La terminaison dans les programmes concurrents d'ordre supérieur

Simon Castellan

22 janvier 2014

Stage effectué sous la tutelle de Daniel Hirschkoff,
au LIP, ENS de Lyon.

Table des matières

Introduction

Je tiens tout d'abord à remercier Daniel pour sa disponibilité et sa bonne humeur constante. Merci aussi à Romain Demangeon pour ses nombreuses réponses à mes questions non moins nombreuses, et à Oliver Laurent pour ses cours de *soft linear logic* accéléré. Merci à toute l'équipe Plume pour son accueil chaleureux pour ce (trop) court tour de manège. Merci enfin à tous ceux qui ont pris un peu (ou beaucoup) de leur temps pour relire ce document.

Ce rapport présente la majeure partie des travaux effectués pendant ces six semaines sur le π-calcul d'ordre supérieur, et plus particulièrement la terminaison de ces calculs, et dans une moindre manière, s'intéresse à la complexité implicite dans le monde concurrent. Dans un premier temps, nous présentons le contexte dans lequel s'inscrit le problème (terminaison, programmes concurrents), dans un second temps, l'état de l'art de la terminaison en π-calcul d'ordre supérieur, et dans un dernier temps, un calcul qui étend les π-calculs d'ordre supérieur dont la terminaison est étudiée. En outre un petit résultat de complexité implicite est donné pour ce langage.

1 Cadre informel du problème

1.1 Programmes séquentiels et programmes concurrents

Les premiers calculateurs avaient pour but de mener de longs et fastidieux calculs de manière automatique, en exécutant une liste d'instructions (*algorithme*) conçue par des humains. Avec l'arrivée de l'ordinateur, et l'explosion de puissance de calcul associée, des algorithmes de plus en plus subtils ont été conçus, tirant partie des nouvelles facultés des machines. La situation s'est peu à peu inversée : la demande en calcul a pris le pas sur l'augmentation régulière de la puissance de calcul, jusqu'à la dépasser récemment, et de nouvelles solutions ont dû être trouvées. La collaboration de plusieurs machines à la résolution d'un même problème est donc une alternative répandue pour diminuer le temps de calcul. On peut citer Seti@home qui fait travailler en parallèle des milliers de machines à travers le monde pour analyser des données scientifiques.

Les programmes qui consistent simplement en une exécution linéaire d'instructions sont appelés *séquentiels* alors que les programmes mettant en scène plusieurs agents communiquant et évoluant en parallèle sont appelés *concurrents*. Écrire un programme séquentiel exempt de bogues est une tâche ardue, mais écrire un programme concurrent exempt de bogues l'est encore plus.

Deux architectures de programmes concurrents sont très courantes :

- ▷ les programmes avec une architecture client/serveur : un serveur est chargé de contrôler l'accès concurrent de plusieurs clients à une même ressource. Par exemple, quand vous effectuez une recherche google, vous accédez à leur base de données en même temps que des milliers d'autres clients.
- ▷ un programme composé de plusieurs *threads* (c'est-à-dire des sous-programmes légers) qui évoluent en parallèle en faisant des tâches différentes. Par exemple, quand vous copiez des fichiers d'un endroit à un autre, on peut imaginer qu'il y a deux *threads* s'exécutant en parallèle, l'un chargé de copier les fichiers, l'autre de mettre à jour la barre de progression et de répondre aux interactions de l'utilisateur.

1.2 Terminaison

Une propriété intéressante, que les programmes devraient toujours vérifier, est de toujours renvoyer un résultat. Pour ce faire, il faut que l'algorithme termine, autrement dit, exécute un nombre fini d'opérations, indépendemment de son entrée. Cependant, savoir si un algorithme termine est une propriété indécidable, c'est-à-dire qu'on ne peut pas automatiquement (par l'intermédiaire d'un programme) dire si un algorithme termine. Tout ce qu'on peut faire, ce sont des approximations de cette propriété, *i.e.* mettre au point un filtre qui permet de n'accepter que des programmes qui terminent. Ce faisant, on sera amené à considérer comme invalides des programmes qui pourtant terminent.

Cependant, même étant donné un programme, savoir s'il termine pour toute entrée est un problème délicat. Par exemple la conjecture de Syracuse, qui postule la terminaison de l'algorithme suivant : on part d'un entier $n \in \mathbb{N}$, si $n = 1$, on s'arrête, sinon, si celui-ci est pair, on répète le processus avc l'entier $n/2$, si celui-ci est impair on répète le processus avec $3n + 1$. Savoir si pour tout entier n, l'algorithme termine est un problème ouvert.

La terminaison dans un cadre concurrent prend un autre sens : les serveurs sont par exemple conçus pour ne pas terminer, en attendant des requêtes, ils ne terminent donc pas (c'est le but). Ce qu'on attend d'un serveur c'est qu'il soit toujours réactif, c'est-à-dire qu'une requête ne puisse pas le faire partir dans une boucle infinie. Dans un programme concurrent pur, cela signifie que pour n'importe quelle interaction fixée correcte, le programme engendrera une quantité d'interaction finie.

Un exemple de processus divergent classique, c'est le serveur qui, à chaque réception d'une requête, démarre une requête sur lui-même. Dès que ce serveur reçoit une requête, il part dans une boucle infinie en générant une infinité de requêtes. Plus généralement, avec des cycles on a des serveurs divergents : si, quand on contacte le serveur A, celui-ci émet une requête pour le serveur B, et réciproquement, alors dès qu'on contacte un des deux serveurs, cela explose.

1.3 Programme concurrent d'ordre supérieur

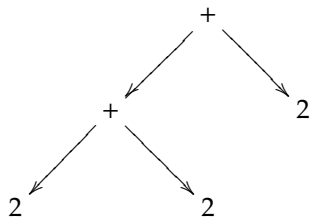
La plupart du temps, les programmes concurrents communiquent en s'échangeant des données comme des nombres entiers, des mots ou même des flots de caractères plus ou moins faciles à interpréter. Un programme concurrent d'ordre supérieur est un programme concurrent capable d'échanger des programmes en communiquant. C'est devenu très courant à présent sur le Web, car à peu près tous les sites web pour ajouter du dynamisme à leur interface, font télécharger aux clients du code JavaScript [1] chargé d'une part d'interagir avec l'utilisateur (*pop-up*, complétion, etc.) et d'autre part d'interagir avec le serveur (mise à jour automatique d'un contenu par exemple).

Cela peut aussi servir à remplacer à chaud du code s'exécutant sur un nœud du réseau, pour faire une mise à jour d'un service qui ne doit pas s'interrompre (*Hot code Swapping*).

1.4 Outils pour la terminaison

Afin d'identifier une classe de programmes comme terminant, il nous faut être capable de parler mathématiquement de deux choses : des programmes et de leur exécution. La première notion est celle de *langage formel* et la seconde de *réduction*.

Langage formel Un langage formel est une abstraction mathématique d'un langage concret qui permet de se débarasser d'aspects inintéressants pour le traitement des programmes (par exemple, des aspects syntaxiques). Prenons comme langage concret les expressions arithmétiques ne faisant intervenir que comme opérateurs l'addition et la multiplication. Alors (2 + 2) + 2 et ((2) + 2) + 2 sont des expressions syntaxiquement différentes, mais pourtant représentent le même calcul. Pour s'abstraire de cela, il faut identifier ce qui est important : les opérateurs ($+$, $\times$) et leurs arguments qui sont à nouveau des expressions. On peut donc représenter les deux expressions sous forme d'un arbre :



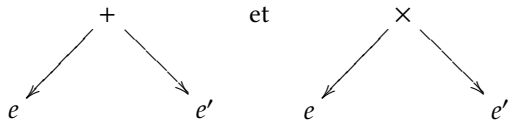
Cette représentation des expressions permet donc d'oublier le parenthésage, les espacements, et toutes les formalités syntaxiques. Avec cette idée en tête, on peut définir le langage formel des expressions arithmétiques ainsi :

$$\begin{array}{rcll} e, e' & ::= & \bar{n} & \text{entier naturel} \\ & | & e + e' & \text{somme} \\ & | & e \times e' & \text{produit} \end{array}$$

1. Ou flash, pour les impurs.

Ce qui signifie :

- ▷ tout entier n tout seul est une expression arithmétique notée $\bar{n}$ (en langage d'arbres, on parle de feuilles, elles sont tout en bas de l'arbre) ;
- ▷ si e et e' sont des expressions arithmétiques, alors



sont des expressions.

Réduction La notion de réduction permet de décrire comment un programme se réduit, c'est-à-dire, s'exécute. On essaye en fait de capturer une étape atomique de calcul, et l'on note $e \to e'$ pour dire que e se réduit en e' par un pas de calcul. Par exemple, on peut définir la réduction pour nos expressions arithmétiques ainsi :

$$\frac{}{\overline{n} + \overline{n'} \to \overline{n + n'}} \qquad \frac{}{\overline{n} \times \overline{n'} \to \overline{n \times n'}} \qquad \frac{g \to g'}{g + d \to g' + d} \qquad \frac{d \to d'}{g + d \to g + d'} \qquad \frac{g \to g'}{g \times d \to g' \times d}$$

$$\frac{d \to d'}{g \times d \to g \times d'}$$

La barre dénote une règle d'inférence et signifie "si ce qui est au-dessus de moi est vrai alors ce qui est en-dessous aussi", et en particulier s'il n'y a rien au-dessus alors ce qui est en-dessous est toujours vrai.

Les deux premières règles signifient que lorsqu'on a une expression de la forme $3 + 3$ ou 2×3, on peut réduire en une seule étape vers 6. Les autres règles signifient qu'on peut réduire "sous" les plus et les multiplier, pour chercher des opérations qu'on peut faire. C'est ce qu'on fait intuitivement quand on réduit $(2 + 2) \times 2$: on a quelque chose de la forme $e \times 2$, qu'on ne sait pas réduire, donc on descend dans e et on trouve $2 + 2$ qu'on sait réduire : donc $(2 + 2) \times 2 \to 4 \times 2$ par la troisième règle, puis $4 \times 2 \to 8$ par la deuxième règle.

Terminaison d'une réduction Muni de notre langage formel et de notre réduction, on voudrait à présent prouver qu'il n'y a pas d'expression arithmétique e admettant une séquence infinie de la forme $e \to e_1 \to e_2 \to \ldots$. Une façon standard de procéder est de trouver une quantité, que l'on appelle *mesure* pour chaque expression qui décroît strictement lors de la réduction. Cette quantité doit être à valeur dans un ensemble avec un ordre bien fondé, comme les entiers naturels, où il n'y a pas de séquences infinies $n_1 > n_2 > \ldots$. La décroissance stricte permet donc d'exclure la divergence des termes. Dans notre cas, on peut remarquer que le nombre d'opérateurs arithmétiques dans l'expression décroît de un lors d'une réduction, c'est-à-dire que si $e \to e'$ alors on a mangé un opérateur arithmétique. Cela permet donc de conclure.

2 π-calcul d'ordre supérieur

Dans cette section, on présente les π-calculs étudiés par la littérature : π-calcul du premier ordre avec une preuve de terminaison dont l'idée sera reprise dans la section 3, et π-calcul d'ordre supérieur, point de départ de mon stage.

2.1 π-calcul

Le π-calcul est un langage formel adapté à la description des interactions entre agents qui communiquent sur des *canaux*. Les termes du π-calcul (appelés processus) sont décrits par la grammaire suivante :

P, Q	$::= P \parallel Q$	composition parallèle
	$a(x).P$	réception sur un canal a
	$!\,a(x).P$	réception répliquée sur un canal a
	$\overline{a}\langle x\rangle$	envoi de x sur a
	$(\nu a)P$	restriction
	0	zéro

Il y a communication lorsqu'un processus qui écrit sur a ($\overline{a}\langle y\rangle$), rencontre un processus qui écoute sur a ($a(x).P$). Dans ce cas-là, ils se synchronisent et le processus $P[y/x]$ (P dans lequel y substitue x) est exécuté. Il y a en fait deux types de réceptions : répliquée ou non. La différence est que dans le cas répliqué, le processus qui écoute demeure après réception d'une donnée. C'est donc une construction qui permet de représenter des serveurs. Formellement on a les deux réductions :

- ▷ communication non répliquée : $a(x).P \parallel \overline{a}\langle y\rangle \longrightarrow P[y/x]$;
- ▷ communication répliquée : $!\,a(x).P \parallel \overline{a}\langle y\rangle \longrightarrow !\,a(x).P \parallel P[y/x]$.

On ajoute à ces deux-là un certain nombre de règles de passage au contexte pour réduire à l'intérieur des compositions parallèles et sous les restrictions. Les restrictions servent à créer un nouveau nom frais qui ne sera accessible que dans P. Par exemple, si on a $a(x).P \parallel (\nu a)\overline{a}\langle y\rangle$, il n'y aura pas communication, car l'utilisation du nom a est *restreinte* dans le sous-terme de droite. En pratique, on considérera que ce genre de situations n'arrivera pas et on supposera que deux variables identiques ne sont pas liées à des endroits différents et qu'aucune variable n'apparaît à la fois libre et liée dans un terme.

Les données échangées en π-calcul varient en fonction du dialecte étudié mais on considèrera ici qu'on pourra échanger uniquement d'autres canaux. Ainsi on peut écrire le terme $!\,a(x).\overline{x}\langle a\rangle$ qui est un processus qui attend sur a une adresse et qui envoie sa propre adresse à cette addresse. [2]

Divergence en π-calcul Pour faire des divergences, on s'aperçoît rapidement qu'il faut utiliser des communications répliquées, car lors de communications non répliquées, on diminue la taille du terme. On peut encoder notre exemple de la section 1.2 ainsi en π-calcul : $!\,a(x).\overline{a}\langle x\rangle \parallel \overline{a}\langle y\rangle$. Ce processus se réduit sur lui-même en une étape. De la même manière, on peut écrire un cycle de serveurs qui se passent les requêtes.

Typage en π-calcul Le typage simple en π-calcul est un peu particulier : au lieu d'attribuer des types aux programmes ("ce programme attend un entier et renvoie deux entiers"), les processus n'ont pas de types et on attribue un type à chaque canal qui représente les données transportées sur le canal. Par exemple, $\sharp\mathrm{int}$ dénote le type d'un canal qui transporte des entiers. Le typage simple permet de rejeter des termes malformés du type $\overline{a}\langle 42\rangle \parallel \overline{a}\langle(42, 42)\rangle$ qui pourrait créer une erreur à l'exécution, mais n'assure en rien la terminaison car l'exemple du paragraphe précédent est accepté.

Idée de terminaison en π-calcul On esquisse ici un système de typage qui permet d'assurer la terminaison des processus présenté dans [DS06] et [Dem10]. L'idée est d'imposer une stratification des canaux :

2. C'est la fonctionnalité "Recommander ce service à un ami" des réseaux sociaux.

$$
\begin{array}{lll}
V, W & ::= & () \quad \text{unit} \\
& | & x \quad \text{variable} \\
& | & \lambda x.\, P \quad \text{processus paramétré}
\end{array}
\qquad
\begin{array}{lll}
P, Q & ::= & 0 \quad \text{zéro} \\
& | & P \parallel Q \quad \text{composition parallèle} \\
& | & \overline{a}\langle V \rangle \quad \text{envoi d'une valeur} \\
& | & a(X).P \quad \text{réception d'une valeur} \\
& | & V\ W \quad \text{application} \\
& | & (\nu a)P \quad \text{restriction}
\end{array}
$$

FIGURE 1 – Syntaxe du π-calcul d'ordre supérieur

chaque canal aura un niveau ($\in \mathbb{N}$). Chaque processus aura un poids qui représentera le niveau du plus haut canal sur lequel le processus envoie des données. La vérification se fait alors au niveau des réceptions répliquées : on vérifie que le corps a un poids strictement inférieur au niveau du canal sur lequel on écoute. Ainsi, quand on fait une communication répliquée, on passe de $!\, a(x).P \parallel \overline{a}\langle y \rangle$ à $!\, a(x).P \parallel P[y/x]$. Et par hypothèse de typage, $P[y/x]$ est plus petit que $\overline{a}\langle y \rangle$, donc on a bien une décroissance. Les règles sont données dans la figure 6, elles seront exploitées dans la section 3.4.

2.2 Présentation du π-calcul d'ordre supérieur

On expose dans cette section le π-calcul d'ordre supérieur traité dans [DHS10] et [LMS10].

En π-calcul d'ordre supérieur, on transmet des valeurs qui sont des processus paramétrés. Formellement, la syntaxe est donnée par la figure 1. On a une très forte restriction sur les fonctions, elles doivent renvoyer un processus. Les types fonctionnels sont donc de la forme $T \to \diamond$, où $\diamond$ désigne le type des processus.

La réduction dans ce calcul se base sur deux règles :

- ▷ la β-réduction : $(\lambda x.\, P)V \to P[V/x]$;
- ▷ la communication : $a(x).P \parallel \overline{a}\langle V \rangle \to P[V/x]$.

Divergences Par rapport au π-calcul classique, on perd la capacité de réception répliquée. Cependant, on peut la simuler, en utilisant du *spawning* : $!\, a(x).P$ peut se représenter par $(\nu b)(S \parallel \overline{b}\langle \lambda x.S \rangle)$ avec $S \stackrel{\text{def}}{=} b(k).\left(k() \parallel a(x).(P \parallel \overline{b}\langle k \rangle)\right)$ L'idée est de se passer une continuation qui représente une écoute, et ensuite de se la renvoyer, pour la recevoir à nouveau (etc). En effet, k désignera toujours S (cette forme de définition permet de faire une sorte de point fixe l'opérateur $R \mapsto R \parallel a(x).P$). En utilisant cette idée, on peut trouver un processus divergent : si on pose $\delta \stackrel{\text{def}}{=} (a(X).X\ () \parallel \overline{a}\langle X \rangle)$, alors $\delta \parallel \overline{a}\langle \lambda x.\, \delta \rangle$ se réduit en lui-même en deux étapes.

Pour assurer la terminaison de ces calculs, il y a deux familles de systèmes de type :

- ▷ les systèmes qui stratifient les canaux comme celui esquissé à la section 2.1 ;
- ▷ les systèmes qui contrôlent la duplication des variables avec des techniques inspirées de la logique linéaire.

Un exemple de système appartenant à la première famille est détaillée dans la section suivante. Pour la seconde famille, ils sont dûs notamment à [LMS10].

$$\textsc{Lambda-Abstr}\ \frac{\Gamma, x : \tau \vdash P : n}{\Gamma \vdash \lambda x.\, P : \tau \to^{1+n} \diamond} \qquad \textsc{Lambda-App}\ \frac{\Gamma \vdash V : \tau \to^{n} \diamond \qquad \Gamma \vdash W : \tau}{\Gamma \vdash V\ W : n}$$

$$\textsc{Chan-Send}\ \frac{\Gamma \vdash V : \tau \qquad \Gamma(a) = \mathrm{Ch}_k(\tau) \qquad \mathrm{lvl}(\tau) < k}{\Gamma \vdash \overline{a}\langle V \rangle : k}$$

FIGURE 2 – Principales règles de typage pour le système « analyse d'envois »

2.3 Terminaison en analysant les envois

Un premier système pour la terminaison de ce calcul est proposé dans [DHS10]. L'idée est un peu la même que celle esquissée dans la section 2.1, mais en analysant les envois : pour envoyer une valeur V sur a, il faut que son "niveau" soit inférieur au poids de a. Comment définit-on le niveau de V ? On voudrait que toutes les valeurs de même type aient le même poids (pour des résultats de *Subject reduction*), on va donc parler des effets au niveau des types. Pour cela, on adopte une grammaire simple pour les types : $T ::= \ \mathrm{unit} \mid \tau \to^{n} \diamond$. Cela signifie qu'une valeur représente soit () soit $\lambda x.\, P$ avec P de poids $n - 1$.[3] Le niveau de T est alors 0 pour unit et n pour $\tau \to^{n} \diamond$. Quelques règles de typage importantes sont données en figure 2. Cela définit par extension deux jugements $\Gamma \vdash P : n$ et $\Gamma \vdash V : \tau$. $\mathrm{Ch}_k(\tau)$ est le type des canaux transportant des valeurs de type τ et de niveau k.

Pour établir la terminaison, on associe à chaque processus bien typé un multi-ensemble $m(P)$ constitué :

- ▷ des niveaux des envois qui ne sont pas dans un message ;
- ▷ des niveaux des abstractions lorsqu'on a un redex de la forme $(\lambda x.\, P)W$.

Cela permet d'établir la terminaison facilement : quand on fait une β-réduction, on mange un $n + 1$ (si le poids du corps de la fonction est n), et on ne génère que des termes de niveau au plus n. Idem quand on fait une communication. Dans la suite de l'exposé, il n'y aura plus besoin de brider autant l'usage des abstractions.

3 Contribution principale du stage : λπ

λπ est une tentative introduite lors du stage de monter plus haut dans l'ordre supérieur (par rapport à [DHS10]) en autorisant toutes les fonctions, et pas seulement celles dont le domaine d'arrivée sont les processus. Plus précisément, le but est de contenir tout le λ-calcul simplement typé, et tout le π-calcul d'ordre supérieur. De plus, on verra à la section 3.4 qu'on contient également tout le π-calcul.

3.1 Syntaxe et sémantique opérationelle.

La grammaire des valeurs et des types est donnée par la figure 3 et la sémantique opérationnelle par la figure 4. On supposera que tous les termes n'ont aucune variables libres et variables liées de même nom et qu'une même variable n'est pas liée à deux endroits différents. On note $t[u/x]$ la substitution de x par u dans t en évitant les captures.

$\equiv$ désigne la congruence structurelle entre processus définie de manière habituelle (en étendant la définition de [SW01]) sur les termes de λπ. Notons qu'aucune stratégie n'est privilégiée (appel par nom,

3. Le -1 est là pour assurer une preuve de terminaison facile.

t, u	$::= x$	variable	$\tau, \sigma ::=$	$\sigma \to \tau$	type fonctionnel
	$\mid \lambda x.\, t$	λ-abstraction		$\mid e$	processus de poids e
	$\mid t\ u$	application			
	$\mid a(x).\, t$	réception sur a			
	$\mid \overline{a}\langle t \rangle$	envoi sur a			
	$\mid (\nu a)t$	restriction			
	$\mid t \parallel u$	composition parallèle			
	$\mid 0$	zéro			

FIGURE 3 – Syntaxe de $\lambda\pi$

$$\beta\text{-RED}\ \frac{}{(\lambda x.\, t)\ u \to t[u/x]} \qquad \text{COMMUNICATION}\ \frac{}{a(x).t \parallel \overline{a}\langle u \rangle \to t[u/x]} \qquad \frac{t \to t'}{\overline{a}\langle t \rangle \to \overline{a}\langle t' \rangle} \qquad \frac{t \to t'}{t\ u \to t'\ u}$$

$$\frac{u \to u'}{t\ u \to t\ u'} \qquad \frac{t \to t'}{t \parallel u \to t' \parallel u} \qquad \frac{t \to t'}{(\nu a)t \to (\nu a)t'} \qquad \frac{P \equiv P' \qquad P' \to Q \qquad Q \equiv Q'}{P \to Q'}$$

FIGURE 4 – Sémantique opérationnelle de $\lambda\pi$

par valeur, ...), et qu'on réduit à l'intérieur des messages, cependant on ne réduit pas sous les abstractions et les réceptions.

Quelques exemples de programmes :

- ▷ on a les termes divergents classiques du λ-calcul et du π-calcul : $(\lambda x.x\ x)(\lambda x.x\ x)$ ainsi que $S \parallel \overline{a}\langle S \rangle$ avec $S \stackrel{\text{def}}{=} \lambda z.(a(k).k\ ()\|\overline{a}\langle k \rangle)$
- ▷ on réduit sous les envois, ce qui permet de faire de la passivation (étudié par exemple dans [DHS10]), c'est-à-dire qu'on voit le terme $\overline{a}\langle P \rangle$ (où P représente un processus) non pas comme l'envoi d'un processus mais comme un processus s'exécutant à la localité a. Dans ce contexte, $a(X).Q$ signifie « récupérer le processus s'exécutant à la localité a, l'y ôter et exécuter Q ». Avec ce mécanisme, on peut représenter la technique de *Hot code swapping* qui consiste à remplacer du code s'exécutant sur un nœud à chaud. Avec ce mécanisme, il suffit de faire $a(X).\overline{a}\langle Q \rangle$ pour remplacer par Q le processus s'exécutant sur le nœud a. Cependant, deux processus s'exécutant sur deux localités ne peuvent pas communiquer directement : dans $\overline{a}\langle P \rangle \parallel \overline{b}\langle Q \rangle$, P et Q ne peuvent pas communiquer. On a une solution partielle à ce problème. Notons $a, b, c, \ldots$ pour des localités et $x, y, \ldots$ pour des canaux destinés à la communication. Si on veut envoyer un message sur le canal x à une localité a, on peut écrire $a(P).\overline{a}\langle P \| \overline{x}\langle t \rangle \rangle$ ce qui a pour effet de remplacer le processus s'exécutant sur a par lui-même mis en parallèle avec l'envoi du message désiré. Cependant faire l'inverse (récupérer un message depuis une localité) semble être plus délicat avec la réduction considérée ici.

$$\textsc{Axiome}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Lambda-Abstr}\ \frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \tau \to \sigma} \qquad \textsc{Lambda-App}\ \frac{\Gamma \vdash t : \tau \to \sigma \qquad \Gamma \vdash u : \tau}{\Gamma \vdash t\ u : \sigma}$$

$$\textsc{Zero}\ \frac{}{\Gamma \vdash 0 : \varnothing} \qquad \textsc{Par-Comp}\ \frac{\Gamma \vdash t : e \qquad \Gamma \vdash u : e'}{\Gamma \vdash t \parallel u : e \uplus e'} \qquad \textsc{Chan-Send}\ \frac{\Gamma(a) = \mathrm{Ch}_k(\tau) \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \overline{a}\langle t \rangle : \{k\}}$$

$$\textsc{Chan-Recv}\ \frac{\Gamma(a) = \mathrm{Ch}_k(\tau) \qquad \Gamma, x : \tau \vdash t : e' \qquad e' < \{k\}}{\Gamma \vdash a(x).t : \varnothing} \qquad \textsc{Chan-Res}\ \frac{\Gamma, a : \mathrm{Ch}_k(\tau) \vdash t : \tau}{\Gamma \vdash (\nu a)t : \tau}$$

$$\textsc{Subsumption}\ \frac{\Gamma \vdash t : \sigma \qquad \sigma \preceq \sigma'}{\Gamma \vdash t : \sigma'}$$

FIGURE 5 – Règles de typage de $\lambda\pi$

3.2 Typage

On définit un jugement de typage $\Gamma \vdash t : \tau$ dérivé du λ-calcul simplement typé et du typage de la section 2.1. Notons que les processus ont pour type un multi-ensemble fini d'entiers décrivant les effets de ce processus : les effets sont les niveaux des canaux sur lesquels le processus fait un envoi à la racine. Les règles sont données à la figure 5, où $Ch_k(\tau)$ désigne le types des canaux de niveau k et transportant des valeurs de type τ.

Aux règles naturelles, il faut ajouter une règle de sous-typage pour deux raisons :

- ▷ lors d'une réduction, le poids d'un processus, donc son type peut diminuer et entraîner des complications. Si on a $f\ t \to f\ t'$ et que le poids de t' est inférieur à celui de t, alors f doit changer son type pour accepter t', ce qui n'est pas toujours possible ;
- ▷ si on veut utiliser une fonction (ou un canal) avec des processus de poids différents : $f\ 0 \parallel f\ (\overline{a}\langle x \rangle)$ par exemple, f doit être de type $\varnothing \to \tau$ et $\{\mathrm{lvl}(a)\} \to \tau$ avec ce système sur les processus.

Pour introduire une règle de sous-typage, il nous faut déjà introduire l'ordre sur les types. On le définit ainsi :

- ▷ $e \preceq e'$ si et seulement si $e \leq e'$ pour l'ordre multi-ensemble ;
- ▷ $\sigma \to \tau \preceq \sigma' \to \tau'$ si et seulement si $\sigma' \preceq \sigma$ et $\tau \preceq \tau'$.

***Lemme* 1** (Compatibilité par substitution). — *Soient t, u deux termes tels que $\Gamma, x : \tau \vdash t : \sigma$ et $\Gamma \vdash u : \tau$. Alors $\Gamma \vdash t[u/x] : \sigma$.*

Démonstration. On remarque que dans une dérivation, lorsqu'on descend dans les sous-termes, le contexte ne peut que grossir. Les feuilles de la dérivation de typage de t concernant la variable x sont donc de la forme $\Gamma, \Delta \vdash x : \tau$. Par hypothèse sur la forme des termes (pas de variable étant liée deux fois), on a $\Gamma, \Delta \vdash u : \tau$, et cette feuille peut donc être remplacée par la dérivation $\Gamma, \Delta \vdash u : \tau$ et en faisant cela sur chaque feuille qui parle de x, on obtient une dérivation de $\Gamma \vdash t[u/x] : \sigma$. □

***Lemme* 2** (Subject reduction). — *Si $\Gamma \vdash t : \tau$ et $t \to t'$ alors $\Gamma \vdash t' : \tau$.*

Démonstration. Par induction sur $t \to t'$.

- ▷ cas de la β-réduction : ne pose pas de problèmes avec le lemme 1

- ▷ si on a $a(x).P \parallel \overline{a}\langle v\rangle \longrightarrow P[v/x]$. Si k est le niveau de a alors on a $\Gamma, x : \tau \vdash P : e$ avec $e < \{k\}$ et τ le type des valeurs transportées par a. Alors avec le lemme 1 on a $\Gamma \vdash P[v/x] : e$ puis par sous-typage $\Gamma \vdash P[v/x] : \{k\}$
- ▷ pour les autres règles, il suffit d'appliquer l'hypothèse d'induction.

□

3.3 Une preuve de terminaison

Comme ce langage inclut le λ-calcul simplement typé, j'ai pensé qu'une preuve de réduction par candidats [GLT89] serait adaptée. Pour cela, on définit par induction sur σ, les termes réductibles de type σ :

- ▷ si P est un processus de poids e alors P est réductible si et seulement si il est fortement normalisable ;
- ▷ si t est un terme de type $\tau \longrightarrow \tau'$, alors il est normalisable si et seulement si pour tout terme u réductible de type τ, $t\ u$ est réductible de type τ'.

Par abus de langage, on supposera que tous les termes que l'on considère sont typables, sans forcément le mentionner. Il est facile de vérifier que les termes réductibles vérifient bien les trois critères de réductibilité : malgré l'ajout de règles de réduction, la preuve reste la même que dans [GLT89]. On notera qu'ici les seuls termes neutres sont les abstractions.

1. tout terme réductible est fortement normalisable ;
2. l'ensemble des termes réductibles de type τ est stable par réduction.
3. si t n'est pas une abstraction et que tous les termes t' tels que $t \longrightarrow t'$ sont réductibles, alors t est réductible.

Ensuite, on a encore un lemme qui se prouve de la même façon que dans [GLT89] :

***Lemme* 3.** — *Si t est tel que $\Gamma, x : \tau \vdash t : \sigma$ et si pour tout terme u réductible de type τ, $t[u/x]$ est réductible de type σ alors $\lambda x.\, t$ est réductible de type $\tau \longrightarrow \sigma$.*

À présent, on a besoin de voir ce qu'il se passe du côté des processus. On s'intéresse à un processus de la forme $P \parallel Q$. Il a trois évolutions possibles :

- ▷ soit $P \parallel Q$ évolue vers $P' \parallel Q$ (ou $P \parallel Q'$) sans communication entre P et Q en faisant juste une réduction $P \longrightarrow P'$ (ou $Q \longrightarrow Q'$) ;
- ▷ soit $P \parallel Q$ évolue vers R et les deux processus P et Q communiquent (il ne peuvent pas faire de β-réduction) et donc on a consommé un envoi. Un peu plus formellement, si on a par exemple, $P \parallel Q = a(x).P' \parallel \overline{a}\langle t\rangle \parallel S \longrightarrow P[t/x] \parallel S = R$, alors, on avait $\Gamma \vdash a(x).P \parallel \overline{a}\langle t\rangle \parallel S : \{k\} \uplus e$ et $\Gamma \vdash P[t/x] \parallel S : e' \uplus e$ avec $e' < \{k\}$ par la règle Chan-Recv. Donc, lors d'une communication à la racine du terme, *le poids du processus décroît* : c'est-à-dire qu'on peut trouver une dérivation qui attribue au terme réduit un poids plus faible.

Avec cette remarque, on est prêt à démontrer le lemme central. Cependant, il reste encore le problème de la règle de sous-typage. Le lemme suivant permet d'ignorer les règles de sous-typage dans une dérivation :

***Lemme* 4.** — *Si σ et τ sont des types tels que $\sigma \preceq \tau$, alors tous les éléments réductibles de types σ sont réductibles de type τ.*

Démonstration. Par induction sur σ.

- ▷ Si σ est un type de processus, alors c'est clair.
- ▷ Si $\sigma = \sigma_1 \longrightarrow \sigma_2$ et $\tau = \tau_1 \longrightarrow \tau_2$ avec $\tau_1 \preceq \sigma_1$ et $\sigma_2 \preceq \tau_2$. Soit u un terme réductible de type σ, et t un terme réductible de type τ_1. Il faut montrer que $u\ t$ est un terme réductible de type τ_2. C'est vrai, car

t est réductible de type σ_1 par induction, donc comme u est réductible de type σ, $u\ t$ est réductible de type σ_2 puis réductible de type τ_2, encore par réduction.

□

***Lemme* 5.** — *Soit un terme t tel que $(x_1 : \sigma_1, \dots, x_n : \sigma_n) \vdash t : \tau$. Pour tous $x_1, \dots, x_n, u_1, \dots, u_n$ tels que u_i soit réductible de type σ_i alors $t[u_i/x_i]_i$ (ce qui sera noté $t[\mathbf{u}/\mathbf{x}]$) est réductible.*

Démonstration. On procède par induction sur l'ordre bien fondé sur les dérivations de typage suivant : « $\pi < \pi'$ si et seulement si on a $\pi : \Gamma \vdash t : \_$ et $\pi' : \Gamma' \vdash t' : \tau'$ et que ou bien π est une sous-dérivation de π' ou bien τ, τ' sont des multi-ensembles, $\tau < \tau'$, et $\Gamma = \Gamma'$ ». Cela permet de faire une double induction sur la structure des termes et à la fois sur leur poids. On suppose à présent que $\pi : \Gamma \vdash v : \tau$, et on raisonne par cas sur la structure de v.

Enfin, grâce au lemme précédent on suppose que la dernière règle appliquée n'est pas subsubmption, car sinon on utilise le lemme précédent pour conclure directement par induction.

- ▷ Si $v = t\ t'$, on applique l'hypothèse d'induction sur les dérivations de typage de t et $t' : t[\mathbf{u}/\mathbf{x}]$ est réductible d'un type flèche, et $t'[\mathbf{u}/\mathbf{x}]$ est réductible du type de son argument, donc par définition, $(t\ t')[\mathbf{u}/\mathbf{x}]$ est réductible ;
- ▷ si $v = (\nu a)t$ ou $v = 0$ ou $v = \overline{a}\langle t\rangle$, $v = a(x).t$, on a dans tous les cas très facilement par induction que $v[\mathbf{u}/\mathbf{x}]$ est fortement normalisable ;,
- ▷ si $v = x_i$, alors on conclut car u_i est réductible ;
- ▷ si $v = \lambda y.t$ avec $v : \sigma \to \sigma'$ alors par induction, pour tout u' réductible de type σ, $t[\mathbf{u}/\mathbf{x}, u'/y]$ est réductible de type σ' et donc par le lemme 3, v est réductible ;
- ▷ si $v = P \parallel Q$, c'est le cas délicat de la démonstration. Il faut montrer que $P[\mathbf{u}/\mathbf{x}] \parallel Q[\mathbf{u}/\mathbf{x}]$ est fortement normalisable. On sait par hypothèse que $P[\mathbf{u}/\mathbf{x}]$ et $Q[\mathbf{u}/\mathbf{x}]$ le sont. Supposons avoir une suite infinie de réductions à partir de $v[\mathbf{u}/\mathbf{x}]$. Alors, il y a forcément une communication à partir d'un certain point entre $P[\mathbf{u}/\mathbf{x}]$ et $Q[\mathbf{u}/\mathbf{x}]$ car ces deux processus sont fortement normalisants. On a donc une situation de la forme $P[\mathbf{u}/\mathbf{x}] \parallel Q[\mathbf{u}/\mathbf{x}] \to^* a(x).P' \parallel \overline{a}\langle t\rangle \parallel R \to P'[t/x] \parallel R \to \dots$.
  On sait qu'on a une dérivation $\Gamma, \Delta \vdash P' : e < \{k\}$, où k est le niveau de a. Alors on a une dérivation $\pi' : \Gamma \vdash P'[t/x] : e$. Récapitulons, nous avons $\Gamma \vdash P[\mathbf{u}/\mathbf{x}] \parallel Q[\mathbf{u}/\mathbf{x}] : e_0$, puis $\Gamma \vdash a(x).P' \parallel \overline{a}\langle t\rangle \parallel R : e_0 = \{k\} \uplus e_R$. Donc, on a une dérivation $\Gamma \vdash P'[t/x] \parallel R : e \uplus e_R < e_0$, ce qui permet d'appliquer l'hypothèse d'induction à cette dérivation. Ainsi $P[t/x] \parallel R$ est fortement normalisable, ce qui contredit l'hypothèse initiale de divergence de $v[\mathbf{u}/\mathbf{t}]$.

□

***Théorème* 1.** — *Tout terme bien typé clos de $\lambda\pi$ est fortement normalisable.*

Remarquons que si on autorise la réduction sous les abstractions, la preuve fonctionne toujours.

3.4 Encodage du π-calcul

Lors du passage du π-calcul à λπ, on a perdu deux choses essentielles :

- ▷ la possibilité de faire des réceptions répliquées ;
- ▷ la possibilité de transmettre des canaux.

Le premier point est résolu par le *spawning* (décrit par exemple dans [LMS10]). Bien que λπ ne permette pas toutes les possibilités décrites par cet article, on peut tout de même utiliser la même astuce qu'en section 2.2. Le second point peut se résoudre avec une autre astuce : l'idée est d'encapsuler un canal a par deux fonctions qui représentent les deux capacités sur un canal : capacité de réception et d'émission.

$$\frac{\Gamma(x) = \tau \qquad \Gamma(a) = \sharp^k\tau}{\Gamma \vdash_D \overline{a}\langle x\rangle : k} \qquad \frac{\Gamma, x : \tau \vdash_D P : k \qquad \Gamma(a) = \sharp^k\tau}{\Gamma \vdash_D a(x).P : 0} \qquad \frac{\Gamma, x : \tau \vdash_D P : k' < k \qquad \Gamma(a) = \sharp^k\tau}{\Gamma \vdash_D !\, a(x).P : 0}$$

$$\frac{\Gamma \vdash_D P : k \qquad \Gamma \vdash_D P' : k'}{\Gamma \vdash_D P \parallel P' : \max(k, k')} \qquad \frac{\Gamma, a : \sharp^k\tau \vdash_D P : k}{\Gamma \vdash_D (\nu a)P : k} \qquad \frac{\Gamma(x) = \tau \qquad \Gamma(a) = \sharp^k\tau}{\Gamma \vdash_D \overline{a}\langle x\rangle : k}$$

FIGURE 6 – Règles de typage pour la terminaison en π

Ainsi, si on rajoute les couples à λπ, on peut représenter un canal a par le couple $(\lambda x.\overline{a}\langle x\rangle, \lambda f.\, a(x).\, f\ x)$, ce que l'on notera wrap(a).

On peut donc ainsi traduire les termes du π-calcul en λπ. Pour tout terme t du calcul mentionné en 2.1 dont les règles sont données en figure 6 [4], on associe un terme $[[t]]$ de λπ. Notre encodage distingue deux type de canaux :

- ▷ les canaux natifs qui seront traduits tels quels sur lesquels on peut directement faire des entrées/-sorties (notation : a, b) ;
- ▷ les canaux encapsulés, ce sont des canaux reçus qui sont donc sous la forme d'un couple, pour faire une opération dessus, il faut récupérer la bonne composante (notation : x, y).

Notre traduction prend donc en argument, en plus du processus P, une partition A, X des canaux libres de P dans l'idée que les canaux de A représentent les canaux natifs et ceux de X les canaux encapsulés.

- ▷ $[[a(x).P]]_{A,X} = (\text{snd } a)(\lambda x.\ [[P]]_{A,X\cup\{x\}})$ si $a \in X$;
- ▷ $[[a(x).P]]_{A,X} = a(x).\ [[P]]_{A,X\cup\{x\}})$ si $a \in A$;
- ▷ $[[\overline{a}\langle b\rangle]]_{A,X} = \overline{a}\langle\text{wrap b}\rangle$ si $a, b \in A$;
- ▷ $[[\overline{a}\langle y\rangle]]_{A,X} = \overline{a}\langle y\rangle$ si $(a, y) \in A \times X$;
- ▷ $[[\overline{x}\langle b\rangle]]_{A,X} = \text{fst } x\ (\text{wrap b})$ si $(x, b) \in X \times A$;
- ▷ $[[\overline{x}\langle y\rangle]]_{A,X} = \text{fst } x\ y$ si $x, y \in X$;
- ▷ $[[0]]_{A,X} = 0$;
- ▷ $[[P \parallel Q]]_{A,X} = [[P]]_{A,X} \parallel [[Q]]_{A,X}$;
- ▷ $[[(\nu a)P]]_{A,X} = [[P]]_{\{a\}\cup A,X}$;
- ▷ $[[!\, a(x).P]]_{A,X} = (\nu b)(S \parallel \overline{b}\langle\lambda x.S\rangle)$ où S est le processus $b(k).a(x).\left([[P]]_{A,\{x\}\cup X} \parallel \overline{b}\langle k\rangle \parallel k()\right)$ si $a \in A$, sinon il faut remplacer $a(x)$ par snd $a(\lambda x....)$ comme pour les réceptions non répliquées.

Pour valider notre encodage de π en λπ, on présente deux lemmes :

***Lemme* 6.** — *Simulation Soient P, Q deux processus clos du π-calcul de variables libres A. Si $P \to_\pi Q$ alors $[[P]]_{A,\varnothing} \to^+_{\lambda\pi} Q$*

La simulation ne marche que si on considère que les canaux libres sont encodés nativement (d'où le $X = \varnothing$ du lemme), car si l'on regarde $[[a(x).P \parallel \overline{a}\langle t\rangle]]_{\varnothing,\{a,t\}} = (\text{snd } a(\lambda x.[[P]])) \parallel \text{fst } a\ t$, ce dernier ne se réduit pas..

Démonstration. Rien de très difficile ici, on traite les deux cas de réduction primitifs.

- ▷ Si on a $a(x).P' \parallel \overline{a}\langle b\rangle \to P'[b/x]$, alors $[[P]]_{A,\varnothing} = a(x).[[P]]'_{A,\{x\}} \parallel \overline{a}\langle b\rangle$ ce qui se réduit dans $\lambda\pi$ vers $[[P]]'_{A,\{x\}}[b/x] = [[P'[b/x]]]_{A,\varnothing}$ ce que l'on voulait.
- ▷ Si on a $!\, a(x).P' \parallel \overline{a}\langle b\rangle \to !\, a(x).P' \parallel P'[b/x]$, alors :

4. Ce n'est pas exactement le système original de [DS06], il correspond mieux à notre système pour λπ

$$\textsc{Lambda-Abstr}\ \frac{\Gamma, x:\tau \vdash t:\sigma}{\Gamma \vdash \lambda x.t : \tau \to \sigma} \qquad \textsc{Lambda-App}\ \frac{\Gamma \vdash t:\tau \to \sigma \qquad \Gamma \vdash u:\tau}{\Gamma \vdash t\ u : \sigma} \qquad \textsc{Zero}\ \frac{}{\Gamma \vdash 0 : 0}$$

$$\textsc{Par-Comp}\ \frac{\Gamma \vdash t:k \qquad \Gamma \vdash u:k'}{\Gamma \vdash t \parallel u : \max(k,k')} \qquad \textsc{Chan-Send}\ \frac{\Gamma(a) = \mathrm{Ch}_k(\tau) \qquad \Gamma \vdash t:\tau}{\Gamma \vdash \overline{a}\langle t\rangle : k}$$

$$\textsc{Chan-Recv}\ \frac{\Gamma(a) = \mathrm{Ch}_k(\tau) \qquad \Gamma, x:\tau \vdash t:k' \qquad k' < k}{\Gamma \vdash a(x).P : 0} \qquad \textsc{Chan-Res}\ \frac{\Gamma, a:\mathrm{Ch}_k(\tau) \vdash t:\tau}{\Gamma \vdash (\nu a)t : \tau}$$

$$\textsc{Subsumption}\ \frac{\Gamma \vdash t:\sigma \qquad \sigma \le \sigma'}{\Gamma \vdash t : \sigma'}$$

Figure 7 – Règles de typage de $\lambda\pi$ — entiers

$$\begin{aligned}
[[P]]_{A,\varnothing} &= \nu c(S \parallel \overline{b}\langle \lambda x.S\rangle) \parallel \overline{a}\langle b\rangle \\
&\equiv (\nu c)(S \parallel \overline{S}\langle\,\rangle \parallel \overline{a}\langle b\rangle) \\
&\to (\nu c)(a(x).\left([[P]]'_{A,\{x\}} \parallel \overline{b}\langle S\rangle \parallel (\lambda x.S)\ ()\right) \parallel \overline{a}\langle b\rangle) \\
&\to (\nu c)([[P]]'_{A,\{x\}}[b/x] \parallel \overline{b}\langle S\rangle \parallel S) \\
&\equiv [[P'[b/x]]]_{A,\varnothing} \parallel [[!\,a(x).P']]_{A,\varnothing}
\end{aligned}$$

□

Le système pour $\lambda\pi$ est basé sur des multi-ensembles pour des processus alors que le système pour π utilise des entiers. Pour des raisons d'uniformité, on donne un jugement qui attribue des entiers aux processus pour $\lambda\pi$. Le système avec des multi-ensembles est nécessaire à l'établissement de la preuve de terminaison : on ne peut pas simplement s'en tirer avec une mesure qui décroît à chaque communication à cause de la subtilité de $\lambda\pi$. Le jugement avec des entiers est donné en figure 7. Le lemme suivant énonce une équivalence basique entre les deux systèmes de type.

***Lemme* 7.** — *Soit t un terme de $\lambda\pi$ et k un entier. Alors les deux conditions suivantes sont équivalentes :*

(i) il existe Γ tel que $\Gamma \vdash t : k$ dans le système à entiers ;

(ii) il existe Γ et e tel que $\Gamma \vdash t : e$ dans le système à multi-ensembles et $\max e = k$

On ne s'attarde pas trop sur ce résultat pour des raisons d'espace. Dans tout le reste de cette sous-section, on considère à présent uniquement le système à entiers pour $\lambda\pi$.

***Définition* 1.** — *Encodage des types On définit un encodage des types de π dans ceux de $\lambda\pi$ ainsi :*

- ▷ $[[unit]] = 0$;
- ▷ $[[\sharp^k \tau]] = ([[\tau]] \to k) \times (([[\tau]] \to (k-1)) \to 0)$.

Le $k-1$ est là pour s'assurer que, dans le corps d'une réception, on ne dépasse pas le niveau du canal.

***Théorème* 2.** — *Soit P un terme du π-calcul tel que $\Gamma \vdash_D P : k$ tel que tous les canaux de Γ sont de poids au moins un. Alors pour toute partition A, X du codomaine de Γ, on définit le contexte Γ^* par*

$$\dfrac{\dfrac{\Gamma^*, b : Ch_0(0), p : 0, x : [[T]] \vdash [[P]]_{A,\{x\}\cup B} \parallel p \parallel \overline{b}\langle p\rangle : k' < k}{\Gamma^*, b : Ch_0(0) \vdash b(p).a(x).([[P]]_{A,\{x\}\cup B}) \parallel p \parallel \overline{b}\langle p\rangle : 0} \qquad \dfrac{\Gamma^*, b : Ch_0(0) \vdash S : 0 \qquad \Gamma(b) = Ch_0(0)}{\Gamma^*, b : Ch_0(0) \vdash \overline{b}\langle S\rangle : 0}}{\dfrac{\Gamma^*, b : Ch_0(0) \vdash S \parallel \overline{b}\langle S\rangle : 0}{\Gamma^* \vdash (\nu b)(S \parallel \overline{b}\langle S\rangle) : 0}}$$

FIGURE 8 – Dérivation de typage pour l'écoute répliquée. Hypothèse $\Gamma \vdash_D !\,a(x).P : 0$

▷ $\Gamma^*(a) = Ch_k([[\tau]])$ *si* $a \in A$ *et* $\Gamma(a) = \sharp^k\tau$;
▷ $\Gamma^*(x) = [[\tau]]$ *si* $x \in X$ *et* $\Gamma(x) = \tau$.
On a alors $\Gamma^* \vdash P : k$ *dans le système de* $\lambda\pi$.

Démonstration. Au vu de l'encodage des termes et des types, cela se fait facilement pour tous les cas sauf celui de l'écoute répliquée. Il faut remarquer qu'on peut attribuer au canal utilisé par le serveur (le b de l'encodage) un poids nul, car on écrit sur b derrière une écoute, donc l'écriture ne compte pas. C'est pour ça qu'on a besoin que tous les canaux soient de poids au moins un — hypothèse non restrictive car il suffit de décaler les poids —, car si a est de poids zéro par contre cela ne type pas. Cette remarque étant faite, la dérivation de typage s'écrit et est disponible en figure 8 (seulement le cas où le canal sur lequel on écoute n'est pas encapsulé). □

3.5 Comparaison aux systèmes similaires

3.5.1 Comparaison avec $\lambda\pi_v$

Dans [YH00] est développé un langage très similaire à λπ. Au niveau syntaxique, $\lambda\pi_v$ permet de passer des canaux comme des valeurs de première classe (mais on vient de voir que cette capacité peut être émulée en λπ). Le point sur lequel tout diffère est le système de type. Le système de $\lambda\pi_v$ est plus complexe et vise à obtenir des informations sur l'interface des processus. On est donc en fait plutôt du côté de la vérification de protocoles entre processus que de la terminaison. Leur système laisse passer des termes divergents (comme celui de la section 2.2). Cependant, il y a un encodage trivial de λπ en $\lambda\pi_v$ qui consiste à assigner à tous les processus (peu importe leur poids) le type proc.

3.5.2 Comparaison avec le λ-calcul à régions.

Dans [Ama09], R. Amadio présente un système pour le λ-calcul à régions, qui est une version modifiée du λ-calcul dans laquelle on rajoute des *régions* : ce sont des cases mémoires dans lesquelles on peut lire et écrire par effet de bord. Cela permet de modéliser les références, le passage de messages, etc. Le système de type avec stratification permet également d'assurer la terminaison, et la preuve est également basée sur des candidats de réductibilité. La syntaxe est un peu différente : pour écrire dans une région r on utilise $\mathrm{set}(r, x)$ et $\mathrm{get}(r)$ pour lire dedans. La virgule désigne la composition parallèle (et non pas le séquencement), et pour spécifier la valeur initiale d'une référence avant l'exécution du programme : $\langle r \Leftarrow x\rangle$.

L'idée de la stratification est d'établir un ordre bien fondé entre les régions de sorte que dans une région, il ne peut y avoir que du code qui touche à des régions strictement inférieures. Le programme divergent classique, réécrit en λ-calcul à régions donne $S, \mathrm{set}(b, S)$ (la virgule dénote la composition parallèle) avec

$S \stackrel{\text{def}}{=} (\lambda k.k\,(), \text{set}(b,k))$. Le système de [Ama09] rejette ce terme car dans la région b on place S, un processus qui écrit sur b. Cette manière de stratifier est bien sûre équivalente à celle présentée dans ce rapport [5].

La différence principale se situe au niveau de la lecture d'une région. En λ-calcul à régions, à cause de son caractère généraliste, la lecture sur une région ne supprime pas son contenu (en fait une région peut contenir un nombre arbitraire de valeurs et une lecture sur une région en sélectionne une de façon non déterministe). Cela entraîne une lecture immédiate en λ-calcul à régions : pas besoin d'attendre que quelqu'un écrive sur la région en même temps. Ainsi, l'exemple du *spawning* (cf. section 3.4) qui est parfaitement terminant en π-calcul donne en λ-calcul à régions $S\,(), \langle b \Leftarrow S\rangle$ avec $S \stackrel{\text{def}}{=} \lambda().\,(\lambda k.\,k\,(),\ (\lambda x.\,\text{set}(b,x))\,(\text{get}(a)))\,(\text{get}(b))$. Ce terme diverge (et d'ailleurs il est refusé pour la même raison que le terme précédent).

De plus, la réduction en $\lambda\pi$ est plus générale que celle du λ-calcul à régions (*call-by-value*) et permet donc plusieurs spécialisations différentes sans changer la preuve de terminaison. En effet, le système de type pour ce calcul est très fortement lié à l'appel par valeur (effet nul pour les abstractions et les variables) alors que dans $\lambda\pi$, les types contiennent les effets, et donc il n'y a pas de question à se poser de ce côté.

Notre système permet donc, en tirant partie de la nature des communications en π-calcul de typer plus de termes.

4 Soft λπ

4.1 Introduction

On se place à présent dans le domaine de la complexité implicite. Des tentatives pour importer des méthodes utilisées dans le monde séquentiel (par exemple la *soft linear logic* [Laf04]) ont été faites pour le π-calcul d'ordre supérieur avec [LMS10]. Cette approche consiste à utiliser complètement les méthodes de *Soft Linear Logic* afin d'assurer une borne "polynomiale". Il n'est cependant pas encore très clair ce que signifie la complexité implicite dans un cadre concurrent. L'objectif de cette section sera de définir un sous-ensemble de λπ sur lequel on a des bornes sur le nombre de réductions d'un terme correctement typé.

Notons tout d'abord qu'il est facile d'étendre le système de [LMS10] à λπ. Cependant cela ne donne pas un système très intéressant car il ne supporte pas le *spawning*. L'approche utilisée ici sera d'utiliser les méthodes de *Soft Linear Logic* pour la partie séquentielle, et d'utiliser les arguments qu'on a déjà pour la partie π (à savoir, quand on a $a(x).P$ le poids de P est inférieur strictement au niveau de a).

Le but de la suite de la section est d'exposer les principales idées de la preuve sans forcément la détailler autant qu'à la section précédente.

On rappelle rapidement les idées de *soft linear logic* reprises ici : on distingue aux niveaux des types les valeurs qui peuvent être dupliquées et celles qui doivent être utilisées une seule fois : une fonction de type $\tau \to \sigma$ promet d'utiliser une seule fois son argument (syntaxe : $\lambda x.t$) alors qu'une fonction de type $!\,\tau \to \sigma$ peut utiliser plusieurs fois son argument (mais il y a toujours des restrictions) (notation $\lambda!\,x.t$). Précisément, quand on a un objet de type $!\,\tau$, on peut soit l'utiliser autant de fois qu'on veut en tant qu'objet de type τ ou une fois en tant qu'objet de type $!\,\tau$ (C'est-à-dire le passer à une fonction qui attend un $!\,\tau$).

4.2 Présentation du langage

Comme dit précédemment, on ajoute à la grammaire des termes une construction $\lambda!\,x.t$, et une construction $!\,t$. On rajoute la règle de réduction $(\lambda!\,x.t)(!\,u) \to t[u/x]$. On rajoute dans la grammaire des types la construction τ. Les règles de ce dialecte sont données dans la figure 4.2. Ces règles définissent un jugement

5. Tout terme bien typé de λπ peut l'être en attribuant un niveau différent à chaque canal

$$\frac{\Gamma;\Delta_1 \vdash t : e \qquad \Gamma;\Delta_1 \vdash u : e'}{\Gamma;\Delta_1,\Delta_2 \vdash t \parallel u : e \uplus e'} \qquad \frac{\Gamma, a : \mathrm{Ch}_k(\tau);\Delta \vdash t : \sigma}{\Gamma;\Delta \vdash (\nu a)t : \sigma} \qquad \frac{\Gamma;\Delta \vdash t : \tau \qquad \Gamma(a) = \mathrm{Ch}_k(\tau)}{\Gamma;\Delta \vdash \overline{a}\langle t\rangle : \{k\}}$$

$$\frac{}{\Gamma;\varnothing \vdash 0 : \varnothing} \qquad \frac{\Gamma, x : \tau;\Delta \vdash t : e \qquad \Gamma(a) = \mathrm{Ch}_k(\tau) \qquad e < \{k\}}{\Gamma;\Delta \vdash a(x).t : \varnothing} \qquad \frac{\Gamma, x : \tau;\Delta \vdash t : \sigma}{\Gamma;\Delta \vdash \lambda!\, x.t :! \tau \to \sigma}$$

$$\frac{\Gamma;\Delta, x : \tau \vdash t : \sigma}{\Gamma;\Delta \vdash \lambda x.t : \tau \to \sigma} \qquad \frac{\Gamma;\Delta_1 \vdash t : \tau \to \sigma \qquad \Gamma;\Delta_2 \vdash u : \tau}{\Gamma;\Delta_1,\Delta_2 \vdash t\, u : \sigma} \qquad \frac{\Gamma \cup D(x) = \tau}{\Gamma;\Delta \vdash x : \tau} \qquad \frac{\varnothing;\Delta \vdash t : \tau}{\varnothing;!\,\Delta,\Delta' \vdash !\, t :! \tau}$$

$$\frac{\Gamma \vdash t : \tau \qquad \tau \leq \sigma}{\Gamma \vdash t : \sigma}$$

FIGURE 9 – Règles de Soft$\lambda\pi$

$\Gamma;\Delta \vdash t : \tau$. On a deux contextes, l'un pour gérer les variables linéaires qui ne peuvent apparaître qu'une seule fois (Δ), et qui sera donc géré multiplicativement, et l'autre pour gérer les variables que l'on peut dupliquer comme on le souhaite (Γ) qui sera géré additivement. Quelques remarques :

- ▷ les variables liées par des réceptions sont supposées dupliquables ;
- ▷ on a deux règles pour $\lambda!\, x.t$: l'une où l'on considère que x est dupliquable de type τ et l'autre où x est considéré linéaire de type $!\,\tau$.
- ▷ la règle pour typer $!\, t$ dit essentiellement que si on sait typer t en n'utilisant que des variables linéaires, alors on sait typer $!\, t$ en rajoutant un ! aux types des hypothèses. La notation $!\,\Gamma$ désigne un contexte dont toutes les hypothèses sont de la forme $x :!\,\tau$.

Ce système vérifie encore la réduction assujettie. Pour assurer la terminaison et la borne de notre système, on associe à chaque processus un entier $m(t)$. Cet entier devra rendre compte de deux phénomènes : les communications et les β-réductions. On remarque que le type d'un objet en dit long sur les communications potentielles qu'il peut effectuer : en fait le type d'un objet décrit toutes les communications *à la racine* que peut effectuer un terme. Cependant, à un même type on peut avoir des termes qui font un nombre différent de β-réductions (par exemple $(\lambda x.0)0$ et 0) ou de communications dans un envoi ou dans une application ($\overline{a}\langle\overline{b}\langle 0\rangle \parallel b(x).0\rangle$ et $\overline{a}\langle 0\rangle$). On décompte donc la borne en deux entiers : un pour tenir compte des communications à la racine et un autre pour tenir compte du reste.

4.3 Terminaison de Soft $\lambda\pi$

Tout d'abord, on fait une hypothèse sur les termes : on suppose qu'ils sont typables en attribuant à deux canaux différents des niveaux différents (ceci est équivalent à la typabilité normale) et on ne considérera que des dérivations de cette forme. De plus, de manière classique, nos mesures sont paramétrées par un entier d, la *profondeur* du terme : elle représente le nombre d'occurences de la variable (variable de terme ou de canal) la plus dupliquée apparaîssant dans le terme, et on la notera $D(t)$. Notons que $D(t)$ n'augmente pas au cours de la réduction pour les termes clos, car on ne réduit pas sous les abstractions.

***Lemme* 8.** — *Si on a $\Gamma;\Delta \vdash t : \tau$, alors il existe Γ', Δ' tel que $a \in dom(\Gamma \cup \Delta) \mapsto$ niveau de a dans $\Gamma \cup \Delta$ soit injective et qui vérifient $\Gamma';\Delta' \vdash t : \tau$.*

Communication à la racine On s'occupe tout d'abord des communications à la racine en posant une mesure sur les types, et les termes ayant ce type.

***Définition* 2.** — *À chaque type τ, on associe un entier noté $m^C_d(\tau)$:*

- ▷ *si $\tau = e$ alors $m^C_d(\tau) = \sum_{p\in e}(1+d)^p$;*
- ▷ *si $\tau = !\,\sigma$ alors $m^C_d(\tau) = dm_C(\sigma)$;*
- ▷ *si $\tau = \sigma \to \sigma'$ alors $m^C_d(\tau) = m^C_d(\sigma')$* [6].

Si t vérifie $\Gamma;\Delta \vdash t : \tau$ alors on note $m_C(t) = m_C(\tau)$ en faisant fi de l'ambiguïté sur la dérivation.

***Lemme* 9.** — *Soit t un terme de type τ. Si $t \to t'$ par une communication à la racine, on a $m^C_d(t) > m^C_d(t')$ pour tout $d \geq D(t)$ et pour une certaine dérivation de typage pour t'.*

Démonstration. On fait le calcul dans le cas le plus simple $t = a(x).P \parallel \overline{a}\langle v\rangle \to P[v/x]$. Notons k le niveau de a utilisé dans la dérivation de typage de t. Par un lemme de substitution, $P[v/x]$ et P ont même type et donc même m^C (pour cette dérivation). On peut majorer $m^C_d(P)$: au plus chaque canal de niveau 0 à $k-1$ apparaît d fois (ils ne peuvent apparaître plus) et donc on a $m^C_d(P) \leq \sum_{i=0}^{k-1} d\cdot(d+1)^i = (d+1)^k - 1 < (d+1)^k = m^C_d(a(x).\,P\|\overline{a}\langle v\rangle)$. □

β-réductions et communications cachées Dans les autres cas, on utilise une mesure auxiliaire :

***Définition* 3.** — *On définit par induction mutelle sur t (dont on a fixé une dérivation de typage), $m^\beta_d(t)$ et $m_d(t)$:*

- ▷ $m^\beta_d(\lambda x.\,t) = m^\beta_d(t)$;
- ▷ $m^\beta_d(t\ u) = m^\beta_d(t) + m_d(u) + 1$;
- ▷ $m^\beta_d(x) = 0$;
- ▷ $m^\beta_d(t \parallel u) = m^\beta_d(t) + m^\beta_d(u)$;
- ▷ $m^\beta_d(\overline{a}\langle t\rangle) = m_d(t)$;
- ▷ $m^\beta_d(0) = 0$;
- ▷ $m^\beta_d(a(x).t) = m^\beta_d(t)$;
- ▷ $m^\beta_d(!\,t) = dm^\beta_d(t)$.

et $m_d(t) = m^C(t) + d\cdot m^\beta_d(t)$.

On retrouve ici les expressions classiques de *soft linear logic* modulo une subtilité : aux endroits où on peut avoir des communications, il faut tenir compte du m^C du sous-terme.

***Lemme* 10.** — *On a les résultats :*

- ▷ *pour tout terme t tel que $\Gamma, x:\tau;\Delta \vdash t:\sigma$ et pour tout terme u tel que $\Gamma;\Delta\vdash u:\tau$ on a $m^\beta_d(t[u/x]) \leq m^\beta_d(t) + d\cdot m^\beta_d(u)$.*
- ▷ *pour tout terme t tel que $\Gamma;\Delta, x:\tau \vdash t:\sigma$ et pour tout terme u tel que $\Gamma;\Delta\vdash u:\tau$ on a $m^\beta_d(t[u/x]) \leq m^\beta_d(t) + m^\beta_d(u)$.*

Démonstration. Par induction sur la dérivation de typage de t. □

***Théorème* 3.** — *Si $t \to t'$ et $d \geq D(t)$ alors $m_d(t) > m_d(t')$.*

6. On tiendra compte de $m^C_d(\sigma)$ lors de la seconde mesure, plus particulièrement lorsque la fonction sera appliqué à un argument de type σ

Démonstration. Par induction sur la dérivation de $t \to t'$.

- ▷ Si $(\lambda x.u)v \to u[v/x]$. Alors $m_d(u[v/x]) \leq m_d^C(u)+d\cdot(m_d^\beta(u)+m_d^\beta(v)) < m_d^C(u)+d\cdot m_d^\beta((\lambda x.u)\,v) = m_d(t)$ (par subject reduction ($u[v/x]$ et u ont le même type donc même m^C-mesure) + lemme précédent)
- ▷ Si on a $(\lambda!\,x.u)\,!\,v \to u[v/x]$, idem que le point précédent en tenant compte des définitions pour la mesure de $!\,v$. (x apparaît au plus d fois dans u)
- ▷ Si on a $uv \to uv'$ avec $v \to v'$. On a $m_d(u\,v) = m_d^C(u\,v)+d(m_d^\beta(u)+m_d(v)+1) > m_d^C(u\,v')+d(m_d^\beta(u)+m_d(v')+1)$ ce que l'on veut (car v et v' ont le même type donc même m_d^C-mesure et $m_d(v') > m_d(v)$ par induction)
- ▷ idem si on a $uv \to u'v$
- ▷ idem si c'est $\overline{a}\langle u\rangle \to \overline{a}\langle u'\rangle$
- ▷ si on a $a(x).P \| \overline{a}\langle v\rangle \to P[v/x]$: grâce au lemme 9 on sait que la mesure m^C a décrû. On calcule alors :

$$\begin{aligned} m_d^C(P[v/x]) + dm_d^\beta(P[v/x]) &< m_d^C(t) + d(m_d^\beta(P) + dm_d^\beta(v)) \\ &< m_d^C(t) + dm_d^\beta(a(x).P) + d(m_d^C(v) + dm_d^\beta(v)) \\ &< m_d^C(t) + dm_d^\beta(t) < m_d(t) \end{aligned}$$

□

Un processus donc effectue au plus $m_{D(t)}(t)$ réductions. Cela fait au pire, pour un terme utilisant n canaux et avec k la profondeur maximale, un nombre de l'ordre de $|P|^{n+k}$: en effet, il suffit que les envois se trouvent à profondeur k (c'est-à-dire derrière k bangs). Cela confirme l'intuition pour $k = 1$ ou $n = 1$.

Conclusion

On a vu dans ce rapport un moyen d'allier le λ-calcul et le π-calcul tout en ayant des garanties de terminaison et de temps de calcul. Un tel langage permet d'exprimer à la fois la partie séquentielle et la partie concurrente d'un programme et d'étudier leurs interactions.

Notons que la terminaison en π-calcul d'ordre supérieur reste toujours mystérieuse : on a ces deux façons d'imposer la terminaison, très orthogonales. Un des premiers objectifs du stage a été d'unifier ces deux familles (non détaillé ici pour des raisons d'espace, il s'agit d'une unification au cas par cas), mais il n'en a rien surgi de lumineux.

On s'est également posé la question de savoir si les systèmes qui contrôlaient les variables avaient un équivalent en π-calcul du premier ordre. Il n'a pas été possible de donner une réponse satisfaisante à cette question.

Enfin, le système de λπ pourrait être étendu pour accepter encore plus de termes. La condition $e < \{k\}$ est un peu dérangeante : on manipule des multi-ensembles mais on ne profite pas de l'information supplémentaire (au niveau du typage), pour preuve, la typabilité avec des multi-ensembles ou des entiers est équivalente. Une façon d'exploiter cette information est d'utiliser un système similaire à celui décrit dans [Dem10] (section 4.2). L'idée est de modifier la règle de réception pour tenir compte d'une série d'entrées : quand on a une série de réceptions $a_1(x_1).\ ...\ .a_n(x_n).P$ on ne demande plus que les effets dans P soient inférieurs au niveau de a_n, mais du multi-ensemble constitué des niveaux des a_i.

Références

[Ama09] Roberto M. Amadio. On stratified regions. *CoRR*, abs/0904.2076, 2009.

[Dem10] Romain Demangeon. *Terminaison des systèmes concurrents*. PhD thesis, 2010.

[DHS10] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Termination in higher-order concurrent calculi. *Journal of Logic and Algebraic Programming*, 2010.

[DS06] Y. Deng and D. Sangiorgi. Ensuring termination by typability. *Information and Computation*, 204(7) :1045–1082, 2006.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, April 1989.

[Laf04] Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2) :163–180, 2004.

[LMS10] Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. Light logics and higher-order processes. In *EXPRESS'10*, pages 46–60, 2010.

[SW01] D. Sangiorgi and D. Walker. *The Pi-Calculus : A Theory of Mobile Processes*. Cambridge University Press, 2001.

[YH00] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. In *In LICS 2000*, pages 334–345, 2000.