car Coc ne comprend ni les définitions, ni les lemmes.

4.4.1 Codage des lemmes

Les lemmes sont très simple à coder: il suffit de rajouter dans la preuve principale un radical qui introduit une variable dont le type est l'énoncé du lemme. La syntaxe des termes Coc comprend une construction `let/in` qui permet de réaliser ceci facilement.

4.4.2 Codage des constantes

Le moyen le plus simple pour éviter de manipuler les constantes est de remplacer partout les constantes par leur définition (c'est la δ-réduction). Cette méthode est un peu lourde car le terme à vérifier peut devenir très gros, et l'on perd beaucoup en efficacité.

On présente ici un moyen de transformer une définition en deux axiomes: lorsqu'en Coq, on déclare la définition de la constante x par le terme t de type T:

```
Definition x: T := t.
```

on écrit en Coc:

```
Axiom x: T.
Axiom unfold_x: (P:T->Prop)(P t)->(P x).
```

Remarque Le dernier axiome est le codage imprédicatif de l'égalité de la variable x avec sa définition t. Lorsqu'on applique cet axiome, on remplace x par sa valeur t.

Ce codage n'est pas parfait puisqu'il demande de reprendre la preuve: à chaque fois que l'on utilise la δ-réduction, il faut appliquer le lemme `unfold_x` à la place.

Cette astuce admet une justification métathéorique grâce au lemme de substitution. On transforme un jugement comportant les deux axiomes en un jugement sans ces deux axiomes de la façon suivante:

- On substitue la variable `x` par le terme `t`. Le lemme de substitution nous demande de vérifier que `t` a pour type le terme figurant dans l'environnement (i.e. `T`): cela est vrai si la définition à coder est correcte.
- On substitue l'axiome `unfold_x` par le terme: $\lambda(T \to \mathrm{Prop}).\lambda(0 \;\; \uparrow^1 t).0$ qui est à peu près l'identité polymorphe.

L'effet de ces deux opérations est de remplacer partout la variable x par sa définition, tout en gardant un jugement dérivable.

De cette manière, on dispose d'un codage qui ne remet pas en cause l'efficacité, admettant une justification que cela correspond bien à l'idée de définition.

4.4.3 Re-vérification

Avec les codages précédents, il est possible de traduire des preuves faites en Coq, en preuve acceptable par Coc. Il est souhaitable à ce niveau de bien vérifier que le texte obtenu correspond toujours à ce que l'on veut vérifier.

Il ne reste plus qu'à donner ce texte à Coc, qui validera (ou infirmera) la preuve.

4.4.4 Performances

La vérification du lemme de Newman (fichier de $6,8$ Ko) par Coc prend $0,15s$ utilisateur sur un Pentium Pro 150 (une preuve équivalente, prend environ $2s$ en Coq, mais il ne faudrait pas tenir compte des accès à l'environnement). En fait, on passe plus de la moitié du temps à faire l'analyse syntaxique du fichier.

Si le noyau était mieux intégré dans le système, les performances seraient encore meilleures puisqu'on pourrait passer directement les termes sans passer par la syntaxe concrète. Il faut tout de même toujours pouvoir s'assurer que ce que le noyau vérifie effectivement ce que l'on croit.

Il est rassurant de constater que l'on peut certifier des programmes suffisamment efficaces pour être utilisables.

5 Classifications liées au typage

Dans les deux derniers chapitres, nous prouvons la normalisation forte du Calcul des Constructions. Nous ne détaillerons plus les résultats donnés; d'une part ce travail sort un peu du cadre du sujet, mais il est important car c'est la première fois qu'un tel résultat est formalisé (à notre connaissance). D'autre part, la preuve ne colle plus exactement à la démonstration informelle (contrairement à tout ce que nous avons fait jusqu'ici), même si l'idée s'adapte assez bien.

Cette preuve met en évidence les différences entre la Théorie des Types et la Théorie des Ensembles en ce qui concerne les types dépendants et l'égalité des fonctions (intentionnelle ou extensionnelle).

5.1 Stratification des termes

5.1.1 Définitions

On avait déjà vu avec le lemme `type_case` que les termes typables pouvaient être répartis en trois catégories. Nous allons énoncer des propriétés plus précises à ce sujet. Notamment, on voit que `type_case` a un équivalent calculatoire, et que ces classes sont distinctes.

On commence par définir l'ensemble des classes: termes, types ou ordres. Cela correspond à l'autre façon d'introduire les notation des termes avec trois classes syntaxiques:

$$
\begin{array}{rcl}
K & := & \mathrm{Prop}|\Pi T.K|\Pi K.K' \\
T & := & i \in \mathbb{N}|\Pi T.T'|\Pi K.T|\lambda T.T'|(T\ t)|\lambda K.T|(T\ T') \\
t & := & i \in \mathbb{N}|\lambda T.t|(t\ t')|\lambda K.t|(t\ T)
\end{array}
$$

On pourra remarquer que le terme Kind ne rentre dans aucune de ces classes. Cela tient au fait que Kind ne peut figurer dans un jugement dérivable que dans le terme de droite, et tout seul. La syntaxe alternative remplace les jugements $\Gamma \vdash K$: Kind par un autre type de jugement: $\Gamma \vdash K$. Le principal avantage est de mieux marquer la différence entre les différentes classes de jugements. Mais l'inconvénient est que les règles de typage se multiplient.

Définition 5.1 *On définit l'ensemble des classes: il y en a trois (termes, types et ordres).*

$$Class := Trm|Typ|Ord$$

On parlera aussi de listes de classes, correspondant aux environnements.

Traduction en Coq:

```
Inductive Set class:= Trm:class | Typ: class | Ord: class.
Definition cls:= (list class).

Syntactic Definition nthf:= (nth_def class Trm).
```

Définition 5.2 *On ordonne les classes avec l'ordre $\sqsubset$ défini par:*

$$t \sqsubset T \sqsubset K$$

TRADUCTION EN Coq:

```
Inductive lift_cls: class->class->Prop :=
   lift_t: (lift_cls Trm Typ)
 | lift_T: (lift_cls Typ Ord).
```

Nous démontrerons plus tard que cet ordre traduit la position relative des termes dans le jugement de typage: la classe d'un terme est inférieure à la classe de son type.

Définition 5.3 *La fonction calculant la classe à laquelle appartient un terme:*

$$\begin{array}{rcl} Cl_\Gamma(s) & = & Ord \\ Cl_\Gamma(n) & = & c \text{ si } c \sqsubset Cl_\Gamma(\Gamma(n)) \\ Cl_\Gamma(\lambda A.B) & = & Cl_{\Gamma;A}(B) \\ Cl_\Gamma((u\ v)) & = & Cl_\Gamma(u) \\ Cl_\Gamma(\Pi T.U) & = & Cl_{\Gamma;T}(U) \end{array}$$

TRADUCTION EN Coq:

```
Fixpoint cl_term [t:term]: cls->class :=
  [i:cls]Cases t of
      (Srt _) => Ord
    | (Ref n) => Cases (nthf i n)  of
                   Ord => Typ
                 | _   => Trm
                 end
    | (Abs A B)  => (cl_term B (cons ? (cl_term A i) i))
    | (App u v)  => (cl_term u i)
    | (Prod T U) => (cl_term U (cons ? (cl_term T i) i))
    end.

Fixpoint class_env [e:env]: cls :=
  Cases e of
    nil => (nil ?)
  | (cons t f) => (cons ? (cl_term t (class_env f)) (class_env f))
  end.

Definition cl_judge:= [e:env][t:term](cl_term t (class_env e)).
```

Remarque

En Coq, cette définition doit se faire en trois temps, et le résultat de cette fonction ne veut rien dire si le terme donné en argument est Kind ou s'il est mal formé.

5.1.2 Propriétés d'invariance

Lemme 5.1 *La classe est préservée par relocation:*

$$Cl_{\Gamma;\Delta}(t) = Cl_{\Gamma;x;\Delta^+}(\uparrow^1_{|\Delta|} t)$$

Traduction en Coq:

```
Lemma cl_term_lift: (x:class)(t:term)(k:nat)(f,g:cls)(insert ? x k f g)
                  ->(cl_term t f)=(cl_term (lift_rec (S O) t k) g).

Lemma class_env_lift: (e,f:env)(k:nat)(x:term)(ins_in_env x k e f)
       ->(insert ? (cl_judge e (lift k x)) k (class_env e) (class_env f)).

Lemma cl_judge_lift: (e,f:env)(k:nat)(x,t:term)(ins_in_env x k e f)
         ->(cl_judge e t)=(cl_judge f (lift_rec (S O) t k)).
```

Remarque Ce lemme correspond au lemme d'affaiblissement.

Lemme 5.2 *La classe est préservée par substitution par un terme dont la classe est inférieure au type de la variable substituée:*

$$Cl_\Gamma(v) \sqsubset Cl_\Gamma(V) \Rightarrow Cl_{\Gamma;V;\Delta}(t) = Cl_{\Gamma;(\Delta[v])}(t[|\Delta| := v])$$

Traduction en Coq:

```
Lemma cl_term_subst: (a:class)(G:cls)(x:term)(lift_cls (cl_term x G) a)
               ->(t:term)(k:nat)(E,F:cls)(insert ? a k E F)
                ->(trunc ? k E G)
         ->(cl_term t F)=(cl_term (subst_rec x t k) E).

Lemma sub_env_ins:
  (g:env)(v,V:term)(lift_cls (cl_judge g v) (cl_judge g V))
       ->(k:nat)(e,f:env)(sub_in_env v V k e f)
        ->(trunc ? k f g)
      ->(insert ? (cl_judge g V) k (class_env f) (class_env e)).

Lemma cl_judge_subst:
    (g:env)(v,V:term)(lift_cls (cl_judge g v) (cl_judge g V))
     ->(k:nat)(e,f:env)(sub_in_env v V k e f)
      ->(trunc ? k f g)
    ->(t:term)(cl_judge e t)=(cl_judge f (subst_rec v t k)).
```

Remarque Ce lemme est équivalent au lemme de substitution. Il y a une condition supplémentaire: il faut que le terme par lequel on substitue et celui qui est ôté de l'environnement soient dans deux classes consécutives.

Une propriété du même style pourrait être donnée ici en ce qui concerne la β-réduction, mais comme pour la substitution, une autre condition apparaît: il faut que pour tout radical, l'argument et l'annotation de type dans l'abstraction appartiennent à des classes consécutives.

Une condition suffisante est que le terme soit typable. C'est pourquoi on commence par démontrer le lien entre les cas de `type_case` et la classe.

5.1.3 Lien entre classe et typage

Théorème 15 *Pour tout jugement dérivable* $\Gamma \vdash t : T$,

$$\begin{array}{rcl} \Gamma \vdash T : \mathrm{Prop} & \Leftrightarrow & Cl_\Gamma(t) = Trm \\ \Gamma \vdash T : \mathrm{Kind} & \Leftrightarrow & Cl_\Gamma(t) = Typ \\ T = \mathrm{Kind} & \Leftrightarrow & Cl_\Gamma(t) = Ord \end{array}$$

Traduction en Coq:

```
Lemma class_trm: (e:env)(t,T:term)(typ e t T)->(typ e T (Srt prop))
               ->(cl_judge e t)=Trm.
Lemma class_typ: (e:env)(t,T:term)(typ e t T)->(typ e T (Srt kind))
               ->(cl_judge e t)=Typ.
Lemma class_ord: (e:env)(t:term)(typ e t (Srt kind))->(cl_judge e t)=Ord.

Lemma inv_class: (e:env)(t,T:term)(typ e t T)
                  -> Cases (cl_judge e t) of
                       Trm => (typ e T (Srt prop))
                     | Typ => (typ e T (Srt kind))
                     | Ord => T=(Srt kind)
                     end.
```

Preuve

On commence par démontrer les deux résultats

```
Lemma class_sound_w: (e:env)(t,T:term)(typ e t T)->
             ((n:nat)(x:term)(item ? x e n)->(f:env)(trunc ? (S n) e f)->
                 (((typ f x (Srt prop))->(nthf (class_env e) n)=Typ)
              /\ ((typ f x (Srt kind))->(nthf (class_env e) n)=Ord)))
                 ->  ((typ e T (Srt prop))->(cl_judge e t)=Trm)
                  /\ ((typ e T (Srt kind))->(cl_judge e t)=Typ)
                  /\ ((T=(Srt kind))->(cl_judge e t)=Ord).

Lemma corr_class_env:  (e:env)(wf e)->(n:nat)(x:term)(item ? x e n)->
          (f:env)(trunc ? (S n) e f)->
                 (((typ f x (Srt prop))->(nthf (class_env e) n)=Typ)
              /\ ((typ f x (Srt kind))->(nthf (class_env e) n)=Ord)).
```

Cela rappelle la manière de démontrer les lemmes d'affaiblissement et de substitution en trois temps. ■

Théorème 16 *Le jugement de typage ordonne les termes selon leur classe:*

$$\Gamma \vdash t : T \wedge T \neq \text{Kind} \Rightarrow Cl_\Gamma(t) \sqsubset Cl_\Gamma(T)$$

Traduction en Coq:

```
Lemma corr_lift: (e:env)(t,T,s:term)(typ e t T)->(typ e T s)
            ->(lift_cls (cl_judge e t) (cl_judge e T)).
```

Remarque La deuxième hypothèse de la formulation en Coq sert à écarter la cas $T = \text{Kind}$.

Corollaire 5.1

$$\Gamma \vdash v : V \wedge V \neq \text{Kind} \Rightarrow Cl_{\Gamma;V}(t) = Cl_\Gamma(t[0 := v])$$

Traduction en Coq:

```
Lemma class_subst: (e:env)(t,v,V,s:term)(typ e v V)->(typ e V s)
                ->(cl_judge (cons ? V e) t)=(cl_judge e (subst v t)).
```

Preuve

C'est la combinaison de `cl_judge_subst` et `corr_lift`. ■

5.1.4 Invariance de la classe par β-réduction

Lemme 5.3 *La classe est invariante par β-réduction.*

$$\left.\begin{array}{r}\Gamma \vdash T : K \\ T \rhd^*_\beta U\end{array}\right\} \Rightarrow Cl_\Gamma(T) = Cl_\Gamma(U)$$

Traduction en Coq:

```
Lemma cl_judge_red: (e:env)(T,K:term)(typ e T K)->(U:term)(red T U)
             ->(cl_judge e T)=(cl_judge e U).

Lemma class_env_red1: (e,f:env)(red1_in_env e f)->(wf e)
                ->(class_env e)=(class_env f).
```

Remarque

Ce lemme est équivalent du lemme d'auto-réduction pour les classes.

5.2 Squelettes d'ordre

Il s'agit d'une classification des ordres. Elle fait intervenir la notion de squelette: c'est la partie importante d'un ordre. Elle élimine tout ce qui concerne les types dépendants.

De cette manière, on n'a pas à décrire la projection des termes du Calcul des Constructions vers $\mathcal{F}\omega$, utilisée généralement pour déduire la normalisation forte du Calcul des Constructions à partir de celle de $\mathcal{F}\omega$.

Définition 5.4 *Les squelettes d'ordres sont des arbres binaires à une seule sorte de feuilles.*

Traduction en Coq:

```
Inductive skel: Type :=
   PROP: skel
 | PROD: skel->skel->skel.
```

Définition 5.5 *On définit les fonctions de sous-arbre gauche et droit d'un squelette.*

Traduction en Coq:

```
Definition predl: skel->skel :=
  [a:skel]Cases a of
      (PROD a1 _) => a1
    | _  => PROP
    end.

Definition predr: skel->skel :=
  [a:skel]Cases a of
      (PROD _ a2) => a2
    | _  => PROP
    end.
```

Remarque Elles servent à faciliter l'inversion des prédicats inductifs sur les squelettes.

5.2.1 Décidabilité de l'égalité des squelettes dans la sorte `Type`

Lemme 5.4 *L'égalité de deux squelettes est décidable.*

Traduction en Coq:

```
Lemma EQ_skel: (a,b:skel)(ORT (a===b) (notT (a===b))).
```

Ce résultat nous servira à construire des ensembles de termes (qui sont des objets de type `term` $\rightarrow$ `Prop`), il faut donc que le "OU" de ce résultat soit dans `Type`.

Pour cela, nous avons besoin de dupliquer une partie de la logique dans la sorte `Type`. On redéfinit la négation et la disjonction (voir annexe A.2). On définit aussi les listes non calculatoires d'objets non calculatoires (i.e. situées dans les univers `Type`), ainsi que quelques opérations simples sur ces listes.

5.2.2 Calcul du squelette d'un ordre

Définition 5.6 *Le squelette d'un ordre K se calcule de la façon suivante:*

```
Fixpoint cv_skel [K:term]: cls->skel :=
 [F:cls]Cases K of
     (Prod T U) => ([G:cls]Cases (cl_term U G) (cl_term T F) of
                          Ord Ord => (PROD (cv_skel T F) (cv_skel U G))
                        | Ord Typ => (cv_skel U G)
                        | _   _   => PROP
                        end
                   (cons ? (cl_term T F) F))
   | _  => PROP
   end.
```

Remarque
Cette fonction fait deux choses en même temps: elle détermine le squelette à la fois pour les ordres et pour Kind.

Définition 5.7 *On définit le squelette de type:*

```
Fixpoint typ_skel [T:term]: env->skel :=
 [e:env]Cases T of
    (Srt _) => PROP
  | (Ref k) => (Fix nthk {nthk/1: env->nat->skel :=
                   [e:env][n:nat]Cases e n of
                        (cons _ f) (S m) => (nthk f m)
                      | (cons h f) O     => (cv_skel h (class_env f))
                      | _          _     => PROP
                      end} e k)
  | (Abs A t) => Cases (cl_judge (cons ? A e) t) (cl_judge e A) of
                   Typ Ord => (PROD (cv_skel A (class_env e))
                                    (typ_skel t (cons ? A e)))
                 | Typ Typ => (typ_skel t (cons ? A e))
                 | _   _   => PROP
                 end
  | (App u v) => Cases (cl_judge e u) (cl_judge e v) of
                   Typ Typ => (predr (typ_skel u e))
                 | Typ Trm => (typ_skel u e)
                 | _   _   => PROP
                 end
  | (Prod _ _) => PROP
  end.
```

Remarque
Nous montrerons plus loin que si t est de type T, le squelette de type de t est le squelette de T. C'est pourquoi nous nous permettons cette dénomination ambigüe.

Lemme 5.5 *Si T n'est pas un ordre, son squelette est* PROP.

Traduction en Coq:

```
Lemma skel_not_ord: (T,K:term)(e:env)(lift_cls (cl_judge e T) (cl_judge e K))
                    ->(cv_skel T (class_env e))==PROP.
```

5.2.3 Propriétés d'invariance du squelette

Lemme 5.6 *Le squelette est invariant par relocation.*

Traduction en Coq:

```
Lemma cv_skel_lift: (x:class)(t:term)(f,g:cls)(k:nat)(insert ? x k f g)
                ->(cv_skel t f)==(cv_skel (lift_rec (S O) t k) g).
```

Lemme 5.7 *Le squelette est invariant par substitution par un terme dont la classe est inférieure à celle du type de la variable substituée.*

Traduction en Coq:

```
Lemma cv_skel_subst:
   (g:env)(v,V:term)(lift_cls (cl_judge g v) (cl_judge g V))
    ->(t:term)(k:nat)(e,f:env)(sub_in_env v V k e f)
     ->(trunc ? k f g)
   ->(cv_skel t (class_env e))==(cv_skel (subst_rec v t k) (class_env f)).
```

Lemme 5.8 *Le squelette du type est invariant par relocation.*

Traduction en Coq:

```
Lemma typ_skel_lift: (t:term)(k:nat)(f,g:env)(x:term)(ins_in_env x k f g)
                ->(typ_skel t f)==(typ_skel (lift_rec (S O) t k) g).
```

Lemme 5.9 *Le squelette du type est invariant par substitution par un terme dont la classe est inférieure à celle du type de la variable substituée, et dont le squelette du type est égal au squelette du type de la variable substituée.*

Traduction en Coq:

```
Lemma typ_skel_subst: (g:env)(v,V:term)
    (lift_cls (cl_judge g v) (cl_judge g V))
     ->(typ_skel v g)==(cv_skel V (class_env g))
      ->(t:term)(k:nat)(e,f:env)(sub_in_env v V k e f)
       ->(trunc ? k f g)
        ->(typ_skel t e)==(typ_skel (subst_rec v t k) f).
```

Lemme 5.10 *Deux termes typables et convertibles ont le même squelette.*

Traduction en Coq:

```
Lemma skel_red1: (e:env)(A,T:term)(typ e A T)->(B:term)(red1 A B)
```

```
      ->(cv_skel A (class_env e))==(cv_skel B (class_env e)).
Lemma skel_red: (A,B:term)(red A B)->(e:env)(T:term)(typ e A T)
      ->(cv_skel A (class_env e))==(cv_skel B (class_env e)).
Lemma skel_conv: (e:env)(A,T:term)(typ e A T)->(B,U:term)(typ e B U)
        ->(conv A B)->(cv_skel A (class_env e))==(cv_skel B (class_env e)).
```

5.2.4 Lien entre squelette et typage

On montre que le squelette du type de t est bien le squelette de type de t:

Théorème 17 *Si t est de type T, alors le squelette de T et le squelette de type de t sont égaux.*
TRADUCTION EN Coq:

```
Lemma skel_sound: (e:env)(t,T:term)(typ e t T)
              ->(cv_skel T (class_env e))==(typ_skel t e).
```

Lemme 5.11 *Le squelette de type est invariant par β-réduction.*
TRADUCTION EN Coq:

```
Lemma typ_skel_red1: (e:env)(T,U,K:term)(typ e T K)->(red1 T U)
        ->(typ_skel T e)==(typ_skel U e).
```

Lemme 5.12 *Les β-réductions dans l'environnement ne modifient pas le squelette de type.*
TRADUCTION EN Coq:

```
Lemma typ_skel_red_env: (e:env)(t,s:term)(typ e t s)
            ->(f:env)(red1_in_env e f)
                    ->(typ_skel t e)==(typ_skel t f).
```

6 La preuve de normalisation forte

Pour une présentation générale des preuves de normalisation forte dans les PTS, nous conseillons vivement le chapitre 3 de [18], qui explique clairement les principes des preuves de normalisation forte.

Cette preuve s'inspire en grande partie de la preuve de normalisation forte du système $\mathcal{F}$ d'Altenkirch ([1]).

6.1 Les Candidats

6.1.1 Les schémas de réductibilité

Définition 6.1 *On définit récursivement les schémas de réductibilité comme le plus petit ensemble d'objets tel que:*

- *les schémas d'ordre* `PROP` *sont les ensembles de termes,*
- *les schémas d'ordre* (`PROD` S_1 S_2) *sont les fonctions des schémas d'ordre* S_1 *vers les schémas d'ordre* S_2.

Traduction en Coq:

```
Fixpoint Can [K:skel]: Type :=
    Cases K of
      PROP => (term->Prop)
    | (PROD s1 s2) => ((Can s1)->(Can s2))
    end.
```

Nous introduisons maintenant une égalité sur les schémas légèrement différente de l'égalité usuelle. En théorie des Ensembles, cette égalité se confondrait avec l'égalité de Leibniz. Ce n'est pas le cas en Théorie des Types car les fonctions sont définies intentionnellement. Si l'on posait l'axiome d'extensionnalité, les deux égalités se confondraient, mais nous n'aurons pas recours à cette facilité, afin de montrer qu'elle n'est pas nécessaire.

Cette "égalité" n'est pas réflexive, mais nous n'aurons à considérer que des schémas qui sont égaux à eux-mêmes pour cette égalité (on dit que ces schémas sont *invariants*).

Définition 6.2 *L'égalité extensionnelle sur les schémas de réductibilité se définit par récurrence sur leur ordre:*

- *deux schémas d'ordre* `PROP` *sont égaux ssi ils contiennent les mêmes termes.*
- *deux schémas* C_1 *et* C_2 *d'ordre* (`PROD` S_1 S_2) *sont égaux ssi pour toute paire de schémas invariants d'ordre* S_1 *égaux, leurs images respectives par* C_1 *et* C_2 *sont égales.*

Traduction en Coq:

```
Definition eq_cand: (term->Prop)->(term->Prop)->Prop :=
```

```
    [X,Y:term->Prop](t:term)(X t)<->(Y t).

Fixpoint eq_can [s:skel]: (Can s)->(Can s)->Prop :=
  <[s0:skel](Can s0)->(Can s0)->Prop>Case s of
         eq_cand
         [s1,s2:skel][C1,C2:(Can (PROD s1 s2))]
           (X1,X2:(Can s1))(eq_can s1 X1 X2)
                     ->(eq_can s1 X1 X1)->(eq_can s1 X2 X2)
                      ->(eq_can s2 (C1 X1) (C2 X2))
         end.
```

Lemme 6.1 *L'égalité extensionnelle des schémas est symétrique, et transitive sur l'ensemble des schémas invariants.*

Traduction en Coq:

```
Lemma eq_can_sym: (s:skel)(X,Y:(Can s))(eq_can s X Y)->(eq_can s Y X).
Lemma eq_can_trans: (s:skel)(a,b,c:(Can s))
               (eq_can s a b)->(eq_can s b b)->(eq_can s b c)
                ->(eq_can s a c).
```

6.1.2 Les candidats de réductibilité d'ordre supérieur

Nous entrons maintenant dans le vif du sujet avec la notion de candidat de réductibilité d'ordre supérieur, qui est une généralisation des candidats de réductibilité introduits par Girard pour démontrer la cohérence du Système $\mathcal{F}$ ([9],[10]). Ces objets serviront à interpréter les types et les ordres du Calcul des Constructions.

Cette généralisation est utile lorsque l'on veut interpréter des ordres dont le squelette n'est pas réduit à `PROP`, c'est-à-dire pour les systèmes du cube de Barendregt autorisant la formation du produit (Kind,Kind,Kind). Ce n'était pas le cas pour le Système $\mathcal{F}$ car c'est un système où tous les ordres ont pour squelette `PROP`.

Nous commençons par définir l'ensemble des termes neutres, qui sert dans la définition des candidats de réductibilité de Girard.

Définition 6.3 *Un terme est dit neutre si ce n'est pas une abstraction.*

Traduction en Coq:

```
Definition neutral: term->Prop := [t:term](u,v:term)~(t=(Abs u v)).
```

Cet ensemble sera noté $\mathcal{N}eutral$.

Définition 6.4 *On appelle candidat de réductibilité tout schéma d'ordre* `PROP` *(i.e. un ensemble de termes) X vérifiant les trois propriétés de clôture suivantes:*

$$X \subset \mathcal{SN}$$
$$t \in X \wedge t \rhd_\beta u \Rightarrow u \in X$$
$$t \in \mathcal{N}eutral \wedge (\forall u, t \rhd_\beta u \Rightarrow u \in X) \Rightarrow t \in X$$

TRADUCTION EN Coq:

```
Definition is_cand: (term->Prop)->Prop :=
    [CR:term->Prop]
       ((t:term)(CR t)->(sn t))
     /\((t:term)(CR t)->(u:term)(red1 t u)->(CR u))
     /\((t:term)(neutral t)->((u:term)(red1 t u)->(CR u))->(CR t)).
```

Les candidats de réductibilité contiennent les variables et ils sont clos par β-réduction:

```
Lemma int_var: (n:nat)(CR:term->Prop)(is_cand CR)->(CR (Ref n)).
Lemma cr_red: (R:term->Prop)(is_cand R)->(a,b:term)(R a)->(red a b)->(R b).
```

Définition 6.5 *On définit récursivement l'ensemble des candidats de réductibilité d'ordre supérieur:*

- *un schéma d'ordre* `PROP` *est un candidat si et seulement si c'est un candidat de réductibilité*
- *un schéma d'ordre* `(PROD` S_1 S_2`)` *est un candidat si l'image de tout candidat invariant d'ordre* S_1 *est un candidat d'ordre* S_2.

TRADUCTION EN Coq:

```
Fixpoint is_can [s:skel]: (Can s)->Prop :=
   <[s0:skel](Can s0)->Prop>Case s of
        [CR:term->Prop](is_cand CR)
        [s1,s2:skel][C:(Can s1)->(Can s2)]
          (X:(Can s1))(is_can s1 X)->(eq_can s1 X X)->(is_can s2 (C X))
        end.
```

```
Lemma is_can_prop: (X:term->Prop)(is_can PROP X)->(is_cand X).
```

6.1.3 Les candidats par défaut

Définition 6.6 *On définit récursivement les candidats par défaut:*

- *le candidat par défaut d'ordre* `PROP` *est* $\mathcal{SN}$
- *la candidat par défaut d'ordre* `(PROD` S_1 S_2`)` *est le schéma constant qui retourne le candidat par défaut d'ordre* S_2

TRADUCTION EN Coq:

```
Fixpoint default_can [s:skel]: (Can s) :=
   <[ss:skel](Can ss)>Case s of
       sn
       [s1,s2:skel][_:(Can s1)](default_can s2)
       end.
```

Propriétés du candidat par défaut:

- il est invariant
- c'est effectivement un candidat de réductibilité d'ordre supérieur.

```
Lemma CR_sn: (is_cand sn).
Lemma def_can_cr: (s:skel)(is_can s (default_can s)).
Lemma def_inv: (s:skel)(eq_can s (default_can s) (default_can s)).
```

6.1.4 Le produit de candidats

On définit une opération sur les candidats, ce qui permettra d'interpréter le produit de deux types à partir de leurs interprétations.

Définition 6.7 *Le produit d'un schéma F d'ordre* (PROD S PROP) *par un schéma CR d'ordre* PROP *est un schéma d'ordre* PROP *défini de la façon suivante:*
TRADUCTION EN Coq:

```
Definition Pi: (s:skel)(Can (PROD s PROP))->(term->Prop)->(term->Prop) :=
     [s:skel][F:(Can (PROD s PROP))][CR:term->Prop][t:term]
        (u:term)(CR u)->(C:(Can s))(is_can s C)->(eq_can s C C)
                  ->(F C (App t u)).
```

Lemme 6.2 *Le produit des schémas préserve l'égalité extensionnelle et la notion de candidat de réductibilité d'ordre supérieur.*
TRADUCTION EN Coq:

```
Lemma eq_Pi: (s:skel)(F1,F2:(Can (PROD s PROP)))(X,Y:term->Prop)
                (eq_can (PROD s PROP) F1 F2)->(eq_can PROP X Y)
                   ->(eq_can PROP (Pi s F1 X) (Pi s F2 Y)).

Lemma CR_pi: (s:skel)(F:(Can (PROD s PROP)))(is_can (PROD s PROP) F)
                   ->(CR:term->Prop)(is_cand CR)->(is_cand (Pi s F CR)).
```

Le fait d'interpréter les types par des candidats de réductibilité au lieu de $\mathcal{SN}$ déplace la difficulté de la preuve de normalisation du cas de l'application vers celui de l'abstraction. Les deux lemmes suivants vont nous simplifier le travail au moment de la récurrence principale.

```
Lemma red1_subst_cand: (CR:term->Prop)(is_cand CR)->(T:term)(sn T)
                  ->(m:term)(sn m)->(u:term)(sn u)->(CR (subst u m))
                   ->(v:term)(red1 (App (Abs T m) u) v)->(CR v).

Lemma Pi_sound: (A:term->Prop)(s:skel)(F:(Can s)->(term->Prop))(T,m:term)
          (is_cand A)->(is_can (PROD s PROP) F)
           ->((n:term)(A n)->(C:(Can s))(is_can s C)
               ->(eq_can s C C)->(F C (subst n m)))
            ->(sn T)->(Pi s F A (Abs T m)).
```

6.2 Interprétation des termes et des types

6.2.1 Interprétation en tant que terme

Définition 6.8 *On interprète les variables en tant que terme grâce à une fonction des entiers vers les termes.*

TRADUCTION EN Coq:

```
Definition intt := nat->term.
```

On introduit le décalage de l'interprétation, qui correspond à la relocation des indices de de Bruijn.

```
Definition shift_intt : intt->term->intt :=
   [i:intt][t:term][n:nat]Cases n of
      O     => t
     | (S k) => (i k)
     end.
```

L'interprétation d'un terme se fait par substitution parallèle de ses variables:

```
Fixpoint int_term [t:term]: intt->nat->term :=
  [I:intt][k:nat]Cases t of
     (Srt s) => (Srt s)
   | (Ref n) => Case (le_gt_dec k n) of
                  [_:(le k n)](lift k (I (minus n k)))
                  [_:(gt k n)](Ref n)
                end
   | (Abs A t)  => (Abs (int_term A I k) (int_term t I (S k)))
   | (App u v)  => (App (int_term u I k) (int_term v I k))
   | (Prod A B) => (Prod (int_term A I k) (int_term B I (S k)))
   end.
```

On montre l'équivalence entre substitution et interprétation des termes:

```
Lemma int_term_subst: (t:term)(it:intt)(k:nat)(x:term)
     (subst_rec x (int_term t it (S k)) k)=(int_term t (shift_intt it x) k).
```

6.2.2 Interprétation en tant que type ou ordre

Définition 6.9 *Les interprétations des variables en tant que type ou ordre sont des listes de schémas quelconques.*

TRADUCTION EN Coq:

```
Inductive Int_K: Type :=
    iK: (s:skel)(Can s)->Int_K.
Definition intP := (TList Int_K).
```

On définit l'extension standard d'une interprétation:

```
Definition def_cons: cls->term->intP->intP :=
  [e:cls][K:term](TCons (iK (cv_skel K e) (default_can (cv_skel K e)))).
```

Nous introduisons la fonction qui retourne le candidat passé en argument si le squelette correspond, ou le candidat par défaut s'ils ne correspondent pas.

```
Definition extract_CR: (s:skel)Int_K->(Can s) :=
 [s:skel][i:Int_K]<Can s>Cases i of
   (iK si Ci) =>
        Case (EQ_skel si s) of
          [y:(si===s)]<[s0:skel][_:(si===s0)](Can s0)>Case y of
                              Ci
                              end
          [_:(notT (si===s))](default_can s)
          end
   end.
```

Définition 6.10 *La fonction d'interprétation des types et définie de la façon suivante:*

```
Fixpoint int_typ [T:term]: env->intP->(s:skel)(Can s) :=
 [e:env][ip:intP][s:skel]Cases T of
    (Srt _) => (default_can s)
  | (Ref n) => (extract_CR s (Tnth_def Int_K (iK PROP sn) ip n))
  | (Abs A t) => Cases (cl_judge e A) of
                  Ord => <[s0:skel](Can s0)>Case s of
                           (default_can PROP)
                           [s1,s2:skel][C:(Can s1)](int_typ t (cons ? A e)
                                                      (TCs ? (iK s1 C) ip) s2)
                           end
                | Typ => (int_typ t (cons ? A e)
                                   (def_cons (class_env e) A ip) s)
                | _   =>(default_can s)
                end
  | (App u v) => Cases (cl_judge e u) (cl_judge e v) of
                  Typ Trm => (int_typ u e ip s)
                | Typ Typ => ((int_typ u e ip (PROD (typ_skel v e) s))
                                    (int_typ v e ip (typ_skel v e)))
                | _   _   => (default_can s)
                end
  | (Prod A B) => <[s0:skel](Can s0)>Case s of
                     ([s:skel](Pi s ([C:(Can s)](int_typ B (cons ? A e)
                          (TCs ? (iK s C) ip) PROP)) (int_typ A e ip PROP))
                      (cv_skel A (class_env e)))
                     [s1,s2:skel](default_can (PROD s1 s2))
                     end
  end.
```

6.2.3 Résultats concernant les types dépendants

Les lemmes de cette section sont spécifiques à la Théorie des Types.

```
Lemma extr_eq: (P:(s:skel)(Can s)->Prop)(s:skel)(c:(Can s))(i:Int_K)
  (i==(iK s c))->(P s c)->
  (P s (extract_CR s i)).

Lemma eq_can_extr: (s,si:skel)(X,Y:(Can s))(eq_can s X Y)
         ->(eq_can si (extract_CR si (iK s X)) (extract_CR si (iK s Y))).

Lemma simpl_eq_extr: (s:skel)(X,Y:(Can s))
  (eq_can s X Y)==(eq_can s (extract_CR s (iK s X)) (extract_CR s (iK s Y))).

Lemma inv_eq_int: (sx,sy:skel)(X,X1:(Can sx))(Y,Y1:(Can sy))
       (iK sx X)==(iK sy Y)->(iK sx X1)==(iK sy Y1)
         ->(eq_can sx X X1)==(eq_can sy Y Y1).
```

6.2.4 Interprétations invariantes, équivalentes

On étend la notion d'invariance aux interprétations:

```
Inductive int_inv: intP->Prop :=
  iv_nl: (int_inv TNil)
| iv_cs: (i:intP)(s:skel)(C:(Can s))(int_inv i)->(eq_can s C C)
          ->(int_inv (TCons (iK s C) i)).

Lemma ins_int_inv: (e,f:intP)(k:nat)(y:Int_K)(Tins y k e f)
                    ->(int_inv f)->(int_inv e).
```

Puis la notion d'égalité extensionnelle:

```
Inductive int_eq_can: intP->intP->Prop :=
  ieq_nl: (int_eq_can TNil TNil)
| ieq_cs: (i,i':intP)(s:skel)(X,Y:(Can s))(int_eq_can i i')
          ->(eq_can s X Y)->(eq_can s X X)->(eq_can s Y Y)
           ->(int_eq_can (TCons (iK s X) i) (TCons (iK s Y) i')).

Lemma int_inv_int_eq_can: (i:intP)(int_inv i)->(int_eq_can i i).
```

6.3 Propriété d'invariance de l'interprétation des types

Lemme 6.3 *Les interprétations d'un type dans deux interprétations équivalentes sont équivalentes.*

Traduction en Coq:

```
Lemma int_equiv_int_typ: (T:term)(i,i':intP)(int_eq_can i i')
         ->(s:skel)(e:env)(eq_can s (int_typ T e i s) (int_typ T e i' s)).
```

Lemme 6.4 *L'interprétation des types est invariante par réduction dans l'environnement.*
TRADUCTION EN Coq:

```
Lemma int_typ_red1_env: (e:env)(t,T:term)(typ e t T)
               ->(s:skel)(ip:intP)(int_inv ip)
                ->(f:env)(red1_in_env e f)
                 ->(eq_can s (int_typ t e ip ?) (int_typ t f ip ?)).
```

Lemme 6.5 *L'interprétation des types est invariante par relocation.*
TRADUCTION EN Coq:

```
Lemma nth_lift_int: (y:Int_K)(s0:skel)(ipe,ipf:intP)(n,k:nat)(e,f:env)
     (TIns ? y k ipe ipf)
       -> (int_typ (lift_rec (S O) (Ref n) k) f ipf s0)
            ==(int_typ (Ref n) e ipe s0).

Lemma lift_int_typ: (y:Int_K)(x,T:term)(k:nat)(e,f:env)
      (ins_in_env x k e f)
       ->(ipe,ipf:intP)(TIns ? y k ipe ipf)
        ->(int_inv ipf)
         ->(s:skel)(eq_can s (int_typ T e ipe s)
                             (int_typ (lift_rec (S O) T k) f ipf s)).
```

Lemme 6.6 *L'interprétation des types est invariante par substitution d'une variable de type par un terme dont la classe est Typ, si cette variable est interprétée par l'interprétation de son type.*
TRADUCTION EN Coq:

```
Lemma subst_int_typ: (g:env)(v,V:term)(typ g v V)->(cl_judge g v)=Typ
      ->(ipg:intP)(e:env)(T,K:term)(typ e T K)
       ->(k:nat)(f:env)(ipe,ipf:intP)(trunc ? k f g)->(sub_in_env v V k e f)
        ->(TTrunc ? k ipf ipg)
         ->(TIns ? (iK (typ_skel v g) (int_typ v g ipg (typ_skel v g))) k ipf ipe)
          ->(int_inv ipe)
           ->(eq_can (typ_skel T e) (int_typ T  e ipe (typ_skel T e))
                       (int_typ (subst_rec v T k) f ipf (typ_skel T e))).
```

Lemme 6.7 *L'interprétation des types est invariante par substitution d'une variable de terme par un terme dont la classe est Trm.*
TRADUCTION EN Coq:

```
Lemma subst_int_typ_trm: (g:env)(v,V:term)(typ g v V)->(cl_judge g v)=Trm
            ->(y:Int_K)(e:env)(T,K:term)(typ e T K)
             ->(k:nat)(f:env)(ipe,ipf:intP)(trunc ? k f g)
              ->(sub_in_env v V k e f)->(TIns ? y k ipf ipe)->(int_inv ipe)
               ->~(cl_judge e T)=Trm
                ->(eq_can  (typ_skel T e) (int_typ T e ipe ?)
                          (int_typ (subst_rec v T k) f ipf ?)).
```

Lemme 6.8 *L'interprétation des types est invariante par β-réduction.*
TRADUCTION EN Coq:

```
Lemma int_typ_red1: (e:env)(U,K:term)(typ e U K)
       ->(V:term)(red1 U V)->(ip:intP)(int_inv ip)
        ->~(cl_judge e U)=Trm
         ->(eq_can (typ_skel U e) (int_typ U e ip ?) (int_typ V e ip ?)).

Lemma red_int_typ: (e:env)(U,K:term)(typ e U K)->~(cl_judge e U)=Trm
          ->(ip:intP)(int_inv ip)->(V:term)(red U V)
            ->(eq_can (typ_skel U e) (int_typ U e ip ?) (int_typ V e ip ?)).

Lemma conv_int_typ: (e:env)(U,V,K:term)(conv U V)
         ->(typ e U K)->(typ e V K)
          ->~(cl_judge e U)=Trm->(ip:intP)(int_inv ip)
           ->(eq_can (typ_skel U e) (int_typ U e ip ?) (int_typ V e ip ?)).
```

6.4 Interprétations adaptées

On restreint l'ensemble des interprétations avec une condition supplémentaire: elle doit être adaptée. Cette notion sera définie en deux temps.

Définition 6.11 *Une interprétation est* partiellement *adaptée si elle associe un candidat invariant à chacune des variables.*
TRADUCTION EN Coq:

```
Inductive can_adapt: intP->Prop :=
    ca_nl: (can_adapt (TNl ?))
  | ca_cs: (ip:intP)(s:skel)(x:term)(C:(Can s))(can_adapt ip)
              ->(is_can s C)->(eq_can s C C)
               ->(can_adapt (TCs ? (iK s C) ip)).
```

Lemme 6.9 *Les interprétations partiellement adaptées sont invariantes.*
TRADUCTION EN Coq:

```
Lemma adapt_int_inv: (ip:intP)(can_adapt ip)->(int_inv ip).
```

Lemme 6.10 *Si l'interprétation est partiellement adaptée, alors l'interprétation de n'importe quel type est un candidat.*
TRADUCTION EN Coq:

```
Lemma int_typ_cr: (t:term)(e:env)(ip:intP)(can_adapt ip)
                       ->(s:skel)(is_can s (int_typ t e ip ?)).
```

Définition 6.12 *Une interprétation est adaptée si:*

- *les schémas de réductibilité associés aux variables sont des candidats invariants dont le squelette est le même que celui du type de la variable.*
- *les variables sont interprétées par un terme qui appartient à l'interprétation de son type.*

TRADUCTION EN Coq:

```
Inductive int_adapt: env->intP->intt->Prop :=
  int_nil: (it:intt)(int_adapt (nil ?) (TNl ?) it)
| int_cs: (e:env)(ip:intP)(it:intt)(int_adapt e ip it)
           ->(s:skel)(T:term)(cv_skel T (class_env e))==s
            ->(C:(Can s))(is_can s C)->(eq_can s C C)
             ->(x:term)(int_typ T e ip PROP x)
          ->(int_adapt (cons ? T e) (TCs ? (iK s C) ip) (shift_intt it x)).
```

Lemme 6.11 *Toute interprétation adaptée est partiellement adaptée.*
TRADUCTION EN Coq:

```
Lemma int_can_adapt: (e:env)(ip:intP)(it:intt)(int_adapt e ip it)
                     ->(can_adapt ip).
```

Nous pouvons maintenant démontrer le résultat général, dont la normalisation forte est une conséquence:

Théorème 18 *Si l'interprétation des variables est adaptée à l'environnement* Γ*, alors pour tout jugement* $\Gamma \vdash t : T$*, l'interprétation de* t *appartient à l'interprétation de* T*.*
TRADUCTION EN Coq:

```
Lemma int_sound: (e:env)(t,T:term)(typ e t T)
                 ->(ip:intP)(it:intt)(int_adapt e ip it)
                    ->(int_typ T e ip PROP (int_term t it O)).
```

6.5 L'interprétation standard

Il ne reste plus qu'à exhiber une interprétation adaptée. Celle-ci interprète les variables par le Candidat par défaut et l'identité sur les variables:

```
Fixpoint def_intp [e:env]: intP :=
  Cases e of
    nil        => (TNl ?)
  | (cons t f) => (def_cons (class_env f) t (def_intp f))
  end.

Fixpoint def_intt [e:env]: nat->intt :=
 [k:nat]Cases e of
     nil        => [p:nat](Ref (plus k p))
   | (cons _ f) => (shift_intt (def_intt f (S k)) (Ref k))
   end.
```

On démontre qu'elle est adaptée:

```
Lemma def_intp_can: (e:env)(can_adapt (def_intp e)).
Lemma def_adapt: (e:env)(k:nat)(int_adapt e (def_intp e) (def_intt e k)).
```

Elle ne modifie pas les variables:

```
Lemma def_intt_id: (n:nat)(e:env)(k:nat)(def_intt e k n)=(Ref (plus k n)).
Lemma id_int_term: (e:env)(t:term)(k:nat)(int_term t (def_intt e O) k)=t.
```

6.6 Résultat principal

Il ne reste plus qu'à tout mettre ensemble pour aboutir au résultat:
Rappel (Théorème 8) *Tout terme bien typé est fortement normalisable.*
Traduction en Coq:

```
Theorem fort_norm: (e:env)(t,T:term)(typ e t T)->(sn t).
```

Rappel (Corollaire 2.5) *Tout type habité est fortement normalisable.*
Traduction en Coq:

```
Lemma type_sn: (e:env)(t,T:term)(typ e t T)->(sn T).
```

Conclusion

Nous allons tout d'abord donner les statistiques permettant d'évaluer le coût en ressources humaines et machine de la formalisation d'une théorie mathématique ainsi que son application au développement de programmes certifiés grâce à la technique d'extraction. Nous en tirerons ensuite un bilan quant à la possibilité de développer des programmes de façon formelle.

Statistiques

La taille du développement peut être évaluée grâce au tableau suivant, qui récapitule le nombre de définitions et de lemmes pour le développement. Nous mettons un peu à part la preuve de normalisation forte car, comme nous l'avons déjà mentionné, celle-ci ne rentre pas vraiment dans le cadre que nous nous étions fixé: la certification d'un vérificateur de preuves.

	hors SN	total	extrait
nombre de définitions	25	59	7
nombre de lemmes	108	174	9
taille du source	60 Ko	120 Ko	
nombre de lignes	2500	5000	350

On voit que la preuve de normalisation forte représente la moitié du développement à elle seule. Elle est en particulier très gourmande en définitions. On pouvait évidemment s'y attendre: les preuves de normalisation forte ont la réputation d'être assez complexes, et se prêtent plutôt mal à la formalisation en Théorie des Types.

Le rapport entre le nombre de déclarations extraites et le nombre total de déclarations en Coq peut paraître assez faible (16/233), mais nous le considérons comme assez bon, car c'est un fait largement admis que la preuve de correction d'un algorithme est presque toujours beaucoup plus longue que l'algorithme lui-même.

Temps de développement

Le temps de développement est une donnée importante pour apprécier si la programmation par extraction est prometteuse. En effet, si la vérification formelle d'un programme prend trop de temps, celle-ci risque d'apparaître comme un rêve inaccessible, et les développeurs s'en détourneront inévitablement.

Le prochain tableau indique le temps mis à rédiger la preuve en Coq. Ces delais comprennent le temps nécéssaire pour comprendre la théorie à formaliser, ainsi que l'apprentissage de l'outil Coq.

décidabilité du typage (hors SN)	2 mois
preuve de normalisation forte	3 mois
programmation du noyau Coc	2 jours
adaptation du lemme de Newman	1 jour

Certes, l'élaboration des preuves prend du temps, mais on a comme effet de bord non négligeable que la métathéorie du Calcul des Constructions est entièrement formalisée, ce qui n'était pas indispensable du point de vue de l'extraction. Si nous avions été très pressés d'obtenir un programme extrait, nous aurions pû poser en axiomes les résultats métathéoriques comme la confluence ou le lemme d'auto-réduction (connus depuis 85), auquel cas la vérification des programmes par rapport à ces axiomes n'aurait pris que quelques jours.

On remarquera aussi à quel point le code extrait s'est intégré facilement dans un programme écrit à la main. Cela est dû au fait que les fonctions sont spécifiées clairement, et donc il n'est pas nécessaire d'aller regarder le code engendré automatiquement.

Bilan

Nous avons montré que:

- Le système de Coq permet de parler des objets quasiment de façon naturelle, et la plupart du temps, les preuves formelles suivent fidèlement les preuves informelles.
- Le degré de détail à fournir est assez bon: dans ce rapport, *tous* les lemmes démontrés ont étés présentés; il y a peu de résultats démontrés qui ne valaient pas la peine d'être mentionnés informellement. Pour certains d'entre eux, ce sont des lemmes qui sont passés sous silence du fait de notre connaissance approfondie des notions mises en jeu. Par exemple, le fait qu'un sous-terme d'un terme fortement normalisable soit aussi fortement normalisable est un résultat qui nous paraît évident, probablement car nous nous figurons bien ce que ces notions représentent.
- Des problèmes non intuitifs pouvaient surgir en Théorie des Types. Ces problèmes peuvent en général être largement contournés si on pose quelques axiomes valides en Théorie des Ensembles. Mais ces problèmes n'apparaissent que pour des preuves de complexité logique élevée comme celle de normalisation forte.
- L'extraction produit des programmes lisibles, réutilisables et efficaces. La vérification formelle de programmes de taille non ridicule semble réalisable. Le fait de ne pas être obligé de tout démontrer pour procéder à l'extraction ne fait que nous conforter dans cette affirmation.

Extensions

Le bilan positif que nous venons de dresser nous conduit à considerer sérieusement les extensions qui nous permettent d'envisager à moyen ou long terme l'auto-vérification de Coq. Les extensions sont les suivantes:

- Ajouter les constantes et la δ-réduction dans le calcul. Ceci n'est pas très compliqué, mais cela apporterait beaucoup au confort d'utilisation du programme extrait.

- Les types inductifs: cela ajouterait beaucoup à la métathéorie, car cela ne se raccroche à rien de ce qui a été fait dans ce rapport.
- Les univers: l'ensemble des sortes devient infini. Si la métathéorie ne serait pas beaucoup plus complexe, la preuve de normalisation ne serait probablement plus faisable en Coq, à cause du théorème d'incomplétude de Gödel.
- La numérotation automatique des univers: en Coq, il n'est pas toujours nécessaire de préciser dans quel univers on définit un type: le système gère des contraintes permettant de savoir si une numérotation respectant les règles de typage existe.
- L'extraction: ceci permettrait d'achever le *bootstrap*.

Références

[1] Thorsten Altenkirch. A Formalization of the Strong Normalization Proof for System F in LEGO, In *Proceedings of the International Conference on Typed Lamdba Calculi and Applications, TLCA'93.* Springer-Verlag, LNCS 664, March 1993.

[2] Thorsten Altenkirch. Constructions, Inductive Types and Strong Normalization. Ph. D. Thesis, University of Edinburgh, 1993.

[3] H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science,* Vol II, Elsevier, 1992.

[4] Robert S. Boyer, Gilles Dowek. Towards Checking Proof-Checkers. In Herman Geuvers, editor, *Informal Proceedings of the Nijmegen Workshop on Types for Proofs and Programs,* May 1993.

[5] N.J. De Bruijn. Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indag. Math. Vol. 34 (5), pp. 381–392,* 1972.

[6] Thierry Coquand. Une Théorie des Constructions. Thèse de doctorat, Université Paris 7, 1985.

[7] Thierry Coquand, Gérard Huet. The Calculus of Constructions, in: *Information and Computation Vol. 76, February/March 1988* (ed.A.R.Meyer), Academic Press, London, 95-120.

[8] Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, Christine Paulin-Mohring, César Muñoz, Chetan Murthy, Catherine Parent, Amokrane Saïbi, Benjamin Werner. The Coq Proof Assistant Reference Manual Version 5.10. Rapport Technique 0177. Projet Coq-INRIA Rocquencourt-ENS Lyon. Juillet 95.

[9] J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, Univerité Paris 7, 1972.

[10] J.-Y. Girard, Y. Lafont, P. Taylor. Proofs ans Types. *Cambridge Tracts in Theoretical Computer Science 7.* Cambridge University Press.

[11] Herman Geuvers, Mark-Jan Nederhof. A Modular Proof of Strong Normalization for the Calculus of Constructions. *Journal of Functional Programming,* 1(2):155-189, April 1991.

[12] Gérard Huet. Residual Theory in λ-calculus: A Complete Gallina Development. Rapport de recherche INRIA 2002, 1993.

[13] Gérard Huet. The Constructive Engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. WorldScientific Publishing, 1989. Commemorative Volume for Gift Siromoney.

[14] Catherine Parent. Synthèse de preuves de programmes dans le Calcul des Constructions Inductives. Thèse de doctorat, École Normale Supérieure de Lyon, 1995.

[15] Christine Paulin-Mohring. Extraction de programmes dans le Calcul des Constructions. Thèse de doctorat, Université Paris 7, 1989.

[16] Christine Paulin-Mohring, Benjamin Werner. Synthesis of ML programs in Coq. *Journal of Symbolic Computation–special issue on automated programming*,1993.

[17] Robert Pollack. A Proof Checker for the Extended Calculus of Constructions. Ph. D. Thesis, University of Edinburgh, 1994.

[18] Benjamin Werner. Une Théorie des Constructions Inductives. Thèse de doctorat, Université Paris 7, 1994.

A Modules d'utilité générale

Cette annexe contient les résultats sur des structures couramment utilisées en mathématiques. Ils pourront faire partie des théories incluses dans le système.

A.1 Module MyList

Résultats complémentaires sur les listes d'objets calculatoires.

```
Require Le.
Require Gt.
Require Export PolyList.

Section Listes.

  Variable A:Set.

  Local List:=(list A).

  Inductive item [x:A]: List->nat->Prop :=
    item_hd: (l:List)(item x (cons ? x l) O)
  | item_tl: (l:List)(n:nat)(y:A)(item x l n)->(item x (cons ? y l) (S n)).

  Lemma fun_item: (u,v:A)(e:List)(n:nat)(item u e n)->(item v e n)->(u=v).

  Fixpoint nth_def [d:A; l:List]: nat->A :=
    [n:nat]Cases l n of
         nil         _     => d
       | (cons x _)  O     => x
       | (cons _ tl) (S k) => (nth_def d tl k)
       end.

  Lemma nth_sound: (x:A)(l:List)(n:nat)(item x l n)->(d:A)(nth_def d l n)=x.

  Lemma inv_nth_nl: (x:A)(n:nat)~(item x (nil ?) n).

  Lemma inv_nth_cs: (x,y:A)(l:List)(n:nat)(item x (cons ? y l) (S n))
                        ->(item x l n).

  Inductive insert [x:A]: nat->List->List->Prop :=
      insert_hd: (l:List)(insert x O l (cons ? x l))
    | insert_tl: (n:nat)(l,il:List)(y:A)(insert x n l il)
                   ->(insert x (S n) (cons ? y l) (cons ? y il)).
```

```
  Inductive trunc: nat->List->List->Prop :=
     trunc_O: (e:List)(trunc O e e)
   | trunc_S: (k:nat)(e,f:List)(x:A)(trunc k e f)
                ->(trunc (S k) (cons ? x e) f).



  Lemma item_trunc: (n:nat)(e:List)(t:A)(item t e n)
            ->(Ex [f:List](trunc (S n) e f)).

  Lemma ins_le: (k:nat)(f,g:List)(d,x:A)(insert x k f g)->(n:nat)(le k n)
         ->(nth_def d f n)=(nth_def d g (S n)).

  Lemma ins_gt: (k:nat)(f,g:List)(d,x:A)(insert x k f g)->(n:nat)(gt k n)
              ->(nth_def d f n)=(nth_def d g n).

  Lemma ins_eq: (k:nat)(f,g:List)(d,x:A)
             (insert x k f g)->(nth_def d g k)=x.

  Lemma list_item: (e:List)(n:nat) {t:A|(item t e n)}+{(t:A)~(item t e n)}.

End Listes.



  Fixpoint map [A,B:Set;f:(A->B);l:(list A)]: (list B) :=
      Cases l of
        nil        => (nil B)
      | (cons x t) => (cons ? (f x) (map ? ? f t))
      end.
```

A.2 Module MyLogicType

Définitions de la logique dans Type, et des listes non calculatoires d'objets dans Type.

```
  Inductive ORT [A,B:Type]: Type :=
     LeftT: A->(ORT A B)
   | RightT: B->(ORT A B).

Section TListes.
```

```
  Variable A: Type.

  Inductive TList: Type :=
      TNl: TList
    | TCs: A->TList->TList.

  Fixpoint Tnth_def [d:A; l:TList]: nat->A :=
    [n:nat]Cases l n of
         TNl        _     => d
       | (TCs x _)  O     => x
       | (TCs _ tl) (S k) => (Tnth_def d tl k)
       end.

  Inductive TIns [x:A]: nat->TList->TList->Prop :=
      TIns_hd: (l:TList)(TIns x O l (TCs x l))
    | TIns_tl: (n:nat)(l,il:TList)(y:A)(TIns x n l il)
                    ->(TIns x (S n) (TCs y l) (TCs y il)).

  Inductive TTrunc: nat->TList->TList->Prop :=
     Ttr_O: (e:TList)(TTrunc O e e)
   | Ttr_S: (k:nat)(e,f:TList)(x:A)(TTrunc k e f)
                                    ->(TTrunc (S k) (TCs x e) f).


End TListes.
```

B Programme extrait

Ce fichier est le produit de l'extraction. Il est engendré par Coq.

```
type nat = O
         | S of nat
;;

type (α, β) prod = Pair of α * β
;;

type sumbool = Left
             | Right
;;

type α sumor = Inleft of α
             | Inright
;;

let rec plus n m =
  match n with
    O → m
  | S p → S (plus p m)
;;

let acc_rec f =
  let rec acc_rec x =
    f x (fun y → acc_rec y)
  in acc_rec
;;

let lt_eq_lt_dec n =
  let rec f = function
    O →
      (fun m → let rec f = function
                  O → Inleft Right
                | S n0 → Inleft Left
                in f m)
  | S n0 →
      (fun m → let rec f0 = function
                  O → Inright
                | S n1 →
                    (match f n0 n1 with
                       Inleft x →
                         (match x with
                            Left → Inleft Left
```

```
                                  | Right → Inleft Right)
                              | Inright → Inright)
                         in f₀ m)
  in f n
;;

let le_lt_dec n =
  let rec f = function
    O → (fun m → Left)
  | S n₀ →
       (fun m → let rec f₀ = function
                          O → Right
                        | S n₁ →
                             (match f n₀ n₁ with
                                Left → Left
                              | Right → Right)
                        in f₀ m)
  in f n
;;

type α list = Nil
             | Cons of α ∗ (α list)
;;

type sort = Kind
           | Prop
;;

type term = Srt of sort
            | Ref of nat
            | Abs of term ∗ term
            | App of term ∗ term
            | Prod of term ∗ term
;;

let rec lift_rec n t k =
  match t with
    Srt x → Srt x
  | Ref i →
       (match le_lt_dec k i with
          Left → Ref (plus n i)
        | Right → Ref i)
  | Abs(t,m) → Abs((lift_rec n t k),(lift_rec n m (S k)))
  | App(u,v) → App((lift_rec n u k),(lift_rec n v k))
  | Prod(a,b) → Prod((lift_rec n a k),(lift_rec n b (S k)))
```

```
;;

let lift n t =
  lift_rec n t O
;;

let rec subst_rec n m k =
  match m with
    Srt x → Srt x
  | Ref i →
      (match lt_eq_lt_dec k i with
         Inleft c →
            (match c with
               Left → Ref (match i with
                             O → O
                           | S u → u)
             | Right → lift k n)
       | Inright → Ref i)
  | Abs(a,b) → Abs((subst_rec n a k),(subst_rec n b (S k)))
  | App(u,v) → App((subst_rec n u k),(subst_rec n v k))
  | Prod(t,u) → Prod((subst_rec n t k),(subst_rec n u (S k)))
;;

let list_item e n =
  let rec f = function
    Nil → (fun p → Inright)
  | Cons(a,l0) →
      (fun n0 → match n0 with
                  O → Inleft a
                | S k →
                    (match f l0 k with
                       Inleft u → Inleft u
                     | Inright → Inright))
  in f e n
;;

type env == term list;;

let eq_nat_dec n =
  let rec f = function
    O →
      (fun m → let rec f = function
                  O → Left
                | S n0 → Right
               in f m)
```

```
  | S n0 ->
      (fun m -> let rec f0 = function
                      O -> Right
                    | S n1 ->
                         (match f n0 n1 with
                            Left -> Left
                          | Right -> Right)
                    in f0 m)
  in f n
;;

let eqsort = function
  Kind -> (fun t -> match t with
                       Kind -> Left
                     | Prop -> Right)
| Prop -> (fun t -> match t with
                       Kind -> Right
                     | Prop -> Left)
;;

let eqterm u =
  let rec f = function
    Srt s ->
      (fun v -> match v with
                  Srt s0 ->
                     (match eqsort s s0 with
                        Left -> Left
                      | Right -> Right)
                | Ref n -> Right
                | Abs(t,t0) -> Right
                | App(t,t0) -> Right
                | Prod(t,t0) -> Right)
  | Ref n ->
      (fun v -> match v with
                  Srt s -> Right
                | Ref n0 ->
                     (match eq_nat_dec n n0 with
                        Left -> Left
                      | Right -> Right)
                | Abs(t,t0) -> Right
                | App(t,t0) -> Right
                | Prod(t,t0) -> Right)
  | Abs(t0,t1) ->
```

```
      (fun v → match v with
                   Srt s → Right
                 | Ref n → Right
                 | Abs(t3,t2) →
                      (match f t0 t3 with
                         Left →
                            (match f t1 t2 with
                               Left → Left
                             | Right → Right)
                       | Right → Right)
                 | App(t3,t2) → Right
                 | Prod(t3,t2) → Right)
 | App(t0,t1) →
      (fun v → match v with
                   Srt s → Right
                 | Ref n → Right
                 | Abs(t3,t2) → Right
                 | App(t3,t2) →
                      (match f t0 t3 with
                         Left →
                            (match f t1 t2 with
                               Left → Left
                             | Right → Right)
                       | Right → Right)
                 | Prod(t3,t2) → Right)
 | Prod(t0,t1) →
      (fun v → match v with
                   Srt s → Right
                 | Ref n → Right
                 | Abs(t3,t2) → Right
                 | App(t3,t2) → Right
                 | Prod(t3,t2) →
                      (match f t0 t3 with
                         Left →
                            (match f t1 t2 with
                               Left → Left
                             | Right → Right)
                       | Right → Right))
 in f u
;;

let compute_nf t =
 acc_rec (fun a norm → match a with
```

```
                        Srt s0 → Srt s0
                      | Ref n0 → Ref n0
                      | Abs(t0,t1) → Abs((norm t0),(norm t1))
                      | App(u0,v0) →
                          (match norm u0 with
                             Srt s → App((Srt s),(norm v0))
                           | Ref n → App((Ref n),(norm v0))
                           | Abs(a0,b) →
                                norm (subst_rec (norm v0) b O)
                           | App(a0,b) → App((App(a0,b)),(norm v0))
                           | Prod(a0,b) →
                                App((Prod(a0,b)),(norm v0)))
                      | Prod(t0,u0) → Prod((norm t0),(norm u0))) t
;;

let is_conv u v =
  match eqterm (compute_nf u) (compute_nf v) with
    Left → Left
  | Right → Right
;;

let red_to_prod t =
  match compute_nf t with
    Srt q → Inright
  | Ref r → Inright
  | Abs(s,t0) → Inright
  | App(u,v) → Inright
  | Prod(u,v) → Inleft (Pair(u,v))
;;

let red_to_sort t =
  match eqterm (Srt Kind) t with
    Left → Left
  | Right →
      (match is_conv (Srt Prop) t with
         Left → Left
       | Right → Right)
;;

let infer e t =
  let rec f = function
    Srt s →
      (fun f → match s with
                 Kind → Inright
```

```
              | Prop → Inleft (Srt Kind))
| Ref n →
    (fun e0 → match list_item e0 n with
                Inleft t → Inleft (lift_rec (S n) t O)
              | Inright → Inright)
| Abs(t1,t2) →
    (fun e0 → match f t1 e0 with
                Inleft t →
                  (match red_to_sort t with
                    Left →
                      (match f t2 (Cons(t1,e0)) with
                        Inleft b →
                          (match eqterm (Srt Kind) b with
                            Left → Inright
                          | Right → Inleft (Prod(t1,b)))
                      | Inright → Inright)
                  | Right → Inright)
              | Inright → Inright)
| App(t1,t2) →
    (fun e0 → match f t1 e0 with
                Inleft t →
                  (match red_to_prod t with
                    Inleft p →
                      (match p with
                        Pair(v,ur) →
                          (match f t2 e0 with
                            Inleft b →
                              (match is_conv v b with
                                Left →
                                  Inleft (subst_rec t2 ur O)
                              | Right → Inright)
                          | Inright → Inright))
                  | Inright → Inright)
              | Inright → Inright)
| Prod(t1,t2) →
    (fun e0 → match f t1 e0 with
                Inleft t →
                  (match red_to_sort t with
                    Left →
                      (match f t2 (Cons(t1,e0)) with
                        Inleft b →
                          (match eqterm (Srt Kind) b with
```

```
                                              Left → Inleft (Srt Kind)
                                            | Right →
                                                  (match is_conv (Srt Prop) b with
                                                     Left → Inleft (Srt Prop)
                                                   | Right → Inright))
                                     | Inright → Inright)
                              | Right → Inright)
                       | Inright → Inright)
  in f t e
;;

let check_typ e t tp =
  match infer e t with
    Inleft tp' →
        (match eqterm (Srt Kind) tp' with
           Left →
              (match eqterm (Srt Kind) tp with
                 Left → Left
               | Right → Right)
         | Right →
              (match infer e tp with
                 Inleft s →
                    (match is_conv tp tp' with
                       Left → Left
                     | Right → Right)
               | Inright → Right))
  | Inright → Right
;;

let add_typ e t =
  match infer e t with
    Inleft tp →
        (match red_to_sort tp with
           Left → Left
         | Right → Right)
  | Inright → Right
;;
```

C Noyau Coc

Ce qui suit est le code écrit à la main. Il met en œuvre le programme extrait.

```
#open "genlex";;

#open "infer";;

type expr=
        SRT of sort
      | REF of string
      | ABS of string*expr*expr
      | APP of expr*expr
      | PROD of string*expr*expr
;;

let rec int_of_nat = function
  O → 0
| S k → succ (int_of_nat k)
;;

let rec index x = function
  Nil → raise Not_found
| Cons(y,l) → if x = y then O else S (index x l)
;;
                                                    (* conversions term <=> expr *)

let var_of_ref n = "x"^(string_of_int n);;

let first_fv=ref 0;;

let rec find_free_var ctx=
  let v=var_of_ref !first_fv in
  try index v ctx; incr first_fv; find_free_var ctx
  with Not_found → v
;;

let rec var_indep x= function
  SRT _ → true
| REF y → x≠y
| ABS (y,tt,t) → (var_indep x tt) & ((x=y) or (var_indep x t))
| APP (u,v) → (var_indep x u) & (var_indep x v)
| PROD (y,tt,u) → (var_indep x tt) & ((x=y) or (var_indep x u))
;;

let rec term_of_expr ctx=function
```

```
  SRT s → Srt s
| REF x → Ref (index x ctx)
| ABS (x,tt,t) → Abs ((term_of_expr ctx tt),(term_of_expr (Cons(x,ctx)) t))
| APP (u,v) → App ((term_of_expr ctx u),(term_of_expr ctx v))
| PROD (x,tt,u) → Prod((term_of_expr ctx tt),(term_of_expr (Cons(x,ctx)) u))
;;

let expr_of_term ctx t=
  let rec exp_of_trm ctx=function
    Srt s → SRT s
  | Ref n → (match list_item ctx n with
                Inleft x → REF x
              | Inright → failwith "Fatal: rupture d'invariant!")
  | Abs (tt,t)→ let v=find_free_var ctx in
                 ABS (v,(exp_of_trm ctx tt),(exp_of_trm (Cons(v,ctx)) t))
  | App (u,v)→ APP ((exp_of_trm ctx u),(exp_of_trm ctx v))
  | Prod (tt,u)→ let v=find_free_var ctx in
                  PROD (v,(exp_of_trm ctx tt),(exp_of_trm (Cons(v,ctx)) u))
  in ((first_fv:=0) ; (exp_of_trm ctx t))
;;
```

(* *affichage* *)

```
let string_of_sort=function Kind → "Kind" | Prop → "Prop";;

let rec string_of_expr=function
  SRT s → string_of_sort s
| REF x → x
| ABS (x,tt,t) → "["^x^":"^(string_of_expr tt)^"]"^(string_of_expr t)
| APP (u,v) → "("^(string_of_app u)^" "^(string_of_expr v)^")"
| PROD (x,tt,u) → if var_indep x u
                   then (string_of_arrow tt)^"->"^(string_of_expr u)
                   else "("^x^":"^(string_of_expr tt)^")"^(string_of_expr u)
and string_of_app=function
  APP (u,v) → (string_of_app u)^" "^(string_of_expr v)
| t → string_of_expr t
and string_of_arrow=function
  ABS (x0,x1,x2) → "("^(string_of_expr (ABS (x0,x1,x2)))^")"
| PROD (x0,x1,x2) → "("^(string_of_expr (PROD (x0,x1,x2)))^")"
| t → string_of_expr t
;;

let print_term ctx t=print_string (string_of_expr (expr_of_term ctx t));;
```

(* *code noyau* *)

```
let cTX=ref(Nil:(string list));;

let eNV=ref(Nil:env);;

let exec_infer t= (match (infer !eNV t) with
                    (Inleft tt)→ print_string "Type infere: ";
                                 print_term !cTX tt; print_newline()
                  | Inright → print_string "mal type.\n" )
;;

let exec_axiom x a =
  try index x !cTX; print_endline "Nom deja utilise."
  with Not_found →
          match add_typ !eNV a with
            Left → eNV := Cons(a,!eNV); cTX := Cons(x,!cTX);
                   print_endline (x^" admis.")
          | Right → print_endline ("Le type de "^x^" n'est pas une proposition.")
;;

let exec_check trm typ =
  match check_typ !eNV trm typ with
    Left → print_endline "Correct."
  | Right → print_endline "Echec."
;;

let exec_quit() = print_endline "\nAu revoir..."; exit 0;;

let exec_delete() =
  match !cTX with
    Nil → print_endline "environement deja vide."
  | Cons(ax,ctx') → (match !eNV with
                       Nil → failwith "Delete: rupture d'invariant."
                     | Cons(_,env') → eNV := env'; cTX := ctx';
                                      print_endline (ax^" supprime."))
;;

let exec_list()=
  let rec prt_ctx = function
    Nil → ()
  | Cons (x,l) → prt_ctx l; print_string (x^" ")
  in
  prt_ctx !cTX;
  print_newline()
;;
```

(* *lexer* *)

```
let lexer=make_lexer
          [ "Prop"; "Kind"; "["; "]"; "("; ")"; ":"; "->"; "let"; "in"; "_"; ",";
           ":="; "Quit";"Axiom";"Infer";"Check";"Delete";"List";"." ]
;;
```

(* *parser* *)

```
let rec parse_star p= function
  [⟨ p x ; (parse_star p) l ⟩] → x::l
| [⟨ ⟩] → [ ]
;;

let anon_var= function
  [⟨ 'Kwd "_" ⟩] → "_"
| [⟨ 'Ident x ⟩] → x
;;

let virg_an_var= function
  [⟨ 'Kwd "," ; anon_var x ⟩] → x
;;

let lident= function
  [⟨ anon_var x ; (parse_star virg_an_var) l ⟩] → x::l
;;

let parse_atom= function
  [⟨ 'Kwd "Prop" ⟩] → SRT Prop
| [⟨ 'Kwd "Kind" ⟩] → SRT Kind
| [⟨ 'Ident x ⟩] → REF x
;;

let rec parse_expr= function
  [⟨ 'Kwd "[" ; lident l ; 'Kwd ":" ; parse_expr typ ; 'Kwd "]" ; parse_expr trm ⟩]
  → List.fold_right (fun x t→ABS (x,typ,t)) l trm
| [⟨ 'Kwd "let" ; anon_var x ; 'Kwd ":" ; parse_expr typ ; 'Kwd ":=" ; parse_expr arg ;
    'Kwd "in" ; parse_expr trm ⟩] → (APP ((ABS (x,typ,trm)),arg))
| [⟨ 'Kwd "(" ; parse_expr1 r ⟩] → r
| [⟨ parse_atom at ; (parse_expr2 at) r ⟩] → r
and parse_expr1 =function
  [⟨ 'Kwd "_" ; (parse_end_pi [ "_" ]) r ⟩] → r
| [⟨ 'Ident x ; (parse_expr3 x) r ⟩] → r
| [⟨ parse_expr t1 ; (parse_star parse_expr) l ; 'Kwd ")" ;
    (parse_expr2 (List.fold_left (fun t a→APP (t,a)) t1 l)) r ⟩] → r
```

```
and parse_expr₂ at= function
  [⟨ 'Kwd "->" ; parse_expr t ⟩] → PROD ("_",at,t)
| [⟨ ⟩] → at
and parse_expr₃ x=function
  [⟨ 'Kwd ","; anon_var y ; (parse_end_pi [x;y]) r ⟩] → r
| [⟨ 'Kwd ":"; parse_expr typ ; 'Kwd ")"; parse_expr trm ⟩] → PROD(x,typ,trm)
| [⟨ 'Kwd "->" ; parse_expr t ; (parse_star parse_expr) l ; 'Kwd ")"; str ⟩]
  → parse_expr₂ (List.fold_left (fun t a→APP(t,a)) (PROD ("_",(REF x),t)) l) str
| [⟨ (parse_star parse_expr) l ; 'Kwd ")" ; str⟩]
  → parse_expr₂ (List.fold_left (fun t a→APP(t,a)) (REF x) l) str
and parse_end_pi lb=function
  [⟨ (parse_star virg_an_var) l ; 'Kwd ":" ; parse_expr typ ; 'Kwd ")" ; parse_expr trm ⟩]
  → List.fold_right (fun x t→PROD(x,typ,t)) (lb@l) trm
;;

let parse_term ctx strm=term_of_expr ctx (parse_expr strm);;

let parse_cmd ctx=function
  [⟨ 'Kwd "Infer" ; (parse_term ctx) t ; 'Kwd "." ⟩] → exec_infer t
| [⟨ 'Kwd "Axiom" ; 'Ident x ; 'Kwd ":" ; (parse_term ctx) ax ; 'Kwd "." ⟩]
  → exec_axiom x ax
| [⟨ 'Kwd "Check" ; (parse_term ctx) trm ; 'Kwd ":" ; (parse_term ctx) typ ; 'Kwd "." ⟩]
  → exec_check trm typ
| [⟨ 'Kwd "Quit" ; 'Kwd "." ⟩] → exec_quit()
| [⟨ 'Kwd "Delete" ; 'Kwd "." ⟩] → exec_delete()
| [⟨ 'Kwd "List" ; 'Kwd "." ⟩] → exec_list()
;;

                                                        (* boucle toplevel *)

let rec skip_til_dot= function
  [⟨ 'Kwd "." ⟩] → ()
| [⟨ '_ ; strm ⟩] → skip_til_dot strm
;;

let prompt()= print_string "\nCoC < "; flush stdout;;

let parse_commande strm=
  prompt();
  try parse_cmd !cTX strm
  with (Stream__Parse_error "") | Stream__Parse_failure | Not_found
       → skip_til_dot strm;
          print_endline "\nErreur de syntaxe."
;;
```

```
let rec top_loop= function
  [< parse_commande _ ; strm >] → top_loop strm
| [< >] → print_endline "EOF!"; flush stdout
;;

let go()= top_loop (lexer (Stream_of_channel stdin));;

go();;
```

D Lemme de Newman en Coc

Ceci est le script d'une session en Coc vérifiant la preuve du lemme de Newman.

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


(* Axiomes simulant les definitions *)

Axiom Rstar:A->A->Prop.
Axiom unfold_Rstar: (P:(A->A->Prop)->Prop)
        (P [x,y:A](P0:A->A->Prop)
             ((u:A)(P0 u u))
              ->((u:A)(v:A)(w:A)(R u v)->(P0 v w)->(P0 u w))->(P0 x y))
          ->(P Rstar).

Axiom Rstar':A->A->Prop.
Axiom unfold_Rstar': (P:(A->A->Prop)->Prop)
        (P [x,y:A](P:A->A->Prop)
             (P x x)->((u:A)(R x u)->(Rstar u y)->(P x y))->(P x y))
          ->(P Rstar').

Axiom coherence: A->A->Prop.
Axiom unfold_coherence: (P:(A->A->Prop)->Prop)
        (P [x:A][y:A](P:Prop)((z:A)(Rstar x z)->(Rstar y z)->P)->P)
                ->(P coherence).



(* Les 2 hypotheses du Lemme de Newman *)

Axiom Hyp1:(x:A)(P:A->Prop)((y:A)((z:A)(R y z)->(P z))->(P y))->(P x).
Axiom Hyp2:(x:A)(y:A)(z:A)(R x y)->(R x z)->(coherence y z).


(* Verification du lemme *)

Check

let Rstar_reflexive: (x:A)(Rstar x x)
 := [x:A](unfold_Rstar [P:A->A->Prop](P x x)
```

```
      [P0:A->A->Prop]
        [H:(u:A)(P0 u u)]
          [_:(u:A)(v:A)(w:A)(R u v)->(P0 v w)->(P0 u w)](H x))
in

let Rstar_R: (x:A)(y:A)(z:A)(R x y)->(Rstar y z)->(Rstar x z)
 := [x,y,z:A][t1:(R x y)]
   (unfold_Rstar [P:A->A->Prop](P y z)->(P x z)
     [t2:(P0:A->A->Prop)
          ((u:A)(P0 u u))
           ->((u:A)(v:A)(w:A)(R u v)->(P0 v w)->(P0 u w))->(P0 y z)]
      [P:A->A->Prop][h1:(u:A)(P u u)]
        [h2:(u:A)(v:A)(w:A)(R u v)->(P v w)->(P u w)]
         (h2 x y z t1 (t2 [a,a0:A](P a a0) h1 h2)))
in

let  Rstar_transitive:  (x:A)(y:A)(z:A)(Rstar x y)->(Rstar y z)->(Rstar x z)
 := [x,y,z:A](unfold_Rstar [P:A->A->Prop](P x y)->(Rstar y z)->(Rstar x z)
     [h:(P0:A->A->Prop)
        ((u:A)(P0 u u))
         ->((u:A)(v:A)(w:A)(R u v)->(P0 v w)->(P0 u w))->(P0 x y)]
      (h [a,a0:A](Rstar a0 z)->(Rstar a z) [u:A][H:(Rstar u z)]H
        [u,v,w:A][t1:(R u v)][t2:(Rstar w z)->(Rstar v z)]
           [t3:(Rstar w z)](Rstar_R u v z t1 (t2 t3))))
in

let Rstar'_reflexive: (x:A)(Rstar' x x)
 := [x:A](unfold_Rstar' [P:A->A->Prop](P x x)
   [P:A->A->Prop][H:(P x x)][_:(u:A)(R x u)->(Rstar u x)->(P x x)]H)
in

let Rstar'_R: (x:A)(y:A)(z:A)(R x z)->(Rstar z y)->(Rstar' x y)
 := [x,y,z:A][t1:(R x z)][t2:(Rstar z y)]
    (unfold_Rstar' [P:A->A->Prop](P x y)
      [P:A->A->Prop]
       [_:(P x x)][h2:(u:A)(R x u)->(Rstar u y)->(P x y)](h2 z t1 t2))
in

let Rstar'_Rstar:  (x:A)(y:A)(Rstar' x y)->(Rstar x y)
 := [x,y:A](unfold_Rstar' [P:A->A->Prop](P x y)->(Rstar x y)
     [h:(P:A->A->Prop)
       (P x x)->((u:A)(R x u)->(Rstar u y)->(P x y))->(P x y)]
```

```
     (h [a,a0:A](Rstar a a0) (Rstar_reflexive x) [u:A](Rstar_R x u y)))
in

let Rstar_Rstar':  (x:A)(y:A)(Rstar x y)->(Rstar' x y)
 :=  [x,y:A](unfold_Rstar [P:A->A->Prop](P x y)->(Rstar' x y)
      [h:(P0:A->A->Prop)
       ((u:A)(P0 u u))
        ->((u:A)(v:A)(w:A)(R u v)->(P0 v w)->(P0 u w))->(P0 x y)]
       (h Rstar' [u:A](Rstar'_reflexive u)
         [u,v,w:A][h1:(R u v)]
          [h2:(Rstar' v w)](Rstar'_R u w v h1 (Rstar'_Rstar v w h2))))
in

let coherence_intro :  (x:A)(y:A)(z:A)(Rstar x z)->(Rstar y z)
                                ->(coherence x y)
 := [x,y,z:A][H:(Rstar x z)][H0:(Rstar y z)]
     (unfold_coherence [P:A->A->Prop](P x y)
      [P:Prop][H1:(z0:A)(Rstar x z0)->(Rstar y z0)->P](H1 z H H0))
in

let Rstar_coherence : (x:A)(y:A)(Rstar x y)->(coherence x y)
 :=  [x,y:A][h:(Rstar x y)](coherence_intro x y y h (Rstar_reflexive y))
in

let coherence_sym: (x:A)(y:A)(coherence x y)->(coherence y x)
 := [x,y:A](unfold_coherence [P:A->A->Prop](P x y)->(P y x)
     [H:(P:Prop)((z:A)(Rstar x z)->(Rstar y z)->P)->P][P:Prop]
      [H0:(z:A)(Rstar y z)->(Rstar x z)->P]
       (H P [z:A][H1:(Rstar x z)][H2:(Rstar y z)](H0 z H2 H1)))
in

let Diagram:
   (x:A)((u:A)(R x u)->(y:A)(z:A)(Rstar u y)->(Rstar u z)->(coherence y z))
     ->(y,z,u:A)(R x u)->(Rstar u y)
           ->(v:A)(R x v)->(Rstar v z)->(coherence y z)
 := [x:A][hyp_ind:(u:A)
            (R x u)->(y:A)(z:A)(Rstar u y)->(Rstar u z)->(coherence y z)]
     [y,z,u:A][t1:(R x u)][t2:(Rstar u y)][v:A][u1:(R x v)][u2:(Rstar v z)]
        (unfold_coherence
          [P:A->A->Prop]
           ((x0,y0,z0:A)(R x0 y0)->(R x0 z0)->(P y0 z0))
            ->((u0:A)(R x u0)
```

```
                 ->(y0,z0:A)(Rstar u0 y0)->(Rstar u0 z0)->(P y0 z0))
               ->(coherence y z)
            [Hyp0:(x0,y0,z0:A)(R x0 y0)->(R x0 z0)
                    ->(P:Prop)((z1:A)(Rstar y0 z1)->(Rstar z0 z1)->P)->P]
             [hyp_ind0:(u0:A)(R x u0)->(y0,z0:A)(Rstar u0 y0)->(Rstar u0 z0)
                       ->(P:Prop)((z1:A)(Rstar y0 z1)->(Rstar z0 z1)->P)->P]
              (Hyp0 x u v t1 u1 (coherence y z)
                [z0:A][H:(Rstar u z0)][H0:(Rstar v z0)]
                   (hyp_ind0 u t1 y z0 t2 H (coherence y z)
                     [z1:A][H1:(Rstar y z1)][H2:(Rstar z0 z1)]
                        (hyp_ind0 v u1 z z1 u2
                          (Rstar_transitive v z0 z1 H0 H2)
                          (coherence y z)
                          [z2:A][H3:(Rstar z z2)][H4:(Rstar z1 z2)]
                            (unfold_coherence [P:A->A->Prop](P y z)
                              [P:Prop]
                               [H5:(z3:A)(Rstar y z3)->(Rstar z z3)->P]
                                (H5 z2
                                  (Rstar_transitive y z1 z2 H1 H4) H3)))))
            Hyp2 hyp_ind)
in

let caseRxy:
    (x:A)((u:A)(R x u)->(y,z:A)(Rstar u y)->(Rstar u z)->(coherence y z))
        ->(y,z:A)(Rstar x y)->(Rstar x z)
                ->(u:A)(R x u)->(Rstar u y)->(coherence y z)
 := [x:A][hyp_ind:(u:A)
           (R x u)->(y:A)(z:A)(Rstar u y)->(Rstar u z)->(coherence y z)]
  [y,z:A][h1:(Rstar x y)][h2:(Rstar x z)][u:A][t1:(R x u)][t2:(Rstar u y)]
        (unfold_Rstar' [P:A->A->Prop](P x z)->(coherence y z)
          [hyp_:(P:A->A->Prop)(P x x)
                  ->((u0:A)(R x u0)->(Rstar u0 z)->(P x z))->(P x z)]
           (hyp_ [_:A][a:A](coherence y a)
             (coherence_sym x y (Rstar_coherence x y h1))
             (Diagram x hyp_ind y z u t1 t2))
          (Rstar_Rstar' x z h2))
in

let Ind_proof :
    (x:A)((u:A)(R x u)->(y:A)(z:A)(Rstar u y)->(Rstar u z)->(coherence y z))
            ->(y:A)(z:A)(Rstar x y)->(Rstar x z)->(coherence y z)
 := [x:A][hyp_ind:(u:A)
```

```
          (R x u)->(y:A)(z:A)(Rstar u y)->(Rstar u z)->(coherence y z)]
  [y,z:A][h1:(Rstar x y)][h2:(Rstar x z)]
     (unfold_Rstar' [P:A->A->Prop](P x y)->(coherence y z)
       [hyp_:(P:A->A->Prop)
              (P x x)->((u:A)(R x u)->(Rstar u y)->(P x y))->(P x y)]
        (hyp_ [_:A][a:A](coherence a z) (Rstar_coherence x z h2)
          (caseRxy x hyp_ind y z h1 h2))
       (Rstar_Rstar' x y h1))
in

[x:A](Hyp1 x
   [x:A](y:A)(z:A)(Rstar x y)->(Rstar x z)->(coherence y z) Ind_proof)

: (x:A)(y:A)(z:A)(Rstar x y)->(Rstar x z)->(coherence y z).

(* Affiche "Correct" si tout s'est bien passe *)
```

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