

Introduction à la modularité des calculs

Jean-Pierre Jouannaud

Laboratoire d'Informatique de l'École polytechnique
<http://www.lix.polytechnique.fr/~jouannaud>

Ce texte a été rédigé à la hâte. Il contient donc certainement des typos, des références manquantes, et quelques erreurs. Par ailleurs, il ne reflète que la vision de son auteur. Il se peut donc qu'il soit ultérieurement complété par la propre vision de Claude Marché.

11 janvier 1970

Historique

Curry-Howard

Le début des années 30 a vu fleurir une multitude de formalisations de la notion de calcul. L'une d'elle, celle de Church, avait l'ambition de formaliser la notion de calcul fonctionnel, afin d'en faire la base de ce qui est devenu la logique d'ordre supérieur.

C'était le λ -calcul, formalisation algébrique économe, basée sur :

- un opérateur d'application noté par juxtaposition, permettant de construire le terme (uv) à partir des termes u et v ;
- un opérateur d'abstraction noté λ permettant de construire le terme $(\lambda x : \sigma.u)$ à partir de la variable x et du terme u ;
- une opération de calcul appelée β -réduction

$$((\lambda x.u) v) \rightarrow_{\beta} u\{x \mapsto v\}$$

où $u\{x \mapsto v\}$ désigne le remplacement des occurrences libres de x dans u par v ;

- une opération de renommage appelée α -conversion qui identifie les deux termes $u[\lambda x.v]$ et $u[\lambda y.v\{x \mapsto y\}]$, où y est une variable fraîche, c'est-à-dire n'appartenant pas aux variables libres de $u[\lambda x.v]$;
- une opération appelée extensionnalité ou η -extension qui identifie les deux termes $\lambda x.(ux)$ et u si x n'est pas une variable de u , ce qui traduit le fait que les deux termes $\lambda x.(ux)$ et u ont le même comportement observable lorsqu'on les applique tous deux à un terme v quelconque.

Dans la pratique, on omet les parenthèses superflues.

On sait que le λ -calcul est confluente, mais ne termine pas. De fait, il a la même puissance d'expression que les machines de Turing. Une logique contenant en son sein un calcul Turing-complet est malheureusement nécessairement inconsistente. Pour résoudre ce problème, Church a restreint le λ -calcul de manière à imposer la terminaison, à l'aide d'une discipline de typage exprimée par le système de types de la figure 1, où le jugement $\Gamma \vdash u : \sigma$ affirme que u a le type σ dans l'environnement Γ . Dans la formulation de Church, l'ensemble des types, dits *simples*, est engendré à partir d'un ensemble de types de bases par le constructeur \rightarrow de types fonctionnels. L'opération d'abstraction est changée, permettant maintenant de construire le terme $(\lambda x : \sigma.u)$ à partir de la variable x , du type σ , et du terme u . Les règles de typage sont données à la figure 1.

Variables: $\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$
Abstraction: $\frac{\Gamma \cdot \{x : \sigma\} \vdash t : \tau}{\Gamma \vdash (\lambda x : \sigma.t) : \sigma \rightarrow \tau}$
Application: $\frac{\Gamma \vdash s : \sigma \rightarrow \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash (s t) : \tau}$

FIG. 1 – La discipline des types simples

Le système de typage de Church s’est avéré avoir une propriété fondamentale remarquée par Curry et étudiée par Howard : les types peuvent être vus comme des propositions, et les termes comme des preuves de ces propositions, de telle sorte que le λ -calcul typé simple de Church est isomorphe au fragment implicatif de la logique intuitionniste propositionnelle, la règle de β réduction s’appellant élimination des coupures dans ce cadre.

Le système des *types simples* de Church restreint le calcul à un point que très peu de fonctions sont exprimables, comme l’a montré Schwichtenberg. Pour résoudre ce problème, trois extensions indépendantes fondamentales indépendantes ont été proposées dans un premier temps qui toutes préservent l’interprétation de Curry-Howard :

- Le système T de Gödel [15], dans lequel de nouveaux termes apparaissent avec leur règles de typage représentées à la figure 2 qui permettent de construire une représentation des entiers “à la manière de Peano”,

Zero:		
$\frac{}{\Gamma \vdash 0 : \mathbf{N}}$		
Successeur:		
$\frac{\Gamma \vdash t : \mathbf{N}}{\Gamma \vdash s(t) : \mathbf{N}}$		
Récursur:		
$\frac{\Gamma \vdash x : \mathbf{N} \quad \Gamma \vdash u : \gamma \quad \Gamma \vdash v : \mathbf{N} \rightarrow \gamma \rightarrow \gamma}{\Gamma \vdash \text{rec}(x, u, v) : \gamma}$		

FIG. 2 – Typage des entiers

ainsi que de nouvelles réductions appelées règles de récursion primitive d’ordre supérieur :

$$\text{rec}(0, u, v) \rightarrow u \qquad \text{rec}(s(x), u, v) \rightarrow v(x, \text{rec}(x, u, v))$$

Exercice 0.1 *Montrer la terminaison des règles de réduction (β -réduction et r’ecursion primitive d’ordre supérieur) de système T .*

Quelle est la logique correspondant à T ?

- Le système de De Bruijn, implémenté dans le logiciel *AUTOMATH*, qui fut historiquement le premier assistant de preuves basé sur l’isomorphisme de Curry-Howard. Le système de de Bruijn étend le λ -calcul typé simple de Church en permettant les types produits [11]. D’un point de vue logique, cela revient à ajouter des prédicats.
- Le système de Girard étend le λ -calcul typé simple de Church en permettant les types polymorphes [14]. Généralement appelé *Système F*, le λ -calcul typé polymorphe de Girard a joué un rôle considérable pour plusieurs raisons. La première est que la preuve de normalisation forte a nécessité l’introduction d’un nouvel outil, les *candidats de réductibilité*, qui s’est avéré être un mécanisme essentiel à la compréhension du polymorphisme, et aux preuves de normalisation forte effectuées depuis lors. La seconde est que l’impredicativité du calcul (on définit des objets par quantification sur tous les objets, y compris ceux que l’on est en train de définir) a permis un typage uniforme des entiers de Church, et plus généralement, la possibilité de définir les structures de données inductives habituelles de la programmation. Enfin, Girard a pu montrer qu’il pouvait définir dans son calcul *toutes* les fonctions prouvablement totales dans l’arithmétique d’ordre supérieur.

Ces calcul souffrent néanmoins d'un certain nombre de manques, en particulier la richesse fonctionnelle du calcul de Girard contraste avec sa pauvreté logique, puisque les prédicats ne sont pas des expressions du calcul.

L'étape suivante a consisté à unifier les calculs de Girard et de de Bruijn : le calcul des constructions (CC) était né, qui contient donc toutes les fonctions prouvablement totales dans l'arithmétique d'ordre supérieur, et tous les énoncés de la logiques d'ordre supérieur intuitionniste, les premières étant les preuves des seconds. Son implémentation par Thierry Coquand a constitué la première version du système Coq.

La quête n'était pas terminée. Le calcul des constructions s'est rapidement avéré avoir des lacunes. Comme par exemple l'impossibilité de prouver que 0 et 1 sont différents dans la représentation de Church des entiers. Sans parler de la difficulté pratique de spécifier et prouver dans ce calcul.

L'étape suivante a consisté à unifier CC et T (en généralisant ce dernier), c'est le Calcul des Constructions Inductives (CIC) de Coquand et Paulin, qui est la base du système Coq actuel, qui étend CIC par l'ajout des types coinductifs (travail d'Eduardo Gimenez) et de modules et foncteurs dans la tradition de ML (travaux de Judicaël Courant et de Jacek Chrzaszcz). Tout cela a donné la version actuelle Coq 7.4 [16].

Le lecteur trouvera une excellente synthèse des λ -calculs typés dans [2].

Vers un langage de programmation de preuves

Ces derniers travaux font que le calcul ne se contente plus de contenir un sous calcul fonctionnel permettant de définir des fonctions et même des types par récurrence, mais qu'il ressemble de plus en plus à un langage de spécification (de problèmes existentiels) et de programmation (de preuves), la programmation consistant à rechercher une preuve d'existence.

Les questions qui doivent être posées en permanence, sous peine de perdre un leadership que les concurrents remettent perpétuellement en cause, sont les suivantes :

- les utilisateurs sont-ils satisfaits ?
- Les fondements du calcul sont-ils les bons ?

Au cours des dix dernières années, les utilisateurs ont constamment demandé de transformer Coq de manière à augmenter la productivité des ingénieurs (ou chercheurs) qui l'utilisent. Plus concrètement, la demande constante était d'une part d'augmenter l'automatisation de manière à éviter les preuves faciles, et d'autre part de faciliter la spécification et le traitement automatique du prédicat d'égalité qui n'est pas primitif dans le calcul des constructions.

Est automatique ce qui est *calculatoire*. Dans le calcul des constructions, le calcul intervient dans la règle de typage appellée *conversion* et représentée à la figure 3.

<p>Conversion CC:</p> $\frac{\Gamma \vdash p : M}{\Gamma \vdash p : N} \text{ i f } M \xrightarrow[\beta]{*} N$
--

FIG. 3 – Typage par conversion dans CC

Dans le calcul des constructions inductives, la condition $M \xrightarrow[\beta]{*} N$ est bien sûr remplacée par $M \xrightarrow[\beta \cup \iota]{*} N$, la ι -réduction de CIC décrivant la généralisation de la réduction des récursifs de Gödel pour les types inductifs de CIC.

Cette règle joue un rôle fondamental à plusieurs titres : tout d’abord, elle introduit le calcul comme un moyen automatique de preuve, en permettant par exemple de ramener la preuve de $Pair(2 + 2)$ à la preuve de $Pair(4)$ grâce au calcul —sans intervention de l’utilisateur— de $2 + 2$ en 4 si la fonction $+$ est définie par une abstraction adéquate sur une représentation de Church des entiers ; elle abstrait les calculs du terme preuve, puisque le même terme preuve est conservé, à la fois comme preuve de $Pair(2 + 2)$ et comme preuve de $Pair(4)$. Cette seconde propriété s’avère avoir un effet pratique considérable sur la taille des preuves qui peuvent contenir de très grands calculs.

Le projet INRIA LogiCal a pour but d’exploiter la règle de conversion afin d’automatiser et compacter les preuves. Deux directions sont empruntées, qui s’avèrent en pratique à la fois complémentaires et convergentes, quoique historiquement conçues indépendamment :

- une reformulation complète de la logique (classique, intuitionniste, premier ordre, ordre supérieur) sous l’angle de l’intégration du raisonnement et du calcul. C’est l’ambitieuse démarche adoptée par Gilles Dowek sous le nom de *Déduction modulo* [12] ;
- une intégration des mécanismes de réécriture au calcul des constructions via la règle de conversion. C’est la démarche pragmatique qui a façonné le programme de travail de l’auteur au cours de ces 15 dernières années. C’est cette seconde approche que nous allons détailler dans la suite de cette introduction.

Le Calcul des Constructions Algébriques

L’idée en est très simple. C’est maintenant l’utilisateur qui définit les calculs possibles. Pour cela, il peut déclarer de nouveaux symboles de fonctions avec leur type, définissant ainsi une signature utilisateur \mathcal{F} , ainsi que des règles de réécriture formées sur les termes typables du calcul, c’est le système de réécriture utilisateur R . La règle de conversion est modifiée en conséquence, la réduction utilisée devant tenir compte des règles de réécriture de l’utilisateur en sus de la β -réduction. On obtient ainsi le système de typage de la figure 4 décrivant le calcul des Constructions Algébriques.

<p>Fonctions:</p> $\frac{f : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma \in \mathcal{F} \quad \Gamma \vdash t_1 : \sigma_1 \xi \dots \Gamma \vdash t_n : \sigma_n \xi}{\Gamma \vdash f(t_1, \dots, t_n) : \sigma}$ <p>Conversion CAC:</p> $\frac{\Gamma \vdash p : M}{\Gamma \vdash p : N} \text{ if } M \xrightarrow[\beta \cup R]{*} N$

FIG. 4 – Typage par conversion dans CAC

La mise en oeuvre concrète de la règle de conversion suppose la décidabilité de $\beta \cup R$. En pratique, on passe par le calcul des formes normales de tête dans le cas de CIC, et par les formes normales pour CAC —c’est ce que fait l’implémentation prototype actuelle de CAC. La définition de formes normales de tête pour CAC dans le cas général est un problème qui reste à creuser même si cela a déjà été fait dans des cas particuliers.

La question qui se pose maintenant est de savoir sous quelles conditions la préservation du typage, la confluence et la terminaison de R s’étendent au calcul CAC tout entier. Ce type de question est appelé *modularité*. La modularité suppose en général que les signatures (ici celles de la règle β d’une

part et celle de R d'autre part) soient disjointes. Le cours sera essentiellement consacré à des questions de modularité de la terminaison, et à leur instrumentation pour les besoins du projet LogiCal.

Modularité des λ -calculs

Les questions de modularité ne sont pas nouvelles, elles ont été abordées avant que l'on en ressente le besoin dans le projet LogiCal, par Breazu-Tannen dans le cadre du λ -calcul typé simple, et par Toyama dans le cadre des systèmes de réécriture de premier ordre.

Quoique la question soit parfois complexe en présence de règles d'ordre supérieur, nous ne parlons pas beaucoup de la modularité de la préservation du typage qui ne fait pas appel à des techniques spécifiques.

La question de modularité de la confluence du λ -calcul avait été investiguée dans des cas particuliers nombreux, en particulier par l'école néerlandaise (Barendregt-Klop), avec des résultats positifs ou négatifs suivant les cas. Val Breazu-Tannen¹ a eu le premier l'idée de regarder le problème pour un système de règles confluent arbitraire, et un λ -calcul typé simple [7]. Il y résolvait positivement la question de la confluence —qui fait appel à des techniques élémentaires— et laissait ouverte celle de la terminaison. Cette dernière était résolue l'année suivante pour une discipline de types polymorphe, indépendamment par Breazu-Tannen et Gallier [8] d'une part, et par Okada [29] d'autre part. La preuve faisait appel au candidats de réductibilité de Girard, méthode qui a systématiquement été utilisée par la suite. C'est de ce travail que nous sommes partis.

Exercice 0.2 *Montrer que $\longrightarrow_{\beta \cup R}$ termine pour tout calcul typé pour lequel \longrightarrow_{β} et \longrightarrow_R pris séparément terminent, dans le cas où les règles de R sont soit toutes linéaires gauches, soit toutes linéaires droites.*

Modularité des réécritures de premier ordre

En pratique, considérer R comme un tout n'est pas très réaliste. La spécification est en effet progressive, et utilise les outils de modularité que Coq fournit au spécifieur. Certaines preuves seront menées à bien, puis d'autres plus tard avec un système de règles plus riche. Il est donc essentiel que les propriétés de confluence et de terminaison de la réécriture R soient elle-même modulaires vis-à-vis des sous-systèmes de règles qui le constituent. Le premier à s'être intéressé à ce problème est Toyama, qui résolut positivement la question de la confluence de $R_1 \cup R_2$ dans le cas où les constituants R_1 et R_2 sont confluent et ne partagent pas de symboles de fonctions [31], et négativement celle de la terminaison [30], avec le célèbre contre-exemple :

$$R_1 = \{f(0, 1, x) \rightarrow f(x, x, x)\} \quad R_2 = \{or(x, y) \rightarrow x, or(x, y) \rightarrow y\}$$

Le théorème de Toyama est complexe, et sa preuve non triviale, même si elle fût ultérieurement améliorée par le quatuor Klop-Middeldorp Toyama DeVrijer[23]. Il fût également généralisé par Ohlebusch au cas où R_1 et R_2 partagent des symboles constructeurs uniquement —sous une hypothèse technique additionnelle [28]. Une preuve astucieuse beaucoup plus simple et non publiée fait appel à un mécanisme particulier de complétion, dit sans échec, dû à Hsiang et Rusinowitch [?].

Curieusement, le contre-exemple de Toyama a suscité plus d'intérêt que son théorème, sans doute parce qu'il était contraire à l'intuition. S'en débarrasser n'est pas si simple, comme le montre l'exercice qui suit.

¹Val Tannen fait partie des familles d'origine allemande résidant en Roumanie, dont le régime de Ceaucescu avait roumanisé de force le nom. Émigré aux États-Unis, il a repris son nom d'origine à la chute du régime.

Exercice 0.3 *Étendre le contre-exemple de Toyama au cas où les deux systèmes sont confluents.*

Néanmoins, de nombreux résultats positifs ont été obtenus : la terminaison est modulaire lorsque la preuve de terminaison des systèmes R_1 et R_2 est faite avec des ordres de simplification [?]. L'existence d'une forme normale unique pour tout terme est modulaire [?]. Ce dernier résultat montre d'ailleurs que le contre-exemple de Toyama n'est pas un problème rédhibitoire, car l'existence de formes normales uniques suffit en fait à assurer la cohérence logique, et même la décidabilité du typage si ces formes normales sont calculables.

La thèse déjà ancienne de Middledorp contient de nombreux résultats de modularité [25].

Exercice 0.4 *Montrer que la terminaison est modulaire si R_1 et R_2 n'ont pas de règles duplicantes (une variables x a plus d'occurrences à droite qu'à gauche) ni projetante (un membre droit réduit à une variable).*

Enfin, la méthode de preuve de terminaison due à Aarts et Giesl (paires de dépendances) a fait progresser le domaine de manière très importante. Ces résultats seront traités en cours.

Réécritures de premier ordre modulo

En pratique, certains symboles de fonctions ont souvent des propriétés calculatoires complexes qui ne s'expriment pas sous la forme de réduction mais d'égalités non orientables qui sont alors intégrées à l'algorithme de filtrage permettant de déclencher les règles. Ce type de réécriture présente l'avantage —outre le fait que l'on a pas le choix dans certains cas comme celui des réécritures commutatives— de spécifier de manière très compacte. Voici par exemple une spécification des listes triées par ordre croissant grâce à une unique règle :

$$l_1 \cdot x \cdot l_2 \cdot y \cdot l_3 \rightarrow l_1 \cdot y \cdot l_2 \cdot x \cdot l_3 \quad \text{if } x > y$$

Cette règle suppose bien-sûr que la concaténation des listes, notée ici \cdot , possède la liste vide pour élément neutre (Z) et soit associative (A). Cette règle se trouve de plus être conditionnelle, il faut donc intégrer la vérification que l'entier $x\sigma$ est plus grand strictement que l'entier $y\sigma$ à l'algorithme de recherche d'une substitution σ qui satisfait, étant donné un terme u , la propriété

$$u \xrightarrow[AZ]{*} (l_1 \cdot x \cdot l_2 \cdot y \cdot l_3 \rightarrow l_1 \cdot y \cdot l_2 \cdot x \cdot l_3)\sigma$$

La réécriture dite modulo a été introduite dans le cas associatif-commutatif —le plus important en pratique— par Peterson et Stickel, qui ont étendu partiellement la théorie classique en donnant un critère de confluence complexe, basé sur l'unification associative-commutative pour le calcul des paires critiques [?]. Le traitement mathématique rigoureux de ces techniques apparût plus tard [17]. L'algorithme d'unification associative-commutative est dû à Stickel [?], mais la preuve de terminaison a résisté dix ans jusqu'au travail de François Fages [?]. La généralisation des méthodes de terminaison prit elle-aussi beaucoup de temps [?], avant qu'une généralisation élégante de l'ordre récursif sur les chemins ne soit trouvée par Nieuwenhuis et Rubio [?].

Les questions de modularité se posent bien sûr également dans ce cadre. Rien n'est connu concernant la confluence, cela est sans doute dû à la complexité technique des outils mis en oeuvre par Toyama et ses coauteurs. Nous pensons que la nouvelle preuve —non publiée due à l'auteur— devrait se généraliser sans trop de difficulté, mais le travail reste à faire².

La méthode de Aarts et Giesl a elle-aussi été généralisée au cas associatif-commutatif et a conduit à des résultats de modularité implantés dans le logiciel CiMe qui seront présentés dans le cours [?].

²Si l'un de vous était intéressé, cela pourrait éventuellement faire l'objet d'un travail de mise en train simple avec publication assurée en bout de course.

Réécritures d'ordre supérieur

Les règles de calcul des récursifs de Gödel sont des réécritures d'ordre supérieur, dans la mesure où certaines sous-expressions apparaissant dans ces règles ont un type fonctionnel, voire polymorphe³ :

$$rec(0, u, v) \rightarrow u \quad rec(s(x), u, v) \rightarrow v(x, rec(x, u, v))$$

Dans ces règles, seuls $0, x, s(x)$ ont un type simple, \mathbb{N} . Tous les autres termes ont soit le type polymorphe γ , soit le type fonctionnel polymorphe $\mathbb{N} \rightarrow \gamma \rightarrow \gamma$. Il s'agit de règles typiques d'ordre supérieur, dont le déclenchement requiert l'utilisation d'un algorithme de filtrage identique au filtrage du premier ordre. Toutes les règles de CIC sont de ce type : récursifs déclenchés par filtrage de premier ordre. La raison en est que ces règles sont très spécifiques, adopter un algorithme de filtrage plus expressif ne changerait en rien la notion de calcul.

Cela n'est plus le cas des règles d'ordre supérieur suivantes qui décrivent la mise en forme pré-nexe de formules logiques quantifiées, où les quantificateurs sont des symboles unaires prenant un argument de type fonctionnel de manière à jouer un rôle de lieur :

$$\begin{aligned} P \wedge \forall(\lambda x.Q(x)) &\rightarrow \forall(\lambda x.(P \wedge Q(x))) \\ \forall(\lambda x.Q(x)) \wedge P &\rightarrow \forall(\lambda x.(Q(x) \wedge P)) \\ P \vee \forall(\lambda x.Q(x)) &\rightarrow \forall(\lambda x.(P \vee Q(x))) \\ \forall(\lambda x.Q(x)) \vee P &\rightarrow \forall(\lambda x.(Q(x) \vee P)) \\ P \wedge \exists(\lambda x.Q(x)) &\rightarrow \exists(\lambda x.(P \wedge Q(x))) \\ \exists(\lambda x.Q(x)) \wedge P &\rightarrow \exists(\lambda x.(Q(x) \wedge P)) \\ P \vee \exists(\lambda x.Q(x)) &\rightarrow \exists(\lambda x.(P \vee Q(x))) \\ \exists(\lambda x.Q(x)) \vee P &\rightarrow \exists(\lambda x.(Q(x) \vee P)) \\ \neg(\forall(\lambda x.Q(x))) &\rightarrow \exists(\lambda x.\neg(Q(x))) \\ \neg(\exists(\lambda x.Q(x))) &\rightarrow \forall(\lambda x.\neg(Q(x))) \end{aligned}$$

Cet exemple n'a de sens que si la réécriture utilise un filtrage d'ordre supérieur, c'est-à-dire modulo $\beta\eta$, car l'utilisation du filtrage de premier ordre ne permettrait pas de sortir les quantificateurs pour des termes quelconques. Par exemple, la formule $\phi \wedge \forall(\lambda z.z \vee z)$ n'est pas une instance du membre gauche de la première règle, mais elle en est une $\beta\eta$ -instance.

Cette forme de réécriture d'ordre supérieure a été introduite par Tobias Nipkow, qui l'a utilisée pour spécifier des manipulations de syntaxe abstraite [27]. Elle a ensuite été largement utilisée dans les démonstrateurs, en particulier par Frank Pfenning [?].

Le filtrage modulo $\beta\eta$, c'est-à-dire la recherche des substitutions σ , étant donnés u et v , telles que $u \xrightarrow{\beta\eta}^* v\sigma$ est appelée *filtrage d'ordre supérieur*. Le décidabilité du filtrage d'ordre supérieur est un problème ouvert. L'ordre 1 (filtrage habituel) est le plus simple, il y a au plus une solution. L'ordre 2 reste simple, il y a un nombre fini de solutions, il est dû à Gérard Huet. L'ordre 3 est plus complexe, les solutions sont en nombre infini, il a été résolu par Gilles Dowek. L'ordre 4 a été résolu par Vincent Padovani. L'histoire s'arrête là. Hubert Comon a montré que les solutions d'un problème de filtrage d'ordre n étaient reconnaissable par un automate d'arbre dont les états étaient les solutions de problèmes de filtrage d'ordre $n - 2$, ce qui explique que le cas de l'ordre 5 n'ait rien à voir avec le cas de l'ordre 4.

La confluence de règles de réécriture d'ordre supérieur modulo $\beta\eta$ se ramène à un calcul de paires critiques par unification d'ordre supérieur, comme l'ont montré Tobias Nipkow et Richard Mayr [24].

³Système T avait la même discipline de typage que le calcul des types simples de Church, et les règles étaient donc en fait des schémas de règles, une pour chaque type possible.

Cette dernière est malheureusement indécidable, un résultat de Huet. Toutefois, Dale Miller a montré qu'elle était essentiellement similaire à l'unification du premier ordre dans le cas pratique important des *patterns*, termes dans lesquels toute occurrence de variable libre de type $\alpha_1 \rightarrow \dots \alpha_n \rightarrow \alpha$ est appliqué à n variables liées distinctes possédant les types adéquats [26].

On peut se demander si la confluence des systèmes d'ordre supérieur est modulaire. Ce domaine est très ouvert, mais il est peu probable que l'on puisse obtenir des résultats intéressants en dehors du cas où les membres gauches sont des *patterns*, pour lequel β et η -réductions jouent un rôle très contrôlé lors des unifications permettant de limiter les interactions dues à la signature partagée qui se trouve dans ce cas être celle du λ -calcul.

Terminaison d'ordre supérieur

Plusieurs problèmes se posent, terminaison des différentes formes de réécriture d'ordre supérieur et problème de la modularité associés, mais il se trouve que les seules techniques connues de preuve de terminaison à l'ordre supérieur sont intrinsèquement modulaires et ne permettent de prouver la terminaison de réécritures d'ordre supérieur que si ces dernières sont compatibles avec la β -réduction. Ces méthodes sont de deux types.

Dans une première succession d'articles, Jouannaud et Okada [18, 19], puis Barbanera, Fernandez et Geuvers [1], Blanqui, Jouannaud et Okada [6], enfin Blanqui seul [3] ont élaboré une importante généralisation de la récursion primitive d'ordre supérieur qui assure la terminaison de la relation $\longrightarrow_{\beta \cup R}$. Ces travaux permettent de prouver la normalisation forte du calcul des constructions inductives, y-compris les règles dites d'élimination forte qui opèrent sur des expressions de types [4], comme la règle

$$x \times y = 0 \rightarrow x = 0 \vee y = 0$$

Blanqui a également montré que ces résultats pouvaient être aisément adaptés au cas de la réécriture d'ordre supérieur modulo [5].

Dans une autre succession d'articles plus récents, Jouannaud et Rubio [20, 21], puis Walukiewicz [32] ont élaboré une méthode intrinsèquement plus puissante, basée sur une généralisation à l'ordre supérieur de l'ordre récursif sur les chemins, qui a la propriété de contenir la β -réduction. Cette méthode est intrinsèquement plus puissante dans la mesure où la première apparaît rétrospectivement comme étant un cas très particulier de la seconde, correspondant en gros au cas "sous-terme". Cette méthode peut en fait être vue comme la construction d'un ordre de réécriture qui exprime la méthode des candidats de Girard, et cela explique son expressivité. Outre son expressivité, les avantages de cette dernière méthode sont doubles : elle s'adapte facilement à des besoins particuliers, en particulier à la réécriture modulo $\beta\eta$, et elle est aisément implémentable —quoique pouvant requérir une certaine interactivité avec l'utilisateur.

L'ordre récursif sur les chemins à l'ordre supérieur sera présenté en cours. Il conduit à une conjecture importante, l'existence d'un ordre bien-fondé sur les termes du calcul des constructions qui soit uniforme, c'est-à-dire dont l'expression ne fasse pas de différence entre les termes, les types et les "kinds". L'ordre actuel issu des travaux de Daria Walukiewicz fait une telle différence.

Implémentation de la réécriture dans Coq

L'intégration de la réécriture dans Coq est en cours, mais nécessitera des travaux pendant encore plusieurs années, en particulier si l'on décide un jour de compiler la réduction en présence de réécritures utilisateur, problème qui nécessite une compilation incrémentale qui est loin d'être maîtrisée dans ce cadre.

Theories “Shostak”

En parallèle, nous réfléchissons à une nouvelle extension, complémentaire de la réécriture, du mécanisme de conversion, dont l’idée emprunte aux travaux de Shostak [?]. Ces travaux anciens sont basés sur une technique appelée *clôture par congruence*, découverte indépendamment et au même moment par Shostak, Nelson [?] et Kozen [?]. Il s’agit de décider si deux expressions sont égales dans la théorie équationnelle engendrée par un ensemble fini d’équations closes de premier ordre. Shostak a ensuite montré que cette méthode très efficace puisque de complexité $n \log n$ pouvait s’étendre à d’autres théories, les “théories Shostak”, telles l’arithmétique linéaire ou la théorie des listes avec constructeurs et destructeurs, qui sont très importantes en pratique. Cette méthode est au coeur du système PVS dont elle a fait le succès.

Par la suite, on notera par $Sh(\Gamma)$ la théorie Shostak engendrée par les équations closes issues de l’environnement Γ . La règle de conversion associée est représentée à la figure 5.

<p>Conversion Shostak:</p> $\frac{\Gamma \vdash p : M}{\Gamma \vdash p : N} \text{ iff } M \downarrow_{\beta} \xrightarrow[Sh(\Gamma)]{*} N \downarrow_{\beta}$
--

FIG. 5 – Typage par conversion dans CC-Shostak

La nouveauté, et la difficulté du système de typage introduit ici vient du fait que la conversion utilise des égalités issues du contexte, donc pouvant varier au cours de la preuve. Ces égalités sont généralement des hypothèses introduites par l’utilisateur sous la forme, par exemple, $x : f(0) = 0$. Le type de la variable ainsi déclarée dans l’environnement doit donc être clos pour pouvoir être utilisée dans la conversion, sous peine d’engendrer des inconsistences.

Ce calcul a été implanté sous la forme d’un prototype dans le langage MAUDE [?], en profitant du développement en MAUDE, par Mark Olliver Stehr, de l’Open Calculus of Constructions [?]. Nous avons montré que ce calcul avait la propriété de préservation du typage. Ses autres propriétés méta-théoriques restent à investiger, et il s’agit là d’un travail de thèse, ainsi que la généralisation au cas de certaines égalités non-closes ou d’ordre supérieur. Les difficultés de cette question ne doivent pas être sous-estimées, puisque l’algorithme de Shostak a été utilisé pendant plus de 15 ans avant que l’on s’aperçoive qu’il était incomplet et ne terminait pas toujours. Sa correction a occupé pas mal de monde pendant plusieurs années [?], les premières tentatives s’étant soldées par des échecs.

Les références manquantes peuvent pour la plupart être trouvées dans [9], mais ne sont pas toutes essentielles à la compréhension du cours, heureusement ...

Bibliographie

- [1] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization and confluence in the algebraic- λ -cube. In *Proc. of the 9th Symp. on Logic in Computer Science*, IEEE Computer Society, 1994.
- [2] Henk Barendregt. *Handbook of Logic in Computer Science*, chapter Typed lambda calculi. Oxford Univ. Press, 1993. eds. Abramsky et al.
- [3] BLANQUI, F. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science* (à paraître).
- [4] BLANQUI, F. Inductive types in the Calculus of Algebraic Constructions. In *Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications* (2003), no. 2701 in Lect. Notes Comp. Sci.
- [5] BLANQUI, F. Rewriting modulo in Deduction modulo. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications* (2003), no. 2706 in Lect. Notes Comp. Sci.
- [6] BLANQUI, F., JOUANNAUD, J.-P., AND OKADA, M. Inductive data type systems. *Theoretical Computer Science*, 272(1-2) (2002), 41–68.
- [7] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. of the 3rd Symp. on Logic in Computer Science*, IEEE Computer Society, 1988.
- [8] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1), 1991.
- [9] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.
- [10] Maribel Fernández and Jean-Pierre Jouannaud. Modular termination of term rewriting systems revisited. In Egidio Astesiano, Gianni Reggiov, and Andrzej Tarlecki, editors, *Recent Trends in Data Type Specification*, Lecture notes in Computer Science, vol.906, pages 255–272. Springer-Verlag, 1995. Refereed selection of papers presented at ADT’94.
- [11] N. de Bruijn. The mathematical language Automath, its usage, and some of its extensions. In *Proc. of the Symp. on Automatic Demonstration*, LNCS 125, 1970. Reprinted in : Selected Papers on Automath, edited by R.P. Nederpelt, J.H. Geuvers and R.C. de Vrijer, Studies in Logic, vol. 133. North-Holland, 1994.
- [12] DOWEK, G., HARDIN, T., AND KIRCHNER, C. Theorem proving modulo. *Journal of Automated Reasoning* 31 (2003), 33–72.
- [13] E. Giménez. Structural recursive definitions in type theory. In *Proc. of the 25th Int. Colloq. on Automata, Languages, and Programming*, LNCS 1443, 1998.

- [14] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, France, 1972.
- [15] K. Gödel. On intuitionistic arithmetic and number theory. In M. Davis, editor, *The undecidable*. Raven Press, 1965.
- [16] Gérard Huet, Gilles Kahn and Christine Paulin-Mohring. The Coq Proof Assistant. A Tutorial. Version 7.3. INRIA Rocquencourt and ENS Lyon.
- [17] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *Information and Computation*, 15(4) :1155–1194, 1986.
- [18] Jean-Pierre Jouannaud and Mitsuhiro Okada. Higher-Order Algebraic Specifications. In , editor, *Annual IEEE Symposium on Logic in Computer Science*, Amsterdam, The Netherlands, 1991. IEEE Comp. Soc. Press.
- [19] Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2) :349–391, February 1997.
- [20] Jean-Pierre Jouannaud and Albert Rubio. The higher-order recursive path ordering. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.
- [21] Jean-Pierre Jouannaud and Albert Rubio. Higher-order recursive path orderings “a la carte”. submitted, 2003.
- [22] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems : introduction and survey. *Theoretical Computer Science*, 121(1-2), 1993.
- [23] Jan Willem Klop, Aart Middeldorp, Yoshihito Toyama, and Roel de Vrijer. Modularity of confluence : A simplified proof. *Information Processing Letters*, 49(2) :101–109, 1994.
- [24] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192, 1998.
- [25] Aart Middeldorp. *Modular Properties of Term Rewriting Systems*. PhD thesis, Free University of Amsterdam, Netherland, 1990.
- [26] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proc. of the 1989 Int. Work. on Extensions of Logic Programming*, LNCS 475.
- [27] Tobias Nipkow. Higher-order critical pairs. In *6th IEEE Symp. on Logic in Computer Science*, pages 342–349. IEEE Computer Society Press, 1991.
- [28] Enno Ohlebusch. On the modularity of confluence of constructor-sharing term rewriting systems. In S. Tison, editor, *Proceedings of the Nineteenth International Colloquium on Trees in Algebra and Programming (Edinburgh, UK)*, volume 787 of *Lecture Notes in Computer Science*, pages 262–275, Berlin, April 1994. Springer-Verlag.
- [29] M. Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In *Proc. of the 1989 Int. Symp. on Symbolic and Algebraic Computation*, ACM Press.
- [30] Y. Toyama. Counterexamples to terminating for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3), 1986.
- [31] Y. Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *Journal of the ACM*, 34(1) :128–143, April 1987.
- [32] WALUKIEWICZ-CHRZASZCZ, D. Termination of rewriting in the calculus of constructions. *J. Functional Programming* (à paraître). to be published in 2004.