Rapport de projet GRAL

9 février 2018

Higher Order Unification [1]

Roman Delgado
Gaspard Denis
Alexandre Fernandez
Hugo Hourcade
Sylvain Ung

1. https://github.com/LexTek/unification_lambda_sigma

Table des matières

1 Introduction

Dans le cadre de l'enseignement de Groupe de Recherche en Algorithmique, il va s'agir dans ce projet, qui s'inscrit dans le développement de l'assistant de preuve LaTTe [2] par Frédéric Peschanski, d'implémenter un algorithme d'unification d'ordre supérieur.

L'unification est une opération fondamentale dans les algorithmes de déduction et consiste en la recherche d'une substitution entre un ou plusieurs termes afin de les rendre égaux. L'unification est en quelque sorte une équation à résoudre, on dit alors que les termes sont unifiables si une telle substitution est possible.

L'unification présente plusieurs aspects, notamment l'unification du premier ordre qui est un problème que l'on sait déjà très bien résoudre. Ici on va s'intéresser à l'unification d'ordre supérieur (ou *higher order unification*) qui est un problème semi-décidable puisque l'algorithme se termine pour des entrées qui ont une solution et peut boucler pour des entrées sans solution.

2. `https://github.com/latte-central/latte-checker`

2 État de l'art

2.1 λ-calcul

Nous allons tout d'abord commencer par définir plusieurs notions de λ-calcul qui est l'élément de base sur lequel nous travaillons.

2.1.1 λ-calcul nommé

Soit V un ensemble de variables ($V = x, y, \ldots$). Les termes du λ-calcul sont définis par :

$$a ::= x \mid (a\, a) \mid \lambda x.a$$

Le terme $\lambda x.a$ représente la fonction qui a x associe a (le corps de la fonction), et le terme $(M\, N)$ représente le résultat de la fonction M appliquée à N. Si on applique la fonction $\lambda x.a$ à un terme b, alors on obtient la valeur de a pour laquelle toutes les occurrences de x ont été remplacées par b. Cette opération est appelée β-réduction. Avant d'effectuer une β-réduction, les variables liées qui apparaissent à la fois dans a et dans b doivent se voir subir une α-conversion : on renomme ces variables pour a ou b de sorte qu'aucune variable ne soit capturée par mégarde. Une autre règle, la η-réduction, est aussi ici introduite.

Definition 2.1.1. Soit a un terme et $FV(a)$ l'ensemble des variables libres de a. Pour une évaluation θ liant les variables $x_1, \ldots, x_n$ aux variables $a_1, \ldots, a_n$, la substitution qui étend θ notée $\bar{\theta} = x_1/a_1, \ldots, x_n/a_n$ est définie par :

$$\begin{aligned}
&\bar{\theta}x = \theta x \\
&\bar{\theta}(a\, b) = (\bar{\theta}a\, \bar{\theta}b) \\
&\bar{\theta}(\lambda y.a) = \lambda z(\bar{\theta}\{y/z\}a)
\end{aligned}$$

où z est une nouvelle variable telle que $\theta z = z$, z n'apparaît pas dans a, et $\forall x \in FV(a) \quad z \notin FV(\theta x)$.

Definition 2.1.2. La β-réduction est définie par la règle :

$$(\lambda x.a)b \to \{x/b\}a$$

La η-réduction est définie par la règle :

$$\lambda x.(a\, x) \to a \quad \text{si } x \notin FV(a)$$

2.1.2 Les variables d'unification

Dans un λ-terme, on peut distinguer trois types de variables :
— Les variables libres
— Les variables liées
— Les vraies variables d'unification qui définissent le problème d'unification.

$$v ::= x \mid X$$

Ces dernières seront appelées les *métavariables*. On notera V l'ensemble des variables libres et liées, et H l'ensemble des métavariables.

L'introduction de ces dernières nous permet alors de définir de manière plus précise les termes du λ-calcul :

Definition 2.1.3. L'ensemble $\Lambda(V, H)$ des λ-termes ouverts est défini par :

$$a ::= x \mid X \mid (a\, a) \mid \lambda x.a$$

où $x \in V$ et $X \in H$.

Nous avons donc maintenant deux notions de substitution :
— la première fonctionne sur V : elle est utilisée pour la β-réduction.
— la seconde fonctionne sur H : elle est utilisée pour l'unification.
Le même problème que celui cité précédemment apparaît encore : lors d'une substitution, des variables libres peuvent être capturées par des *binders* certaines d'entre elles peuvent être capturées, comme l'illustre l'exemple suivant :

$$\{X \mapsto x\}(\lambda x.X) = \lambda x.x$$

Nous devons donc de nouveau définir la notion de substitution avec renommage des variables liées :

Definition 2.1.4. Soit θ une évaluation (une fonction de H dans $\Lambda(V, H)$. La substitution $\bar{\theta}$ est l'extension de cette évaluation telle que :

1. $\bar{\theta}(X) = \theta(X)$
2. $\bar{\theta}(x) = x$ si $x \in V$
3. $\bar{\theta}(a\, b) = (\bar{\theta}(a)\, \bar{\theta}(b))$
4. $\bar{\theta}(\lambda y.a) = \lambda z.\bar{\theta}(\{y/z\}a)$ où z est une nouvelle variable.

$\bar{\theta}$ sera maintenant notée θ.

Proposition 2.1.1. *Les substitutions et les réductions commutent.*

2.1.3 λ-calcul avec la notation de de Bruijn

Dans le λ-calcul nommé vu précédemment, on nomme toujours les variables liées alors que ces noms ne sont en rien utiles à notre problème.Cette gestion du renommage des variables, qui peut parfois être très complexe, peut être évitée en utilisant une autre formulation du λ-calcul : la notation de de Bruijn. On en présente ici le principe (en ajoutant directement les métavariables).

Definition 2.1.5. L'ensemble $\Lambda_{DB}(H)$ des λ-termes dans la notation de de Bruijn est défini par :

$$a ::= n \mid X \mid \lambda\, a \mid (a\, a)$$

où n est un entier supérieur ou égal à 1 et $X \in H$.

On a donc remplacé ici les variables liées par des indices, mais on conserve les noms pour les métavariables. De plus, la notation de de Bruijn introduit aussi la notion de référentiel : les V-variables (i.e. les variables liées, mais pas les métavariables) sont ordonnées dans une liste $(x_0 \ldots x_n)$ qui définit le référentiel R.

Notre implémentation des lambda-sigma termes en de Brujin est similaire à la définition ci-dessus :

```
(* lambda terms *)
type term =
  | Var  of int
  | XVar of name
  | Abs  of ty * term
  | App  of term * term
```

Definition 2.1.6. Soit R un référentiel. Soit $a \in \Lambda(V, H)$ tel que toutes les V-variables libres de a sont déclarées dans R. La translation de de Bruijn de a, notée $tr(a, R)$ est définie par :

1. $tr(x, R) = j$, où j est l'indice de la première occurence de x dans R
2. $tr(X, R) = X$
3. $tr((a\,b), R) = (tr(a, R)\,tr(b, R))$
4. $tr(\lambda x.a, R) = \lambda(tr(a, x.R))$

Ainsi, pendant une translation de de Bruijn, le référentiel est incrémenté lorsque l'on rencontre un λ. Par exemple :

$$\lambda x\,\lambda y.(x(\lambda z.(z\,x))y)$$

s'écrit maintenant :

$$\lambda(\lambda(2(\lambda 1\,3)1))$$

Definition 2.1.7. La $\lambda - longueur$ d'une occurrence u dans un terme a est le nombre d'abstractions précédant u. Elle est notée $|u|$.

On doit maintenant définir la substitution qui correspond à la substitution des éléments de V.

Definition 2.1.8. Soit $a \in \Lambda_{DB}(H)$. Le terme a^+, appelé *lift* de a, est défini par $a^+ = lft(a, 0)$ où $lft(a, i)$ est défini par :

1. $lft((a_1\,a_2), i) = (lft(a_1, i)\,lft(a_2, i))$
2. $lft(\lambda a, i) = \lambda(lft(a, i + 1))$
3. $lft(X, i) = X$
4. $lft(m, i) = m + 1$ si $m > i$, m sinon.

On définit maintenant l'analogue de la V-substitution de $\Lambda(V, H)$.

Definition 2.1.9. La substitution par b à la λ-longueur $(n - 1)$ dans a, notée $\{n/b\}a$, est définie par :

$$\begin{aligned}
\{n/b\}(a_1\,a_2) &= (\{n/b\}a_1\,\{n/b\}a_2) \\
\{n/b\}X &= X \\
\{n/b\}\lambda a &= \lambda(\{n+1/b^+\}a) \\
\{n/b\}m &= m-1 \quad \text{si } m > n\ (m \in FV(a)) \\
&\qquad\ b \quad \text{si } m = n\ (m \text{ lié par le } \lambda \text{ du } \beta\text{-redex}) \\
&\qquad\ m \quad \text{si } m < n\ (m \in BV(a))
\end{aligned}$$

Implémentation Voici notre implémententation du lift et de la substitution :

```
let rec lift (t : term) (i : int) =
  match t with
  | Var(n) -> if n > i then Var(n+1) else t
  | XVar(_) -> t
  | Abs(ty, t) -> Abs(ty, lift t (i+1))
  | App(t1, t2) -> App((lift t1 i), (lift t2 i))

(* lift (up arrow) operation : increments free de brujin
   indices *)
let lift_plus (t : term) = lift t 0

(* substitution for beta reduction *)
let rec subst (a: term) (b: term) (n: int) =
  match a with
  | Var(m) -> if m > n then Var(n-1)
    else (if n = m then b else Var(m))
  | XVar(_) -> a
  | Abs(ty, t1) -> Abs(ty, subst t1 (lift_plus b) (n+1))
  | App(t1, t2) -> App((subst t1 b n), (subst t2 b n))
```

Definition 2.1.10. La β-réduction, dans ce contexte, est définie naturellement par :

$$((\lambda a)b) \to \{1/b\}a$$

où $\{1/b\}$ est la substitution de l'indice 1 par le terme b.

Par exemple, le terme $\lambda x.((\lambda y.(x\,y))x)$ s'écrit $\lambda((\lambda(2\,1))1)$ dans la notation de de Bruijn. Il se réduit en $\lambda x.(x\,x)$, ou, dans la notation de de Bruijn, en $\lambda(\{1/1\}(2\,1)) = \lambda(1\,1)$.

Definition 2.1.11. La η-réduction dans $\Lambda_{DB}(H)$ est définie par la règle :

$$\lambda(a\,1) \to b \quad \text{si } b \in \Lambda_{DB}(H) \text{ est tel que } a = b^+.$$

Proposition 2.1.2. *Pour un terme a de $\Lambda_{DB}(H)$, il existe un terme b tel que $a = b^+$ si et seulement si, pour toute occurrence u d'indice p dans a, $p \neq |u| + 1$.*

Enfin, nous définissons la H-substitution. θ^+ est défini par $\theta^+ = \{X_1/a_1^+, \ldots, X_n/a_n^+\}$ quand $\theta = \{X_1/a_1, \ldots, X_n/a_n\}$.

Definition 2.1.12. Soit θ une évaluation de H dans $\Lambda_{DB}(H)$. La substitution $\bar{\theta}$ associée est définie par les règles :

1. $\bar{\theta}(X) = \theta(X)$
2. $\bar{\theta}(n) = n$
3. $\bar{\theta}(a_1, a_2) = (\bar{\theta}(a_1)\bar{\theta}(a_2))$
4. $\bar{\theta}(\lambda a) = \lambda(\bar{\theta}^+(a))$

La relation entre la substitution dans le λ-calcul simple et la substitution dans le λ-calcul avec les indices de de Bruijn s'exprime selon la proposition suivante :

Proposition 2.1.3. *Soit $a \in \Lambda(V, H)$ et $\theta = \{X_1/a_1, \ldots, X_n/a_n\}$ un terme et une substitution d'un λ-calcul simple. Soit $\theta^r = \{X_1/tr(a_1, R), \ldots, X_n/tr(a_n, R)\}$. Alors $tr(\theta(a), R) = \theta^r(tr(a, R))$.*

2.1.4 Le $\lambda\sigma$-calcul

Le $\lambda\sigma$-calcul implémente le λ-calcul écrit avec la notation de de Bruijn, ainsi que les β et η réductions. Il est différent de part la manière dont ils traitent les substitutions. Essayons de comprendre l'origine du $\lambda\sigma$-calcul.

Soit F une fonction de A dans A. Par exemple : $F(0) = 0$, $F(S(0)) = S(S(0))$, $F(S(S(0))) = S(S(S(S(0))))$, etc. Une autre manière de définir cet opérateur va consister à introduire une nouvelle fonction intermédiaire f permettant de placer l'opération F en interne. Nous allons aussi introduire des règles permettant de réécrire les expressions contenant f en d'autres expressions ne contenant pas forcément f. Par exemple :

$$f(0) \to 0$$
$$f(S(X)) \to S(S(f(x)))$$

Dans le λ-calcul simple, les applications sont déjà placées en interne, mais les substitutions et le *lifting* se traitent encore de façon externe. Le $\lambda\sigma$-calcul, lui, place de façon interne les substitutions et le *lifting*.

Notations utiles. Les listes sont représentées par un opérateur *cons* (noté « . ») et un opérateur pour la liste vide (noté id). La substitution $a_1.a_2.\ldots.a_n.id$ remplace 1 par a_1, …, n par a_n, et décrémente par n tous les autres indices (libres) dans le terme. L'opérateur qui internalise le *lifting* est noté $\uparrow$, et peut être vu comme une substitution infinie $2.3.4.\ldots$. On introduit également un opérateur de composition $\circ$. Ainsi, l'indice $n+1$ s'écrit :

$$1[\uparrow \circ \cdots \circ \uparrow] = 1[\uparrow^n]$$

Les termes du $\lambda\sigma$-calcul sont alors construits de la manière suivante :

Definition 2.1.13. Soit X un ensemble de métavariables de termes et Y un ensemble de métabariables de substitution. L'ensemble $I_{\lambda\sigma}(H, Y)$ des termes et des substitutions explicites est défini par :

$$a ::= 1 \mid x \mid (a\ a) \mid \lambda a \mid a[s]$$
$$s ::= y \mid id \mid \uparrow \mid a.s \mid s \circ s$$

où $x \in X$ et $y \in Y$.

Proposition 2.1.4. *Chaque $\lambda\sigma$-terme dans sa forme normale pour le $\lambda\sigma$ est d'une des formes suivantes :*

1. *$\lambda\, a$ où a est une forme normale*

2. $(a\, b_1 \dots b_p)$ *où* a *et les* b_i *sont des formes normales, et* a *est soit* 1, $1[\uparrow^n]$, X, *ou* $X[s]$ *où* s *est un terme de substitution dans sa forme normale différent de id*
3. $a_1 \dots a_p \cdot \uparrow^n$ *où* $a_1, \dots, a_p$ *sont dans leurs formes normales et* $a_p \neq n$

Proposition 2.1.5. *Le H-grafting et la λσ-réduction commutent.*

Implémentation Pour notre implémentation, nous n'avons pas eu besoin de métavariables de substitution. L'implémentation des $\lambda\sigma$-terme est :

```
(* lambda sigma substitutions and terms *)
type s_subst =
  | Id
  | Shift
  | Cons of s_term * s_subst
  | Comp of s_subst * s_subst
and s_term =
  | S_One
  | S_Xvar of name
  | S_App of s_term * s_term
  | S_Abs of ty * s_term
  | S_Tsub of s_term * s_subst
```

2.2 Forme normale

Comme on a pu le voir précédemment, les λ-expressions impliquent un certain nombre de réécritures via les β-réductions. Ainsi, on peut définir la forme normale d'un λ-terme comme étant le terme réduit sur lequel on ne peut plus appliquer de réductions. Obtenir une telle forme est cruciale puisque l'algorithme d'unification que nous allons tenter d'implémenter travaille sur ces termes. Nous allons voir ici 2 notions de forme normale pour les λ-termes : *β-normal form* et *η-long normal form.*

2.2.1 β-normal form

Definition 2.2.1. Soit a un λ-terme correctement typé et de la forme *β-normal*, alors a est de la forme :

$$\lambda x_1 \dots \lambda x_n . h . e_1 \dots e_p$$

avec $n, p \geqslant 0$, h une constante, une variable liée ou une méta-variable, appelée *tête* de a et $e_1 \dots e_p$ sont des λ-termes β-normalisés appelés *arguments* de h.

Il existe plusieurs qualifications aux β normal forms (que l'on abrègera à partir de maintenant en *bnfs*) que nous allons maintenant définir.

Definition 2.2.2. Un λ-terme *bnf* est dit *rigid* si sa tête est une constante ou une variable liée. Sinon, on dit qu'il est *flexible*, sa tête est alors une métavariable.

Pour obtenir la β-normal form, il faut appliquer la règle de réduction β jusqu'a ce que l'on ne puisse plus réduire. cela fonctionne dans le λ-calcul car il est confluant avec la β-réduction, ie. il n'existe qu'une seule forme normale.

Dans le cadre du $\lambda\sigma$-calcul des règles de réduction supplémentaires sont nécessaires, les règles réductions σ travaillent sur les substitution explicites. Cependant, avec l'introduction des méta-variables, la règle β et les règles σ ne suffisent pas pour assurer la confluence du $\lambda\sigma$-calcul. L'utilisation de la règle η accompagné d'un système de types simples permet de rendre le $\lambda\sigma$-calcul confluent.

Les règles de réductions du $\lambda\sigma$-calcul sont :

Beta :
$$(\lambda a)b \to a[b.id]$$

App :
$$(a\,b)[s] \to (a[s]\,b[s])$$

VarCons :
$$1[a.s] \to a$$

Id :
$$a[id] \to a$$

Abs :
$$(\lambda a)[s] \to \lambda(a[1.(s\circ\uparrow)])$$

Clos :
$$(a[s])[t] \to a[s \circ t]$$

IdL :
$$id \circ s \to s$$

ShiftCons :
$$\uparrow \circ(a.s) \to s$$

AssEnv :
$$(s_1 \circ s_2) \circ s_3 \to s_1 \circ (s_2 \circ s_3)$$

MapEnv :
$$(a.s) \circ t \to a[t].(s \circ t)$$

IdR :
$$s \circ id \to s$$

VarShift :
$$1.\uparrow \to id$$

Scons :
$$1[s].(\uparrow \circ s) \to s$$

Eta :
$$\lambda(a\,1) \to b \text{ si } a =_\sigma b[\uparrow]$$

Implémentation L'implémentation de ces règles est :

```
let beta_red_s (t: s_term) =
  match t with
  | S_App (S_Abs (ty, a), b) -> S_Tsub (a, Cons (b, Id))
  | _ -> t

let eta_red_s (t: s_term) =
  match t with
  | S_Abs (ty, S_App(a, S_One)) -> unshift_s a
  | _ -> t

(* All reduction rules for terms *)
let reduce_term_s (t : s_term) =
  match t with
  | S_Tsub (S_App (a,b), s) -> S_App (S_Tsub (a,s),
     S_Tsub (b,s))
  | S_Tsub (S_One, Cons (a,b)) -> a
  | S_Tsub (a, Id) -> a
  | S_Tsub (S_Abs (ty, a), s) -> S_Abs (ty, S_Tsub (a,
     Cons (S_One, Comp (s, Shift))))
  | S_Tsub (S_Tsub (a,s),t) -> S_Tsub (a, Comp (s,t))
  | _ -> t

(* All reduction rules for subst *)
let reduce_subst_s (s : s_subst) =
  match s with
  | Comp (Id, s1) -> s1
  | Comp (Shift, Cons (a,s1)) -> s1
  | Comp (Comp (s1, s2), s3) -> Comp (s1, Comp (s2,s3))
  | Comp (Cons (a,s1),t) -> Cons (S_Tsub (a,t), Comp (s1,
     t))
  | Comp (s1, Id) -> s1
  | Cons (S_One, Shift) -> Id
  | Cons (S_Tsub (S_One, s1), Comp (Shift, s2)) -> if s1
     = s2 then s1 else s
  | _ -> s
```

Proposition 2.2.1. *Les propriétés du $\lambda\sigma$-calcul (sans types et avec la règle η) sont :*

1. *Le $\lambda\sigma$-calcul est localement confluent sur n'importe quel $\lambda\sigma$-terme, ouvert ou fermé.*
2. *Le $\lambda\sigma$-calcul est confluent sur les termes sans variables de substitutions*
3. *Le $\lambda\sigma$-calcul n'est pas confluent sur les termes avec des variables de substitutions.*

2.2.2 η-long normal form

Dans cette partie nous allons définir la notion de η-long normal form utilisée par Dowek pour tenter de résoudre le problème de l'unification dans [2].

Definition 2.2.3. Soit a un $\lambda\sigma$-terme de type $A_1 \to \ldots \to A_n \to B$ dans le contexte Γ et de la forme $\lambda\sigma$-normal. La η-long normal form de a, notée a', est défini par :

1. Si $a = \lambda_C b'$ alors $a' = \lambda_C b'$.
2. Si $a = (\mathtt{k} b_1 \ldots b_p)$ alors $a' = \lambda_{A_1} \ldots \lambda_{A_n}(k + n\ c_1 \ldots c_p\ \mathtt{n'} \ldots \mathtt{1'})$ avec c_i la η-long normal form de la forme normale de $b_i[\uparrow^n]$.
3. Si $a = (X[s] b_1 \ldots b_p$ alors $a' = \lambda_{A_1} \ldots \lambda_{A_n}(X[s']\ c_1 \ldots c_p\ \mathtt{n'} \ldots \mathtt{1'})$ avec c_i la η-long normal form de la forme normale de $b_i[\uparrow^n]$ et si $s = d_1 \ldots d_q.\ \uparrow^k$ alors $s' = e_1 \ldots e_q.\ \uparrow^{k+n}$ où e_i est η-long form de $d_i[\uparrow^n]$.

Definition 2.2.4. La *long normal form* d'un terme est la *η-long form* de sa $\beta\eta$-normal form.

Proposition 2.2.2. *Deux termes sont $\beta\eta$-équivalents si et seulement si ils ont la même long normal form.*

Implémentation Voici l'implémentation de la η-long normal forme que nous utilisons pour notre fonction d'unification (TODO METTRE LA REF).

```
let rec eta_long_normal_form (t : s_term) (typ : ty) (m_c
    : meta_var_str) : s_term =
  match t with
  | S_One | S_Xvar _ | S_Tsub (_,_) -> t
  | S_App (t1,t2) ->
        let (ty,rest) = get_before_last_type typ in
        let left = (match t1 with
            | S_One -> S_Tsub (S_One,(s_shift_n 2))
            | S_Tsub(S_One,s1) -> S_Tsub(S_One,Comp(Shift
                ,s1))
            | S_Tsub(S_Xvar n,s1) ->
                let meta_type = fst (Map_str.find n m_c)
                    in
                let s_prime =
                  apply_fun_subst s1 (fun t ->
                      eta_long_normal_form t meta_type m_c)
                    in S_Tsub(S_Xvar n,s_prime)
            | _ -> failwith "eta_long_normal_form can't
                happend you should be in normal form") in
        let right = eta_long_normal_form (
            normalise_lambda_sigma (S_Tsub(t2,(s_shift_n
            1)))) rest m_c in
                    S_Abs(ty,S_App(left,right))
  | S_Abs (ty1,t1) ->
    S_Abs(ty1,eta_long_normal_form t1 typ m_c)
```

La fonction est très proche de la définition cependant ont nécessite simplement une fonction auxiliaire afin de récupérer l'avant dernier type d'un type flèche : `get_before_last_type` . La seule différence avec la définition est que au lieu de considérer des applications n-aires on ne peut considérer qu'une application à la fois.

2.3 Typage

Afin de réaliser l'unification, les types sont nécessaires. TODO NOUS EN REPARLERONS PLUS TARD METTRE LE LIEN Les règles pour les termes sont les mêmes que celles du λ-calcul simplement typé. Voici les règles impliquant des substitutions :

Clos :

$$\frac{\Gamma \vdash s \rhd \Gamma' \qquad \Gamma' \vdash a : A}{\Gamma \vdash a[s] : A}$$

Id :

$$\Gamma \vdash id \rhd \Gamma$$

Shift :

$$A.\Gamma \vdash \uparrow \rhd \Gamma$$

Cons :

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash s \rhd \Gamma'}{\Gamma \vdash a.s \rhd A.\Gamma'}$$

Comp :

$$\frac{\Gamma \vdash s'' \rhd \Gamma'' \qquad \Gamma'' \vdash s' \rhd \Gamma'}{\Gamma \vdash s' \circ s'' \rhd \Gamma'}$$

Proposition 2.3.1. *Le $\lambda\sigma$-calcul typé est :*

- *confluent*
- *faiblement normalisable*

L'algorithme de typage que nous utilisons est celui du λ-calcul simplement typé avec inférence de type.

Implémentation Focalisons nous simplement sur l'implémentation des règles concernant les substitutions :

```
and type_check_cont c t_sub m_c =
  match t_sub with
  | Id              -> c
  | Shift           -> (match c with
      | []      -> raise No_inference
      | hd::tl -> tl)
  | Cons(t, s)    -> let c_s = type_check_cont c s m_c in
    let ty_t = type_check_inf c t m_c in
    ty_t::c_s
  | Comp(s1, s2) -> let c_s2 = type_check_cont c s2 m_c
     in
    type_check_cont c_s2 s1 m_c
```

3 Unification

Dans notre cas, nous nous intéressons à l'unification *higher order*, à l'opposition de l'unification *first order* étudiée par Robinson en 1965 [6][5]. Nous avons trouvé 2 implémentations possibles que nous avons essayé de mettre en place :

— Dowek
— Huet

3.1 Algorithme de Dowek

3.1.1 Precooking

En $\lambda\sigma$-calcul, les opérations grafting et de réduction commutent. C'est pour cette raison que l'on peut utiliser le grafting dans l'unification higher-order du $\lambda\sigma$-calcul. En effet l'unification dans le $\lambda\sigma$-calcul calcule un grafting qui rends les termes égaux, tandis que l'unification dans le lambda-calcul calcule une substitution. Pour réaliser l'unification higher-order, on souhaite travailler avec des graftings.

Pour réaliser l'unification dans le λ-calcul, on va traduire les termes du λ-calcul vers le $\lambda\sigma$-calcul, cette opération est nommée precooking :

Definition 3.1.1. Soit $a \in \Lambda_{DB}(X)$ tel que $\Gamma \vdash a : T$. Pour chaque variable X de type U dans le terme a, on associe le type U dans le contexte Γ dans le $\lambda\sigma$-calcul. Le precooking de a depuis le λ-calcul vers le $\lambda\sigma$-calcul est défini par $a_F = F(a, 0)$ où $F(a, n)$ est :

1. $F((\lambda_B a), n) = \lambda_B(F(a, n + 1))$
2. $F(k, n) = 1[\uparrow^{k-1}]$
3. $F((ab), n) = F(a, n)F(b, n)$
4. $F(X, n) = X[\uparrow^n]$

3.1.2 Algorithme d'unification

Nous allons présenter l'algorithme d'unification utilisé dans le papier de Dowek [2]. Comme dit précédemment, une solution est un grafting des métavariables vers les termes.

D'un point de vus formelle on cherche à appliquer des règles d'unification (présentées plus tard) sur un système d'équations. Nous avons dû faire un choix afin de représenter cela dans notre système. Voici les différentes structures que nous avons définies :

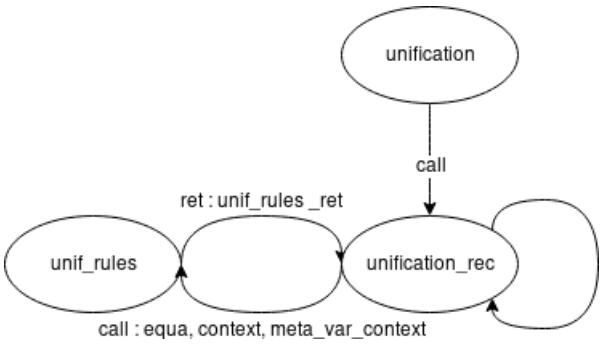
```
type equa =
  | DecEq of s_term * s_term
  | Exp

type and_list = equa list

type unif_rules_ret =
  | Ret of (and_list * meta_var_str) list
  | Rep of name * s_term * and_list
  | Nope
  | Fail
```

Le type `equa` permet de représenter les équations. Nous avons besoin d'un cas `exp` qui sera utilisé par notre fonction d'unification pour traiter le cas où il n'y a plus aucunes équations sur lesquelles appliquer des règles, nous y reviendrons plus tard.

Le type `and_list` nous permet de représenter les disjonctions du système d'équations. Et enfin le type `unif_rules_ret` est notre type de retour personnalisé pour l'algorithme d'unification. Avant d'expliquer ce type, il est important de parler de l'architecture des fonctions que nous avons définies pour l'unification.



Étant donné que certaines règles peuvent produire de nouveaux systèmes d'équations étant d'autres solutions potentielles le constructeur `Ret` retourne des listes de solutions. Le constructeur `Replace` permet de notifier que la règle doit être suivie d'un remplacement, le `Nope` notifie que l'on n'a pas trouvé de pattern dans l'équation permettant de trouver. `Fail` notifie que l'on a trouver une contradiction.

Présentons maintenant l'ensemble des règles d'unification (pour chaque règle on présente la partie formelle et l'implémentation) :

Dec-λ :

$$P \wedge \lambda_A a =^?_{\lambda\sigma} \lambda_A b$$
$$\rightarrow$$
$$P \wedge a =^?_{\lambda\sigma} b$$

Cette règle permet simplement de rentrer sous les λ si les deux termes de l'équation sont des abstractions.

```
| DecEq (S_Abs (typ1, t1), S_Abs (typ2, t2)) ->
    if eq_typ typ1 typ2 then Ret [([ DecEq (t1, t2)], ctx
        )] else Fail
```

Dec-App :

$$P \wedge (n\, a_1 \ldots a_p) =^?_{\lambda\sigma} (n\, b_1 \ldots b_p)$$
$$\rightarrow$$
$$P \wedge (\wedge_{i=1\ldots p} a_i =^?_{\lambda\sigma} b_i)$$

Cette règle crée de nouvelles équations entre chacun des arguments. Il est à noter qu'avec notre représentation nous ne possédons pas d'applications à plusieurs

arguments, nous n'avons donc pas à traiter ce cas général mais le sous-cas réduit à une seule application. De plus, cette règle peut trouver une contradiction si les deux fonctions ne sont pas égales. Ce qui est traduit par la règle suivante :

Dec-Fail :

$$P \wedge (n\, a_1 \dots a_p) =^?_{\lambda\sigma} (m\, b_1 \dots b_q)$$
$$\rightarrow$$
$$F$$
si $n \neq m$

Ces deux cas sont gérer par notre implémentation dans le traitement du même pattern.

```
| DecEq (S_App (t1, t2), S_App (t3, t4)) ->
  if t1 = t3 then Ret [([DecEq (t2, t4)], ctx)]
    else Fail
```

Replace :

$$P \wedge X =^?_{\lambda\sigma} a$$
$$\rightarrow$$
$$\{X \mapsto a\}(P) \wedge X =^?_{\lambda\sigma} a$$
si $X \in TVar(P)$, $X \notin TVar(a)$ et a une métavariable $\implies a \in TVar(P)$

Le replace est une règle particulière car elle se contente simplement de remplacer une variable d'unification si celle-ci ne comporte pas de substitution. Étant donné notre implémentation, nous ne pouvons pas effectuer ce traitement dans la fonction appliquant les règles directement. C'est pour cette raison que nous avons rajouté un constructeur `Repl` dans le type `unif_rules_ret`. Grâce à cela, notre fonction principale sera à même d'effectuer ce remplacement.

```
(* REP *)
  | DecEq (S_Xvar (n), t) -> Rep (n, t, [e])
```

Normalize :

$$P \wedge a =^?_{\lambda\sigma} b$$
$$\rightarrow$$
$$P \wedge a' =^?_{\lambda\sigma} b'$$
si a ou b n'est pas une forme normale longue

a' (resp. b') est la forme normale longue de a (resp. b) si a (resp. b) n'est pas une variable résolue et a (resp

De même que pour replace cette règle est un peu spéciale puisque étant donné que l'on souhaite travailler uniquement avec des termes normalisés nous nous assurons au cours de l'algorithme d'appeler la fonction de normalisation aux différents endroits où celle ci est nécessaire.

Exp-App :

$P \wedge X[a_1 \dots a_p.\ \uparrow^n] =^?_{\lambda\sigma} (m\, b_1 \dots b_q)$

$\rightarrow$

$P \wedge X[a_1 \dots a_p.\ \uparrow^n] =^?_{\lambda\sigma} (m\, b_1 \dots b_q)$

$\quad \wedge \vee_{r \in R_p \cup R_i} \exists H_1, \dots, H_k, X =^?_{\lambda\sigma} (r\, H_1 \dots H_k)$

si X est un type atomique et n'est pas résolu

où $H_1, \dots, H_k$ sont des variables de types appropriés, qui n'apparaissent pas dans P, avec les contextes Γ_{H_i}

Cette règle permet de supprimer les substitutions associées aux variables d'unification. Malgré son apparente complexité, le but de cette règle est de créer un ensemble de nouvelles solutions (ce qui va entraîner des appels récursifs à notre fonction d'unification).

```
(* EXP APP *)
  | DecEq (S_Tsub (S_Xvar(x), s), t) ->
    let (typ, _) = Map_str.find x ctx in
    let lst = find_var_end_typ ct typ in
    create_disjunctions (S_Xvar (x)) ct ctx lst
```

Exp-λ :

P

$\rightarrow$

$\exists Y : (A.\Gamma \vdash B), \quad P \wedge X =^?_{\lambda\sigma} \lambda_A Y$

si $(X : \Gamma \vdash A \rightarrow B) \in TVar(P), Y \notin TVar(P)$, et X n'est pas une variable résolue

Cette règle a pour but d'essayer de continuer l'unification lorsque plus aucune des règles précédentes ne sont applicables. C'est pour cette raison que nous avons rajouter le constructeur `Exp` comme type pour les équations. Cela va donc forcer notre fonction à exécuter cette règle. Celle-ci consiste à utiliser la *η-expansion* sur une variable du contexte qui n'est pas résolue. Il faut également que cette variable ait un type `arrow` (sinon il est impossible d'appliquer la *η-expansion*).

Nous avons donc présenté l'ensemble des règles nécessaires à l'unification, voici maintenant la fonction `unification_rec` :

```
unification_rec (s: and_list) (su : (and_list *
    unif_rules_ret list)) (ctx : meta_var_str) (ct :
    context) : ((and_list * meta_var_str) list) option =
  if is_that_finished ctx then Some [(s,ctx)]
  else
    let (old_liste, ret_liste) = su in
    (match s with
     | [] -> (
        match look_res_list ret_liste with
        | FullNope -> (
          match unif_rules Exp ct ctx with
          | Ret l -> start_unification_list l ct su
          | Rep (res_name,res_term,res_s) ->
              unification_rec (replace_and_list res_name
              res_term (old_liste @ res_s)) ([],[]) (
              put_metaVar_true res_name ctx) ct
          | Nope -> None
          | Fail -> None)
        | OneRet -> unification_rec (fst su) ([],[]) ctx
            ct
        | Failed -> None)
    | a :: tl ->
      let (old_liste, ret_liste) = su in
      let ret = unif_rules a ct ctx in
      let new_su = (old_liste @ [a] ,ret_liste @ [ret])
          in
      (match ret with  (* unification_rec tl new_su *)
        | Ret l -> start_unification_list l ct new_su
        | Rep (res_name,res_term,res_s) ->
            unification_rec (replace_and_list res_name
            res_term (old_liste @ res_s)) ([],[]) (
            put_metaVar_true res_name ctx) ct
        | Nope -> None
        | Fail -> None))
```

3.2 Algorithme de Huet

L'algorithme de Huet prend en entrée une équation ou un système d'équation représentant des problèmes d'unification. Les membres de chaque équation doivent avoir été mises en η-long normal form.

3.2.1 Quelques définitions préliminaires

On appelle tête d'un terme en η-long normal form la première variable sous les n lambda qui commencent ce terme. Cette tête peut donc être un indice de Debruijn (si la variable en question est une variable lié ou libre) ou une métavariable.

On dit d'une équation d'unfication qu'elle est rigide-rigide si les têtes des deux termes de l'équation sont des indices de Debruijn, flexible-rigide si l'une

des têtes et une métavariable et l'autre un indice de Debruijn ou flexible-flexible si les deux têtes sont des métavariables.

Nous avons donc besoin de quelques primitives pour l'implémentation.

```
type terminal =
| Success
| Failure

type equationsys =
| Eq of s_term * s_term
| SysEq of equationsys * equationsys
| NotAnEq

type simplresult =
| Term of terminal
| Sys of equationsys

let rec extract_head ( t1 : s_term ) : s_term =
  match t1 with
  | S_Abs (ty, t) -> extract_head (t)
  | S_App (t2, t3) -> t2
  | _ -> t1

let rec is_rigid_rigid ( t1 : s_term ) ( t2 : s_term ) :
    bool =
  if (is_number(extract_head (t1))&&is_number(
     extract_head (t2))) then true else false

let rec is_flexible_rigid ( t1 : s_term ) ( t2 : s_term )
    : bool =
  if ((!is_number(extract_head (t1))&&is_number(
     extract_head (t2)))||(!is_number(extract_head (t1))
     &&is_number(extract_head (t2)))) then true else
     false

let rec is_flexible_flexible ( t1 : s_term ) ( t2 :
   s_term ) : bool =
  if (!is_number(extract_head (t1))&&!is_number(
     extract_head (t2))) then true else false
```

3.2.2 Principe de l'algorithme de Huet

Ces concepts sont importants pour l'algorithme de Huet. Celui-ci s'arrête en indiquant un succés d'unification si toutes les équations du système sont de forme flexible-flexible, car cela signifie que les têtes des termes sont toutes des métavariables et qu'il sera alors trivial de trouver une substitution qui permettra d'unifier ces équations.

3.2.3 Procédure SIMPL

Ainsi, l'algorithme de Huet commence par détecter les équations rigide-rigide et tente d'appliquer la procédure `SIMPL`.

La procédure `SIMPL` va sélectionner une équation rigide-rigide dans le système. Si les têtes de ces deux termes sont des indices de Debruijn différents, alors `SIMPL` retourne un rapport d'échec, le système n'est pas unifiable. Sinon, on va remplacer cette équation par un ensemble de k équations d'unification. La i-ème équation ainsi créée consistera en un problème d'unification entre les i-ème termes succédant les têtes des deux termes, sur lesquels on aura pris soin de remettre les n lambdas des termes originaux.

Cette procédure `SIMPL` va boucler jusqu'à ce qu'il n'existe plus aucune équation rigide-rigide dans le système.

A l'issue de la procédure `SIMPL`, s'il n'existe que des équations flexible-flexible dans le système, le problème est considéré unifiable. Sinon, s'il reste des équations flexible-rigide, on leur applique la procédure `MATCH`, qui aura pour but de trouver une substitution de la méta-variable en tête de l'un des termes en un nouveau terme.

Implémentation :

```
let rec get_new_eq (s: equationsys)(t:s_term) :
    equationsys = match s with
  | NotAnEq -> NotAnEq
  | Eq (t1, t2) -> match t1 with
    | S_Abs (ty1, t3) -> match t2 with
      | S_Abs (ty2, t4) -> apply_lambda(get_new_eq (Eq(t3
          ,t4), t)  ,ty)
      | _ -> Eq (t3,t4)
    | S_App (t3, t4) -> match t2 with
      | S_App (t5, t6) -> if t3 == t then get_new_eq (Eq
          (t4, t6), t) else SysEq(Eq(t3,t5), get_new_eq (
          Eq(t4,t6)))
      | _-> Eq(t1,t2)
  | _ -> Eq(t1,t2)

  let rec huet_simpl ( s: equationsys ) : simplresult =
  if  contains_rigid_rigid (s) then
    match get_rigid_rigid (s) with
    | Eq (t1, t2) -> if get_number (extract_head(t1)) !=
        get_number (extract_head(t2)) then Failure else
      huet_simpl(Sys_Eq(delete_from_sys(Eq(t1,t2), s),
          get_new_eq(Eq(t1,t2), extract_head(t1))))
    | _ -> Failure
  else
    if contains_flexible_rigid (s) then s else Success
```

3.2.4 Procédure MATCH

La procédure `MATCH` va essayer d'appliquer de manière non-déterministe deux procédures différentes, la procédure d'imitation et la procédure de projection.

3.2.5 Procédure d'imitation

La procédure d'imitation consiste à remplacer dans le terme flexible toutes les occurrences de la tête de ce terme (qui se trouve donc être une métavariable) par un terme en *η-long normal form* constitué de plusieurs lambda correspondant au type de la métavariable (par exemple si la métavariable est de type $B_1 \to B_2 \to \ldots \to B_p$ ont aura p lambdas respectivement de type $B_1 B_2 \ldots B_p$), suivis d'un indice de Debruijn (égal au nombre de termes suivant initialement la métavariable dans le terme flexible de l'équation + l'indice de Debruijn qui est la tête du terme rigide de l'équation - le nombre de lambda qui précèdent initialement la tête du terme), suivis de termes (un nombre de terme égal au nombre de termes succédant la tête du terme rigide de l'équation dans le dit terme) qui sont constitués d'applications d'une nouvelle métavariable à une application à un indice de Debruijn égal au nombre de termes après la tête dans le terme rigide de l'équation initiale appliquée à une application d'un indice de Debruijn égal à ce nombre de termes - 1 ... et ainsi de suite jusqu'à 1. On a ainsi introduit plusieurs nouvelles méta-variables de même type que la méta-variable initiale qu'on va tenter d'unifier par la suite.

Implémentation partielle :

```
let rec get_new_eq (s: equationsys)(t:s_term) :
    equationsys = match s with
  | NotAnEq -> NotAnEq
  | Eq (t1, t2) -> match t1 with
    | S_Abs (ty1, t3) -> match t2 with
      | S_Abs (ty2, t4) -> apply_lambda(get_new_eq (Eq(t3
          ,t4), t)  ,ty)
      | _ -> Eq (t3,t4)
    | S_App (t3, t4) -> match t2 with
      | S_App (t5, t6) -> if t3 == t then get_new_eq (Eq
          (t4, t6), t) else SysEq(Eq(t3,t5), get_new_eq (
          Eq(t4,t6)))
      | _-> Eq(t1,t2)
  | _ -> Eq(t1,t2)

  let rec huet_simpl ( s: equationsys ) : simplresult =
  if  contains_rigid_rigid (s) then
    match get_rigid_rigid (s) with
    | Eq (t1, t2) -> if get_number (extract_head(t1)) !=
        get_number (extract_head(t2)) then Failure else
      huet_simpl(Sys_Eq(delete_from_sys(Eq(t1,t2), s),
          get_new_eq(Eq(t1,t2), extract_head(t1))))
    | _ -> Failure
  else
    if contains_flexible_rigid (s) then s else Success
```

3.2.6 Procédure de projection

La procédure de projection va créer plusieurs substitutions comme suit : pour le i-ème terme suivant la tête du terme flexible, si ce terme a un type ayant pour cible le même type que la métavariable en tête du terme, on créait une substitution de la méta-variable par un terme composé d'un nombre de lambda égal au nombre de termes suivant la tête dans le terme flexible initial, d'un indice de Debruijn égal à ce nombre de termes $-i+1$, d'un nombre de termes correspondant au type du i-ème terme choisi (par exemple s'il est de type $D_1 \rightarrow D_2 \rightarrow \ldots \rightarrow D_q$, on créait q termes) chacun composé d'une métavariable(telle que la j-ème métavariable ainsi créée soit de type $B_1 \rightarrow \ldots \rightarrow B_{p_1} \rightarrow D_j$ où $B_1 \rightarrow \ldots \rightarrow B_{p_1} \rightarrow A$ étant le type de la métavariable tête de terme initial), suivi d'indices de Debruijn décroissants du nombre de termes succédant à la tête de terme initial jusqu'à 1. Pour chaque terme valide, on créait donc un nouveau problème d'unification. L'unification est un succès (respectivement un échec) si tous les problèmes ainsi générés aboutissent à un succès (respectivement à un échec)

3.2.7 Conclusion sur l'alorithme d'Huet

La procédure `MATCH` va donc introduire de nouvelles méta-variables et la procédure `SIMPL` va supprimer des indices de Debruijn, le but étant de n'obtenir des équations qu'entre des termes dont la tête est une métavariable. Cependant, il est possible que l'algorithme tourne en boucle, générant à chaque fois de nouvelles métavariables qui n'amèneront jamais à des équations flexible-flexible. Il est donc possible que l'algorithme de Huet ne se termine pas. Dans le cas contraire, il indiquera une unification possible ou une impossibilité d'unification.

4 Unification *higher order* par l'exemple

Nous allons, dans cette partie, résoudre un problème simple d'unification pour illustrer le fonctionnement de l'algorithme.

Soit le problème suivant, encodé en de Bruijn, que l'on va essayer de résoudre :

$$\lambda_A y.(Xa) =^?_{\beta\eta} \lambda_A y.a$$

avec

$$a : X$$

$$X : A \to A$$

Cette équation est encodé en de Bruijn en utilisant le contexte suivant :

$$\Gamma = A.nil$$

On obtient alors :

$$\lambda(X\mathtt{2}) =^?_{\beta\eta} \lambda\mathtt{2}$$

On obtient ensuite après l'étape de ***precooking*** :

$$\lambda(X[\uparrow]\mathtt{2}) =^?_{\lambda\sigma} \lambda\mathtt{2}$$

On applique la règle **Exp-λ** :

$$\exists Y((X[\uparrow]\mathtt{2}) =^?_{\lambda\sigma} \mathtt{2}) \wedge (X =^?_{\lambda\sigma} \lambda Y)$$

avec $\Gamma_Y = A.\Gamma$ et $T_Y = A$. En appliquant la règle **Replace** on a :

$$\exists Y(((\lambda Y)[\uparrow]\mathtt{2}) =^?_{\lambda\sigma} \mathtt{2}) \wedge (X =^?_{\lambda\sigma} \lambda Y)$$

Avec **Normalize** on a :

$$\exists Y(Y[\mathtt{2}.\uparrow] =^?_{\lambda\sigma} \mathtt{2} \wedge (X =^?_{\lambda\sigma} \lambda Y)$$

En appliquant la règle **Exp-App** on a :

$$\exists Y(Y[\mathtt{2}.\uparrow] =^?_{\lambda\sigma} \mathtt{2}) \wedge (X =^?_{\lambda\sigma} \lambda Y) \wedge (Y = \mathtt{1})$$

$$\vee$$

$$\exists Y(Y[\mathtt{2}.\uparrow] =^?_{\lambda\sigma} \mathtt{2}) \wedge (X =^?_{\lambda\sigma} \lambda Y) \wedge (Y = \mathtt{2})$$

On réduit ensuite avec la règle **Replace**

$$(\mathtt{1}[\mathtt{2}.\uparrow] =^?_{\lambda\sigma} \mathtt{2}) \wedge (X =^?_{\lambda\sigma} \lambda\mathtt{1}) \vee (\mathtt{2}[\mathtt{2}.\uparrow] =^?_{\lambda\sigma} \mathtt{2}) \wedge (X =^?_{\lambda\sigma} \lambda\mathtt{2})$$

Enfin, en normalisant avec la règle **Normalize**, on a :

$$(X =^?_{\lambda\sigma} \lambda\mathtt{1}) \vee (X =^?_{\lambda\sigma} \lambda\mathtt{2})$$

On a donc 2 solutions possibles au problème posé initialement.

5 Conclusion

Au terme de ce projet de Groupe de Recherche en Algorithmique, nous avons, dans une certaine mesure, réussi à proposer une implémentation du *higher order unification*. Comme l'a voulu la "philosophie" de cet enseignement, le point de départ de ce projet fut la lecture d'articles de recherche. La toute première difficulté a donc été de sélectionner les articles les plus pertinents par rapport à notre objectif mais aussi les plus clairs et réalisables de notre point de vue.

La seconde difficulté a ensuite été la lecture elle-même des articles, autrement dit, rechercher des solutions à un problème relativement compliqué et nouveau pour nous en tant qu'étudiant par la compréhension de ces derniers. Ainsi, le travail de groupe a été primordial durant cette phase de préparation, permettant aux personnes les plus à l'aise avec les notions qui sont abordées de les expliquer plus simplement au reste du groupe.

En ce sens, au-delà des résultats obtenus lors de ce travail de groupe, on retiendra surtout l'aspect recherche et exploration de résultats théoriques difficiles à appréhender à partir de la manière dont ils sont présentés dans les textes. L'un de nos objectifs à travers ce rapport a donc été de vulgariser et de présenter ces résultats du mieux que l'on peut et de manière didactique.

Références

[1] Flávio LC de Moura, Mauricio Ayala-Rincón, and Fairouz Kamareddine. Higher-order unification : A structural relation between huet's method and the one based on explicit substitutions. *Journal of Applied Logic*, 6(1) :72–108, 2008.

[2] Gilles Dowek, Thérese Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In *Logic in Computer Science, 1995. LICS'95. Proceedings., Tenth Annual IEEE Symposium on*, pages 366–374. IEEE, 1995.

[3] Gilles Dowek, Thérese Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions : The case of higher-order patterns. In *JICSLP*, pages 259–273, 1996.

[4] Gerard P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1) :27–57, 1975.

[5] J Alan Robinson. Computational logic : The unification computation. *Machine intelligence*, 6(63-72) :10–1, 1971.

[6] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1) :23–41, 1965.