Université de Nice Sophia-Antipolis

Master 2 Mathématique

Rapport de stage

présenté en juin 2007

par

Cody ROUX

Types, Logique et Coercions Implicites

Responsable de stage : Loïc POTTIER

Remerciements

Mes premiers remerciements s'étendent d'abord à mon directeur de stage, M. Loïc Pottier, qui est une réserve inépuisable d'excellentes idées et d'encouragements, ainsi que M. André Hirschowitz qui a su me donner la motivation pour la matière et le sujet, et qui m'a aidé à me focaliser sur les choses essentielles.

Je voudrais également remercier toute l'équipe Marelle et tout particulièrement Ioana Paşca et Nicolas Julien, leur support moral et académique s'est révélé indispensable. M'ont également beaucoup aidé M. Gille Barthe pour m'avoir aidé à comprendre mon sujet, Pierre Abbrugiati pour m'avoir toujours écouté quelque soient mes questions.

Enfin je voudrais remercier tous les autres qui, consciemment ou inconsciemment, m'ont aidé à compléter ce mémoire.

Table des matières

1 Introduction

Presque cent ans se sont écoulées depuis la publication des *Principae Mathématicae* de Russell. Les enseignements de ces ouvrages furent clairs : s'il est en théorie possible de rendre complètement formel l'ensemble des concepts et des objets manipulés par les mathématiciens d'aujourd'hui, la pratique (naïve) en est complètement prohibitive. Au fil du siècle, la pratique des fondations s'est tournée vers d'autres bases, notamment la théorie des ensembles, la logique et les mathématiques se sont progressivement séparées, et la formalisation complète des mathématique a été abandonnée par tous sauf quelques écoles de pensée.

Evidemment, l'avènement de l'informatique change la donne. La Théorie des Types renaît de ses cendres, les Types ne sont plus seulement un concept proche de celui d'ensemble, mais acquièrent un contenu calculatoire et fournissent des garanties sur le comportement d'un *programme informatique*. L'*isomorphisme de Curry-Howard* permet alors de former un pont entre les deux concepts, et donne la motivation pour continuer la formalisation de théories mathématiques, cette fois grandement assistée par les outils informatiques qui révolutionnent notre capacité à effectuer des vérifications mécaniques et fastidieuses.

Cependant, la tâche n'en devient pas facile pour autant. Les mathématiques telles qu'elles sont pratiquées font très souvent appel à la capacité de voir un même objet sous plusieurs points de vue différents ; il peut même être suggéré que presque tout résultat profond utilise ce genre de changement de point de vue. De façon parallèle, en informatique, il est intéressant que certains objets d'un type puissent être vus comme des objets d'un type différent, l'exemple fondateur étant celui de l'*héritage*. Ces deux concepts sont liés aux notions de *coercions implicites* et de *sous typage*.

La bonne compréhension des coercions implicites est donc nécessaire pour continuer à développer l'objectif actuel de formalisation. Après avoir introduit les concepts en jeu, nous donnons certains problèmes dont il faut tenir compte lorsqu'on tente d'intégrer les coercions dans un système de typage, nous démontrons notamment un résultat d'indécidabilité. Nous restreindrons ensuite le discours aux types simples, où nous donnerons une méthode pour tenter de comprendre comment donner une coercion d'un type en un autre. Nous essayerons de généraliser au cas des types dépendants, et nous donnerons des idées de développements futurs.

2 La Notion de Sous Typage

Dorénavant, nous nous plaçons dans un système de types du genre des *PTS*, par exemple le Calcul des Constructions (voir [3],[10]), un exemple motivateur est celui du lambda-calcul simplement typé défini de la manière suivante :

- L'ensemble des types :

$$\mathcal{T} := \mathcal{V} \mid \mathcal{T} \to \mathcal{T}$$

  où $\mathcal{V}$ est un ensemble de variables de types
- L'ensemble des termes :

$$\Lambda := V \mid \lambda V : \mathcal{T}.\Lambda \mid (\Lambda\Lambda)$$

  où V est un ensemble de variables de termes
- les règles de typage :

$$\frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash (f\ t) : B}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}$$

Comme de coutume, nous noterons $A_1 \to A_2 \ldots A_{n-1} \to A_n$ pour $A_1 \to (A_2 \ldots (A_{n-1} \to A_n) \ldots)$ et $t\ u_1 \ldots u_n$ au lieu de $(\ldots (t\ u_1) \ldots) u_n$.

La notion de typage est issu d'une motivation informatique, celle d'avoir une information sur le contenu calculatoire d'un objet. La définition du sous-typage peut alors naturellement s'exprimer comme la propriété (informelle) suivante : *Un type A est un* sous-type *d'un type B si tout calcul effectué sur un objet de type B peut être effectué sur un objet de type A*. Un objet de type A peut être "vu comme" un objet de type B. Si A est un sous type de B (éventuellement dans un contexte Γ) on note $\Gamma \vdash A < B$ et on peut introduire la règle de typage :

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A < B}{\Gamma \vdash t : B}$$

Il est naturel d'imposer à cette relation d'être une relation de préordre, c'est à dire réflexive et transitive : en effet, si un objet x de type A peut avoir le même comportement calculatoire qu'un objet de type B qui a son tour peut avoir le même comportement qu'un objet de type C, alors x peut clairement se comporter comme un objet de type C. Cependant cette relation n'est pas forcément antisymétrique (donc pas une relation d'ordre), car il est possible pour deux types d'avoir le même comportement calculatoire sans

pour autant être égaux, d'autant plus que la notion d'égalité entre les types peut être définie de façon différentes selon la théorie en question.

Enfin il faut prendre en considération le rapport de cette relation de sous-typage avec les constructeurs de type. A titre d'exemple, plaçons-nous dans le cadre du Calcul des Constructions (voir [10]) dont l'ensemble des termes est donné par :

$$\mathcal{T} : = \mathcal{V} \mid \prod \mathcal{V} : \mathcal{T}.\mathcal{T} \mid \lambda \mathcal{V} : \mathcal{T}.\mathcal{T} \mid \Box \mid \star$$

et les règles de typage

$$\frac{}{\emptyset \vdash} \text{ Ctxt}$$

$$\frac{\Gamma \vdash T : s \qquad s \in \{\Box, \star\} \qquad x \notin \Gamma}{\Gamma ; x : T \vdash} \text{ Var}$$

$$\frac{s = \Box \text{ ou } \star}{\Gamma \vdash s : \Box} \text{ Axiom}$$

$$\frac{x \in \Gamma}{\Gamma \vdash x : \Gamma_x} \text{ Name}$$

$$\frac{\Gamma \vdash T : s_1 \qquad \Gamma ; x : T \vdash U : s_2 \qquad s_1, s_2 \in \{\Box, \star\}}{\Gamma \vdash \prod x : T.U : s_2} \text{ Prod}$$

$$\frac{\Gamma ; x : T \vdash t : U \qquad \Gamma \vdash \prod x : T.U : s}{\Gamma \vdash \lambda x : T.t : \prod x : T.U} \text{ Lam}$$

$$\frac{\Gamma \vdash t : \prod x : T.U \qquad \Gamma \vdash u : T}{\Gamma \vdash (t\ u) : U[x \leftarrow u]} \text{ App}$$

$$\frac{\Gamma \vdash T : s \qquad \Gamma \vdash x : U \qquad T =_\beta U}{\Gamma \vdash x : T} \text{ Conv}$$

On définit alors l'assertion $\Gamma \vdash A < B$ comme étant la conjonction des assertions $\Gamma \vdash A : \Box, \Gamma \vdash B : \Box$ et $A < B$ où $<$ est une relation de sous-typage (à définir, nous donnerons une définition possible dans la prochaine section). Comment se comporte le sous typage vis-à-vis de ces constructeurs ? La première remarque à faire est que les seuls termes qui ne sont pas des variables de type ou $\Box$ ou $\star$ et qui sont des types sont nécessairement construits avec le constructeur $\prod$. Il suffit donc d'examiner le comportement de celui-ci. C'est alors qu'apparait le phénomène de la *contravariance*.

On introduit la règle suivante :

$$\frac{\Gamma \vdash A' < A \qquad \Gamma, x\colon A' \vdash B' < B}{\Gamma \vdash \prod x\colon A.B < \prod x\colon A'.B'}$$

On remarque d'abord que si on a $\Gamma, x\colon A \vdash B\colon \square$ alors on a $\Gamma, x\colon A' \vdash B\colon \square$ grâce à la règle de sous-typage.

Le constructeur $\prod x\colon T_1.T_2$ est covariant par rapport à la relation de sous typage en T_2 et contravariant en T_1. Pourquoi cela ? Il faut examiner la motivation derrière le sous-typage. Un terme f de type $\prod x\colon A.B$ permet, à partir d'un terme de type A, de fournir un terme de type B (B dépend éventuellement de x). Si $B < B'$ il est alors possible, à partir d'un terme de type A, de fournir un terme de type B', en utilisant la règle de sous-typage appliqué au résultat précédent. Ceci explique $\prod x\colon A.B < \prod x\colon A.B'$, c'est à dire la covariance en le premier argument. Si de plus $A' < A$ alors pour tout terme $x\colon A'$ il est possible, en appliquant f, d'obtenir un terme de type B et donc un terme de type B'. Ceci explique la contravariance en le premier argument.

Cette règle rend difficile l'interprétation intuitive des types comme des ensembles et le sous-typage comme une inclusion. En effet si E et F sont des ensembles et $E' \subset E$ et $F \subset F'$ alors en général l'ensemble des fonctions de E dans F, souvent notée F^E, n'est pas naturellement inclus dans $F'^{E'}$. Au contraire, il semblerait qu'il puisse y avoir "plus" de fonctions de E dans F que de fonctions de E' dans F, chaque fonction de E' dans F pouvant à priori se prolonger de plusieurs manières vers une fonction de E dans F. Un point de vue plus intuitif pourrait se trouver dans l'ensemble des fonctions continues de E vers F (que l'on suppose munis d'une structure adéquate). Si $f \in F^E$ est continue, alors la restriction de f à E' (munie par exemple de la topologie induite de E) est encore continue ; cependant il existe, en général, des fonctions continues de E' dans F qui ne peuvent pas se prolonger en des fonctions continues sur tout l'ensemble E.

Nous avons donc établi quelles étaient les propriétés désirables d'une théorie des types avec une relation de sous-typage. Quelles sont les motivations d'un tel système ? Les deux motivations principales sont informatiques et mathématiques.

La motivation informatique est déja claire dans ce qui précède : nous voulons être capable de manipuler un objet en fonction de ses capacités calculatoires. Si cet objet présente les mêmes aspects calculatoires qu'un objet d'un type différent, nous voulons être capables d'effectuer les mêmes opérations sur lui que nous aurions pu effectuer sur un objet du deuxième type. Un exemple phare de ce type de motivation est l'héritage. Un objet va pouvoir "oublier" certains champs de son constructeur être utilisé par les mêmes processus qu'un objet de type "parent".

La motivation mathématique est issue de la pratique quotidienne et informelle des mathématiques. Lorsque nous parlons d'un objet mathématique,

il peut souvent être vu comme ayant deux aspects très différents. On peut par exemple voir un morphisme de structure comme une simple fonction (cet aspect rejoint alors la notion d'héritage mentionnée ci-dessus), ou un polinôme comme soit une liste (finie) de coefficients soit comme une fonction. Ce genre d'ambiguité est omniprésente en mathématique, et avoir la capacité de formaliser ce genre d'ambiguités semble être un enjeu important pour la capacité de formaliser de façon acceptable le corpus des mathématiques scolaires (un enjeu qui commence à devenir réaliste).

Il nous reste maintenant à trouver une définition effective de la relation de sous typage et à expliquer comment elle s'intègre dans le cadre d'une théorie des types, par exemple le calcul des constructions. Ce but va nous pousser à définir le concept de *coercion implicite*. Malheureusement, nous verrons qu'il n'est pas facile de donner une définition intéressante et effective.

3 Coercions Implicites dans le Calcul des Constructions

Pour définir la relation de sous typage, une façon possible de procéder est de se donner un ensemble de termes fonctionnels, appellés *coercions*. Nous donnons alors la définition :

Définition 1 *pour deux types A et B, $A < B$ si il existe une coercion c telle que c soit de type $A \to B$ (ou $\prod x\colon A.B$).*

La règle de typage dans un système avec coercions s'énoncerait alors :

$$\frac{\Gamma \vdash t\colon A \qquad \Gamma \vdash c\colon A \to B \text{ et } c \text{ est une coercion}}{\Gamma \vdash t\colon B}$$

3.1 Les Coercions dans le Système Coq

Un système de ce type est implémenté dans le logiciel *Coq* [5], une implémentation du Calcul des Constructions Inductives (une extension du Calcul des Constructions) de la façon suivante (voir [18] et [9]) :

On définit une *classe à n paramêtres* comme étant un *identificateur* pour un terme de type $\prod x_1\colon A_1 \ldots x_n\colon A_n.s$ avec s une *sorte*. On donne également deux classes spéciales :`SORTCLASS` et `FUNCLASS` à 0 paramêtres, qui vont servir a définir les coercions vers un terme de type $Type$ ou un terme de type fonctionnel. Il est alors possible de définir une coercion de la façon suivante :

1. On donne au système un terme f de type

$$\prod x_1\colon A_1 \ldots x_n\colon A_n. \prod y\colon C\ x_1 \ldots x_n.D\ u_1 \ldots u_m$$

avec C une classe a n paramêtres non égale à `SORTCLASS` ou `FUNCLASS`, et D une classe a m paramêtres, $u_1, \ldots, u_m$ des termes. La forme du type de f s'appelle la condition d'*héritage uniforme*.

2. Les identificateurs C et D sont ajoutés comme sommets à un *graphe d'héritage* Δ ainsi qu'une arrête entre ces deux sommets indexée par f.
3. Si la définition d'une coercion crée un deuxième chemin entre deux sommets du graphe Δ, seul le chemin le plus ancien est conservé.

La condition d'héritage uniforme garantit le fait que les n premiers arguments de f peuvent être considérés comme implicites. Un terme de type $C\ t_1 \ldots t_n$ pourra alors être coercé en un terme de type
$(D\ u_1 \ldots u_m)[x_1 \leftarrow t_1 \ldots x_n \leftarrow t_n]$ grâce à la fonction $f\ t_1 \ldots t_n$. Les coercions peuvent être composées selon un chemin dans le graphe d'héritage.

Il est utile de signaler lorsque une coercion est employée pour typer un terme t, que le terme manipulé par système n'est pas le terme t mais plutôt le terme reconstruit en appliquant les coercions nécessaires, transparentes à l'affichage.

Ce système a plusieurs défauts.

- Il est nécessaire de définir un constructeur pour chaque type pour lequel on veut faire une coercion.
- Nous sommes soumis à la condition d'héritage uniforme.
- Les coercions entre types n'induisent pas automatiquement de coercions entre types inductifs construits à partir de ceux-ci, en particulier il n'y a pas d'héritage.
- Il n'est possible de définir qu'un nombre fini de coercions, en particulier il n'est pas possible d'utiliser le polymorphisme si les types ne sont pas implicites.
- Etant donné un terme de la forme $(f\ t)$, avec $f\colon \prod x\colon A.B$ et $t\colon C$ l'algorithme cherche une coercion de C vers A, mais ne cherche pas de coercion de $\prod x\colon A.B$ vers un type de la forme $\prod x\colon C.X$.

En particulier, il est impossible de former des coercions du type suivant : $A \rightarrow (A \rightarrow A)$, $(A \rightarrow B) \rightarrow B$, $\forall E\colon \mathtt{Set}, \forall P\colon E \rightarrow \mathtt{Prop}, SubsP \rightarrow E$ avec *Subs* qui modélise le sous ensemble $\{x\colon E \mid Px\}$ et qui peut être défini en Coq par la commande :

$$\mathtt{Record}\ Subs(P\colon E \rightarrow \mathtt{Prop})\{x\colon E; in\colon Px\}$$

Nous pouvons essayer de généraliser cette construction afin de se libérer de certaines de ces contraintes. Cependant, il apparait rapidement que donner des coercions plus générales a souvent le défaut fatal de rendre indécidable le typage. Cette décidablité est pourtant nécessaire pour toute application pratique de la théorie des types. Notons que cette indécidablité est caractéristique de systèmes sans unicité des types, voir par exemple [20].

Donnons une preuve formelle de cette assertion, en étendant le Calcul des

Constructions avec un système qui n'a pas la première limitation précisée ci-dessus.

3.2 L'Indécidabilité du Typage avec Coercions

Définition 2 *On rajoute la règle suivante au Calcul des Constructions :*

$$\frac{\Gamma \vdash t\colon C[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n][y \leftarrow u]}{\Gamma \vdash t\colon D[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n][y \leftarrow u]}\ [\textstyle\prod x_1\colon A_1 \ldots x_n\colon A_n.\, \prod y\colon Cx_1 \ldots x_n.D; n] \in \boldsymbol{Coer}$$

où ***Coer*** *est une liste finie de doublets de la forme :*
$[\prod x_1\colon U_1 \ldots x_m\colon U_m.\, \prod y\colon V x_1 \ldots x_m.W; m]$*, avec* $m \in \mathbb{N}$*, qui désignent les coercions définies par l'utilisateur, avec la restriction importante :* $\prod x_1\colon A_1 \ldots x_n\colon A_n.\, \prod y\colon Cx_1 \ldots x_n.D$ *doit être un terme d'*ordre *moins de* 3 *(voir [19]) ; ceci pour rendre décidable le filtrage sur* D*. L'entier* n *désigne alors la profondeur de la variable sur laquelle porte la coercion. Nous pouvons appeler le nouveau calcul ainsi défini le* Calcul des Constructions Coercives (CCC)

Faisons quelques remarques :
- Les arguments $x_1 \ldots x_n$ sont implicites. Ceci est garanti par le fait qu'il sont en arguments dans le type de y. Ceci est nécessaire car une coercion ne peut avoir qu'un seul argument explicite.
- Il n'y a pas de condition supplémentaires sur C ou D. En particulier, C peut être un produit. Ceci est la différence fondamentale avec l'implémentation courante de Coq, dans lequel le typage est décidable.

Définition 3 *Un* semi système de Thue *est la donnée d'un alphabet* Σ *et d'une relation* $R \subset \Sigma^* \times \Sigma^*$*, où* Σ^* *est l'ensemble des mots finis sur l'alphabet* Σ*. On ne considère ici que des alphabets finis et des relations finies.*

Etant donné un semi système de Thue, deux mots $u, v \in \Sigma^*$*, une* réécriture de u vers v *est une suite de mots* $(u_0, \ldots, u_n)$ *telle que :*

1. $u_0 = u$*,* $u_n = v$
2. *pour tout* $i < n$*,* $u_i = u_i' \cdot m \cdot u_i''$*,* $u_{i+1} = u_i' \cdot m' \cdot u_i''$ *et* $(m, m') \in R$*, avec* $a \cdot b$ *désignant la concaténation de deux mots* a *et* b*.*

Théorème 1 *Le problème suivant (*Problème de Thue*) est indécidable : Etant donné un semi système de Thue* (Σ, R) *et deux mots* u *et* v*, existe-t-il une réécriture de* u *vers* v *?*

Une démonstration complète peut se trouver dans [11].

Ceci nous permet de démontrer :

Théorème 2 *Le typage dans le CCC est indécidable.*

Démonstration

Montrons que la décidablilité du typage est équivalent au problème de Thue :

Soit (Σ, R) un semi système de Thue, u et v deux mots de Σ^*. On définit alors un contexte Γ dans le calcul des constructions inductives constitué de :

- Pour chaque élément c de Σ (supposé fini) une variable de type nommée $[c]$.
- Une relation binaire sur les types, c'est à dire une variable $Prod$ de type $Type \rightarrow Type \rightarrow Type$.
  On notera $A * B$ au lieu de $Prod\ A\ B$, et $A_1 * A_2 * \ldots * A_n$ pour $(\ldots(A_1 * A_2) * \ldots * A_n)$. Pour tout élément de Σ^* on associe un type dans le calcul des constructions par le biais de la transformation : $[c_1 c_2 \ldots c_k] = [c_1] * [c_2] * \ldots * [c_k]$.
- Les coercions :

  $$a_1 := [\forall U\ V\ A\ B\ C\colon Type, \forall y\colon U * (A * B) * C * V, U * A * (B * C) * V; 5]$$
  $$a_2 := [\forall U\ V\ A\ B\ C\colon Type, \forall y\colon U * A * (B * C) * V, U * (A * B) * C * V; 5]$$

  qui dénotent l'associativité.
- Pour chaque couple (m, n) de R (également fini par hypothèse) on associe la coercion :

  $$r_i := [\forall U\ V\colon Type, \forall y\colon U * [m] * V, U * [n] * V; 2]$$

- Une variable de type $Type$ "inerte" notée 1.
- la variabe $x\colon 1 * [v] * 1$

On affirme alors : le jugement $\Gamma \vdash x\colon 1 * [u] * 1$ est dérivable dans le calcul des constructions coercif si et seulemnent si u se réécrit en v dans le semi-système de Thue Σ, R.

Pour typer x il faut nécessairement, si $u \neq v$, appliquer au moins une fois la règle de coercion, x étant atomique, on ne peut appliquer que ces règles et la règle **Ax**. Montrons maintenant que pour chaque coercion, si $x\colon 1 * [m] * 1$ avant application de la coercion avec $m \in \Sigma^*$, alors $x\colon 1 * [n] * 1$ après application de la coercion avec $n \in \Sigma^*$ et n se réécrit en m dans le système (Σ, R). Si la coercion est l'une des a_i, alors c'est clair par structure des a_i et par l'associativité de la concaténation dans Σ^*.

Si la coercion est l'une des r_i alors il est clair que le m et n sont les mots associés à la règle de réécriture de R correspondante.

On peut alors conclure qu'il est possible de prouver $x\colon 1 * [u] * 1$ dans le CCC avec les coercions a_i, r_i si et seulement si u se réécrit en v dans le système de Thue considéré, le typage est donc indécidable. □

4 Des Coercions dans les Types Simples

Les types dépendants étant trop complexes pour avoir un algorithme simple de typage prenant en compte les coercions implicites, intéressons nous pour l'instant au λ-calcul simplement typé, et posons-nous la question de savoir s'il est possible de définir une description des coercions intéressantes entre deux types. Ce genre de problème est abordé dans [17] avec certaines différences.

Nous avons décidé d'adopter un point de vue proche de celui de [15] et de faire intervenir la notion d'*isomorphisme de Curry-Howard*. Rappelons en quoi consiste ce concept pour le λ-calcul simplement typé : l'isomorphisme de Curry-Howard construit une bijection entre les propositions de la logique implicative minimale et les types d'une part, et entre les preuves de ces propositions et les termes des types correspondants d'autre part.

Rappelons les règles de la logique minimale :

$$\frac{}{\Gamma, A \vdash A}\ \mathbf{Ax}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \mathbf{Abs}$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \mathbf{Cut}$$

Sous cet isomorphisme, les coercions définies par l'utilisateur sont vues comme des *hypothèses* qui constituent un contexte $\mathcal{C}$ et construire une coercion du terme t de type A vers un terme de type B revient alors à donner une démonstration de

$$\mathcal{C}, A \vdash B$$

Dans une logique à définir, mais dont les règles doivent être admissibles en logique minimale implicative. Il sera alors possible, grâce à l'isomorphisme, de reconstruire un terme t' de type B à partir du terme t, c'est ce terme qui sera le coercé de A vers B. De quoi doit avoir l'air cette logique ? Certaines conditions s'imposent.

1. L'hypothèse A doit forcément être "consommée" une et une seule fois par la preuve. Il est en effet impensable qu'une coercion d'un terme t de type A vers un terme t' de type B ne fasse pas intervenir t. En particulier, si une des coercions f est de type $U \to V$ et $B \equiv U \to V$ alors le terme t' ne doit pas être constitué du seul terme f.
2. L'ordre des arguments ne doit pas être dérangé : si par exemple $A \equiv U \to U \to V$, $B \equiv U \to U \to V'$ et qu'il y a dans $\mathcal{C}$ une coercion f de type $V \to V'$, le terme t' formé ne doit pas être $\lambda x\colon U.\lambda y\colon U.f\ (t\ y\ x)$, mais $\lambda x\colon U.\lambda y\colon U.f\ (t\ x\ y)$.

3. Il ne doit pas non plus être possible de "dupliquer" l'hypothèse A. Nous ne voulons pas, par exemple, former de coercion entre les types $A \to A \to B$ et $A \to B$ sans avoir défini de coercion appropriée.

4.1 les règles de la logique des coercions

Certaines de ces conditions sont remniscientes de la *logique linéaire* [14] et de la *relevance logic*[1]. Cependant ces logiques sont encore un peu trop expressives. Après plusieurs tentatives, nous avons choisi de définir le système suivant :

Nous séparons le contexte en deux parties, une partie contient les coercions, l'autre contient une unique hypothèse, elle correspond au type du terme dont on veut donner une coercion. On définit :

Définition 4 *Une* proposition *est un terme sur la signature* $\mathcal{P} := \mathcal{V} \mid \mathcal{P} \to \mathcal{P}$ *Avec* $\mathcal{V}$ *un ensemble de variables.*

Définition 5 *Un* contexte *est un couple* $\langle \mathcal{C}, H \rangle$ *avec* $\mathcal{C}$ *un ensemble de propositions et* H *une proposition.*

On notera $\mathcal{C}; H$ au lieu de $\langle \mathcal{C}, H \rangle$ et $\mathcal{C}, T$ au lieu de $\mathcal{C} \cup \{T\}$.

On définit ensuite la *logique des coercions* $\vdash_{\mathbf{coer}}$ avec les règles suivantes :

$$\frac{}{\mathcal{C}; H \vdash_{\mathbf{coer}} H}\ \mathbf{Ax}$$

$$\frac{\mathcal{C}, A \to B; H \vdash_{\mathbf{coer}} A}{\mathcal{C}, A \to B; H \vdash_{\mathbf{coer}} B}\ \mathbf{App}$$

$$\frac{\mathcal{C}; C \vdash_{\mathbf{coer}} A \qquad \mathcal{C}; B \vdash_{\mathbf{coer}} D}{\mathcal{C}; A \to B \vdash_{\mathbf{coer}} C \to D}\ \mathbf{Contr}$$

$$\frac{\mathcal{C}; A \vdash_{\mathbf{coer}} B \qquad \mathcal{C}; B \vdash_{\mathbf{coer}} C}{\mathcal{C}; A \vdash_{\mathbf{coer}} C}\ \mathbf{Cut}$$

Nous écrirons $\vdash$ au lieu de $\vdash_{\mathbf{coer}}$ lorsque le contexte sera clair.

La première règle permet de former la coercion identité. On remarque qu'elle ne s'applique que à la partie droite du contexte : c'est ceci qui permet de remplir la première condition exprimée ci-dessus. La seconde règle est la règle fondamentale, c'est elle qui permet d'appliquer les coercions. La troisième règle permet d'obtenir la propriété de contravariance, sa forme nous donne les garanties de la seconde et la troisième condition ci-dessus.

Enfin la coupure garantit la transitivité de la relation. Il n'est pas évident que cette règle soit nécessaire ; en effet, beaucoup de logiques admettent un théorème d'elimination des coupures, c'est à dire que la règle **Cut** est admissible dans cette logique. Ici cependant la règle n'est pas admissible ; ceci vient d'un manque de symétrie dans les règles, il n'est pas possible de les appliquer à gauche du $\vdash$ comme dans un calcul des séquents. Il est possible de compléter les trois autres règles afin d'avoir un véritable calcul des séquents et de prouver l'élimination des coupures, ce sera l'objet de la section 5. Ceci est nécessaire pour prouver la décidablilité de la déduction dans cette logique.

4.2 la sémantique

Examinons la forme des termes produits par ces règles sous les lumières de l'isomorphisme de Curry-Howard : les règles correspondantes avec les annotations de λ-termes sont

$$\frac{}{\mathcal{C}; t\colon H \vdash t\colon H}\ \mathbf{Ax}$$

$$\frac{\mathcal{C}, f\colon A{\to}B; t\colon H \vdash u\colon A}{\mathcal{C}, f\colon A{\to}B; t\colon H \vdash (fu)\colon B}\ \mathbf{App}$$

$$\frac{\mathcal{C}; x\colon C \vdash a\colon A \qquad \mathcal{C}; y\colon B \vdash d\colon D}{\mathcal{C}; t\colon A{\to}B \vdash \lambda x\colon C.d[y{\leftarrow}(ta)]\colon C{\to}D}\ \mathbf{Contr}$$

$$\frac{\mathcal{C}; x\colon A \vdash b\colon B \qquad \mathcal{C}; y\colon B \vdash c\colon C}{\mathcal{C}; x\colon A \vdash c[y{\leftarrow}b]\colon C}\ \mathbf{Cut}$$

Donnons un exemple. Soit A une variable de type et $\mathcal{C}$ l'ensemble de coercions $\{A{\to}(A{\to}A)\}$. Nous voulons montrer $\mathcal{C}; A \vdash A{\to}A{\to}A$. Nous pouvons effectuer la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{}{\mathcal{C}; A \vdash A}\ \mathbf{Ax}}{\mathcal{C}; A \vdash A{\to}A}\ \mathbf{App} \qquad \dfrac{\dfrac{}{\mathcal{C}; A \vdash A}\ \mathbf{Ax} \qquad \dfrac{}{\mathcal{C}; A \vdash A}\ \mathbf{Ax}}{\mathcal{C}; A{\to}A \vdash A{\to}A{\to}A}\ \mathbf{App}}{\mathcal{C}; A \vdash A{\to}A{\to}A}\ \mathbf{Contr}\ \mathbf{Cut}$$

En rajoutant les annotations, on obtient le terme suivant :

$$\phi\colon A{\to}(A{\to}A); t\colon A \vdash \lambda x\colon A.\lambda y\colon A.\phi[(\phi t)x]y\colon A{\to}A{\to}A$$

Cela correspond au terme que l'on désire avoir lorsqu'une coercion est éffectuée entre A et $A{\to}A{\to}A$. On remarque que, si on efface les annotations de type et la fonction ϕ, on obtient le terme (non typé) $\lambda x.\lambda y.txy{\to}_\eta t$. Il semble

naturel que, une fois effacée l'information de typage ainsi que les coercions employées, le terme obtenu soit le terme de départ, modulo η-équivalence. Nous verrons qu'il en est ainsi (théorème 1) de toutes les coercions construites dans notre système.

Dans toute la suite, soit $\mathcal{C}$ un ensemble de coercions (variables de type $U \rightarrow V$), t une variable de type A, et u un terme de type B.

Lemme 1 *Si dans la logique des coercions $\mathcal{C}; t\colon A \vdash u\colon B$ alors toute occurrence d'une variable de coercion $f \in \mathcal{C}$ dans u est en partie gauche d'une application. De plus u est de la forme $\lambda x\colon T.u'$ ou $(f\ u')$ avec $f \in \mathcal{C}$.*

Démonstration
Induction évidente sur la dérivation de u. □

Lemme 2 *Avec les notations précédentes, si $\mathcal{C}; t\colon A \vdash u\colon B$ alors l'unique occurrence de t dans u est soit à gauche soit á droite d'une application, soit $u = t$.*

Démonstration
Encore une induction immédiate sur la dérivation de u. □

Définition 6 *On définit* l'effacement $\mathit{eff}\colon \Lambda \rightarrow \Lambda^{\boldsymbol{pur}}$ *inductivement sur la structure d'un terme de la façon suivante :*

- $\mathit{eff}(x) = x$ *si x est une variable*
- $\mathit{eff}(\lambda x\colon A.t) = \lambda x.\mathit{eff}(t)$
- $\mathit{eff}(ft) = \mathit{eff}(t)$ *si $f \in \mathcal{C}$, $(\mathit{eff}(f)\mathit{eff}(t))$ sinon*

Nous pouvons alors énoncer le théorème évoqué ci-dessus :

Proposition 1 *Supposons que $\mathcal{C}; t : A \vdash u : B$. Alors* l'effacement $\mathit{eff}(u)$*est η-réductible à t.*

Tout terme de cette forme seras dit sous *forme coercive.*

Démonstation : Par induction sur la dérivation $\mathcal{C}; t : A \vdash u : B$.

- Si la dernière règle utilisée est **Ax** alors $t = u$ et donc $\mathit{eff}(u) \twoheadrightarrow_\eta t$.
- Si la dernière règle utilisée est **App** alors $u = (fu')$ et on a $\mathit{eff}(u) = \mathit{eff}(u') \twoheadrightarrow_\eta t$.
- Si la dernière règle est **Contr** alors $u = \lambda x : T.u'(tu''(x))$ et donc par hypothèse d'induction $\mathit{eff}(u) \twoheadrightarrow_\eta \lambda x.tx \rightarrow_\eta t$.
- Si la dernière règle est **Cut** alors le résultat est clair.

□

On a même une réciproque :

Proposition 2 *Soit un λ-terme typé u avec une variable libre $t\colon A$ tel que :*

1. *le terme u est de type B dans le contexte $\mathcal{C}, t\colon A$*
2. *u est de forme coercive, c'est à dire $\mathit{eff}(u) \twoheadrightarrow_\eta t$*

alors il existe une déduction $\mathcal{C}; t\colon A \vdash u\colon B$ dans la logique des coercions.

Démonstration : Nous raisonnons par induction sur la structure du terme u. Le terme u est soit t, soit une abstraction, soit une application.

- Si $u = u_1 u_2$ est une application, alors $\mathit{eff}(u) \twoheadrightarrow_\eta t$ implique $u_1 = f$ pour un certain $f \in \mathcal{C}$. Par hypothèse d'induction, on peut prouver le jugement de typage $\mathcal{C}; t\colon A \vdash u_2\colon B'$ et donc $\mathcal{C}; t\colon A \vdash fu\colon B$ par application de la règle **App**.
- Si $u = \lambda x\colon T.u'$ alors on a $\mathit{eff}(u') \twoheadrightarrow_\eta tx$ et donc u' est de la forme $u_1(u_2(t)u_3(x))$ avec u_1, u_2, u_3 sous forme coercive. On peut alors appliquer l'hypothèse d'induction pour construire $\lambda x\colon T.u_1(t\ u_3(x))$ grâce à **Contr**, puis utiliser **Cut** pour construire u.

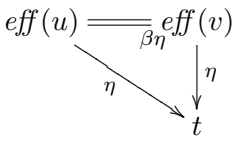
□

Ceci nous permet de comprendre la structure des coercions que produit notre système logique. Cette structure est stable par réduction β et η, et les termes produits sont fortement normalisants :

Lemme 3 *Si avec les notations précédentes u est sous forme coercive alors*

1. *u est fortement normalisant*
2. *pour tout λ-terme v, $u =_{\beta\eta} v \Rightarrow$ v est sous forme coercive.*

Démonstration : Le terme u étant typable dans le λ-calcul simplement typé avec le contexte $\mathcal{C}, t\colon A$, il est fortement normalisant d'après le théorème de normalisation forte (voir [4]). D'autre part, supposons que l'on ait $\mathit{eff}(u) \twoheadrightarrow_\eta t$ avec t une variable. Alors, grâce à la propriété de Church-Rosser de la $\beta\eta$-réduction du λ-calcul [2], on a pour tout terme v $\beta\eta$-équivalent à u :



Ce qui montre la seconde partie du lemme.

5 L'Elimination des Coupures

Il reste maintenant à donner une procédure de décision, ce qui permettra, étant donné $\mathcal{C}, A, B$ de voir si il existe une coercion de A vers B. Pour cela, il est nécessaire d'éliminer la règle **Cut**, car celle ci peut s'appliquer à chaque étape d'une preuve et rend donc difficile la recherche d'un algorithme de décision. Nous l'avons précisé, il faut pour cela rajouter des règles à notre

logique pour donner un symétrique à droite de chaque règle qui s'applique à gauche, c'est à dire **App** et **Contr**, la règle **Ax** étant déja symétrique.

On rajoute donc les règles suivantes (annotées) à **Coer** :

$$\frac{\mathcal{C}, f : A \to B; t : B \vdash c : C}{\mathcal{C}, f : A \to B; u : A \vdash c[t \leftarrow (f\,u)] : C}\ \textbf{Appin}$$

Avec u une variable *fraîche* c'est à dire non liée dans c.

$$\frac{\mathcal{C}'; x : A \vdash d : D \qquad \mathcal{C}'; y : E \vdash b : B \qquad \mathcal{C}'; z : C \vdash h : H}{\mathcal{C}'; t : D \to E \vdash h\,[z \leftarrow (f\ \lambda x : A.b[y \leftarrow (td)])] : H}\ \textbf{Contrin}$$

Avec $\mathcal{C}' = \mathcal{C}, f : (A \to B) \to C$.

On veut alors montrer :

Théorème 3 *Soit Δ une dérivation du séquent $\mathcal{C}; t : A \vdash u : B$. Il existe une dérivation Δ' du séquent $\mathcal{C}; t : A \vdash u' : B$ avec u' la forme normale de u qui n'utilise pas la règle **Cut**.*

Nous allons utiliser notre connaissance de la forme du terme u pour prouver ce théorème. Le théorème de "subject reduction" du lambda calcul simplement typé nous assure que tout terme coercif, si il est mis sous forme normale, garde le même type que celui de départ. Etant donné une coercion u, on peut la normaliser d'après le lemme 3, puis construire le terme normal dans la logique des coercions en utilisant la proposition 2. Il suffit donc de démontrer le théorème ci-dessus pour les termes u en forme β-normale.

Lemme 4 *Supposons, avec les notations précédentes, que $u = c(t)$ et $c(t) = c_0(d_0(t))$, avec c_0 et d_0 sous forme coercive. Alors si on peut construire $c_0(x)$ et $d_0(t)$ sans la règle **Cut**, et que $c(t)$ est en forme normale, on peut construire le terme $c(t)$ sans la règle **Cut**.*

Démonstration : Nous procédons par induction sur la taille du terme d_0. Si $d_0(t) = t$ alors $c(t) = c_0(t)$ peut être construit sans coupure par hypothèse. Sinon d'après la forme des règles d'inférence de la logique des coercions, $d_0(t)$ peut être de deux formes possibles :

- $d_0(t) = (f d_1(t))$ avec $f \in \mathcal{C}$ d'après le lemme 1. On peut alors appliquer la règle **appin** au terme $c_0(t)$ pour obtenir le terme $c_0(ft)$ sans coupure. On peut alors appliquer l'hypothèse d'induction appliquée à d_1 et $c_0(ft)$ pour obtenir le terme $c(t) = c_0(f d_1(t))$
- $d_0(t) = \lambda x : T.d_1(d_2(t)d_3(x))$. Nous pouvons en effet observer que la seule façon de construire un terme de la forme $\lambda x : T.u$ est par application de la règle **Contr** puis d'application des seules règles **Appin** et **Contrin**. Après application de la règle **Contr** le terme est égal à

$\lambda x : T.d_1(t\ d_3(x))$, puis les règles **Appin** et **Contrin** n'effectuent que des remplacements sur la variable t.
Alors d'après le lemme 2, soit $c_0(t) = t$ et on peut conclure, soit l'occurrence de t dans $c_0(t)$ est en partie droite d'une application, soit elle est en partie gauche. Elle ne peut en fait pas être en partie gauche d'une application, car alors $c_0(d_0(t))$ contiendrait un redex enjendré par l'abstraction dans d_0. Cette occurrence est donc en partie droite d'une application, et d'après le lemme 1, on peut écrire $c_0(t) = c_1(ft)$ pour un certain $f \in \mathcal{C}$. Il est donc possible d'appliquer la règle **contrin** pour construire le terme

$$c_2(t) = c_1(f\ \lambda x : T.d_1(t\ d_3(x)))$$

On peut alors conclure grâce à l'hypothèse d'induction appliquée à d_2 et c_2.

Ceci nous permet de démontrer le théorème 3 : par induction évidente sur la taille du terme en utilisant le lemme 4 $\square$
A partir de maintenant, nous noterons $A \vdash B$ au lieu de $\mathcal{C}; A \vdash B$ si l'ensemble de coercions n'est pas ambigu.

6 L'Algorithme d'Inférence de Type

Nous pouvons maintenant remarquer la chose suivante : en regardant les règle de déduction du bas vers le haut (*bottom-up*) la taille des types diminue pour les règles **Contr** et **Contrin**, et elle est bornée par la taille des types dans $\mathcal{C}$ pour les règles **App** et **Appin**. Seule la règle **Cut** posait problème, il suffit d'appliquer le théorème 3 pour s'en débarasser. Lorsque nous voulons démontrer un séquent de la forme $\mathcal{C}; A \vdash B$ en appliquant les règles ci-dessus, les types apparaissant dans les séquents seront de taille bornée. Nous nous déplaçons donc dans un ensemble fini de séquents possibles, ceci nous donne un moyen de décider cette logique.

6.1 La Décidabilité de *coer*

Etant donné un ensemble de coercions $\mathcal{C}$ on peut donner l'algorithme suivant pour décider, étant donné deux types A et B si $\mathcal{C}; A \vdash B$ et, le cas échéant, donner le terme de coercion associé :

$\textsc{Colog}(A\colon \textsc{type}, B\colon \textsc{type}, L\colon \textsc{seqlist}, t\colon \textsc{var})\colon \textsc{term}$

if $\langle A, B\rangle \in L$
then return FAIL
else $L \leftarrow \langle A, B\rangle :: L$
try
Ax$(A = B)$
return t
App$(f\colon C \rightarrow B \in \mathcal{C})$
return $f(\textsc{Colog}(A, C, L, t))$
Contr$(A = A_1 \rightarrow A_2, B = B_1 \rightarrow B_2)$
return let $y = (t\ \textsc{Colog}(B_1, A_1, L, x))$ **in** $\lambda x\colon B_1.\textsc{Colog}(A_2, B_2, L, y)$
Appin$(g\colon A \rightarrow C \in \mathcal{C})$
return let $y = (g\ t)$ **in** $\textsc{Colog}(C, B, L, y)$
Contrin$(A = A_1 \rightarrow A_2, f : (B_1 \rightarrow B_2) \rightarrow C \in \mathcal{C})$
return
let $z =$
$(\textbf{let}\ y = (t\ \textsc{Colog}(B_1, A_1, L, x))\ \textbf{in}\ (f\ \lambda x\colon B_1.\textsc{Colog}(A_2, B_2, L, y)))$
in $\textsc{Colog}(C, B, L, z)$

On décide alors si $\mathcal{C}; A \vdash B$ en appelant `Colog`$(A, B, [], t)$, si c'est le cas alors l'algorithme renvoie un terme de preuve, sinon il renvoie `FAIL`.

Le **try** dans cette procédure n'est pas déterministe : il faut essayer toutes les possibilités afin de construire un *arbre d'appels récursifs*. Une branche de cet arbre est alors *élagué* si un appel se termine en échec (c'est le *fail*). On veut alors récuperer les noeuds non coupés de cet arbre. Cette méthode de programmation peut rappeler les méthodes déclaratives présentes dans le langage `PROLOG` par exemple.

Il n'est pas évident que cet algorithme termine. Cependant, la terminaison est garantie par le fait que la taille des types d'entrée dans les appels de la procédure sont bornées, et que la liste passée en argument grandit strictement à chaque appel. Il arrive donc un endroit dans chaque branche dans lequel soit le séquent $\langle A, B\rangle$ est présent dans la liste, soit $A = B$. L'arbre des appels est donc à chemins finis et chaque arrête à un nombre de fils fini, l'arbre est donc fini, l'algorithme termine.

6.2 L'Algorithme de Typage

Nous voulons maintenant, étant donné un terme dans le λ-calcul simplement typé et un ensemble de coercions, donner un algorithme de typage pour déterminer si le terme est typable dans notre système de types avec coercions et si c'est le cas, donner son type.

Remarquons d'abord que notre système est suffisamment puissant pour modéliser la surcharge de fonctions. Par exemple, considérons deux fonctions,

$+_{\mathbf{R}}: \mathbf{R}\to\mathbf{R}\to\mathbf{R}$ et $+_{\mathbf{N}}: \mathbf{N}\to\mathbf{N}\to\mathbf{N}$ que nous voulons dénoter avec un seul symbole de fonction, $+$. Il suffit alors de créer un type "intersection" I_+ habité par le seul terme $+$ ainsi que deux coercions $c_1: I_+\to\mathbf{R}\to\mathbf{R}\to\mathbf{R}$ et $c_2: I_+\to\mathbf{N}\to\mathbf{N}\to\mathbf{N}$ qui à $+$ associent $+_{\mathbf{R}}$ et $+_{\mathbf{N}}$ respectivement. I_+ étant un type atomique n'apparaissant nulle part d'autre que dans le type du terme $+$ il est clair que dans tout terme bien typé contenant $+$ appliqué à des arguments une des deux coercions aura étée utilisée. Notons que I_+ peut être vu comme le type $\mathbf{N}\to\mathbf{N}\to\mathbf{N}\wedge\mathbf{R}\to\mathbf{R}\to\mathbf{R}$, ce qui permet de donner une idée de l'intérêt d'étendre la logique des coercions avec des connecteurs $\wedge$, $\vee$ etc, ou une extension du λ-calcul du type de celui étudié dans [7].

Pour typer un terme du lambda calcul simplement typé, il est nécessaire de procéder inductivement sur la structure du terme. Dans le cas sans coercions (et sans inférence de types) l'algorithme est simple : dans le cas d'une abstraction $\lambda x: A.t$, il suffit de typer le terme t avec le contexte $x: A$ et de renvoyer le type $A\to B$ si t est de type B. Dans le cas d'une application $(t\ u)$, il faut typer t et u. Le terme est bien typé dans le cas où le type de t est de la forme $A\to B$ et le type de u est A. Le type du terme est alors B.

Les choses sont évidemment plus compliquées en présence de coercions, un terme pouvant avoir éventuellement une infinité de types possibles. Par exemple dans le cas d'une unique coercion $f: A\to(A\to A)$, tout terme t auquel on peut attribuer le type A peut également être attribué le type $A\to A$, $A\to A\to A$, $A\to A\to A\to A$, etc.

Il n'est donc, *a-priori*, pas possible d'énumérer tous les types possibles des sous-termes d'un terme afin de pouvoir choisir les types permettant de typer le terme complet. Il faut cependant chercher à effectuer cette énumération, car en effet il peut arriver que la première coercion trouvée ne soit pas la bonne pour pouvoir typer le terme complet. Par exemple, si on considère l'ensemble de coercions $\{c_1: H\to A\to B, c_2: H\to A\to A\to B\}$ et le terme $f\ a_1\ a_2$ avec $f: H$ et $a_1, a_2: A$ alors on peut typer le sous terme $f\ a_1$ grâce à la coercion c_1 pour lui donner le type B. Il est alors impossible de typer le terme complet sans revenir en arrière pour donner un type différent au terme $f\ a_1$, on appelle cela le *backtracking*.

Une solution naïve serait de chercher à typer une série d'applications en une étape, c'est à dire typer a_1 et a_2 (ici seul le type A est possible) et ensuite chercher a trouver un type pour f de la forme $A\to A\to X$ pour X un certain type. Cependant ceci pose encore problème, comme le montre l'exemple suivant :

On se donne les coercions $c_1: I_+\to(N\to N\to N), c_2: I_+\to(D\to D\to R)$, $c_3: N\to D$ et on se donne les variables $+: I_+, \times: R\to R\to R, n: N, m: N, r: R$. Nous voulons typer le terme $\times\ (+\ n\ m)\ r$. Pour typer ce terme avec l'idée ci-dessus, nous cherchons à typer $+\ n\ m$, ce qui est possible et donne le type N puis nous cherchons une coercion de $R\to R\to R$ vers $N\to R\to X$ pour un certain X. Cette coercion n'existe pas. Il faut donc un algorithme qui permette de backtracker afin d'essayer tous les types possibles pour les arguments des

fonctions.

Il existe en général une infinité de types possibles dans ce cas. Cependant, il suffit de considérer un nombre fini de types grâce au lemme suivant :

Lemme 5 *Soit $\mathcal{C}$ un ensemble de coercions et F un type. Soit I un ensemble d'indices ($I \subseteq \mathbb{N}$) et $(F_1^i)_{i\in I}$ et $(F_2^i)_{i\in I}$ l'ensemble des types tels que $F \vdash_{\mathbf{coer}} F_1^i \to F_2^i$.*

Il existe $J \subset I$ fini *tel que $\forall i \in I, \exists j \in J$ tel que*

$$F_1^i \vdash_{\mathbf{coer}} F_1^j$$

et

$$F_2^j \vdash_{\mathbf{coer}} F_2^i$$

Nous pouvons même donner explicitement la famille $(F_\epsilon^j)_{j\in J,\epsilon=1,2}$: On peut évidemment supposer qu'il existe un entier naturel n tel que $J = \{k \in \mathbb{N} \mid k \leq n\}$. Si $F \equiv A \to B$ alors $F_1^1 = A$ et $F_2^1 = B$; pour tout $j > 1$ les F_1^j sont les types V et les F_2^j sont les types W tels que $\exists f \in \mathcal{C}$ avec $f : U \to (V \to W)$; si F est atomique, alors les F_1^j et F_2^j sont de la deuxiemme forme pour tout $j \in J$.

Les F_ϵ^j ne dépendent donc que de $\mathcal{C}$, à part éventuellement F_1^1 et F_2^1 qui peuvent dépendre de F.

Démonstration : Nous démontrons le lemme par induction sur la longeur de la dérivation de $F \vdash F_1^i \to F_2^i$ dans le système avec les trois règles **Ax**, **App**, **Contr** et la coupure :

- Cas **Ax** : $F_1^1 \to F_2^1 = F = F_1^i \to F_2^i$, donc $F_1^i \vdash F_1^1$ et $F_2^1 \vdash F_2^i$ de façon triviale.
- Cas **App** : $\exists f \in \mathcal{C}$ tel que $f : H \to F_1^i \to F_2^i$, donc $i \in J$ et on peut conclure.
- Cas **Contr** : $F = F_1^1 \to F_2^1$ et on a $F_1^i \vdash F_1^1$ et $F_2^1 \vdash F_2^i$.
- Cas **Cut** : On a $F \vdash H$ et $H \vdash F_1^i \to F_2^i$. Par hypothèse d'induction appliquée à $H \vdash F_1^i \to F_2^i$, on a soit $H \equiv H_1 \to H_2$ et $F_1^i \vdash H_1$, soit $H \equiv H_1 \to H_2$ et $\exists j \in J, j > 1$ tel que $F_1^i \vdash H_1^j$, soit H atomique et $\exists j \in J$ tel que $F_1^i \vdash H_1^j$. Dans les deux derniers cas, il existe $j' \in J$ ($j' = j$ ou $j + 1$) tel que $F_1^{j'} = H_1^j$.

  Il suffit donc de traiter le premier cas. Dans ce cas, par hypothèse d'induction appliquée à $F \vdash H_1 \to H_2$, il existe $k \in J$ tel que $H_1 \vdash F_1^k$ et donc $F_1^i \vdash F_1^k$ par transitivité du sous-typage. On vérifie sans difficultés que $F_2^{j'} \vdash F_2^i$.

L'algorithme proposé est le suivant : On se donne un contexte avec une liste `VAR` de variables x indexées par leur type T_x. Etant donné un type T, les éléments des familles finies $(T_1^j)_{j\in J}$ et $(T_2^j)_{j\in J}$ définies dans le lemme précédent seront notées, pour $j \in J$, $\mathtt{Ar}_1^j(T)$ et $\mathtt{Ar}_2^j(T)$ respectivement.

$\textsc{Cotype}(t\colon \textsc{term}, \textsc{var})$

match t **with**

$x \in \textsc{var}$ **return** T_x

$\lambda x\colon T.t'$ **return** $T \to \textsc{Cotype}(t', \langle x, T\rangle :: \textsc{var})$

$(t_1\ t_2)$ **try**

$local = \textsc{Colog}(\textsc{Cotype}(t_2, \textsc{var}), \mathtt{Ar}_1^j(\textsc{Cotype}(t_1, \textsc{var})), [], t)$ **with** $j \in J$

if $local \neq \textsc{fail}$ **return** $\mathtt{Ar}_2^j(\textsc{Cotype}(t_1, \textsc{var}))$

else return $\textsc{fail}$

Le **try** ici est encore non déterministe : il essaye toutes les possibilités pour $j \in J$. C'est en ceci que l'algorithme effectue du *backtracking*. La correction de cet algorithme est facile à vérifier. Il faut vérifier que cet algorithme est *complet* c'est à dire qu'un terme est typable en présence de coercions seulement si l'algorithme renvoie un résultat (différent de `FAIL`). La complétude de cet algorithme est garantie par le lemme 5, pour typer $(t_1\ t_2)$, il suffit de vérifier que le type de l'argument se coerce vers un des F_1^j pour déterminer si l'application est typable, pour tous les types possibles de t_1 ; en effet si il existe une coercion du type de t_1 vers un type de la forme $T_{t_2} \to X$ avec T_{t_2} un type possible de t_2. Par le lemme, si un tel type existe, alors $T_{t_2} \vdash F_1^j$ pour un certain $j \in J$. L'ensemble J étant toujours fini, nous pouvons énumérer tous les cas.

7 La Cohérence

Lorsqu'on introduit un système de coercions, il est nécessaire de garantir que toute coercion entre deux types A et B est unique, c'est à dire

$$\forall c_1, c_2\colon A \to B \text{ coercions},\ c_1 =_{\beta\eta} c_2$$

Ceci est clairement faux dans notre système, en effet il est possible de définir $\mathcal{C} = \{f\colon A \to B, g\colon A \to B\}$ ou encore $\mathcal{C} = \{f\colon A \to A\}$ dans lequel cette unicité n'est pas présente. Il est cependant intéressant de se demander quelles limitations sur les éléments de $\mathcal{C}$ permettent d'avoir cette propriété.

Définition 7 *Soit $\mathcal{C}$ un ensemble de coercions. On dit que $\mathcal{C}$ est* cohérent *si pour tout types U, V et toute dérivation $t\colon U \vdash u_1\colon V$ et $t\colon U \vdash u_2\colon V$ dans la logique des coercions, alors $u_1 =_{\beta\eta} u_2$.*

Nous conjecturons ce résultat initial :

Conjecture 1 *Supposons que $\mathcal{C} = \{f\colon A \to B\}$ et que $A \not\equiv B$. Alors $\mathcal{C}$ est cohérent.*

La démonstration de cette proposition nous échappe pour l'instant, elle semble cependant accessible ; dans le cas oú A n'est pas un sous terme de B qui n'est lui-même pas un sous terme de A, le résultat découle du résultat de cohérence dans [15].

Une conjecture un peu plus ambitieuse dont la vérité est moins certaine peut se formuler ainsi :
On considère un ensemble $\mathcal{C}$ de coercions, et on remplace les coercions de la forme $f : (A \to B) \to (C \to D)$ par les coercions $f_1 : C \to A$ et $f_2 : B \to D$ ainsi de suite jusqu'à qu'il n'y ait plus que des coercions de type $A \to B$ ou $A \to (B \to C)$ ou $(A \to B) \to C$ avec A, B, C atomiques. On note $\mathcal{C}^*$ l'ensemble de coercions ainsi obtenu et on regarde $\mathcal{C}^*$ comme un graphe dans lequel les sommets sont les sources et buts des coercions et les arrêtes sont les coercions. On formule alors la conjecture :

Conjecture 2 *$\mathcal{C}$ est cohérent si et seulement si $\mathcal{C}^*$ est* sans cycles.

La seconde conjecture implique trivialement la première, et il est clair que $\mathcal{C}^*$ sans cycles est une condition neccesaire (tout cycle dans $\mathcal{C}^*$ permet de construire deux coercions distinctes). Il suffirait donc de montrer qu'elle est suffisante. Certains résultats concernant la cohérence ont étés démontrés dans [6] et [8], ils se généralisent peut- être à la situation présente.

Il peut également être intéressant d'étudier la situation lorsqu'il existe des équations entre coercions, par exemple $f \circ g = id$ avec $f, g \in \mathcal{C}$ et id l'identité, ou d'essayer de voir quelles équations il faut rajouter à un ensemble de coercions afin de le rendre cohérent. Ces questions sont la continuation logique du présent travail.

8 Conclusion

Nous avons détaillé un système de sous-typage dans le cas du λ-calcul simplement typé, et montré la décidabilité de la relation de sous typage, et donné un algorithme de typage en présence de coercions. Nous avons également montré l'indécidabilité de typer un terme dans un cadre plus général. Il est cependant désirable de donner des méthodes de sous-typage coercif dans des cadres plus généraux ainsi que des systèmes plus puissants de coercions, par exemple une infinité de coercions.

Ce types de généralisations a déja été étudié (par exemple dans [6] et [8]) avec des résultats intéressants, mais le typage devient rapidement indécidable dès que le système de coercions devient trop expressif (voir [13]). Une extension possible de notre travail serait d'essayer d'incorporer du polymorphisme ou de la dépendance dans notre logique ou encore des types inductifs, par exemple en s'appuyant sur les travaux dans [12],[16].

Un autre axe intéressant serait d'essayer de donner une extension du système de typage en présence de *types implicites* inférés à partir des arguments explicites des fonctions. Un exemple important est le problème suivant : typer le terme $eq\ x\ y$ avec $eq\colon \forall T\colon Type, T \to T \to Prop$, $x\colon A$, $y\colon B$ et une coercion $c\colon A \to B$. Les algorithme de typages habituels infèrent le type T grâce au type de x, et tentent alors de typer $(eq\ A)\ x\ y$, ce qui est impossible. Pour pouvoir typer ce terme, il faudrait un système qui résolve les inéquations $A \leq T, B \leq T$ dans le système de sous typage défini par les coercions (la solution ici serait $T = B$). Ce genre de situation semble (à notre connaissance) échapper aux études actuelles sur le sous-typage coercif.

Références

[1] A. R. Anderson and N. D. Belnap. *Entailment. The Logic of Relevance and Necessity, Volume 1.* U.S.A., 1975.

[2] H. Barendregt. *The Lambda Calculus : Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, revised edition, 1984.

[3] H. Barendregt. Introduction to Generalised Type Systems. 1(2) :125–154, April 1991.

[4] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford Science Publications, 1992. Volume 2.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development— Coq'Art : The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. 2004.

[6] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. 93 :172–221, 1991.

[7] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. 117(1) :115–135, February 1995.

[8] G. Chen. Subtyping calculus of constructions. In I. Prívara and P. Ruzicka, editors, *Proceedings of MFCS'97*, volume 1295, pages 189–198, 1997.

[9] Coq development team. *The Coq Proof Assistant Reference Manual, version 8.1*, 2006.

[10] T. Coquand and G. Huet. The Calculus of Constructions. 76(2/3) :95–120, February/March 1988.

[11] Martin Davis. *Computability and Unsolvability.* 1982.

[12] Gilles Dowek and Ying Jiang. Eigenvariables, bracketing and the decidability of positive minimal predicate logic. *Theor. Comput. Sci.*, 360(1-3) :193–208, 2006.

[13] Giorgio Ghelli. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139(1–2) :131–162, 1995.

[14] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50 :1–102, 1987.

[15] Longo, Milsted, and Soloviev. A logic of subtyping. In *LICS : IEEE Symposium on Logic in Computer Science*, 1995.

[16] Z. Luo. Coercive subtyping. 9(1) :105–130, February 1999.

[17] F. Pottier. A framework for type inference with subtyping. In *Proceedings of ICFP' 98*. ACM Press, 1998.

[18] A. Saïbi. Typing algorithm in type theory with inheritance. 24, 1997.

[19] J. G. Springintveld. Third-order matching in the polymorphic lambda calculus. In G. Dowek, J. Heering, K. Meinke, and B. Möller, editors, *Proceedings of the Second International Workshop on Higher-Order Algebra, Logic, and Term Rewriting (HOA '95)*, volume 1074, pages 221–238. Springer-Verlag, 1996.

[20] J. B. Wells. The undecidability of Mitchell's subtyping relation. Tech. Rep. 95-019, Comp. Sci. Dept., Boston Univ., December 1995.