Université de Paris VII - Denis Diderot
D.E.A. Sémantique, Preuves, Langages
Année 2002-2003

Intégration de procédures de décision en théorie des types

Stage réalisé par
Pierre-Yves Strub

Sous la direction de
Jean-Pierre Jouannaud et Frédéric Blanqui

Dans l'équipe
Récriture et Preuves

du
Laboratoire d'informatique
de l'École Polytechnique

Ce travail a été en partie financé par le projet CNRS - Urbana Champaign - Calculs par récriture, logique et comportement.

Table des matières

Introduction

On désigne généralement par *théorie des types* tout formalisme logique dont les objets sont des λ-termes typés. Il en existe aujourd'hui plusieurs qui se distinguent par la discipline de types plus ou moins riche des objets, ainsi que par la logique sous-jacente permettant de parler de ces derniers. Ces formalismes sont à la base d'une `ingénierie des mathématiques formelles` en cours d'émergence.

A titre d'exemples de formalisme, on peut citer la Logique d'Ordre Supérieure de A. CHURCH, la Théorie des Types de P. MARTIN-LÖF et celui qui nous intéressera dans ce stage : le Calcul des Constructions de T. COQUAND et G. HUET[5, 6].

Le Calcul des Constructions, basé sur l'isomorphisme de CURRY - DE BRUIJN - HOWARD, est un formalisme particulièrement expressif du point de vue algorithmique. Il permet les raisonnements de type mathématique pour lesquels il est possible de vérifier *mécaniquement* la validité des constructions faites à l'intérieur du système. Une telle propriété présente l'avantage d'éliminer toute erreur humaine, critère important dans des domaines tels que la certification de programmes où les preuves ne présentent pas nécessairement de grosses difficultés conceptuelles, mais nécessitent souvent des raisonnements par cas longs et fastidieux.

L'expressivité algorithmique du Calcul des Constructions permet la simplification des preuves par l'introduction de calculs. Par exemple, dans un formalisme où l'addition est définie de manière calculatoire, la preuve de $2+2 = 4$ est immédiate, le système de calcul identifiant syntaxiquement les termes $2+2$ et 4. Dans le cas contraire, une preuve entière utilisant les axiomes de l'addition et de l'égalité aurait du être donnée. Par la suite, des extentions au Calcul des Constructions ont été fournies afin de renforcer cette expressivité algorithmique. On peut citer le Calcul des Constructions Inductives, de T. COQUAND et C. PAULIN[7], introduisant les types récursifs primitifs ; ou encore le Calcul des Constructions Algébriques[2] dans lequel le comportement calculatoire est paramétré par un système de récriture donné a priori par l'utilisateur.

On compte de nombreux logiciels de vérification formelle. Le premier fut `Automath`, développé en 1960 par l'équipe de N.G. DE BRUIJN. Ensuite, apparurent des systèmes tels que `HOL` et `Isabelle` à Cambridge, `OYSTER` à Edimbourg ou `ALF` à Göteborg. Il existe deux implémentations concurrentes basées sur le Calcul des Constructions : `LEGO` et `Coq`, développées respectivement à Edimbourg et dans le projet INRIA LOGICAL. La version originelle du logiciel `Coq` était basée sur le Calcul des Constructions. La version distribuée aujourd'hui est basée sur le Calcul des Constructions Inductives[17]. Une version basée sur le Calcul des Constructions Algébriques est en cours de développement.

Parallèlement, de nombreuses procédures de décisions (i.e. des algorithmes pouvant raisonner sur la satisfiabilité de classes de formules dans une théorie décidable) ont été décrites pour des théories du premier ordre rencontrées en pratique. On peut citer, e.g., le calcul propositionnel, l'arithmétique entière et réelle, les théories associées aux structures de données algébriques[12] ou une théorie équationnelle close[11]. L'avantage majeur de telles procédures est leur efficacité. Malheureusement, cette dernière, obtenue par une analyse poussée du domaine d'étude, vient au prix d'une spécialisation qui va à l'encontre d'une vision basée sur la généralité des mécanismes mis en oeuvre.

Pour pallier à ce problème, en 1979, NELSON et OPPEN décrivirent un algorithme général pour la combinaison de procédures de décision [10]. Quelques années plus tard, en 1984, SHOSTAK[15] proposa une autre procédure de combinaison, extrêmement efficace mais dans un cadre plus restreint. Malgré sa popularité et une utilisation intensive pendant plus d'une décennie, l'algorithme de SHOSTAK s'est révélé incomplet et sa preuve de correction partiellement fausse. Au cours des 5 dernières années, un nombre impressionnant d'articles tentèrent de corriger ces erreurs non triviales. Aujourd'hui, les versions complètes de la procédure de SHOSTAK sont vues[8] comme une instance de l'algorithme plus général de NELSON et OPPEN et des versions efficaces ont été données très récemment[4, 13, 14].

L'objectif de ce stage, se déroulant au Laboratoire d'Informatique de l'École Polytechnique (`LIX`), est l'introduction, dans le Calcul des Constructions, de procédures de décision (fournies a priori par l'utilisateur qui pourra à l'avenir se servir dans des bibliothèques de procédures disponibles) combinées grâce à l'algorithme de SHOSTAK. Une première étude métathéorique partielle du nouveau calcul est donnée ainsi que l'implémentation d'un prototype dans le langage `Maude`.

Chapitre 1

L'algorithme de Shostak

Nous introduisons ici la procédure de décision qui sera utilisée dans la règle de conversion du Calcul des Constructions.

En premier lieu, nous décrirons l'algorithme de NELSON et OPPEN[10] qui permet d'obtenir une procédure de décision pour une union disjointe de théories *convexes* en possédant chacune une. Ensuite, une fois introduites les théories Shostak, nous verrons comment l'algorithme de NELSON et OPPEN peut être raffiné dans le cas de théories Shostak[4, 14].

1.1 Préliminaires

Si Σ est une signature du premier ordre et X un ensemble dénombrable de variables, on note $T_\Sigma(X)$ l'ensemble des termes construits sur Σ et X. On utilisera les meta-variables a, b... t, u... pour dénoter les termes et x, y... pour les variables. Si π est une position, $a|_\pi$ est le sous-terme de a à la position π et $a[b]_\pi$ est le terme obtenu par le remplacement de $a|_\pi$ par b. On note $\equiv$ l'égalité syntaxique.

Dans toute la suite, le seul symbole de prédicat autorisé sera l'égalité de Leibnitz (notée $\approx$). Les Σ-*litéraux* sont donc les équations ($a \approx b$) et les inéquations ($a \not\approx b$) sur Σ. Un litéral (positif ou négatif) est noté $a \star b$. Les Σ-*formules* sont construites à partir des Σ-litéraux en utilisant les connecteurs logiques usuels.

Une équation est dite *simple* si et seulement si elle est de la forme $x \approx y$. Un ensemble conjonctif (resp. disjonctif) d'équations simples est appelé une *requête* (resp. *réponse*). Par défaut, un ensemble d"équations sera vu de manière conjonctive. Les méta-variables Γ, Λ... dénoteront des ensembles de litéraux.

Une Σ-formule ϕ est *satisfiable* (resp. *valide*) si et seulement si ϕ s'évalue en vrai pour un (resp. pour tout) Σ-modèle et une (resp. toute) assignation de variables. Une Σ-*théorie* $\mathcal{T}$ est un ensemble de Σ-équations closes. Un $\mathcal{T}$-modèle est un Σ-modèle tel que toute formule de $\mathcal{T}$ s'évalue en vrai dans ce dernier. Si ϕ est une Σ-formule, on dit que ϕ est $\mathcal{T}$-satisfiable (resp. $\mathcal{T}$-valide) si et seulement si $\mathcal{T} \cup \{\phi\}$ est satisfiable (resp. valide) dans tout $\mathcal{T}$-modèle. La $\mathcal{T}$-validité se note $\mathcal{T} \Vdash \phi$.

Une *procédure de décision* pour une théorie $\mathcal{T}$ est un algorithme qui décide si $\mathcal{T} \Vdash \Lambda$ ou non pour tout ensemble de formules sans quantificateur, problème qui se ramène habituellement à un problème de $\mathcal{T}$-satisfiabilité.

Les théories $\mathcal{T}_1, \cdots, \mathcal{T}_n$ sont *disjointes* si elles sont définies sur des signatures $\Sigma_1, \cdots, \Sigma_n$ disjointes. Un terme sur $T_{\cup_i \Sigma_i}(X)$ est dit *mixte*. Un terme t sur $T_{\cup_i \Sigma_i}(X) - X$ est un i-*terme* si et seulement si son symbole de tête appartient à Σ_i. Il est de plus *pur* si tous ses sous-termes sont des i-termes. (Par convention, une variable est un j-terme pour tout j.) Si t est un i-terme, un j-*alien* ($j \neq i$) pour t est tout sous-terme de t qui est un j-terme. Un j-alien à la position π de t est de plus *maximal* si et seulement si pour toute position ρ

strictement préfixe de π, $t|_\rho$ est un i-terme. Un j-sous-terme (j pouvant être égal à i) $t|_\pi$ de t est dit *pur de pureté maximale dans* t si et seulement si $t|_\pi$ est pur et pour toute position ρ strictement préfixe de ρ, $t|_\rho$ n'est pas un j-terme. (Rq. un j-alien de t, pur de pureté maximale dans t, n'est pas nécessairement un j-alien maximal de t.)

1.2 L'algorithme de Nelson et Oppen

On se donne n théories stablement infinies et convexes (Cf. ci-dessous) $\mathcal{T}_1 \cdots \mathcal{T}_n$, respectivement définies sur les signatures $\Sigma_1 \cdots \Sigma_n$ et disjointes deux à deux, pour lesquelles on souhaite déterminer la satisfiabilité d'un ensemble Γ de Σ-équations, où $\Sigma = \cup_i \Sigma_i$. On suppose de plus que pour chaque théorie $\mathcal{T}_i$, on possède une procédure de décision $\mathcal{P}_i$.

Définition 1.2.1 (théorie convexe) Une théorie est **convexe** si et seulement si, pour tout ensemble de littéraux Λ, dès que $\mathcal{T} \Vdash \Lambda \rightarrow (a_1 \approx b_1 \vee \cdots \vee a_n \approx b_n)$, on a $\exists i. \mathcal{T} \Vdash \Lambda \rightarrow a_i \approx b_i$.

Définition 1.2.2 (théorie stablement infinie) Une Σ-théorie $\mathcal{T}$ est **stablement infinie** si et seulement si pour toute Σ-formule $\mathcal{T}$-satisfiable ϕ sans quantificateurs, il existe une $\mathcal{T}$-interprétation $\mathcal{A}$ dont le domaine A est infini.

Il existe de nombreuses théories convexes et stablement infinies comme l'arithmétique linéaire entière et réelle, les théories des structures algébriques ou la théorie des tableaux.

La procédure, qui décide de la satisfiabilité d'un ensemble Γ de $\mathcal{T}$-littéraux, se déroule en deux phases :

Abstraction : les procédures de décision $\mathcal{P}_i$ de chaque théorie ne sachant pas travailler sur des littéraux mixtes, une phase de *purification* est donc nécessaire. Ainsi, si un littéral de Γ est de la forme $a[b]_\pi \star c$[1], avec b un i-terme pur de pureté maximale dans a, b est abstrait par un variable fraîche x. Le nouveau littéral obtenu est donc $a[x]_\pi \star c$ et une nouvelle i-équation pure $x \approx c$ doit être prise en compte. Après un nombre fini d'itération, Γ ne contient plus que des littéraux simples et on possède une famille d'ensembles d'équations $(\phi_i)_i$ où chaque ϕ_i contient les Σ_i-équations pures générées pendant la phase d'abstraction.

Propagation des égalités : ensuite, Γ va être saturé avec les égalités générées par les différentes procédures de décision $\mathcal{P}_i$. Ainsi, si $x \approx y$ est une équation telle que $\mathcal{T}_i, \Phi_i, \Gamma \Vdash x \approx y$ (qui peut être décidé grâce à $\mathcal{P}_i$) mais qui n'est pas satisfiable dans Γ ($\Gamma \not\Vdash x \approx y$), alors elle sera rajoutée à Γ. Deux cas peuvent alors se produire. Soit l'un des $\phi_j \wedge \Gamma$ devient insatisfiable pour un certain j (ce qui peut encore être déterminé par $\mathcal{P}_j$), et alors l'ensemble Γ d'origine était $\mathcal{T}$-insatisfiable. Soit on arrive à saturation de Γ et l'ensemble Γ d'origine était satisfiable.

La procédure est donnée sous la forme de règles d'inférence (Cf. figure 1.1) décrivant l'évolution de l'état de l'algorithme, ce dernier étant représenté par une *configuration* telle que définie ci-dessous :

Définition 1.2.3 (Configuration) Une **configuration** est :

- soit un triplet $< \Delta \ ; \ \Gamma \ ; \ \phi_1 \cdots \phi_n >$ où Δ est un ensemble disjonctif de littéraux simples, Γ est un ensemble de $\mathcal{T}$-littéraux et chaque ϕ_i est un ensemble de littéraux de la forme $x \approx a$ où a est un i-terme pur ;
- soit le symbole $\bot$.

Une configuration est dite **propre** si et seulement si elle est différente de $\bot$.

Définition 1.2.4 Une configuration $< \Delta ; \Gamma ; \phi_1 \cdots \phi_n >$ est **satisfiable** si et seulement si $\Delta \wedge \Gamma \wedge \phi_1 \wedge \cdots \wedge \phi_n$ est $\mathcal{T}$-satisfiable. $\bot$ est considéré comme non-satisfiable.

1. $\approx$ étant considéré comme syntaxiquement symétrique, il n'y a aucune perte de généralité à effectuer les transformations exclusivement sur le membre gauche de $\approx$.

Définition 1.2.5 Une configuration $\mathcal{C}$ se réduit en une configuration $\mathcal{C}'$, noté $\mathcal{C} \Rightarrow \mathcal{C}'$, si et seulement si on peut passer de $\mathcal{C}$ à $\mathcal{C}'$ par une régle du système d'inférence de la figure 1.1.

$$\frac{\Delta\ ;\ \Gamma \uplus \{a \star b\}\ ;\ \phi_1 \cdots \phi_i \cdots \phi_n}{\Delta\ ;\ \Gamma \uplus \{a[z]_\pi \star b\}\ ;\ \phi_1 \cdots \phi_i \cup \{z \approx a|_\pi\} \cdots \phi_n}$$

((AB)STRACT$_i$) si $a|_\pi \notin X$, $a|_\pi$ est pur de pureté maximale dans a et z est une variable fraîche[3]

$$\frac{\Delta\ ;\ \Gamma \uplus \{x \star y\}\ ;\ \phi_1 \cdots \phi_n}{\Delta \cup \{x \star y\}\ ;\ \Gamma\ ;\ \phi_1 \cdots \phi_n}$$

((AR)RANGE)

$$\frac{\Delta\ ;\ \Gamma\ ;\ \phi_1 \cdots \phi_n}{\Delta \cup \{x \star y\}\ ;\ \Gamma\ ;\ \phi_1 \cdots \phi_n}$$

((DE)DUCT$_i$) si $\mathcal{T}_i$,$\phi_i \Vdash \Lambda \rightarrow x \star y$, $\Lambda \subseteq \Delta$ est une requête et $\Delta \not\Vdash x \star y$

$$\frac{\Delta\ ;\ \Gamma\ ;\ \phi_1 \cdots \phi_n}{\bot}$$

((CO)NTRADICT$_i$) si $\phi_i \wedge \Delta$ est insatisfiable

FIG. 1.1 – *Procédure de décision pour une union de théories convexes*

On peut alors énoncer le théorème justifiant que le système d'inférence est une procédure de décision pour $\mathcal{T}$:

Théorème 1.2.6 *[4, 14] Soit Γ un ensemble de Σ-équations. On note $\mathcal{C}_\xi$ la configuration $< \emptyset\ ;\ \Gamma\ ;\ \emptyset \cdots \emptyset >$. Alors, ξ est satisfiable si et seulement si il existe une configuration $\mathcal{C}$ propre, irréductible pour $\Rightarrow$, telle que $\mathcal{C}_\Gamma \Rightarrow^* \mathcal{C}$.*

Bien entendu, tel que, l'algorithme est très peu utilisable, la règle DEDUCT$_i$ ne fournissant aucun moyen de générer *efficacement* les nouvelles équations. C'est ce que propose de résoudre l'algorithme de SHOSTAK, décrit dans la section suivante, en raffinant la règle DEDUCT$_i$.

1.3 L'algorithme de Shostak

En premier lieu, il nous faut définir ce qu'est une théorie SHOSTAK :

Définition 1.3.1 (théorie canonisable) Une théorie $\mathcal{T}$ est **canonisable** si et seulement si elle admet une fonction calculable σ (**canoniseur**) qui décide le problème du mot de manière syntaxique ; i.e., telle que si a et b sont des $\mathcal{T}$-termes purs, alors $\mathcal{T} \Vdash a \approx b \Leftrightarrow \sigma(a) \equiv \sigma(b)$. De plus, σ doit être idempotente et si un terme a est en forme canonique (i.e. $\sigma(a) \equiv a$), alors il doit en être de même pour tous ses sous-termes.

Définition 1.3.2 (théorie solvable) Une théorie $\mathcal{T}$ est **solvable** si et seulement si elle admet une procédure calculable solve (**solveur**) telle que (si a et b sont des $\mathcal{T}$-termes purs) :

- si $a \approx b$ est $\mathcal{T}$-satisfiable, alors $\mathrm{solve}(a \approx b)$ renvoie une forme résolue $x_1 \approx t_1 \cdots x_n \approx t_n$ équisatisfiable à $a \approx b$; i.e. :
  - $\forall i. x_i \in \mathrm{vars}(a \approx b)$;

3. La notion de variable fraîche est une façon informelle de parler de variables quantifiées existentiellement.

- $\forall i,j.i \neq j \Rightarrow x_i \not\equiv x_j$;
- $\forall i,j.i \neq j \Rightarrow x_i \notin \text{vars}(t_j)$;
- $\mathcal{T} \Vdash \forall X.(a \approx b \Leftrightarrow \exists Y.(x_1 \approx t_1 \wedge \cdots \wedge x_n \approx t_n))$
  où $X \equiv \text{vars}(a \approx b)$ et $Y \equiv \text{vars}(x_1 \approx t_1, \cdots, x_n \approx t_n) - X$.

– $\bot$ sinon.

Définition 1.3.3 (théorie Shostak) Une théorie est dite **Shostak** si elle est canonisable, solvable et convexe.

Il existe de nombreuse théorie *Shostak*, comme l'arithmétique linéaire ou la théorie des listes. Des canoniseurs et solveurs sont présentés en annexe A pour ces deux théories.

Une procédure de décision peut être facilement décrite pour les théories SHOSTAK. E.g., l'algorithme décrit en figure 1.2 décide de la satisfiabilité d'en ensemble d'équations Γ.

$$\frac{\Gamma \uplus \{a \approx b\}}{\bot} \quad \text{(CONTRADICTION}_1\text{)} \quad \text{si } \sigma(a) \not\equiv \sigma(b) \qquad \frac{\Gamma \uplus \{a \not\approx b\}}{\bot} \quad \text{(CONTRADICTION}_2\text{)} \quad \text{si } \sigma(a) \equiv \sigma(b)$$

$$\frac{\Gamma \uplus \{a \approx b\}}{\Gamma\theta} \quad \text{(SOLVE)} \quad \text{si } \theta = \text{solve}(a \approx b) \neq \bot$$

FIG. 1.2 – *Procédure de décision pour une théorie* SHOSTAK

Soient $\mathcal{T}_0$ une théorie equationnelle pure et $\mathcal{T}_1 \cdots \mathcal{T}_n$ des théories Shostak, respectivement définies sur les signatures $\Sigma_1 \cdots \Sigma_n$, toutes disjointes deux à deux. On s'intéresse maintenant à une procédure de décision pour la théorie $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n$.

La procédure est décrite par le système d'inférences de la figure 1.3 (où la règle NORM$_i$ est explicitée par la suite) qui décrit l'évolution de l'état de l'algorithme, ce dernier étant représenté par une *configuration* comme définie dans la section 1.2.

L'algorithme de SHOSTAK optimise celui de NELSON et OPPEN comme suit :

Un partage de variable : lors de l'abstraction du sous-terme $a|_\pi$ de a, ce dernier peut-être remplacé par une variable existante connue pour être égale à $a|_\pi$ dans une des théorie ;

Une recherche syntaxique des équations lors de la saturation : la grande optimisation de l'algorithme de SHOSTAK est de proposer un moyen efficace pour générer les équations lors de la phase de saturation. Un premier exemple avait été donné par NELSON et OPPEN dans leur algorithme de *Congruence Closure*[4][11] qui décide de l'égalité pour une théorie équationnelle pure. SHOSTAK découvrit[15] alors qu'il est possible d'exporter le principe de la *Congruence Closure* aux théories portant son nom. Le principe est de maintenir une structure d'*Union-Find* grâce à laquelle il est possible de déterminer si $x \approx y$ simplement en vérifiant si $\text{find}(x) \equiv \text{find}(y)$.

Dans l'algorithme présenté ici, la deuxième optimisation est représentée par une phase de normalisation des équations (NORM$_i$) qui doit permettre de déduire syntaxiquement le maximum d'égalités entre variables. Pour cela, pour chaque théorie $\mathcal{T}_i$, un opérateur $\mathcal{N}_i$ devant respecter les conditions ci-dessous est fourni :

Terminaison $\exists k.\mathcal{N}_i^k(\phi_i,\Delta) \equiv \mathcal{N}_i^{k+1}(\phi_i,\Delta)$;

Equisatisfiabilité $\mathcal{T}_i \Vdash \phi_i \wedge \Delta \Leftrightarrow \mathcal{N}_i(\phi_i,\Delta) \wedge \Delta$;

Complétude Si $\mathcal{T}_i, \phi_i, \Delta \Vdash x \approx y$ et $\Delta(x) \not\equiv \Delta(y)$, alors, il existe un entier k et un terme a tels que $\mathcal{N}_i^k(\phi_i,\Delta)$ contient les équations $x \approx a$ et $y \approx a$.

4. Une description de l'algorithme de NELSON et OPPEN est donnée en annexe B

$$\frac{\Delta\ ;\ \Gamma \uplus \{a \star b\}\ ;\ \phi_0 \cdots \phi_i \cdots \phi_n}{\Delta\ ;\ \Gamma \cup \{a[z]_\pi \star b\}\ ;\ \phi_0 \cdots \phi_i \cup \{z \approx a|_\pi\} \cdots \phi_n}$$ (ABSTRACT$_i$) si $a|_\pi \notin X$, $a|_\pi$ est pur de pureté maximale dans a et z est une variable fraîche.

$$\frac{\Delta\ ;\ \Gamma \uplus \{x \star y\}\ ;\ \phi_0 \cdots \phi_n}{\Delta \cup \{x \star y\}\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_n}$$ (ARRANGE)

$$\frac{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_n}{\bot}$$ (CONTRADICT$_i$) si $\Delta \wedge \phi_i$ est insatisfiable.

$$\frac{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_i \cup \{x \approx a, y \approx a\} \cdots \phi_n}{\Delta \cup \{x \approx y\}\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_i \cup \{x \approx a, y \approx a\} \cdots \phi_n}$$ (TDEDUCT$_i$) si $\Delta(x) \not\equiv \Delta(y)$.

$$\frac{\Delta\ ;\ \Gamma \uplus \{a \star b\}\ ;\ \phi_0 \cdots \phi_i \cup \{z \approx c\} \cdots \phi_n}{\Delta\ ;\ \Gamma \cup \{a[z]_\pi \star b\}\ ;\ \phi_0 \cdots \phi_i \cup \{z \approx c\} \cdots \phi_n}$$ (TSHARE$_i$) si $a|_\pi \notin X$, $a|_\pi$ est pur de pureté maximale dans a et $\sigma_i(\Delta(a|_\pi)) \equiv c$.

$$\frac{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_i \cdots \phi_n}{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots \mathcal{N}_i(\phi_i,\Delta) \cdots \phi_n}$$ (NORM$_i$) si $\mathcal{N}_i(\phi_i,\Delta) \not\equiv \phi_i$.

FIG. 1.3 – *Procédure de décision pour une union de théories Shostak*

La figure 1.4 présente les règles de normalisation. Dans le cas d'une théorie équationnelle pure (i.e. pour $\mathcal{T}_0$), il suffit de prendre NORM$_i$ = SUBST$_i$. (On récupère alors la phase de normalisation de l'algorithme de *Congruence Closure*.) Dans celui d'une théorie *Shostak*, il faut alors prendre NORM$_i$ = CANONIZE$_i$ $\oplus$ SOLVE$_i$ $\oplus$ SUBSTITUTE$_i$ [5].

On peut alors énoncer le théorème justifiant que le système d'inférence est une procédure de décision pour $\mathcal{T}$:

Théorème 1.3.4 *[4, 14] Soit Γ un ensemble de Σ-équations. On note $\mathcal{C}_\Gamma$ la configuration $< \emptyset\ ;\ \Gamma\ ;\ \emptyset \cdots \emptyset]$. Alors, Γ est satisfiable si et seulement si il existe une configuration $\mathcal{C}$ propre, irréductible pour $\Rightarrow$, telle que $\mathcal{C}_\Gamma \Rightarrow^* \mathcal{C}$.*

1.4 Stratégies

Les stratégies considérées ici ont pour but, dans le cas d'une éventuelle implantation, d'augmenter considérablement les performances en restreignant les suites de réductions possibles tout en ne modifiant pas la sémantique opérationnelle du système :

Définition 1.4.1 (Stratégie) Une stratégie est n'importe quel terme régulier défini par la grammaire :

$$e ::= r \mid e + e \mid e^* \mid e.e \mid e \oplus e$$

où r décrit l'ensemble des règles du système d'inférence.

5. $R_1 \oplus R_2$ indique l'application de la règle R_1 ou R_2 en donnant priorité à R_1. $\oplus$ est bien entendu associatif à gauche.

$$\frac{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_i \uplus \{x \approx a\} \cdots \phi_n}{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_i \cup \{x \approx \Delta(a)\} \cdots \phi_n} \quad \begin{array}{l}((\text{SU})\text{BST}_i)\\ \text{si } \Delta(a) \not\equiv a.\end{array}$$

$$\frac{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_i \uplus \{x \approx a\} \cdots \phi_n}{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_i \cup \{x \approx \sigma_i(a)\} \cdots \phi_n} \quad \begin{array}{l}((\text{CA})\text{NONIZE}_i)\\ \text{si } \sigma_i(a) \not\equiv a.\end{array}$$

$$\frac{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots \phi_i \cup \{x \approx a, y \approx b\} \cdots \phi_n}{\Delta\ ;\ \Gamma\ ;\ \phi_0 \cdots (\phi_i \cup \{x \approx a, y \approx b\} \cup \text{solve}_i(a = b))^2 \cdots \phi_n} \quad \begin{array}{l}((\text{SO})\text{LVE}_i)\\ \text{si } \Delta(x) \equiv \Delta(y),\ a \not\equiv b \text{ et}\\ a \approx b \text{ est } \mathcal{T}_i\text{-satisfiable.}\end{array}$$

FIG. 1.4 – *Règles de normalisation*

Définition 1.4.2 (Réduction suivant une stratégie) Soit $\mathcal{C}$ et $\mathcal{C}'$ deux configurations. $\mathcal{C}$ se réduit en $\mathcal{C}'$ suivant la stratégie e, noté $\mathcal{C} \Rightarrow_e \mathcal{C}'$, si et seulement si $\mathcal{C} \Rightarrow_e \mathcal{C}'$ est dérivable dans le système d'inférence donné en figure 1.5.

$$\frac{\mathcal{C} \Rightarrow \mathcal{C}' \text{ par application de la règle } r}{\mathcal{C} \Rightarrow_r \mathcal{C}'}$$

$$\frac{\mathcal{C} \Rightarrow_e \mathcal{C}' \quad \mathcal{C}' \Rightarrow_{e'} \mathcal{C}''}{\mathcal{C} \Rightarrow_{e.e'} \mathcal{C}''} \qquad \frac{\mathcal{C}_0 \Rightarrow_e \cdots \Rightarrow_e \mathcal{C}_n \not\Rightarrow_e}{\mathcal{C} \Rightarrow_{e^*}}$$

$$\frac{\mathcal{C} \Rightarrow_e \mathcal{C}'}{\mathcal{C} \Rightarrow_{e+e'} \mathcal{C}'} \qquad \frac{\mathcal{C} \Rightarrow_{e'} \mathcal{C}'}{\mathcal{C} \Rightarrow_{e+e'} \mathcal{C}'} \qquad \frac{\mathcal{C} \Rightarrow_e \mathcal{C}'}{\mathcal{C} \Rightarrow_{e \oplus e'} \mathcal{C}'} \qquad \frac{\mathcal{C} \not\Rightarrow_e \mathcal{C}' \quad \mathcal{C} \Rightarrow_{e'} \mathcal{C}'}{\mathcal{C} \Rightarrow_{e \oplus e'} \mathcal{C}'}$$

FIG. 1.5 – *Sémantique du langage de stratégies*

Intuitivement, $.$ et $+$ gardent leur sens usuel, $*$ représente l'application exhaustive et $\oplus$ un opérateur de choix, associatif à gauche, qui privilégie son argument de gauche. Une propriété essentielle est que toute stratégie est correcte vis-à-vis de $\Rightarrow$, i.e. que si $\mathcal{C} \Rightarrow_e \mathcal{C}'$, alors $\mathcal{C} \Rightarrow^* \mathcal{C}'$. Reste à caractériser quand une stratégie est une $\mathcal{T}$-procédure de décision :

Proposition 1.4.3 *Soit e une stratégie. Si pour tout configuration $\mathcal{C}$:*

- *il existe une configuration $\mathcal{C}'$ telle que $\mathcal{C} \Rightarrow_e \mathcal{C}'$;*
- *$\mathcal{C} \Rightarrow_e \mathcal{C}'$ implique $\mathcal{C}'$ est irréductible pour $\Rightarrow$;*

alors e est une $\mathcal{T}$-procédure de décision.

Grâce à ce système de stratégie, CONCHON et KRSTIC ont montré[4] qu'il est possible de caractériser l'algorithme de SHANKAR et RUESS[14] (Cf. figure 1.6).

$$(\text{abstraction} \,.\, (\text{Co} \oplus \text{merge} \oplus \text{infer} \oplus \text{normalize})^*)^*$$

où :

$$\begin{aligned}
\text{abstraction} &= (\text{Va}^1 \oplus \cdots \oplus \text{Va}^m) \,.\, \text{Su}_0^* \\
\text{merge} &= (\text{So}_1 \,.\, \text{Ca}_1^*) + \cdots + (\text{So}_n \,.\, \text{Ca}_n^*) \\
\text{infer} &= (\text{TDe}_0 + \cdots + \text{TDe}_n) \,.\, \text{Su}_0^* \\
\text{normalize} &= (\text{Su}_1 + \cdots + \text{Su}_n) \,.\, (\text{Su}_1^* \cdots \text{Su}_n^*) \\
\text{Va} &= (\text{TSh} \oplus \text{ASC})^* \,.\, \text{Ar} \\
\text{Tsh} &= \textstyle\sum_i \text{TSh}_i \\
\text{ASC} &= \textstyle\sum_i \text{Ab}_i \,.\, \text{Su}_i^* \,.\, \text{Ca}_i^*
\end{aligned}$$

FIG. 1.6 – *Stratégie caractérisant l'algorithme de* SHANKAR *et* RUESS

Chapitre 2

Le Calcul des Constructions avec Procédures de Décision

Il s'agit maintenant d'introduire l'algorithme de SHOSTAK au sein de la règle de conversion du Calcul des Construction (CC). Rappelons que les termes de CC sont les termes définis par :

$$t ::= \overbrace{s \in \{\star, \Box\}}^{\text{sorte}} \mid \overbrace{x \in X}^{\text{variable}} \mid \overbrace{tt}^{\text{application}} \mid \overbrace{[x:t]t}^{\text{abstraction}} \mid \overbrace{(x:t)t}^{\text{produit dépendant}}$$

On se donne une théorie de l'égalité pure $\mathcal{T}_0$ et n théories Shostak $\mathcal{T}_1 \cdots \mathcal{T}_n$, définies respectivement sur les signatures $\Sigma_0 \cdots \Sigma_n$, toutes disjointes deux à deux. On note $\mathcal{T} = \cup_i \mathcal{T}_i$ et $\Sigma = \cup_i \Sigma_i$. Pour chaque théorie $\mathcal{T}_i$, on note $\mathcal{A}_i = \{a_j^i\}_{1 \leq j \leq \alpha;\ \alpha \in \omega \cup \{\omega\}}$ l'ensemble des axiomes de $\mathcal{T}_i$. De plus, on associe à chaque symbole de fonction une sorte s_f de CC et un terme clos τ_f de CC de la forme $\tau_f = T_1 \to \cdots \to T_n \to U$ avec $n = \text{arite}(f_i)$.

Dans toute le reste de la section, on se place dans le Calcul des Constructions étendu par Σ. (I.e. dans le Calcul des Constructions Algébriques[2, 3] (CCA) paramétré par un système de récriture vide sur la signature Σ) Les termes de CCA étendent ceux de CC comme suit :

$$t ::= \cdots \mid f \in \Sigma.$$

Les sortes et types choisis pour les symboles de fonction doivent être tels que :

$$a \in \bigcup_i \mathcal{A}_i \Rightarrow \emptyset \vdash_{CC} a : \star.$$

Enfin, on note eq le prédicat de l'égalité de LEIBNIZ, i.e. le plus petit prédicat de type $T_{\text{eq}} = (A : \star)(A \to A \to \star)$ vérifiant:

$$(A : \star)(x, y : A).(\text{eq}\ A\ x\ y) \to ((P : A \to \star).(Px) \to (Py)).$$

$(\text{eq}\ A\ x\ y)$ sera également noté $x \approx_A y$ par la suite.

Les notions de positions, d'environnements de typage et de substitutions gardent leur sens usuel de CC. Un terme est *pré-algébrique* si et seulement si son symbole de tête est un symbole de fonction. Un terme pré-algébrique est totalement appliqué s'il est de la forme $f\overrightarrow{x}$ avec $|\overrightarrow{x}| = \text{arite}(f)$. Un terme est *algébrique* si tous ses sous-termes (au sens large) différents d'une variable sont pré-algébriques totalement appliqués. Un terme algébrique est clairement assimilable à un terme de premier ordre sur Σ.

Les meta-variables π, ρ... désigneront des positions ; θ, ξ... des substitutions et Γ, Δ... des environnements de typage. Le système de typage de CC est rappelé en figure 2.1.

$$\frac{}{\vdash \star : \Box}\ (\textsc{Axiom}) \qquad \frac{\vdash \tau_f : s_f}{\vdash f : \tau_f}\ (\textsc{Symbol}) \qquad \frac{\Gamma \vdash T : s}{\Gamma, x : T \vdash x : T}\ \begin{array}{l}(\textsc{Variables})\\ (x \notin \mathrm{dom}(\Gamma))\end{array}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash U : s}{\Gamma, x : U \vdash t : T}\ \begin{array}{l}(\textsc{Weak})\\ (x \notin \mathrm{dom}(\Gamma))\end{array} \qquad \frac{\Gamma \vdash U : s \quad \Gamma, x : U \vdash V : s'}{\Gamma \vdash (x : U)V : s'}\ (\textsc{Product})$$

$$\frac{\Gamma, x : U \vdash v : V \quad (x : U)V : s}{\Gamma \vdash [x : U]v : (x : U)V}\ (\textsc{Abtraction})$$

$$\frac{\Gamma \vdash t : (x : U)V \quad \Gamma \vdash u : U}{\Gamma \vdash tu : V\{x \to u\}}\ (\textsc{Application})$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : s \quad \Gamma \vdash T' : s' \quad T \to^*_\beta \leftarrow T'}{\Gamma \vdash t : T'}\ (\textsc{Conversion})$$

FIG. 2.1 – *Typage du Calcul des Constructions avec termes algébriques*

2.1 Description du nouveau calcul

On désire maintenant intégrer une procédure de décision au sein de la règle de conversion de CC. Comme ces procédures ne savent travailler que sur des termes de 1er ordre, il faut trouver un moyen d'étendre ces dernières aux termes de CC. En résumé, pour vérifier si deux termes t et u sont convertibles sous un environnement de typage Γ, les actions suivantes vont être effectuées :

- u et t sont respectivement β-réduits vers t' et u' en un nombre quelconque de pas ;
- extraction de l'environnement de typage des équations Δ closes de 1er ordre, i.e. des formules closes ;
- transformation des parties non algébriques de u' et t' afin d'obtenir des termes algébriques u'' et t''. Pour cela, de nouveaux symboles de fonction vont être introduits pour chaque construction non algébrique et les dites constructions seront remplacées par ces symboles. E.g., pour le produit, un symbol $pi/2$ sera introduit et un terme de la forme ab sera transformé en un autre de la forme $pi(\hat{a},\hat{b})$ [1] ;
- Vérification à l'aide de l'algorithme de SHOSTAK de la validité de $\mathcal{T} \Vdash \Delta \to (u'' \approx t'')$.

Pour purifier les termes de CC en terme du premier ordre, il nous faut tout d'abord introduire de nouveau symbole de fonctions, tous distincts deux à deux et n'apparaissant pas dans Σ :

- pour chaque sorte s, un symbole de constante $\hat{s}$;
- pour chaque entier n, un symbole de constante $\hat{n}$;
- pour chaque symbole de fonction f de Σ, un symbole de constante $\hat{f}$;
- trois nouveaux symboles app, pi et $lamb$, tous d'arité 2.

On note Σ'_0 l'ensemble des nouveaux symboles introduits.

On peut alors définir le processus de purification des termes de CC en termes du 1er ordre ainsi que celui d'extraction des équations d'un environnement de typage.

1. $\hat{a}$ et $\hat{b}$ dénotent les transformés de a et b par ce processus

Définition 2.1.1 ("Algébraïsation") Soit t un terme de CC. On définit l'**algébraïsation** de t, notée $\alpha(t)$, par :

- $\alpha(s) = \hat{s}$
- $\alpha([x : t_1]t_2) = lamb(\alpha(t_1), \alpha(t_2))$
- $\alpha((x : t_1]t_2) = pi(\alpha(t_1), \alpha(t_2))$
- $\alpha(x) = \begin{cases} x \text{ si } x \text{ est libre dans } t \\ \hat{n} \text{ où } n \text{ est l'indice de DE BRUIJN de cette occurence de } x \text{ dans } t \text{ sinon} \end{cases}$
- $\alpha(f\overrightarrow{u}) = \begin{cases} f(\alpha(u_1), \cdots, \alpha(u_n)) \text{ si } |\overrightarrow{u}| = \text{arite}(f) \\ app(\cdots(app(app(\hat{f}, \alpha(u_1)), \alpha(u_2)), \cdots), \alpha(u_n)) \end{cases}$
- $\alpha(t\overrightarrow{u}) = app(\cdots(app(app(\alpha(t), \alpha(u_1)), \alpha(u_2)), \cdots), \alpha(u_n))$ si $t \notin \Sigma$

L'intérêt d'une telle phase est de conserver intégralement les parties non algébriques d'un terme de CC grâce à des symboles de fonctions qui resteront neutres vis-à-vis de l'algorithme de SHOSTAK.

Définition 2.1.2 (Extraction des équations) On définit eq, fonction sur les environnements, qui extrait syntaxiquement, de l'environnement de typage passé en argument, les littéraux du 1er ordre sur $\cup_i \Sigma_i$ par :

- $\text{eq}(\emptyset) = \emptyset$;
- $\text{eq}([x : (\overrightarrow{z} : \overrightarrow{T})(a \approx_T b)]\,\Gamma) = \{a \approx b\} \cup \text{eq}(\Gamma)$ si a et b sont algébriques et clos
- $\text{eq}([x : t]\,\Gamma) = \text{eq}(\Gamma)$ sinon.

On peut désormais définir le nouveau calcul.

Définition 2.1.3 Soit t et u deux termes de CC et Γ un environnement de typage. t et u sont **équivalent sous** Γ, noté $t \sim_\Gamma u$, si et seulement si $\mathcal{T} \Vdash \text{eq}(\Gamma) \to (\alpha(t) \approx \alpha(u))$.

Remarque 2.1.4 Formellement, ce qu'il faut réellement vérifier dans la définition précédente est la validité de $\text{eq}(\Gamma) \to (\alpha(t) \approx \alpha(u))$ dans la théorie $\mathcal{T}'$, cette dernière possédant exactement les mêmes axiomes que $\mathcal{T}$ mais étant définie sur la signature $\Sigma_0 \uplus \Sigma'$.

Définition 2.1.5 (Relation de conversion) Soient T et U deux termes de CC ; Γ et Γ' deux environnements de typage valides :

- $T\ \mathbb{C}_\Gamma\ U$ si et seulement si $T \sim_\Gamma U$ et $\exists s, s'.(\Gamma \vdash T : s \wedge \Gamma \vdash U : s')$;
- $\Gamma\ \mathbb{C}\ \Gamma'$ si et seulement si $\Gamma = \overrightarrow{x} : \overrightarrow{T}$, $\Gamma' = \overrightarrow{x} : \overrightarrow{T'}$ et, soit $|\overrightarrow{x}| = 0$ ou il existe j tel que $T_j\ \mathbb{C}_{x_1:T_1\cdots x_{j-1}:T_{j-1}}\ T'_j$ et $\forall i \neq j.T_i = T'_i$.

Définition 2.1.6 Le nouveau calcul est obtenu en remplaçant la règle de conversion par :

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : s \quad \Gamma \vdash T' : s' \quad T \to^*_\beta \sim_\Gamma \leftarrow^*_\beta T'}{\Gamma \vdash t : T'} \quad \text{(CONVERSION)}$$

2.2 Quelques résultats métathéoriques

On note $\text{CC}_\mathcal{T}$ ce nouveau calcul. On peut facilement vérifier que $\text{CC}_\mathcal{T}$ est bien une extension de CC.

Proposition 2.2.1 *$\sim_\Gamma$ est une relation réflexive, transitive et symétrique.*

PREUVE Corollaire immédiat de la réflexivité, transitivité et symétrie de $\approx$. ∎

Proposition 2.2.2 $\Gamma \vdash_{CC} t : T \Rightarrow \Gamma \vdash_{CC_\mathcal{T}} t : T$.

PREUVE Par induction sur la dérivation de $\Gamma \vdash t : T$. Découle immédiatement de la réflexivité de $\sim_\Gamma$. ∎

Proposition 2.2.3 *Un bon nombre de propriétés basiques de CC[1] sont toujours valables :*

- ***(Sous-terme)*** *Les sous-termes d'un terme typable sont typables ;*
- ***(Environnement valide)*** *Soit $\Gamma = \overrightarrow{x} : \overrightarrow{T}$ un environnement valide. Alors, pour tout i, il existe une sorte s telle que $x_1 : T_1 \cdots x_{i-1} : T_{i-1} \vdash x_i : T_i$ et $x_1 : T_1 \cdots x_i : T_i \vdash x_i : T_i$;*
- ***(Remplacement)*** *Si $\Gamma, y : W, \Gamma' \vdash t : T$, $y \in X$ et $z \in X - \mathrm{dom}(\Gamma, y : W, \Gamma')$, alors $\Gamma, z : W, \Gamma'\{y \to z\} \vdash t\{y \to z\} : T\{y \to z\}$;*
- ***(Affaiblissement)*** *Si $\Gamma \vdash t : T$, $\Gamma \subseteq \Gamma'$ et Γ' est un environnement de typage valide, alors $\Gamma' \vdash t : T$;*
- ***(Transitivité)*** *Si Γ et Δ sont deux environnements de typage valides tels que $\Delta \vdash \Gamma$ (i.e. $\forall x \in \mathrm{dom}(\Gamma).\Delta \vdash x : x\Gamma$), alors $\Gamma \vdash t : T \Rightarrow \Delta \vdash t : T$;*
- ***(Permutation faible)*** *Si $\Gamma, y : A, z : B, \Gamma' \vdash t : T$ et $\Gamma \vdash B : s$ alors $\Gamma, z : B, y : A, \Gamma' \vdash t : T$.*
- ***(Correction des types)*** *Si $\Gamma \vdash t : T$, alors $T = \Box$ ou $\Gamma \vdash T : s$.*

PREUVE Preuves très similaires à celles dans [2], ces dernières ne dépendant pas du fait que la relation de conversion dépende de l'environnement de typage. ∎

Proposition 2.2.4 (Conversion pour les environnements) *Si $\Gamma \vdash t : T$ et $\Gamma \mathbb{C} \Gamma'$, alors $\Gamma' \vdash t : T$.*

Définition 2.2.5 (Substitution bien typée) Soient Γ et Δ deux environnements valides. Une substitution est **bien typée entre Γ et Δ**, $\theta : \Gamma \rightsquigarrow \Delta$, si et seulement si $\forall x \in \mathrm{dom}(\Gamma).\Delta \vdash x\theta : x\Gamma\theta$.

Proposition 2.2.6 (Compatibilité de la substitution vis-à-vis de α) *Soient t un terme de CC et θ une substitution. Alors, $\alpha(t\theta) = \alpha(t)\alpha(\theta)$ où $\alpha(\{x_1 \to t_1 \cdots x_n \to t_n\}) = \{x_1 \to \alpha(t_1) \cdots x_n \to \alpha(t_n)\}$.*

PREUVE Simple induction sur la structure de t. ∎

Proposition 2.2.7 (Stabilité par substitution) *Soient Γ, Δ, t, T et θ tels que $\Gamma \vdash t : T$ et $\theta : \Gamma \rightsquigarrow \Delta$. Si tous les $\mathcal{T}$-modèles satisfaisant les équations de $\mathrm{eq}(\Delta)$ satisfont les équations de $\mathrm{eq}(\Gamma)$, alors $\Gamma \vdash t\theta : T\theta$.*

PREUVE Par induction sur la dérivation de $\Gamma \vdash t : T$. Le seul cas problématique est bien sur la règle de conversion. Si on a la dérivation

$$\frac{\dfrac{\Pi_1}{\Gamma \vdash t : T} \quad \dfrac{\Pi_2}{\Gamma \vdash T : s} \quad \dfrac{\Pi_3}{\Gamma \vdash T' : s'} \quad T \to_\beta^* U \sim_\Gamma U' \leftarrow_\beta^* T'}{\Gamma \vdash t : T'}$$

il suffit de montrer que $T\theta \to_\beta^* \sim_\Delta \leftarrow_\beta^* T'\theta$ pour terminer la démonstration. Par substitutivité de β, on a $T\theta \to_\beta^* U\theta$ et $T'\theta \to_\beta^* U'\theta$. Reste à montrer $U\theta \sim_\Delta U'\theta$, i.e. $\mathcal{T} \Vdash \mathrm{eq}(\Delta) \to \alpha(U\theta) \approx \alpha(U'\theta)$.

Soient $\mathcal{M}$ un $\mathcal{T}$-modèle de domaine M et ρ une assignation satisfaisant les équations de $\mathrm{eq}(\Delta)$. Supposons que $\alpha(U\theta) \approx \alpha(U'\theta)$ n'est pas satisfaite par $\mathcal{M}$ et ρ, i.e. $[\![\alpha(U\theta)]\!]_\rho^\mathcal{M} \neq_M [\![\alpha(U'\theta)]\!]_\rho^\mathcal{M}$. Soit ρ' l'assignation qui à tout x associe $\rho(x)$ si $x \notin \mathrm{dom}(\theta)$ et $[\![\alpha(x\theta)]\!]_\rho^\mathcal{M}$ sinon. Comme les équations de $\mathrm{eq}(\Delta)$ sont closes, $[\![\mathrm{eq}(\Delta)]\!]_{\rho'}^\mathcal{M} =_M [\![\mathrm{eq}(\Delta)]\!]_\rho^\mathcal{M}$. Donc, $[\![\mathrm{eq}(\Delta)]\!]_{\rho'}^\mathcal{M}$ s'évalue en vrai, et par hypothèse de la proposition, $[\![\mathrm{eq}(\Gamma)]\!]_{\rho'}^\mathcal{M}$ s'évalue est vrai. Comme $\mathcal{T} \Vdash \mathrm{eq}(\Gamma) \to (\alpha(U) \approx \alpha(U'))$, $[\![U]\!]_{\rho'}^\mathcal{M} =_M [\![U']\!]_{\rho'}^\mathcal{M}$.

Or, par simple induction sur la structure de $\alpha(U)$, il est facile de voir que $[\![\alpha(U)]\!]_{\rho'}^\mathcal{M} =_M [\![\alpha(U\theta)]\!]_\rho^\mathcal{M}$. La plupart des cas sont immédiats, le plus souvent à l'aide de la proposition précédente. Seul $\alpha(U) \equiv x$ mérite un peu plus d'attention :

- si $x \in \mathrm{dom}(\theta)$, alors $[\![\alpha(U)]\!]_{\rho'}^\mathcal{M} = \rho'(x) = [\![\alpha(x\theta)]\!]_\rho^\mathcal{M} = [\![\alpha(U\theta)]\!]_\rho^\mathcal{M}$;
- sinon, $[\![\alpha(U)]\!]_{\rho'}^\mathcal{M} = \rho'(x) = \rho(x) = [\![x]\!]_\rho^\mathcal{M} = [\![\alpha(U\theta)]\!]_\rho^\mathcal{M}$.

De même, $[\![U']\!]_{\rho'}^\mathcal{M} =_M [\![U'\theta]\!]_\rho^\mathcal{M}$. Ainsi, $[\![U]\!]_{\rho'}^\mathcal{M} \neq_M [\![U']\!]_{\rho'}^\mathcal{M}$, ce qui est impossible.

Définition 2.2.8 On définit les ensembles de termes suivants :

- l'ensemble $\mathbb{K}$ des *types de prédicat* définit par $\mathbb{K} = \{K \mid \Gamma \vdash K : \Box\}$. On peut remarquer que tout terme $K \in \mathbb{K}$ est de la forme $(\overrightarrow{x} : \overrightarrow{T})\star$;

- l'ensemble $\mathcal{K}$ des termes de la forme $(\overrightarrow{x} : \overrightarrow{T})\star$ appelés *kinds* ;
- l'ensemble $\mathbb{P}$ des *prédicats* définit par $P = \{T \mid \Gamma \vdash T : K \wedge \Gamma \vdash K : \square\}$;
- l'ensemble $\mathbb{O}$ des objets définit par $\mathbb{O} = \{t \mid \Gamma \vdash t : T \wedge \Gamma \vdash T : \star\}$.

Définition 2.2.9 (Chapeau d'un terme de $\mathtt{CC}_\tau$) On augmente les termes de $\mathtt{CC}_\tau$ par le symbole Ω. Pour chaque terme t, on définit $\mathcal{A}(t)$ par :

$$\mathcal{A}(t) = \left\{ \rho \in \mathrm{Pos}(t) \;\middle|\; \left\{ \begin{array}{l} t|_\rho \text{ est un terme algébrique} \\ \text{pour tout préfixe stricte de } \rho,\ t|_\rho \text{ n'est pas algébrique} \end{array} \right. \right\}.$$

Le chapeau de t, noté $\mathrm{cap}(t)$, est défini par $\mathrm{cap}(t) = t[\Omega]_{\mathcal{A}(t)}$.

Lemme 2.2.10 (Conservation du chapeau non algébrique) *Soit t et t' deux termes de $\mathtt{CC}_\tau$ et Γ un environnement de typage tels que $t \sim_\Gamma t'$. Alors, $\mathrm{cap}(t) = \mathrm{cap}(t')$.*

Dans toute la suite, on note $\to_h$ la β-réduction de tête et $\to_{\not h}$ la β-réduction interne. On se propose alors, tout comme dans [3], de montrer une batterie de petits lemmes sur les *kinds* permettant de montrer la subject reduction pour $\to_h$ sur les types.

Lemme 2.2.11 *On appelle une mauvaise kind tout terme de la forme $[y : W]K$ ou wK avec $K \in \mathcal{K}$.*

1. *Si $\Gamma \vdash t : \square$ alors $t \in \mathcal{K}$;*
2. *Si $K \in \mathcal{K}$ et $\Gamma \vdash K : L$ alors $L = \square$;*
3. *Une mauvaise kind n'est pas typable ;*
4. *Si $t \to_h t'$ et t' contient une mauvaise kind, alors t aussi ;*
5. *Si $\Gamma \vdash T : s$ et $T \to_\beta^* K \in \mathcal{K}$ alors $s = \square$ et $T \in \mathcal{K}$;*
6. *Si $T\ \mathbb{C}_\Gamma^*\ K$ et $\Gamma \vdash K : \square$ alors $\Gamma \vdash T : \square$ et $T \in \mathcal{K}$;*
7. *Si $K = (\overrightarrow{x} : \overrightarrow{T})\star$, $K' = (\overrightarrow{x'} : \overrightarrow{T'})\star$ et $K\,\mathbb{C}_\Gamma^*\,K'$, alors $|x| = |x'|$ et $\forall i. T_i \mathbb{C}_{\Gamma_i}^* T_i'$ où $\Gamma_i = \Gamma, x_1 : T_1, \cdots, x_i : T_i$;*
8. *Si $t\ \mathbb{C}_\Gamma^*\ t'$ et $\Gamma \vdash t : \star$, alors $\Gamma \vdash t' : \star$.*

PREUVE

1. Par inversion, comme aucune conversion ne peut prendre place ($\square$ n'est pas typable), t est de la forme $(\overrightarrow{x} : \overrightarrow{T})\star$.
2. Par induction sur la taille de K. Si $K = \star$, alors par inversion $L = \square$. (Aucune conversion ne peut prendre place, $\square$ n'étant pas typable.) Si $K = (x : T)K'$ avec $K' \in \mathcal{K}$, alors, par inversion, $\Gamma, x : T \vdash K : s$ et $s\ \mathbb{C}_\Gamma^*\ L$. Par hypothèse d'induction, $s = \square$. Ainsi, s n'étant pas typable, $s\ \mathbb{C}_\Gamma^+\ L$ est impossible et donc $L = \square$.
3. (a) Supposons $\Gamma \vdash [y : W]K : T$. Par inversion, $\Gamma, y : W \vdash K : L$ et $\Gamma \vdash (y : W)L : s$. Par (2), $L = \square$ et par propriété de typage des sous formules, $(y : W)L$ ne peut pas être typé.
   (b) Supposons $\Gamma \vdash wK : T$. Par inversion, $\Gamma \vdash w : (x : L)V$ et $\Gamma K : L$. Par (2), $L = \square$ et par propriété de typage des sous formules, $(x : L)V$ ne peut pas être typé.
4. Posons $t = ([x : U]v)u$ et supposons que $t' = v\{x \to u\}$ contient une mauvaise *kind* L. Si u ou v contiennent une mauvaise *kind*, alors t également. Ainsi, supposons que ni u ni v ne contiennent aucune mauvaise *kind*.
   (a) si $L = wK$, alors v doit avoir un sous terme de la forme $w'(\overrightarrow{x} : \overrightarrow{T})x$ et u doit appartenir à $\mathcal{K}$. Ainsi, t possède également une mauvaise *kind*.
   (b) si $L = [y : W]K$, alors v doit avoir un sous terme de la forme $[y : W](\overrightarrow{x} : \overrightarrow{T})x$ et u doit appartenir à K. Ainsi, t possède également une mauvaise *kind*.
5. Par standardisation, il existe $K' \in \mathcal{K}$ tel que $T \to_h^* K' \to_{\not h}^* K$. Supposons $T \to_h^* T' = ([x : U]v)u \to_h v\{x \to u\} = K'$. Si u ou v sont des *kinds*, alors T' est clairement une mauvaise *kind*. Sinon, v doit contenir un sous terme de la forme $(\overrightarrow{x} : \overrightarrow{T})\star$ et u doit appartenir à $\mathcal{K}$. Dans ce cas, T' contient également une mauvaise *kind*. Ainsi, dans tous les cas, T' contient une mauvaise *kind*. Par (4), T contient également une mauvaise *kind* et par (3), il n'est pas typable. Ainsi, $T = K' \to_{\not h}^* K$ avec $T \in \mathcal{K}$; et par (2), $s = \square$.

6. Par induction sur le nombre de conversions entre T et K. Supposons que $\Gamma \vdash T : s$, $T \to_\beta^* U \sim_\Gamma K' \leftarrow_\beta^* K$ et $\Gamma \vdash K : \Box$. Comme $K' \in \mathcal{K}$, par lemme 2.2.10, $U \in \mathcal{K}$. Par (5), $s = \Box$ et $T \in \mathcal{K}$.
7. Conséquence immédiate de (6).
8. Conséquence immédiate de (7). ∎

Définition 2.2.12 Soit $\mathcal{R}$ un système de récriture sur les termes de $\mathtt{CC}_\tau$. Un environnement $\Gamma = \overrightarrow{x} : \overrightarrow{T}$ se récrit par $\mathcal{R}$ en un environnement $\Gamma' = \overrightarrow{x'} : \overrightarrow{T'}$, noté $\Gamma \to_\mathcal{R} \Gamma'$, si et seulement si $\overrightarrow{x} = \overrightarrow{x'}$ et, soit $|\overrightarrow{x}| = 0$, soit il existe j tel que $T_j \to_\mathcal{R} T'_j$ et pour tout $i \neq j$, $T_i = T'_i$.

Théorème 2.2.13 (Subject reduction pour β^{P_ω}) *[3] On note β^{P_ω} la réduction de β aux rédex de la forme $([x : T]U)t$ où U est un prédicat. Alors, β^{P_ω} préserve le typage.*

PREUVE La preuve se fait par induction sur la dérivation de $\Gamma \vdash t : T$, en démontrant en parallèle que (i) si $\Gamma \to_{\beta^{P_\omega}} \Gamma'$, alors $\Gamma' \vdash t : T$ et (ii) si $t \to_{\beta^{P_\omega}} t'$ alors $\Gamma \vdash t' : T$. Le cas non trivial est celui d'une formation d'un rédex de tête par l'application de la règle APP.

Ainsi, on a une réduction de tête de la forme $([x : U']v)u \to_{\beta^{P_\omega}} v\{x \to u\}$ avec $\Gamma \vdash ([x : U']v)u : T$. Par inversion de la règle APP, il existe U et V tels que $\Gamma \vdash [x : U']v : (x : U)V$ et $\Gamma \vdash u : U$ avec $V\{x \to u\}\ \mathbb{C}_\Gamma^*\ T$. Ensuite, par inversion par la règle LAMB de $\Gamma \vdash [x : U']v : (x : U)V$, il existe V' tel que $\Gamma, x : U' \vdash v : V'$ et $(x : U')V'\ \mathbb{C}_\Gamma^*\ (x : U)V$.

Par le lemme 6 (6) et (7), on a $U\ \mathbb{C}_\Gamma^*\ U'$ et $V\ \mathbb{C}_{\Gamma,\, x:U}^*\ V'$. Par conversion sur les environnements et sur les types, on obtient alors $\Gamma, x : U \vdash v : V$. Comme $\theta = \{x \to u\}$ est une substitution valide de Γ dans Γ vérifiant les hypothèses du lemme 2.2.7, on obtient par substitution que $\Gamma \vdash v\{x \to u\} : V\{x \to u\}$. Enfin, par conversion, $\Gamma \vdash v\{x \to u\} : T$. ∎

Lemme 2.2.14 (Compatibilité du produit) *Soient $(x : T_1)U_1$ et $(x : T_2)U_2$ deux termes de $\mathtt{CC}$ et Γ un environnement de typage tels que $(x : T_1)U_1\ \mathbb{C}_\Gamma\ (x : T_2)U_2$. Alors, $T_1\ \mathbb{C}_\Gamma\ T_2$ et $U_1\ \mathbb{C}_{\Gamma,\, x:T_1}\ U_2$.*

PREUVE En décomposant $\mathbb{C}_\Gamma$, on obtient qu'il existe deux termes $(x : T'_1)U'_1$ et $(x : T'_2)U'_2$ tels que $(x : T_1)U_1 \to_\beta^* (x : T'_1)U'_1$, $(x : T_2)U_2 \to_\beta^* (x : T'_2)U'_2$ et $(x : T'_1)U'_1 \sim_\Gamma (x : T'_2)U'_2$. Or :

$$
\begin{array}{rcl}
(x : T'_1)U'_1 \sim_\Gamma (x : T'_2)U'_2 & \Rightarrow & \mathcal{T} \Vdash \mathrm{eq}(\Gamma) \to \alpha((x : T'_1)U'_1) \approx \alpha((x : T'_2)U'_2) \\
& \Rightarrow & \mathcal{T} \Vdash \mathrm{eq}(\Gamma) \to pi(\alpha(T'_1), \alpha(U'_1)) \approx pi(\alpha(T'_2), \alpha(U'_2)) \\
(*) & \Rightarrow & \left\{ \begin{array}{l} \mathcal{T} \Vdash \mathrm{eq}(\Gamma) \to \alpha(T'_1) \approx \alpha(U'_1) \\ \mathcal{T} \Vdash \mathrm{eq}(\Gamma) \to \alpha(T'_2) \approx \alpha(U'_2) \end{array} \right. \\
& \Rightarrow & \left\{ \begin{array}{l} \mathcal{T} \Vdash \mathrm{eq}(\Gamma) \to \alpha(T'_1) \approx \alpha(U'_1) \\ \mathcal{T} \Vdash \mathrm{eq}(\Gamma, x : U_1) \to \alpha(T'_2) \approx \alpha(U'_2) \end{array} \right. \\
& \Rightarrow & \left\{ \begin{array}{l} \alpha(T'_1) \sim_\Gamma \alpha(T'_2) \\ \alpha(U'_1) \sim_{\Gamma,\, x:U_1} \alpha(U'_2) \end{array} \right.
\end{array}
$$

La propriété $(*)$ est une conséquence immédiate du fait que pi est un symbole de fonction non interprété.

Enfin, par hypothèse, il existe une sorte s telle que pour tout i, $\Gamma \vdash (x : T_i)U_i : s$. Par propriété de typage des sous-termes, T_i et U_i sont typables. Par règle de formation du produit les types des T_i et des U_i sont nécessairement des sortes. Ainsi, $T_1\ \mathbb{C}_\Gamma\ T_2$ et $U_1\ \mathbb{C}_{\Gamma,\, x:T_1}\ U_2$. ∎

Lemme 2.2.15 (Commutation de β et $\sim_\Gamma$) *Soit Γ un environnement de typage, et t, u et v trois termes tels que $t \sim_\Gamma u$ et $t \to_\beta^* v$. Alors, il existe w tel que $u \to_\beta^* w$ et $v \sim_\Gamma w$.*

Théorème 2.2.16 (Subject reduction pour β) *Si $\Gamma \vdash t : T$ et $t \to_\beta t'$, alors $\Gamma \vdash t' : T$.*

PREUVE Comme pour le théorème 2.2.13, la preuve se fait par induction sur la dérivation de $\Gamma \vdash t : T$, en démontrant en parallèle que (i) si $\Gamma \to_\beta \Gamma'$, alors $\Gamma' \vdash t : T$ et (ii) si $t \to_\beta t'$ alors $\Gamma \vdash t' : T$. Le cas non trivial est celui d'une formation d'un rédex de tête par l'application de la règle APP.

Dans ce cas, on a une réduction de tête de la forme $([x : U']v)u \to_\beta v\{x \to u\}$ avec $\Gamma \vdash ([x : U']v)u : T$. Par inversion de la règle APP, il existe U et V tels que $\Gamma \vdash [x : U']v : (x : U)V$ et $\Gamma \vdash u : U$ avec $V\{x \to u\}\ \mathbb{C}^*_\Gamma\ T$. Comme précédemment, il nous faut montrer que $\Gamma \vdash v\{x \to u\} : V\{x \to U\}$. Le résultat ayant été montré lorsque v est un prédicat, il nous suffit de faire la démonstration pour v un objet ; i.e., quand $\Gamma \vdash (x : U)V : \star$.

Par inversion par la règle LAMB de $\Gamma \vdash [x : U']v : (x : U)V$, il existe V' tel que $\Gamma, x : U' \vdash v : V'$ et $(x : U')V'\ \mathbb{C}^*_\Gamma\ (x : U)V$. On a donc une série de conversion de la forme :

$$(x : U)V = Z_0\ \mathbb{C}_\Gamma\ Z_1\ \mathbb{C}_\Gamma\ Z_2\ \mathbb{C}_\Gamma\ \cdots\ \mathbb{C}_\Gamma\ Z_n = (x : U')V'.$$

Par hypothèse, tous les Z_i sont typables, et par le lemme 6, comme $\Gamma \vdash Z_0 : \star$, $\forall i.\Gamma \vdash Z_i : \star$.

On cherche des produits bien typés π_i t.q. $\forall i.\pi_i \mathbb{C}_\Gamma\ \pi_{i+1}$ avec $\pi_0 = (x : U)V$ et $\pi_n = (x : U')V'$. Supposons $Z_0 \to^*_\beta W_0 \sim_\Gamma Y_0 \leftarrow^*_\beta Z_1 \to^*_\beta W_1 \sim_\Gamma Y_1 \leftarrow^*_\beta Z_2$. Comme Z_0 est un produit, il en est de même pour W_0 et par lemme 2.2.10, pour Y_0. Par standardisation, il existe un produit π_1 tel que $Z_1 \to^*_h \pi_1 \to^*_{\not h} Y_0$. Par subject reduction de β^{P_ω}, π_1 est bien typé de type $\star$. De plus, on sait que $Z_1 \to^*_\beta W_1$, et par confluence de β, il existe W_1' tel que $\pi_1 \to^*_\beta W_1'$ et $W_1 \to^*_\beta W_1'$. Par commutation de $\to^*_\beta$ et $\sim_\Gamma$, on déduit qu'il existe Y_1' tel que $Y_1 \to^*_\beta Y_1'$ et $W_1' \sim_\Gamma Y_1'$. Ainsi, $\pi_1\ \mathbb{C}_\Gamma\ Z_2$ et on termine par induction.

Par application itéré du lemme 2.2.14 de compatibilité du produit, on obtient que $U\ \mathbb{C}^*_\Gamma\ U'$ et $V\ \mathbb{C}^*_{\Gamma,\ x:U} V'$. On conclut enfin comme pour le théorème 2.2.13. ∎

Une étude de la normalisation forte de β pour les nouveaux termes du calcul est nécessaire afin de terminer l'étude métathéorique de nouveau calcul. Une méthode serait, à l'instar de [9], de convertir les théories SHOSTAK en des systèmes de récriture de premier ordre puis d'utiliser les résultats de [2] sur l'introduction de règle de récritures au sein de CC. Ou alors, afin de pouvoir étendre par la suite le Calcul des Constructions par des procédures de décision qui ne sont pas transformables en systèmes de récriture respectant les hypothèses de CCA, on peut faire une preuve directe de normalisation. Une première approche serait de reprendre la preuve avec les candidats de réductibilités de GIRARD, en utilisant les mêmes candidats de réductibilité de CC quotientés par $\sim_\emptyset$ (i.e. la conversion sans utiliser les équations de l'environnement de typage).

Enfin, avant de conclure ce chapitre, on peut étudier la question de savoir si le calcul CC'_τ défini avec la relation de conversion $\leftrightarrow^*_{\beta\cup\sim_\Gamma}$ (i.e. avec une définition qui paraît plus naturelle) serait équivalent à CC_τ. Ce problème, qui peut paraître simple de premier abord, soulève en fait la question de savoir si $\sim_\Gamma$ possède une propriété de subject reduction dans le sens que si $\Gamma \vdash t : T$ et $t \sim_\Gamma t'$ alors $\Gamma \vdash t' : T$. Dans le cas de SHOSTAK, il est possible de prouver cette propriété en transformant $\sim_\Gamma$ en un système de récriture de premier ordre [9] puis en utilisant les travaux de [2]. Seulement, ce problème de subject reduction pour $\sim_\Gamma$ risque de devenir un point central de l'étude métathéorique d'un calcul où des procédures de décision autres que SHOSTAK seraient utilisées.

Chapitre 3

Implantation du calcul

En s'appuyant sur les travaux de STEHR et MESEGUER [16], un prototype de $\mathtt{CC}_\tau$ a été implémenté en `Maude`[1]. `Maude`, fortement inspiré par la famille des langages `OBJ`, est un langage déclaratif dans le sens qu'un programme `Maude` est une théorie logique et que l'exécution d'un programme `Maude` est une déduction logique. Pour ce faire, deux logiques sont fournies. L'une par le biais des modules *fonctionnels* et l'autre, qui est un sur-ensemble du premier, par les modules *système*.

Au niveau théorique, les modules fonctionnels sont des théories en logique équationnelle avec appartenance, i.e. une logique de HORN avec sortes, sous-typage entre sortes, opérateurs avec polymorphisme *ad hoc* et équations structurelles (telles que associativité ou commutativité), dont les axiomes sont des égalités (potentiellement conditionnelles) $t = u$ ou de la forme $t : s$ (appartenance, également potentiellement conditionnelles) indiquant que le terme t à la sorte s. Opérationnellement, les équations, en les orientant de gauche à droite, forment un système de récriture, supposé confluent et fortement normalisant.

Les modules systèmes sont des théories en logique par récriture, i.e. la donnée d'une théorie $\mathcal{T}$ en logique équationnelle avec appartenance et de régles R de récriture étiquetées (potentiellement conditionnelles) sur la signature de $\mathcal{T}$. Aucune condition de confluence ou de terminaison n'est requise pour le système R.

La logique par récriture est *réflexive* dans le sens où il existe une théorie *universelle* U qui peut représenter n'importe quelle théorie T et n'importe quels termes t et $'t$ de T en des termes $\overline{T}$, $\overline{t}$ et $\overline{t'}$, tels que :

$$T \vdash t \to t' \Leftrightarrow U \vdash < \overline{T}, \overline{t} > \to < \overline{T}, \overline{t'} >.$$

Et comme il est possible de représenter U dans lui-même, il est alors possible de répéter le processus et d'aboutir ainsi à une *tour* de réflexions :

$$T \vdash t \to t' \Leftrightarrow U \vdash < \overline{U}, \overline{< \overline{T}, \overline{t} >} > \to < \overline{U}, \overline{< \overline{T}, \overline{t'} >} >.$$

C'est ainsi que `Maude` est un langage réflexif et qu'il est donc possible d'étendre le langage dans lui même. Un tel mécanisme est utile, e.g., pour pouvoir récrire au sein des modules systèmes avec une stratégie. La réflexivité a également été utilisée pour incorporer dans `Maude` les modules paramétrés et la programmation objet.

1. `http://maude.cs.uiuc.edu/`

Conclusion

Ce rapport ne présente qu'une première approximation quant à l'introduction de procédures de décisions au sein du Calcul des Constructions. Outre la fin de la preuve de cohérence logique, des extensions peuvent être apportées :

Meilleure extraction des équations Actuellement, l'extraction des équations se fait exclusivement syntaxiquement. On peut bien sur imaginer des extractions plus poussées par des tactiques sur l'environnement de typage. E.g., dans un environnement $\Gamma = \Gamma', p_1 : A, p_2 : A \to B$, si B est une équation du premier ordre, on souhaiterait qu'elle soit utilisée ;

Autres procédures de décision Il serait agréable pour l'utilisateur de pouvoir utiliser différentes procédures de décision voire des procédures de semi-décision d'ordre supérieur ;

Intégration dans CCA Le calcul étendu dans ce rapport est CCA avec un système de récriture vide. La version finale de ce calcul devrait bien sur être CCA dans toute sa généralité étendu par des procédures de décision.

Bibliographie

[1] H. Barendregt. Lambda calculi with types. Dans *S. Abramski, D. Gabbay and T. Maibaum, éditeurs, Handbook of logic in omputer science*, volume 2, pages 411–414. Oxford University Press, 1992.

[2] F. Blanqui. *Théorie des types et récriture*. Thése de Doctorat, Université Paris XI - Orsay, 2001.

[3] F. Blanqui. Definitions by rewriting in the Calculus of Constructions. ?(?):?–?, 2003. To appear.

[4] Sylvain Conchon et Sava Krstic. Strategies for combining decision procedures. Dans *Proceedings of the 9th Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619, pages 537–553. Lecture Notes in Computer Science, 2003.

[5] Th. Coquand. *Une théorie des constructions*. Thése de Doctorat, Université Paris VII, 1985.

[6] Th. Coquand et G. Huet. The calculus of constructions. *Informations and Computation*, 76(2), 1988.

[7] Th. Coquand et C. Pauling-Mohring. Inductively defined types. Dans *P. Martin-Löf and G. Mints, éditeurs*, volume 417. Springer-Verlag, Lecture Notes in Computer Science, 1990.

[8] H. Ganzinger. Shostak light. Dans *A. Voronkov, éditeur, Automated Deduction*, volume 2392, pages 332–347. Lecture Notes in Artificial Intelligence, 2002.

[9] Deepak Kapur. Shostak's congruence closure as completion. Dans H. Comon, éditeur, *Proceedings of the 8th International Conference on Rewriting Techniques and Applications*, volume 1232. Springer-Verlag, 1997.

[10] G. Nelson et D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[11] G. Nelson et D.C. Oppen. Fast decision procedure based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, 1980.

[12] D.C. Oppen. Reasoning about recursively defined data structures. *Journal of the Association for Computing Machinery*, 27(3):403–411, 1980.

[13] H. Ruess et N. Shankar. Deconstructing shostak. Dans *Proceedings of the sixteenth IEEE Symposium On Logic In Computer Sciences (LICS'01)*, pages 19–28. IEEE Computer Society Press, 2001.

[14] N. Shankar et Harald Ruess. Combining shostak theories. Invited paper for Floc'02/RTA'02, 2002.

[15] R.E. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[16] M. Stehr et J. Meseguer. Pure type systems in rewriting logic, 1999.

[17] Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 7.4*. INRIA Rocquencourt, France, 2003. `http://coq.inria.fr/`.

Annexe A

Exemples de solveurs et canoniseurs

A.1 La théorie des listes

Théorie $\mathcal{T}_\mathcal{L}$ basée sur la signature $\{\mathrm{car}_{/1}\,,\ \mathrm{cdr}_{/1}\,,\ \mathrm{cons}_{/2}\}$ et les axiomes :

$\mathrm{cons}(\mathrm{car}(x),\ \mathrm{cdr}(x)) = x$	Axiome de construction
$\mathrm{car}(\mathrm{cons}(x, y)) = x$ $\mathrm{cdr}(\mathrm{cons}(x, y)) = y$	Axiomes de sélection
$\mathrm{car}(x) \neq x$ $\mathrm{cdr}(x) \neq x$ $\mathrm{car}(\mathrm{car}(x)) \neq x$ $\vdots$	Axiomes d'acyclicité

- Le canoniseur $\sigma_{\mathcal{T}_\mathcal{L}}$ est donné par le système de récriture[1] obtenu en orientant les axiomes de sélection et de construction de la gauche vers la droite.
- Pour résoudre $s \approx t$, il suffit de réduire la configuration $[\{s \approx t\}\ ;\ \emptyset]$ par le système d'inférence donné en figure A.1. Soit on aboutit sur une configuration de la forme $[\emptyset\ ;\ S]$, avec S en forme résolue. (Le système d'inférence réduit des configurations de la forme $[E\ ;\ S]$ avec E et S des ensembles d'équations et S toujours en forme résolue) Dans ce cas, $\mathrm{solve}(s \approx t) = S$. Soit $[s \approx t\ ;\ \emptyset]$ se réduit sur $\perp$ et S est insatisfiable ($\mathrm{solve}(s \approx t) = \perp$).

A.2 L'arithmétique linéaire

- Le canoniseur renvoie l'expression arithmétique sous la forme d'une somme ordonnée de monômes. E.g. $\sigma(2y + x + 5 + 2x + z - z) = 3x + 2y + 5$;
- Le solveur est basé sur l'algorithme d'Euclide.
  E.g. $\mathrm{solve}(3x + 5y = 1) = \{x = -3 + 5k, y = 2 - 3k\}$.

1. Confluent et fortement normalisant.

$$\frac{\{\mathrm{cons}(x,y)=z\}\uplus E\ ;\ S}{\{x=\mathrm{car}(z), y=\mathrm{cdr}(z)\}\cup E\ ;\ S}\quad \text{(DÉCOMPOSITION)}$$

$$\frac{\{\mathrm{car}(x)=y\}\uplus E\ ;\ S}{\{x=\mathrm{cons}(y,k)\}\cup E\ ;\ S}\quad \begin{array}{l}\text{(SOLVE}_1\text{)}\\ k \text{ fraîche}\end{array} \qquad \frac{\{\mathrm{cdr}(x)=y\}\uplus E\ ;\ S}{\{x=\mathrm{cons}(k,y)\}\cup E\ ;\ S}\quad \begin{array}{l}\text{(SOLVE}_2\text{)}\\ k \text{ fraîche}\end{array}$$

$$\frac{\{x=a\}\uplus E\ ;\ S}{\sigma_{\mathcal{T}_\mathcal{L}}(E[x:=a])\ ;\ S\circ\{x=a\}}\quad \begin{array}{l}\text{(ÉLIMINATION)}\\ x\notin \mathrm{vars}(a)\end{array}$$

$$\frac{\{a=a\}\uplus E\ ;\ S}{E\ ;\ S}\quad \text{(TRIVIAL)} \qquad \frac{\{a=b\}\uplus E\ ;\ S}{\bot}\quad \begin{array}{l}\text{(CONTRADICTION)}\\ b \text{ est un sous-terme strict de}\\ a \text{ (ou inversement)}\end{array}$$

où $S\circ S' = S'\cup\{x=\sigma_{\mathcal{T}_\mathcal{L}}(aS') \mid x=a\in S\}$

FIG. A.1 – *Résolution dans la théorie des listes*

Annexe B

L'algorithme de Nelson et Oppen

On présente ici l'algorithme de NELSON et OPPEN, procédure de décision pour les théories équationnelles pures, tel que décrit originellement dans [11][1].

Soit $G = (V, E)$ un graphe orienté acyclique dont les noeuds sont étiquetés (avec un ensemble fini d'étiquettes ξ) et avec potentiellement des arcs multiples entre deux sommets. Pour chaque sommet s, on note $\lambda(s)$ l'étiquette de s et $\mu(s)$ le nombre d'arcs sortant de s. De plus, on considère que les arcs sortant sont ordonnés. Ainsi, pour tout $i \in \{1 \cdots \mu(s)\}$, $s[i]$ désigne le $i^{\text{ème}}$ arc sortant de s.

Définition B.0.1 (Congruence) Soit R une relation binaire sur V. Deux sommets s et t sont **congruents sous** R si et seulement si $\lambda(s) = \lambda(t)$, $\mu(s) = \mu(t)$ et $\forall i \in \{1 \cdots \lambda(s)\}.(s[i], t[i]) \in R$. R est **congruente** si et seulement si pour tous sommets s et t tels que t et s sont congruents sous R, alors $(s, t) \in R$.

Proposition-Définition B.0.2 *Soit R une relation binaire sur V. Il existe toujours une plus petite relation $\hat{R}$ contenant R^* et congruente. $\hat{R}$ est la* ***fermeture par congruence*** *de R.*

On se donne maintenant une signature Σ et on considère la théorie équationnelle pure contruite sur Σ. On souhaite décider de la satisfiablité d'une conjonction de littéraux de la forme $t_1 \approx u_1 \wedge \cdots \wedge t_n \approx u_n \wedge r_1 \not\approx s_1 \wedge \cdots \wedge r_p \not\approx s_p$ (notée ϕ par la suite). L'algorithme comprend deux étapes : la représentation des termes par un graphe puis la fermeture par congruence de ce dernier.

Définition B.0.3 Soit S un ensemble de Σ-termes fermés par la relation sous-terme. S est représenté par le plus petit graphe acyclique tel que :

- chaque variable x est représentée par une feuille étiquetée par x ;
- chaque terme $f(t_1, \cdots, t_n)$ est représenté par un sommet s tel que $\lambda(s) = f$, $\mu(s) = \text{arite}(f)$ et les noeuds $v[1] \cdots v[n]$ représentent respectivement les termes $t_1 \cdots t_n$.

avec la restriction supplémentaire de partage maximal ; i.e., pour tous termes s et t de S, si s est un sous-terme de t alors le graphe représentant s doit être un sous-graphe de celui représentant t.

Si G est un graphe représentant S et a un terme de S alors on note $\nu(a)$ la racine du sous-graphe de G représentant a.

L'algorithme se déroule ainsi :

1. Soit $S = \{t_1, u_1, \cdots, t_n, u_n, r_1, s_1, \cdots, r_p, s_p\}$. On note G le graphe représentant la fermeture par sous terme de S et R la relation identité sur G ;

1. On peut aujourd'hui trouver d'autres versions de Congruence Closure telle que la Congruence Closure abstraite.

2. Pour chaque équation $t_i \approx u_i$, $R \leftarrow \text{MERGE}(R, (\nu(t_i), \nu(v_i)))$; où MERGE, définie ci-dessous, calcule la fermeture par congruence de $R \cup \{(\nu(t_i), \nu(u_i))\}$;
3. ϕ est alors satisfiable si et seulement si pour chaque inéquation $r_i \not\approx s_i$, $(\nu(r_i), \nu(s_i)) \notin R$.

Reste à définir MERGE. MERGE manipulant des relations qui sont en fait des relations d'équivalence, les primitives UNION et FIND de TARJAN sont fortement utilisées.

Définition de MERGE

procedure MERGE(G, R, s, t)
  $G = (E, V)$ un graphe telle que définie en début de section
  R une relation d'équivalence sur V congruente
  s et t sont deux sommets de G
output La fermeture par congruence de $R \cup \{(s, t)\}$
if FIND(s) = FIND(t) **then**
  $P_s \leftarrow$ {les prédecesseurs de tous les noeuds équivalence, pour R, à s};
  $P_t \leftarrow$ {les prédecesseurs de tous les noeuds équivalence, pour R, à t};
  $R \leftarrow$ UNION(R, s, t);
  for all $(x, y) \in P_s \times P_t$ **do**
    if FIND$(R, s) \neq$ FIND(R, t) et s et t sont congruents sous R **then**
      $R \leftarrow$ MERGE(G, R, x, y)
    end if
  end for
end if
return R

Annexe C

Bribes du prototype

Le source est séparé en quatre parties principales.

C.1 Langage de stratégies

La premiere définit un langage de stratégie utilisé par la suite pour l'algorithme de SHOSTAK

```
fmod OBJECT-LEVEL-STRATEGY is
  protecting QID .

  sorts Strategy StratInOut .

  op  [_]   : Qid -> Strategy [ctor] .
  op  _||_  : Strategy Strategy -> Strategy [gather (E e) prec 90 ctor] .
  op  _&&_  : Strategy Strategy -> Strategy [gather (E e) prec 85 ctor] .
  op  _&?_  : Strategy Strategy -> Strategy [gather (E e) prec 85 ctor] .
  op  _*    : Strategy -> Strategy          [prec 80 ctor] .
  op  _+    : Strategy -> Strategy          [prec 80 ctor] .
  op  _?    : Strategy -> Strategy          [prec 80 ctor] .

  op  failure          : -> [StratInOut] [ctor] .
  op  reduceWithStrat  : Strategy [StratInOut] -> [StratInOut] .
```

La fonction `reduceWithStrat` s'occupe alors de réduire un terme de sorte `StratInOut` suivant une stratégie donnée. Pour exemple, on peut donner la définition de la réduction pour ||.

```
  vars ?R      : [StratInOut] .
  vars S S1 S2 : Strategy .
  vars I       : StratInOut .

  eq  reduceWithStrat(S, failure) = failure .

  ceq reduceWithStrat(S1 || S2, I) = ?R
        if ?R :=  reduceWithStrat(S1, I)
        /\ ?R =/= failure .
```

```
  eq  reduceWithStrat(S1 || S2, I) = reduceWithStrat(S2, I) [owise] .

endfm
```

C.2 Théorie Shostak

La seconde définit ce qu'est une théorie SHOSTAK et met en place l'algorithme. Tout d'abord, il nous faut définir la syntaxe des termes de 1er ordre,...

```
fmod SHOSTAK-SYN is
  protecting NAT .
  protecting QID .
  protecting ORACLE .
  protecting COMMON-SETS .

  vars X Y Z   : Qid .
  vars m n p q : Nat .

  *************
  *** Terms ***
  *************
  sorts Trm Var SFct UFct Fct . *** SFct : interpreted
  sorts TrmList               . *** UFct : uninterpreted

  subsorts Var < Trm .
  subsorts SFct UFct  < Fct .

  subsorts Trm < TrmList .

  op lmt    : -> TrmList [ctor] .
  op _;_    : TrmList TrmList   -> TrmList
              [ctor assoc gather (e E) prec 80 id: lmt] .

  op {_}  : Qid -> Var .
  op _(_) : Fct TrmList -> Trm .
```

...des littéraux,...

```
  sorts    TrmOrientEq TrmEq TrmIEq TrmLit .
  subsorts TrmEq TrmIEq < TrmLit .

  vars TL : TrmLit .

  op  _===_ : Trm Trm -> TrmEq [ctor comm] .
  op  _=/=_ : Trm Trm -> TrmIEq [ctor comm] .
  op  _-->_ : Trm Trm -> TrmOrientEq [ctor] .

  sorts    LitSet OrientEqSet .
  subsorts TrmLit < LitSet .
  subsorts TrmOrientEq < OrientEqSet .
```

```
  op  mt : -> LitSet [ctor] .
  op  __ : LitSet LitSet -> LitSet [ctor comm assoc id: mt] .

  op  mt : -> OrientEqSet [ctor] .
  op  __ : OrientEqSet OrientEqSet -> OrientEqSet [ctor comm assoc id: mt] .
```

...et des DAG (Directed Acyclic Graph),...

```
  sorts    Rew RewSet .
  subsorts Rew < RewSet .

  op  _=>_ : Var Trm -> Rew [ctor] .
  op  mt   : -> RewSet [ctor] .
  op  __   : RewSet RewSet -> RewSet [ctor comm assoc id: mt] .
```

On peut alors, e.g., écrire très simplement l'application d'un DAG, vu comme un système de récriture, à un terme.

```
  op  _[_] : Trm RewSet -> Trm .
  op  _[_] : TrmList RewSet -> TrmList .

  vars F   : Fct .
  vars SR  : RewSet .
  vars X Y : Qid .
  vars TS  : TrmList .

  eq  F(TS)[SR]          = F(TS[SR]) .
  eq  {X}[mt]            = {X} .
  eq  {X}[({X} => T) SR] = T .
  ceq {X}[({Y} => T) SR] = {X}[SR] if X =/= Y .

  eq  lmt[SR] = lmt .
  eq  (T ; TS)[SR] = (T[SR]) ; (TS[SR]) .
endfm
```

Reste à définir ce qu'est qu'une théorie SHOSTAK : une théorie de premier ordre avec deux opérateurs `canon` et `solve`.

```
fmod SHOSTAK-TH is
  protecting SHOSTAK-SYN .

  op  arity  : Fct -> Nat .
  op  canon  : Trm -> Trm .
  op  solve  : TrmOrientEq -> [RewSet] .
  op  bottom : -> RewSet .

  vars  F             : Fct .
  vars  n             : Nat .

  op  [_]             : Fct -> UFct [ctor] .
  op  {_}             : Nat  -> UFct .
  ops pi lam app      : -> UFct [ctor] .
```

```
  eq  arity(pi)     = 2 .
  eq  arity(lam)    = 2 .
  eq  arity(app)    = 2 .
  eq  arity([F])    = 0 .
  eq  arity({n})    = 0 .
```

Enfin, avant de pouvoir écrire l'algorithme de SHOSTAK, il nous faut définir ce qu'est une configuration...

```
fmod SHOSTAK is
  sorts Defun DefunSet .
  subsorts Defun < DefunSet .

  vars DSS  DSS' : DefunSet .
  vars DSS1 DSS2 : DefunSet .

  op  <_[_]=_> : UFct TrmList Trm -> Defun [ctor] .
  op  mt       : -> DefunSet [ctor] .
  op  __       : DefunSet DefunSet -> DefunSet [ctor comm assoc id: mt] .

  sorts Config .
  vars  C C' : Config .

  op bottomCo  : -> Config .
  op  [_;_;_] : LitSet RewSet DefunSet -> Config [ctor] .
```

...et décrire comment étendre quelques opérations, comme la canonisation, sur des listes.

```
  vars X Y Z   : Qid .
  vars m n p q : Nat .

  vars F F1 F2        : Fct .
  vars uF             : UFct .
  vars sF             : SFct .
  vars S T S' T'      : Trm .
  vars TS TS' TS1 TS2 : TrmList .

  vars TL         : TrmLit .
  vars EQ         : TrmEq .
  vars IEQ        : TrmIEq .
  vars LIT LIT'   : TrmLit .
  vars STL STL'   : LitSet .
  vars SR SR' SR1 : RewSet .


  op  reduce : LitSet RewSet -> LitSet .
  op  reduce : DefunSet RewSet -> DefunSet .

  op  _@_     : RewSet RewSet -> RewSet .
  op  canonSR : RewSet -> RewSet .
  op  canonL  : TrmList -> TrmList .
```

On peut alors décrire les différentes règles de l'algorithme.

```
op  shostakReduce : Config        -> [StratInOut] .
op  shostakSolve  : Config        -> [StratInOut] .
op  shostakElim   : Config        -> [StratInOut] .
op  shostakComp   : Config        -> [StratInOut] .
op  shostakContr  : Config        -> [StratInOut] .

op  shostakSolveN : Config RewSet -> Config .

subsorts Config < StratInOut .

ceq shostakReduce ([ STL ; SR ; DSS ]) = [ STL' ; SR ; DSS' ]
      if STL' := reduce(STL, SR)
      /\ DSS' := reduce(DSS, SR) .

ceq shostakSolve ([ ( S === T ) STL ; SR ; DSS ]) =
    if   SR' =/= bottom
    then shostakSolveN([ STL ; SR ; DSS ], SR')
    else bottomCo
    fi if SR' :=  solve(S --> T) .

eq  shostakElim([ ( S === S ) STL ; SR ; DSS ]) =
    [ STL ; SR ; DSS ] .

eq  shostakComp ([ STL ; SR ; (< F [ TS ] = T >)
                               (< F [ TS ] = T' >) DSS]) =
    [ (T === T') STL ; SR ; (< F [ TS ] = T >) DSS ] .

ceq shostakContr ([ (S =/= T) STL ; SR ; DSS ]) = bottomCo
      if S == T .

ceq shostakSolveN ([ STL ; SR ; DSS ], SR') =
    [ STL ; (SR @ SR1) SR1 ; DSS ]
      if SR1 := canonSR(SR') .
```

Pour finir, reste à définir la stratégie d'exécution de l'algorithme...

```
op  shostakStrategy : -> Strategy [ctor] .
op  shostak : LitSet TrmEq -> Bool .

eq  shostakStrategy = ['reduce] &?
      (['elim-triv]+ || ['contr] || (['solve] &? ['reduce]) || ['comp])* .

ceq shostak(STL, S === T) = C' == bottomCo
      if C  := termLitToConfig(STL (S =/= T))
      /\ C' := reduceWithStrat(shostakStrategy, C) .
```

... et le processus de conversion d'un ensemble de littéraux vers une configuration.

```
sorts AbsResult AbsResultSet AbsResultLitSet AbsResultLit .

op  <_;_>    : TrmList DefunSet -> AbsResultSet .
```

```
  op  <_;_>    : LitSet  DefunSet -> AbsResultLitSet .
  op  {_;_}    : Trm     DefunSet -> AbsResult .
  op  {_;_}    : TrmLit  DefunSet -> AbsResultLit .

  op  abstractS : TrmList -> AbsResultSet .
  op  abstract  : Trm     -> AbsResult .
  op  abstractS : LitSet  -> AbsResultLitSet .
  op  abstract  : TrmLit  -> AbsResultLit .

  eq  abstractS((lmt).TrmList) = < lmt ; mt > .
  ceq abstractS((S ; TS)) = < (T ; TS') ; DSS1 DSS2 >
        if { T   ; DSS1 } := abstract(S)
        /\ < TS' ; DSS2 > := abstractS(TS) .

  eq  abstract({X}) = { {X} ; mt } .

  ceq abstract(sF(TS)) = { canon(sF(TS')) ; DSS }
        if < TS' ; DSS > := abstractS(TS) .

  ceq abstract(uF(TS)) = { {X} ; DSS < uF [ canonL(TS') ] = {X} > }
        if X := NEW
        /\ < TS' ; DSS > := abstractS(TS) .

  eq  abstractS((mt).LitSet) = < mt ; mt > .
  ceq abstractS(LIT STL) = < LIT' STL' ; DSS1 DSS2 >
        if { LIT' ; DSS1 } := abstract(LIT)
        /\ < STL' ; DSS2 > := abstractS(STL) .

  ceq abstract((S === T)) = { (canon(S') === canon(T')) ; DSS1 DSS2 }
        if { S' ; DSS1 } := abstract(S)
        /\ { T' ; DSS2 } := abstract(T) .

  ceq abstract((S =/= T)) = { (canon(S') =/= canon(T')) ; DSS1 DSS2 }
        if { S' ; DSS1 } := abstract(S)
        /\ { T' ; DSS2 } := abstract(T) .

  op  termLitToConfig : LitSet -> Config .

  ceq termLitToConfig(STL) = [ STL' ; mt ; DSS ]
        if < STL' ; DSS > := abstractS(STL) .
endfm
```

C.3 Le Calcul des Constructions

La troisième partie s'occupe de décrire un algorithme de vérification de type pour $\texttt{CC}_\tau$. Comme pour le premier ordre, il nous faut tout d'abord introduire la syntaxe de $\texttt{CC}_\tau$ dans le formalisme `CINNI` développé dans [16].

```
fmod ROUPTS-CC+-SYN is
  protecting QID .
  protecting FPTS-SPEC-CC+ .
  protecting SHOSTAK .
```

```
  *** Terms
  sorts   CVar CTrm .
  subsort Fct CVar Sorts < CTrm .

  op  _{_}   : Qid Nat -> CVar [ prec 0 gather (E &) ctor ] .
  op  __     : CTrm CTrm -> CTrm [ prec 21 gather (E e) ctor ] .
  op  [_:_]_ : Qid CTrm CTrm -> CTrm [ prec 30 gather (& & E) ctor ] .
  op  {_:_}_ : Qid CTrm CTrm -> CTrm [ prec 30 gather (& & E) ctor ] .

  sorts CTrmList .
  subsorts CTrm < CTrmList .

  op  cmt    : -> CTrmList [ctor] .
  op  _;_    : CTrmList CTrmList -> CTrmList [ctor assoc id: cmt] .
  op  length : CTrmList -> Nat .

  eq  length(nil) = 0 .
  eq  length(T ; CTL) = (s length(CTL)) .

  op  type  : Fct -> CTrm .
  op  sorts : Fct -> Sorts .
endfm
```

On définit ensuite toutes les opérations basiques sur les termes de $\mathtt{CC}_\tau$, comme les opérateurs de substitution explicite ou la β-réduction. (Le code, long et peu intéressant, n'est pas détaillé ici)

```
mod ROUPTS-CC+ is
  extending ROUPTS-CC+-TH .

  vars s s' s1 s2 s3 : Sorts .
  vars X Y Z : Qid .
  vars A B M N O P Q R T A' B' M' N' T' T1 T2 T3 U U' : CTrm .
  vars F F' : Fct .
  vars Ct Ct' : CCt .
  vars S   : Subst .
  vars n m : Nat .

  sorts Subst .

  op [_:=_]   : Qid CTrm -> Subst [ prec 30 gather (& &) ] .
  op [shift_] : Qid -> Subst [ prec 30 gather (&) ] .
  op [lift__] : Qid Subst -> Subst [ prec 30 gather (& &) ] .
  op __       : Subst CTrm -> CTrm [ prec 30 gather (E E) ] .

  op  betaNormalForm     : CTrm -> CTrm .
  op  betaNormalFormEq   : CTrm CTrm -> Bool .
```

La phase suivante est de créer un opérateur `shostakify` qui effectue l'opération d'algébraïsation d'un terme de $\mathtt{CC}_\tau$.

```
  vars QL QL' : QidList .
```

```
vars CTL CTL' : CTrmList .

vars 1T 1T'             : Trm .
vars 1TL 1TL' 1TL1 1TL2 : TrmList .

op  [_]               : Sorts -> UFct [ctor] .
op  shostakify        : CTrm -> Trm .
op  deBruijn          : CVar QidList Nat -> Trm .
op  shostakify        : CTrm QidList -> [Trm] .
op  shostakifyList    : CTrmList QidList -> [TrmList] .
op  shostakifyProduct : CTrm QidList TrmList -> [Trm] .
op  termListToApp     : TrmList -> [Trm] .

eq  arity([s]) = 0 .

eq  deBruijn(X{n},   qmt,      m) = {X} .
eq  deBruijn(X{0},   (X ; QL), m) = {m}(lmt) .
eq  deBruijn(X{s n}, (X ; QL), m) = deBruijn(X{n}, QL, (s m)) .
ceq deBruijn(X{n},   (Y ; QL), m) = deBruijn(X{n}, QL, (s m))
      if X =/= Y .

eq  shostakify(T) = shostakify(T, qmt) .

eq  shostakify(X{n}, QL) = deBruijn(X{n}, QL, 0) .
eq  shostakify({X : A} M, QL) =
      pi(shostakify(A, QL) ; shostakify(M, (X ; QL))) .
eq  shostakify([X : A] M, QL) =
      lam(shostakify(A, QL) ; shostakify(M, (X ; QL))) .
eq  shostakify(s, QL) = [s](lmt) .
eq  shostakify(Ct, QL) = [Ct](lmt) .
eq  shostakify(T, QL) = shostakifyProduct(T, QL, lmt) [owise] .

eq  shostakifyList(cmt, QL) = lmt .
ceq shostakifyList((T ; CTL), QL) = 1T ; 1TL
      if 1T  := shostakify(T, QL)
      /\ 1TL := shostakifyList(CTL, QL) .

ceq shostakifyProduct(M N, QL, 1TL) =
      shostakifyProduct(M, QL, (1T ; 1TL))
      if 1T := shostakify(N, QL) .

ceq shostakifyProduct(F, QL, 1TL) = termListToApp([F](lmt) ; 1TL)
      if length(1TL) < arity(F) .
ceq shostakifyProduct(F, QL, 1TL) = termListToApp((F(1TL1)) ; 1TL2)
      if length(1TL) >= arity(F)
      /\ [1TL1 ; 1TL2] := splitAtPos(1TL, arity(F)) .

eq  shostakifyProduct(T, QL, 1TL) =
      termListToApp(shostakify(T) ; 1TL) [owise] .

eq  termListToApp(1T) = 1T .
ceq termListToApp(1TL ; 1T) = app(termListToApp(1TL) ; 1T)
      if 1TL =/= lmt .
```

Pour finir, reste à décrire l'algorithme de vérification de type pour $\texttt{CC}_\tau$. Ce dernier fonctionne comme un solveur de contraintes sur un ensemble de jugements dont le fonctionnement est décrit dans [16]. Succinctement, les différentes contraintes sont :

$u = t$	u est syntaxiquement identique à t
$u \leftrightarrow t$	$u \mathbb{C} t$
$u \leftrightarrow^b t$	$u \sim t$
$t\texttt{Sort}$	t est une sorte
$(t_1, t_2, s)\texttt{Rule}$	t_1 et t_2 sont des sortes et une règle de formation pour le produit avec les sortes t_1, t_2 et s existe
$\Gamma \vdash t \rightarrow: ?T$	$?T$ est instanciable en un type T tel que $\Gamma \vdash t : T$
$\Gamma \vdash ((M \rightarrow: ?A)(N \rightarrow: ?B)) \rightarrow: C$	$\Gamma \vdash M \rightarrow: ?A$, $\Gamma \vdash N \rightarrow: ?B$ et $\Gamma \vdash (MN) \rightarrow: C$.

```
*** Contexts
sorts Context .

op  emptyContext : -> Context .
op  _:_ : Qid CTrm -> Context .
op  _,_ : Context Context -> Context [assoc id: emptyContext ] .

vars G G1 G2 : Context .

sorts Judgement .

op  _||-    : Context -> Judgement [ gather (&) ] .
op  _|-_:_  : Context CTrm CTrm -> Judgement [ gather (& & &) ] .
op  _||-_:_ : Context CTrm CTrm -> Judgement [ gather (& & &) ] .

op _=_           : CTrm CTrm -> Judgement .
op _<->_         : CTrm LitSet CTrm -> Judgement .
op _<-b->_       : CTrm LitSet CTrm -> Judgement .
op _Sort         : CTrm -> Judgement .
op '(_,_,_')Rule : CTrm CTrm CTrm -> Judgement .

op _|-_:_   : Context CTrm CTrm -> Judgement [ gather (& & &) ] .
op _|-_->:_ : Context CTrm CTrm -> Judgement [ gather (& & &) ] .
op _|-'(_->:_')'(_->:_')->:_
            : Context CTrm CTrm CTrm CTrm -> Judgement .

vars J  : Judgement .
vars LS : LitSet .

sorts    JudgementList .
subsorts Judgement < JudgementList .

op ejl : -> JudgementList .
op __  : JudgementList JudgementList -> JudgementList [assoc id: ejl] .

vars JS : JudgementList .

sorts Configuration .
```

```
op  {{_}} : JudgementList -> Configuration [gather (&)] .
```

Pour pouvoir manipuler des termes non instanciés, il nous faut également définir des métavariables ainsi que des opérateurs pour les instancier.

```
sorts    MetaCVar .
subsorts MetaCVar < CTrm .

op  ? : Qid -> MetaCVar .

vars MV MV' : MetaCVar .

op  <_:=_>_ : MetaCVar CTrm CTrm -> CTrm [prec 59 gather (& & &)] .
op  <_:=_>_ : MetaCVar CTrm Subst -> Subst [prec 59 gather (& & &)] .
op  <_:=_>_ : MetaCVar CTrm Context -> Context [prec 59 gather (& & &)] .
op  <_:=_>_ : MetaCVar CTrm Judgement -> Judgement
              [prec 59 gather (& & &)] .
op  <_:=_>_ : MetaCVar CTrm JudgementList -> JudgementList
              [prec 59 gather (& & &)] .

op  lookup : Context CVar -> [CTrm] .
```

Les règles suivantes s'occupent de la vérification de type.

```
op  termsBetaNormAreEq : CTrm CTrm -> Bool .

eq  termsBetaNormAreEq(M, N) =
      if   betaNormalFormEq(M, N)
      then true
      else shostak(mt, shostakify(M) === shostakify(N))
      fi .

rl  [Subst] :  {{ (MV = A) JS }} => {{ < MV := A > JS }} .

crl  [Conv] :  {{ (M <-> N) JS }} => {{ (M' <-b-> N') JS }}
               if M' := betaNormalForm(M)
               /\ N' := betaNormalForm(N) .

crl  [Conv] :  {{ (M <-b-> N) JS }} => {{ JS }}
                  if termsBetaNormAreEq(M, N) .

rl  [Sort] :   {{ (s Sort) JS }} => {{ JS }} .

rl  [Rule] :   {{ ((s1, s2, MV) Rule) JS }} =>
               {{ (MV = Rules(s1, s2)) JS }} .

rl  [Ax] :     {{ (G |- s ->: MV) JS }} =>
               {{ (MV = Axioms(s)) JS }} .

crl [Lookup] : {{ (G |- X{m} ->: MV) JS }} =>
               {{ (MV = lookup(G, X{m})) JS }}
               if lookup(G, X{m}) : CTrm .
```

```
  rl  [Fct] :    {{ (G |- F ->: MV) JS }} =>
                 {{ (MV = type(F)) (G |- type(F) : sorts(F)) JS }} .

  rl  [Cct] :    {{ (G |- Ct ->: MV) JS }} =>
                 {{ (MV = Prop) JS }} .

  rl  [Pi] :     {{ (G |- {X : A} B ->: MV) JS }} =>
                 {{ (G |- A ->: ?(NEW1)) (?(NEW1) Sort)
                    (G, (X : A) |- B ->: ?(NEW2))
                  ((?(NEW1), ?(NEW2), MV) Rule) JS }} .

  rl  [Lda] :    {{ (G |- [X : A] M ->: MV) JS }} =>
                 {{ (G |- A ->: ?(NEW1)) (?(NEW1) Sort)
                    (G, (X : A) |- M ->: ?(NEW2))
                    (MV = {X : A} ?(NEW2)) JS }} .

  rl  [App1] :   {{ (G |- (M N) ->: MV) JS }} =>
                 {{ (G |- M ->: ?(NEW1)) (G |- N ->: ?(NEW2))
                    (G |- (M ->: ?(NEW1))(N ->: ?(NEW2)) ->: MV) JS }} .

  rl  [App2] :   {{ (G |- (M ->: {X : A} B)(N ->: A') ->: MV) JS }} =>
                 {{ (A <-> A') (MV = [X := N] B) JS }} .

  crl [Norm1] :  {{ (T Sort) JS }} => {{ (T' Sort) JS }}
                 if T' := betaNormalForm(T) /\ T =/= T' .

  crl [Norm2] :  {{ ((A, B, T) Rule) JS }} => {{ ((A', B, T) Rule) JS }}
                 if A' := betaNormalForm(A) /\ A =/= A' .

  crl [Norm3] :  {{ ((A, B, T) Rule) JS }} => {{ ((A, B', T) Rule) JS }}
                 if B' := betaNormalForm(B) /\ B =/= B' .

  crl [Norm4] :  {{ (G |- (M ->: A)(N ->: B) ->: T) JS }} =>
                 {{ (G |- (M ->: A')(N ->: B) ->: T) JS }}
                 if A' := betaNormalForm(A) /\ A =/= A' .

  rl  [Aux] :    {{ (G |- M : A) JS }} =>
                 {{ (G |- A ->: ?(NEW1)) (?(NEW1) Sort)
                    (G |- M ->: ?(NEW2)) (?(NEW2) <-> A) JS }} .

  rl  [Ctxt1] :  {{ (emptyContext ||-) JS }} => {{ JS }} .

  rl  [Ctxt2] :  {{ (G, (X : A) ||-) JS }} =>
                 {{ (G ||-) (G |- A ->: ?(NEW)) (?(NEW) Sort) JS }} .

  rl  [Main] :   {{ (G ||- M : A) JS }} =>
                 {{ (G ||-) (G |- M : A) JS }} .
endm
```

C.4 La théorie des listes

Pour finir, on définit le canoniseur et solveur pour la théorie des listes.

```
fmod ROUPTS-CC+-LIST is
  extending  ROUPTS-CC+ .
  extending  OBJECT-LEVEL-STRATEGY .
  protecting ORACLE .

  vars  F              : Fct .
  vars  n              : Nat .
  vars  X Y Z          : Qid .
  vars  H T L L' L1 L2 : Trm .
  vars  STO            : OrientEqSet .
  vars  RS RS'         : RewSet .

  ops cons nil car cdr : -> SFct [ctor] .
  ops f g a            : -> UFct [ctor] .

  eq  arity(nil)  = 0 .
  eq  arity(cons) = 2 .
  eq  arity(car)  = 1 .
  eq  arity(cdr)  = 1 .
  eq  arity(f)    = 1 .
  eq  arity(g)    = 2 .
  eq  arity(a)    = 0 .

  eq  sorts(nil) = Prop .
  eq  sorts(car) = Prop .
  eq  sorts(cdr) = Prop .
  eq  sorts(cons) = Prop .
  eq  sorts(f) = Prop .
  eq  sorts(g) = Prop .
  eq  sorts(a) = Prop .

  sorts CCt .
  subsorts CCt < CTrm .

  ops elt list : -> CCt .

  eq  type(nil) = list .
  eq  type(cons) = {'_ : elt} {'_ : list} list .
  eq  type(car) = {'_ : list} elt .
  eq  type(cdr) = {'_ : list} list .
  eq  type(f) = {'_ : elt} elt .
  eq  type(g) = {'_ : elt} {'_ : elt} elt .
  eq  type(a) = elt .
```

Le canoniseur se décrit simplement par quelques équations.

```
  op  canonTT : Trm -> Trm .
```

```
eq  canon(cons(H ; T)) = canonTT(cons(canon(H) ; canon(T))) .
eq  canon(car(L))      = canonTT(car(canon(L))) .
eq  canon(cdr(L))      = canonTT(cdr(canon(L))) .
eq  canon({X})         = {X} .

eq  canonTT(cons(car(L) ; cdr(L))) = L .
eq  canonTT(car(cons(H ; T)))      = H .
eq  canonTT(cdr(cons(H ; T)))      = T .
eq  canonTT(T)                     = T [owise] .
```

Pour le solveur, il nous faut utiliser une stratégie comme dans l'algorithme de SHOSTAK.

```
sorts SolveIO .

op   [_;_]   : OrientEqSet RewSet -> SolveIO .
op   bottomC : -> SolveIO .
op   canonSTO : OrientEqSet -> OrientEqSet .

vars SIO     : SolveIO .

eq  canonSTO(mt) = mt .
eq  canonSTO((L1 --> L2) STO) =
      (canon(L1) --> canon(L2)) canonSTO(STO) .

op  solveDecomp  : SolveIO -> [SolveIO] .
op  solveSolve1  : SolveIO -> [SolveIO] .
op  solveSolve2  : SolveIO -> [SolveIO] .
op  solveTrivial : SolveIO -> [SolveIO] .
op  solveElim    : SolveIO -> [SolveIO] .
op  solveCanon   : SolveIO -> [SolveIO] .

eq  solveDecomp([ (cons(H ; T) --> L) STO ; RS ]) =
      [ (H --> car(L)) (T --> cdr(L)) STO ; RS ] .

eq  solveSolve1([ (car(L1) --> L2) STO ; RS ]) =
      [ (L1 --> cons(L2 ; {NEW})) STO ; RS ] .

eq  solveSolve2([ (cdr(L1) --> L2) STO ; RS ]) =
      [ (L1 --> cons({NEW} ; L2)) STO ; RS ] .

eq  solveTrivial([ (L --> L) STO ; RS ]) =
      [ STO ; RS ] .

ceq solveElim([ ({X} --> L) STO ; RS ]) =
      if   X in var(L')
      then bottomC
      else [ STO @ ({X} => L) ; ({X} => L') (RS # ({X} => L)) ]
      fi if L' := canon(L) .

ceq solveCanon([ STO ; RS ]) = [ canonSTO(STO) ; RS ]
      if STO =/= mt .

*** strategy
```

```
  subsorts SolveIO < StratInOut .

  op  solveStrategy : -> Strategy .

  eq  solveStrategy =
        (['canon] &? ['trivial]* &?
         (['decomp] || ['solve1] || ['solve2] || ['elim])*
        )* .

  ceq solve(L1 --> L2) = RS
        if SIO          := [ (L1 --> L2) ; mt ]
        /\ [ STO ; RS ] := reduceWithStrat(solveStrategy, SIO) .

  eq  solve(L1 --> L2) = bottom [owise] .

endfm
```