Rapport de stage Mim2
–
Construction de programmes fonctionnels à partir d'exemples par unification

Guillaume Burel

19 janvier – 26 mars 2004

Stage encadré par
Tobias Nipkow
Martin Strecker

Technische Universität München
Fakultät für Informatik
Lehrstuhl IV : Software & Systems Engineering
Boltzmann Str. 3
85748 Garching bei München

Table des matières

Introduction

Il est parfois fastidieux pour un programmeur de devoir réécrire des petits programmes, au cas où il n'a pas accès à une bibliothèque les contenant. Par exemple, écrire un programme fonctionnel pour la soustraction sur les entiers naturels peut sembler simple à première vue, mais la première idée qui vient `sub (Suc m) n = Suc (sub m n)` n'est pourtant pas correcte. Alors, il serait intéressant d'avoir un outil qui permette de générer automatiquement de telles fonctions, le programmeur n'ayant plus qu'à fournir quelques exemples, et à vérifier que le programme qui lui est retourné correspond bien à son attente, et pas seulement aux exemples fournis. Par exemple, pour obtenir la fonction retournant la longueur d'une liste, il aurait juste à donner comme exemples $?length[] = 0$, $?length[1;2] = 2$ et $?length[4;3;5] = 3$.

Ce problème de programmation à partir d'exemples a été essentiellement étudié à la fin des années 70 et au début des années 80, par entre autre Summers et Biermann. Néanmoins, une méthode applicable dans la pratique n'a pas véritablement vu le jour. Ces travaux ont été repris plus récemment par Hagiya ([Hag90, Hag91]), qui a eu l'idée d'utiliser l'unification d'ordre supérieur pour résoudre ces problèmes. Malgré tout, son approche présente encore quelques inconvénients : d'une part la méthode utilisée dans les années 80, l'*inférence inductive*, n'est pas la plus simple pour transformer le problème initial avec les exemples en problème d'unification, d'autre part, s'il existe des algorithmes pour résoudre ce type d'unification d'ordre supérieur, ceux-ci sont prévus pour un cadre trop général et sont par la même inefficient pour nos problèmes particuliers.

Mon stage a consisté à trouver une méthode qui permette de trouver de telles fonctions de manière la plus efficiente possible. Dans la section 1, j'indiquerai dans quel contexte je me place, et je poserai quelques définitions utiles par la suite. Dans la section 2, je rappellerai l'algorithme présenté par Huet [Hue75] pour résoudre les problèmes d'unification d'ordre supérieur, puis je montrerai dans quelle mesure le fait de considérer la programmation à partir d'exemples comme un problème particulier d'unification est bien adapté, avant de proposer une extension de l'algorithme de Huet qui permet de résoudre ces problèmes. Enfin, dans la section 3, je démontrerai quelques propriétés de l'algorithme et j'indiquerai certains résultats obtenus par l'implémentation.

1 Termes, types

1.1 Termes

J'utilise par la suite une variante personnelle du λ-calcul avec notation de De Bruijn (voir [dB72, Nip93]). Les λ-termes avec notation de De Bruijn sont générés par la grammaire suivante[1] :

$$T = i \mid \Lambda T \mid T_1 T_2$$

où les variables i ont valeur dans l'ensemble des entiers naturels. Les notations de De Bruijn sont utilisées pour éviter les conflits entre noms de variables sans avoir

[1] j'utilise le symbole Λ à la place de λ de façon à différencier les deux notations par la suite, bien qu'on utilise le second normalement

recours aux α-conversions. La valeur d'une variable i représente la profondeur d'abstraction de cette variable dans le terme, c'est-à-dire elle correspond à la variable introduite par la $i$ème abstraction autour de celle-ci.

Exemple 1.1*:* Le terme représenté en notation classique par $\lambda x.x$ s'écrit avec les notations de De Bruijn $\Lambda 0$, de même $\lambda xy.x(yx)$ s'écrit $\Lambda\Lambda 1(0\,1)$.

Les variables libres doivent également être représentées par des entiers, on doit donc décider d'une énumération de celle-ci. La suite de β réduction suivante

$$(\lambda xy.y(x\,y))\lambda x.a \rightsquigarrow_\beta \lambda y.y((\lambda x.a)y) \rightsquigarrow_\beta \lambda y.y\,a$$

devient alors

$$(\Lambda\Lambda 0(1\,0))\Lambda 1 \rightsquigarrow_\beta \Lambda 0((\Lambda 2)0) \rightsquigarrow_\beta \Lambda 0\,1$$

On voit ici la variable libre a changer de représentant dans la notation de De Bruijn, passant de la valeur 1 à 2 puis revenant à 1. Une même variable libre peut même posséder des représentants différents dans le même terme, comme dans $(\lambda x.a)a$ qui s'écrit alors $(\Lambda 1)0$.

Ce renommage des variables libres n'est pas très intuitif au niveau du calcul, puisque les variables libres peuvent changer de valeur, et une variable libre peut même avoir deux représentants différents dans un même terme. Dans la suite, nous voulons utiliser des constructeurs qu'on aimerait pouvoir repérer facilement au sein d'un terme. C'est pourquoi nous introduisons un ensemble de constantes $\mathcal{C}onst$. De même, nous allons travailler avec des problèmes d'unification, nous avons donc besoin d'un ensemble infini dénombrable d'inconnues $\mathcal{U}nk$. Enfin, nous ajoutons aussi un ensemble de *constantes récursives* dont le rôle est expliqué dans la section suivante.

Les termes que nous considérons par la suite sont donc générés par la grammaire suivante :

$$T = i \mid \mathcal{C}onst \mid ?\mathcal{U}nk \mid \mathcal{R}ec\_\mathcal{C}onst \mid \Lambda T \mid T_1 T_2$$

Nous ne conservons que les termes bien typés, les types étant ceux du λ-calcul classique.

1.2 Types

Nous considérons les même types récursifs que dans Isabelle/HOL (`datatype`, voir [Ber98][2]), où on peut définir le type récursif t par :

$$\text{datatype } (\alpha_1, \dots, \alpha_k)t = C_1\,\tau_{1,1} \dots \tau_{1,n_1} \mid \dots \mid C_c\,\tau_{c,1} \dots \tau_{c,n_c}$$

où les types $\tau_{i,j}$ sont soit des types non-récursifs formés à partir des variables de type $\alpha_1, \dots, \alpha_k$, soit le type $(\alpha_1, \dots, \alpha_k)t$ lui-même.

Pour chacun de ces types, il existe un combinateur :

$$\begin{array}{ll} t\_\text{rec} :: & (\tau_{1,1} \to \dots \to \tau_{1,n_1} \to \beta \to \dots \to \beta) \to \\ & \dots \to \\ & (\tau_{c,1} \to \dots \to \tau_{c,n_c} \to \beta \to \dots \to \beta) \to \\ & (\alpha_1, \dots, \alpha_k)t \to \\ & \beta \end{array}$$

[2]en fait, on utilise une version légèrement simplifiée, puisqu'on ne traite pas les cas de récursivité mutuelle, auquel on peut toutefois parvenir en généralisant

auquel sont associés les règles de réecriture suivantes :

$$t\_\mathrm{rec}\, f_1 \dots f_c\, (C_i\, a_1 \dots a_{n_i}) = f_i\, a_1 \dots a_{n_i}\, (t\_\mathrm{rec}\, f_1 \dots f_c\, a_{l_1}) \dots (t\_\mathrm{rec}\, f_1 \dots f_c\, a_{l_{k_i}})$$

où les l_j sont tels que $\tau_{i,j}$ soit égal à $(\alpha_1, \dots, \alpha_k)t$.

Exemple 1.2*:* Les entiers naturels sont définis ainsi :

$$\text{datatype } nat = Zero \mid Suc\, nat \quad ;$$

le combinateur associé est la fonction nat_rec définie par :

$$\begin{array}{lcl} nat\_\mathrm{rec}\, f\, g\, Zero & = & f \\ nat\_\mathrm{rec}\, f\, g\, (Suc\, n) & = & g\, n\, (nat\_\mathrm{rec}\, f\, g\, n) \end{array}$$

Les listes sont définies par :

$$\text{datatype } (\alpha)list = Nil \mid Cons\, \alpha\, (\alpha)list \quad ;$$

le combinateur associé est la fonction $list\_\mathrm{rec}$ définie par :

$$\begin{array}{lcl} list\_\mathrm{rec}\, f\, g\, Nil & = & f \\ list\_\mathrm{rec}\, f\, g\, (Cons\, x\, xs) & = & g\, x\, xs\, (list\_\mathrm{rec}\, f\, g\, n) \end{array} .$$

L'ensemble des fonctions t_rec décrit l'ensemble des constantes récursives $\mathcal{R}ec\_\mathcal{C}onst$.

1.3 Termes récursivement flexibles

Définition 1 (Forme normale de tête récursive).
Un terme est forme normale de tête*, s'il ne s'agit pas de l'application d'une abstraction avec un autre terme, autour de laquelle sont ajoutées éventuellement quelques abstractions. ($t \neq \Lambda \dots \Lambda(\Lambda e_1)e_2$). Un tel terme t peut être abrégé par une expression de la forme $\Lambda \dots \Lambda @\, e_1\, e_2 \dots e_p$, ou même $\Lambda^n @\, e_1\, e_2 \dots e_p$, avec $@ \in i \cup \mathcal{C}onst \cup ?\mathcal{U}nk \cup \mathcal{R}ec\_\mathcal{C}onst$. @ est appelée la* tête *de t, $e_1, \dots, e_p$ sont ses* arguments*.*

*Si $@ \in \mathcal{R}ec\_\mathcal{C}onst$, c'est-à-dire $@ = type\_\mathrm{rec}$, où $type$ est définis par le biais de c constructeurs, e_{c+1} est appelé l'*argument récursif *de t.*

Le terme t est en forme normale de tête récursive*, s'il est en forme normale de tête et s'il satisfait la définition inductive suivante*
- *la tête de t est dans $i \cup \mathcal{C}onst \cup ?\mathcal{U}nk$, ou*
- *la tête de t est dans $\mathcal{R}ec\_\mathcal{C}onst$, et l'argument récursif de t est en forme normale de tête récursive, et est différent d'un constructeur correspondant au type de la tête de t.*

Exemple 1.3*:* $\Lambda\Lambda nat\_\mathrm{rec}\, f\, g\, h\, i$ est en forme normale de tête, sa tête est $nat\_\mathrm{rec} \in \mathcal{R}ec\_\mathcal{C}onst$, sont argument récursif est h ($c = 2$ dans ce cas).

Remarque:

1. Les formes (β-)normales sont un cas particulier de forme normale de tête, et, comme les réductions associées avec les combinateurs d'un type récursif sont Church-Rosser, on peut définir une forme normale plus générale, qui est un cas particulier de forme normale de tête récursive. Tout terme bien typé peut donc être réduit en chacune de ces formes. Les formes normales de têtes (resp. récursives) ne sont par contre pas uniques.

2. Une forme normale de tête d'un terme bien typé peut être obtenue en appliquant des β-réductions uniquement sur les sous-termes gauches au moment de la recherche de motifs $(\Lambda e_1)e_2$. De même, une forme normale de tête récursive peut être obtenu en appliquant les β-réductions et celles associées aux combinateurs uniquement sur les sous-termes gauches, puis en appliquant ce processus récursivement sur l'argument récursif.

   Ces formes sont intéressantes car elles sont plus facile à calculer, et elles débouchent en général sur des termes plus petits que les formes normales.

Exemple 1.4*:* $t = (\lambda x.nat\_\mathrm{rec}\, f\, ((\lambda y.y)\, g)\, (Suc\, x))\, n$

$nat\_\mathrm{rec}\, f\, ((\lambda y.y)\, g)\, (Suc\, n)$ est une forme normale de tête de t, mais pas une forme normale de tête récursive.

La forme β-normale de t est $nat\_\mathrm{rec}\, f\, g\, (Suc\, n)$

$g\, n\, (nat\_\mathrm{rec}\, f\, ((\lambda y.y)\, g)\, n)$ est une forme normale de tête récursive de t.

La forme normale de t est $g\, n\, (nat\_\mathrm{rec}\, f\, g\, n)$.

Définition 2 (Sous-terme récursif).
Le sous-terme récursif *d'un terme t en forme normale de tête récursive, est donné par la définition inductive suivante :*

- *si la tête de t est dans $i \cup \mathcal{C}onst \cup ?\mathcal{U}nk$, c'est le terme t lui-même,*
- *si la tête de t est dans $\mathcal{R}ec\_\mathcal{C}onst$, c'est le sous-terme récursif de l'argument récursif de t.*

Exemple 1.5*:* Le sous-terme récursif de $Cons\, a\, (Cons\, b\, Nil)$ est $Cons\, a\, (Cons\, b\, Nil)$.

Le sous-terme récursif de $nat\_\mathrm{rec}\, f\, g\, (list\_\mathrm{rec}\, h\, i\, l)$ est l.

Remarque: Le sous-terme récursif est en forme normale de tête, et sa tête est dans $i \cup \mathcal{C}onst \cup ?\mathcal{U}nk$.

Définition 3 (Terme récursivement flexible (terme rec-flexible)).
Un terme en forme normale de tête récursive est un terme récursivement flexible *(abrégé par la suite par* terme rec-flexible*), si la tête de son sous-terme récursif est dans $?\mathcal{U}nk$.*

Exemple 1.6*:* $?F$ et $\lambda x.nat\_\mathrm{rec}\, f\, g\, (list\_\mathrm{rec}\, h\, i\, (?F\, j\, x))$ sont des termes rec-flexibles.

$Cons\, ?A\, ?As$ et $\lambda x.nat\_\mathrm{rec}\, f\, g\, x$ ne sont pas des termes rec-flexibles.

Remarque: Hagiya introduit utilise aussi des types récursifs dans [Hag90], et ce qu'il appelle *the recursion operator* **R** correspond à peu près aux combinateurs. Il introduit des termes **R**-flexibles, qui correspondent à nos termes récursivement flexibles pour ce **R**, et utilise aussi l'unification pour programmer à partir d'examples. La principale différence avec notre approche est la façon dont il transforme les exemples initiaux en un problème d'unification.

2 Extension de l'algorithme de Huet

2.1 L'algorithme de Huet

Je rappelle ici un algorithme présente par Huet dans [Hue75], qui résout les problèmes d'unification pour le λ-calcul typé (unification d'ordre supérieur).

On se restreindra ici au cas où les termes sont unifiés modulo η-équivalence, cas qui est également présenté dans [Hue75], qui est plus simple et qui suffit pour l'usage que nous en aurons par la suite.

2.1.1 Problème d'unification

Définition 4 (Substitution).
Une substitution *est une fonction de l'ensemble des inconnues vers l'ensemble des termes $\sigma : \mathcal{U}nk \to T$ qui préserve les types, et telle que $\sigma(x) \neq x$ pour seulement un nombre fini de $x \in \mathcal{U}nk$.*

Une telle fonction peut alors être représentée par l'ensemble des couples $x, \sigma(x)$ pour lesquels $\sigma(x) \neq x$.

L'application d'une substitution σ à un terme t est le terme t dans lequel les inconnues sont remplacées par leur image par σ.

Définition 5 (Problème d'unification).
Un problème d'unification *est un ensemble (fini) de couples de termes de même type, appelés* équations

Une unification, *i.e. une solution à ce problème, est une substitution σ telle que pour toute équation s, t de ce problème (notée par la suite $s =^? t$), on ait $\sigma(s) = \sigma(t)$.*

Exemple 2.1*:* La substitution $\sigma : \{?F \mapsto Suc\}$ est une solution du problème d'unification $\{Suc(?F\,?X) =^? ?F(Suc\,?X)\}$.

Un terme en forme normale de tête est appelé *flexible* si sa tête est dans $\mathcal{U}nk$, il est appelé *rigide* sinon.

2.1.2 La procédure SIMPL

La procédure SIMPL consiste à simplifier le problème de façon à n'avoir que des équations de la forme terme flexible $=^?$ terme rigide, en supposant que les termes sont déjà en forme normale de tête η-longue (un terme t est en forme η-longue si le terme $\lambda x.t\,x$ n'est pas bien typé, i.e. il n'existe pas de terme t' tel que $t' \leadsto_\eta t$).

Il fonctionne de la manière suivante :

1. S'il n'existe pas d'équation $e_1 =^? e_2$ avec e_1 et e_2 rigide tous les deux, passer à l'étape 2,
   sinon on peut écrire $e_1 = \Lambda^{n_1} @_1\, e_1^1\, e_2^1 \ldots e_{p_1}^1$ et $e_2 = \Lambda^{n_2} @_2\, e_1^2\, e_2^2 \ldots e_{p_2}^2$.
   Si $n_1 \neq n_2$ ou $@_1 \neq @_2$, le problème n'a pas de solution.
   sinon on peut noter $n_1 = n_2 =: n, p_1 = p_2 =: p$. On enlève $e_1 =^? e_2$ du problème et on ajoute les équations suivantes : $\Lambda^n e_i^{1\prime} =^? \Lambda^n e_i^{2\prime}$ où $e_i^{j\prime}$ est la forme normale de tête de e_i^j.
   Répéter cet étape.
2. Replacer chaque équation $e_1 =^? e_2$ où e_1 est rigide et e_2 est flexible par $e_2 =^? e_1$.
3. S'il n'y a plus que des équations terme flexible $=^?$ terme flexible, le problème est résolu (ces équations sont aisément résolubles), sinon on passe à la procédure MATCH.

2.1.3 La procédure MATCH : imitation et projection

Après avoir appliqué la procédure SIMPL, il existe au moins une équation de la forme terme flexible $=^?$ terme rigide. On peut alors noter ces termes $\Lambda^{n_1} ?F\, e_1^1\, e_2^1 \dots e_{p_1}^1$ et $\Lambda^{n_2} C\, e_1^2\, e_2^2 \dots e_{p_2}^2$.

Comme on considère les formes normales de tête η-longues, et que ces termes ont le même type, on a $n_1 = n_2$.

La procédure MATCH retourne un ensemble de substitutions possibles pour l'inconnue $?F$, par le biais de deux règles :

1. **imitation** On imite le second terme, c'est-à-dire que la tête du premier terme est substituée par celle du second. La substitution résultante est $\{?F \mapsto \Lambda^{p_1} C\, E_1 \dots E_p\}$, où $E_i = ?H_i\, 0\, 1 \dots (p_1 - 1)$, $?H_i$ étant une inconnue du type convenable inutilisée jusque là, pour $i \in \{1, \dots, p\}$.
2. **projection** On remplace la tête du premier terme par une fonction retournant un de ses arguments.

   Si on suppose que le type de $?F$ est $\beta_1 \to \dots \to \beta_{p_1} \to \beta$, alors pour tout $i \in \{0 \dots (p_1 - 1)\}$ tel que le type du $i$ème argument convient, c'est-à-dire $\beta_i = \gamma_1 \to \dots \to \gamma_m \to \beta$, on peut considérer la substitution $\{?F \mapsto \Lambda^{p_1} i\, E_1 \dots E_m\}$ où $E_j = ?H_j\, 0\, 1 \dots (p_1 - 1)$, $?H_j$ étant une inconnue du type convenable inutilisée jusque là, pour $j \in \{1, \dots, p\}$.

2.1.4 Arbres de recherche

La succession des procédures SIMPL et MATCH permet de construire un arbre de recherche pour l'unification. Les noeuds de l'arbre de recherche sont des résultats de l'application de la procédure SIMPL à un problème d'unification, c'est-à-dire ils peuvent être soit un problème d'unification auquel on peut appliquer MATCH, soit un noeud F (comme failure) en cas d'échec de SIMPL, ou S en cas de succès. Les arêtes de l'arbre sont étiquetées par des substitutions.

On applique SIMPL au problème initial, ce qui nous donne la racine de l'arbre. Si celle-ci est différente de S ou F, on applique alors MATCH, ce qui nous donne une liste de substitutions possibles pour une certaine inconnue. On obtient alors les fils du noeud courant en appliquant une substitution à chacun des termes du problème initial, puis en appliquant SIMPL. On obtient l'arbre complet, potentiellement infini car l'algorithme ne termine pas forcément, en réitérant ceci avec les fils.

Une solution du problème est obtenue à partir d'un noeud S en réunissant l'ensemble des substitutions sur la branche menant à ce noeud.

Exemple 2.2: On considère le problème d'unification suivant :

$$\{?F(Suc\, ?X) =^? Suc(?F\, ?X)\} \tag{1}$$

L'arbre de recherche obtenu est représenté sur la figure 1, sur laquelle je reviens aux notations classiques pour plus de clarté.

On obtient ainsi une infinités de solutions : la première est représentée sur la figure, c'est $\{?F \mapsto \lambda x.x\}$, puis on a $\{?F \mapsto \lambda x.Suc(?H\, x); ?H \mapsto \lambda x.x\}$ qui est équivalente à $\{?F \mapsto \lambda x.Suc\, x\}$ (voir exemple 2.1), et ainsi de suite.

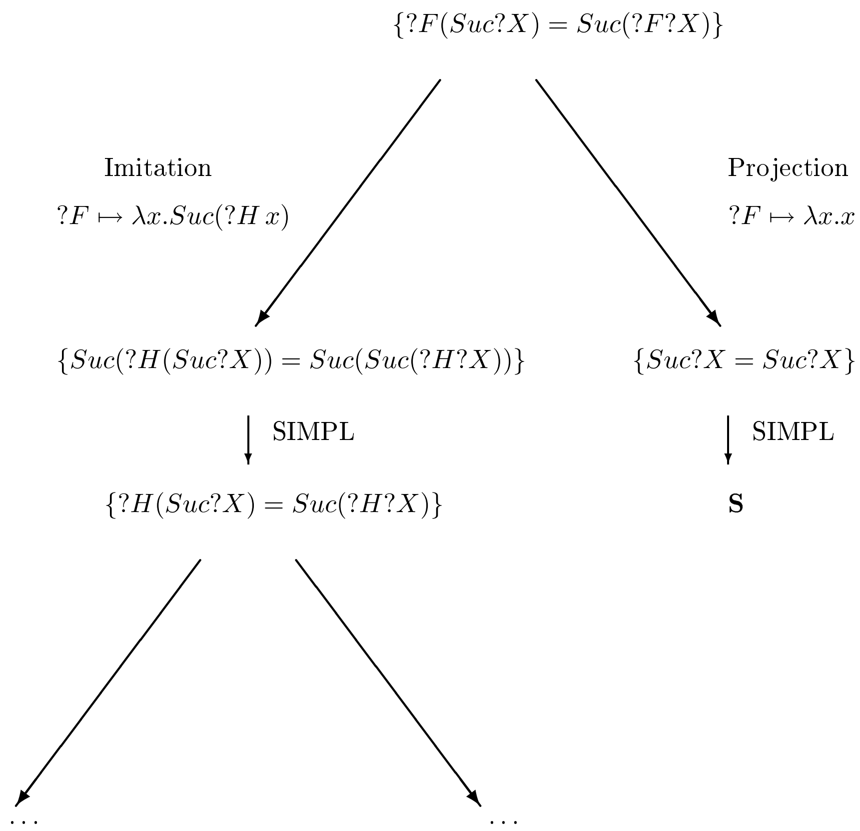

FIG. 1 – Arbre de recherche associé au problème (1)

2.2 Motivation pour un algorithme d'unification

2.2.1 Une première approche

La première approche que j'ai eue pour générer des programmes fonctionnels à partir d'exemples est la suivante :

Etant donnés le type τ de la fonction recherchée, et les exemples qu'elle doit satisfaire, on construit tous les termes de type τ dont la profondeur est bornée par une certaine constante (la profondeur d'un terme est celle de l'arbre le représentant). On applique ensuite les exemples à ces termes, et on ne garde que ceux qui les satisfont effectivement. S'il n'y en a pas, on recommence en augmentant la profondeur des termes.

L'algorithme suivant permet de faire ceci :

```
TERMS type consts depth =
  R ← {}
  if depth > 0
  then
    (* le terme construit commence par une abstraction, il est
       de la forme λx.f':
       notez que la variable x ainsi rajoutée par cette
       abstraction est considérée comme une nouvelle constante
```

```
     lors de l'appel récursif pour créer le sous-terme f'. *)
  if (type = α → β)
    let x = NEW_VARIABLE α in
      R ← MAP (fun t. λx.t)
               (TERMS β (consts ∪ {x}) (depth - 1))
  fi

  (* le terme construit commence par une constante, c-à-d
     c'est l'application d'une constante à une suite d'autres
     arguments que l'on construit récursivement. *)
  forall c ∈ consts do
    if (type(c) = type)
    then R2 ← R2 ∪ {c}
    else if (type(c) = τ1 → ... → τn → type)
    then
      for i = 1 to n do
        ai ← TERMS τi consts (depth - 1)
      done
      R ← R ∪ {c a1 ... an}
    fi
  done
fi
return R

SEARCH type constraints consts =
  R ← {}
  depth ← 1
  while R = {} do
    forall t ∈ TERM type consts depth do
      if (∀ c ∈ constraints, SATISFY t c)
      then R ← R ∪ t
      fi
    done
    depth ← depth + 1
  done
  return R
```

La fonction `NEW_VARIABLE` crée une nouvelle variable d'un type donné. Dans le cas de la notation de De Bruijn, il faut en fait augmenter les variables existantes de 1, et associer 0 avec le type donné. `MAP f s` où `f` est une fonction et `s` un ensemble renvoie l'ensemble image `f(s)`. `SATISFY t c` applique simplement l'exemple `c` au terme `t`, et renvoie vrai si celui-ci est satisfait.

L'ensemble des constantes utilisées au début de l'algorithme sont les constantes associées aux types des termes que l'on cherche à construire, par exemple $\{Zero, Suc, nat\_\mathrm{rec}\}$. On peut en fait ajouter n'importe quel terme que l'algorithme considérera comme une constante (puisque qu'il n'a besoin de connaître que son type), ce qui permet d'ajouter des fonctions auxiliaires pour faciliter l'algorithme. Ainsi on peut utiliser l'addition (que l'on peut exprimer à l'aide de nat_rec, voir ci-dessous) pour trouver plus facilement la multiplication (grâce à l'égalité $(a+1) \times b = a \times b + a$). On pourrait bien entendu chercher la multiplication en utilisant uniquement nat_rec, mais la profondeur de recherche sera alors plus grande, et on devra construire plus de termes n'étant pas solution du problème.

2.2.2 Désavantages

Cet algorithme est surprenament rapide sur des petits exemples, par exemple pour l'addition avec les contraintes $?F\,0\,3 = 3$ et $?F\,2\,4 = 6$, ce qui résulte sur le terme $\lambda m.nat\_\mathrm{rec}\,m\,(\lambda n.Suc)$.

Néanmoins, dès qu'on recherche des termes un peu plus compliqués qui nécessitent d'utiliser plus d'une fois nat_rec (par exemple pour la soustraction, qu'on peut par exemple obtenir avec le terme $\lambda m.nat\_\mathrm{rec}\,m\,(\lambda n.nat\_\mathrm{rec}\,n\,(\lambda n\,r.n)))$, la vérification des contraintes peut devenir vraiment inefficiente. En effet, la fonction d'Ackermann (ici en utilisant des notations à la ML) :

```
let rec ack (n, m) =
  match (n, m) with
    Zero , m      -> Suc m
  | Suc n, Zero   -> ack n 1
  | Suc n, Suc m -> ack n (ack (Suc n) m)
```

peut elle aussi être écrite en utilisant uniquement deux occurrences de nat_rec, par exemple avec le terme
$nat\_\mathrm{rec}\,Suc\,(\lambda n\,r_1.nat\_\mathrm{rec}\,(r_1\,(Suc\,Zero))\,(\lambda m\,r_2.r_1\,r_2))$. Au moment de l'application des exemples aux termes, on peut donc obtenir des résultats gigantesques (je rappelle que `ack (4, .)` est équivalente à une tour d´exponentielles).

Pour résoudre ce problème, il faut chercher à réduire au maximum le nombre de nat_rec, par exemple en leur attribuant un poids supérieur vis-à-vis de la profondeur du terme. Comme on l'a vu, cela ne suffit pourtant pas, puisque certaines fonctions nécessitent l'utilisation de plusieurs nat_rec. On peut alors utiliser des fonctions auxiliaires bien choisies (le prédécesseur pour la soustraction et l'addition pour la multiplication par exemple).

On pourrait aussi regretter que cet algorithme ne cherche pas les termes à η-équivalence près. Au niveau de la tête du terme, ce n'est pas un problème, puisque si f et $\lambda x.f\,x$ sont deux solutions, la première a une profondeur moindre. On peut néanmoins obtenir des termes η-équivalents, qui ont donc le même comportement face aux exemples, pour lesquels la règle η s'applique à l'intérieur des termes. Par exemple, si $f\,g\,(Suc\,Zero)$ est solution, $f\,(\lambda x.g\,x)\,(Suc\,Zero)$ aussi, et elles ont la même profondeur, donc seront toutes deux générées lors de la même recherche.

Enfin, le principal problème de cet algorithme est qu'il construit l'ensemble des termes du type donné sans se soucier des contraintes (exemples) qui pourraient pourtant guider cette construction :

- on peut parfois dire si un terme ne satisfera pas les contraintes sans avoir à le construire entièrement. Par exemple, avec la contrainte $?F\,x = Zero$, on peut supprimer tous les termes commençant par Suc.
- quand deux termes sont très proches, par exemple
  $\lambda n.nat\_\mathrm{rec}\,n\,(\lambda m\,r.Suc\,r)\,n$ et $\lambda n.nat\_\mathrm{rec}\,Zero\,(\lambda m\,r.Suc\,r)\,n$,
  avec la contrainte $?F\,100 = 200$, on aimerait ne pas avoir à appliquer deux fois les mêmes règles de réécriture au moment de la vérification des

contraintes. En effet, même si un seul de ces termes est correct, ils vont être développés de la même façon par les règles de réécriture associées à nat_rec en $\lambda n.(\lambda m\, r.Suc\, r)\, 99\, ((\lambda m\, r.Suc\, r)\, 98 \ldots ((\lambda m\, r.Suc\, r)\, Zero\, f)))$ pour $f \in \{n; Zero\}$, puis on pourra appliquer quasiment les mêmes β-réductions, et enfin la comparaison du terme obtenu avec le résultat donné dans l'exemple sera aussi la même au début.

L'idée alors est de passer les contraintes en arguments à la fonction `TERMS`, qui les modifiera avant de les utiliser pour construire les sous-termes. Le problème est que ces contraintes ainsi modifiées pourront alors contenir plus d'une inconnue, ce qui n'était pas le cas jusque là :

Exemple 2.3*:* Si on a la contrainte $?F\, 2 = 0$, et qu'on veut construire les termes commençant par nat_rec, on va vouloir créer un terme de la forme $nat\_\mathrm{rec}\, f\, g$. La contrainte modifiée sera alors $nat\_\mathrm{rec}\, f\, g\, 2 = 0 \rightsquigarrow^* g\, 1 (g\, 0\, f) = 0$, et ne pourra donc pas être utilisée directement pour créer séparément les sous-termes f et g.

2.2.3 Le problème d'unification

L'exemple 2.3 nous montre comment le problème initial peut être vu comme un cas particulier de problème d'unification : on peut y voir les sous-termes f et g comme des variables inconnues, et les contraintes deviennent alors effectivement des équations.

Néanmoins, ce problème d'unification ne correspond pas exactement à l'unification d'ordre supérieur proposé par Huet, puisqu'on recherche ici en fait une unification modulo une théorie équationnelle (celle introduite par les t_rec), c'est-à-dire on recherche une substitution telle que pour toute équation $s =^? t$ on ait $\sigma(s)$ et $\sigma(t)$ qui se réduisent au même terme via les règles associées aux t_rec et via β-réduction.

Il existe déjà des algorithmes pour résoudre cette E-unification d'ordre supérieur, en particulier celui présenté par Snyder [Sny90] (voir aussi ceux de [DJ95, QW96]). Il est utilisé par Hagiya [Hag91] pour résoudre un problème similaire au nôtre. Néanmoins, l'algorithme de Snyder est présenté dans le cas général, et n'est pas approprié pour notre problème particulier pour les raisons suivantes :

- il est donné pour n'importe quel ensemble d'équations E, même si on ne peux pas trouver de règles de réduction associées qui vérifie des propriétés intéressantes comme Church-Rosser par exemple. Comme notre système de réduction possède des formes normales, on peut penser qu'on peut utiliser ce fait pour rendre la recherche plus simple.
- le fonctionnement de cet algorithme est le même que celui de Huet, auquel on rajoute une nouvelle règle appelée *paramodulation paresseuse* (lazy paramodulation) : étant donnés deux termes u et v que l'on souhaite unifier, on crée pour chaque équation $l = r$ de $E \cup E^{-1}$ deux sous-problèmes $\beta =^? l$ et $u[\beta \leftarrow r] =^? v$, où β est un sous-terme de u et $u[\beta \leftarrow r]$ est le terme u dans lequel β est remplacé par r. On a donc facilement une explosion des sous-problèmes à traiter, et dans la pratique il faudrait choisir quelles équations de $E \cup E^{-1}$ et quels sous-termes de u il faut traiter prioritairement.
- les termes que l'on veut unifier sont particulier : ils résultent des contraintes. La partie droite de chaque équation est alors toujours un terme complètement rigide (i.e. sans aucune inconnue), à moins que l'on veuille exprimer

des propriétés de la fonction recherchée plus générales que la simple application sur des exemples.
- on ne cherche pas ici à avoir toutes les solutions satisfaisant le problème, une seule (la plus simple si possible) nous suffit. Voir la section 3.1.2 pour plus de détails.

Nous proposons donc par la suite un algorithme qui permet de résoudre le problème d'unification issu des exemples.

2.3 Modification de l'algorithme de Huet

2.3.1 Modification de SIMPL

La procédure SIMPL est modifiée afin de rendre compte des équivalences entre termes dues au règles associées aux constantes récursives. On distingue alors deux types de termes : ceux qui sont rec-flexibles, et les autres que l'on appellera *rigides* (même si ils ne correspondent pas exactement à ceux que l'on nommait ainsi avant).

On suppose donc par la suite que les termes sont non plus uniquement en forme normale de tête, mais en forme normale de tête récursive. La seule modification est alors dans l'étape 1, quand on décompose une équation entre deux termes rigides en l'ensemble des équations sur les arguments. Ces arguments doivent alors être mis en forme normale de tête récursive.

La deuxième étape est inutile, puisque comme on l'a vu, les termes apparaissant à droite dans les équations sont toujours rigides.

Au final, on se retrouve donc avec un ensemble d'équations rec-flexible $=^?$ rigide, le cas de succès étant quand l'ensemble d'équations est vide.

2.3.2 Modification de MATCH

Pour pouvoir construire des fonctions récursives qui satisfont les exemples, il faut ajouter en plus de l'imitation et de la projection une nouvelle règle qui permet d'introduire les t_rec, puisque ces dernières ne permettent pas d'ajouter des constantes ne faisant pas partie du problème initial.

Cette règle, que je nomme *récursion*, fonctionne d'une manière similaire à l'imitation, mais introduit des constantes récursives extérieures au lieu de la constante en tête du second terme.

Elle s'applique sur une équation du type flexible $=^?$ rigide, avec les mêmes notations que dans la section 2.1.3. On veut rajouter une constante récursive t_rec, le type t étant défini par le biais de c constructeurs, avec les mêmes notations qu'en 1.2.

t_rec est alors du type polymorphe suivant : $(\tau_{1,1} \to \ldots \to \tau_{1,n_1} \to \beta \to \ldots \to \beta) \to \ldots \to (\tau_{c,1} \to \ldots \to \tau_{c,n_c} \to \beta \to \ldots \to \beta) \to (\alpha_1, \ldots, \alpha_k)t \to \beta$

On introduit alors la substitution $\{?F \mapsto \Lambda^{p_1} t\_\mathrm{rec}\, E_1 \ldots E_{c+1}\}$, où $E_i = ?H_i\, 0\, 1 \ldots (p_1 - 1)$, $?H_i$ étant une inconnue du type convenable, c'est-à-dire $\beta_1 \to \ldots \to \beta_{p_1} \to \tau_{i,1} \to \ldots \to \tau_{i,n_i} \to \beta \to \ldots \to \beta$ pour $i \leq c$, et $\beta_1 \to \ldots \to \beta_{p_1} \to (\alpha_1, \ldots, \alpha_k)t \to \beta$ pour $i = c+1$, inutilisée jusque là, pour $i \in \{1, \ldots, c+1\}$.

Après avoir appliqué cette règle, on obtient un terme rec-flexible dans une équation qui après application de SIMPL, ne peut être modifié par aucune des règles existantes (imitation, projection, récursion). Il faut donc modifier l'imitation et la projection afin que celles-ci traitent aussi ce type d'équation. L'idée est de les appliquer sur l'argument récursif plutôt que sur la tête du terme lui-même. En effet, la tête de celui-ci est par définition dans $\mathcal{U}nk$.

Néanmoins, on a également besoin d'un terme à faire correspondre en tant que partie droite de l'équation. Dans le cas de la projection, ce n'est pas un problème, puisqu'on n'utilise que le terme de gauche si on regarde bien. Pour l'imitation, on a besoin de construire un terme que l'on cherche à imiter.

On a alors deux cas possibles :

- le terme de gauche lui-même est flexible, on peut appliquer la règle imitation comme avant avec le terme de droite initial.
- la tête du terme de gauche est une constante récursive, l'argument récursif est alors le $c+1$ème argument d'une constante récursive d'un type récursif t nécessitant c constructeurs. L'argument récursif a donc le type t. On utilise comme terme de droite tous les constructeurs possibles du type t (i.e. $C_1 \ldots C_c$) dans une forme η-longue. Seule la tête de ce terme étant utilisée dans la règle d'imitation, on n'a pas à se soucier de l'égalité des types des termes de gauche et de droite (sinon on peut rajouter des arguments non-significatifs aux constructeurs).

2.3.3 Exemple : construction de la fonction prédécesseur

Exemple 2.4*:* Nous traitons ici de façon quasi exhaustive comment construire la fonction prédécesseur à partir des exemples $?F\,0 =^? 0$ et $?F\,2 =^? 1$ (soit $?F\,Zero =^? Zero$ et $?F\,Suc(Suc\,Zero) =^? Suc\,Zero$), on utilisera les notations classiques pour plus de simplicité (notamment du fait de l'ambiguïté des variables liées et des entiers naturels). On se restreindra à une seule utilisation de la règle récursion (voir la section suivante). On trouvera sur la figure 2 page 35 un schéma de l'arbre de recherche.

Les équations ne sont pas modifiées par SIMPL, on passe directement à MATCH. On a deux choix possibles pour savoir quelle équation on doit traiter. On ne traite ici que le cas où on choisit $?F\,0 =^? 0$.

La règle d'imitation nous donne la substitution $?F \mapsto \lambda x.0$, ce qui donne les équations suivantes $0 =^? 0$ et $0 =^? 1$. SIMPL renvoie alors F, puisqu'on a deux termes rigides dont la tête est différente.

La règle de projection nous donne la substitution $?F \mapsto \lambda x.x$, ce qui donne les équations suivantes : $0 =^? 0$ et $2 =^? 1$. SIMPL revoit également F, puisque la seconde équation est d'abord simplifiée en $1 =^? 0$, soit deux termes rigides dont la tête est différente.

La règle de récursion nous donne la substitution
$?F \mapsto \lambda x.nat\_\mathrm{rec}\,(?G_1\,x)\,(?G_2\,x)\,(?G_3\,x)$ avec $?G_1$ de type $nat \to nat$, $?G_2$ de type $nat \to nat \to nat \to nat$ et $?G_3$ de type $nat \to nat$. On obtient ainsi les équations $nat\_\mathrm{rec}\,(?G_1\,0)\,(?G_2\,0)\,(?G_3\,0) =^? 0$ et $nat\_\mathrm{rec}\,(?G_1\,2)\,(?G_2\,2)\,(?G_3\,2) =^? 1$, immodifiées par SIMPL.

A partir de ce nouveau problème, on peut appliquer la règle imitation modifiée. Comme le type nat est construit à l'aide de deux constructeurs, il y a deux possibilités :

- $?G_3 \mapsto \lambda x.0$, on obtient alors les équations $?G_1\,0 =^? 0$ et $?G_1\,2 =^? 1$. On est retombé alors sur le problème initial. Comme on a décidé de ne plus utiliser de règle récursion, il n'y a pas de solutions à partir de ce noeud.

– $?G_3 \mapsto \lambda x.Suc\,(?H_1\, x)$, avec $?H_1$ de type $nat \to nat$. On obtient alors les équations $?G_2\, 0\,(?H_1\, 0)\,(nat\_\text{rec}\,(?G_1\, 0)\,(?G_2\, 0)\,(?H_1\, 0)) =^? 0$ et $?G_2\, 2\,(?H_1\, 2)\,(nat\_\text{rec}\,(?G_1\, 2)\,(?G_2\, 2)\,(?H_1\, 2)) =^? 1$, inchangées par SIMPL. A partir de ce noeud, si on applique la règle imitation on obtient la substitution $?G_2 \mapsto \lambda x\, y\, z.0$ dont SIMPL renvoit F pour la même raison que pour $?F$ ci-dessus.
  Si on applique la règle projection on a trois substitutions possibles : $?G_2 \mapsto \lambda x\, y\, z.x$, $?G_2 \mapsto \lambda x\, y\, z.y$ et $?G_2 \mapsto \lambda x\, y\, z.z$. La première donne les équations $0 =^? 0$ et $2 =^? 1$, SIMPL renvoie F. La deuxième donne les équations $?H_1\, 0 =^? 0$ et $?H_1\, 2 =^? 1$, on est retombé sur le problème initial, il n'y a pas de solution à partir de ce noeud. La troisième donne les équations $nat\_\text{rec}\,(?G_1\, 0)\,(?G_2\, 0)\,(?H_1\, 0) =^? 0$ et $nat\_\text{rec}\,(?G_1\, 2)\,(?G_2\, 2)\,(?H_1\, 2) =^? 1$, on retombe sur le problème obtenu après l'application de la règle récursion.

On a fini avec les sous-arbres obtenus par imitation, on revient à ce problème. Si on applique la règle de projection, on obtient la substitution $?G_3 \mapsto \lambda x.x$, et les équations $?G_1\, 0 =^? 0$ et $?G_2\, 2\, 1\,(?G_2\, 2\, 0\,(?G_1\, 2)) =^? 1$, inchangées par SIMPL.

On peut alors appliquer les règles imitation et projection sur la première équation, qui donnent resp. les substitutions $?G_1 \mapsto \lambda x.0$ et $?G_1 \mapsto \lambda x.x$. Toutes deux transforment la première équation en $0 =^? 0$, qui est éliminée par SIMPL (pas d'arguments, donc pas de sous-problèmes créés). La deuxième devient resp. $?G_2\, 2\, 1\,(?G_2\, 2\, 0\, 0) =^? 1$ et $?G_2\, 2\, 1\,(?G_2\, 2\, 0\, 2) =^? 1$, toutes deux inchangées par SIMPL. On traite par la suite le cas général $?G_2\, 2\, 1\,(?G_2\, 2\, 0\, i) =^? 1$ pour $i \in \{0, 2\}$.

Si on applique la règle imitation, on obtient la substitution $?G_2 \mapsto \lambda x\, y\, z.Suc\,(?H_2\, x\, y\, z)$ avec $?H_2$ de type $nat \to nat \to nat \to nat$, ce qui donne l'équation $Suc\,(?H_2\, 2\, 1\,(Suc\,(?H_2\, 2\, 0\, i))) =^? 1$, que SIMPL transforme en $?H_2\, 2\, 1\,(Suc\,(?H_2\, 2\, 0\, i)) =^? 0$. A partir de ce noeud, on peut appliquer imitation ou projection, ce qui donne les substitutions :

– $?H_2 \mapsto \lambda x\, y\, z.0$ qui donne l'équation $0 =^? 0$; SIMPL renvoit alors S, on a obtenu une première solution au problème.
– $?H_2 \mapsto \lambda x\, y\, z.x$, $?H_2 \mapsto \lambda x\, y\, z.y$ et $?H_2 \mapsto \lambda x\, y\, z.z$, qui donnent respectivement les équations $2 =^? 0$, $1 =^? 0$ et $Suc\, i =^? 0$; SIMPL renvoit F dans tous les cas.

Si on applique la règle projection, on obtient les substitutions $?G_2 \mapsto \lambda x\, y\, z.x$, $?G_2 \mapsto \lambda x\, y\, z.y$ et $?G_2 \mapsto \lambda x\, y\, z.z$, qui donnent resp. les équations $2 =^? 1$, $1 =^? 1$ et $i =^? 1$. Seule la deuxième se simplifie en S, les deux autres en F (je rappelle que $i \in \{0, 2\}$). On a alors une deuxième solution.

L'algorithme nous a donc essentiellement donné deux solutions (plus celles obtenues à partir des sous-problèmes qui étaient équivalents à un problème déjà vu). Ces deux solutions sont les suivantes : $?F \mapsto \lambda n.nat\_\text{rec}\, 0\,(\lambda n\, r.Suc\, 0)\, n$ et $?F \mapsto \lambda nat\_\text{rec}\, 0\,(\lambda n\, r.n)\, n$. Elles correspondent aux fonctions ML suivantes :

```
let rec solution_un n =
  match n with
    Zero   -> Zero
  | Suc m  -> Suc Zero

let rec solution_deux n =
```

```
match n with
  Zero  -> Zero
| Suc m -> m
```

Seule la deuxième correspond effectivement à la fonction prédécesseur attendue, mais la première satisfait également les exemples donnés. Pour n'obtenir que la seconde, il aurait fallu ajouter un autre exemple au problème initial ($?F\,1 =^? 0$ par exemple).

2.3.4 Extensions/Restrictions possibles

On ne considère que les formes normales de tête récursives, et pas directement les formes normales générales, pour les raisons suivantes :

- Si la branche de l'arbre de recherche sur laquelle on est n'aboutit à aucune solution, le calcul des arguments non-récursifs aura été inutile. Celui des arguments récursifs par contre est nécessaire puisqu'il est utilisé dans les nouvelles règles d'imitation et de projection.
- Même si cet algorithme est plus "intelligent" que la première approche proposée, il peut potentiellement aussi créer la fonction d'Ackermann. Si on développe tous les arguments de chaque terme, on peut obtenir des termes énormes, or l'application pratique de cet algorithme montre que la question de la mémoire est importante.

Pour la même raison que pour la première approche, on va chercher à limiter le nombre de constantes récursives utilisées. Pour cela, on peut n'autoriser qu'un nombre fixe de constantes récursives par branches de l'arbre de recherche, en associant à chaque noeud de celui ci un ensemble de constantes récursives que la règle récursion peut consommer, cet ensemble étant transmis aux fils en enlevant la constante éventuellement consommée.

On peux aussi penser à utiliser des fonctions auxiliaires. Toutefois, ceci n'est plus aussi simple que dans la première approche proposée, car la règle récursion a besoin du polymorphisme des constantes récursives. On a alors deux solutions, soit utiliser uniquement des fonctions auxiliaires polymorphes, comme les fonctions t_case[3], où alors proposer une deuxième règle récursion plus simple pour les fonctions auxiliaires n'utilisant pas de récursion.

Par ailleurs, comme on l'a vu, il faut choisir une équation parmi le problème pour lui appliquer les règles imitation, projection et récursion. Huet a montré que ce choix n'est pas important envers la correction de l´algorithme, mais que considérer l'ensemble d'équations comme une file était en pratique plus efficient.

L'extension de la règle imitation est en fait inutile, comme le laisse deviner l'exemple suivant :

Exemple 2.5*:* On a un première substitution due à la règle récursion $?F \mapsto nat\_\mathrm{rec}\,?G\,?H\,?I$, puis on applique la règle imitation avec le constructeur Suc, ce qui donne la substitution $?I \mapsto Suc\,?J$. On obtient alors la substitution $?F \mapsto nat\_\mathrm{rec}\,?G\,?H\,(Suc\,?J)$, qui est équivalente à $?F \mapsto ?H\,?J\,(nat\_\mathrm{rec}\,?G\,?H\,?J)$. Par la suite, on aura probablement une substitution $?H \mapsto t$ ce qui donnera

[3]par exemple, pour le type *list*, la fonction $list\_\mathrm{case}$ est associée aux règles $list\_\mathrm{case}\,f\,g\,Nil = f$ et $list\_\mathrm{case}\,f\,g\,(Cons\,x\,xs) = g\,x\,xs$

au final $?F \mapsto t\,?J\,(nat\_\text{rec}\,?G\,t\,?J)$, qu'on aura pu obtenir différemment en appliquant d'abord la substitution $?F \mapsto t\,?K\,?L$ au départ.

Enfin on peut penser à une règle récursion étendue aux équations rec-flexible $=^?$ rigide, en l'appliquant aux arguments récursifs comme pour la projection. Néanmoins, ceci n'apporte pas un degré d'expression supplémentaire, comme on peut s'en convaincre avec l'exemple suivant :

Exemple 2.6*:* On considère la fonction suivante : $nat\_\text{rec}\,f\,g\,(list\_\text{rec}\,i\,j\,l)$. f est de type α, l de type $\beta\,list$; g est par conséquent de type $nat \rightarrow \alpha \rightarrow \alpha$, i de type nat et j de type $\beta \rightarrow \beta\,list \rightarrow nat \rightarrow nat$.

Cette fonction est équivalente à la suivante :
$list\_\text{rec}\,(nat\_\text{rec}\,f\,g\,i)\,(\Lambda^3 nat\_\text{rec}\,f\,g\,(j\,2\,1\,(list\_\text{rec}\,i\,j\,1)))\,l$. En effet, si $l = Nil$, alors elles se réduisent toutes deux en $nat\_\text{rec}\,f\,g\,i$, et si $l = Cons\,x\,q$ en $nat\_\text{rec}\,f\,g\,(j\,x\,q\,(list\_\text{rec}\,i\,j\,q))$.

Remarque: La troisième variable du sous-terme $\Lambda^3 nat\_\text{rec}\,f\,g\,(j\,2\,1\,(list\_\text{rec}\,i\,j\,1))$ n'est pas utilisée, le premier $list_$rec est en fait un $list_$case. Ceci est important, comme on cherche à minimiser le nombre de constantes récursives à utiliser.

3 Résultats

3.1 Correction et non-terminaison de l'algorithme

3.1.1 Sûreté

L'algorithme proposé est sain :

Théorème 1.
Soit un ensemble d'équations formant un problème d'unification E, l'ensemble de règles de simplification associées aux combinateurs R, et une substitution σ retournée par l'algorithme décrit ci-dessus.

Alors pour toute équation $s =^? t$ de E on a $\sigma(s) \leftrightsquigarrow^*_R \sigma(t)$.

Démonstration. Elle provient essentiellement de celle de la sûreté de l'algorithme de Huet, qu'il a démontrée dans la section 4.1 de [Hue75].

Celle-ci provient uniquement de la procédure SIMPL, ce qui est logique puisque MATCH ne fait que proposer des substitutions possibles pour la suite de l'algorithme. La seule différence entre notre procédure SIMPL et celle de Huet concerne les formes normales des termes. Comme on a pris soin d'utiliser des formes normales de tête récursives à la place des formes normales de tête, on a donc bien l'égalité entre termes d'une équation, modulo la théorie équationnelle induite par R, après avoir appliqué une substitution aboutissant à un noeud S. □

3.1.2 Complétude

La notion de complétude est plus difficile à définir dans notre cas, puisqu'on cherche à construire des programmes à partir d'exemples. Une seule solution nous suffit, le seul cas posant problème étant quand l'algorithme ne renvoie aucune solution. Ceci est moins critique que pour la sûreté, puisque si aucune fonction satisfiant les exemples n'est trouvée, le programmeur peut tenter à la main soit de se convaincre qu'il n'en existe effectivement pas, soit de la construire lui-même.

Notons d'une part que si une solution existe, l'algorithme propose bien au minimum une solution (à condition de ne pas borner le nombre de combinateurs récursifs), car les fonctions correspondant au pattern-matching des exemples peuvent être construits à condition de disposer d'un nombre suffisant de t_rec (ou même en fait de t_case).

De plus, la complétude de l'algorithme de Huet, également démontrée dans [Hue75], nous laisse croire que nous couvrons une grande partie des programmes fonctionnels.

Dans la version donnée, l'algorithme n'est toutefois pas complet. En effet, il ne permet pas d'utiliser des arguments "globaux" à l'intérieur des récursions. Pour cela, il faudrait à nouveau modifier la règle de récursion pour qu'elle renvoie la substitution $\{?F \mapsto \Lambda^{p_1} t\_\mathrm{rec}\, E_1 \ldots E_{c+1}\, 0 \ldots (p-1)\}$, où $E_i = ?H_i\, 0\, 1 \ldots (p_1-1)$, en changeant le type des $?H_i$. Toutefois, cette version de l'algorithme devient hautement inefficace, du fait de l'explosion de la taille des types des inconnues. La recherche de la fonction soustraction à partir d'exemples prend alors des heures à la place de quelques dixièmes de secondes. Mon avis est que dans la plupart des cas étudiés, cette extension n'est pas utile. De plus, on peut toujours lancer la version épurée jusqu'à une certaine profondeur, puis lancer la version complète si la première ne donne pas de solutions. Pour informations, la fonction d'Ackermann nécessite la version complète de l'algorithme, et c'est le seul exemple de fonctions "usuelles" pour lequel cela semble être le cas.

Si on utilise cette version généralisée et qu'on applique la règle récursion à tout terme rec-flexible (qui est comme on l'a vu en 2.6 équivalent), il semble alors que l'on puisse obtenir tout programme fonctionnel pouvant s'écrire comme composition de programmes fonctionnels primitifs récursifs (classe des programmes fonctionnels récursifs d'ordre supérieur). La preuve de ceci semble toutefois difficile à établir.

3.1.3 Non-terminaison de l'algorithme

Si on permet d'utiliser une infinité de fois la règle récursion, l'algorithme ne termine évidemment pas, puisque si on applique la règle récursion puis une règle imitation avec un constructeur nécessitant un argument du même type (par exemple *Suc* ou *Cons*), on retombe sur un terme flexible, sur lequel on peut répéter ce processus.

On peut trouver un exemple simple sans même utiliser de constantes récursives dont l'exécution avec l'algorithme ne termine pas :

Exemple 3.1*:* Soit un type de base (i.e. non curryfié) b et un élément B de ce type. On considère le problème suivant : $?F\,(\lambda x.x) =^? B$, où $?F$ a le type $(b \to b) \to b$.

Si on applique la règle projection, on obtient la substitution $?F \mapsto \lambda x.x\,(?G\,x)$ avec $?G$ de type $(b \to b) \to b$, ce qui donne l'équation $(\lambda x.x\,(?G\,x))(\lambda x.x) \rightsquigarrow ?G\,(\lambda x.x) =^? B$. On est retombé sur le problème initial, l'algorithme ne termine donc pas.

Remarque: Il admet pour solution $\lambda x.B, \lambda x.x\,B, \lambda x.x\,(x\,B), \ldots$. Si on se contente d'une solution, l'algorithme termine donc après la première imitation.

On a déjà vu ce phénomène dans l'exemple 2.4, où des sous-problèmes étaient équivalents à leur parents. Une solution pour permettre à l'algorithme de terminer serait de construire non plus un arbre de recherche, mais un graphe orienté où on revient vers le noeud parent quand on obtient un problème équivalent. Si on se contente d'une solution, on peut même carrément marquer les noeuds qui reviennent vers les parents comme F. Néanmoins, ceci ne suffit pas, car on peut trouver un exemple où on a besoin d'une infinité de problèmes non équivalents :

Exemple 3.2*:* Toujours avec le même type b, on considère le problème $?F\,(\lambda x.x\,B) =^? B$, avec $?F$ de type $((b \to b) \to b) \to b$.

Si on applique la règle projection, on obtient la substitution $?F \mapsto \lambda x.x\,(?G\,x)$ avec $?G$ de type $((b \to b) \to b) \to b \to b$, ce qui donne l'équation $?G\,(\lambda x.x\,B)\,B =^? B$.

En appliquant une nouvelle fois la règle projection, on obtient la substitution $?G \mapsto \lambda x\,y.x\,(?H\,x\,y)$ qui donne $?H\,(\lambda x.x\,B)\,B\,B =^? B$, et en répétant ce processus on obtient une infinité de noeuds non équivalents.

Remarque: La encore, le problème possédant une solution ($\lambda x.B$ par exemple), on peut s'arrêter quand on l'a trouvée (après la première imitation ici aussi). Néanmoins, si le problème n'a pas de solutions (par exemple, on considérant un élément C de type b différent de B, en rajoutant l'équation $?F\,(\lambda x.x\,B) =^? C$), l'algorithme ne termine effectivement pas.

3.2 Largeur de l'arbre de recherche, performances

3.2.1 Pour l'addition

On recherche l'addition grâce aux exemples suivants :

1. $?F\,0\,2 =^? 2$
2. $?F\,2\,3 =^? 5$
3. $?F\,1\,2 =^? 3$,

avec une seule utilisation de nat_rec. Avec deux exemples, on a besoin d'une profondeur de l'arbre de recherche de 5 avant de trouver les premières solutions qui sont : $\lambda a\,b.nat\_\mathrm{rec}\,a\,(\lambda c\,d.Suc\,d)\,b$ et son symétrique (du fait de la commutativité de l'addition) $\lambda a\,b.nat\_\mathrm{rec}\,b\,(\lambda c\,d.Suc\,d)\,a$.

L'arbre de recherche en fonction du nombre d'exemples utilisés a la forme suivante :

	1 exemple		2 exemples		3 exemples	
Profondeur	Largeur	Solutions	Largeur	Solutions	Largeur	Solutions
1	2	1	2	0	2	0
2	4	1	4	0	4	0
3	6	5	7	0	7	0
4	9	8	12	0	11	0
5	3	22	10	2	11	2
6	0	28	8	5	7	2
7	0	28	7	10	3	2
8			7	17	2	2
9			4	24	0	2
10			2	28	0	2

A noter que le nombre de solution correspond au nombre de fonctions renvoyées par l'algorithme, et non au nombre de fonctions correspondant effectivement à l'addition. Les nombre de noeuds indiqué est celui du nombre de noeuds ne donnant pas S ou F à la profondeur indiquée, c'est-à-dire le nombre de noeuds restant à calculer.

Comme on le voit, la recherche avec trois exemples renvoie uniquement les bonnes solutions.

Sur l'ordinateur sur lequel j'ai effectué les tests (un i686 cadencé à 1,7Ghz), la recherche prend 0.007s.

3.2.2 Pour la soustraction

Comme on l'a vu, la soustraction nécessite 2 occurrences de nat_rec. On utilise les exemples suivants :

1. $?F\,0\,2 =^? 0$
2. $?F\,2\,0 =^? 2$
3. $?F\,3\,2 =^? 1$
4. $?F\,3\,1 =^? 2$.

On peut construire la solution $\lambda a\,b.nat\_\mathrm{rec}\,a\,(\lambda c\,d.nat\_\mathrm{rec}\,d\,(\lambda e\,f.e)\,d)\,b$ à partir de la profondeur 7. Deux branches permettent de la générer, ce qui montre la redondance de l'algorithme.

Du fait de l'utilisation de deux combinateurs, l'arbre de recherche est plus large, comme indiqué dans le tableau suivant :

Profondeur	Largeur	Solutions
1	1	0
2	2	0
3	5	0
4	13	0
5	22	0
6	47	0
7	34	2
8	7	2
9	0	2

La recherche prend alors 0.065s.

3.2.3 Problèmes avec la version plus complète de l'algorithme

Je note ici quelques résultats obtenus avec la version plus complète de l'algorithme, avec le passage d'arguments "globaux" (voir section 3.1.2). On voit ici pourquoi elle n'est pas utilisable dans la pratique.

On considère la construction de la soustraction à partir des trois premiers exemples indiqués ci-dessus.

On peut théoriquement trouver la solution
$\lambda a\,b.nat\_\mathrm{rec}\,(\lambda c\,d.a)\,(\lambda c\,d\,e\,f.nat\_\mathrm{rec}\,(\lambda\,g\,h\,i\,j\,k\,l.d)\,(\lambda g\,h\,i\,j\,k\,l\,m\,n.g)\,d\,a\,b\,c\,d\,e\,f)$ $b\,a\,b$ à partir de la profondeur 7 (on comprend, devant la taille de ce terme, d'où provient l'inefficacité). L'implémentation de l'algorithme que j'ai écrite n'est malheureusement pas suffisante pour trouver cette solution, du fait d'un problème de la gestion de la mémoire.

Pour les profondeurs plus faibles, on obtient les résultats suivants :

Profondeur	Largeur	Solutions
1	1	0
2	2	0
3	5	0
4	17	0
5	54	0
6	243	0

On voit qu'ici le problème grossit beaucoup plus vite que dans les cas précédent. Les termes sont également plus importants. Pour la profondeur 6, on nécessite déjà environ 7 minutes (sur une autre machine toutefois, un super sparc, mais qui dispose de 3 Go de mémoire).

La solution dans ce cas est de ne pas utiliser deux nat_rec mais plutôt un nat_rec et un nat_case. On obtient alors entre autres la solution
$\Lambda^2 nat\_\mathrm{case}\,(\Lambda^2 1)\,(\Lambda^3 nat\_\mathrm{rec}\,(\Lambda^5 Zero)\,(\Lambda^7 2)\,1\,4\,3\,2\,1\,0)\,0\,1\,0$.

L'arbre de recherche à alors la forme suivante :

Profondeur	Largeur	Solutions
1	2	0
2	4	0
3	8	0
4	29	0
5	87	0
6	290	0
7	1130	138

L'arbre de recherche augmente ici de façon aussi rapide que dans le cas précédant. Les termes étant néanmoins plus petits, on ne nécessite alors que de 1min 50s pour trouver les solutions. On pourrait bien entendu aller plus vite en se contentant de la première solution trouvée.

3.3 Améliorations possibles

Comme on l'a vu, l'implémentation proposée (cf. Annexe A) est loin d'être satisfaisante. D'une part elle n'est pas complète, d'autre part, elle n'est pas du tout optimisée. Cela provient essentiellement du parcours de l'arbre de recherche que j'effectue, qui cherche à être en profondeur afin de préserver la mémoire sans l'être tout à fait. En effet, même sans les appliquer, l'algorithme calcule toutes les substitutions possibles avant de passer aux fils, et les gardent donc en mémoire. Il vaudrait mieux passer au fils directement après le calcul d'une substitution. Néanmoins, les substitutions possibles sont bien moins grosses que leur application sur les termes, et changer ceci ne devrait pas apporter de grands progrès.

De plus, on calcule ici toutes les solutions de profondeur fixée, alors que nous n'avons besoin que d'une. Il serait plus intelligent de quitter l'algorithme dès qu'une solution est trouvée.

On pourrait aussi supprimer les noeuds correspondant aux problèmes déjà traités, comme suggéré en 2.3.4. Mais ceci nécessiterait de garder en mémoire ces derniers, ce qui a priori est une mauvaise idée.

Malgré tout, la principal amélioration qui resterait a fournir serait de rendre le parcours dans l'arbre de recherche plus intelligent en orientant la recherche,c'est-à-dire parcourir en premier les fils dont on pense qu'ils fourniront plus probablement une solution. Néanmoins, ce concept reste vague, car il est difficile de dire quels sont ces fils. La meilleur solution serait de proposer une heuristique obtenue à partir de l'observation des arbres de recherche sur des exemples simples.

Enfin, il reste à trouver un moyen de rendre l'algorithme plus complet sans toutefois faire trop de concessions au niveau de l'efficacité.

Conclusion

On a donc proposé ici deux extensions à l'algorithme de Huet, qui permettent de résoudre le problème de la construction de programmes fonctionnels à partir d'exemples. La première n'est pas complète, mais fourni des résultats sur la plupart des exemples usuels qu'on lui fournit (l'algorithme a par exemple été testé pour les fonctions prédécesseur, addition, soustraction, multiplication, longueur de liste, concaténation de liste...). Elle a l'avantage surtout d'être très rapide. L'autre version, certes plus complète, est trop peu efficiente pour pouvoir être utilisée en pratique.

Ces difficultés font de l'algorithme proposé uniquement un outil d'aide au programmeur, et pas véritable de générateur automatique de programmes fonctionnels auquel on peut se fier les yeux fermés. Toutefois, la nature même du problème supposait une telle conclusion. La preuve de la sûreté de l'algorithme nous aide au moins à garantir la validité des solutions proposées. C'est au programmeur alors de trouver le nombre d'exemples adéquats afin de trouver les solutions sans problèmes, et de vérifier celles-ci après.

On peut toutefois proposer cet algorithme pour des applications moins proches du problème initial, comme en intelligence artificielle où l'être virtuel essaierait de déduire des règles générales de la façon dont le monde qui l'entoure fonctionne, en générant de telles fonctions à partir des exemples que lui apporte son expérience, en affinant le modèle toujours un peu plus en récoltant de nouveaux exemples.

Une autre piste intéressante serait d'essayer de généraliser l'algorithme proposé au cas général de l'unification d'ordre supérieur modulo une théorie équationnelle. Néanmoins, il faudrait alors obtenir une preuve de la complétude de l'algorithme donné.

Références

[Ber98] Stefan Berghofer. Definitorische konstruktion induktiver datentypen in isabelle/hol. Master's thesis, Institut für Informatik, Technische Universität München, 1998.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem. *Indagationes Mathematicae*, 34 :381–392, 1972.

[DJ95] Daniel J. Dougherty and Patricia Johann. A combinatory logic approach to higher-order E-unification. *Theoretical Computer Science*, 139(1–2) :207–242, 1995.

[Hag90] Masami Hagiya. Programming by example and proving by example using higher-order unification. In *Proceedings of the tenth international conference on Automated deduction*, pages 588–602. Springer-Verlag New York, Inc., 1990.

[Hag91] Masami Hagiya. Synthesis of rewrite programs by higher-order and semantic unification. *New Gen. Comput.*, 8(4) :403–420, 1991.

[Hue75] G.P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1 :27–57, 1975.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.

[QW96] Zhenyu Qian and Kang Wang. Modular higher-order equational preunification. *J. Symb. Comput.*, 22(4) :401–424, 1996.

[Sny90] Wayne Snyder. Higher order e-unification. In *Proceedings of the tenth international conference on Automated deduction*, pages 573–587. Springer-Verlag New York, Inc., 1990.

A Annexe : implémentation de l'algorithme en OCaml

A.1 Définitions : `defs.ml`

On définit les types, quelques fonctions utiles et les formes normales de tête, plus faciles à utiliser avec l'algorithme.

```
type 'a types = Base of 'a | Curry of 'a types * 'a types

(* Var are bounded variables, Const are context constants or constructors
   (Zero, Suc for instance), Unk are free variables with type in form
   (args types, base type), Rec_const are recursive constant (nat_rec for
   instance)
 *)
type 'a term =
    Var of int
  | Const of string
  | Abs of 'a term
  | App of 'a term * 'a term
  | Unk of string * ('a types list * 'a types)
  | Rec_const of string

(* more accurate form for terms: recursive head normal form *)
type 'a head_normal_form =
     n : int;
      head : 'a term;
      args : 'a term list;
      p : int;
      alphas : 'a types list;
      beta : 'a types;
      t : 'a types

(* table containing types of the constants *)
let type_table : (string, string types) Hashtbl.t = Hashtbl.create 10
```

```
let add_type = Hashtbl.add type_table

let find_type t =
  try Hashtbl.find type_table t
  with Not_found -> failwith ("untyped constant: "^t)


(* table containing constructors associated with recursive constants *)
let constructors_table : (string, string head_normal_form list) Hashtbl.t
    = Hashtbl.create 10

let add_constructors = Hashtbl.add constructors_table

let find_constructors t =
  try Hashtbl.find constructors_table t
  with Not_found -> failwith ("unknown constructor: "^t)


(* table containing base type associated with recursive constants *)
let base_types_table : (string, string types) Hashtbl.t = Hashtbl.create 10

let add_base_types = Hashtbl.add base_types_table

let find_base_types t =
  try Hashtbl.find base_types_table t
  with Not_found -> failwith ("unknown constructor: "^t)


(* this execption is raised when an expression of type 'a head_normal_form
   does not represent a term in recursive head normal form (typically when
   the head is actually an abstraction)
*)
exception Not_normal_form of string term * string types




(* Representation of types with lists: *)
(* uncurryfy ('b1 -> 'b2 -> ... -> 'bn -> 'a) = ['b1, ..., 'bn]
   if 'a is a basis type *)
let rec uncurryfy = function
  | Curry(a, b) -> a::(uncurryfy b)
  | b -> []

(* rev_uncurryfy ('b1 -> 'b2 -> ... -> 'bn -> 'a) = ['bn, ..., 'b1], 'a
   if 'a is a basis type *)
let rev_uncurryfy t =
  let rec aux accu = function
    | Curry(a, b) -> aux (a::accu) b
    | b -> accu, b
```

```
  in aux [] t

(* rev_curryfy 'a ['bn, ..., 'b1] = ('b1 -> 'b2 -> ... -> 'bn -> 'a) *)
let rec rev_curryfy b = function
  | x::q -> rev_curryfy (Curry (x, b)) q
  | [] -> b


(* creation of new variables *)
let unk_counter = ref (-1)

let new_unk (l, a) =
  let rec aux l = function
    | Curry (a, b) -> aux (a::l) b
    | b -> l, b
  in
  unk_counter := !unk_counter + 1;
  if (!unk_counter = max_int) then
    failwith "Too many variable created!"
  else
    Unk (string_of_int !unk_counter, aux l a)


(* term to head_normal_form, s must be in long recursive head normal form *)
let term_to_hnf (s, t) =
  let rec apps l i = function
    | App (a, b) -> apps (b::l) (i+1) a
    | Abs _ ->
raise (Not_normal_form (s, t))
    | c -> c, l, i
  in
  let rec abs alphas n = function
    | Abs s, Curry (a, b) -> abs (a::alphas) (n+1) (s, b)
    | Abs _, b ->
raise (Not_normal_form (s, t))
    | c, b ->
let h, args, p = apps [] 0 c in
 n = n; head = h; args = args; p = p; alphas = alphas;
  beta = b; t = t
  in
  let _::q = [t] in  (* without this, polymorph type can't be generalized *)
  abs q 0 (s, t)


(* abs r n abstract the term r n times *)
let rec abs r = function
  | 0 -> r
  | n -> abs (Abs r) (n-1)

(* inverse function of term_to_hnf *)
```

```
let hnf_to_term e =
  let rec app c = function
    | [] -> c
    | ei::es -> app (App (c, ei)) es
  in abs (app e.head e.args) e.n, e.t

(* type of the head of an term in rhnf representation *)
let hnf_head_types h =
  match h.head with
  | Var j -> List.nth h.alphas j
  | Const s -> find_type s
  | Rec_const s -> find_type s
  | Unk (_, (a, b)) -> rev_curryfy b a
  | _ ->
      raise (let r = hnf_to_term h in Not_normal_form (fst r, snd r))
```

A.2 β-réduction des termes et réécriture : `rewriting.ml`

```
open Defs

(* implementation of a rewriting system *)
exception Not_match

(* look if s can match t, return the substitution in that case, raise
   Not_match otherwise *)
let match_on_top s t =
  let rec ind = function
    | Abs s, Abs t -> ind (s, t)
    | App (s, t), App (u, v) -> List.rev_append (ind (s, u)) (ind (t, v))
    | Unk (i, _), c -> [i, c]
    | s, t -> if s = t then [] else raise Not_match
  in
  let rec check accu = function
    | [] -> accu
    | (i, c)::q ->
try
  if List.assoc i q = c then check accu q
  else raise Not_match
with Not_found -> check ((i, c)::accu) q
  in
  check [] (ind (s, t))

(* apply the substitution θ *)
let rec apply_subst theta = function
  | Abs s -> Abs (apply_subst theta s)
  | App (s, t) -> App (apply_subst theta s, apply_subst theta t)
  | Unk (f, _) as s -> (
      try
List.assoc f theta
```

```
      with Not_found -> s )
  | s -> s

(* rewrite the term t on top using rule s ⇝ t *)
let rec rewrite_on_top (s, s') t =
  apply_subst (match_on_top s t) s'

(* the same, with a system of rewriting rules *)
let rewrite_system_on_top sss t =
  let rec aux = function
    | [] -> raise Not_match
    | x::s ->
try
  rewrite_on_top x t
with
  Not_match -> aux s
  in
  aux sss

(* rewriting in only to the left part of the terms, using an external
   function in addition (typically the β-reduction)
*)
let rewrite_system_on_left_func f sss t =
  let rec ind args t =
    try rewrite_system_on_top sss t
    with Not_match ->
      match t with
    | Abs u -> Abs (ind [] u)
    | App (u, v) -> App (ind (v::args) u, v)
    | Rec_const s as u ->
let n = List.length (find_constructors s) in
let rec app_args c = function
  | 0, x::_ -> App(c, ind [] x)
  | n, x::q -> app_args (App(c, x)) (n-1, q)
  | _ -> failwith "function asn't enough arguments"
in
abs (app_args u (n, args)) (n+1)
    | c -> raise Not_match
  in
  let rec rew_loop m =
    try
      rew_loop (f (ind [] m))
    with Not_match -> m
  in
  rew_loop (f t)

(* the same, but applied on the whole term *)
let rewrite_system_func f sss t =
  let rec ind t =
    try rewrite_system_on_top sss t
```

```
    with Not_match ->
      match t with
    | Abs u -> Abs (ind u)
    | App (u, v) -> (
try
  App (ind u, v)
with Not_match -> App (u, ind v)
       )
    | c -> raise Not_match
  in
  let rec rew_loop m =
    try
      rew_loop (f (ind m))
    with Not_match -> m
  in
  rew_loop (f t)


(* β-reductions: *)
let rec shift i d = function
  | Var j when j >= d -> Var (j+i)
  | Abs s -> Abs (shift i (d+1) s)
  | App (s, t) -> App (shift i d s, shift i d t)
  | s -> s

let replace t =
  let rec aux i = function
    | Var j ->
if j = i then shift i 0 t
else (if j < i then Var j else Var (j-1))
    | App (u, v) -> App ((aux i u), (aux i v))
    | Abs u -> Abs (aux (i + 1) u)
    | t -> t
  in aux 0

(* β-head normal form *)
let rec beta_hnf = function
  | Abs s -> Abs (beta_hnf s)
  | App (s, t) -> (
      match beta_hnf s with
      | Abs u -> beta_hnf (replace t u)
      | u -> App (u, t) )
  | t -> t

(* β-normal form *)
let beta_nf t =
  let rec aux = function
    | Abs s -> Abs (aux s)
    | App (s, t) -> App (aux s, aux (beta_hnf t))
    | t -> t
```

```
  in aux (beta_hnf t)
```

A.3 Algorithme d'unification : unif.ml

```
open Defs
open Printer

exception F  (* failure of the unification *)

type 'a subst = string * 'a term

type 'a request = 'a head_normal_form * 'a head_normal_form

type 'a result_tree =
    'a subst list list


(* printing primitives *)
let print_constraints l =
   List.iter
    (fun a -> Queue.iter (fun (e1, e2) ->
      parse_print (fst(hnf_to_term e1));
      print_string ", ";
      pretty_print (fst(hnf_to_term e2));
      print_string "; ";) a;
      print_string " | "
    )
    l

let print_subst theta =
  List.iter
    (fun (s, t) ->
      print_Ind s;
      print_string " -> ";
      pretty_print t;
      print_string "; ";
    )
    theta

let print_terms print_func theta =
  List.iter
    (fun t ->
      print_func t;
      print_string "; ";
    )
    theta

let rec print_dec = function
```

```
  | 0 -> ()
  | n -> print_char '>'; print_dec (n-1)



(* return the rec-flexible argument of the term e (in rhnf) *)
let rec rec_flexible_arg e rules =
   match e.head with
  | Rec_const s ->
      let n = List.length (find_constructors s) in
      let r = List.nth e.args n in
      let h = term_to_hnf (r, find_base_types s) in
      rec_flexible_arg h rules
  | s -> (fst (hnf_to_term e))


(* apply the variable represented by the indixes 0 to k - 1 to c *)
let rec apply_var c = function
  | 0 -> c
  | k -> apply_var (App(c, Var (k - 1))) (k-1)


(* the SIMPL procedure *)
let simpl n rules =
  let rec simpl n accu =
    let rec dec q n ns = function
    | [], [], [] -> q
    | e1 :: l1, e2 :: l2, t::ts ->
dec (Queue.push q (term_to_hnf
   (Rewriting.rewrite_system_on_left_func Rewriting.beta_nf rules (abs e1 n),
    rev_curryfy t ns),
   term_to_hnf
   (Rewriting.rewrite_system_on_left_func Rewriting.beta_nf rules (abs e2 n),
    rev_curryfy t ns)))
  n ns (l1, l2, ts)
    | _ -> raise F
    in
    if Queue.is_empty n then accu
    else
      let (e1, e2), n' = Queue.pop n in
      match e1.head, e2.head with
      | Var i, Var j ->
  if i = j
  then
    simpl (dec n' e1.n e1.alphas
     (e1.args, e2.args, uncurryfy (hnf_head_types e1)))
      accu
  else raise F
      | Const s, Const t ->
  if s = t
```

```
  then
    simpl (dec n' e1.n e1.alphas
     (e1.args, e2.args, uncurryfy (hnf_head_types e1)))
      accu
  else raise F
      | Unk (_, _), Unk (_, _) -> simpl n' accu (* not necessary actually *)
      | Unk (_, _), _ -> simpl n' (Queue.push accu (e1, e2))
      | _, Unk (_, _) -> simpl (Queue.add_as_first n' (e2, e1)) accu (* idem *)
      | Rec_const _, _ -> simpl n' (Queue.push accu (e1, e2))
      | _, Rec_const _ -> simpl (Queue.add_as_first n' (e2, e1)) accu (* idem *)
      | _ -> raise F
  in Queue.pop (simpl n (Queue.create ()))


(* the MATCH procedure *)
(* imitation rule *)
let rec imitation cl (e1, e2) theta =
  match e1.head with
  | Unk (f, (l, t)) ->
      let rec apply_new_unk c = function
| Curry (a, b) ->
    apply_new_unk (App(c, apply_var (new_unk (l, a)) e1.p))
      b
| _ -> c
      in
      ((f, abs (apply_new_unk e2.head (hnf_head_types e2)) e1.p), cl)::theta
(* imitation in case of a rec_flexible term: none (see example 2.5) *)
  | Rec_const s ->
      theta
  | _ -> pretty_print (fst (hnf_to_term e1));
      print_newline ();
      pretty_print (fst (hnf_to_term e2));
      print_newline ();
      failwith "Not in simplified form (flex-rigid or rec_flex-rigid)"

(* projection rule *)
let proj rules cl (e1i, e2i) theta =
let rec proj cl (e1, e2) theta =
  match e1.head with
  | Unk (f, (l, beta)) ->
      let rec aux accu i = function
| [] -> accu
| ai::q -> let rec apply_new_unk c = function
    | Curry (a, b) -> apply_new_unk (App (c, apply_var (new_unk (l, a)) e1.p)) b
    | b ->
if b = beta
then ((f, abs c e1.p), cl) :: accu
else accu
in aux (apply_new_unk (Var i) ai) (i+1) q
      in aux theta 0 l
```

```
(* projection in case of a rec_flexible term *)
  | Rec_const s ->
      proj cl (term_to_hnf (rec_flexible_arg e1 rules, find_base_types s), e2) theta
  | _ ->
      failwith "Not in simplified form (flex-rigid or rec_flex-rigid)"
in
proj cl (e1i, e2i) theta


(* recursion rule *)
(* Constants are given in form 2-uples:
    - const: name of the const
    - typ: types of the other arguments of the constant, possibly polymorph
   The set of the recursiv contant we can use is given as a queue
*)
let rec_const cl (e1, e2) theta ((const, typ), nb) =
  if nb <= 0 then theta else (
  let cl = Queue.push cl ((const, typ), nb-1) in
  match e1.head with
  | Unk (f, (l, beta)) ->
      let rec apply_new_unk c k = function
| a::b -> apply_new_unk (App (c, apply_var (new_unk (l, a)) k)) k b
| [] -> c
      in
      ((f, abs (apply_new_unk (Rec_const const)
  e1.p (typ beta)) e1.p), cl)
      :: theta
(* the more complete version is :
      ((f, abs (apply_var (apply_new_unk (Rec_const const)
  e1.p (typ (rev_curryfy beta l))) e1.p) e1.p), cl)
      :: theta
*)

(* we don't use a rec_const iterativly, but to do so will be equivalent
   (see example 2.6) *)
  | Rec_const s -> theta
  | _ -> failwith "Not in simplified form (flex-rigid or rec_flex-rigid)"
 )


(* apply the substitutions ?F ↦ t to the constraints n,
   and perform rewriting rules *)
let apply_subst (f, t) n system =
  let rec aux = function
    | Abs s -> Abs (aux s)
    | App (s, t) -> App (aux s, aux t)
    | Unk (g, _) when f = g -> t
    | c -> c
  in
  let rec aux2 h =
```

```
    term_to_hnf
      (
       let e, ty = (hnf_to_term h) in
       Rewriting.rewrite_system_on_left_func Rewriting.beta_nf system (aux e),
       ty )
  in
  Queue.map
    (fun (e1, e2) -> aux2 e1, aux2 e2)
    n

let rec substs_in_term substs = function
  | Abs s -> Abs (substs_in_term substs s)
  | App (u, v) -> App (substs_in_term substs u, substs_in_term substs v)
  | Unk (j, _) as s -> (
      try
List.assoc j substs
      with Not_found -> s
     )
  | s -> s


(* unification procedure *)
let unif_consts consts rules n prof punc =
  let rec unif_consts cl (es, n) prof =
    if prof <= 0 then (nb_nodes := !nb_nodes + 1; [])
    else
      let subst =
let rec fold cl theta = function
  | [] -> theta
  | c :: q -> let _, cl = Queue.pop cl in
    fold (Queue.push cl c)
      (rec_const cl es theta c)
      q
in
let qcl = Queue.from_list cl in
fold qcl
  (proj rules qcl es (imitation qcl es []))
  cl
      in
      (match subst with [] -> [] (* no subsitution possible, we have failed *)
      | _ ->

    (List.fold_left
       (fun l ((s, t), cl) ->
 List.rev_append
   (List.rev_map
      (fun substs -> (s, substs_in_term substs t)::substs)
      (try
unif_consts (Queue.to_list cl)
  (simpl (apply_subst (s, t) (Queue.push n es) rules) rules)
```

```
  (prof-1)
      with
      | F -> []
      | Queue.Empty -> [[]]
                          (* Success, we return the empty substitution *)
      )
   )
   l
       )
       []
       subst
    ) )
  in
  try
    unif_consts consts (simpl n rules) prof
  with
  | F -> []
  | Queue.Empty -> [[]]


(* get the final substitution from the search tree *)
let final_subst s t =
  List.map (fun l -> Rewriting.beta_nf (List.assoc s l) ) t
```

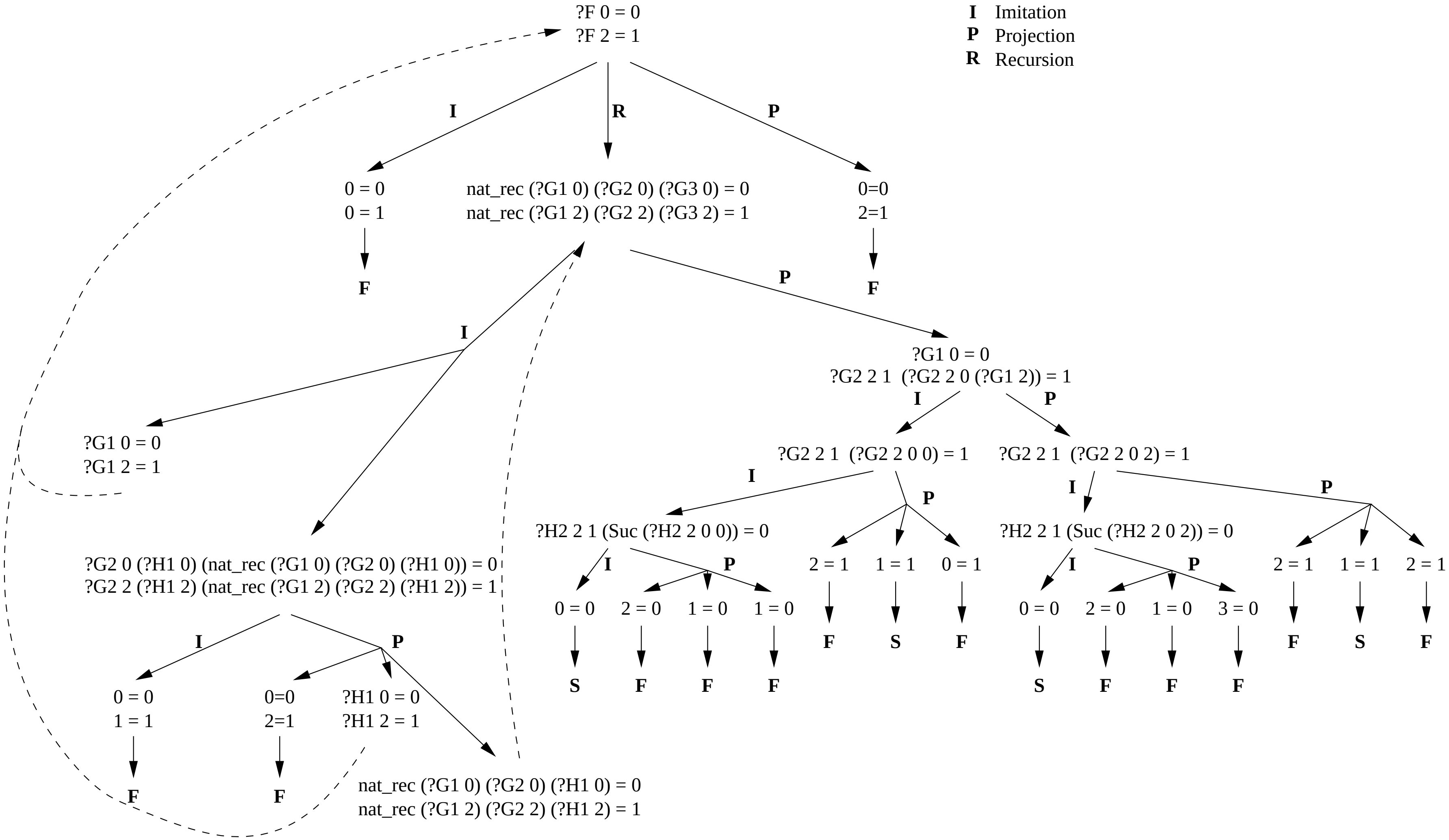

FIG. 2 – Arbre de recherche pour la fonction prédécesseur (cf section 2.4 p.14)