Élaboration pour le λΠ-calcul modulo

Rafaël Bocquet
Supervisé par Fréderic Blanqui
Équipe Deducteam au LSV, Cachan

23 août 2017

Le contexte général

Les termes de lambdas-calculs avec types dépendants sont une représentation efficace de preuves pour différentes logiques. Un terme peut cependant contenir de l'information redondante que l'on voudrait pouvoir omettre lors de l'écriture. Le λΠ-calcul modulo est une extension du λΠ-calcul, qui peut être vu comme un λ-calcul avec types dépendants minimal, dont la conversion est enrichie par des règles de réécritures arbitraires données par l'utilisateur.

Le problème étudié

On étudie le problème de l'élaboration pour le λΠ-calcul modulo, c'est à dire de la reconstruction à partir d'un terme incomplet, dont certains sous-termes ont été omis, d'un terme complet bien typé. Différentes variantes de tels algorithmes sont présents dans la plupart des assistants de preuve, tels que Coq ou Agda. On veut ici savoir comment prendre compte de la réécriture pour proposer un algorithme d'élaboration pour le λΠ-calcul modulo. Une composante essentielle de l'élaboration sera la génération de contraintes d'unification. On s'interesse donc en particulier à la résolution de problèmes d'unification dans le λΠ-calcul modulo.

La contribution proposée

On généralise des algorithmes d'élaboration et d'unification existants au λΠ-calcul modulo, en s'intéressant en particulier à l'interaction des algorithmes avec la présence de réécriture. Une implémentation partielle de ces algorithmes a été implémentée en OCaml.

Les arguments en faveur de sa validité

La plupart des problèmes d'élaboration issus de termes écrits par un utilisateur sont assez simples pour être résolus facilement par l'algorithme sans spécifiquement prendre en compte la réécriture. Pouvoir résoudre des

problèmes plus difficiles reste important pour la facilité d'utilisation d'un assistant de preuve. Lorsque ces algorithmes doivent faire un choix en raison de la présence d'un système de réécriture, le meilleur choix dépend souvent du système de réécriture particulier.

Le bilan et les perspectives

On peut généraliser les algorithmes d'élaboration et d'unification utilisés dans les assistants de preuves existants basés sur des λ-calculs à types dépendants à des systèmes incorporant de la réécriture arbitraire. Traiter les difficultés supplémentaires apportées par la réécriture semble nécessiter de s'adapter aux règles de réécriture particulières.

Table des matières

1 Introduction

Les théories des types dépendantes sont des systèmes logiques, dont les preuves sont représentées par les termes d'un langage, ce qui permet la construction de vérificateurs de preuves, ainsi que d'assistants de preuve. Écrire des termes de ces langages peut cependant demander à un utilisateur de donner plus d'information que nécessaire, ce qui peut rendre l'écriture, ainsi que la lecture, de ces preuves difficile. On cherche donc à avoir un langage *externe* dans lequel l'utilisateur peut écrire des termes de preuves, et un algorithme *d'élaboration* qui reconstruit à partir de ces termes un terme du langage *interne*. De tels algorithmes ont été étudiés pour différents systèmes ([1, 2, 3]) et sont utilisés par la plupart des assistants de preuve basés sur une théorie des types dépendante.

On s'intéresse à des théories des types dépendantes dont la relation de conversion a été enrichie par un système de réécriture. Considérer des systèmes de réécritures arbitraires permet d'avoir une logique plus expressive, d'avoir des termes plus petits en replaçant des preuves par des calculs, ... Plus particulièrement, le travail a été effectué dans le contexte du λΠ-calcul modulo, qui peut être vu comme un cadre minimal pour ce travail : c'est

un λ-calcul dépendamment typé pour lequel on peut spécifier un système de réécriture, avec peu d'autres particularités.

De même que pour les assistants de preuves cités, l'élaboration passe par la résolution de problèmes d'unification, et le cœur du travail présenté se situe au niveau de la construction d'un algorithme d'unification pour le $\lambda\Pi$-calcul modulo.

2 $\lambda\Pi$-calcul modulo

On présente dans cette section le $\lambda\Pi$-calcul modulo.

On notera $A, B, \ldots$ pour des types, $a, b, u, v, \ldots$ pour des termes arbitraires, $x, y, z, \ldots$ pour des variables, et $c, d, \ldots$ pour des symboles de constantes. La syntaxe des termes du $\lambda\Pi$-calcul modulo est générée par :

$$A, B, u, v ::= \mathsf{Type} \mid \mathsf{Kind} \mid x \mid u\ v \mid \Pi x : A.B \mid \lambda x : A.b \mid c$$

On se donne également un contexte global Σ, ou signature, qui contient un ensemble de constantes typées, ainsi qu'un ensemble de règles de réécritures.

La relation de réduction entre termes $\triangleright$ est générée par la réduction β, l'expansion η et les règles de réécritures de Σ. On notera $\equiv$ la relation de conversion entre termes, clôture transitive de $\triangleright$.

Le jugement de typage $\Gamma \vdash a : A$, défini inductivement.

2.1 Typage

Les règles de typage du $\lambda\Pi$-calcul modulo sont :

$$\frac{}{\emptyset \ \mathsf{wf}} \qquad \frac{\Gamma \ \mathsf{wf} \qquad \Gamma \vdash A : s \qquad s \in \{\mathsf{Type}, \mathsf{Kind}\}}{\Gamma, x : A \ \mathsf{wf}}$$

$$\frac{\Gamma \ \mathsf{wf}}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}} \ \textsc{Type} \qquad \frac{\Gamma \ \mathsf{wf} \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \ \textsc{Variable}$$

$$\frac{\Gamma \vdash u : \Pi x : A.B \qquad \Gamma \vdash v : A}{\Gamma \vdash u \ v : B[x := v]} \ \textsc{Application}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s} \ \textsc{Pi}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.b : \Pi x : A.B} \ \textsc{Lambda}$$

$$\frac{(c : A \in \Sigma)}{\Gamma \vdash c : A} \ \textsc{Constante}$$

$$\frac{\Gamma \vdash t : A \qquad A \equiv B}{\Gamma \vdash t : B} \ \textsc{Conversion}$$

2.2 Propriétés

Certaines propriétés du $\lambda\Pi$-calcul ne sont plus toujours valides après ajout d'un système de réécriture. On restreint donc le systèmes de réécriture pour en récupérer certaines. En particulier, on le restreint de manière à ce que la réduction $\triangleright$ soit confluante, termine, et préserve le typage.

2.3 Exemple

On donne un exemple de termes pouvant être écrits dans le $\lambda\Pi$-calcul modulo, en utilisant la syntaxe de Dedukti :

```
Nat : Type.
Z : Nat.
S : Nat → Nat.

plus : Nat → Nat → Nat.
plus Z y ⇝ y.
plus x Z ⇝ y.
plus (S x) y ⇝ S (plus x y).
plus x (S y) ⇝ S (plus x y).
plus x (plus y z) ⇝ plus (plus x y) z.

A : Type.

Vec : Nat → Type.
Nil : Vec Z.
Cons : Π n:Nat. A → Vec n → Vec (S n).

append : Π n:Nat. Π m:Nat. Vec n → Vec m →
Vec (plus n m).
append _ _ Nil ys ⇝ ys.
append _ _ xs Nil ⇝ ys.
append _ m (Cons n x xs) ys ⇝ Cons (plus n m) x (append n m xs ys).
append n _ xs (append m l ys zs) ⇝
    append (plus n m) l (append n m xs ys) zs.
```

On remarque que le typage des deux dernières règles associées à `append` nécessite certaines de règles de réécriture de `plus`.

En rouge sont des sous-termes que l'utilisateur peut vouloir omettre et que le système devrait pourvoir inférer. En effet, ces termes sont ici déjà présents dans les types des autres arguments de `append` ou de `Cons`.

3 Unification

Le problème d'unification consiste à trouver une substitution des variables de termes afin de les rendre convertible. Les sous problèmes de l'unification d'ordre supérieure ainsi que de l"unification équationnelle pour certains systèmes d'équations étant déjà indécidables [4], le problème d'unification pour le $\lambda\Pi$-calcul modulo est aussi indécidable. On ne pourra donc pas résoudre n'importe quel problème d'unification. Cependant, on peut espérer pouvoir résoudre les problèmes d'unification produits dans le cadre de l'élaboration.

Dans cette section, on donne les définitions de l'unification et les choix

des représentations des problèmes et solutions, afin de pouvoir décrire dans la section 4 la procédure d'élaboration. L'approche adoptée pour la résolution des problèmes sera donnée dans la section 5.

3.1 Métavariables

On étend la syntaxe des termes du calcul par des métavariables, qui représentent les sous-termes qui doivent être instanciés dans une solution à un problème d'unification.

$$A, B, u, v ::= \ldots \mid ?_i[\overrightarrow{u}]$$

Le jugement de typage est étendu par un contexte des métavariables $\mathcal{M}$, qui contient les types des métavariables, et d'une règle d'inférence pour les métavariables.

Definition 1 (Typage des métavariables)**.**

$$\frac{(\overrightarrow{x : A} \vdash ?_i[\overrightarrow{x}] : B) \in \mathcal{M} \qquad \overrightarrow{\mathcal{M}, \Gamma \vdash u : A[\overrightarrow{x := u}]}}{\mathcal{M}, \Gamma \vdash ?_i[\overrightarrow{u}] : B[\overrightarrow{x := u}]}$$

L'algorithme d'unification demandera de connaitre à quels arguments une métavariable est appliquée. Mais, en raison de la présence de types dépendants dans le $\lambda\Pi$-calcul, un type n'est pas suffisant pour fixer l'arité d'une métavariable appliquée. (Par exemple, si $?_i$: $\mathsf{if}\ b\ \mathsf{then}\ \mathsf{Nat}\ \mathsf{else}\ \mathsf{Nat} \to \mathsf{Nat}$, alors $?_i\ \mathsf{true}$ et $?_i\ \mathsf{false}$ ont respectivement pour types Nat et $\mathsf{Nat} \to \mathsf{Nat}$). Ainsi, plutôt que de juste associer des types aux métavariables ($\vdash ?_i : B$), $\mathcal{M}$ associe aux métavariables des types dans un contexte ($\overrightarrow{x : A} \vdash ?_i[\overrightarrow{x}] : B$). Cela permet de garder une notion d'arité pour les métavariables et d'identifier les occurrences des métavariables qui ne sont appliquées exactement au nombre de termes prévu par leur arité.

On notera $\mathcal{T}(\mathcal{M})$ l'ensemble des termes dans le métacontexte $\mathcal{M}$.

3.2 Problème d'unification

On choisit d'avoir des contraintes d'unification hétérogènes, dont les deux termes à unifier peuvent avoir des types différents, dans des contextes différents, plutôt que des contraintes hétérogènes. Cela permet de représenter plus de problèmes, et en particulier de pouvoir simplifier certaines contraintes d'unification. Dans toute solution les deux termes deviennent cependant convertibles, et ont donc alors le même type.

La représentation de ces contraintes hétérogènes utilise des paires de contextes de même taille. On notera $\Gamma_1 \ddagger \Gamma_2$ une telle paire. Les types apparaissant dans ces contextes devant être convertibles dans toute solution, on notera souvent une telle paire de contextes comme un seul contexte Γ, en

explicitant quand nécessaire. On écrit $x : A_1 \ddagger A_2 \in \Gamma_1 \ddagger \Gamma_2$ pour $x : A_1 \in \Gamma_1 \wedge x : A_2 \in \Gamma_2$, ou $x : A \in \Gamma_1 \ddagger \Gamma_2$ dans le cas où A_1 et A_2 sont convertibles. De même, on notera $\Gamma, x : A_1 \ddagger A_2$ ou $\Gamma, x : A$ pour l'extension d'une paire de contextes.

Cette représentation des contraintes est introduite par Adam Gundry dans sa thèse [5]. Dans [6], Jason Reed utilise également des contraintes hétérogènes, mais avec un prédicat de typage global au problème d'unification.

Definition 2 (Problème d'unification)**.** *Une contrainte d'unification* $\Gamma_1 \ddagger \Gamma_2 \vdash t_1 : A_1 \stackrel{?}{=} t_2 : A_2$ *dans un métacontexte* $\mathcal{M}$ *est donnée par une paire de contextes* $\Gamma_1 \ddagger \Gamma_2$ *et deux termes bien typés* $\Gamma \vdash t_1 : A_1$ *et* $\Gamma \vdash t_2 : A_2$.
Un problème d'unification $(\mathcal{M}, \mathcal{P})$ *est une conjonction de contraintes d'unification dans un même méta-contexte* $\mathcal{M}$.

Definition 3 (Solution)**.** *Une solution à un problème d'unification* $(\mathcal{M}, \mathcal{P})$ *est une substitution bien typée* σ *des métavariables de* $\mathcal{M}$ *vers des termes avec des métavariables dans un nouveau métacontexte* $\mathcal{M}'$ *telle que pour tout* $\Gamma \vdash t_1 \stackrel{?}{=} t_2 \in \mathcal{P}$, $t_1[\sigma] \equiv t_2[\sigma]$.

On notera alors $\mathcal{M}, \mathcal{P} \rightsquigarrow_\sigma \mathcal{M}'$.

Une solution ne doit pas nécessairement substituer toutes les variables de $\mathcal{M}$. Dans une solution au problème d'élaboration, toutes les métavariables devront être substituées, car on souhaite obtenir des termes sans métavariables, mais l'élaboration se décomposera en plusieurs résolutions distinctes de problèmes d'unification.

3.3 Règles d'unification

Pour présenter un algorithme d'unification, on donne un ensemble de règles applicables à certains problèmes, ce qui définit une relation entre problèmes d'unification.

Definition 4 (Règle d'unification)**.** *Une règle d'unification est une paire de problèmes* $(\mathcal{M}, \mathcal{P})$ *et* $(\mathcal{M}', \mathcal{P}')$ *et une substitution* τ *des métavariables de* $\mathcal{M}$ *vers des termes de* $\mathcal{T}(\mathcal{M}')$*, tels que pour toute solution* σ *de* $(\mathcal{M}', \mathcal{P}')$*,* $\tau\sigma$ *est une solution de* $(\mathcal{M}, \mathcal{P})$.

On notera $(\mathcal{M}, \mathcal{P}) \rightsquigarrow_\tau (\mathcal{M}', \mathcal{P}')$ une règle d'unification, en laissant souvent implicites τ ou des sous-ensembles de $\mathcal{M}$ et $\mathcal{P}$ sur lesquels la règle n'agit pas.

On se donne quelques règles d'unification basiques dans la definition 5.

On peut réduire des termes apparaissant dans un problème d'unification. On pourrait considérer des problèmes dont tous les termes sont toujours en forme normale, mais il peut être utile pour certaines règles de garder des termes proches de ceux écrits par l'utilisateur. La plupart des règles

d'unification dépendront de la forme normale ou de la forme normale de tête, et appliqueront implicitement la règle de réduction.

La deuxième règle augmente la taille du contexte d'une métavariable quand son type est convertible à un type de fonction (dépendante).

Definition 5 (Règles d'unification basiques)**.**

$$\frac{\mathcal{M} \rhd \mathcal{M}' \qquad \mathcal{P} \rhd \mathcal{P}'}{\mathcal{M}, \mathcal{P} \rightsquigarrow \mathcal{M}', \mathcal{P}'} \textsc{ Réduction}$$

$$(\overrightarrow{x : A} \vdash ?_i : \Pi(y : B).C, -) \rightsquigarrow_{[?_i[\overrightarrow{x}] := \lambda y:B.?'_i[\overrightarrow{x}, y]]} (\overrightarrow{x : A}, y : B \vdash ?'_i : C, -)$$

Les quatre règles qui suivent peuvent être vues comme des règles élémentaires permettant de décomposer les règles de la section 5, et ne seront pas considérées dans l'algorithme d'unification.

$$\frac{\overrightarrow{x : A} \vdash u : B}{((\overrightarrow{x : A} \vdash ?_i : B) + \mathcal{M}, \mathcal{P}) \rightsquigarrow (\mathcal{M}, \mathcal{P})[?_i[\overrightarrow{x}] := u]} \textsc{ Instanciation}$$

$$\frac{}{(\mathcal{M}, (\Gamma \vdash u \stackrel{?}{=} u) = \mathcal{P}) \rightsquigarrow (\mathcal{M}, \mathcal{P})} \textsc{ Suppression de contrainte}$$

$$\frac{\overrightarrow{x : A} \vdash B : Type}{(\mathcal{M}, \mathcal{P}) \rightsquigarrow ((\overrightarrow{x : A} \vdash ?_i : B) + \mathcal{M}, \mathcal{P})} \textsc{ Ajout de métavariable}$$

$$\frac{\Gamma_1 \vdash u : A \qquad \Gamma_2 \vdash v : B}{(\mathcal{M}, \mathcal{P}) \rightsquigarrow (\mathcal{M}, (\Gamma \vdash u \stackrel{?}{=} v) + \mathcal{P})} \textsc{ Ajout de contrainte}$$

4 Élaboration

L'élaboration est la reconstruction à partir d'un terme entré par l'utilisateur dans une certaine syntaxe externe, d'un terme du calcul interne. Dans notre cas, on veut reconstruire un terme bien typé du $\lambda\Pi$-calcul modulo à partir d'un terme non typé dont certains sous-termes sont laissé implicites, ce qui est représenté en étendant la syntaxe des termes :

$$A, B, u, v ::= \ldots \mid ?$$

Ces termes ne contiennent pas de métavariables. On notera $\mathcal{T}(?)$ l'ensemble de ces termes.

Definition 6 (Élaboration)**.** *Un problème d'élaboration* (Γ, u) *est composé d'un contexte bien-formé* Γ *de* $\mathcal{T}(\emptyset)$ *et d'un terme* u *de* $\mathcal{T}(?)$*.*

Une solution à un problème (Γ, u) *est un terme* v *de* $\mathcal{T}(\emptyset)$ *raffinant* u *(obtenu par substitution des sous-termes implicites de* u*), bien typé dans* Γ*.*

Dans le cas où un terme n'a pas de sous-terme implicite, le problème d'élaboration se ramène à la vérification de typage.

4.1 Élaboration bidirectionnelle

On définit la procédure d'élaboration par deux relations définies inductivement :

$$\mathcal{M}, \Gamma \vdash u \Uparrow u' : A, \mathcal{M}' \qquad\qquad \mathcal{M}, \Gamma \vdash u : A \Downarrow u', \mathcal{M}'$$

Cela définit un algorithme bidirectionnel d'élaboration. $\Gamma \vdash u \Uparrow u' : A$ correspond à l'élaboration d'un terme u vers un terme u', en inférant un type A. $\Gamma \vdash u : A \Downarrow u'$ permet de guider l'élaboration quand on connait déjà le type du terme élaboré.

On écrira, pour simplifier, $\Gamma \vdash A : s \Downarrow A'$ pour la vérification que A est un type de sorte s, ce qui doit en fait aussi inférer s.

$$\frac{\mathcal{M}_0, \Gamma \vdash u \Uparrow u' : A', \mathcal{M}_1 \qquad \mathcal{M}_1, \Gamma \vdash A \stackrel{?}{=} A' \rightsquigarrow \mathcal{M}_2}{M_0, \Gamma \vdash u : A \Downarrow u', M_2}$$

Cette première règle est un cas par défaut de $\Downarrow$, qui procède en inférant le type de u, avant de l'unifier avec le type attendu.

On omettra les métacontextes des autres règles.

On considère une seule autre règle pour $\Downarrow$, associée à λ.

$$\frac{\Gamma \vdash A_1 : s \Downarrow A_1' \qquad \Gamma \vdash A_1' \stackrel{?}{=} A_2 \rightsquigarrow \ldots \qquad \Gamma \vdash b : B \Downarrow b'}{\Gamma \vdash \lambda x : A_1.b : \Pi x : A_2.B \Downarrow \lambda x : A_1.b'} \Downarrow\textsc{Lambda}$$

L'élaboration remplace un sous-terme implicite par une nouvelle métavariable, dont le type est également une nouvelle métavariable.

$$\frac{?_i \notin \mathcal{M} \qquad ?_j \notin \mathcal{M}}{\mathcal{M}, \Gamma \vdash ? \Uparrow ?_i : A, \mathcal{M} + (\Gamma \vdash ?_i[\Gamma] : ?_j[\Gamma], \Gamma \vdash ?_j[\Gamma] : s)} \Uparrow\textsc{Implicite}$$

Les règles correspondant à la plupart des autres constructions correspondent aux règles de typage.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Uparrow x : A} \Uparrow\textsc{Variable} \qquad \frac{c : A \in \Sigma}{\Gamma \vdash c \Uparrow c : A} \Uparrow\textsc{Constante}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \Downarrow A' \qquad \Gamma, x : A' \vdash B \Uparrow B' : s}{\Gamma \vdash \Pi x : A.B \Uparrow \Pi x : A'.B' : s} \Uparrow\textsc{Pi}$$

$$\frac{\Gamma \vdash A : s \Downarrow A' \qquad \Gamma, x : A' \vdash b \Uparrow b' : B}{\Gamma \vdash \lambda x : A.b \Uparrow \lambda x : A'.b' : \Pi A'.B} \Uparrow\textsc{Lambda}$$

Il reste à traiter l'application. On pourrait également donner une règle déduite de la règle de typage :

$$\frac{\Gamma \vdash u \Uparrow u' : A \qquad \Gamma \vdash A \stackrel{?}{=} \Pi x : A'.B \rightsquigarrow \ldots \qquad \Gamma \vdash v : A' \Downarrow v'}{\Gamma \vdash u\ v \Uparrow u'\ v' : B[x := v']}$$

On fait le choix cependant d'utiliser les règles proposées dans [1] :

$$\frac{\Gamma \vdash t \Uparrow t' : T \qquad \Gamma \vdash t' : T \mid \overrightarrow{us} \Uparrow v : V}{\Gamma \vdash t\ \overrightarrow{us} \Uparrow v : V} \Uparrow\textsc{Application}$$

$$\frac{}{\Gamma \vdash t\ \overrightarrow{(x_i := u_i : A_i)} : T \mid \emptyset \Uparrow t\ \overrightarrow{u_i} : T} \mathcal{E}^T\textsc{Vide}$$

$$\frac{\begin{array}{c} T \rhd \Pi x : A.B \qquad \Gamma \vdash v : A \Downarrow v' \\ \Gamma \vdash t\ \overrightarrow{(x_i := u_i : A_i)}\ (x := v' : A) : B[x := v'] \mid \overrightarrow{vs} \Uparrow v : V \end{array}}{\Gamma \vdash t\ \overrightarrow{(x_i := u_i : A_i)} : T \mid v\ \overrightarrow{vs} \Uparrow v : V} \mathcal{E}^T\textsc{Produit}$$

$$\frac{\begin{array}{c} T \rhd ?_i[\ldots] \qquad \Gamma \vdash v \Uparrow v' : A \\ \mathcal{M} \rightsquigarrow \mathcal{M} + (\Gamma, \overrightarrow{x_i : A_i}, x : A \vdash ?_j : s) \\ \Gamma \vdash T \stackrel{?}{=} \Pi x : A.?_j[\Gamma, \overrightarrow{x_i}, x] \rightsquigarrow \\ \Gamma \vdash t\ \overrightarrow{(x_i := u_i : A_i)}\ (x := v' : A) :?_j[\Gamma, \overrightarrow{x_i}, x] \mid \overrightarrow{vs} \Uparrow v : V \end{array}}{\Gamma \vdash t\ \overrightarrow{xs := us} : T \mid v\ \overrightarrow{vs} \Uparrow v : V} \mathcal{E}^T\textsc{Flexible}$$

La présence de règles $\mathcal{E}^T$Produit et $\mathcal{E}^T$Flexible séparées permet de mieux exploiter la bidirectionnalité de l'algorithme. Dans le cas Produit, on connait déjà le domaine de l'application, et on peut l'utiliser pour vérifier le type de l'argument. Dans le cas Flexible, on infère le type de l'argument.

La présence des règles $\mathcal{E}^T$ permet de traiter l'application à tous les arguments, et dans la règle $\mathcal{E}^T$Flexible de créer une nouvelle métavariable dépendant des arguments déjà rencontrés de l'application, ce qui permet d'inférer des types de fonctions dépendantes.

4.2 Exemple

On donne l'exemple des contraintes d'unification générées par l'élaboration de l'un des termes contenant des termes implicites de l'exemple de la section 2.3. Les métavariables apparaissent toutes ici appliquées aux variables du contexte ($[n : \mathsf{Nat}, m : \mathsf{Nat}, x : A, xs : \mathsf{Vec}\ n, ys : \mathsf{Vec}\ m]$), ce que l'on remplacera par $[\dots]$.

Terme :

$$\mathsf{Cons}\ ?\ x\ (\mathsf{append}\ ?\ ?\ xs\ ys)$$

Élaboration :

$$\frac{\cdots \vdash ?_i : \mathsf{Nat} \qquad \cdots \vdash ?_j : \mathsf{Nat} \qquad \cdots \vdash ?_k : \mathsf{Nat} \qquad \mathsf{Vec}\ ?_j[\dots] \stackrel{?}{=} \mathsf{Vec}\ n \qquad \mathsf{Vec}\ ?_k[\dots] \stackrel{?}{=} \mathsf{Vec}\ m \qquad \mathsf{Vec}\ ?_i[\dots] \stackrel{?}{=} \mathsf{Vec}\ (\mathsf{plus}\ ?_j[\dots]\ ?_k[\dots])}{n : \mathsf{Nat}, m : \mathsf{Nat}, x : A, xs : \mathsf{Vec}\ n, ys : \mathsf{Vec}\ n \vdash \mathsf{Cons}\ ?\ x\ (\mathsf{append}\ ?\ ?\ xs\ ys) \Uparrow \mathsf{Cons}\ ?_i[\dots]\ x\ (\mathsf{append}\ ?_j[\dots]\ ?_k[\dots]\ xs\ ys)}$$

Solution :

$$?_j[\dots] := n \qquad ?_k[\dots] := m \qquad ?_i[\dots] := \mathsf{plus}\ n\ m$$

5 Résolution des problèmes d'unification

5.1 Procédure de simplification

L'unification d'ordre supérieure est indécidable, mais on peut identifier certains fragments décidables, notamment celui des *motifs* décrit par Dale Miller pour le λ-calcul simplement typé [7]. On peut décrire l'algorithme de décision par des règles d'unification à appliquer jusqu'à arriver à une solution, ou un échec. Ce fragment est cependant trop restreint pour résoudre suffisamment de problèmes d'unifications. On peut cependant utiliser ces règles d'unification sur un problème hors du fragment des motifs, ce qui permet parfois de résoudre tout de même le problème, car des contraintes hors du fragment peuvent y entrer après instanciation, et donne sinon une simplification du problème [6, 8]. En effet, les règles d'unification décrivant cette procédure ont la particularité de préserver l'ensemble des solutions du problème.

On dit qu'un terme est rigide si aucune réduction ne peut s'effectuer en tête du terme dans aucune substitution de métavariables. En l'absence de réécriture, cela ne demande que d'inspecter la tête du terme, mais en présence de réécriture, on veut . On dit qu'un sous-terme v d'un terme u est en position rigide si tous les sous-termes de u contenant v sont rigides.

On donne ici la description d'une telle procédure de simplification pour

le $\lambda\Pi$-calcul modulo. Les règles sont à considérer en addition aux règles déjà données dans la definition 5.

Definition 7 (Règles d'unifications de la procédure de simplification)**.**

Décompositions rigide-rigide

$\Gamma \vdash \Pi x : A_1.B_1 \stackrel{?}{=} \Pi x : A_2.B_2 \quad \rightsquigarrow \quad \Gamma \vdash A_1 \stackrel{?}{=} A_2$
$\Gamma, x : (A_1 \ddagger A_2) \vdash B_1 \stackrel{?}{=} B_2$

$\Gamma \vdash \lambda x : A_1.B_1 \stackrel{?}{=} \lambda x : A_2.B_2 \quad \rightsquigarrow \quad \Gamma \vdash A_1 \stackrel{?}{=} A_2$
$\Gamma, x : (A_1 \ddagger A_2) \vdash B_1 \stackrel{?}{=} B_2$

$\Gamma \vdash R\ \overrightarrow{x_1} \stackrel{?}{=} R\ \overrightarrow{x_2} \quad \rightsquigarrow \quad \Gamma \vdash \overrightarrow{x_1} \stackrel{?}{=} \overrightarrow{x_2}$
(Si $R\ \overrightarrow{x_1}$ et $R\ \overrightarrow{x_2}$ sont rigides)

Échec

$\Gamma \vdash R_1\ \overrightarrow{x_1} \stackrel{?}{=} R_1\ \overrightarrow{x_2} \quad \rightsquigarrow \quad \bot$
(Si $R_1\ \overrightarrow{x_1}$ et $R_2\ \overrightarrow{x_2}$ sont rigides et $R_1 \neq R_2$)

$\Gamma \vdash ?_i[\sigma] \stackrel{?}{=} u \quad \rightsquigarrow \quad \bot$
(Si σ est un motif et
$?_i$ apparait en position fortement rigide dans u)

Instanciation

$\Gamma \vdash ?_i[\sigma] \stackrel{?}{=} u \quad \rightsquigarrow \quad ?_i[\sigma] := u$
(Si σ est un motif, $?_i$ n'apparait pas dans u,
toutes les variables libres de u sont dans σ et
les deux types de toutes les variables de σ sont convertibles)

5.1.1 Non-déterminisme

Lorsque cette procédure de simplification aboutit à une solution en instanciant toutes les métavariables, la solution obtenue est unique malgré la possibilité de choix non-déterministes de règles à appliquer (en raison de la préservation de l'ensemble des solutions). Cependant, lorsque cette procédure est utilisée pour simplifier des problèmes avant l'application de règles heuristiques, le problème obtenu après simplification n'est pas nécessairement unique.

On peut construire des problèmes d'unification pour lesquels différents problèmes simplifiés peuvent être obtenus en raison de la présence de réécriture :

On considère cette signature :

A : **Type**
B : **Type**
X : A
Y : B

F : A → B

G : A → B
F X ⇝ Y
G X ⇝ Y

H : B → A
H (F a) → X
H Y → X

Le problème $?_i \stackrel{?}{=} F(?_k), ?_i \stackrel{?}{=} G(?_k)$ sera simplifié ou en $?_i := F?_k, F?_k \stackrel{?}{=} G?_k$, ou en $?_i := G?_k, F?_k \stackrel{?}{=} G?_k$. Une solution possible est $?_k := X, ?_i := Y, ?_j := Y$. En présence de contraintes supplémentaires contenant $H?_i$ comme sous-terme, selon les choix de la procédure de simplification, ce sous-terme sera où non simplifié vers X, ce qui peut affecter la suite de l'unification.

5.2 Heuristiques

La procédure de simplification décrite permet de résoudre une partie des problèmes venant de l'élaboration, mais reste insuffisante. On cherche donc à l'étendre avec différentes règles heuristiques, à appliquer lorsque le problème ne peut plus être simplifié.

Un exemple de contrainte, qui peut apparaitre lors de l'élaboration d'applications d'un terme de type $\Pi n : Nat.\Pi m : Nat.\mathsf{Vec}\ (n+m) \to \mathsf{Vec}\ n$, qui ne serait pas résolue par l'algorithme de simplification est $\mathsf{Vec}\ (n + ?_i) \stackrel{?}{=} \mathsf{Vec}\ (n+m)$. Cette contrainte serait simplifiée en $n + ?_i \stackrel{?}{=} n + m$, qui ne peut être simplifiée, car $n + ?_i$ n'est pas un terme rigide.

Definition 8 (Heuristique d'unification au premier ordre)**.**

$$\Gamma \vdash F\ \overrightarrow{x_1} \stackrel{?}{=} F\ \overrightarrow{x_2} \rightsquigarrow \Gamma \vdash \overrightarrow{x_1} \stackrel{?}{=} \overrightarrow{x_2}$$

6 Conclusion

Des procédures d'élaboration et d'unification pour le λ-calcul modulo ont été introduites. Si l'algorithme d'élaboration n'a pas vraiment besoin d'un traitement particulier de la présence de réécriture, l'unification pourrait en bénéficier. Il semble cependant que cela nécessite de spécialiser les règles additionnelles d'unification aux système de réécriture. On pourrait donc chercher à obtenir un algorithme d'unification extensible par l'utilisateur permettant de spécifier un comportement de l'algorithme adapté aux règles de réécriture considérées. Des éléments extensibles d'algorithmes d'unification, comme les *Type Classes* [9], les *indices d'unification* [10], ont déjà été étudiées.

Références

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[2] Beta Ziliani and Matthieu Sozeau. A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. *J. Funct. Program.*, 27 :e10, 2017.

[3] Leonardo Mendonça de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *CoRR*, abs/1505.04324, 2015.

[4] Gilles Dowek. Higher-order unification and matching. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 1009–1062. 2001.

[5] Adam Michael Gundry. *Type inference, Haskell and dependent types.* PhD thesis, University of Strathclyde, Glasgow, UK, 2013.

[6] Jason Reed. Higher-order constraint simplification in dependent type theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages : Theory and Practice, LFMTP '09, McGill University, Montreal, Canada, August 2, 2009*, pages 49–56, 2009.

[7] Dale Miller. Unification of simply typed lamda-terms as logic programming. In *Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*, pages 255–269, 1991.

[8] Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, pages 10–26, 2011.

[9] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 278–293, 2008.

[10] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 84–98, 2009.