Substitutions explicites linéaires et préservation de la normalisation forte: une formalisation en Coq

Daniel El Ouraoui & Micaela Mayero & Damiano Mazza

Laboratoire d'Informatique de Paris Nord, UMR 7030 CNRS - Université Paris 13

Résumé

Le calcul de substitutions explicites linéaires (LSC) introduit par Beniamino Accattoli et Delia Kesner s'appuie sur la logique linéaire pour décrire finement l'opération de substitution en lambda-calcul. Ses bonnes propriétés concernant la réécriture en font un des calculs les mieux adapté à l'étude de la dynamique de la réduction du lambda-calcul et des langages de programmations fonctionnels en général. La préservation de la normalisation forte (PSN) est une de ses propriétés les plus importantes. Accattoli a remarqué que PSN admet une preuve particulièrement simple en LSC utilisant un résultat général de Kesner. Ce résultat affirme, sous certaines hypothèses vérifiées par LSC, que pour démontrer PSN il suffit de montrer que la normalisation forte d'une substitution implicite implique la normalisation forte de la substitution explicite correspondante (propriété IE). Dans cet article nous présentons une formalisation et une preuve en Coq de la propriété IE pour LSC.

1. Introduction

Motivés par la volonté de rapprocher le λ-calcul des implantations des langages de programmation, Abadi, Cardelli, Curien et Lévy ont introduit le $\lambda\sigma$-calcul, basé sur la notion de *substitution explicite* [1]. L'idée fondamentale est de donner un statut syntaxique à la notion de substitution, en la faisant apparaître explicitement dans le processus de calcul, au lieu de la traiter comme une notion externe, définie au niveau méta-théorique.

Quelques temps après l'introduction des substitution explicites, Melliès [13] trouva que la définition naïve proposée par Abadi et al. présentait des problèmes : il existe des λ-termes fortement normalisant (c'est-à-dire, n'ayant aucun chemin de réduction infini) qui ne sont plus fortement normalisant quand on les considère comme des $\lambda\sigma$-termes, c'est-à-dire, quand on les évalue en suivant les règles de calcul associées aux substitutions explicites. Il se posait ainsi la question de formuler un calcul de substitutions explicites jouissant de la propriété de *préservation de la normalisation forte* (ou PSN, de l'anglais *preservation of strong normalization*) : un calcul Λ_{exp} muni d'un plongement $(\cdot)^\circ : \Lambda \longrightarrow \Lambda_{\mathrm{exp}}$ du λ-calcul pur habituel tel que, pour tout $t \in \Lambda$,

simulation : $t \to_\beta t'$ implique $t^\circ \to^* t'^\circ$;

PSN : t fortement normalisant implique t° fortement normalisant ;

où $\to_\beta$ est la notion habituelle de β-réduction et $\to^*$ est (la clôture réflexive-transitive de) la notion de réduction du calcul avec substitution explicites Λ_{exp}. [1]

Plusieurs calculs de substitutions explicites on été introduits pour répondre à la question ci-dessus, mentionnons par exemple [12, 9, 7, 11]. Une approche abstraite de la question, qui est celle plus ou moins explicitement adoptée par les deux derniers travaux, consiste à s'inspirer des

1. Cette présentation est un peu simpliste, normalement on a envie que le plongement satisfasse aussi d'autres propriétés, mais la propriété de simulation ci-dessus suffit à rendre la question de la PSN non-banale.

réseaux de preuves de la logique linéaire. Ceux-ci contiennent, en quelque sorte, une notion de substitution explicite primitive ayant un bon comportement logique et, donc, calculatoire (ceci est lié à la décomposition de l'implication intuitionniste en une implication purement linéaire, couplée à un opérateur de « ressource » ; les constructions syntaxiques correspondant à cet opérateur s'avèrent être précisément des substitutions explicites). C'est justement en suivant cette approche jusqu'à son essence que Accattoli et Kesner ont formulé le *λ-calcul structurel* [5] et, ensuite, le *λ-calcul avec substitutions explicites linéaires* (LSC, [3]). La preuve de la propriété PSN est un résultat essentiel dans la théorie des substitutions explicites. Cette preuve, qui pour ce dernier calcul, présente des similarités avec un calcul précédemment considéré par Milner [14], est l'objet de notre étude.

La preuve de la propriété PSN pour le LSC est particulièrement élégante, car elle se sert d'une approche abstraite due à Kesner [10]. Dans son travail, Kesner introduit une théorie générale pour étudier, entre autre, la préservation de la normalisation forte dans les calculs avec substitutions explicites. L'un des résultats fondamentaux de cette théorie est d'avoir isolé une propriété garantissant PSN de manière tout à fait générale, pourvu que l'on adhère au cadre axiomatique de la théorie. C'est ce que Kesner appelle *propriété* **IE** (« de l'**I**mplicite à l'**E**xplicite »). Elle se formule de la façon suivante :

$$(\mathbf{IE}) : \qquad \forall\ u \in \mathrm{SN},\ t\{u/x\} \in \mathrm{SN} \quad \text{implique} \quad t[u/x] \in \mathrm{SN},$$

où SN dénote l'ensemble des termes fortement normalisables du calcul avec substitutions explicites en question, $t\{u/x\}$ dénote la substitution *implicite* (la notion définie au niveau « méta ») et $t[u/x]$ est le terme dénotant la substitution *explicite*. Le LSC étant inclus dans le cadre de la théorie de Kesner, la preuve de la propriété de PSN pour ce calcul se réduit donc à montrer que le LSC jouit de la propriété **IE**. C'est cette dernière preuve, due originalement à Accattoli [4], que nous avons formalisé en Coq [6]. Il faut remarquer que la preuve que la propriété **IE** implique PSN a été formalisée en Coq dans [8], ce qui permet de dire (à quelques détails près, voir la conclusion) que la preuve de la propriété PSN pour le LSC est désormais entièrement formalisée.

Nous commencerons par présenter la cadre théorique de notre formalisation, en particulier les substitutions explicites linéaires, les propriétés PSN et **IE**. Puis nous détaillerons la formalisation Coq. Dans la conclusion nous ferons état des points saillants et difficultés de cette formalisation avant de présenter les travaux futurs.

2. Présentation du cadre théorique

Dans cette section nous détaillerons les définitions et les propriétés nécessaires à la formalisation et à la preuve Coq. Le lecteur intéressé pourra également se référer, en annexe, à une transcription plus détaillée, effectuée dans le cadre de notre travail, de la preuve originale [4].

2.1. Le calcul avec substitutions explicites linéaires

Le langage de termes du LSC est le suivant :

$$t, u ::= x \mid \lambda x.t \mid tu \mid t[u/x].$$

où les occurrences de x dans t sont liées dans le terme $t[u/x]$. Dans la suite, l'ensemble des variables libres d'un terme t sera noté $\mathrm{fv}(t)$. La réduction du LSC utilise les notions de contextes suivantes :

$$C ::= \langle\cdot\rangle \mid \lambda x.C \mid Ct \mid tC \mid C[t/x] \mid t[C/x] \qquad \text{(contextes généraux)}$$
$$L ::= \langle\cdot\rangle \mid L[t/x] \qquad \text{(contextes de substitution)}$$

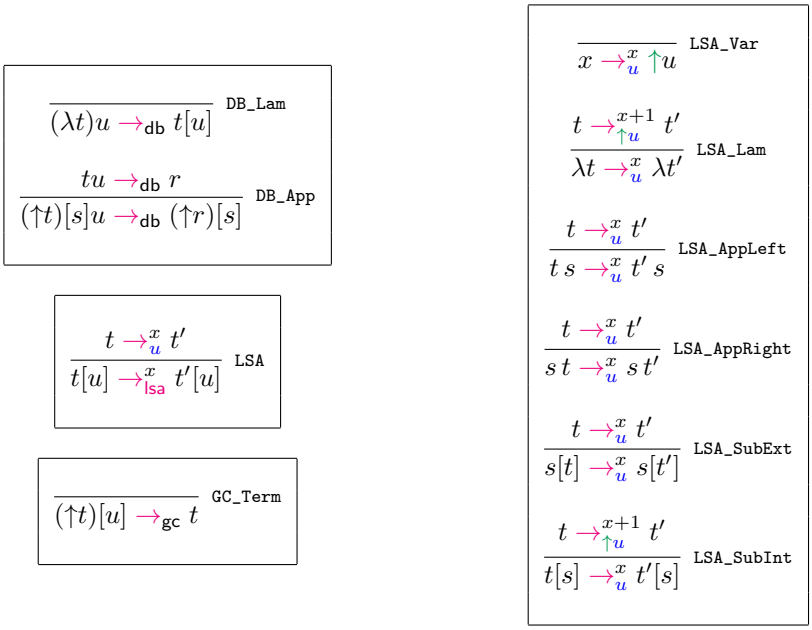

FIGURE 1 – Décomposition "inductive" des règles de réduction

On peut maintenant introduire les règles de réduction :

$$\begin{aligned} L\langle\lambda x.t\rangle u &\to_{\mathsf{db}} L\langle t[u/x]\rangle \\ C\langle x\rangle[u/x] &\to_{\mathsf{ls}} C\langle u\rangle[u/x] \\ t[u/x] &\to_{\mathsf{gc}} t \qquad\qquad \text{si } x \notin \mathrm{fv}(t) \end{aligned}$$

où, dans la règle ls, on suppose que le contexte C ne lie pas l'occurrence de x substituée. Les règles $\to_{\mathsf{db}}$, $\to_{\mathsf{ls}}$ et $\to_{\mathsf{gc}}$ sont appelées respectivement *distant beta-réduction*, *linear substitution* et *garbage collection*. On notera, dans toute la suite, $\to$ la clôture par contextes (généraux) de l'union de $\to_{\mathsf{db}}$, $\to_{\mathsf{ls}}$ et $\to_{\mathsf{gc}}$.

Cette formulation intuitive et concise ne permet cependant pas simplement de définir ces réductions en Coq. Nous avons donc effectué un travail préliminaire, consistant à décomposer ces règles. Cette décomposition consiste à incorporer la définition inductive des contextes dans celle des règles.

Les règles décomposées sont définies dans la figure 1. Notons deux choses :

— la règle $\to_{\mathsf{ls}}$ fait l'objet d'une double décomposition : l'utilisation des indices de de Bruijn, que nous détaillerons en section 3 nous oblige à considérer une version paramétrée de $\to_{\mathsf{ls}}$ notée $\to_{\mathsf{lsa}}^x$ (où x est un indice de de Bruijn). Cette réduction est à son tour paramétrée par un terme ($\to_u^x$), ce qui facilite l'utilisation de l'induction de Coq et est motivée par le fait qu'une application de la règle $\to_{\mathsf{ls}}$ est toujours de la forme $t[u/x] \to_{\mathsf{ls}} t'[u/x]$, correspondant à $t \to_u^x t'$.

— $\to_u^x$ se décompose bien en six cas correspondants aux six cas de la définition des contextes généraux ci-dessus. De même pour $\to_{\mathsf{db}}$ qui se décompose en deux cas correspondant aux deux cas de la définition des contextes de substitution.

2.2. De l'Implicite à l'Explicite (IE)

Rappelons la propriété **IE** :

$$(\mathbf{IE}): \qquad \forall\ u \in \mathrm{SN},\ t\{u/x\} \in \mathrm{SN} \quad \text{implique} \quad t[u/x] \in \mathrm{SN},$$

Notons w le terme $t[u/x]$. Il est bien connu que pour montrer que $w \in \mathrm{SN}$, il suffit de démontrer que $w' \in \mathrm{SN}$ pour tout $w \to w'$. La preuve se fait par induction forte sur la paire

$$(\eta(t\{u/x\}) + \eta(u), |t|_x)$$

où $\eta(w)$ dénote la longueur de la plus longue réduction du terme w (supposé SN) et $|t|_x$ dénote le nombre d'occurrences libres de x dans t.

Cette preuve est principalement basée sur l'utilisation des deux lemmes intermédiaires suivant :

Lemme 1. *Soient $t, u \in \mathrm{LSC}$, si $u \to u'$ alors $t\{u/x\} \to^* t\{u'/x\}$.*

Lemme 2. *Soient $t, u \in \mathrm{LSC}$, si $t \to t'$ alors $t\{u/x\} \to t'\{u/x\}$.*

La preuve du lemme 1 se fait par induction sur t, celle du lemme 2 par induction sur la définition de la réduction $t \to t'$. Se référer à l'annexe pour plus de détails.

3. Formalisation

Dans cette section nous présenterons les grandes lignes de la formalisation Coq [2] correspondant aux notions décrites à la section 2. Nous commencerons par énoncer les lemmes préliminaires nécessaires à la définitions des termes, des substitutions, puis les propriétés de réduction. Enfin nous énoncerons le lemme **IE**.

3.1. Définitions préliminaires

3.1.1. Le prédicat SN

La preuve de la propriété **IE** s'appuie sur la définition de SN. Cette dernière n'étant pas le but principal de ce travail, est pour le moment axiomatisée.

L'axiomatisation requiert l'introduction des paramètres fonctionnels `sn`: `term` → `Prop` et η: `term` → `nat`, tels que `sn t` signifie que `t` est fortement normalisant et η(`t`) représente la longueur de la plus longue réduction de `t`. Notons qu'une définition non axiomatique de η nécessiterait au préalable que `t` soit SN et utiliserait le lemme de Koenig. L'utilisation restreinte que nous faisons de cette fonction η nous permet de ne pas en donner une définition explicite.

L'axiomatisation de SN comprends 7 axiomes dont les deux principaux sont :

```
Axiom Ax1 : forall t : term,
(forall t': term,  t −→ t' → sn t' ) → sn t.

Axiom Ax2 : forall t t': term,
sn t → t −→ t' → (sn t' ∧ η(t') < η(t)).
```

où $-\!\!\to$ représente la réduction $\to$ définie à la section 2.1. Dans la suite nous utiliserons également la notation $-\!\!\to$ `x` pour représenter $\to_{\mathsf{x}}$ avec $\mathsf{x} \in \{\mathsf{db}, \mathsf{ls}, \mathsf{gc}\}$.

2. Le code source Coq est disponible à l'adresse `http://lipn.univ-paris13.fr/~mayero/IE/`

3.1.2. Termes et substitution implicite

Nous utilisons les indices de de Bruijn pour formaliser les variables des termes. La formalisation est tout à fait standard. Remarquons toutefois que l'utilisation des formalisations déjà existantes ne nous est pas possible à cause de la présence des substitutions explicites qui introduisent un lieur supplémentaire. Par exemple, le terme $\lambda x.y(\lambda z.yz)[w/y]$ s'écrit $\lambda.0(\lambda.10)[j+1]$, où j est l'indice associé à la variable w.

La définition inductive Coq des termes s'écrit ainsi :

```
Inductive term : Set :=
 | TVar : nat → term
 | TAbs : term → term
 | TApp : term → term → term
 | TSub : term → term → term.
```

Il est bien connu que l'opération de substitution implicite utilisant des indices de de Bruijn impose une réindexation des variables libres du terme substitué. Par exemple, $(\lambda.0(\lambda.10))\{(\lambda.\underline{1}0)/1\}$ sans réindexation donnerait $\lambda.0(\underline{\lambda}.(\lambda.\underline{1}0)0)$ où la variable correspondant à $\underline{1}$ dans le premier terme, qui est libre, est capturée par le $\underline{\lambda}$ dans le second terme, ce qui est faux. Le résultat correct de la substitution est $\lambda.0(\lambda.(\lambda.\underline{3}0)0)$.

Cette opération de réindexation, que l'on appelle *lift* (notée $\uparrow_k^n$), est définie de la manière suivante :

```
Fixpoint lift_rec n t {struct t} : nat → term :=
  fun k ⇒
    match t with
      | TVar i  ⇒
          (* si k<=i then ... else ... *)
        match le_gt_dec k i with
          | left _ ⇒ TVar (n + i)
          | right _ ⇒ TVar i
        end
      | \u ⇒ \(lift_rec n u (S k))
      | u · v ⇒ (lift_rec n u k) · (lift_rec n v k)
      | u [v]  ⇒ (lift_rec n u (S k))[(lift_rec n v k)]
    end.
```

où les notations `\u`, `u · v` et `u [v]` représentent respectivement les constructeurs `TAbs`, `TApp` et `TSub`.

On utilisera également dans la suite les notations $\uparrow^n$ pour $\uparrow_0^n$ et $\uparrow$ pour $\uparrow^1$ qui correspondent aux définitions suivantes :

```
Definition lift n t := lift_rec n t 0.
Definition shift t := lift 1 t.
```

A l'aide de cette définition, nous pouvons formaliser l'opération de substitution implicite ainsi :

```
Fixpoint subst_rec u t {struct t} : nat → term :=
  fun k ⇒
    match t with
      | TVar i ⇒
        (* si i=k then ... else ... *)
        match eq_nat_dec k i with
          | left _ ⇒ u
          | right _ ⇒ TVar i
        end
      | \v ⇒ \(subst_rec (shift u) v (S k))
```

```
    | w·v ⇒ (subst_rec u w k)·(subst_rec u v k)
    | w[v] ⇒ (subst_rec (shift u) w (S k))[(subst_rec u v k)]
  end.
```

Les deux lemmes suivants sont souvent utilisés dans nos preuves. Le premier énonce que lorsque l'on compose deux lifts il est toujours possible de permuter l'ordre de réindexation des variables :

```
Lemma lift_commut: forall a k i j p, k <= p →
  lift_rec i (lift_rec j a p) k = lift_rec j (lift_rec i a k) (i + p).
```

Le deuxième lemme propage la réindexation au sein d'une substitution :

```
Lemma commut_lift_subst_rec : forall t u n k h, k <= h →
    lift_rec n (t {h := u}) k = (lift_rec n t k) {(n+h) := (lift_rec n u k)}.
```

où `t{h := u}` représente $t\{u/h\}$.

On démontre aussi le résultat suivant, exprimant la propriété (tout à fait attendue) que la substitution d'un terme à une variable dont il n'y a pas d'occurrence libre est sans effet :

```
Lemma subst_nothing' :
  forall u u',  forall j n, n > 0 →
   (lift_rec n u j) { j := u'} = (lift_rec n u j).
```

3.2. Les réductions

La figure 1 décrit comment nous avons décomposé les règles $\rightarrow_{\mathsf{ls}}$, $\rightarrow_{\mathsf{db}}$ et $\rightarrow_{\mathsf{gc}}$. Dans cette partie nous allons décrire les définitions de ces règles en Coq. La difficulté majeure réside dans la gestion des indices de de Bruijn.

Nous ne nous attarderons pas sur la formalisation des règles $\rightarrow_{\mathsf{db}}$ et $\rightarrow_{\mathsf{gc}}$ qui ne présentent pas de difficulté particulière. Ce sont toutes les deux des types inductifs dont le nombre de constructeurs est égal au nombre de règles obtenues après la décomposition (figure 1).

C'est surtout pour la règle $\rightarrow_{\mathsf{ls}}$ qu'une gestion particulière des indices de de Bruijn est nécessaire. Cette gestion se fait grâce à l'ajout d'un paramètre entier (i dans le code) qui représente le nombre de lieurs et nous permet ainsi d'arriver à la racine du terme. Nous avons donc les réductions $\rightarrow_u^i$ et $\rightarrow_{\mathsf{lsa}}^i$ correspondant à la figure 1 :

```
Inductive lsa_def (u :term): nat → term→ term→ Prop:=
  | LSA_Var :forall i,
      lsa_def u i (TVar i) (shift u)
  | LSA_Lam :  forall t t':term,forall i :nat,
                lsa_def (shift u) (S i) t t' →
                lsa_def u i (\t) (\t')
  | LSA_AppLeft : forall t t' s:term,forall i:nat,
                   lsa_def u i t t' →
                   lsa_def u i (t·s) (t'·s)
  | LSA_AppRight : forall t t' s:term,forall i:nat,
                    lsa_def u i t t' →
                    lsa_def u i (s·t) (s·t')
  | LSA_SubExt :  forall t t' s:term,forall i:nat,
                  lsa_def u i t t' →
                  lsa_def u i (s[t])  (s[t'])
  | LSA_SubInt : forall t t' s:term,forall i:nat,
                  lsa_def (shift u) (S i) t t' →
                  lsa_def u i (t[s])  (t'[ s]).
```

```
Inductive lsa (i:nat) : term → term → Prop :=
| LSA : forall t t' u,  lsa_def u i t t'  → lsa i (t[u]) (t'[ u]).
```

La réduction $\to_{\mathsf{ls}}$ est donc définie par $\to^{0}_{\mathsf{lsa}}$:

```
Definition ls := lsa 0.
```

Nous pouvons maintenant définir la réduction $\to$ (notée $\longrightarrow$ en Coq) :

```
Inductive red : term → term → Prop :=
  | DB : forall t t':term, t  —→ db t' → red t t'
  | LS : forall t t':term, t  —→ ls t' → red t t'
  | GC : forall t t':term, t  —→ gc t' → red t t'
  | RedContextTAbs:
      forall t t':term,
      red t t' →
      red (\t) (\t')
  | RedContextTAppLeft:
      forall t t' v:term,
      red t t' →
      red (t · v) (t' · v)
  | RedContextTAppRight:
      forall v v' t:term,
      red v v' →
      red (t · v) (t · v')
  | RedContextSubExt :
      forall t t' u:term,
      red t t' →
      red (t[u]) (t'[ u])
  | RedContextSubInt :
      forall t u u':term,
      red u u' →
      red (t[u]) (t[u']).
```

Comme énoncé antérieurement, la définition ci-dessus est une définition intuitive issue de la décomposition des règles afin de gérer plus simplement dans les preuves les indices de de Bruijn. Cependant, cette définition interagit mal avec l'opérateur de lift ; lorsqu'une preuve doit gérer simultanément la réduction $\to_{\mathsf{ls}}$ et l'opérateur de lift, il faut retarder au maximum la réindexation des indices dans la définition inductive de $\to^{x}_{u}$. Nous obtenons les nouvelles règles suivantes :

$$\frac{}{x \rightsquigarrow^{x}_{u} \uparrow^{x+1}_{0} u}\ \texttt{LSA\_Var1}$$

$$\frac{t \rightsquigarrow^{x+1}_{u} t'}{\lambda t \rightsquigarrow^{x}_{u} \lambda t'}\ \texttt{LSA\_Lam1}$$

Les autres cas de la définition de $\rightsquigarrow^{x}_{u}$ (`lsa_d_ret` en Coq) sont identiques à ceux de $\to^{x}_{u}$ à l'exception du cas `LSA_SubInt` qui est modifié de manière similaire à `LSA_Lam` (on enlève le lift).

Nous montrons que les deux définitions sont équivalentes, modulo une réindexation :

```
Lemma ls_equivalence: forall t t' u,forall i,
            lsa_def (lift i u) i t t' ↔ lsa_d_ret u i t t'.
```

L'utilisation de ce lemme d'équivalence avec $i = 0$ nous permet d'obtenir que la réduction $\rightsquigarrow_u^0$ coïncide avec $\rightarrow_u^0$ et offre donc une manière équivalente de définir $\rightarrow_{\mathsf{ls}}$. Cette définition équivalente est indispensable à la preuve des deux propriétés suivantes (dont l'intérêt sera explicité dans la suite) :

```
Lemma lsa_red : forall t t' u :term,
  lsa_def u 0 t t' → t'{0:= (shift u)} = t {0:= (shift u)}.

Lemma One_step_ls : forall t t' u,
  (t [u]) −→ ls (t' [u]) → num_occ t' 0 < num_occ t 0.
```

où `num_occ t i` est le nombre d'occurrences libres de la variable correspondant à l'indice `i` relativement à la racine de `t`.

Les preuves de `lsa_red` et `One_step_ls` se font par induction sur $\rightsquigarrow_u^i$. Même si $t[u] \rightarrow_{\mathsf{ls}} t'[u]$ est équivalent par définition à $t \rightarrow_u^0 t'$, et donc à $t \rightsquigarrow_u^0 t'$, l'induction requiert un i générique. C'est ici que l'on comprend la nécessité d'utiliser $\rightsquigarrow_u^i$: pour $\rightarrow_u^i$, le cas de base de l'induction (`LSA_Var`) est faux dès que $i > 0$.

Il est maintenant naturel de se demander si l'on peut se passer de $\rightarrow_u^x$ aussi dans le reste de la formalisation et définir $\rightarrow_{\mathsf{ls}}$ directement via $\rightsquigarrow_u^i$. Nous pensons que ce serait possible mais la définition de $\rightarrow_u^i$ nous semble plus naturelle et nous avons préféré l'utiliser prioritairement.

3.3. La propriété IE

La propriété s'énonce en Coq de la manière suivante :

```
Theorem IEProp : forall t u:term, sn u ∧ sn (t{0 := u}) → sn(t[u]).
```

Comme décrit précédemment, la preuve se fait en analysant tous les cas de réductions en une étape possible du terme `t[u]`. Le but est de démontrer `sn(w)` pour tout `w` ainsi obtenu, ce qui permet de déduire `sn(t[u])` par l'axiome `Ax1`.

Pour démontrer `sn(w)`, on utilise l'induction forte sur la paire lexicographique :

$(\eta(\texttt{t}\ \{0 := \texttt{u}\}) + \eta(\texttt{u})$, `num_occ t` $0)$.

La présence de η dans cette paire est justifiée par l'axiome `Ax2` qui énonce que *eta* décroit strictement à chaque pas de réduction.

Les lemmes 1 et 2 sont utilisés dans les cinq cas `RedContext`... de la définition inductive `red`.

Comme mentionné dans la section 2, le lemme 2 se démontre par induction sans obstacle particulier. A l'inverse, la preuve du lemme 1 présente une difficulté due aux indices de de Bruijn. En effet, il est nécessaire de démontrer que la réduction est transparente par rapport à l'opérateur de lift :

$$u \rightarrow u' \Rightarrow \uparrow u \rightarrow \uparrow u'$$

Cette propriété découle du lemme 2 une fois observé que

$$\uparrow u = u\{i_1 + 1/i_1\} \cdots \{i_n + 1/i_n\}$$

où $i_1, \ldots, i_n$ sont les indices correspondants aux variables libres de u. Le fichier `list_subst.v` est consacré à la formalisation de cette observation, non triviale à cause de la présence d'une suite de substitutions implicites de longueur arbitraire. Celle-ci est représentée par une liste de paires qui doit rester triée afin qu'une substitution n'affecte pas les substitutions précédentes, situation rendue possible par les indices de de Bruijn.

Il nous reste maintenant trois cas pour compléter la preuve de la propriété **IE** : `DB`,`LS` et `GC`. Le premier et le dernier ne comportent pas de difficulté particulière. Le deuxième est plus délicat. Nous avons déjà énoncé dans la section 3.2 les lemmes `lsa_red` et `One_step_ls`. C'est dans ce cas, celui de `LS`, que nous utilisons ces deux lemmes. Le premier affirme que, si l'on regarde une substitution explicite comme si elle était une substitution implicite, la réduction $\to_{\mathsf{ls}}$ ne fait rien : $t[u/x] \to_{\mathsf{ls}} t'[u/x]$ implique $t\{u/x\} = t'\{u/x\}$ (les indices de de Bruijn requièrent l'introduction d'un lift sur u dans la formalisation de cette propriété). La conséquence majeure est que, lors de l'application d'une étape de $\to_{\mathsf{ls}}$, la première composante de la paire utilisée pour l'induction reste inchangée. Le second affirme que la deuxième composante de la paire diminue strictement, ce qui permet d'appliquer l'induction.

4. Conclusion et travaux futurs

Nous avons formalisé et prouvé en Coq, modulo une liste de substitutions à introduire (voir les travaux futurs), la propriété **IE** (« de l'**I**mplicite à l'**E**xplicite »), pièce manquante à la preuve formelle de la propriété PSN [8]. Il s'agit d'un premier pas vers une formalisation de résultats qui concerne la correspondance entre les machines abstraites et le LSC.

Cette formalisation fait environ 1300 lignes de Coq, ce qui n'est en définitive pas très long. Néanmoins nous avons dû résoudre quelques difficultés qui, comme souvent, n'étaient pas apparentes dans la preuve papier-crayon. Il s'agit en particulier de l'utilisation des indices de de Bruijn pour formaliser les variables des termes et qui nous ont conduit, entre autres, à décomposer les règles de réduction, à introduire un entier comme paramètre alors que c'est la valeur de cet entier à 0 qui nous intéressait, à définir la règle $\to_{\mathsf{ls}}$ de deux manières différentes accompagnée de la preuve d'équivalence correspondante, etc.

Un autre point remarquable concerne la formalisation de la réduction « à distance » du LSC : les règles sont données modulo la présence d'un contexte arbitraire, même les règles de base ne sont pas locales, ce qui est beaucoup plus lourd. Ceci s'explique par le fait que le LSC réfléchit au niveau de la réécriture des termes la réécriture des réseaux de preuves de la logique linéaire, et ce qui est local dans les réseaux, ne l'est pas dans les termes.

Notre travail s'étant focalisé sur la preuve de la propriété **IE**, nous avons axiomatisé les propriétés de SN. Afin de compléter totalement cette formalisation, il reste donc à définir SN et *eta* afin de prouver les axiomes. Un autre travail futur plus conséquent consisterait à étudier la pertinence de prouver la généralisation suivante de la propriété **IE** :

$$(\mathbf{IE}) : \qquad \forall\ u \in \mathrm{SN},\ t\{u/x\}\overline{v} \in \mathrm{SN} \quad \text{implique} \quad t[u/x]\overline{v} \in \mathrm{SN},$$

où $\overline{v}$ est une suite d'arguments. Si cette suite est vide, on retrouve la formulation de **IE** traitée dans ce papier. C'est cette généralisation de **IE** qui est à présent utilisée dans la démonstration que **IE** implique PSN pour le LSC. Même si la nécessité de cette version forte de **IE** n'est pas connue actuellement (il se peut que PSN soit démontrable pour LSC directement à partir de la propriété **IE** habituelle), une formalisation complète de PSN à l'état présent des connaissances requiert la formalisation de cette version généralisée. Celle-ci ne devrait pas présenter de difficultés majeures, du moins pas plus que celles rencontrées dans la formalisation de la propriété d'équivalence entre lift et la suite de substitution implicites discutée en section 3.3.

Une dernière question potentiellement intéressante concerne l'utilisation des indices de de Bruijn. L'utilisation de Coq rend ce choix plus ou moins obligatoire, ce qui force par la suite à formuler toutes les propriétés de manière plus générale et souvent moins intuitive (typiquement, avec un paramètre $i \in \mathbb{N}$ lorsque le seul cas qui nous intéresse en pratique est $i = 0$). On peut cependant envisager une formalisation dans un assistant à la preuve capable de gérer les lieurs (et l'α-équivalence) de manière primitive (comme Abella [2]). Il serait intéressant de développer une telle formalisation et d'en étudier le rapport coût/bénéfice par rapport à celle proposée dans ce papier.

Remerciements Nous tenons à remercier Beniamino Accattoli et Delia Kesner pour leurs clarifications sur certains points techniques de cette preuve.

Références

[1] M. Abadi, L. Cardelli, P. Curien, and J. Lévy. Explicit substitutions. In *Proceedings of POPL*, pages 31–46, 1990.

[2] The Abella theorem prover.

[3] B. Accattoli. An abstract factorization theorem for explicit substitutions. In *Proceedings of RTA*, pages 6–21, 2012.

[4] B. Accattoli. Milner's calculus. Unpublished manuscript, 2012.

[5] B. Accattoli and D. Kesner. Preservation of strong normalisation modulo permutations for the structural lambda-calculus. *Logical Methods in Computer Science*, 8(1), 2012.

[6] The Coq reference manual.

[7] R. David and B. Guillaume. A lambda-calculus with explicit weakening and explicit substitution. *Mathematical Structures in Computer Science*, 11(1) :169–206, 2001.

[8] W. de Carvalho Segundo, F. L. C. de Moura, and D. Ventura. Formalizing a named explicit substitutions calculus in coq. In *Joint Proceedings of the MathUI, OpenMath and ThEdu Workshops and Work in Progress track at CICM co-located with Conferences on Intelligent Computer Mathematics (CICM 2014), Coimbra, Portugal, July 7-11, 2014.*, 2014.

[9] J. Forest. A weak calculus with explicit operators for pattern matching and substitution. In *Proiceedings of RTA*, pages 174–191, 2002.

[10] D. Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5(3), 2009.

[11] D. Kesner and S. Lengrand. Resource operators for lambda-calculus. *Inf. Comput.*, 205(4) :419–473, 2007.

[12] J. Lévy and L. Maranget. Explicit substitutions and programming languages. In *Proceedings of FSTTCS*, pages 181–200, 1999.

[13] P. Melliès. Typed lambda-calculi with explicit substitutions may not terminate. In *Proceedings of TLCA*, pages 328–334, 1995.

[14] R. Milner. Local bigraphs and confluence : Two conjectures : (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 175(3) :65–73, 2007.

Annexe : esquisse de la preuve papier-crayon

We give here a proof of the strong version of the **IE** property for the LSC, communicated to us by Beniamino Accattoli.

We start with two preliminary results:

Lemma 1. *If $u \to u'$, then $t\{u/x\} \to^* t\{u'/x\}$.*

Proof. By induction on t. □

Lemma 2. *If $t \to t'$, then $t\{u/x\} \to t'\{u/x\}$.*

Proof. By induction on the derivation of $t \to t'$. □

Theorem 1 (The IE property)**.** *Let $t, u, v_1, \ldots, v_n$ be terms, and let $w := t\{u/x\}v_1 \cdots v_n$. Then,* $\mathrm{SN}(w)$ *and* $\mathrm{SN}(u)$ *imply* $\mathrm{SN}(t[u/x]v_1 \cdots v_n)$.

Proof. In the sequel, we write $\vec{v}$ for $v_1 \ldots v_n$. The proof is by induction on the lexicographically ordered pair

$$(\eta(w) + \eta(u), |t|_x),$$

where η is the function mapping a strongly normalizing term to the length of its longest reduction sequence and $|t|_x$ denotes the number of free occurrences of x in t.

Let $t_0 := t[u/x]v_1 \cdots v_n$. To prove $\mathrm{SN}(t_0)$, it is enough to prove that, whenever $t_0 \to t_0'$, we have $\mathrm{SN}(t_0')$. We consider all possible cases, according to where the reduction is performed within t_0:

1. $u \to u'$. We conclude by applying the induction hypothesis to $t, u', \vec{v}$. In fact, $\eta(u') < \eta(u)$ and, by Lemma 1, $t\{u/x\} \to^* t\{u'/x\}$, so $\eta(t\{u'/x\}\vec{v}) \leq \eta(w)$.
2. $t \to t'$. We conclude by applying the induction hypothesis to $t', u, \vec{v}$. Indeed, thanks to Lemma 2 $t\{u/x\} \to t'\{u/x\}$, so $\eta(t'\{u/x\}\vec{v}) < \eta(w)$.
3. $v_i \to v_i'$ for some $1 \leq i \leq n$. We apply the induction hypothesis to $t, u, v_1, \ldots, v_i', \ldots, v_n$, which holds because $\eta(t\{u/x\}v_1 \cdots v_i' \cdots v_n) < \eta(w)$ by Lemma 2.
4. $t_0 \to t_0'$ by means of a gc rule applied at top-level, *i.e.*, $|t|_x = 0$ and $t_0' = tv_1 \cdots v_n = w$, so we trivially conclude because $\mathrm{SN}(w)$ by hypothesis.
5. $t_0 \to t_0'$ by means of a ls rule applied at top-level, *i.e.*, $t = C\langle x \rangle$ and $t_0' = C\langle u \rangle[u/x]v_1 \ldots v_n$. We let $t' := C\langle u \rangle$ and conclude by applying the induction hypothesis to $t', u, \vec{v}$. In fact, note that $t'\{u/x\} = t\{u/x\}$, so $t'\{u/x\}\vec{v} = w$. On the other hand, $|t'|_x < |t|_x$, because one occurrence of x was substituted and, by α-renaming, u may be assumed not to contain x.
6. $t_0 \to t_0'$ by means of a dB rule applied at top-level, *i.e.*, $t = L\langle \lambda y.r \rangle$ and $t_0' = L\langle r[v_1/y] \rangle[u/x]v_2 \ldots v_n$. We let $t' := L\langle r[v_1/y] \rangle$ and conclude by applying the induction hypothesis to $t', u, v_2, \ldots, v_n$. In fact,

$$\begin{aligned} w &= L\langle \lambda y.r \rangle\{u/x\}v_1v_2 \cdots v_n \\ &= L\{u/x\}\langle \lambda y.r\{u/x\} \rangle v_1v_2 \cdots v_n \\ &\to L\{u/x\}\langle r\{u/x\}[v_1/y] \rangle v_2 \cdots v_n \\ &= L\langle r[v_1/y] \rangle\{u/x\}v_2 \cdots v_n && (x \text{ does not appear in } v_1) \\ &= t'\{u/x\}v_2 \cdots v_n \end{aligned}$$

   so $\eta(t'\{u/x\}v_2 \cdots v_n) < \eta(w)$.

□