Types Simples, Logique et Coercions Implicites

Cody Roux

Types Simples, Logique et Coercions Implicites

Cody Roux[1]

1: Laboratoire Lorrain de Recherche en Informatique et ses Applications,
LORIA - Campus Scientifique - BP 239 - 54506 Vandoeuvre-lès-Nancy Cedex, France
cody.roux@loria.fr

Résumé

Nous définissons un systeme général de coercions implicites dans les types simples et donnons un algorithme d'inférence de type. Ceci est réalisé grâce à une logique adéquate, inspirée de la logique linéaire, pour laquelle nous prouvons l'élimination des coupures.

1. Introduction

Les notions de coercion implicite et de sous-typage sont devenues importantes durant les dix dernières années, ceci à cause d'une motivation double. 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 est probable que tout résultat profond utilise ce genre de changement de point de vue. Les preuves mathematiques contiennent évidamment aussi énormément d'information implicite. 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*, mais il se trouve également l'enjeu de l'utilisation modulaire de mêmes morceaux de code.

La bonne compréhension des coercions implicites est donc nécessaire pour continuer à développer l'objectif actuel de formalisation. Nous presentons un système général de coercions implicites dans les types simples et donnons un algorithme d'inférence des types en présence de coercions. Ceci est réalisé en appliquant l'isomorphisme de Curry-Howard et en presentant le sous-typage comme un theoreme d'une certaine logique. L'élimination des coupures dans cette logique correspond à la preuve de transitivité de la relation de sous-typage et nous permet de donner un algorithme d'inférence du sous-typage. Il est alors possible de donner un algorithme de typage en présence de coercions, qui dépend d'une notion de "type principal". Nous donnons alors des idées de développements futurs, notamment en ce qui concerne la cohérence du système.

Ce travail est issu du travail de Master de l'auteur. Nous voulons remercier M. Loic Pottier pour son exellent travail d'encadrement, et toute l'equipe Marelle, en particulier Ioana Pasca pour son support.

2. La notion de sous-typage

Dorénavant, nous nous plaçons dans le λ-calcul simplement typé (à la Church) défini de la manière suivante :

- L'ensemble des types :
$$\mathcal{T} := \mathcal{V} \mid \mathcal{T} \rightarrow \mathcal{T}$$
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 :

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

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

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

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 $(\ldots (t\ u_1) \ldots) u_n$. Nous définissons de plus $t[x \leftarrow u]$ pour la substitution de la variable x par le terme u dans le terme t. Comme souvent dans de telles situations, nous adopterons la convention de Barendregt [2], qui permet de laisser toutes les questions d'α-conversion comme exercice au lecteur.

La définition du sous-typage peut s'exprimer comme une relation de préordre $\leq$ sur les types ainsi que la règle de typage suivante :

$$\frac{\Gamma \vdash t\colon A \qquad A \leq B}{\Gamma \vdash t\colon B}$$

Il est naturel d'imposer à cette relation d'être une relation de préordre. Cependant, cette relation peut très bien ne pas être antisymétrique ; on a alors $A \equiv B$ sans avoir pour autant $A = B$. Ceci ne pose en général pas de problème, sauf quand le sous-typage est défini en termes de coercions implicites, où il se pose la question de la *cohérence*. Nous en reparlerons dans la section 6.

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*.

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 polynôme 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 de la relation de sous-typage et à donner l'algorithme de typage correspondant. nous allons pour cela utiliser le concept de *coercion implicite*. Malheureusement, nous verrons qu'il n'est pas facile de donner une définition intéressante et effective.

3. 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 nous nous intéressons au λ-calcul simplement typé. Ce genre de problème est abordé dans [14] avec une restriction sur les jugements de sous-typages possibles : les contraintes de sous-typage ne peuvent se faire entre deux types construits, nous ne pouvons par exemple avoir $A \to B \leq C \to D$ sans avoir $C \leq A$ et $B \leq D$, nous pensons qu'il est intéressant de lever cette restriction, notamment pour les exemples issus de motivation mathematiques. Une présentation des algorithmes utilisés dans cette situation peut se trouver dans [15].

Nous avons décidé d'adopter un point de vue proche de celui de [11] 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.

Il est intéressant de noter que ces restrictions sont sujettes a discussion. Nous savons, par exemple, qu'un étudiant confronté à une notation fonctionnelle dont les arguments ont étés inversés peut nénmoins comprendre le sens de l'expression dans le cas où l'expression vue n'aurait pas de sens. Il est possible qu'un relachement des conditions ci-dessus permettent de simuler ce genre de comportement et donc de simplifier l'interaction d'un utilisateur avec un prouveur de théorèmes par exemple.

Il est également naturel de demander à la partie $\mathcal{C}$ du contexte de ne contenir que des coercions (définies par l'utilisateur par exemple), c'est à dire des types fonctionnels. Cependant, il est possible d'imaginer un contexte qui contient des types non fonctionnels, qui pourraient représenter des arguments à appliquer de façon implicite dès que possible, par exemple l'univers dans lequel on se place. Nous aurions alors, si U représente le type de cet univers, une coercion entre toutes les fonctions $U \to A$ et A. Encore une fois, ce genre de convention est adoptée de façon systematique en mathématiques. Nous n'explorons pas plus en avant ce point de vue dans la suite du papier, et dorénavent, on suppose que $\mathcal{C}$ ne contient que des arguments fonctionnels.

3.1. Les règles de la logique des coercions

Certaines de ces conditions sont remniscientes de la *logique linéaire* [9] 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 qui suit.

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

Définition 1 *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 2 *Un* contexte *est un couple* $\langle \mathcal{C}, H \rangle$ *avec* $\mathcal{C}$ *un ensemble de propositions de la forme* $P_1 \to P_2$ *et* P_1, P_2, H *des propositions.*

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 condition numéro 1 détaillée dans la section précédente. 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.

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 4. Ceci est nécessaire pour prouver la décidablilité de la déduction dans cette logique.

Nous pouvons faire la remarque suivante : la notation $\mathcal{C}; H$ n'est pas sans rappeler les systèmes à *bénitier* (voir par exemple [10]) dans lesquels une partie du séquent est mise à l'écart du reste des hypothèses. Le but de ces systèmes est de mieux gérer la recherche de preuve. Cependant nous n'avons ici aucune régle qui permet de "remplir" le bénitier, et nos motivations sont différentes. Nous reviendrons sur cette analogie plus tard.

Définition 3 *Nous dirons qu'un séquent est* dérivable *si il existe une preuve de ce séquent dans la logique des coercions.*

3.2. les termes

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 :

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

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) $\lambda x.\lambda y.txy \twoheadrightarrow_\eta t$. 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 $U{\to}V$), t une variable de type A, et u un terme de type B.

Lemme 1 *Si dans la logique des coercions $\mathcal{C}; t\colon A \vdash u\colon B$ est dérivable 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 $\mathcal{C}; t\colon A \vdash u\colon B$ est dérivable alors l'unique occurrence de t dans u est soit à gauche soit à droite d'une application, soit $u = t$.*

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

Définition 4 *On définit* l'effacement $\mathit{eff}\colon \Lambda{\to}\Lambda^{\mathbf{pur}}$ *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 la variable t.*
Tout terme de cette forme sera dit sous *forme coercive.*

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

- 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 $\mathit{eff}(u) \twoheadrightarrow_\eta \lambda x.tx{\to_\eta}t$.
- Si la dernière règle est **Cut** alors le résultat est clair.

□

On a même une réciproque :

Proposition 2 *Soit u un λ-terme typé 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 $\mathcal{C}; t\colon A \vdash u\colon B$ est dérivable.

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

- Si $u = u_1u_2$ 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.

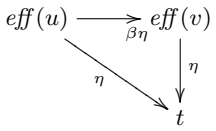
□

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, $u \twoheadrightarrow_{\beta\eta} v \Rightarrow$ 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 [3]). 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.

Remarquons que toute la puissance de la normalisation forte du λ-calcul simplement typé n'est pas neccesaire : les termes n'ayant ici qu'une seule occurence pour chaque variable, la normalisation est bien plus simple à montrer.

4. 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. Ceci est analogue à l'utilisation des symmetries pour montrer l'élimination des coupures dans le calcul des sequents de Genzen (mais de façon bien plus simple, puisque nous utilisons déja un théorème de normalisation forte).

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

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

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

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

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

Nous pouvons vérifier que ces règles sont admissibles dans la logique des coercions avec la coupure.

On veut alors montrer :

Théorème 1 *Soit un séquent $\mathcal{C}; t\colon A \vdash u\colon B$ dérivable. Il existe une dérivation du séquent $\mathcal{C}; t\colon A \vdash u'\colon B$ 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. Comme notre calcul défini est un sous-calcul du lambda calcul simplement typé, le théorème de "subject reduction" du lambda calcul simplement typé s'applique et 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 :

- $d_0(t) = (fd_1(t))$ 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 $c(t) = c_0(fd_1(t))$
- $d_0(t) = \lambda x : T.d_1(d_2(t)d_3(x))$. Nous pouvons en effet observer que la seule façon de construire un terme de la forme $\lambda x\colon T.u$ 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 1 : par induction sur la taille du terme en utilisant le lemme 4. □
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.

5. 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 1 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 construire une procédure de décision pour cette logique.

5.1. 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.

5.2. L'Algorithme de Typage

Supposons donné un λ-terme avec annotations de types et un ensemble de coercions. Nous voulons donner un algorithme qui détermine 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, $+_{\mathbf{R}}\colon \mathbf{R}\rightarrow\mathbf{R}\rightarrow\mathbf{R}$ et $+_{\mathbf{N}}\colon \mathbf{N}\rightarrow\mathbf{N}\rightarrow\mathbf{N}$ 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 $c_1\colon I_+\rightarrow\mathbf{R}\rightarrow\mathbf{R}\rightarrow\mathbf{R}$ et $c_2\colon I_+\rightarrow\mathbf{N}\rightarrow\mathbf{N}\rightarrow\mathbf{N}$ 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 $\mathbf{N}\rightarrow\mathbf{N}\rightarrow\mathbf{N} \wedge \mathbf{R}\rightarrow\mathbf{R}\rightarrow\mathbf{R}$, ce qui permet de donner une idée de l'intérêt d'étendre la logique des coercions avec des connecteurs $\wedge$, $\vee$ etc, ou d'utiliser une extension du λ-calcul du type de celui étudié dans [5].

Pour typer un terme du λ-calcul il est nécessaire de procéder inductivement sur la structure du terme. Dans le cas sans coercions (et sans inférence de types à la curry) l'algorithme est simple : dans le cas d'une abstraction $\lambda x\colon A.t$, il suffit de typer le terme t avec le contexte $x\colon A$ et de renvoyer le type $A\rightarrow B$ 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 $A\rightarrow B$ 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 $f\colon A{\to}(A{\to}A)$, 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 $\{c_1\colon H{\to}A{\to}B, c_2\colon H{\to}A{\to}A{\to}B\}$ et le terme $f\ a_1\ a_2$ avec $f\colon H$ et $a_1, a_2\colon A$ 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 $f\ a_1$, (faire du *backtracking*).

Une solution naïve serait de chercher à typer une série d'applications en une étape, c'est à dire (pour cet exemple) chercher à 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 $c_1\colon I_+{\to}(N{\to}N{\to}N), c_2\colon I_+{\to}(D{\to}D{\to}R), c_3\colon N{\to}D$ et on se donne les variables $+\colon I_+, \times\colon R{\to}R{\to}R, n\colon N, m\colon N, r\colon R$. 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 revenir en arrière 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 $f\colon U{\to}(V{\to}W)$; 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.

Ces types correspondent à une certaine notion de "type principal" c'est à dire qu'il suffit de connaitre ces types pour effectuer le typage.

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 $f\colon H{\to}F_1^i{\to}F_2^i$, 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.

COTYPE(t: TERM, VAR)
  match t **with**
    $x \in$ VAR **return** T_x
    $\lambda x\colon T.t'$ **return** $T \to$ COTYPE$(t', \langle x, T\rangle :: \text{VAR})$
    $(t_1\ t_2)$ **try**
    local = COLOG(COTYPE$(t_2, \text{VAR}), \mathtt{Ar}_1^j(\text{COTYPE}(t_1, \text{VAR})), [], t)$ **with** $j \in J$
    if *local* $\neq$ FAIL **return** $\mathtt{Ar}_2^j(\text{COTYPE}(t_1, \text{VAR}))$
    else return 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 5, 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.

6. 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 5 *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 où 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 [11].

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 $f\colon (A \to B) \to (C \to D)$ par les coercions $f_1\colon C \to A$ et $f_2\colon B \to D$ 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 [4] et [6], ils se généralisent peut-être à la situation présente.

Il est également remarquable que les systèmes à bénitier évoqués plus haut ont étés cr'e'es dans le but d'identifier certaines preuves $\beta\eta$-équivalentes. Certains résultats dans ce domaine pourraient donc peut-être s'appliquer dans cette situation.

Il peut aussi 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.

7. 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. 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 en présence d'une infinité de coercions.

Ce types de généralisations a déja été étudié (par exemple dans [4] et [6]) avec des résultats intéressants, mais le typage devient rapidement indécidable dès que le système de coercions devient trop expressif (voir [8]). 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 [7],[12].

Nous pensons que le point de vue du sous-typage comme une inférence logique peut potentiellement se généraliser afin d'intégrer des procédures de décision éventuellement complexes au système d'inférence de types, pour d'un coté alléger les définitions et faciliter les preuves dans un système de preuve interactif basé sur la théorie des types (Coq, Agda, Epigram...), et de l'autre faciliter la programmation dans les languages fortement typés (OCaml, Haskell...).

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: $\forall T$: $Type, T \to T \to Prop$, x: A, y: B et une coercion c: $A \to B$. 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.

Enfin, il serait possible d'étudier ce qui se passe dans le cas d'un typage à la Curry. L'absence d'information de types ou une information partielle peut permettre de définir une notion naturelle de sous-typage, comme celui détaillé dans [13]. La question est de savoir quelles sont les relations entre ce sous-typage et celui engendré par des coercions implicites.

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. 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.

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

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

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

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

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

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

[10] Hugo Herbelin Séquents qu'on calcule : de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes. thèse de doctorat, 1995.

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

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

[13] A. Miquel The Implicit Calculus of Constructions : Extending Pure Type Systems with an Intersection Type Binder and Subtyping In *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'01*, Krakow, Poland, 2–5 May 2001

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

[15] F. Henglein and J. Rehof The Complexity of Subtype Entailment for Simple Types. In *Logic in Computer Science*, pages 352-361, 1997