Etude de l'extension de la notion d'abstraction du lambda-calcul.

Guillaume Malod

GRAME
Laboratoire de recherche en informatique musicale
Rapport technique TR-020809

Etude de l'extension de la notion d'abstraction du lambda-calcul.

Guillaume Malod
<malod@igd.univ-lyon1.fr>

Grame
Centre National de Création Musicale
9, rue du Garet BP 1185
FR - 69202 Lyon Cedex 01

Introduction

Le programme *Elody* repose principalement sur les concepts du λ-calcul, notamment les notions d'abstraction et d'application. L'abstraction classique a été étendue dans [**3**] afin de permettre des manipulations plus puissantes de λ-termes. Cette extension reposait sur une relation de généralité définie entre deux termes.

Nous étudions ici plus précisément comment définir une relation de généralité entre deux termes, dans un but un peu différent du précédent: nous souhaitons pouvoir représenter des ensembles de termes, avec comme intuition qu'un terme représente l'ensemble de tous les termes moins généraux que lui.

On cherche donc d'abord à obtenir une définition précise à partir de cette idée intuitive, en envisageant différentes définitions possibles, et on eétudie les conséquences du choix qui semble le plus logique. La définition adoptée permet de définir une relation d'ordre et une relation d'équivalence qu'il convient de caractériser. On montre ensuite qu'il est possible de définir la borne supérieure et la borne inférieure d'un ensemble fini de termes, ce qui correspond à l'union et à l'intersection pour les ensembles de termes.

On obtient finalement un cadre théorique assez net autour de la notion de généralité, qui permet d'envisager d'autres développements en manipulant des ensembles de termes. A titre d'exemple, on présente les algorithmes permettant d'effectuer les opérations sur les termes discutées précédemment, et enfin le code source commenté d'une implémentation simple en Caml.

Généralisation

1. Ensembles engendrés

Une abstraction est un terme du λ-calcul de la forme $u = \lambda x_1 \ldots \lambda x_m.u'$ avec $m \geq 1$ et u' un terme ne commençant pas par une abstraction. Les variables x_i sont les *variables de tête* de u et u' le *corps*.
Les termes qui ne sont pas des abstractions sont appelés *termes pleins*: le terme $(\lambda x \lambda y.(x\ y)\ \ a)$ (a est une variable libre) est un exemple de terme plein, même si après une β-réduction à gauche on obtient une abstraction. C'est sur ces termes que nous portons d'abord notre attention. On pourrait même ne considérer que des *termes de base*, dans un langage avec des constructeurs additionnels: des termes qui ne comportent plus d'abstractions ou d'applications et ne sont donc que la représentation extensionnelle d'une donnée.
Dans un premier temps on s'intéresse à la représentation et la manipulation d'ensembles de termes pleins. Une façon simple de représenter un ensemble de termes pleins c'est de considérer l'ensemble des termes obtenus à partir de u en effectuant, dans le corps de u, des substitutions sur les variables de tête, ce qui s'énonce plus précisément dans la définition suivante.

Définition 1. Soit $u = \lambda x_1 \ldots \lambda x_m.u'$ ($m \geq 0$), où u' n'est pas une abstraction, on appelle *ensemble engendré* par u et on note $G(u)$ l'ensemble des termes pleins de la forme $u'[x_1/a_1, \ldots, x_m/a_m]$, où les a_i sont des termes.

Il y a deux cas possibles dans cette définition: soit u est vraiment une abstraction ($m \geq 1$) et on obtient un ensemble $G(u)$ qui contient les termes engendrés par u mais pas le terme u lui-même, soit u était en fait un terme plein ($m = 0$) et l'ensemble engendré est réduit à $\{u\}$.
Une abstraction est ainsi en quelque sorte un terme "creux", les variables de tête donnent le nom des cases que l'on pourra remplir pour obtenir un terme plein. Remarquons que l'on souhaite obtenir un terme plein, donc on remplit toutes les cases de u' afin d'obtenir un terme qui n'est plus une abstraction.

La substitution utilisée ici est celle décrite dans [**2**] (page 7), qui garantit qu'il n'y aura pas de capture de variable en renommant au préalable les variables liées de u'. Ce renommage conduit à un terme α-équivalent à u', ce qui ne pose pas de problème car on manie en fait des classes d'équivalence de λ-termes. Un terme u doit

être vu comme un représentant de sa classe, et il peut donc être remplacé par un autre représentant si besoin. De même, dans la suite, le symbole $=$ entre deux termes signifie qu'ils sont α-équivalents, qu'ils représentent la même classe. égaux

2. Généralisation

Maintenant qu'on a une façon de représenter les ensembles on aimerait bien pouvoir utiliser ces représentations. La première chose qu'on peut faire quand on a deux ensembles c'est les comparer, i.e. déterminer si l'un est inclus dans l'autre.
Si on a une abstraction u et un terme plein v, $G(v) \subseteq G(u)$ ssi $v \in G(u)$, donc ssi v s'obtient en remplaçant les variables de tête de u, dans le corps de u, par des termes. On sait donc établir la relation d'inclusion entre deux ensembles engendrés l'un par une abstraction, l'autre par un terme plein, et cette inclusion correspond à une relation sur les termes. Dans le cas cité, on peut dire que u est plus général que v au sens où v s'obtient par substitutions dans le corps de u. Remarquons que dans ce cas v est β-équivalent à $(u\ \ a_1\ \ \ldots\ \ a_m)$, en utilisant les notations de la définition.
Ce que l'on veut ensuite c'est pouvoir comparer deux ensembles engendrés quelconques, et non plus juste un ensemble engendré par une abstraction et un singleton correspondant à un terme plein. Pour cela il nous faut généraliser la relation obtenue pour pouvoir comparer deux termes quelconques, et en particulier deux abstractions. Nous suivons ici le parti pris de vouloir conserver l'équivalence avec les propriétés des ensembles engendrés, ce qui nous amène a la définition naturelle suivante.[1]

Définition 2. Soit u et v deux termes, on dit que u est *plus général* que v et on note $u \geq v$ si $G(v) \subseteq G(u)$.

On dira aussi que u est une *généralisation* de v ou que v est *moins général* que u.

3. Premières propriétés de la relation de généralité

3.1. Relation d'ordre.

On a noté notre relation $\leq$, donc on aimerait bien que ce soit une relation d'ordre (la notion de généralisation définie dans [**3**] était bien une relation d'ordre). En fait ce n'est pas tout à fait le cas, mais presque.

[1]On peut aussi faire le choix de ne plus s'attacher aux ensembles mais de vouloir conserver le fait que si un terme u est plus général qu'un terme v, il existe des termes $a_1, \ldots, a_m$ tels que v soit β-équivalent à $(u\ \ a_1\ \ \ldots\ \ a_m)$. Cet aspect sera brièment étudié à la section 5.1.

PROPOSITION 1. $\leq$ *est un pré-ordre.*

Un pré-ordre est une relation réflexive et transitive. La relation de généralité satisfait clairement ces deux conditions, cela provient de la définition à partir de l'inclusion sur les ensembles.
Elle n'est pas un ordre car il manque l'anti-symétrie : on peut avoir $e \leq f$ et $f \leq e$ sans que e et f ne soient égaux a α-conversion près. C'est le cas par exemple des termes t_1 et t_2 suivants.

$$\begin{aligned} t_1 &= \lambda x \lambda y.(x\ y) \\ t_2 &= \lambda x \lambda y.(y\ x) \end{aligned}$$

A partir d'un pré-ordre on peut toujours récupérer une relation d'ordre en définissant une relation d'équivalence associée.

DÉFINITION 3. On dira que e et f sont *G-équivalents* (on notera $e \sim f$) si on a à la fois $e \leq f$ et $f \leq e$.

Il est encore une fois simple de vérifier que $e \sim f$ est une relation d'équivalence, à savoir réflexive, transitive et symétrique. En revenant aux ensembles, u et v sont G-équivalents ssi ils engendrent le même ensemble de termes pleins. C'est bien le cas pour t_1 et t_2.

Cet ordre admet un maximum qui est l'ensemble de tous les termes pleins, défini par l'identité. Il admet un nombre infini d'éléments minimaux, c'est-à-dire des éléments qui n'ont pas d'élément strictement inférieur. Ces éléments minimaux sont exactement les ensembles définis par un terme plein t, ou par un terme G-équivalent à un terme plein, donc de la forme $\lambda x_1 \ldots \lambda x_n.t$ (cf. section 3.3).

3.2. Caractérisation de la relation de généralisation.

La définition que nous avons utilisée jusqu'à présent provenait directement de la propriété fondamentale qu'on souhaitait avoir pour les ensembles engendrés. C'est une définition extérieure aux termes u et v que l'on compare. La propriété suivante donne une caractérisation de la relation de généralisation qui s'appuie plus directement sur la structure des termes que l'on compare. Elle sera utile pour la suite et pour implémenter un algorithme.

PROPOSITION 2. *Soit deux termes* $u = \lambda x_1 \ldots \lambda x_m.u'$ *et* $v = \lambda y_1 \ldots \lambda y_n.v'$ *où* u' *et* v' *ne sont pas des abstractions, les propositions suivantes sont équivalentes:*

(i) $u \geq v$.

(ii) *pour tous termes* $b_1, \ldots, b_n$, *il existe des termes* $a_1, \ldots, a_m$ *tels que:*

$$u'[x_1/a_1, \ldots, x_m/a_m] = v'[y_1/b_1, \ldots, y_n/b_n].$$

(iii) *il existe des termes* $t_1, \ldots, t_n$, *tels que:*

$$u'[x_1/t_1, \ldots, x_m/t_m] = v'.$$

NOTATION. Dans la preuve de cette proposition et dans la suite, nous écrirons souvent $u[\bar{x}/\bar{a}]$ à la place de $u[x_1/a_1, \ldots, x_m/a_m]$, lorsqu'il n'y a pas d'ambiguïté sur les indices.

PREUVE.

(i) $\Rightarrow$ (ii). On suppose que $u \geq v$. Soit $b_1, \ldots, b_n$ des termes, alors $v'[\bar{y}/\bar{b}]$ est un terme engendré par v, donc appartenant à $G(v)$. Comme $G(v) \subseteq G(u)$ ce terme est aussi engendré par u, et il existe des termes $a_1, \ldots, a_m$ tels que $u'[\bar{x}/\bar{a}] = v'[\bar{y}/\bar{b}]$.

(ii) $\Rightarrow$ (iii). On prend les variables y_i comme termes b_i. Cela nous donne directement l'existence des termes t_i tels que $u'[\bar{x}/\bar{t}] = v'[\bar{y}/\bar{y}]$, donc $u'[\bar{x}/\bar{t}] = v'$.

(iii) $\Rightarrow$ (i). On veut montrer que $G(v) \subseteq G(u)$. Soit donc un terme $w = v'[\bar{y}/\bar{b}]$ engendré par v. On a alors $w = u'[\bar{x}/\bar{t}][\bar{y}/\bar{b}]$, et enfin $w = u'[\bar{x}/\bar{t'}]$, avec $t'_i = t_i[\bar{y}/\bar{b}]$ (cf. [**2**], lemme 10). Donc w est engendré par u et w appartient à u. □

3.3. Caractérisation de la relation d'équivalence.

Une variable de tête d'une abstraction u est dite *inutile* si elle n'apparaît pas libre dans le corps de u. Par exemple z est inutile dans $t_3 = \lambda x \lambda y \lambda z.(x\ y)$, et t_3 engendre les mêmes termes pleins que t_1. Il est facile de voir que cette propriété est vraie en général, à savoir qu'on peut toujours enlever les variables de tête inutiles pour obtenir un terme G-équivalent.

LEMME 1. *Soit* $u = \lambda x_1 \ldots \lambda x_m.u'$ *et* i *tel que* x_i *ne soit pas une variable libre de* u'. *Alors* u *et* $\lambda x_1 \ldots \lambda x_{i-1} \lambda x_{i+1} \ldots \lambda x_m.u'$ *sont G-équivalents.*

On a vu par ailleurs que les termes t_1 et t_2, qui n'ont pas de variables de tête inutiles, sont G-équivalent. On remarque sur cet exemple qu'ils sont en fait égaux à permutation des variables de tête près. Cette propriété aussi est vraie en général.

LEMME 2. *Soit deux termes* $u = \lambda x_1 \ldots \lambda x_m.u'$ *et* $v = \lambda y_1 \ldots \lambda y_n.v'$ *où* u' *et* v' *ne sont pas des abstractions, tels que chaque* x_i *soit libre dans* u' *et chaque* y_j *soit libre dans* v'. *Alors* u *et* v *sont G-équivalents ssi* $m{=}n$ *et il existe une permutation* σ *de* $\{1, \ldots, n\}$ *telle que* $u'[x_1/y_{\sigma(1)}, \ldots, x_n/y_{\sigma(n)}] = v'$.

PREUVE. On suppose que u et v sont G-équivalents, et que après renommage éventuel, les variables y_i ne sont pas libres dans u' et les variables x_i ne sont pas libres dans v'. On utilise la caractérisation de la relation de généralité qui affirme qu'il existe des termes $a_1, \ldots, a_m$ et des termes $b_1, \ldots, b_n$ tels que: $u'[\bar{x}/\bar{a}] = v'$ et $v'[\bar{y}/\bar{b}] = u'$.
On en déduit donc que $u'[\bar{x}/\bar{a}][\bar{y}/\bar{b}] = u'$. On applique le lemme10 de[**2**], ce qui nous donne: $u'[\bar{x}/\bar{t}] = u'$, avec $t_i = a_i[\bar{y}/\bar{b}]$. Par définition de l'$\alpha$-équivalence, cette égalité ne peut être satisfaite que si $t_i = x_i$ pour tout i. a_i ne peut être égal à x_i car $u'[\bar{x}/\bar{a}] = v'$, les x_i ne sont pas libres dans v' mais sont libres dans u'. Donc chaque x_i dans u' est en fait substitué par une variable y_j pour obtenir v'.
De plus, chaque y_j est substitué à au moins un x_i, car chaque y_j apparaît libre dans v'. On en déduit que $m \geq n$ et qu'il existe une surjection σ de $\{1, \ldots, m\}$ dans $\{1, \ldots, n\}$ telle que $u'[x_1/y_{\sigma(1)}, \ldots, x_n/y_{\sigma(n)}] = v'$.

En faisant le même raisonnement à partir de $v'[\bar{y}/\bar{b}][\bar{x}/\bar{a}] = v'$, on obtient $m \leq n$, soit $m = n$. L'application σ est donc une surjection de l'ensemble $\{i, \ldots, n\}$ dans lui-même et donc une bijection.

Réciproquement, on suppose qu'on a u et v qui satisfont les conditions du lemme. Soit t un terme engendré par v, il existe des termes b_i tels que $t = v'[\bar{y}/\bar{b}]$. Donc $u'[x_1/b_{\sigma(1)}, \ldots, x_n/b_{\sigma(n)}] = t$ et t est aussi engendré par u. On peut procéder dans l'autre sens, en utilisant la bijection inverse τ pour montrer que tout terme engendré par u est aussi engendré par v, et finalement que les termes u et v sont G-équivalents. □

L'association des deux lemmes précédents permet d'obtenir facilement une caractérisation de la relation d'équivalence: une fois qu'on a ôté les variables inutiles de chaque côté, la seule solution qui reste pour que u et v soient G-équivalents est qu'ils soient égaux à permutation des variables de tête près, ce qui nous donne la proposition suivante.

PROPOSITION 3. *Soit deux termes $u = \lambda x_1 \ldots \lambda x_m.u'$ et $v = \lambda y_1 \ldots \lambda y_n.v'$ où u' et v' ne sont pas des abstractions, les propositions suivantes sont équivalentes:*

(i) $u \sim v$.
(ii) *soit I l'ensemble des indices i tels que $x_i \in FV(u')$ et J l'ensemble des indices j tels que $y_j \in FV(v')$, il existe une bijection σ entre I et J telle que:*

$$u'[x_i/y_{\sigma(i)},\ i \in I] = v'.$$

4. Propriétés de la relation de généralité en tant que relation d'ordre

La relation d'ordre que nous avons introduite n'est pas totale, car des termes comme $\lambda x \lambda y.(x\ y)$ et a ne sont pas comparables. Cependant elle présente d'autres propriétés qui structurent l'ensemble des (classes d'équivalence de) termes.

4.1. Borne inférieure.

On a un ordre, donc on peut se demander si deux termes quelconques u et v admettent toujours une borne inférieure, c'est-à-dire un terme w tel que w soit plus général que u et v, et si un autre terme t est plus général que u et v alors t est plus général que w.

Pour les besoins de la démonstration qui va suivre nous introduisons une nouvelle relation sur les couples composés d'un uple de variables et d'un terme. Cette relation doit plutôt être considérée comme une abbréviation afin d'alléger les notations dans la démonstration. Elle correspond à la nécessité de faire une démonstration par induction sur un λ-terme quelconque.

Définition 4. Soit u et v deux termes, qui peuvent être des abstractions, $x_1, \ldots, x_m$, $y_1, \ldots, y_n$ des variables, on dira que $(x_1, \ldots, x_m; u)$ engendre $(y_1, \ldots, y_n; v)$, et on notera $((x_1, \ldots, x_m; u) \preccurlyeq (y_1, \ldots, y_n; v)$, s'il existe des termes $a_1, \ldots, a_m$ tels que $u[\bar{x}/\bar{a}] = v$.

Lemme 3. *Soit $u_0 = (x_1, \ldots, x_m; u)$ et $v_0 = (y_1, \ldots, y_n; v)$ deux couples et $a_1, \ldots, a_m$, $b_1, \ldots, b_n$ des termes tels que $u[\bar{x}/\bar{a}] = v[\bar{y}/\bar{b}]$. Alors il existe w et des variables $z_1, \ldots, z_p$ tels que $w_0 = (z_1, \ldots, z_p; w)$ soit engendré par u_0 et par v_0, et tel que si un couple t_0 est engendré par u_0 et par v_0, alors t_0 est engendré par w_0.*

Preuve. Par induction sur u. On appelle $(*)$ l'hypothèse $u[\bar{x}/\bar{a}] = v[\bar{y}/\bar{b}]$.

Si u est une variable, c'est soit une des variables x_i, soit une variable différente.

- Si c'est une variable x différente des x_i, alors $(*)$ devient $u = v[\bar{y}/\bar{b}]$, donc $u_0 \preccurlyeq v_0$, et on pose $w_0 = u_0$.
- Si c'est une variable x_i, alors $u[x_i/v] = v$, donc $u_0 \succcurlyeq v_0$, et on pose $w_0 = v_0$.

Si u est une abstraction: $u = \lambda x.u'$, et $(*)$ devient $\lambda x.u'[\bar{x}/\bar{a}] = v[\bar{y}/\bar{b}]$. Pour que cette égalité soit satisfaite, il faut ou bien que v soit une des variables y_j, et que le terme b_j correspondant soit adéquat, ou bien que v soit une abstraction aussi.

- Si v est une variable v_j, on se retrouve dans le cas où $u_0 \preccurlyeq v_0$, et on pose $w_0 = u_0$.

- Si v est une abstraction $\lambda x.v'$, on obtient $u'[\bar{x}/\bar{a}] = v'[\bar{y}/\bar{b}]$, c'est-à-dire qu'on a les hypothèses d'induction pour $(\bar{x}; u')$ et $(\bar{y}; v')$. On trouve donc $(\bar{z}; w')$ qui a les propriétés voulues pour $(\bar{x}; u')$ et $(\bar{y}; v')$. En posant ensuite $w = \lambda x.w'$ et $w_0 = (\bar{z}; w)$, on obtient un couple ayant les propriétés voulues pour u_0 et v_0. En effet, si $t_0 = (t_1, \ldots, t_k; t)$ est engendré par u_0 et v_0, t est nécessairement une abstraction $t = \lambda x.t'$ et $(\bar{t}; t')$ est engendré par $(\bar{x}; u')$ et $(\bar{y}; v')$, donc par $(\bar{z}; w')$. Enfin on en déduit que t_0 est engendré par w_0.

Si u est une application, on peut encore discuter: si v est une variable y_j, $u_0 \leq v_0$ et on prend u_0, sinon v est aussi une application et le raisonnement est similaire au cas de l'abstraction ci-dessus. □

PROPOSITION 4. *Soit u et v deux termes tels qu'il existe un terme t satisfaisant $t \leq u$ et $t \leq v$, alors l'ensemble $\{u, v\}$ admet une borne inférieure, que l'on notera $inf(u, v)$.*

PREUVE. Soit $u = \lambda x_1 \ldots \lambda x_m.u'$ et $v = \lambda y_1 \ldots \lambda y_n.v'$, où u' et v' ne sont pas des abstractions, les hypothèses impliquent l'existence de termes $\bar{a}$ et $\bar{b}$ tels que $u'[\bar{x}/\bar{a}] = t' = v'[\bar{y}/\bar{b}]$. On se retrouve donc dans les conditions d'application du lemme précédent. On obtient donc $w = \lambda z_1 \ldots \lambda z_p.w'$ tel que $(\bar{z}; w')$ soit la borne inférieure (au sens de l'ordre de la définition 4) de $(\bar{x}; u')$ et $(\bar{y}; v')$.
Soit maintenant $e = \lambda e_1 \ldots \lambda e_k.e'$ un terme tel que $e \leq u$ et $e \leq v$. Cela veut dire qu'il existe des termes $\bar{a}'$ et $\bar{b}'$ tels que $e' = u'[\bar{x}/\bar{a}']$ et $e' = v'[\bar{y}/\bar{b}']$, donc $(\bar{e}; e')$ est inférieur à $(\bar{x}; u')$ et $(\bar{y}; v')$. Alors $(\bar{e}; e')$ est inférieur à $(\bar{z}; w')$, et $e \leq w$. □

PROPOSITION 5. *L'opération "borne inférieure" est commutative: $inf(u, v)$ existe ssi $inf(u, v)$ existe et dans ce cas ces termes sont G-équivalents.*
L'opération "borne inférieure" est associative: soit u, v et w ayant un minorant commun, alors les termes $inf(inf(u, v), w)$ et $inf(u, inf(v, w))$ existent et sont G-équivalents.

PREUVE. La commutativité découle directement de la définition.
Soit u, v et w des termes ayant un minorant commun t. On pose $t_1 = \inf(\inf(u, v), w)$, $t_2 = \inf(u, \inf(v, w))$ et $t_3 = \inf(\inf(u, v), \inf(v, w))$. Il est facile de voir que $t_1 \geq t_3$. Réciproquement, montrons que $t_3 \geq t_1$. t_1 est inférieur à $\inf(u, v)$. t_1 est inférieur à v et à w, donc t_1 est inférieur à $\inf(v, w)$. Ainsi t_1 est inférieur aux deux termes dont t_3 est une borne inférieure, donc par définition de la borne inférieure, t_1 est inférieur à t_3, et enfin G-équivalent à t_3.
On peut montrer de même que t_2 et t_3 sont G-équivalents, donc t_1 et t_2 sont G-équivalents. □

Si $u_1, \ldots, u_n$ $(n \geq 1)$ sont des termes admettant un minorant commun, on peut ainsi parler de la borne inférieure de $\{u_1, \ldots, u_n\}$.

4.2. Borne supérieure.

On obtient aussi l'existence de la borne supérieure de deux termes, mais il faut d'abord démontrer un petit lemme technique qui servira plusieurs fois.

LEMME 4. *Soit u un terme, l'ensemble des classes d'équivalences de termes plus généraux que u est fini.*

PREUVE. Soit $u_0 = \lambda x_1 \dots \lambda x_m.u$. On procède par induction sur u.

Si u est une variable, il y a deux cas selon qu'elle appartient aux variables de t^te de u_0 ou non.

- Si u est une variable x distincte des x_i, alors si $t_0 \geq u_0$, avec $t_0 = \lambda z_1 \dots \lambda z_k.t$, il existe des termes $a_1, \dots, a_k$ tels que $x = t[\bar{z}/\bar{a}]$. Cela nous donne deux choix possibles: soit $t = x$ et t_0 est dans la classe de u_0, soit t est l'une des variables z_i, et t_0 est dans la classe de l'identité.
- Si u est une variable x_i, alors u_0 est dans la classe de l'identité et n'a pas de majorant strict.

Si u est une abstraction, $u = \lambda x.u'$ et pour que t_0 soit plus général que u_0 il faut soit que t soit une des variables t_i, auquel cas on a la classe de l'identité, soit que t soit aussi une abstraction $\lambda x.t'$, et on peut appliquer l'hypothèse d'induction à t', ce qui nous donne un nombre fini de possiblités pour la classe de t' et donc celle de t_0.

Si u est une application, on procède de même, $u = (u_1\ u_2)$, et soit la classe de t_0 est celle de l'identité, soit $t = (t_1\ t_2)$, et le nombre de classes possibles pour t est le produit du nombre de classes obtenues pour t_1 et t_2, lorsqu'on applique l'hypothèse d'induction à t_1 et u_1 d'une part et à t_2 et u_2 d'autre part. □

PROPOSITION 6. *Soit u et v deux termes, alors l'ensemble $\{u, v\}$ admet une borne supérieure, que l'on notera $sup(u, v)$.*

PREUVE. Soit $u = \lambda x_1 \dots \lambda x_m.u'$ et $v = \lambda y_1 \dots \lambda y_n.v'$, où u' et v' ne sont pas des abstractions. L'identité étant plus générale que tous les termes, il existe toujours au moins un terme qui soit une généralisation commune à u et à v. Soit maintenant deux termes t_1 et t_2, tous deux des généralisations communes à u et v. Alors $\inf(t_1, t_2)$ existe, car $t_1 \geq u$ et $t_2 \geq u$, et $\inf(t_1, t_2)$ est aussi une majoration commune à u et v, qui est inférieure à t_1 et à t_2. Il suffit alors de considérer l'ensemble (fini d'après le lemme 4) des classes de généralisations communes à u et à v et d'en prendre la borne inférieure w: cet ensemble est non-vide (car il y a la classe de l'identité) et la borne inférieure existe (car u par exemple est un minorant commun de toutes ces classes). w est bien une généralisation commune à u et à v et si t en est une autre, t est plus général que w. □

La propriété suivante est l'analogue de celle pour la borne inférieure et se démontre de la même manière.

PROPOSITION 7. *L'opération "borne supérieure" est commutative: $sup(u, v)$ et $sup(u, v)$ sont G-équivalents.*
L'opération "borne supérieure" est associative: soit u, v et w des termes, alors les termes $sup(sup(u, v), w)$ et $sup(u, sup(v, w))$ existent et sont G-équivalents.

On peut donc aussi définir la borne supérieure d'un ensemble fini non-vide de termes.

4.3. Caractérisation des ensembles engendrés.

Revenons maintenant à nos ensembles de termes pleins. On a pris comme fondement la notion d'ensemble engendré par une abstraction. Il est clair que qu'un ensemble de termes pleins quelconque n'est pas forcément représentable par un terme. Il est donc intéressant de se demander si on peut reconnaître les ensembles engendrés. On commence pour cela par introduire d'autres ensembles, les ensembles définissables, qui sont assez facilement caractérisables.

DÉFINITION 5. Soit u un terme, on appelle *ensemble définissable* (par u) et on note $D(u)$ l'ensemble des termes v tels que $v \leq u$.

PROPOSITION 8. *Soit D un ensemble de termes, D est définissable ssi:*

- *D est non-vide.*
- *pour tout terme u appartenant à D, pour tout terme v, si $v \leq u$ alors v appartient à u.*
- *pour tous termes u et v appartenant à D, la borne supérieure de u et v appartient à D.*

PREUVE. Soit D un ensemble définissable: $D = D(u)$. Alors $u \in D$, donc D est non-vide. Soit $v \in D$ et $w \leq v$, par transitivité $w \leq u$ et donc $w \in D$. Soit v et w deux termes appartenant à D, alors on a $v \leq u$ et $w \leq u$. Par définition de la borne supérieure, $\sup(v, w) \leq u$ et donc $\sup(v, w)$ appartient à D.

Soit D un ensemble satisfaisant les trois conditions de la propriété. D est non-vide, donc on considère un terme u_0 dans D. Ou bien tous les termes de D sont inférieurs à u_0 et on s'arrête, ou bien il existe un terme v_0 de D qui n'est pas inférieur à u_0. On pose dans ce cas $u_1 = \sup(u_0, v_0)$, si bien que u_1 est strictement plus grand que u_0. On poursuit le procédé pour construire une séquence strictement croissante de termes plus généraux que u_0. D'après le lemme 4, ce processus doit s'arrêter et on obtient donc un terme u tel que tous les termes de D soient inférieurs à u. De plus si

w est inférieur à u, w appartient à D. Donc D est exactement l'ensemble des termes inférieurs à u, c'est-à-dire $D(u)$. □

Il est clair que l'on a une bijection entre les classes d'équivalence de termes et les ensembles définissables d'une part, et entre les classes de termes et les ensembles engendrés d'autre part, donc on a une bijection entre les ensembles engendrés et les ensembles définissables. Ansi, à chaque ensemble engendré G correspond un unique ensemble définissable D et G est l'ensemble des termes pleins appartenant à D.
En fait on peut obtenir une caractérisation plus précise des ensembles engendrés, qui se montre de manière analogue à la proposition précédente.

PROPOSITION 9. *Soit G un ensemble de termes pleins. G est un ensemble engendré ssi pour tous termes $a_1, \ldots, a_n$ appartenant à G, tout terme plein inférieur à* $\sup\{a_1, \ldots, a_n\}$ *appartient à G.*

4.4. Opérations sur les ensembles engendrés.

DÉFINITION 6. Soit G et H deux ensembles engendrés, on appelle *union engendrée* de G et H et on note $G \sqcup H$ le plus petit ensemble engendré K tel que $G \subseteq K$ et $H \subseteq K$.

PREUVE. C'est une définition, mais il faut tout de même prouver qu'elle a un sens, i.e. que pour tous G et H ensembles engendrés, il existe un unique K plus petit ensemble engendré les contenant tous deux. C'est une conséquence directe de la proposition 6. G et H sont respectivement engendrés par u et v. Soit $K = G(\sup(u, v))$, les propriétés qui font de K l'union engendrée de G et H découlent de la définition de la borne supérieure. □

L'union engendrée de deux ensembles n'est pas en général égale à l'union ensembliste de ces deux ensembles, parce qu'il n'existe pas forcément de terme qui engendre l'union ensembliste. Si on prend par exemple $u = \lambda x.(a\ x)$ et $v = \lambda x.(x\ b)$, où a et b sont des variables libres, $G(u)$ est l'ensemble des termes de la forme $(a\ c)$ pour c un terme, et $G(v)$ est l'ensemble des termes de la forme $(c\ b)$, où c est un terme. Par contre l'union engendrée est l'ensemble $G(\lambda x \lambda y.(x\ y))$ qui est l'ensemble des termes de la forme $(c\ d)$, où c et d sont des termes quelconques.

DÉFINITION 7. Soit G et H deux ensembles engendrés d'intersection non-vide, on appelle *intersection engendrée* de G et H et on note $G \sqcap H$ le plus petit ensemble engendré K tel que $G \subseteq K$ et $H \subseteq K$.

PREUVE. La preuve correspond exactement à celle donnée pour l'union. Si les deux ensembles considérés sont d'intersection non-vide, alors les termes u et v qui les engendrent admettent une borne inférieure, qui engendre l'intersection engendrée de nos ensembles. □

Contrairement au cas de l'union engendrée, l'intersection engendrée, quand elle existe, est égale à l'intersection ensembliste.

PROPOSITION 10. *Soit G et H deux ensembles engendrés d'intersection non-vide, alors $G \sqcap H = G \cap H$.*

PREUVE. On va montrer que $G \cap H$ est un ensemble engendré, en utilisant la caractérisation des ensembles définissables donnée en 8.
Soit u et v des termes engendrant G et H respectivement. Alors on montre d'abord que l'intersection D des ensembles définissables $D(u)$ et $D(v)$ est un ensemble définissable. D est non-vide par hypothèse. Soit w appartenant à D, et soit t un terme moins général que w. Alors w appartient à $D(u)$, donc par propriété t aussi, et w appartient à $D(v)$ donc par propriété t aussi. Donc t appartient à D. Soit maintenant deux termes w_1 et w_2 appartenant à D. Ils appartiennent tous deux à $D(u)$ donc leur borne supérieure aussi. De même la borne supérieure de w_1 et w_2 appartient à $D(v)$. Donc elle appartient à D. D satisfait donc les propriétés de la caractérisation des ensembles définissables.
$G \cap H$ est l'ensemble des termes pleins correspondant à l'ensemble définissable D, donc $G \cap H$ est un ensemble engendré, et c'est le plus grand qui soit contenu dans G et H, donc c'est l'intersection engendrée de G et H. □

4.5. Structure de treillis.

Un treillis c'est un ensemble partiellement ordonné où tout sous-ensemble fini non-vide admet une borne supérieure et une borne inférieure. Nous sommes exactement dans ce cas à part qu'un sous-ensemble fini non-vide n'a pas forcément une borne inférieure, ce qui correspond en fait au cas où les ensembles engendrés sont d'intersection vide. Cela se produit parce qu'on n'a pas de terme qui engendre l'ensemble vide, un ensemble engendré est toujours non-vide. Par contre on a un terme qui majore tous les termes, l'identité, autrement dit un terme qui engendre l'ensemble de tous les termes pleins, et qui est donc un maximum pour notre ordre. Si on avait un terme représentant l'ensemble vide, on aurait d'une part l'existence de la borne inférieure dans tous les cas, d'autre part un minimum pour notre ordre, ce qui nous donnerait un treillis borné.
Cependant il est difficile de voir quel terme pourrait être un minimum et représenter l'ensemble vide, car il faudrait trouver un terme plus petit que tout autre terme.

5. Autres notions de généralisation

5.1. Notions plus strictes.

Après avoir étudié la relation naturelle que l'on est amené à définir si l'on s'intéresse aux ensembles engendrés, nous pouvons essayer de voir les autres choix possibles pour

une définition de la relation de généralisation. Reprenons la caractérisation de notre relation donnée par la proposition 2.

On considère deux termes $u = \lambda x_1 \ldots \lambda x_m.u'$ et $v = \lambda y_1 \ldots \lambda y_m.v'$ où u' et v' ne sont pas des abstractions, $u \geq v$ ssi il existe des termes $t_1, \ldots, t_n$ tels que:

$$u'[x_1/t_1, \ldots, x_m/t_m] = v'.$$

Cette définition, même si elle coïncide avec celle provenant des ensembles engendrés, ne fait plus référence à ces ensembles, mais seulement à la structure de u et v. On pourrait donc partir de cette définition, et la modifier, pour obtenir d'autres relations de généralisation. La principale chose que l'on peut modifier dans cette définition c'est d'ajouter des contraintes sur le type de termes $t_1, \ldots, t_n$ que l'on peut substituer aux variables de u'. Voici quelques conditions que l'on pourrait imposer.

(i) *Uniques.* On exige que les termes $t_1, \ldots, t_n$ soient uniques. Cela revient exactement à dire qu'une abstraction qui comporte au moins une variable inutile n'est jamais plus générale qu'un autre terme. De tels termes étaient appelés *expressions constantes* dans [**3**]. Avec cette condition, deux termes qui engendrent le même ensemble ne sont plus forcément G-équivalents ($\lambda x \lambda y.x$ et $\lambda x.x$ par exemple).

(ii) *Propres.* On exige que les termes $t_1, \ldots, t_n$ soient des sous-termes propres de v'. Cela empêche l'identité d'être plus générale que tous les termes. Avec cette condition on perd l'existence d'une borne supérieure dans tous les cas.

(iii) *Ordonnés.* On exige que $m \geq n$ et que les termes substitués soient de la forme suivante:
- pour $1 \leq i \leq m - n$, t_i est un sous-terme de v'.
- pour $m - n + 1 \leq i \leq m$, t_i est la variable $y_{i-(m-n)}$.

C'est la condition la moins naturelle, mais elle a pour conséquence que si u est plus général que v, alors il existe des termes $t_1, \ldots, t_{m-n}$ tels que $(u \;\; t_1 \;\; \ldots \;\; t_{m-n})$ soit β-équivalent à v. Cette condition est très stricte, elle aussi empêche des termes qui engendrent le même ensemble d'être G-équivalents (par exemple $\lambda x \lambda y.(x\ y)$ et $\lambda y \lambda x.(x\ y)$).

Ces conditions, même si elles sont relativement naturelles, n'ont pas été choisies au hasard: si on les prend toutes on obtient la notion de généralisation définie dans [**3**]. Elles ne permettent pas d'avoir une partie des propriétés énoncées précédemment, mais elles définissent une relation d'ordre sans avoir à passer par une équivalence sur les termes. Les conditions (i) et (ii) notamment font qu'on se rapproche de la structure syntaxique du λ-calcul. La relation qu'elles définissent est plus forte au sens où si u est plus général que v pour cette relation, alors c'est aussi vrai pour la relation introduite avec les ensembles engendrés, mais la réciproque n'est pas vraie.

5.2. Généralisation fonctionnelle.

Il existe une approche différente qu'on aurait pu envisager pour la définition d'une relation de généralité. Celle que nous avons suivie jusqu'à présent pourrait être qualifiée de structurelle, au sens où la structure des termes intervient clairement (comme le montre la caractérisation du lemme 2).
On aurait pu utiliser la notion de calcul présente dans le λ-calcul (la β-réduction) et introduire une nouvelle définition en disant toujours que u est plus général que v si l'ensemble engendré par u contient celui engendré par v, mais en définissant l'ensemble engendré par u comme étant tous les termes $(u\ a_1\ \ldots\ a_m)$ et les termes qui leur sont β-équivalents. La nouvelle relation peut donc se caractériser en disant que u est plus général que v ssi pour tous termes $b_1, \ldots, b_n$, il existe des termes $a_1, \ldots, a_m$ tels que $(u\ a_1\ \ldots\ a_m)$ et $(v\ b_1\ \ldots\ b_n)$ soient β-équivalents.

Cette approche pose deux problèmes, en-dehors de la complexité survenant si on veut gérer la β-équivalence.

Tout d'abord si u est normalisable, u est résoluble (cf. [**2**]). Si u est en plus un terme clos, cela a pour conséquence que pour tout terme v, il existe $a_1, \ldots, a_m$ tels que $(u\ a_1\ \ldots\ a_m)$ soit β-équivalent à v. Un terme clos et normalisable quelconque serait alors forcément plus général que tous les termes.

L'autre problème c'est la propriété suivante ([**1**]).

THÉORÈME (Scott). *Soit A un ensemble non-trivial de λ-termes (i.e. A n'est pas vide et n'est pas l'ensemble de tous les λ-termes). Si A est stable par β-équivalence alors A n'est pas récursif.*

Considérons alors un terme u et tel que u ne soit pas plus général que tous les autres termes. Alors l'ensemble $D(u)$ des termes moins généraux que u n'est pas trivial (il est non-vide car il contient u). Il est stable par β-équivalence, car si v appartient à $D(u)$, alors pour tous termes $b_1, \ldots, b_n$, il existe des termes $a_1, \ldots, a_m$ tels que $(u\ a_1\ \ldots\ a_m)$ et $(v\ b_1\ \ldots\ b_n)$ soient β-équivalents, et il est clair que si w est β-équivalent à v cette propriété sera aussi vraie pour w. Donc $D(u)$ n'est pas récursif, et la fonction qui prend deux termes u et v et décide si u est plus général que v non plus.

Algorithmes

Nous ne présentons pas en détail l'algorithme permettant de décider si un terme e est plus général qu'un terme f, ou de construire les bornes inférieure et supérieure de e et f. De même nous ne décrivons pas les fonctions auxiliaires utilisées. Ces informations sont disponibles sous forme de commentaires dans le code source.

Les trois algorithmes fonctionnent selon le même principe de base: on descend en parallèle dans le corps des deux termes e et f. Ce qui change c'est le comportement adopté selon les étiquettes du noeud u de e et du noeud v de f où on se trouve. Ces différents comportements sont résumés dans les trois tableaux suivants, avec à chaque fois une explication des cas rencontrés.

6. Relation de généralité

(u, v)	Atom y	variable de tête Var y	variable liée Var y	Abs (y,v1)	App (v1,v2)
Atom x	x = y ?	$\perp$	$\perp$	$\perp$	$\perp$
variable de tête Var x	test v	test v	test v	test v	test v
variable liée Var x	$\perp$	$\perp$	alpha? (x,y)	$\perp$	$\perp$
Abs (x,u1)	$\perp$	$\perp$	$\perp$	rec (u1,v1)	$\perp$
App (u1,u2)	$\perp$	$\perp$	$\perp$	$\perp$	rec (u1,v1) rec (u2,v2) compatible?

x = y ? On a un atome de chaque côté et on teste si c'est exactement le même atome

test v Toute cette ligne correspond au cas où on a une variable de tête Var x dans le terme u. On va alors pouvoir définir une substitution associant à Var x le terme v si et seulement si les seules variables libres de v sont des variables de tête de f. Dans le cas où v est une variable on peut répondre rapidement, mais tous ces cas peuvent être traités d'un seul coup.

$\perp$ Si on se trouve dans l'un de ces cas lors du parcours de e et f, e ne peut pas être une généralisation de f.

alpha? (x, y) On a une variable liée de chaque côté, e est une généralisation de f ssi ces variables renvoient à la même abstraction rencontrée lors de la descente.

rec (u1, v1) On continue à descendre par un appel récursif sur u1 et v1.

compatible? Après l'appel récursif sur (u1, v1) et (u2, v2), il faut vérifier que les affectations établies de chaque côté sont compatibles, i.e. qu'une même variable de tête de e ne reçoit pas deux valeurs non α-équivalentes.

7. Borne supérieure

(u, v)	Atom y	variable de tête Var y	variable liée Var y	Abs (y,v1)	App (v1,v2)
Atom x	x = y ? ou try_to_link	try_to_link	try_to_link	try_to_link	try_to_link
variable de tête Var x	try_to_link	try_to_link	try_to_link	try_to_link	try_to_link
variable liée Var x	try_to_link	try_to_link	alpha? (x,y)	try_to_link	try_to_link
Abs (x,u1)	try_to_link	try_to_link	try_to_link	rec (u1,v1)	try_to_link
App (u1,u2)	try_to_link	try_to_link	try_to_link	try_to_link	sequence rec (u1,v1) rec (u2,v2)

x = y ? On a un atome de chaque côté et on teste si c'est exactement le même atome, sinon on établit un lien.

try_to_link Si on a deux termes différents u et v, on va essayer d'établir un lien entre u et v. Cela veut dire que l'on veut placer une variable à cet endroit dans la borne supérieure, qui sera substituée par u pour donner e et par v pour donner f. Cette opération n'est possible que si les variables libres de u sont toutes des variables de tête de e et les variables libres de v sont toutes des variables de tête de f. Si ce n'est pas le cas on renvoie les variables qui posent problème car il faudra remonter au moins jusqu'aux abstractions correspondantes avant de pouvoir établir un lien.

alpha? (x, y) On a une variable liée de chaque côté, e est une généralisation de f ssi ces variables renvoient à la même abstraction rencontrée lors de la descente, si ce n'est pas le cas il faut remonter au moins jusqu'aux abstractions correspondantes.

rec (u, v) On continue à descendre par un appel récursif sur u et v.

sequence On effectue l'appel récursif sur (u1, v1) puis sur (u2, v2). Ainsi si on a établi un lien (u3, v3) lors de l'appel sur (u1, v1) et qu'on doit en établir un autre lien (u4, v4) lors de l'appel sur (u2, v2), et si u3 est α-équivalent à u4 et v3 est α-équivalent à v4, on pourra donner le même nom de variable dans la borne supérieure.

8. Borne inférieure

(u, v)	Atom y	variable de tête Var y	variable liée Var y	Abs (y,v1)	App (v1,v2)
Atom x	x = y ?	link_f	$\bot$	$\bot$	$\bot$
variable de tête Var x	link_e	link_e	link_e	link_e	link_e
variable liée Var x	$\bot$	link_f	alpha? (x,y)	$\bot$	$\bot$
Abs (x,u1)	$\bot$	link_f	$\bot$	rec (u1,v1)	$\bot$
App (u1,u2)	$\bot$	link_f	$\bot$	$\bot$	rec (u1,v1) rec (u2,v2) compatible?

x = y ? On a un atome de chaque côté et on teste si c'est exactement le même atome, sinon on établit un lien.

link_e On a une variable de tête de e pour u. On va donc pouvoir affecter v à u, à condition que les variables libres de v soient des variables de tête de f, et on mettra v dans la borne inférieure.

link_f Cas symétrique pour l'autre côté.

$\bot$ Si on a deux termes différents u et v, et ni u ni v n'est une variable de tête, on ne peut pas avoir de borne inférieure.

alpha? (x, y) On a une variable liée de chaque côté, e est une généralisation de f ssi ces variables renvoient à la même abstraction rencontrée lors de la descente, si ce n'est pas le cas il faut remonter au moins jusqu'aux abstractions correspondantes.

rec (u, v) On continue à descendre par un appel récursif sur u et v.

compatible? On doit vérifier que les affectations effectuées dans u1 et dans u2 sont compatibles, i.e. qu'une mêmem variable de tête de e ne reçoit pas deux valeurs distinctes. De même pour les affectations effectuées dans v1 et v2.

Implémentation en Caml

```
(* ******************************************************* *)
(* fonctions generales sur les listes                      *)
(* ******************************************************* *)



(* # first_position;;                                      *)
(* - : 'a -> 'a list -> int = <fun>                        *)
(* first_position prend un element x et une liste l        *)
(* et renvoie la position                                  *)
(* de la premiere occurence de x dans l,                   *)
(* et l'entier 0 si x n'apparait pas.                      *)

let first_position x =
  let rec aux pos x = function
    | [] -> 0
    | hd :: _ when hd = x -> pos
    | _ :: tail -> aux (pos + 1) x tail
  in aux 1 x
;;



(* # last_position;;                                       *)
(* - : 'a -> 'a list -> int = <fun>                        *)
(* last_position prend un element x et une liste l         *)
(* et renvoie la  position                                 *)
(* de la derniere occurence de x dans la liste l,          *)
(* et l'entier 0 si x n'apparait pas.                      *)

let last_position x =
  let rec aux pos pos_x x = function
    | [] -> pos_x
    | hd :: tail when hd = x -> aux (pos + 1) pos x tail
    | _ :: tail -> aux (pos + 1) pos_x x tail
  in aux 1 0 x
;;
```

```
(* # sublist;;                                                *)
(* - : 'a list -> 'a list -> bool = <fun>                    *)
(* sublist list1 list2 determine si tous les elements de    *)
(* list1 appartiennent a list2.                              *)

let rec sublist small_list big_list =
  match small_list with
    | [] -> true
    | hd :: tail ->
      if (List.mem hd big_list)
      then sublist tail big_list
      else false
;;



(* # list_subtract;;                                         *)
(* - : 'a list -> 'a list -> 'a list = <fun>                 *)
(* list_subtract prend list1 et list2 et renvoie la liste   *)
(* des elements de list1 qui ne sont pas dans list2,        *)
(* dans l'ordre de leur apparition dans list1.               *)

let list_subtract list1 list2 =
  let not_in_list2 = function x -> not (List.mem x list2) in
  List.filter not_in_list2 list1
;;



(* # add_to_set;;                                            *)
(* - : 'a -> 'a list -> 'a list = <fun>                      *)
(* add_to_set prend un element et l'ajoute au debut         *)
(* d'une liste s'il ne figure pas dans celle-ci, et         *)
(* renvoie la liste intacte sinon.                           *)

let add_to_set x set =
  if (List.mem x set)
  then set
  else x :: set
;;



(* # list_to_set;;                                           *)
(* - : 'a list -> 'a list = <fun>                            *)
(* list_to_set prend une liste et la transforme en          *)
(* en senemble, ie en supprimant les repetitions, en        *)
(* conservant l'ordre d'apparition des elements de la       *)
```

```
(* liste de depart.                                          *)

let list_to_set list =
  let rec aux set = function
    | [] -> set
    | hd :: tail -> aux (add_to_set hd set) tail
  in List.rev (aux [] list)
;;



(* # list_intersect;;                                        *)
(* - : 'a list -> 'a list -> 'a list = <fun>                 *)
(* list_intersect renvoie une liste contenant les           *)
(* qui sont presents dans chacune des deux listes passees   *)
(* en arguments.                                             *)

let list_intersect list1 list2 =
  let rec aux acc list1 = function
    | [] -> acc
    | hd :: tail ->
      if (List.mem hd list1) && not (List.mem hd acc)
      then aux (hd :: acc) list1 tail
      else aux acc list1 tail
  in aux [] list1 list2
;;







(* ******************************************************** *)
(* definition d'un terme: Atom sert a representer un terme *)
(* dont la structure ne peut plus etre separee, App est    *)
(* l'application d'un terme a un autre, et servira aussi   *)
(* pppour tous les autres operateurs, on representera      *)
(* Op arg1 ... argn par App Op arg1 ... argn. Abs est l'   *)
(* abstraction d'une variable dans un terme. le nom d'une  *)
(* variable Var est un terme, ce qui nous permettra de     *)
(* garder l'information provenant des termes que l'on      *)
(* generalise.                                              *)
(* ******************************************************** *)



type term =
  | Atom of string
  | Var of term
```

```
  | Abs of (term * term)
  | App of (term * term)
;;
```

```
(* ****************************************************** *)
(* fonctions generales sur les termes                     *)
(* ****************************************************** *)
```

```
(* # free_vars;;                                          *)
(* - : term -> term list = <fun>                          *)
(* free_vars prend en entree un terme u et renvoie la     *)
(* liste des variables libres de u, dans l'ordre d'un     *)
(* parcours en profondeur d'abord.                        *)

let free_vars term =
  let rec aux bounded_vars = function
    | Atom _ -> []
    | Var x when (List.mem x bounded_vars) -> []
    | Var x -> [x]
    | Abs (x, u) -> aux (x :: bounded_vars) u
    | App (u, v) -> (aux bounded_vars u) @
                    (aux bounded_vars v)
  in list_to_set (aux [] term)
;;
```

```
(* # alpha_eq;;                                           *)
(* - : term * term -> bool = <fun>                        *)
(* alpha prend deux termes et determine s'ils sont alpha  *)
(* equivalents.                                           *)

let alpha_eq u v =
  let rec aux bounded_u bounded_v = function
    | Atom x, Atom y when x = y -> true
    | Var x, Var y ->
      let a = (first_position x bounded_u)
      and b = (first_position y bounded_v)
      in
      (a = b) && (a > 0 || x = y)
    | Abs (x, u), Abs (y, v) ->
      aux (x :: bounded_u) (y :: bounded_v) (u, v)
```

```
    | App (u1, u2) , App (v1, v2) ->
      (aux bounded_u bounded_v (u1, v1)) &&
      (aux bounded_u bounded_v (u2, v2))
    | _ -> false
  in aux [] [] (u, v)
;;



(* # split_term;;                                          *)
(* - : term -> term list * term = <fun>                    *)
(* split_term prend un terme et le coupe,                  *)
(* renvoyant d'une part la liste de                        *)
(* ses variables de tete dans l'ordre inverse de leur      *)
(* apparition en tete du terme , et d'autre part le        *)
(* corps prive de toutes ses abstractions de tete.         *)

let split_term term =
  let rec aux headvars = function
    | Abs (x, u) -> aux (x :: headvars) u
    | u -> (headvars, u)
  in aux [] term
;;



(* # rebuild_term;;                                        *)
(* - : term list -> term -> term = <fun>                   *)
(* rebuild_term prend un terme et une liste de             *)
(* variables et ajoute une abstraction en tete au terme    *)
(* pour chaque variable de la liste. la premiere variable  *)
(* correspond a l'abstraction de tete la plus a droite     *)
(* dans le resultat. c'est l'operation inverse de          *)
(* split_term.                                             *)

let rec rebuild_term headvars term =
  match headvars with
    | [] -> term
    | hd :: tail -> rebuild_term tail (Abs (hd, term))
;;



(* # lambda_closure;;                                      *)
(* - : term -> term = <fun>                                *)
(* lambda_closure ajoute une abstraction de tete pour      *)
(* chaque variable libre d'un terme. ces abstractions de   *)
(* sont dans l'ordre d'apparition des variables libres du  *)
(* terme lors d'un parcours en profondeur.                 *)
```

```
let lambda_closure u =
  let fv_u = free_vars u
  in rebuild_term (List.rev fv_u) u
;;



(* # clean;;                                                  *)
(* - : term -> term = <fun>                                   *)
(* clean prend un terme et renomme ses variables de tete     *)
(* Atom "h1", Atom "h2",... et ses variables liees           *)
(* internes Atom "b1", Atom "b2",... de maniere a obtenir    *)
(* un terme alpha-equivalent                                 *)

let clean term =
  let rec headbuild length term =
    if length = 0
    then term
    else headbuild (length - 1)
                   (Abs (Atom ("h" ^ (string_of_int length)), term))
  and
  aux headvars boundvars =
  let headlength = List.length headvars in
  function
    | Var x ->
      let bound_pos = first_position x boundvars in
      if bound_pos > 0
      then
        let bound_num = (List.length boundvars) - bound_pos + 1 in
        Var (Atom ("b" ^ (string_of_int bound_num)))
      else
        let head_pos = first_position x headvars in
        if head_pos > 0
        then
          let head_num = headlength - head_pos + 1 in
          Var (Atom ("h" ^ (string_of_int head_num)))
        else Var x
    | Abs (x, u) ->
      let bound_num = (List.length boundvars) + 1 in
      let bound_name = "b" ^ (string_of_int bound_num) in
      Abs (Atom bound_name, aux headvars (x :: boundvars) u)
    | App (u, v) ->
      App (aux headvars boundvars u,
           aux headvars boundvars v)
    | x -> x
  in
  let (headvars, body) = split_term term in
  let clean_body = aux headvars [] body in
```

```
  headbuild (List.length headvars) clean_body
;;



(* # canonical_form;;                                          *)
(* - : term -> term = <fun>                                    *)
(* canonical_form met un terme sous forme canonique, ie       *)
(* en supprimant les abstractions de tete inutiles et en      *)
(* reordonnant celles-ci pour qu'elles correspondent a        *)
(* l'ordre d'apparition des variables dans un parcours        *)
(* en profondeur d'abord.                                     *)

let canonical_form u =
  let freevars_u = free_vars u
  and (_, body_u) = split_term u
  in
  let headvars_u =
    list_subtract (free_vars body_u) freevars_u in
  rebuild_term (List.rev headvars_u) body_u
;;




(* ******************************************************** *)
(* fonctions auxiliaires pour is_gen                        *)
(* ******************************************************** *)



(* on definit un type affectation qui represente            *)
(* l'association d'un nom de variable (name) et d'un        *)
(* d'un terme, correspondant aux substitutions qu'on va     *)
(* effectuer dans les variables de tete du terme e si on    *)
(* veut montrer que e est plus general que f.               *)

type affectation = {name:term; value:term};;



(* # find_value_of;;                                        *)
(* - : term -> affectation list -> term = <fun>             *)
(* find_value_of cherche dans une liste d'affectations      *)
(* la valeur associee a un nom de variable et la renvoie.   *)
(* si ce nom de variable                                    *)
(* n'a pas d'affectation dans la liste, elle leve une       *)
```

```
(* exception Not_found                                          *)

let rec find_value_of name = function
  | [] -> raise Not_found
  | affect :: end_list when affect.name = name ->
    affect.value
  | _ :: end_list -> find_value_of name end_list
;;



(* # merge;;                                                    *)
(* - : affectation list -> affectation list                    *)
(*           -> affectation list = <fun>                       *)
(* merge prend deux listes d'affectations, provenant par       *)
(* exemple des branches de App(u,v), et les fusionne si        *)
(* elles sont compatibles, c'est-a-dire si une meme            *)
(* variable n'est pas associee a deux valeurs non alpha        *)
(* equivalentes. on leve l'exception Not_compatible sinon      *)
(* avec le nom de la variable qui pose probleme, et            *)
(* App(u,v), ou u et v sont les valeurs contradictoires        *)
(* affectees a la variable.                                    *)

exception Not_compatible of (term * term);;

let rec merge affect_list1 = function
  | [] -> affect_list1
  | affect :: end_list2 ->
    (
    try
      let value = (find_value_of affect.name affect_list1)
      in
      if (alpha_eq value affect.value)
      then merge affect_list1 end_list2
      else raise (Not_compatible (affect.name,
                                  App (value, affect.value)))
    with
      Not_found ->
        affect :: (merge affect_list1 end_list2)
    )
;;



(* # is_gen;;                                                   *)
(* - : term * term -> bool = <fun>                             *)
(* is_gen determine si e est une generalisation de f.          *)
(* la fonction auxiliaire aux prend en entree head_e           *)
```

```
(* (liste des variables de tete de e), head_f (liste des   *)
(* variables de tete de f), bounded_e (liste des           *)
(* variables liees rencontrees en descendant dans e),      *)
(* bounded_f (liste des variables liees rencontrees en     *)
(* descendant dans f) et fait un matching sur l'etiquette  *)
(* d'un sous-arbre de e et d'un sous-arbre de f. en cas de *)
(* succes, is_gen renvoie la liste des affectations qui    *)
(* montrent que e est une generalisation de f, avec le     *)
(* constructeur Is_gen_success, en cas                     *)
(* d'echec is_gen renvoie une des raisons de l'echec, avec *)
(* le constructeur Is_gen_failure.                         *)

type is_gen_result =
  | Is_gen_failure of (term * term)
  | Is_gen_success of affectation list
;;



let is_gen e f =

  let rec aux head_e head_f bounded_e bounded_f = function

(* si on a le meme atome a gauche et a droite, ils se      *)
(* correspondent sans besoin de substitution.              *)

    | Atom x, Atom y when x = y -> Is_gen_success []

(* si a gauche on a une variable de tete de e, et un       *)
(* terme a droite dont les seules variables libres         *)
(* eventuelles sont parmis les variables de tete de f,     *)
(* on peut associer a la variable de e le terme de         *)
(* droite. si le terme de droite a des mauvaises           *)
(* variables, on renvoie un echec avec Var x et une des    *)
(* variables qui pose probleme.                            *)

    | Var x, v  when (List.mem x head_e) ->
      let badvars = list_subtract (free_vars v) head_f in
      if badvars = []
      then Is_gen_success [{name = x; value = v}]
      else Is_gen_failure (Var x, Var (List.hd badvars))

(* si on a une variable liee de chaque cote, correspondant *)
(* aux memes abstractions rencontrees pendant la           *)
(* descente (position x bounded_e = position y bounded_f), *)
(* on a une correspondance sans substitution.              *)

    | Var x, Var y when (first_position x bounded_e) =
                        (first_position y bounded_f)
```

```
      -> Is_gen_success []

(* si on a une abstraction de chaque cote, on ajoute       *)
(* les variables liees a nos listes et on descend.         *)

    | Abs (x, u), Abs (y, v) ->
      aux (list_subtract head_e [x])
          (list_subtract head_f [y])
          (x :: bounded_e) (y :: bounded_f) (u, v)

(* si on a un constructeur App de chaque cote, on          *)
(* descend dans les deux branches, si ca marche de         *)
(* chaque cote on verifie                                  *)
(* avec merge que les substitutions pour u1,v1             *)
(* sont compatibles avec celles pour u2,v2, ie qu'on       *)
(* n'a pas decide de donner deux valeurs differentes       *)
(* a une meme variable.                                    *)
(* s'il y a echec d'un cote on renvoie une des raisons     *)
(* de l'echec                                              *)

    | App (u1, u2), App (v1, v2) ->
      (
      match
        aux head_e head_f
            bounded_e bounded_f (u1, v1),
        aux head_e head_f
            bounded_e bounded_f (u2, v2)
      with
        | Is_gen_success affect_u1v1, Is_gen_success affect_u2v2 ->
          (
          try
            Is_gen_success (merge affect_u1v1 affect_u2v2)
          with
            Not_compatible u -> Is_gen_failure u
          )
        | Is_gen_failure fail, _ -> Is_gen_failure fail
        | _, v -> v
      )

(* dans tous les autres cas u ne peut pas etre             *)
(* une generalisation de v, on marque l'echec.             *)

    | u, v -> Is_gen_failure (u, v)

  in
  let (head_e, body_e) = split_term e
  and (head_f, body_f) = split_term f
  in
  aux head_e head_f [] [] (body_e, body_f)
```

```
;;



(* # is_eq;;                                                  *)
(* - : term -> term -> bool = <fun>                           *)
(* is_eq teste si deux termes sont equivalents, par la       *)
(* definition, ie en testant s'ils sont une                  *)
(* generalisation l'un de l'autre.                           *)

let is_eq u v =
  match is_gen u v, is_gen u v
  with
    | Is_gen_success _, Is_gen_success _ -> true
    | _ -> false
;;




(* ******************************************************** *)
(* fonctions auxiliaires pour upper_bound                   *)
(* ******************************************************** *)



(* le type ub_link_result sert en interne a la fonction     *)
(* upper_bound. a un noeud donne du parcours du terme de    *)
(* gauche et du terme de droite, on renvoie en cas de       *)
(* succes la borne superieure correspondante et             *)
(* les liens deja etablis entre le terme de gauche et le    *)
(* terme de droite. en cas d'echec on renvoie la liste      *)
(* des variables qui posent probleme a gauche et a droite   *)
(* afin de pouvoir eventuellement parvenir a faire un lien  *)
(* plus haut.                                               *)

type ub_link_result =
  | Ub_link_success of (term * ((term * term) list))
  | Ub_link_failure of (term list * term list)
;;



(* le type ub_result correspond a la valeur renvoyee a la   *)
(* fin par la fonction upper_bound. en cas de succes on     *)
(* donne la borne superieure obtenue, en cas d'echec on     *)
```

```
(* renvoie les variables qui ont pose probleme.    *)

type ub_result =
  | Upper_bound_success of term
  | Upper_bound_failure of (term list * term list)
;;



(* # find_eq;;                                              *)
(* - : term * term -> (term * term) list                    *)
(*                 -> term * term = <fun>                   *)
(* find_eq cherche dans une liste de liens si               *)
(* il en existe un equivalent a (u1, u2), et renvoie ce     *)
(* lien dans ce cas, et transmet l'exception Not_found      *)
(* sinon.                                                   *)

let find_eq (u1, u2) list =
  let test_eq =
    function (v1, v2) -> (alpha_eq u1 v1) &&
                         (alpha_eq u2 v2)
  in
  List.find test_eq list
;;



(* # make_link;;                                            *)
(* - : term -> term -> (term * term) list                   *)
(*          -> term * (term * term) list = <fun>            *)
(* make_link prend deux termes u et v, et une liste de      *)
(* liens linklist, et renvoie un lien equivalent et la      *)
(* meme liste si un lien equivalent existait deja,          *)
(* et le lien (u, v)                                        *)
(* et la liste apres ajout de (u, v) si ce lien est nouveau*)

let make_link u v linklist =
  try
    let  (eq_u, eq_v) = find_eq (u, v) linklist in
    (Var (App (eq_u, eq_v)), linklist)
  with
    | Not_found -> (Var (App (u, v)), (u, v) :: linklist)
;;



(* # try_to_link;;                                          *)
(* - : term list ->                                         *)
(*    term list -> (term * term) list -> term               *)
```

```
(*              -> term -> ub_link_result = <fun>          *)
(* try_to_link prend deux termes u et v, la liste des      *)
(* variables de tete du terme de gauche, et celle du terme *)
(* de droite, et la liste des liens deja etablis. cette    *)
(* fonction regarde si on peut faire un lien entre u et v, *)
(* ie si u et v ne contiennent pas d'autres variables      *)
(* libres que que les variables de tete du terme           *)
(* correspondant, et dans ce cas elle lie u et v. sinon    *)
(* elle renvoie pour u et v la liste des variables qui     *)
(* posent probleme.                                        *)

let try_to_link head_e head_f linklist u v =
  let badvars_u = list_subtract (free_vars u) head_e
  and badvars_v = list_subtract (free_vars v) head_f
  in
  if badvars_u = [] && badvars_v = []
  then Ub_link_success (make_link u v linklist)
  else Ub_link_failure (badvars_u, badvars_v)
;;



(* # linklist_to_varlist;;                                 *)
(* - : (term * term) list -> term list = <fun>             *)
(* chaque lien entre un sous-terme u de e et un sous-terme *)
(* v de f deviendra une variable dans la borne superieure  *)
(* de e et f. le nom de cette variable sera App (u, v),    *)
(* afin de conserver les termes qui par substitution nous  *)
(* permetront de retrouver e et f. linklist_to_varlist     *)
(* transforme une liste de liens en la liste des noms de   *)
(* variable correspondant.                                 *)

let linklist_to_varlist list =
  List.map (function u -> (App u)) list
;;



(*# upper_bound;;                                         *)
(* - : term -> term -> ub_result = <fun>                  *)
(* upper_bound prend deux termes e et f et renvoie        *)
(* Upper_bound_success avec la borne suprieure obtenue   *)
(* ou Upper_bound_failure avec deux listes contenant des  *)
(* variables libres de e et de f respetivement.           *)

let upper_bound e f =

(* la fonction aux prend la liste des variables de tete de *)
(* e (head_e), de f(head_f), deux listes qui servent a     *)
```

```
(* stocker les variables liees quand on rencontre une      *)
(* abstraction (bounded_e et bounded_f), une liste de      *)
(* liens deja tablis (linklist) et fait une analyse de    *)
(* cas selon les noeuds de e et f ou l'on se trouve.       *)

  let rec aux head_e head_f bounded_e bounded_f linklist =
  function

(* si on a le meme atome de chaque cote, on le met dans la *)
(* borne superieure.                                       *)

    | Atom x, Atom y when x = y ->
      Ub_link_success (Atom x, linklist)

(* si on a une variable liee de chaque cote (x et y),      *)
(* on teste si                                             *)
(* elles se referent a la meme abstraction, auquel cas on  *)
(* met une variable Var (App (x, y)) dans la borne         *)
(* superieure en construction, sinon on renvoie [x] et [y] *)
(* comme listes de mauvaises variables qui nous forcent a  *)
(* remonter.                                               *)

    | Var x, Var y when (List.mem x bounded_e) &&
                        (List.mem y bounded_f) ->
      let pos_x = first_position x bounded_e
      and pos_y = first_position y bounded_f
      in
      if pos_x = pos_y
      then Ub_link_success (Var (App (x, y)), linklist)
      else  Ub_link_failure ([x], [y])

(* si on a une abstraction de chaque cote, on appelle      *)
(* aux sur u et v, en ajoutant x et y a nos listes de      *)
(* variables libres et en les enlevant de nos listes de    *)
(* variables de tete. si on parvient a construire un       *)
(* resultat pour aux de u et v, on reconstruit             *)
(* l'abstraction correspondante avec comme nom de          *)
(* variable App (x, y). en cas d'echec on enleve x et y    *)
(* aux listes de mauvaises variables renvoyees par         *)
(* l'appel sur u et v. si ces listes sont alors vides      *)
(* on peut etablir un lien a ce niveau, sinon on           *)
(* transmet les listes de mauvaises variables.             *)

    | Abs (x, u), Abs (y, v) ->
      (
      match aux (list_subtract head_e [x])
        (list_subtract head_f [y])
        (x :: bounded_e) (y :: bounded_f)
        linklist (u, v)
```

```
      with
        | Ub_link_success (gen_uv, linklist_uv) ->
          Ub_link_success (Abs (App (x, y), gen_uv),
                           linklist_uv)
        | Ub_link_failure (badvars_u, badvars_v) ->
          let fvleft = list_subtract badvars_u [x]
          and fvright = list_subtract badvars_v [y]
          in
          if fvleft = [] && fvright = []
          then Ub_link_success
            (make_link (Abs (x, u)) (Abs (y, v)) linklist)
          else Ub_link_failure (fvleft, fvright)
      )

(* si on a App de chaque cote, on appelle d'abord aux avec *)
(* u1 et v1. en cas d'echec on calcule les mauvaises       *)
(* variables de u2 et v2 et on renvoie tout ca. en cas de  *)
(* succes on appelle aux sur u2 et v2, en transmettant     *)
(* la liste des liens etablis apres l'appel sur u1 et v1.  *)
(* en cas de succes de ce deuxieme appel on renvoie le     *)
(* terme construit a partir du resultat de nos deux appels *)
(* et la nouvelle liste de liens. en cas d'echec de ce 2eme*)
(* appel on renvoie les listes de variables temoignant de  *)
(* cet echec                                               *)

    | App (u1, u2), App (v1, v2) ->
      (
      match aux head_e head_f bounded_e
                bounded_f linklist (u1, v1)
      with
        | Ub_link_failure (badvars_u1, badvars_v1) ->
          let badvars_u2 =
            list_subtract (free_vars u2) head_e
          and badvars_v2 =
            list_subtract (free_vars v2) head_f
          in
          Ub_link_failure (badvars_u1 @ badvars_v1,
                           badvars_u2 @ badvars_v2)
        | Ub_link_success (gen_u1v1, linklist_u1v1) ->
          (
          match aux head_e head_f bounded_e
                    bounded_f linklist_u1v1 (u2, v2)
          with
            | Ub_link_success (gen_u2v2, linklist_u1v1u2v2) ->
              Ub_link_success (App (gen_u1v1, gen_u2v2),
                               linklist_u1v1u2v2)
    | fail -> fail
  )
      )
```

```
(* dans tous les autres cas on se retrouve avec des        *)
(* sous-termes u et v differents, donc on est oblige       *)
(* d'essayer de creer un lien a ce niveau.                 *)

    | u, v -> try_to_link head_e head_f linklist u v

(* on commence par separer les variables de tete dans e et *)
(* f. ensuite on appelle aux avec des listes vides pour    *)
(* les variables liees de e et f et les liens.             *)
(* on convertit le resultat pour renvoyer                  *)
(* Upper_bound_success ou Upper_bound_failure.             *)

  in
  let (head_e, body_e) = split_term e
  and (head_f, body_f) = split_term f
  in
  match aux head_e head_f [] [] [] (body_e, body_f)
  with
    | Ub_link_success (result, linklist) ->
      let varlist = linklist_to_varlist linklist in
      Upper_bound_success (rebuild_term varlist result)
    | Ub_link_failure u->
      Upper_bound_failure u
;;



(* # clean_ub;;                                            *)
(* - : term -> term -> term = <fun>                        *)
(* applique upper_bound et nettoie le resultat.            *)

let clean_ub x y = match (upper_bound x y) with
    | Upper_bound_success result ->
      Upper_bound_success (clean result)
    | u -> u
;;







(* ******************************************************* *)
(* fonctions auxiliaires pour lower_bound                  *)
(* ******************************************************* *)
```

```
(* le type lb_link_result sert en interne a la fonction    *)
(* lower_bound. a un noeud donne du parcours du terme de   *)
(* gauche et du terme de droite, on renvoie en cas de      *)
(* succes le terme de la borne inferieure correspondant et *)
(* deux listes: la premiere pour les affectations entre    *)
(* une variable du terme de gauche et un sous-terme du     *)
(* terme de droite, la deuxieme pour les affectations      *)
(* entre une variable du terme de droite et un sous-terme  *)
(* du terme de gauche. en cas d'echec on renvoie un couple *)
(* compose d'un sous-terme de gauche et d'un sous-terme    *)
(* de droite qui ne se correspondent pas, ou le nom d'une  *)
(* variable a laquelle on donne deux affectations          *)
(* differentes.                                            *)

type lb_link_result =
  | Lb_link_failure of (term* term)
  | Lb_link_success of (term * affectation list
                * affectation list)
;;



(* le type lb_result est renvoye a la fin par la fonction  *)
(* lower_bound. on renvoie la borne inferieure obtenue en  *)
(* cas de succes, et un couple temoignant de l'echec sinon.*)

type lb_result =
  | Lower_bound_failure of (term * term)
  | Lower_bound_success of term
;;



(* # rename_freevars;;                                     *)
(* - : term -> term -> term = <fun>                        *)
(* rename_freevars prend deux termes, name et term, et     *)
(* remplace chaque occurence d'une variable Var x libre    *)
(* dans term par la variable Var App (x, name).            *)

let rename_freevars name term =
  let rec aux bounded name = function
    | Var x when List.mem x bounded -> Var x
    | Var x -> Var (App (x, name))
    | Abs (x, u) -> Abs (x, aux (x :: bounded) name u)
    | App (u, v) -> App (aux bounded name u, aux bounded name v)
    | u -> u
  in aux [] name term
;;
```

```
(* # rename_varlist;;                                          *)
(* - : term -> term list -> term list = <fun>                  *)
(* rename_varlist renomme de maniere similaire une liste      *)
(* de noms de variables.                                      *)

let rename_varlist name list =
  List.map (function x -> App (x, name)) list
;;



(* # lower_bound;;                                             *)
(* - : term -> term -> term = <fun>                            *)
(* lower_boud calcule la borne inferieure de deux termes      *)
(* elle renvoie Lower_bound_success et la borne inferieure    *)
(* en cas de succes, et Lower_bound_failure et u couple       *)
(* temoignant de cet echec.                                   *)

let lower_bound e f =

(* la fonction aux prend la liste des variables de tete de    *)
(* e (head_e), de f(head_f), deux listes qui servent a        *)
(* stocker les variables liees quand on rencontre une         *)
(* abstraction (bounded_e et bounded_f),                      *)
(* et fait une analyse de                                     *)
(* cas selon les noeuds de e et f ou l'on se trouve.          *)

  let rec aux head_e head_f bounded_e bounded_f = function

(* si on a deux atomes identiques, on met cet atome dans      *)
(* borne inferieure, sans avoir a faire d'affectations        *)

    | Atom x, Atom y when x = y -> Lb_link_success (Atom x, [], [])

(* si on a deux variables x,y liees se referant a la meme     *)
(* abstraction, on renvoie la variable de nom App (x, y),     *)
(* sans faire d'affectations. sinon c'est un echec pour       *)
(* la borne inferieure et on renvoie Var x, Var y comme       *)
(* temoins.                                                   *)

    | Var x, Var y when (List.mem x bounded_e) &&
                        (List.mem y bounded_f) ->
      let a = first_position x bounded_e
      and b = first_position y bounded_f
      in
      if a = b
```

```
      then Lb_link_success (Var (App (x, y)), [], [])
      else Lb_link_failure (Var x, Var y)

(* Si on a une variable de tete x a gauche, un sous-terme  *)
(* t de f a droite, on va pouvoir mettre t dans la borne   *)
(* inferieure et affecter la valeur t a la variable de nom *)
(* x, a condition que les variables libres de t soient des *)
(* variables libres de f. si ce n'est pas le cas on ne     *)
(* peut pas avoir de borne inferieure, on renvoie Var x et *)
(* une des variables posant probleme.                      *)

    | Var x, v when (List.mem x head_e) ->
      let fv_v = (free_vars v) in
      let badvars_v = list_subtract fv_v head_f in
      if badvars_v = []
      then Lb_link_success (v, [{name = x; value = v}], [])
      else Lb_link_failure (Var x, List.hd badvars_v)

(* cas symetrique du precedent.                            *)

    | u, Var y when (List.mem y head_f) ->
      let fv_u = (free_vars u) in
      let badvars_u = list_subtract fv_u head_e in
      if badvars_u = []
      then Lb_link_success (u, [], [{name = y; value = u}])
      else Lb_link_failure (List.hd badvars_u, Var y)

(* si on a Abs de chaque cote, on appelle aux sur u et v.  *)
(* en cas de succes on reconstruit l'abstraction avec le   *)
(* terme obtenu et comme nom de variable App (x, y). en    *)
(* cas d'echec on transmet celui-ci.                       *)

    | Abs(x,u), Abs(y,v) ->
      (
      match aux (list_subtract head_e [x])
                (list_subtract head_f [y])
                (x::bounded_e) (y::bounded_f) (u,v)
      with
        | Lb_link_success (lbound_uv, affect_u, affect_v) ->
          Lb_link_success (Abs (App (x, y), lbound_uv),
                           affect_u, affect_v)
        | x -> x
      )


(* si on a App de chaque cote, on s'appelle sur u1, v1 et  *)
(* u2, v2. en cas de succes des deux appels, on verifie    *)
(* que les affectations sur u1 et sur u2 sont compatibles  *)
(* et de meme pour celles sur v1 et v2. on renvoie alors   *)
```

```
(* le terme et les listes d'affectations fusionnees. si    *)
(* elles ne sont pas compatibles on renvoie un echec pour  *)
(* le borne inferieure avec le couple (var_name, var_name) *)
(* ou var_name est le nom d'une variable recevant des      *)
(* affectations non-equivalentes.                          *)
(* en cas d'echec d'un des deux appels a aux on transmet   *)
(* cet echec.                                              *)

    | App(u1,u2), App(v1,v2) ->
      (
      match (aux head_e head_f bounded_e bounded_f (u1,v1),
             aux head_e head_f bounded_e bounded_f (u2,v2))
      with
        | Lb_link_success (lbound_u1v1, affect_u1, affect_v1),
          Lb_link_success (lbound_u2v2, affect_u2, affect_v2) ->
          (
          try
            Lb_link_success (App (lbound_u1v1, lbound_u2v2),
                             merge affect_u1 affect_u2,
                             merge affect_v1 affect_v2)
          with
            Not_compatible u -> Lb_link_failure u
          )
        |  Lb_link_failure u, _ -> Lb_link_failure u
        | _, fail -> fail
      )

    | u,v -> Lb_link_failure (u, v)

(* on commence par separer les variables de tete de chaque *)
(* terme. ensuite on regarde si il y a des variables de    *)
(* tete qui ont le meme nom. si c'est le cas on renomme    *)
(* toutes les variables de tete de e en changeant x en     *)
(* App(x, Atom "left") et toutes celles de f avec Atom     *)
(* "right". on appelle aux et on reconstruit les variables *)
(* de tete de la borne inferieure avec lambda_closure en   *)
(* en cas de succes.                                       *)

  in let (head_e, body_e) = split_term e
     and (head_f, body_f) = split_term f
  in
  match
    if (list_intersect head_e head_f) = []
    then aux head_e head_f [] [] (body_e, body_f)
    else aux (rename_varlist (Atom "left") head_e)
             (rename_varlist (Atom "right") head_f)
             [] []
             (rename_freevars (Atom "left") body_e,
              rename_freevars (Atom "right") body_f)
```

```
  with
    | Lb_link_success (result, affect_e, affect_f) ->
      Lower_bound_success (lambda_closure result)
    | Lb_link_failure u -> Lower_bound_failure u
;;



(* # clean_lb;;                                              *)
(* - : term -> term -> term = <fun>                          *)
(* clean_lb calcule la borne inferieure et nettoie le       *)
(* resultat.                                                 *)

let clean_lb u v =
  match (lower_bound u v) with
    | Lower_bound_success lb -> Lower_bound_success (clean lb)
    | x -> x
;;
```

Bibliography

[1] H.P. Barendregt, *The lambda calculus* (North-Holland, Amsterdam, 1985).
[2] J.L. Krivine, *Lambda-calcul, types et modèles* (Masson, Paris, 1990).
[3] G. Leplatre, *Développement d'un opérateur d'abstraction généralisée pour le langage de programmation musicale Elody* (Mémoire de stage au GRAME, Lyon, 1997).