Preuve de la conservation des types du π-calcul en COQ

Loïc Henry-Gréard

28 août 1998

Abstract

Ce rapport présente une méthode de codage du π-calcul dans le système d'aide à la preuve COQ, en vue d'effectuer dans cet environnement des preuves de propriétés du π-calcul. La méthode employée pour coder les noms consiste en la séparation syntaxique des noms libres (paramètres) et des noms liés (variables), en vue d'éviter les renommages implicites dans les substitutions, technique employée par J.McKinna et R.Pollack [4]. La preuve effectuée ici est celle de la propriété de conservation des types (*subject reduction*) d'un π-calcul monadique typé.

Introduction

π-calcul monadique typé

Le π-calcul a été introduit en 1989 par Robin Milner, Joachim Parrow et David Walker [6], comme extension à l'algèbre de processus CCS, et ajoute à l'aspect synchronisation entre processus de celui-ci, l'expression de la mobilité, dans le but de modéliser complètement au niveau fondamental les processus concurrents. Une description précise des fondements du π-calcul et de sa sémantique se trouve dans [6]. Une extension naturelle du π-calcul monadique consiste a ne plus considérer comme valeurs simplement des noms, mais des n-uplets de noms : le *π-calcul polyadique*. La version polyadique du π-calcul expliquée par R. Milner dans [5] permet d'exprimer une première notion de discipline sur l'emploi des noms dans les processus, puisque une réduction peut s'avérer incorrecte par mauvaise arité. Une seconde contrainte sur l'emploi des noms est introduite dans les travaux de D. Sangiorgi [7] [8] et concerne la *directionnalité* des noms, c'est à dire leur capacité à être utilisés comme émetteurs, récepteurs, ou les deux (ou rien). Cette discipline d'emploi des noms est implémentée en associant des *types* à ceux-ci, et en définissant un jugement de correction d'un processus vis à vis des types des noms qu'il contient.

La preuve présentée ici comme application du codage du π-calcul est dans ce contexte la propriété de conservation des types, qui donne la préservation de ce jugement de correction du typage par la relation de réduction.

Codage des abstractions

En fonction de la syntaxe retenue pour les termes, et surtout du type de sémantique adopté, la difficulté réside principalement dans la représentation de la liaison des noms dans les termes. En π-calcul (ce calcul est détaillé dans la section suivante), cette liaison existe sous deux formes:

Les récepteurs: Dans le processus $p(x).Q$, le nom x est lié dans le terme Q, en attente d'être substitué par un autre nom avant que Q puisse agir (Q est "gardé" par p). Il s'agit d'une abstraction semblable à celle du λ-calcul.

Les restrictions Dans le processus $(\nu x : t)Q$, le nom x est uniquement lié par une contrainte de portée. Ses occurences dans Q ne sont pas destinées à être substituées, et il existe en revanche la possibilité d'augmenter la portée d'un nom, si celui-ci vient à être communiqué à l'extérieur. En fait, si l'on ne traite que des processus finis, c'est à dire sans opérateur $!P$, on conserve la même sémantique en réécrivant le terme avec un renommage judicieux des noms restreints:

$$a(x).P \mid (\nu x : t)(\bar{a}[x].Q \mid (\nu x : t')R_x) \quad \Rightarrow \quad a(x).P \mid a[b].Q \mid R_c$$

On a besoin de la restriction pour exprimer que le processus $!(\nu x : t)\bar{a}[x]$ émet un nom distinct à tous les processus qui attendent une communication sur a.

La solution choisie pour coder ces abstractions doit répondre à plusieurs objectifs : permettre d'effectuer simplement des substitutions en évitant les captures, permettre de coder simplement la relation de sémantique, et enfin d'éviter dans les preuves sur ces processus, de devoir prendre en compte l'α-conversion, c'est à dire de permettre de manipuler des termes qui représenteront le quotient de l'ensemble des processus par α-équivalence.

Un codage direct des noms par un ensemble muni d'une égalité décidable rend difficile une expression directe de l'opération de substitution, ne serait-ce que de substituer des noms pour des noms. $(\nu x)P\{y/z\}$ est défini simplement pour $x = z$ et pour $x \neq z$, $y \neq x$, mais implique un renommage pour $x \neq z$, $y = x$.

Dans le travail de D. Hirschkoff [3], le codage est effectué en utilisant les indices de de Bruijn, c'est à dire en remplaçant les noms liés par un pointeur sur leur lieur, codé par le nombre de lieurs à traverser. Cette méthode permet d'être sûr de manipuler les processus modulo l'α-équivalence, puisque deux termes α-équivalents sont indiscernables dans ce codage, mais en revanche la formalisation de la sémantique en un prédicat inductif est relativement compliquée, la plus grande partie des preuves consistant en des manipulations des indices de de Bruijn ne correspondant pas à notre représentation intuitive des termes.

La méthode adoptée ici est la transposition à la syntaxe et à la sémantique du π-calcul de celle appliquée dans [4] aux λ-calculs typés dits "purs". L'idée de base de cette méthode consiste à distinguer par deux constructeurs différents, les noms en position libre des noms en position liée.

COQ

COQ ([2]) est un outil d'aide à la preuve fondé sur le Calcul des Constructions Inductives (CCI). Il permet de manipuler interactivement des termes du CCI représentant des preuves d'assertions, et de construire ces termes par l'intermédiaire de tactiques représentatives de raisonnements mathématiques. COQ est également capable d'extraire automatiquement des programmes de preuves constructives à partir de leurs spécifications. Nous n'utiliseront pas cette possibilité ici.

La preuve de la propriété de conservation des types est très simple, une fois définis une quantité de lemmes techniques qui fournissent à COQ les outils de base pour manipuler les termes comme on le ferait naturellement pour une démonstration sur le papier. Une partie de ces lemmes est donnée en section 3, ainsi que certaines étapes de la preuve des propriétés de conservation des types, pour chaque type de transition.

Préalablement à cela, la sémantique de référence pour le π-calcul (section 1.2), ainsi que la méthode de codage dans COQ pour notre sémantique (section 1.3) seront présentés. On justifiera aussi formellement la correction de la semantique codée dans COQ. Enfin le système de typage sera introduit en section 2.

1 Sémantique

On présente dans cette section un π-calcul monadique typé, sous forme d'une part de sa syntaxe et de sa sémantique tels qu'ils sont présentés usuellement, d'autre part de son codage en vue d'effectuer des preuves en COQ.

1.1 Le π-calcul monadique typé : syntaxe

1.1.1 syntaxe de référence

Le travail de formalisation a été effectué pour un π-calcul monadique classique, auquel est rajouté une annotation explicite de type pour la restriction:

$$P \quad ::= \quad \bar{n}m.P \mid n(m).P \mid (\nu n : t)P \mid (P \mid Q) \mid !P \mid P + Q \mid [n = m]P \mid 0$$

où P,Q,... représentent les processus, n, m, ... un ensemble infini de *noms* considéré comme dénombrable. t, t',... représentent des *types*, associés aux noms. Les différents constructeurs introduits dans cette version monadique du π-calcul sont ceux de [6]. L'opérateur INPUT $n(m).P$ exprime un agent qui attend sur un canal nommé n un donnée qui sera substituée dans son corps P. L'opérateur output $\bar{n}m.P$ permet de transmettre comme donnée le nom m sur le canal n. L'opérateur $(\nu n : t)P$ restreint la visibilité du nom n au corps P. $!P$ code les processus au comportement infini (cf. [5]). On note Pr_I l'ensemble des processus syntaxiquement décrits ainsi.

1.1.2 syntaxe en COQ

Le choix effectué ici pour le codage du π-calcul dans COQ est largement inspiré de [4], où il est appliqué à un langage fonctionnel typé, en particulier en ce qui concerne le codage des variables (des *noms* dans la terminologie π-calcul). Dans ce système, qui est une alternative par exemple au codage de de Bruijn [3], les noms sont codés par un constructeur différent selon qu'ils sont en position libre dans un terme (on les appelle alors des *paramètres*), ou en position liée (ce sont alors des *variables*). On note Pr_P l'ensembles des termes du π-calcul codés de cette manière (cela correspond à l'ensemble `proc` dans notre formalisation COQ).

Les expressions COQ seront le plus souvent transcrites en des notations plus conformes à l'usage dans la littérature, en particulier un ensemble de règle d'inférence dénote naturellement une définition inductive d'un prédicat. Le code directement extrait du source COQ sera écrit `en caractères ‘‘machine à écrire`".

Les paramètres sont notés p, q, r, les variables x, y, z, les noms n, m, et les processus P, Q, R. Suivant la notation de [4], les ensembles de paramètres et des variables sont respectivement `PP` et `VV`. On note Pr_P l'ensemble des processus décrits dans COQ par la même syntaxe que celel vu précédemment, les noms en position libre étant remplacés par un type:

```
Inductive name : Set :=
  pname : PP -> name |
  vname : VV -> name.
```

On définit un prédicat

```
fresh: PP -> proc -> Prop
```

noté $p \notin \mathrm{P}(P)$, qui indique qu'un paramètre donné ne se trouve pas dans les noms libres d'un terme. On note de façon similaire $x \notin \mathrm{V}(P)$ lorsque la variable x ne se trouve pas dans les variables apparaissant dans le processus P.

On fixe par des axiomes `PP decidable` et `VV decidable` , la décidabilité de l'égalité pour les ensembles `PP` et `VV`, et par les axiomes du fichier `exists.v` le fait que l'on puisse toujours trouver un paramètre ou une variable n'apparaissant pas dans un processus donné (réserve infinie de paramètres et de variables).

La définitions des processus est la suivante en COQ:

```
Inductive proc : Set :=
  nil : proc |
  inp : name -> VV -> proc -> proc |
  out : name -> name -> proc -> proc |
  par : proc -> proc -> proc |
  res : VV -> type -> proc -> proc |
  ban : proc -> proc |
  sum : proc -> proc -> proc |
  mat : name -> name -> proc -> proc.
```

Par exemple, le processus $(\nu x : t)(\bar{p}x \mid x(y))$ sera représenté par le terme:

```
(res x t (par (out (pname p) (vname x) nil)
              (inp (vname x) y nil)))
```

en supposant bien sûr que `x` et `y` sont déclarés comme de type `VV`, et `p` de type `PP`.

Rappelons que dans le π-calcul, les lieurs sont:

- le préfixe INPUT, par exemple le processus $p(x).P$ où les occurences de x sont liées dans P,
- les restrictions, par exemple le processus $(\nu x : t)P$ où les occurences de x sont liées par la restriction.

La principale raison technique de faire cette distinction entre variables et paramètres est qu'il devient inutile de se soucier des renommages lorsque l'on substitue un paramètre à un nom à l'intérieur d'un terme. En effet, un paramètre ne peut pas être capturé, puisque les lieurs ne désignent que des variables, dont le constructeur est différent de celui des paramètres.

On verra néanmoins qu'en suivant la sémantique de référence, on est amenés également à substituer des variables pour des noms, ce qui rend possibles d'éventuelles captures (par exemple dans $(p(x).\bar{q}r)\{x/q\}$). Cette difficulté est liée aux règles OPEN/CLOSE de la sémantique qui impliquent de créer une abstraction lors de l'extrusion de portée. Elle n'apparaît pas dans [4].

1.2 Sémantique de référence

On considère les termes modulo l'α-équivalence, les opérateurs de liaison étant l'INPUT et la restriction. La relation de transition est donc considérée comme une relation paramétrée par des noms, entre $Pr_I/\simeq_\alpha$ et $Pr_I/\simeq_\alpha$.

La formalisation s'appuie sur une sémantique opérationnelle structurée décrite fig. 1, c'est à dire sur une système de transitions étiquetées des processus. On n'a pas représenté ici les règles symmétriques des règles PARl, SUMl, COMl et CLOSEl.

Une alternative pour représenter le comportement des termes du π-calcul est l'utilisation d'un ensemble de règles de réduction (non étiquetées), prises modulo une relation d'équivalence (congruence structurelle) notée $\equiv$. Cette méthode a le mérite de séparer les évolutions des processus, représentant des calculs, des règles structurelles par exemple de commutativité de l'opérateur parallèle, ou d'extrusion de portée (exportation de noms, cf. fig. 1, règles OPEN et CLOSE). Néanmoins, la règle: $\dfrac{P \equiv P' \longrightarrow Q' \equiv Q}{P' \longrightarrow Q'}$ rend plus compliquées les preuves construites inductivement sur la dérivation d'une réduction. En outre, la notion de réduction ainsi définie se limite aux réductions internes, alors que la SOS rend compte des possibilités d'interaction d'un processus inséré dans un contexte.

La propriété de conservation des types sur de telles transitions extérieures indique non seulement la conservation de l'environnement de typage, mais le cas échéant l'évolution de l'environnement de typage. Ainsi il y aura une propriété différente pour chaque type de transition.

Les actions retenues pour caractériser les transitions sont les mêmes quatre types d'action que [6] :

$$\mu \quad ::= \quad xy \mid \bar{x}y \mid \bar{x}(y:t) \mid \tau$$

soit respectivement l'*input*, l'*output*, le *bound output* et l'action silencieuse. L'action *bound output* représente l'envoi d'un nom nouveau pour son récepteur, et en conséquence, en plus du nom destiné à être lié par *extrusion de portée*, l'action doit être paramétrée par le type du nom exporté.

$$\text{INPUT} \quad \frac{}{x(y).X \xrightarrow{xz} X\{z/y\}}$$

$$\text{OUTPUT} \quad \frac{}{\bar{x}y.X \xrightarrow{\bar{x}y} X}$$

$$\text{COMl} \quad \frac{P \xrightarrow{xy} P' \qquad Q \xrightarrow{\bar{x}y} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\text{OPEN} \quad \frac{P \xrightarrow{\bar{x}y} P'}{(\nu y : t)P \xrightarrow{\bar{x}(y:t)} P'} \qquad x \neq y$$

$$\text{CLOSEl} \quad \frac{P \xrightarrow{xy} P' \qquad Q \xrightarrow{\bar{x}(y:t)} Q'}{P|Q \xrightarrow{\tau} (\nu y : t)(P'|Q')} \qquad y \notin \mathrm{fv}(P)$$

$$\text{RES} \quad \frac{P \xrightarrow{\mu} P'}{(\nu x : t)P \xrightarrow{\mu} (\nu x : t)P'} \qquad x \notin \mathrm{n}(\mu)$$

$$\text{PARl} \quad \frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q} \qquad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset$$

$$\text{BANG} \quad \frac{!P|P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'}$$

$$\text{PLUSl} \quad \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}$$

$$\text{MATCH} \quad \frac{P \xrightarrow{\mu} P'}{[x = x]P \xrightarrow{\mu} P'}$$

Figure 1: *Sémantique d'un π-calcul monadique*

Les conditions auxiliaires apposées aux règles d'inférence concernent le problème d'extrusion de portée. La condition $x \neq y$ sur la règle OPEN indique qu'un processus ne peut pas émettre sur un canal privé, ce qui rend par exemple le processus $(\nu x : t)\bar{x}[y]$ équivalent à 0 en terme de comportement (bisimilarité). La condition sur CLOSE indique qu'un processus qui reçoit un nom exporté reçoit un nom nouveau pour lui. C'est une condition anti-capture, qui peut obliger à un renommage. La condition sur la règle RES exprime que le nom restreint à un processus étant "invisible de l'extérieur", par définition il ne peut ni émettre, ni recevoir, ni utiliser ce nom comme canal de communication. Enfin la condition sur PAR est une condition contre une éventuelle capture d'un nom de Q. En effet, Une action de *bound output* agit comme un lieur du nom exporté sur le processus résultant. La règle PAR correspond a une augmentation structurelle du processus résultant, il faut donc considérer les noms de Q comme susceptibles d'être liés. Les règle PLUS et MATCH correspondent également à une augmentation du processus original, mais le processus résultant est identique, elles n'ont donc pas besoin de telles conditions auxiliaires.

1.3 Sémantique avec paramètres

On construit la relation de transitions étiquetées dans COQ à partir de la sémantique de référence que l'on s'est fixée. Les actions restent les mêmes, les noms qui apparaissent dans les actions sont des paramètres:

```
Inductive act : Set :=
  aout : PP -> PP -> act |
  ainp : PP -> PP -> act |
  about : PP -> PP -> type -> act |
  tau : act.
```

On conserve, comme dans [4], comme invariant la propriété `closed`, des processus entièrement paramétrés des deux côtés de la flèche de réduction. Ce prédicat vérifie que le processus ne contient pas de variables représentant des noms libres (comme par exemple x dans $p(y).P_y \mid \bar{x}q$). Ces processus représentent les processus "bien formés".

Les règles d'inférence sont adaptées à partir de la sémantique de référence pour devenir les constructeurs d'une propriété inductive de COQ

```
sem : proc -> act -> proc -> Prop
```

extrusion de portée La règle OPEN remplace la variable dont le lieur a disparu par un paramètre frais. Au moment où on recrée la restriction

INPUT
$$\frac{}{p(x).P \xrightarrow{pq} P\{q/x\}}$$

OUTPUT
$$\frac{}{\bar{p}q.P \xrightarrow{\bar{p}q} P}$$

COMl
$$\frac{P \xrightarrow{pq} P' \qquad Q \xrightarrow{\bar{p}q} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

OPEN
$$\frac{P\{q/x\} \xrightarrow{\bar{p}q} P'}{(\nu x : t)P \xrightarrow{\bar{p}(q:t)} P'} \qquad \begin{array}{l} q \notin \mathrm{P}(P) \\ p \neq q \end{array}$$

CLOSEl
$$\frac{P \xrightarrow{pq} P' \qquad\qquad Q \xrightarrow{\bar{p}(r:t)} Q'}{P|Q \xrightarrow{\tau} (\nu x : t)(P'\{x/q\}|Q'\{x/r\})} \qquad \begin{array}{l} q \notin \mathrm{P}(P) \\ x \notin \mathrm{V}(P') \cup \mathrm{V}(Q') \end{array}$$

RES
$$\frac{\forall q \quad P\{q/x\} \xrightarrow{\mu} P'\{q/y\}}{(\nu x : t)P \xrightarrow{\mu} (\nu y : t)P'}$$

PARl
$$\frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q} \qquad \begin{array}{l} \mu = \bar{p}(q : t) \Rightarrow \\ q \notin \mathrm{P}(Q) \end{array}$$

BANG
$$\frac{!P|P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'}$$

PLUSl
$$\frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}$$

MATCH
$$\frac{P \xrightarrow{\mu} P'}{[p = p]P \xrightarrow{\mu} P'}$$

Figure 2: *π-calcul monadique avec substitution de paramètres*

(règle CLOSE), on remplace ces paramètres par la variable liée, en s'assurant qu'il n'y a pas de capture possible, c'est à dire que la variable est "nouvelle". Il faut noter que cette condition auxiliaire invalide la propriété $\xrightarrow{\mu} \circ \simeq_\alpha = \xrightarrow{\mu}$. En effet, on a:

$$\begin{array}{rcl} (\nu x:t)\bar{p}[x] \mid p(y).\bar{y}[q] \mid (\nu x:t)P & \xrightarrow{\tau} & (\nu y:t)(0 \mid \bar{y}[q] \mid (\nu x:t)P) \\ & \simeq_\alpha & (\nu x:t)(0 \mid \bar{x}[q] \mid (\nu x:t)P) \end{array}$$

Alors qu'on ne peut pas obtenir:

$$(\nu x:t)\bar{p}[x] \mid p(y).\bar{y}[q] \mid (\nu x:t)P \xrightarrow{\tau} (\nu x:t)(0 \mid \bar{x}[q] \mid (\nu x:t)P)$$

restrictions La sémantique du π-calcul exprime par la règle RES la même chose que la règle ξ du λ-calcul, traitée dans [4]. Elle s'appuie sur un codage de l'abstraction par l'ensemble des substitutions possibles de la variable liée, explicité par [1]. Il faut noter que cette règle est la même que le règle P$\tilde{\text{R}}$1-BIND de [4], et qu'une relation de transition, extensionellement équivalente, peut être obtenue avec un principe d'induction plus faible en adoptant la règle d'inférence associée à une condition auxiliaire:

$$\frac{P\{p/x\} \xrightarrow{\mu} P'\{p/y\}}{(\nu x:t)P \xrightarrow{\mu} (\nu y:t)P'} \quad p \notin \mathrm{P}(P) \land p \notin \mathrm{P}(\mu)$$

L'équivalence, prouvée dans [4] par induction sur la longueur des termes, n'a pas été reformalisée dans le cadre de cette étude.

1.4 Correction du codage

On justifie le codage du π-calcul effectué en COQ, en formalisant sa correspondance avec la sémantique usuelle du π-calcul. Le but de cette formalisation est de justifier en particulier la correction des conditions auxiliaires et des quantifications sur les paramètres apportées aux règles transcrites. De même que `PP` et `VV` représentent les ensembles de paramètres et de variables intervenant dans le codage, on note `NN` l'ensembles des noms intervenant dans la notation informelle (figure 1). On suppose les ensembles `PP`, `VV` et `NN` dénombrables et on choisit $b_P : \mathtt{NN} \to \mathtt{PP}$ une bijection de l'ensembles des noms vers celui des paramètres. On note $b_I = b_P^{-1}$.

On peut alors construire une bijection entre les classes d'α-équivalence de l'ensemble des processus décrits par la syntaxe usuelle (Pr_I) et de l'ensemble des processus décrits par la syntaxe paramétrée (Pr_P). Soit $T_P : Pr_I \to Pr_P$ cette traduction. Les actions μ intervenant dans la relation semantique de la figure 1 peuvent également être traduites en actions parametrées intervenant dans la sémantique de la figure 2. Cette traduction est notée $b_P(\mu)$.

Pour établir la fonction de traduction T_P, on définit d'abord une suite $(x_n)_{n\in\mathbf{N}}$ d'éléments distincts deux à deux de VV. Puis on définit:

$$\begin{array}{rcl}
T_P(n(m).Q,N) & = & b_P(n)(x_N).(T_P(Q,N+1)\{x_N/b_P(m)\}) \\
T_P(\bar{n}m.Q,N) & = & \overline{b_P(n)}b_P(m).T_P(Q,N) \\
T_P((\nu n:t)Q,N) & = & (\nu x_N:t)T_P(Q,N+1)\{x_N/b_p(n)\} \\
T_P(P \mid Q,N) & = & T_P(P,N) \mid T_P(Q,N) \\
 & . & \\
 & . & \\
 & . &
\end{array}$$

Lemme 1.4.1 *Pour tout processus $P \in Pr_I$, et pour tous entiers naturels N et M, on a*

$$T_P(P,M) \simeq_\alpha T_P(P,N)$$

Pour définir la fonction inverse de T_P, $T_I : Pr_P \to Pr_I$, on prend une suite $(p_N)_{N\in M}$ d'éléments de PP deux à deux distincts. Les termes étants syntaxiquement finis, l'ensemble $Pars(P) = \{N \in \mathbf{N} \mid p_N \in \mathbf{fn}(P)\}$ est toujours fini. On note $N_0^P = \max Pars(P) + 1$. On note (n_N) la suite des $(b_I(p_N))$. On définit la fonction T_I:

$$\begin{array}{rcl}
T_I\,(p(x).Q,N) & = & b_I(p)(n_N).T_I(Q\{p_N/x\},N+1) \\
T_I\,(\bar{p}q.Q,N) & = & \overline{b_I(p)}b_I(q).T_I(Q,N) \\
T_I\,((\nu x:t)Q,N) & = & (\nu n_N:t)T_I(Q\{p_N/x\},N+1) \\
T_I\,(P \mid Q,N) & = & T_I(P,N) \mid T_I(Q,N) \\
 & . & \\
 & . & \\
 & . &
\end{array}$$

Propriété 1.4.1 *Si P, P' sont deux processus de Pr_I, et μ une action, et si l'on a la transition dans la sémantique usuelle, $P \xrightarrow{\mu} P'$, alors on peut construire une transition $Q \xrightarrow{b_P(\mu)} Q'$ de la sémantique paramétrée telle que $Q \simeq_\alpha T_P(P,0)$ et $Q' \simeq_\alpha T_P(P',0)$.*

La preuve se fait pas induction sur la dérivation de la transition dans la sémantique de la figure 1. Par exemple pour la règle INPUT, on a

$$n(m).Q \xrightarrow{nk} Q\{k/m\}$$

Avec $T_P(n(m).Q,0) = b_P(n)(x_0).(T_P(Q,1)\{x_0/b_P(m)\})$, qui, selon la sémantique codée dans COQ, se réduit en:

$$b_P(n)(x_0).(T_P(Q,1)\{x_0/b_P(m)\}) \xrightarrow{b_P(n)b_P(k)} T_P(Q,1)\{x_0/b_P(m)\}\{b_P(k)/x_0\}$$

Pour la règle RES : si $P \xrightarrow{\mu} P'$, alors par hypothèse de récurrence, il existe Q et $Q' \in Pr_P$ tels que $Q \xrightarrow{b_P(\mu)} Q'$. Soit $p = b_P(m)$, on a par hypothèse $p \notin \mathrm{P}(b_P(\mu))$, donc $Q\{q/p\} \xrightarrow{b_P(\mu)} Q'\{q/p\}$ pour tout paramètre q. Or $Q\{q/p\} = Q\{x_N/p\}\{q/x_N\}$ pour tout pour un N pris suffisamment grand pour éviter les captures. De même pour Q'. On peut donc appliquer la règle RES de la figure 2:

$$\frac{\forall q \quad Q\{x_N/p\}\{q/x_N\} \xrightarrow{b_P(\mu)} Q\{x_N/p\}\{q/x_N\}}{(\nu x_N : t)Q\{x_N/p\} \xrightarrow{b_P(\mu)} (\nu x_N : t)Q'\{x_N/p\}}$$

Or

$$T((\nu m : t)P, N) = (\nu x_N : t)T(P, N+1)\{x_N/p\}$$

donc comme $Q \simeq_\alpha T(P, N+1)$, on a bien des processus de même α-classe que les traductions.

Pour la règle OPEN : par hypothèse : $\dfrac{P \xrightarrow{\bar{n}m} P'}{(\nu m : t)P \xrightarrow{\bar{n}(m:t)} P'} \quad n \neq m$

On note $p = b_P(n)$ et $q = b_P(m)$. Par hypothèse de récurrence, on a également $Q \xrightarrow{\bar{p}q} Q'$ avec Q et Q' α-convertibles avec les traductions de P et Q. L'hypothèse $p \neq q$ est vérifiée car b_P est une bijection. En prenant N assez grand pour éviter les captures, on a donc $Q\{x_N/q\}\{q/x_N\} \xrightarrow{\bar{p}q} Q'$. La condition $q \notin \mathrm{P}(Q\{x_N/q\})$ étant toujours respectée, on peut appliquer la règle de transition et obtenir que $(\nu x_N : t)Q\{x_N/q\} \xrightarrow{\bar{p}(q:t)} Q'$, d'où la conclusion.

Pour la règle CLOSE : si l'on a:

$$\frac{P \xrightarrow{nm} P' \qquad Q \xrightarrow{\bar{n}(m)} Q'}{P \mid Q \xrightarrow{\tau} (\nu m : t)(P' \mid Q')}$$

On note $p = b_P(n)$ et $q = b_P(m)$. Par hypothèse de récurrence $R_P \xrightarrow{pq} R_{P'}$ et $R_Q \xrightarrow{\bar{p}(q)} R_{Q'}$. L'hypothèse $q \notin \mathrm{P}(Q_P)$ est vérifiée car $m \notin \mathtt{fv}(P)$. Soit N assez grand pour que l'hypothèse $x_N \notin \mathrm{V}(R_{P'}) \cup \mathrm{V}(R_{Q'})$ soit vérifiée. On a donc d'après la règle CLOSE:

$$R_P \mid R_Q \xrightarrow{\tau} (\nu x_N : t)(R_{P'}\{x_N/q\} \mid R_{Q'}\{x_N/q\})$$

On peut alors vérifier que les deux membres de la transition sont bien α-équivalents aux traductions attendues.

Propriété 1.4.2 *Si on a une transition $P \xrightarrow{\mu} P'$ de la sémantique avec paramètres, la transition*

$$T_I(P, N_0^P) \xrightarrow{b_I(\mu)} T_I(P', N_0^{P'})$$

est une transition valide suivant les règles de la figure 1 (modulo $\simeq_\alpha$).

Là aussi la preuve se fait règle par règle, avec par exemple pour la règle INPUT: On note $n = b_I(p)$, $k = b_I(q)$ et $N = \max N_0^{p(x).P} N_0^{P\{q/x\}}$ On a:

$$T_I(p(x).P, N) = n(n_N).T_I(P\{p_N/x\}, N+1)$$

Qui se réduit d'après les règles de la figure 2 en:

$$n(n_N).T_I(P\{p_N/x\}, N+1) \xrightarrow{\bar{n}k} T_I(P\{p_N/x\}, N+1)\{k/n_N\}$$

Or $T_I(P\{p_N/x\}, N+1)\{k/n_N\} = T_I(P\{p_N/x\}\{q/p_N\}, N+1)$, et également $p_N \notin \mathrm{P}(P)$, donc

$$T_I(p(x).P, N) \xrightarrow{\bar{n}k} T_I(P\{q/x\}, N+1)$$

Pour la règle RES:

$$\frac{\forall q \quad P\{q/x\} \xrightarrow{\mu} P'\{q/y\}}{(\nu x : t)P \xrightarrow{\mu} (\nu y : t)P'}$$

Par hypothèse de récurrence, pour tout paramètre q, on a:

$$T_I(P\{q/x\}, N) \xrightarrow{b_I(\mu)} T_I(P'\{q/y\}, N)$$

donc comme $p_N \notin \mathrm{P}(P)$, pour un q_0 tel que $q_0 \notin \mathbf{n}(\mu)$:

$$T_I(P\{p_N/x\}\{q_0/p_N\}, N) \xrightarrow{b_I(\mu)} T_I(P'\{p_N/y\}\{q_0/p_N\}, N)$$

Si $n = b_I(q_0)$, l'hypothèse $n \notin \mathbf{n}(b_I(\mu))$ est vérifiée, et l'on a donc:

$$(\nu n : t)T_I(P\{p_N/x\}, N)\{n/n_N\} \xrightarrow{b_I(\mu)} (\nu n : t)T_I(P'\{p_N/y\}, N)\{n/n_N\}$$

Les membres gauche et droite de cette relation sont α-équivalents aux traductions attendues.

2 Le système de types

Les propriétés auxquelles doit répondre le système de types sont présentées dans cette section. Le système de types peut être implémenté de plusieurs manières, permettant ou pas de construire pour tous les processus un environnement de typage correct.

2.1 Les types

Le système de type utilisé dans notre π-calcul monadique transporte pour chaque nom deux informations:

TypNil $$\frac{}{\Gamma \vdash 0}$$

TypInp $$\frac{\begin{array}{c}\forall q \notin \mathrm{P}(P) \\ <\Gamma(p)>\leq_c I \qquad \Gamma \oplus (q \mapsto [\Gamma(p)]) \vdash P\{q/x\}\end{array}}{\Gamma \vdash p(x).P}$$

TypOut $$\frac{<\Gamma(p)>\leq_c O \qquad \Gamma(q) \leq [\Gamma(p)] \qquad \Gamma \vdash P}{\Gamma \vdash \bar{p}[q].P}$$

TypRes $$\frac{\begin{array}{c}\forall q \notin \mathrm{P}(P) \\ \Gamma \oplus (q \mapsto t) \vdash P\{q/x\}\end{array}}{\Gamma \vdash (\nu x : t)P}$$

TypPar $$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P|Q}$$

TypBan $$\frac{\Gamma \vdash P}{\Gamma \vdash !P}$$

TypPlus $$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P + Q}$$

TypMatch $$\frac{<\Gamma(p)>=<\Gamma(q)>= B \qquad \Gamma \vdash P}{\Gamma \vdash [p = q]P}$$

Figure 3: *Règles de typage pour le π-calcul monadique*

- Une information de directionnalité notée $< t >$, et appelée *capacité* de t. Cette information exprime simplement la capacité du nom à être utilisé comme canal pour une communication. Les capacités R (Read), W (Write), B (Both) et N (Nil), sont reliés en un petit treillis par une relation de sous-typage des capacités notée $\leq_c$:

$$\begin{tikzcd} & \mathrm{N} \arrow[dl, dash] \arrow[dr, dash] & \\ \mathrm{R} \arrow[dr, dash] & & \mathrm{W} \arrow[dl, dash] \\ & \mathrm{B} & \end{tikzcd}$$

  Les noms de capacités R peuvent être en position de récepteurs, comme p dans $p(x).P$, ceux de capacités W peuvent être émetteurs, comme p dans $\bar{p}[q].P$, ceux de capacité B peuvent donc être récepteurs et émetteurs, et ceux de capacités N ne peuvent être ni récepteurs ni émetteurs, mais peuvent être présents dans le terme, comme q dans $\bar{p}[q].P$.

- Une information sur le type des noms pouvant circuler par un canal désigné par le nom en question. Ce type est noté $[t]$. Dans le cadre d'un π-calcul monadique, $[t]$ représente lui-même un type puisque les seuls objets pouvant être transmis sont des noms. Si on étend le cadre au π-calcul polyadique par exemple, $[t]$ sera un ensemble de types représentant les contraintes sur l'ensemble de noms transmis simultanément par un canal (y compris une information d'arité).

En prenant comme données un certain ensemble de types (donnée de type `Set` en COQ) et des fonctions $< \_ >$ et $[\_]$, on définit des axiomes sur la relation de sous-typage, fondés sur la covariance des types récepteurs et la contravariance des types émetteurs (fig. 4). On suppose de plus la relation de sous-typage transitive.

On remarque que quelle que soit la représentation des types, ceux-ci ne peuvent pas former un inf-demi treillis, puisque par exemple deux types t_1 et t_2 tels que $< t_1 >=< t_2 >= B$ et $[t_1] \neq [t_2]$ n'ont pas de minimum (il est impossible de trouver un type qui soit à la fois sous-type de l'un et de l'autre).

2.2 Les environnements

Les processus sont dits correctement typés (fig 3) ou non dans un environnement qui fournit pour chacun de leur noms libres (c'est à dire de leurs paramètres), un type exprimant ce qu'ils peuvent transmettre, ainsi que la direction dans laquelle ils peuvent le transmettre. Ce jugement est noté

STRefl
$$\frac{}{t \leq t}$$

STNil
$$\frac{< t' >= N}{t \leq t'}$$

STRead
$$\frac{< t_1 > \leq_c R \qquad < t_2 >= R \qquad [t_1] \leq [t_2]}{t_1 \leq t_2}$$

STWrite
$$\frac{< t_1 > \leq_c W \qquad < t_2 >= W \qquad [t_2] \leq [t_1]}{t_1 \leq t_2}$$

Figure 4: *Règles de sous-typage pour les noms du π-calcul*

$\Gamma \vdash P$ ([8]). Ces environnements notés Γ, Δ, sont des fonctions totales de l'ensemble des paramètres vers l'ensemble des types.

exemple: Dans Γ tel que $< \Gamma(p) >= B$, $< \Gamma(q) >= N$, $< \Gamma(r) >= N$, les processus $\bar{p}[q]$ et $p(x).\bar{x}[r]$ sont correctement typés individuellement, mais dans le premier cas, il faut $< [\Gamma(p)] >= N$ à cause de la contravariance pour le typage des noms capables démettre, et dans le second cas $< [\Gamma(p)] > \leq_c W$ à cause de la règle de typage. Le processus $\bar{p}[q] \mid p(x).\bar{x}[r]$ Ne peut donc être correctement typé dans un environnement qui interdit à q d'émettre. Pour des arguments détaillés sur la correction de l'information de directionalité dans le typage, vis à vis de la "mauvaise" utilisation des canaux dans les processus, voir [7].

On note $\oplus$ (opérateur binaire, associatif à gauche), l'opération telle que $\Gamma \oplus (p \mapsto t)$ est la fonction des paramètres dans les types, qui à p associe t et à tout $q \neq p$ associe $\Gamma(q)$.

3 Preuve de la conservation des types

Les preuves des propriétés d'autoréduction se font très simplement par induction sur la dŕivation des transitions étiquetées, après avoir défini un ensemble de résultats techniques dont les plus significatifs sont donnés ci-après. Bien sûr ces résultats font eux-même appel a quantité d'autres lemmes.

3.1 Lemmes sur les substitutions

Ces lemmes, ainsi que ceux de la section suivante, sont purement techniques, et leur démonstration en COQ est triviale, par induction sur la structure des processus dans la plupart des cas, sachant que la démonstration est alors allégée en définissant préalablement un lemme semblable sur les `name`, pour factoriser la vérification de la propriété sur les processus comprenant des noms libres (*output*, *input* et *match*).

Lemme 3.1.1 (subs par after subs var)

$$\forall P, p, q, x \quad q \notin P(P) \Longrightarrow P\{q/x\}\{p/q\} = P\{p/x\}$$

Lemme 3.1.2 (subs var after subs par)

$$\forall P, p, q, x \quad x \notin V(P) \Longrightarrow P\{x/p\}\{q/x\} = P\{q/p\}$$

3.2 Lemmes sur les paramètres

Lemme 3.2.1 (fresh after subs)

$$\forall P, p, q, x \quad p \notin P(P) \wedge p \neq q \Longrightarrow p \notin P(P\{q/x\})$$

Lemme 3.2.2 (fresh before subs)

$$\forall P, p, q, x \quad p \notin P(P\{q/x\}) \Longrightarrow p \notin P(P)$$

Lemme 3.2.3 (listening on known name)

$$\forall P, P', p, q \quad P \xrightarrow{pq} P' \Longrightarrow p \in P(P)$$

Des propriétés semblables sont bien sûr vraies pour les actions output et bound output.

Lemme 3.2.4 (fresh after trans)

$$\forall P, \mu, P' \quad P \xrightarrow{\mu} P' \Longrightarrow \forall p\; p \notin P(P) \wedge p \notin P(\mu) \Longrightarrow p \notin P(P')$$

3.3 Lemmes de sous-typage

Lemme 3.3.1 (subtyping extension)

$$\forall \Gamma, \Delta, p, t_1, t_2 \quad \Delta \leq_e \Gamma \wedge t_1 \leq t_2 \Longrightarrow \Delta \oplus (p \mapsto t_1) \leq_e \Gamma \oplus (p \mapsto t_2)$$

Lemme 3.3.2 (subtyping)

$$\forall \Gamma, P \quad \Gamma \vdash P \Longrightarrow \forall \Delta\; \Delta \leq_e \Gamma \Longrightarrow \Delta \vdash P$$

3.4 Les Interversions de paramètres

On introduit pour les besoins de la preuve l'opérateur d'interversion de paramètres sur les processus, environnements, noms et paramètres : `swap_proc`, `swap_env`, `swap_name` et `swap_par`. Cette opération est dans tous les cas notée $X\{p \leftrightarrow q\}$ où X est un processus, environnement, nom ou paramètre. Cette opération consiste à substituer à toutes les occurrences de p, une occurrence de q, et à toutes les occurrences de q, une occurrence de p. Dans le cas des environnements, $\Gamma\{p \leftrightarrow q\}$ représente la fonction qui à p associe $\Gamma(q)$, à q associe $\Gamma(p)$, et à tout r autre que p et q associe $\Gamma(r)$.

Un certain nombre de propriétées sont prouvées pour cet opérateur, notamment:

Lemme 3.4.1 (typing after swap)

$$\forall \Gamma, P \quad \Gamma \vdash P \Longrightarrow \forall p, q \quad \Gamma\{p \leftrightarrow q\} \vdash P\{p \leftrightarrow q\}$$

3.5 Lemmes de typage

Lemme 3.5.1 (addenv unused name)

$$\forall \Gamma, P \quad \Gamma \vdash P \Longrightarrow \forall t, q \notin \mathcal{P}(P) \quad \Gamma \oplus (q \mapsto t) \vdash P$$

Lemme 3.5.2 (redundant addenv)

$$\forall \Gamma, P \quad \Gamma \vdash P \Longrightarrow \forall \Delta, p, t\, p \notin \mathcal{P}(P) \wedge \Gamma =_e \Delta \oplus (p \mapsto t) \Longrightarrow \Delta \vdash P$$

Lemme 3.5.3 (type with other subs)

$$\forall \Gamma, P, p, q, x, t \quad p \notin \mathcal{P}(P) \wedge q \notin \mathcal{P}(P) \wedge \Gamma \oplus (p \mapsto t) \vdash P\{p/x\} \Longrightarrow \Gamma \oplus (q \mapsto t) \vdash P\{q/x\}$$

preuve C'est la preuve de ce lemme qui fait intervenir les interversions de paramètres dans les termes. En effet (en se plaçant dans le cas non trivial où $p \neq q$), de

$$\Gamma \oplus (p \mapsto t) \vdash P\{p/x\}$$

on déduit (3.4.1)

$$(\Gamma \oplus (p \mapsto t))\{p \leftrightarrow q\} \vdash P\{p/x\}\{p \leftrightarrow q\}$$

Or

$$(\Gamma \oplus (p \mapsto t))\{p \leftrightarrow q\} =_e \Gamma \oplus (q \mapsto t) \oplus (p \mapsto \Gamma(q))$$

d'où la conclusion puisque p n'apparaît pas dans $P\{q/x\}$.

Lemme 3.5.4 (subs typing)

$$\forall \Gamma, P \quad \Gamma \vdash P \Longrightarrow \forall p, q \quad \Gamma(p) \leq \Gamma(q) \Longrightarrow \Gamma \vdash P\{p/q\}$$

3.6 Transitions INPUT

La propriété prouvée pour les réductions OUTPUT est la suivante:

$$P \xrightarrow{pq} P' \Longrightarrow \Gamma \vdash P \Longrightarrow \Gamma(q) \leq [\Gamma(p)] \Longrightarrow \Gamma \vdash P'$$

La preuve se fait par induction sur la relation de transitions, seuls les cas pouvant générer des actions INPUT ayant à être envisagés. Dans chaque cas, on fait une inversion du prédicat de typage en fonction de la syntaxe des processus impliquée dans une règle de transition donnée.

3.6.1 Cas de la règle INPUT

On a $\Gamma \vdash p(x).Q$, donc pour tout $s \notin \mathrm{P}(Q)$,

$$\Gamma \oplus (s \mapsto [\Gamma(p)]) \vdash Q\{s/x\}$$

On déduit également des hypothèses:

$$\Gamma \oplus (s \mapsto [\Gamma(p)])(q) \leq \Gamma \oplus (s \mapsto [\Gamma(p)])(s)$$

(pour simplifier la preuve en COQ, on fait en plus l'hypothèse que s est différent de q).

On peut donc substituer dans la relation de typage, le paramètre q au paramètre s (lemme 3.5.4):

$$\Gamma \oplus (s \mapsto [\Gamma(p)]) \vdash Q\{s/x\}\{q/s\}$$

Or s est absent du terme typé, on peut donc oublier de l'ajouter dans l'environnement (lemme 3.5.2):

$$\Gamma \vdash Q\{s/x\}\{q/s\}$$

Le fait que $Q\{q/x\} = Q\{s/x\}\{q/s\}$ permet de conclure (lemme 3.1.1).

3.6.2 Cas de la règle RES

La preuve est la même pour la rêgle RES quel que soit le type de réduction, donc elle ne sera explicitée que pour les INPUT.

Si $\Gamma \vdash (\nu x : t)P$, on veut obtenir pour un $r \notin \mathrm{P}(P')$ quelconque,

$$\Gamma \oplus (r \mapsto t) \vdash P'\{r/y\}$$

. Soit $s \notin \mathrm{P}(P)$, avec s différent de p, de q et de r. Or:

$s \notin \mathrm{P}(P\{r/x\})$ (lemme 3.2.1) $\Longrightarrow$
$s \notin \mathrm{P}(P'\{r/y\})$ (lemme 3.2.4), $\Longrightarrow$
$s \notin \mathrm{P}(P')$ (lemme 3.2.2).

On peut donc appliquer le lemme 3.5.3 et conclure si l'on a:

$$\Gamma \oplus (s \mapsto t) \vdash P'\{s/y\}$$

Or l'hypothèse de récurrence est:

$$\forall r \quad \Delta \vdash P\{r/x\} \Longrightarrow \Delta(q) \leq [\Delta(p)] \Longrightarrow \Delta \vdash P'\{r/y\}$$

On remplaçe dans cette proposition Δ par $\Gamma \oplus (s \mapsto t)$ et r par s. La premiere condition est vraie car $\Gamma \vdash (\nu x : t)P$, la seconde hypothèse est vraie car $s \neq p$ et $s \neq q$.

3.7 Transitions OUTPUT

Le résultat sur les transitions OUTPUT est en deux partie:

- Un résultat de préservation des types:

$$P \xrightarrow{\bar{p}q} P' \Longrightarrow \Gamma \vdash P \Longrightarrow \Gamma \vdash P'$$

- Une contrainte sur l'environnement qui résulte de la correction du typage initial:

$$P \xrightarrow{\bar{p}q} P' \Longrightarrow \Gamma \vdash P \Longrightarrow \Gamma(q) \leq [\Gamma(p)]$$

3.8 Transitions Bound OUTPUT

On prouve comme pour OUTPUT les deux propriétés suivantes:

- une propriété de préservation des types:

$$P \xrightarrow{\bar{p}(q:t)} P' \Longrightarrow \Gamma \vdash P \Longrightarrow \Gamma \oplus (q \mapsto t) \vdash P'$$

- une transition *bout* exporte un type compatible avec l'environnement de typage:

$$P \xrightarrow{\bar{p}(q:t)} P' \Longrightarrow \Gamma \vdash P \Longrightarrow t \leq [\Gamma(p)]$$

3.8.1 Cas de la règle OPEN

Par hypothèse, on a:

$$\frac{P\{q/x\} \xrightarrow{\bar{p}q} P'}{(\nu x : t)P \xrightarrow{\bar{p}(q:t)} P'} \quad \text{pour } q \notin \mathrm{P}(P) \text{ et } p \neq q. \quad \text{ainsi que } \Gamma \vdash (\nu x : t)P.$$

préservation des types La propriété de préservation des types découle directement de la propriété similaire de l'action OUTPUT: $\Gamma \vdash (\nu x : t)P$, donc $\forall r$ n'apparaissant pas dans P, $\Gamma \oplus (r \mapsto t) \vdash P\{r/x\}$ (par la règle de typage TypRes), donc en particulier pour q, donc en appliquant la propriété de conservation des types prouvée pour les transitions OUTPUT à la transition $P\{q/x\} \xrightarrow{\bar{p}q} P'$ on obtient:

$$\Gamma \oplus (q \mapsto t) \vdash P'$$

exportation d'un type correct On utilise également la propriété similaire de OUTPUT: on en déduit en effet

$$(\Gamma \oplus (q \mapsto t))\,(q) \leq [(\Gamma \oplus (q \mapsto t))\,(p)]$$

Or $p \neq q$, donc on peut en conclure :

$$t \leq [\Gamma(p)]$$

3.8.2 Cas de la règle PAR

On a besoin ici de notre condition auxiliaire sur la règle d'inférence: Prenons la règle PARl:

$$\frac{P \xrightarrow{\bar{p}(q:t)} P'}{P|Q \xrightarrow{\bar{p}(q:t)} P'|Q}$$

avec comme condition supplémentaire : $q \notin \mathrm{P}(Q)$. Le rśultat attendu est $\Gamma \oplus (q \mapsto t) \vdash P' \mid Q$.

$\Gamma \oplus (q \mapsto t) \vdash P'$

- est donné directement par l'hypothèse de récurrence.

- Le résultat $\Gamma \oplus (q \mapsto t) \vdash Q$ découle du fait que $q \notin \mathrm{P}(Q)$ ainsi que de l'hypothèse $\Gamma \vdash Q$: on modifier dans un environnement de typage le type d'un nom qui n'est pas utilisé dans le processus typé (lemme 3.5.1).

3.9 Transitions internes

On prouve la propriété suivante :

$$P \xrightarrow{\tau} P' \Longrightarrow \Gamma \vdash P \Longrightarrow \Gamma \vdash P'$$

3.9.1 Cas de la règle COM

Considérons la règle COMl. La dérivation de la transition se termine par:

$$\frac{P \xrightarrow{pq} P' \qquad Q \xrightarrow{\bar{p}q} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

On a donc $\Gamma \vdash P \mid Q$, on veut obtenir $\Gamma \vdash P' \mid Q'$.

Pour prouver $\Gamma \vdash P'$ à partir de $\Gamma \vdash P$, on utilise la propriété de préservation des types pour les transitions INPUT. Pour remplir la condition annexe $\Gamma(q) \leq [\Gamma(p)]$ de cette propriété, on utilise la propriété d'émission d'un type compatible par la transition $Q \xrightarrow{\bar{p}q} Q'$.

Le fait que l'on puisse déduire $\Gamma \vdash Q'$ de $\Gamma \vdash Q$ découle simplement de la propriété de préservation des types pour la transition $Q \xrightarrow{\bar{p}q} Q'$.

3.9.2 Cas de la règle CLOSE

On prend la règle CLOSEl par exemple. La dérivation de la transition se termine par:

$$\frac{P \xrightarrow{pq} P' \qquad\qquad Q \xrightarrow{\bar{p}(r:t)} Q'}{P|Q \xrightarrow{\tau} (\nu x : t)(P'\{x/q\}|Q'\{x/r\})}$$

on sait en outre que $q \notin \mathrm{P}(P)$, c'est à dire que P reçoit bien un nom inconnu pour lui, et que $x \notin \mathrm{V}(P') \cup \mathrm{V}(Q')$, c'est à dire qu'il n'y a pas de risque de capture de la nouvelle liaison créée par cette règle.

De $Q \xrightarrow{\bar{p}(r:t)} Q'$ et de $\Gamma \vdash Q$, on déduit $t \leq [\Gamma(p)]$. Donc $(\Gamma \oplus (q \mapsto t))(q) \leq [\Gamma(p)]$. D'autre part, q n'apparaissant pas dans P,

$$\Gamma \vdash P \Longrightarrow \Gamma \oplus (q \mapsto t) \vdash P$$

par la propriété de préservation des types sur les INPUT :

$$\Gamma \oplus (q \mapsto t) \vdash P'$$

or $P' = P'\{x/q\}\{q/x\}$ puisque x n'apparaît pas dans P'. Donc pour tout s libre dans $P'\{x/q\}$ (lemme 3.5.3),

$$\Gamma \oplus (s \mapsto t) \vdash P'\{x/q\}\{s/x\}$$

D'autre part, r n'apparaissant pas dans Q, on a $\Gamma \oplus (r \mapsto t) \vdash Q'$, par la propriété de préservation des types sur les émissions-extrusions donc

$$\forall s \notin \mathrm{P}(Q'\{x/r\}) \quad \Gamma \oplus (s \mapsto t) \vdash Q'\{x/r\}\{s/x\}$$

de la même manière que précédemment. On peut donc conclure:

$$\Gamma \vdash (\nu x : t)(P'\{x/q\} | Q'\{x/r\})$$

4 Conclusion et extensions possibles

Le codage adopté ici a permis de transcrire presque directement la sémantique usuelle dans COQ, et d'effectuer très simplement la preuve de la propriété de préservation des types. Il faut noter que certaines erreurs faites initialement de transcription de la sémantique ont pu être détectées en butant sur la preuve de la préservation des types, ou la preuve de certaines propriétés auxiliaires de la relation de typage, ce qui est un argument pour justifier le système de typage adopté, et en faveur de la mécanisation des preuves.

On a utilisé ici une version très simple du π-calcul. Le passage à la polyadicité permet de gagner en expressivité du système de type. En effet, si le π-calcul polyadique est codable en π-calcul monadique, l'information d'arité des valeurs transportées par les noms ne peut pas être exprimée dans le cadre d'un typage du π-calcul monadique (on perdrait de toute façon la récursivité mutuelle). Un prolongement naturel de ce travail consisterait donc en une extension au π-calcul polyadique.

A Le codage du π-calcul en COQ

```
Parameter PP : Set.
Parameter VV : Set.

Axiom PP_decidable : (p,q:PP){(p=q)}+{(~p=q)}.
Axiom VV_decidable : (x,y:VV){(x=y)}+{(~x=y)}.

Inductive name : Set :=
  pname : PP -> name |
  vname : VV -> name.

Inductive proc : Set :=
  nil : proc |
  inp : name -> VV -> proc -> proc |
```

```
  out : name -> name -> proc -> proc |
  par : proc -> proc -> proc |
  res : VV -> type -> proc -> proc |
  ban : proc -> proc |
  sum : proc -> proc -> proc |
  mat : name -> name -> proc -> proc.

Definition subs_var_name : name -> name -> VV -> name :=
  [n,m:name][x:VV]
  Cases n of
  (pname p) => n |
  (vname v) => (
    Case (VV_decidable x v) of
      [_:(x=v)]m
      [_:(~x=v)]n
    end
               )
  end.

Fixpoint subs_var_proc [P:proc] : name -> VV -> proc :=
  [n:name][x:VV]
  Cases P of
    nil => nil |
    (inp m v qq) => (inp (subs_var_name m n x) v
      Case (VV_decidable x v) of
       [_:(x=v)]qq
[_:(~x=v)](subs_var_proc qq n x)
      end) |
    (out m1 m2 qq) => (out (subs_var_name m1 n x)
                           (subs_var_name m2 n x)
   (subs_var_proc qq n x)) |
    (par qq rr) => (par (subs_var_proc qq n x)
                        (subs_var_proc rr n x)) |
    (res v t qq) => (res v t
      Case (VV_decidable x v) of
       [_:(x=v)]qq
[_:(~x=v)](subs_var_proc qq n x)
      end) |
    (ban qq) => (ban (subs_var_proc qq n x)) |
    (sum qq rr) => (sum (subs_var_proc qq n x)
                        (subs_var_proc rr n x)) |
    (mat m1 m2 qq) => (mat (subs_var_name m1 n x)
                           (subs_var_name m2 n x)
                           (subs_var_proc qq n x))
```

```
  end.


Definition subs_par_name : name -> name -> PP -> name :=
  [n,m:name][p:PP]
  Cases n of
  (pname q) => (
    Case (PP_decidable p q) of
      [_:(p=q)]m
      [_:(~p=q)]n
    end
              ) |
  (vname v) => n
  end.

Fixpoint subs_par_proc [P:proc] : name -> PP -> proc :=
  [n:name][p:PP]
  Cases P of
  nil => nil |
  (inp m v qq) => (inp (subs_par_name m n p) v (subs_par_proc qq n p)) |
  (out m1 m2 qq) => (out (subs_par_name m1 n p)
                         (subs_par_name m2 n p)
                         (subs_par_proc qq n p)) |
  (par qq rr) => (par (subs_par_proc qq n p)
                      (subs_par_proc rr n p)) |
  (res v t qq) => (res v t (subs_par_proc qq n p)) |
  (ban qq) => (ban (subs_par_proc qq n p)) |
  (sum qq rr) => (sum (subs_par_proc qq n p)
                      (subs_par_proc rr n p)) |
  (mat m1 m2 qq) => (mat (subs_par_name m1 n p)
                         (subs_par_name m2 n p)
                         (subs_par_proc qq n p))
  end.

Inductive freshname [p:PP] : name -> Prop :=
  freshp : (q:PP)~p=q -> (freshname p (pname q)) |
  freshv : (v:VV)(freshname p (vname v)).

Inductive fresh [p:PP] : proc -> Prop :=
  frnil : (fresh p nil) |
  frinp : (m:name)(v:VV)(Q:proc)(freshname p m)->(fresh p Q)->
         (fresh p (inp m v Q)) |
  frout : (m1,m2:name)(Q:proc)(freshname p m1)->
         (freshname p m2)->(fresh p Q)->(fresh p (out m1 m2 Q)) |
```

```
  frpar : (P,Q:proc)(fresh p P)->(fresh p Q)->(fresh p (par P Q)) |
  frres : (v:VV)(t:type)(Q:proc)(fresh p Q)->(fresh p (res v t Q)) |
  frban : (Q:proc)(fresh p Q)->(fresh p (ban Q)) |
  frsum : (P,Q:proc)(fresh p P)->(fresh p Q)->(fresh p (sum P Q)) |
  frmat : (m1,m2:name)(Q:proc)(freshname p m1)->(freshname p m2)->
          (fresh p Q)->(fresh p (mat m1 m2 Q)).

Inductive freshvarname [v:VV] : name -> Prop :=
  freshvp : (p:PP)(freshvarname v (pname p)) |
  freshvv : (x:VV)~v=x->(freshvarname v (vname x)).

Inductive freshvar [v:VV] : proc -> Prop :=
  fvnil : (freshvar v nil) |
  fvinp : (m:name)(x:VV)(Q:proc)(freshvarname v m)->
          ~v=x->(freshvar v Q)->(freshvar v (inp m x Q)) |
  fvout : (m1,m2:name)(Q:proc)(freshvarname v m1)->(freshvarname v m2)->
          (freshvar v Q)->(freshvar v (out m1 m2 Q)) |
  fvpar : (P,Q:proc)(freshvar v P)->(freshvar v Q)->(freshvar v (par P Q)) |
  fvres : (x:VV)(t:type)(Q:proc)~v=x->(freshvar v Q)->(freshvar v (res x t Q)) |
  fvban : (Q:proc)(freshvar v Q)->(freshvar v (ban Q)) |
  fvsum : (P,Q:proc)(freshvar v P)->(freshvar v Q)->(freshvar v (sum P Q)) |
  fvmat : (m1,m2:name)(Q:proc)(freshvarname v m1)->(freshvarname v m2)->
          (freshvar v Q)->(freshvar v (mat m1 m2 Q)).
```

B Les types

```
Inductive capa : Set :=
  Read : capa |
  Write : capa |
  Both : capa |
  Nil : capa.

(* axiomes verifies par la rel. de sous-typage *)
Inductive capst : capa -> capa -> Prop :=
  capst_refl : (c:capa)(capst c c) |
  capst_nil  : (c:capa)(capst c Nil) |
  capst_both : (c:capa)(capst Both c).

Parameter type : Set.

Parameter getcap : type -> capa.
```

```
Parameter getobj : type -> type.

Inductive typest : type -> type -> Prop :=
typest_refl : (t:type)(typest t t) |
typest_nil : (t,t':type)((getcap t')=Nil)->(typest t t') |
typest_read : (t,t':type)((getcap t')=Read)->
         (capst (getcap t) Read)->
                     (typest (getobj t) (getobj t'))->
      (typest t t') |
typest_write : (t,t':type)((getcap t')=Write)->
                      (capst (getcap t) Write)->
                      (typest (getobj t') (getobj t))->
                      (typest t t').


Axiom typest_trans : (t,t':type)(typest t t')->(t'':type)(typest t' t'')->(typest t t'').

Definition env : Type := PP -> type.

Inductive typing : env -> proc -> Prop :=
  tnil : (G:env)(typing G nil) |
  tinp : (G:env)(p:PP)(x:VV)(P:proc)
         (capst (getcap (G p)) Read)->
  ((q:PP)(fresh q P)->
           (typing (addenv G q (getobj (G p))) (subs_var_proc P (pname q) x)))->
          (typing G (inp (pname p) x P)) |
  tout : (G:env)(p,q:PP)(P:proc)
         (capst (getcap (G p)) Write)->
          (typest (G q) (getobj (G p)))->
  (typing G P)->
          (typing G (out (pname p) (pname q) P)) |
  tpar : (G:env)(P,Q:proc)(typing G P)->(typing G Q)->(typing G (par P Q)) |
  tres : (G:env)(x:VV)(t:type)(P:proc)
         ((q:PP)(fresh q P)->(typing (addenv G q t) (subs_var_proc P (pname q) x)))->
         (typing G (res x t P)) |
  tban : (G:env)(P:proc)(typing G P)->(typing G (ban P)) |
  tsum : (G:env)(P,Q:proc)(typing G P)->(typing G Q)->(typing G (sum P Q)) |
  tmat : (G:env)(p,q:PP)(P:proc)((getcap (G p))=Both)->((getcap (G q))=Both)->
        (typing G P)->
         (typing G (mat (pname p) (pname q) P)).
```

C Le codage de la sémantique

```
Inductive act : Set :=
  aout : PP -> PP -> act |
  ainp : PP -> PP -> act |
  about : PP -> PP -> type -> act |
  tau : act.

Inductive freshact [p:PP] : act -> Prop :=
  faout : (q1,q2:PP)~p=q1->~p=q2->(freshact p (aout q1 q2)) |
  fainp : (q1,q2:PP)~p=q1->~p=q2->(freshact p (ainp q1 q2)) |
  fabout : (q1,q2:PP)(t:type)~p=q1->~p=q2->(freshact p (about q1 q2 t)) |
  ftau : (freshact p tau).

Inductive sem : proc -> act -> proc -> Prop :=
  sinp : (p,q:PP)(x:VV)(Q:proc)(sem (inp (pname p) x Q) (ainp p q) (subs_var_proc Q (pname q) x)) |
  sout : (p,q:PP)(Q:proc)(sem (out (pname p) (pname q) Q) (aout p q) Q) |
  scoml : (P,P',Q,Q':proc)(p,q:PP)
           (sem P (ainp p q) P')->(sem Q (aout p q) Q')->
           (sem (par P Q) tau (par P' Q')) |
  scomr : (P,P',Q,Q':proc)(p,q:PP)
           (sem P (aout p q) P')->(sem Q (ainp p q) Q')->
           (sem (par P Q) tau (par P' Q')) |
  sopen : (P,P':proc)(p,q:PP)(x:VV)(t:type)
          (fresh q P)->~p=q->
          (sem (subs_var_proc P (pname q) x) (aout p q) P')->
          (sem (res x t P) (about p q t) P') |
  sclosel : (P,P',Q,Q':proc)(p,q,r:PP)(t:type)(x:VV)
            (fresh q P)->
            (freshvar x P')->(freshvar x Q')->
            (sem P (ainp p q) P')->(sem Q (about p r t) Q')->
            (sem (par P Q) tau (res x t (par (subs_par_proc P' (vname x) q)
                                             (subs_par_proc Q' (vname x) r)))) |
  scloser : (P,P',Q,Q':proc)(p,q,r:PP)(t:type)(x:VV)
            (fresh q P)->
            (freshvar x P')->(freshvar x Q')->
            (sem P (ainp p q) P')->(sem Q (about p r t) Q')->
    (sem (par Q P) tau (res x t (par (subs_par_proc Q' (vname x) r)
                                     (subs_par_proc P' (vname x) q)))) |
  sres : (P,P':proc)(mu:act)(x,y:VV)(t:type)
           ((q:PP)(sem (subs_var_proc P (pname q) x) mu (subs_var_proc P' (pname q) y)))->
           (sem (res x t P) mu (res y t P')) |
```

```
  sban : (P,P':proc)(mu:act)(sem (par (ban P) P) mu P')->
         (sem (ban P) mu P') |
  sparl : (P,P',Q:proc)(mu:act)((p,q:PP)(t:type)(mu=(about p q t))->(fresh q Q))->
          (sem P mu P')->(sem (par P Q) mu (par P' Q)) |
  sparr : (P,P',Q:proc)(mu:act)((p,q:PP)(t:type)(mu=(about p q t))->(fresh q Q))->
          (sem P mu P')->(sem (par Q P) mu (par Q P')) |
  ssuml : (P,P',Q:proc)(mu:act)(sem P mu P')->
          (sem (sum P Q) mu P') |
  ssumr : (P,P',Q:proc)(mu:act)(sem P mu P')->
  (sem (sum Q P) mu P') |
  smat : (P,P':proc)(p:PP)(mu:act)(sem P mu P')->
          (sem (mat (pname p) (pname p) P) mu P').
```

D structure des preuves

La plupart des preuves sont faites par induction soit sur les types concrets définis, soit sur les relations (α-conversion, typage ou transition).

Par exemple la preuve de la conservation du type dans le cas d'une transition silencieuse (transitions τ) :

```
Theorem tau_red : (P,P':proc)(sem P tau P')->
(G:env)(typing G P)->(typing G P').
Proof.
Intros P P' reduction.
Apply tau_ind with Pr:=[P,P':proc](G:env)(typing G P)->(typing G P').
(* *****************************************************************
Regle COM1
   ***************************************************************** *)
Intros P P' Q Q' p q redinp redout G typed.
Inversion_clear typed.
Cut (typest (G q) (getobj (G p))).
Intro subtype.
Apply tpar.
Apply inp_autored with P:=P0 p:=p q:=q.
Assumption.
Assumption.
Assumption.
Apply out_autored with P:=Q p:=p q:=q.
Assumption.
Assumption.
```

```
Apply out_emits_correct_type with P:=Q P':=Q'.
Assumption.
Assumption.
(* ********************************************************************
Regle COMr
   ******************************************************************** *)
Intros P P' Q Q' p q redinp redout G typed.
Inversion_clear typed.
Cut (typest (G q) (getobj (G p))).
Intro subtype.
Apply tpar.
Apply out_autored with P:=Q p:=p q:=q.
Assumption.
Assumption.
Apply inp_autored with P:=P0 p:=p q:=q.
Assumption.
Assumption.
Assumption.
Apply out_emits_correct_type with P:=Q P':=Q'.
Assumption.
Assumption.
(* ********************************************************************
      Regle CLOSEl
   ******************************************************************** *)
Intros P P' Q Q' p q r t x fresh_q fvp fvq redinp redout G typed.
Inversion_clear typed.
Apply tres.
Intros s fresh_s.
Inversion_clear fresh_s.
Simpl.
Apply tpar.
Apply type_with_other_subs with p:=q.
Apply fresh_masked_by_var.
Assumption.
Cut (subs_var_proc (subs_par_proc P'0 (vname x) q) (pname q) x)=P'0.
(Intro dummy; Rewrite dummy).
Apply inp_autored with p:=p q:=q P:=P0.
Assumption.
Apply addenv_unused_name.
Assumption.
Assumption.
Cut (addenv G q t q)=t.
(Intro bam; Rewrite bam).
Cut (addenv G q t p)=(G p).
```

```
(Intro bom; Rewrite bom).
Apply bout_emits_correct_type with P:=Q P':=Q' q:=r.
Assumption.
Assumption.
Apply gettype_not_added_name.
(Red; Intro).
Cut ~(fresh p P0).
Intro.
Rewrite H3 in fresh_q.
(Elim H4; Assumption).
Apply inp_on_known_name with q:=q P':=P'0.
Assumption.
Apply gettype_added_name.
Transitivity (subs_par_proc P'0 (pname q) q).
Apply subs_var_after_subs_par.
Assumption.
Apply inefficient_subs_par.
Apply type_with_other_subs with p:=r.
Apply fresh_masked_by_var.
Assumption.
Cut (subs_var_proc (subs_par_proc Q' (vname x) r) (pname r) x)=Q'.
(Intro foo; Rewrite foo).
Apply bout_autored with P:=Q p:=p.
Assumption.
Assumption.
Transitivity (subs_par_proc Q' (pname r) r).
Apply subs_var_after_subs_par.
Assumption.
Apply inefficient_subs_par.
(* *****************************************************************
Regle CLOSEr
   ***************************************************************** *)
Intros P P' Q Q' p q r t x fresh_q fvp fvq redinp redout G typed.
Inversion_clear typed.
Apply tres.
Intros s fresh_s.
Inversion_clear fresh_s.
Simpl.
Apply tpar.
Apply type_with_other_subs with p:=r.
Apply fresh_masked_by_var.
Assumption.
Cut (subs_var_proc (subs_par_proc Q' (vname x) r) (pname r) x)=Q'.
(Intro bam; Rewrite bam).
```

```
Apply bout_autored with P:=Q p:=p.
Assumption.
Assumption.
Transitivity (subs_par_proc Q' (pname r) r).
Apply subs_var_after_subs_par.
Assumption.
Apply inefficient_subs_par.
Apply type_with_other_subs with p:=q.
Apply fresh_masked_by_var.
Assumption.
Cut (subs_var_proc (subs_par_proc P'0 (vname x) q) (pname q) x)=P'0.
(Intro bam; Rewrite bam).
Apply inp_autored with p:=p q:=q P:=P0.
Assumption.
Apply addenv_unused_name.
Assumption.
Assumption.
Cut (addenv G q t q)=t.
(Intro boom; Rewrite boom).
Cut (addenv G q t p)=(G p).
(Intro bang; Rewrite bang).
Apply bout_emits_correct_type with q:=r P:=Q P':=Q'.
Assumption.
Assumption.
Apply gettype_not_added_name.
(Red; Intro).
Cut ~(fresh p P0).
Intro not_fresh.
Rewrite H3 in fresh_q.
(Elim not_fresh; Assumption).
Apply inp_on_known_name with P':=P'0 q:=q.
Assumption.
Apply gettype_added_name.
Transitivity (subs_par_proc P'0 (pname q) q).
Apply subs_var_after_subs_par.
Assumption.
Apply inefficient_subs_par.
(* ********************************************************************
       Regle RES
   ******************************************************************** *)
Intros x y t' P P' cond_sem cond_pr G typed.
Inversion_clear typed.
Apply tres.
Intros r fresh_r.
```

```
Cut (Ex [s:PP](fresh s P0)/\~r=s).
Intro exists; Elim exists.
Intros s s_props.
Elim s_props.
Intros fresh_s r_not_s.
Apply type_with_other_subs with p:=s.
Apply fresh_before_subs with q:=r x:=y.
Apply fresh_after_trans with P:=(subs_var_proc P0 (pname r) x) mu:=tau.
Apply cond_sem.
Apply fresh_after_subs.
(Red; Intro; Elim r_not_s; Symmetry; Assumption).
Assumption.
Apply ftau.
Assumption.
Apply cond_pr.
Apply H.
Assumption.
Apply fresh_and_different.
(* *********************************************************************
       Regle BANG
   ********************************************************************* *)
Intros P P' red cond G typed.
Inversion_clear typed.
Apply cond.
Apply tpar.
(Apply tban; Assumption).
Assumption.
(* *********************************************************************
       Regle PARl
   ********************************************************************* *)
Intros P P' Q red cond G typed.
Inversion_clear typed.
Apply tpar.
(Apply cond; Assumption).
Assumption.
(* *********************************************************************
       Regle PARr
   ********************************************************************* *)
Intros P P' Q red cond G typed.
Inversion_clear typed.
Apply tpar.
Assumption.
Apply cond.
Assumption.
```

```
(* *********************************************************************
       Regle SUMl
   ********************************************************************* *)
Intros P P' Q red cond G typed.
Inversion_clear typed.
(Apply cond; Assumption).
(* *********************************************************************
       Regle SUMr
   ********************************************************************* *)
Intros P P' Q red cond G typed.
Inversion_clear typed.
(Apply cond; Assumption).
(* *********************************************************************
       Regle MATCH
   ********************************************************************* *)
Intros P P' r red cond G typed.
Apply cond.
Inversion_clear typed.
Assumption.
Assumption.
Qed.
```

References

[1] Tom Melham Andrew Gordon. Five axioms of alpha-conversion. In J. Harrison J. von Wright, J. Grundy, editor, *TPHOL'96*, volume 1125, pages 173–190. Springer-Verlag, 196.

[2] JF. Filliâtre G.Huet P.Manoury C.Muñoz C.Murthy C.Parent C.Paulin-Mohring A. Saibi B.Werner C.Cornes, J.Courant. *The Coq Proof Assistant Reference Manual*. Projet Coq, Inria Rocquencourt and CNRS-ENS Lyon, 1997.

[3] Daniel Hirschkoff. A full formalisation of pi-calculus theory in the calculus of constructions, 1997.

[4] Robert Pollack James McKinna. Some lambda calculus and type theory formalized, 1997.

[5] Robin Milner. The polyadic pi-calculus : a tutorial. Technical Report ECS-LFCS-91-180, LFCS, 1991.

[6] Walker Milner, Parrow. A calculus of mobile processes. Technical Report ECS-LFCS-89-85/6, LFCS, 1989.

[7] Davide Sangiorgi. Typing and subtyping for mobile processes. *Math. Struct. in Comp. Science*, 11, 1995.

[8] Davide Sangiorgi. An interpretation of typed objects into typed pi-calculus. INRIA RR-3000, 1996.