JoCaml et le join-calcul

Samuel Mimram

25 novembre 2003

Table des matières

Le π-calcul est un outil qui permet de modéliser la programmation concurrente en rendant compte de l'indéterminisme dans les programmes parallèles. Il est intéressant de par la richesse qu'il offre du fait du passage de noms qu'il contient (en particulier le λ-calcul, les calculs orientés objet ou encore les calculs impératifs peuvent y être implémentés de façon simple). L'expressivité de ce calcul semble bonne, mais on peut s'interroger sur sa capacité à fonder un langage dans lequel on pourrait implémenter des programmes concurrents de façon relativement naturelle. C'est dans l'optique de créer un véritable langage de programmation tout en gardant la richesse du π-calcul que le join-calcul a été créé.

En effet, intuitivement, en π-calcul, le programme de chaque acteur (les acteurs étant par exemple plusieurs ordinateurs connectés par un réseau) doit suivre le schéma suivant : « si j'ai fini mon calcul et que le résultat de celui-ci est utilisé par un autre acteur alors je peux commencer tel autre calcul ». Cependant, la structure suivante semble plus naturelle : « si les résultats de tel et tel calculs sont disponibles alors tel autre calcul peut être fait » ; c'est globalement l'approche qui a été choisie pour élaborer le join-calcul. De plus l'utilisation de canaux globaux en π-calcul est discutable : un processus peut écouter sur un canal sans qu'on aie voulu qu'il en ait le droit (ceci est lié aux problèmes de sécurité) et leur implémentation est difficile (comment envoyer un message sur un canal global ? faut-il systématiquement faire des *braodcasts* ?).

Un prototype de langage de programmation fondé sur ce calcul a été réalisé, JoCaml, qui est une extension du langage OCaml avec des constructions propre au join-calcul.

1 Le join-calcul

1.1 Le join-calcul

Soit $\mathcal{N}$ un ensemble de noms de canaux. Pour $x_1, \ldots, x_n$ noms de canaux nous noterons $\tilde{x}$ un n-uplet $(x_1, \ldots, x_n)$. La syntaxe des processus du join-calcul est la suivante :

$$P \quad ::= \quad x\langle u\rangle \quad | \quad P_1|P_2 \quad | \quad \texttt{def}\ x\langle u\rangle|y\langle v\rangle \triangleright P_1\ \texttt{in}\ P_2$$

De même qu'en π-calcul, la construction $x\,\langle u\rangle$ signifie qu'on envoie le nom de canal u sur le canal x et la construction $P_1|P_2$ signifie que les processus P_1 et P_2 sont actifs en parallèle. L'opérateur $\triangleright$ est quant à lui véritablement nouveau. Intuitivement, il permet d'introduire des règles de *réaction* entre processus de façon *dynamique*. En effet, la construction $x\,\langle u\rangle\,|y\,\langle v\rangle \triangleright P_1$ peut se lire : « si un processus envoie un nom de canal, que nous appellerons u, sur le canal x et qu'un autre processus envoie un nom de canal, que nous appellerons v, sur le canal y, alors le processus P_1 (dans lequel on aura bien sûr remplacé les noms u et v par les noms de canaux effectivement reçus) est créé ». En ce sens, elle permet de définir les réactions qui peuvent se produire. La construction $\texttt{def}\ x\,\langle u\rangle\,|y\,\langle v\rangle \triangleright P_1\ \texttt{in}\ P_2$ définit la réaction précédemment expliquée tout en liant x et y dans P_2. Le fait que x soit lié dans P_2 rend le protocole « sécurisé » : aucun autre processus ne peut écouter sur ce même canal x. Cette construction est extrêmement riche puisque c'est elle qui remplace véritablement à la fois la restriction, la réplication et la réception du π-calcul comme nous le verrons dans la suite.

Sur ces processus on peut définir les ensembles des variables reçues (rv), des variables définies (dv) et des variables libres (fv), qui nous seront utiles par la suite, par

$$\begin{aligned}
\mathrm{rv}(x\,\langle\tilde{v}\rangle) &\stackrel{\text{déf}}{=} \{u \in \tilde{v}\}\\
\mathrm{rv}(J|J') &\stackrel{\text{déf}}{=} \mathrm{rv}(J) \uplus \mathrm{rv}(J')
\end{aligned}$$

$$\begin{aligned}
\mathrm{dv}(x\,\langle\tilde{v}\rangle) &\stackrel{\text{déf}}{=} \{x\}\\
\mathrm{dv}(J|J') &\stackrel{\text{déf}}{=} \mathrm{dv}(J) \cup \mathrm{dv}(J')\\
\mathrm{dv}(J \triangleright P) &\stackrel{\text{déf}}{=} \mathrm{dv}(J)
\end{aligned}$$

$$\begin{aligned}
\mathrm{fv}(x\,\langle\tilde{v}\rangle) &\stackrel{\text{déf}}{=} \{x\} \cup \{u \in \tilde{v}\}\\
\mathrm{fv}(\texttt{def}\ D\ \texttt{in}\ P) &\stackrel{\text{déf}}{=} (\mathrm{fv}(P) \cup \mathrm{fv}(D)) - \mathrm{dv}(D)\\
\mathrm{fv}(P|P') &\stackrel{\text{déf}}{=} \mathrm{fv}(P) \cup \mathrm{fv}(P')\\
\mathrm{fv}(J \triangleright P) &\stackrel{\text{déf}}{=} \mathrm{dv}(J) \cup (\mathrm{fv}(P) - \mathrm{rv}(J))
\end{aligned}$$

La notation $\uplus$ indique que rv est en réalité un multi-ensemble mais on peut le considérer comme un ensemble si l'on prend soin de bien renommer les variables.

La congruence structurelle $\equiv$ sur les processus est définie comme la plus petite relation d'équivalence telle que, pour tous processus P, Q, R et S, et pour toutes définitions D et D' telles que $\mathrm{dv}(D)$ et $\mathrm{dv}(D')$ ne contiennent que des noms frais (*i.e.* des noms n'appartenant pas à l'ensemble des noms libres des processus considérés), on ait

$$\begin{aligned}
P|Q &\equiv Q|P\\
(P|Q)|R &\equiv P|(Q|R)\\
P|(\texttt{def}\ D\ \texttt{in}\ Q) &\equiv \texttt{def}\ D\ \texttt{in}\ P|Q\\
\texttt{def}\ D\ \texttt{in def}\ D'\ \texttt{in}\ P &\equiv \texttt{def}\ D'\ \texttt{in def}\ D\ \texttt{in}\ P
\end{aligned}$$

si $P \equiv_\alpha Q$ alors $P \equiv Q$
si $P \equiv Q$ alors $P|R \equiv Q|R$
si $R \equiv S$ et $P \equiv Q$ alors $\texttt{def}\ J \triangleright R\ \texttt{in}\ P \equiv \texttt{def}\ J \triangleright S\ \texttt{in}\ Q$

Les hypothèses sur $\mathrm{dv}(D)$ et $\mathrm{dv}(D')$ nous empêchent par exemple d'écrire $P|(\texttt{def}\ D\ \texttt{in}\ Q) \equiv \texttt{def}\ D\ \texttt{in}\ P|Q$ dans le cas où $\mathrm{dv}(D) \cap \mathrm{fv}(P) \not\equiv \emptyset$. On a en effet

par exemple $x\langle y\rangle \,|\texttt{def}\; x\langle u\rangle \,|y\langle v\rangle\; \texttt{in}\; x\langle v\rangle \not\equiv \texttt{def}\; x\langle u\rangle \,|y\langle v\rangle\; \texttt{in}\; x\langle v\rangle \,|x\langle y\rangle$ du fait que la construction `def in` lie le nom x dans le processus qui suit le `in`.

En, pour décrire l'évolution dynamique du système, on définit un système de transition étiqueté $\xrightarrow{\delta}$ (avec $\delta \in \{D\} \cup \{\tau\}$ où $\{D\}$ représente l'ensemeble des définitions) sur les processus qui est la plus petite relation telle que, pour tout $D = x\langle u\rangle \,|y\langle v\rangle \triangleright R$ on ait

$$x\langle s\rangle \,|y\langle t\rangle \xrightarrow{D} R[{}^{s}/_{u}, {}^{t}/_{v}]$$

(on note $[{}^{s}/_{u}]$ la subsitution qui « remplace » u par s dans un terme) et pour chaque transition $P \xrightarrow{\delta} P'$ on ait

$$\begin{array}{rcll} P|Q & \xrightarrow{\delta} & P'|Q & \\ \texttt{def}\; D\; \texttt{in}\; P & \xrightarrow{\delta} & \texttt{def}\; D\; \texttt{in}\; P' & \text{si } \mathrm{fv}(D) \cap \mathrm{dv}(\delta) = \emptyset \\ \texttt{def}\; \delta\; \texttt{in}\; P & \xrightarrow{\tau} & \texttt{def}\; \delta\; \texttt{in}\; P' & \text{si } \delta \neq \tau \\ Q & \xrightarrow{\delta} & Q' & \text{si } P \equiv Q \text{ et } P' \equiv Q' \end{array}$$

La relation $\rightarrow$ de réduction des processus du join-calcul est alors définie par $\rightarrow \stackrel{\text{déf}}{=} \xrightarrow{\tau}.$

1.2 La CHAM

La CHAM (acronyme de *CHemical Abstract Machine*) est une machine abstraite qui permet de définir la sémantique opérationnelle du join-calcul. Son nom vient de la métaphore qui consiste à considérer les les processus comme des molécules qui sont *mélangées* dans une solution. Lorsque plusieurs molécules se rencontrent, elles peuvent *réagir* entre elles pour former une autre molécules. Informellement, les constructions du type $A|B \triangleright C$ permettent de définir les réactions possibles : si une molécule de type A rencontre une molécule de type B, une réaction peut avoir lieu et cette réaction va consommer les deux molécules pour former un molécule de type C. Le rôle de la CHAM va être de catalyser ces réactions.

Le modèle de la CHAM utilise des solutions d'ordre supérieur du type $\mathcal{R} \vdash \mathcal{M}$ où $\mathcal{R}$ et $\mathcal{M}$ sont deux multi-ensembles. Les molécules (processus) de $\mathcal{M}$ représentent les processus actifs en parallèle et $\mathcal{R}$ définit les règles de réaction courantes ; ces règles peuvent évoluer de façon dynamique, c'est pourquoi le modèle peut être qualifié de *réflexif*. Les éléments P de $\mathcal{M}$ sont des processus définis de façon formelle par

$$P \quad ::= \quad x\langle \tilde{v}\rangle \quad | \quad \texttt{def}\; D\; \texttt{in}\; P \quad | \quad P|P$$

Dans la définition précédente, D est une définition de réaction, de même que les éléments de $\mathcal{R}$. Ces derniers sont de la forme

$$D \quad ::= \quad J \triangleright P \quad | \quad D \wedge D$$

où J est un *join pattern* de la forme

$$J \quad ::= \quad x\langle \tilde{v}\rangle \quad | \quad J|J$$

Les règles de la CHAM sont les suivantes :

$$
\begin{array}{lllllll}
\text{(str-join)} & & \vdash & P|Q & \rightleftharpoons & \vdash & P, Q \\
\text{(str-and)} & D \wedge E & \vdash & & \rightleftharpoons & D, E & \vdash \\
\text{(str-def)} & & \vdash & \texttt{def}\ D\ \texttt{in}\ P & \rightleftharpoons & D\sigma_{\mathrm{dv}} & \vdash\ P\sigma_{\mathrm{dv}} \\
 & & & & & & \\
\text{(red)} & J \triangleright P & \vdash & J\sigma_{\mathrm{rv}} & \rightarrow & J \triangleright P & \vdash\ P\sigma_{\mathrm{rv}}
\end{array}
$$

La substitution σ_{dv} dans (str-join) renomme les noms de canaux de $\mathrm{dv}(D)$ en des noms frais et distincts (*i.e.* on a $\mathrm{Dom}(\sigma_{\mathrm{dv}}) \cap \mathrm{fv}(\mathcal{R} \vdash \mathcal{M}) = \emptyset$). La substitution σ_{rv} dans (str-def) remplace les noms reçus de $rv(J)$ par les noms de canaux effectivement reçus (on aura évidemment adapté les définitions de dv et rv aux nouvelles définitions des processus, etc.).

Les règles dans le sens gauche vers droite sont appelées *règles de chauffage* car elles ont tendance à casser les molécules. Et inversement, les règles dans le sens droite vers gauche sont appelées *règles de refroidissement*. Ce sont ces règles qui justifient l'appellation de catalyseur de la CHAM.

Lemme 1. *La congruence structurelle $\equiv$ est la plus petite relation d'équivalence qui contient toutes les paires (P, Q) de processus tels que $\vdash P \rightleftharpoons^* \vdash Q$.*

La relation de réduction $\xrightarrow{\tau}$ contient exactement toutes les paires de processus (quotientés par $\equiv$) tels que $\vdash P \rightarrow \vdash Q$.

Une extension de la CHAM existe, la DCHAM (avec D pour *Distributed*), qui implémente la migration de code (les localités, les constructions du type `go` et la résistance aux pannes) dont nous ne verrons que l'implémentation dans JoCaml.

1.3 Expressivité

De même qu'en π-calcul, il est relativement simple de montrer qu'on peut simuler le passage d'un n-uplet de valeurs dans un canal au lieu d'une seule valeur. C'est pourquoi précédemment nous nous sommes implicitement permis d'envoyer des n-uplets de noms de canaux sur les canaux alors qu'*a priori* seul le passage d'un seul nom de canal est permis par la syntaxe. Cet encodage implicite nous sera utile par la suite.

Par contre il on ne peut pas considérer que les join-patterns sont des valeurs du premier ordre car c'est cette restriction qui assure que le calcul reste typable.

1.4 Observabilité

Pour pouvoir considérer les processus de l'extérieur et dire que deux processus qui font « la même chose » sont équivalents, il nous faut définir formellement ce que l'on peut observer d'un processus. Informellement, la seule façon pour un processus de communiquer avec l'extérieur, est d'envoyer un message sur un canal dont le nom est libre dans celui-ci et d'attendre une réponse de la part du processus l'englobant.

On définit donc, pour chaque nom libre x, un barbelé asynchrone en sortie seulement (*asynchronous, output-only barb* pour les anglophiles), noté $\Downarrow_x$, qui teste la capacité qu'a un processus d'émettre quel que nom que ce soit sur x.

Dans la suite, on écrira $\rightarrow^*$ pour $(\rightarrow \cup \rightleftharpoons)^*$. On a :

$$P \Downarrow_x \quad \stackrel{\text{déf}}{=} \quad x \in \text{fv}(P) \wedge \exists \tilde{v}, \mathcal{R}, \mathcal{M}, \emptyset \vdash P \rightarrow^* \mathcal{R} \vdash \mathcal{M}, x\langle\tilde{v}\rangle$$

La congruence observationnelle $\approx$ est alors définie comme la plus grande relation d'équivalence qui soit un raffinement des barbelés en sortie $\Downarrow_x$, c'est-à-dire close par réduction faible et qui est une congruence pour les définitions et les compositions en parallèle : pour tous processus P et Q, si $P \approx Q$ alors

1. $\forall x \in \mathcal{N}$, si $P \Downarrow_x$ alors $Q \Downarrow_x$
2. si $P \rightarrow^* P'$ alors $\exists Q', Q \rightarrow^* Q'$ et $P' \approx Q'$
3. $\forall D, \texttt{def}\ D\ \texttt{in}\ P \approx \texttt{def}\ D\ \texttt{in}\ Q$
4. $\forall R, R|P \approx R|Q$

Deux processus équivalents au sens de $\approx$ sont indistinguables de l'extérieur dans le sens où à chaque fois que l'un peut envoyer un nom de canal sur un canal, l'autre le peut aussi et cela reste vrai quelque soit l'environnement dans lequel ils sont et cela reste vérifié si les deux processus font un pas de réduction. On aura par exemple :

$$\begin{array}{rcll} P & \approx & Q & \text{si } P \equiv Q \\ z\langle x\rangle & \not\approx & z\langle y\rangle & \\ \texttt{def}\ z\langle t\rangle \triangleright t\langle u\rangle\ \texttt{in}\ z\langle x\rangle & \not\approx & \texttt{def}\ z\langle t\rangle \triangleright t\langle u\rangle\ \texttt{in}\ z\langle y\rangle & \\ \texttt{def}\ u\langle z\rangle \triangleright v\langle z\rangle\ \texttt{in}\ x\langle u\rangle & \approx & x\langle v\rangle & \end{array}$$

Dans les deux calculs, il n'est pas possible d'observer la réception d'un message (on a par exemple en π-calcul : $x(u).\overline{x}u \approx_\pi 0$) et l'on ne peut pas distinguer deux canaux distincts qui ont le même comportement extérieur – la congruence $\approx_\pi$ étant définie comme la congruence barbelée asynchrone dont les barbes sont les émissions sur les canaux libres. Cette seconde propriété est celle qui motive la définition de l'*équateur* entre deux canaux x et y :

$$M^\pi_{x,y} \quad \stackrel{\text{déf}}{=} \quad !x(u).\overline{y}u|!y(v).\overline{x}v$$

Ce processus reçoit en permanence des valeurs sur x et les renvoie sur y et vice-versa de sorte que, quel que soit le canal utilisé pour l'émission, la valeur émise peut être lue sur l'autre canal après un pas de réduction interne. Il permet ainsi de rendre indistinguables les canaux x et y.

Lemme 2. *Pour tous π-processus Q et R tels que $Q[^x/_y] \approx_\pi R[^x/_y]$, on a $M^\pi_{x,y}|Q \approx_\pi M^\pi_{x,y}|R$.*

1.4.1 Implémentation du λ-calcul

Comme habituellement, la syntaxe des termes du λ-calcul est :

$$T \quad ::= \quad x \quad | \quad \lambda x.T \quad | \quad TT$$

Lors de l'interprétation des termes du λ-calcul dans le join calcul, le choix d'une stratégie d'évaluation des λ-termes doit être fait. Nous présentons l'implémentation deux stratégies : par nom et par valeur.

Stratégie d'appel par nom Dans cette stratégie, on effectue la β-réduction la plus à gauche et aucune réduction ne peut avoir lieu sous un λ. L'interprétation des termes du λ-calcul est la suivante :

$$\begin{aligned}
[\![x]\!]_v &\stackrel{\text{déf}}{=} x\langle v\rangle \\
[\![\lambda x.T]\!]_v &\stackrel{\text{déf}}{=} \texttt{def}\ \kappa\langle x, w\rangle = [\![T]\!]_w\ \texttt{in}\ v\langle\kappa\rangle \\
[\![TU]\!]_v &\stackrel{\text{déf}}{=} \texttt{def}\ x\langle u\rangle = [\![U]\!]_u\ \texttt{in def}\ w\langle\kappa\rangle = \kappa\langle x, v\rangle\ \texttt{in}\ [\![T]\!]_w
\end{aligned}$$

Intuitivement, le processus $[\![T]\!]_v$ va envoyer sur le canal v un processus qui va faire office de « serveur d'évaluation » du λ-terme T, accessible par le canal κ. Pour avoir le résultat de l'évaluation de T, il faut faire une requète sur le canal κ en lui donnant comme argument deux noms de canaux : x qui va répondre aux requètes de $[\![T]\!]$ pour savoir la valeur de son argument, et w sur lequel sera finalement envoyé le résultat si l'évaluation termine.

L'image de la traduction est exactement sous-ensemble déterministe du join-calcul c'est-à-dire l'ensemble de processus sans composition parallèle, join-pattern ou de « $\wedge$ » dans leur définition. Les réductions pour les processus de ce sous-ensemble sont bien sûr séquentielles.

Stratégie d'appel par valeur en parallèle Dans cette stratégie, le λ-terme TU peut être β-réduit lorsque les λ-termes T et U ont eux-même été β-réduits. Les réductions de T et de U peuvent se faire en parallèle. Là encore, aucune réduction ne peut s'effectuer sous un λ.

$$\begin{aligned}
[\![x]\!]_v &\stackrel{\text{déf}}{=} v\langle x\rangle \\
[\![\lambda x.T]\!]_v &\stackrel{\text{déf}}{=} \texttt{def}\ \kappa\langle x, w\rangle = [\![T]\!]_w\ \texttt{in}\ v\langle\kappa\rangle \\
[\![TU]\!]_v &\stackrel{\text{déf}}{=} \texttt{def}\ \tau\langle\kappa\rangle\,|u\langle w\rangle = \kappa\langle w, v\rangle\ \texttt{in}\ [\![T]\!]_t|[\![U]\!]_u
\end{aligned}$$

Cette encodage, bien que non déterministe est confluent.

Montrons maintenant que le join-calcul et le π-calcul ont le même pouvoir expressif. Plus précisément :

Théorème 1. *Le join-calcul et le π-calcul asynchrone on le même pouvoir expressif quant à leurs congruences faibles barbelées en sortie seulement (i.e. respectivement $\approx$ et $\approx_\pi$ qui ont été définies plus haut pour le join-calcul et pour le π-calcul).*

1.4.2 Implémentation du π-calcul

Pour encoder le π-calcul dans le join calcul, à chaque nom de canal x du π-calcul, on associe deux canaux x_{i} (pour les réceptions) et x_{o} (pour les émissions). Les émetteurs envoient simplement des valeurs sur le canal x_{i} ; les destinataires définissent un nom de canal pour leur continuation et l'envoient comme « offre de réception » sur x_{i}. Un encodage naïf serait alors

$$\begin{aligned}
[\![P|Q]\!]_\pi &\stackrel{\text{déf}}{=} [\![P]\!]_\pi|[\![Q]\!]_\pi \\
[\![\nu x.P]\!]_\pi &\stackrel{\text{déf}}{=} \texttt{def}\ x_{\text{o}}\langle v_{\text{o}}, v_{\text{i}}\rangle\,|x_{\text{i}}\langle\kappa\rangle \triangleright \kappa\langle v_{\text{o}}, v_{\text{i}}\rangle\ \texttt{in}\ [\![P]\!]_\pi \\
[\![\overline{x}v]\!]_\pi &\stackrel{\text{déf}}{=} x_{\text{o}}\langle v_{\text{o}}, v_{\text{i}}\rangle \\
[\![x(v).P]\!]_\pi &\stackrel{\text{déf}}{=} \texttt{def}\ \kappa\langle v_{\text{o}}, v_{\text{i}}\rangle \triangleright [\![P]\!]_\pi\ \texttt{in}\ x_{\text{i}}\langle\kappa\rangle \\
[\![!x(v).P]\!]_\pi &\stackrel{\text{déf}}{=} \texttt{def}\ \kappa\langle v_{\text{o}}, v_{\text{i}}\rangle \triangleright x_{\text{i}}\langle\kappa\rangle\,|[\![P]\!]_\pi\ \texttt{in}\ x_{\text{i}}\langle\kappa\rangle
\end{aligned}$$

Malheureusement cet encodage n'est pas totalement abstrait[1] (*fully abstract*). Par exemple, la traduction $[\![\overline{x}a|\overline{x}b|x(u).\overline{y}u]\!]_\pi$ ne peut plus être réduite car il n'y a pas de νx l'englobant. Pire, $[\![x(u).\overline{x}u]\!]_\pi$ laisse un barbelé sur x_i qui révèle la présence d'une entrée pour x, nous permettant de distinguer le processus du processus vide 0 alors que l'on a $x(u).\overline{x}u \approx_\pi 0$.

Pour protéger la traduction des contextes « hostiles » (*i.e.* susceptibles de faire des observations qui permettraient de distinguer les encodages de deux processus équivalents), les noms qui résultent des noms libres dans les π-processus doivent mettre en place un pare-feu qui assurera la protocole : chaque canal x est représenté par plusieurs paire $x_\mathrm{o}, x_\mathrm{i}$, ces paires étant « reliées » par un équateur, et de nouvelles paires sont créées dynamiquement pour garantir la sécurité du protocole lorsqu'une paire est transmise ou reçue. On est donc amené à définir les contextes suivants :

$$\begin{array}{rcl}
\mathcal{P}_x[\,] & \stackrel{\text{déf}}{=} & \texttt{def}\ x_\mathrm{l}\,\langle v_\mathrm{o}, v_\mathrm{i}\rangle\,|x_\mathrm{i}\,\langle\kappa\rangle \triangleright \kappa\,\langle v_\mathrm{o}, v_\mathrm{i}\rangle\ \texttt{in def}\ x_\mathrm{o}\,\langle v_\mathrm{o}, v_\mathrm{i}\rangle \triangleright p\,\langle v_\mathrm{o}, v_\mathrm{i}, x_\mathrm{l}\rangle\ \texttt{in}\ [\,] \\
\mathcal{E}_x[\,] & \stackrel{\text{déf}}{=} & \mathcal{P}_x\,[x_\mathrm{e}\,\langle x_\mathrm{o}, x_\mathrm{i}\rangle\,|[\,]] \\
\mathcal{M}[\,] & \stackrel{\text{déf}}{=} & \texttt{def}\ p\,\langle x_\mathrm{o}, x_\mathrm{i}, \kappa\rangle \triangleright \mathcal{P}_y[\kappa\,\langle y_\mathrm{o}, y_\mathrm{i}\rangle\,|[\![M^\pi_{x,y}]\!]]\ \texttt{in}\ [\,]
\end{array}$$

Pour chaque nom libre x, $\mathcal{P}_x$ encode la création d'un nouveau proxy pour sa sortie, $\mathcal{E}_x$ fait de même et exporte le proxy sur x_e et enfin $\mathcal{M}$ définit récursivement le créateur p de proxy pour toute la traduction.

Théorème 2. *Pour tous π-processus P et Q, on a : $P \approx_\pi Q$ si et seulement si $\mathcal{E}[[\![P]\!]_\pi] \approx \mathcal{E}[[\![Q]\!]_\pi]$ avec $\mathcal{E}[[\![P]\!]_\pi] \stackrel{déf}{=} \mathcal{M}[\mathcal{E}_{x_1}[\ldots\mathcal{E}_{x_i}[\ldots\mathcal{E}_{x_k}[[\![P]\!]_\pi]\ldots]\ldots]]$ où $\{x_i\} = \mathrm{fv}(P)$.*

1.4.3 Implémentation dans le π-calcul

La traduction inverse est plus simple car le join-calcul peut être considéré comme du π-calcul avec des restrictions sur les constructions de communication.

$$\begin{array}{rcl}
[\![Q|R]\!]_\mathrm{j} & \stackrel{\text{déf}}{=} & [\![Q]\!]_\mathrm{j}|[\![R]\!]_\mathrm{j} \\
[\![x\,\langle v\rangle]\!]_\mathrm{i} & \stackrel{\text{déf}}{=} & \overline{x}v \\
[\![\texttt{def}\ x\,\langle u\rangle\,|y\,\langle v\rangle \triangleright Q\ \texttt{in}\ R_\mathrm{j}]\!] & \stackrel{\text{déf}}{=} & \nu xy.\,(!x(u).y(v).[\![Q]\!]_\mathrm{j}|R_\mathrm{j})
\end{array}$$

Là encore, la mise en œuvre d'un pare-feu est nécessaire :

$$\begin{array}{rcl}
R_{xy} & \stackrel{\text{déf}}{=} & !x(v).\nu v_\mathrm{e}.(\overline{r}v_\mathrm{e}v|\overline{y}v_\mathrm{e}) \\
\mathcal{R}[\,] & \stackrel{\text{déf}}{=} & \nu r.!r(x, x_\mathrm{e}).R_{xx_\mathrm{e}}|[\,] \\
\mathcal{E}^\pi_x[\,] & \stackrel{\text{déf}}{=} & \nu x.(R_{xx_\mathrm{e}}|[\,])
\end{array}$$

Théorème 3. *Pour tous processus Q et R du join-calcul on a : $Q \approx R$ si et seulement si $\mathcal{E}^\pi[[\![Q]\!]_\mathrm{j}] \approx_\pi \mathcal{E}^\pi[[\![R]\!]_\mathrm{j}]$.*

[1] Rappelons que si $\mathcal{P}_1$ et $\mathcal{P}_2$ sont deux calculs processus avec respectivement $\approx_1$ et $\approx_2$ comme équivalence sur leurs processus alors $\mathcal{P}_2$ est dit *plus expressif* que $\mathcal{P}_1$ s'il exsite un *encodage totalement abstrait* $[\![\ ]\!]_{1\to2}$ de $\mathcal{P}_1$ dans $\mathcal{P}_2$ c'est-à-dire que pour tous processus P, Q de $\mathcal{P}_1$ l'on ait :

$$P \approx_1 Q \quad \Leftrightarrow \quad [\![P]\!]_{1\to2} \approx_2 [\![Q]\!]_{1\to2}$$

2 JoCaml

Nous avons vu que le join-calcul semblait être plus efficace que le π-calcul en ce qui concerne la programmation de part la forme des constructions qu'il offre. Cependant l'on ne peut véritablement juger de cela qu'après avoir effectivement programmé avec langage fondé sur ce calcul. C'est pourquoi il est intéressant de regarder les possibilités des JoCaml qui est une extension de OCaml avec des constructions propres au join calcul.

Le prototype de langage de programmation JoCaml est disponible gratuitement à l'adresse : `http://pauillac.inria.fr/jocaml/`.

Nous supposerons que le lecteur est déjà familiarisé avec OCaml et nous ne détaillerons que les constructions spécifiques à JoCaml.

2.1 Nouvelles constructions

Conceptuellement, de nouveaux objets ont été introduits dans JoCaml : les canaux. Ils peuvent être soit synchrones soit asynchrones et sont des valeurs du premier ordre. On les crée avec les mots clé `let def`.

Pour créer un canal asynchrone, on fait suivre le nom du canal d'un point d'exclamation (`!`) comme dans l'exemple suivant :

```
        Join Objective Caml version 1.07

# let def echo! x = print_int x;
  ;;
val echo : <<int>> = chan
```

Cette définition crée le canal `echo` de type `<<int>>` qui va transporter des données de type `int`. Lorsqu'on envoie une valeur (qui doit être un entier) dans ce canal, aucune valeur n'est retournée (même pas `unit`) car le canal est asynchrone et la fonction `print_int` est appelée avec comme argument la valeur envoyée sur le canal ; là encore, le canal étant asynchrone il n'est pas possible de prédire quand la fonction sera appelée. La fonction `print_int` renvoyant la valeur `()` de type `unit`, le point-virgule à la fin de la première ligne est indispensable pour « oublier » cette valeur. Si plusieurs processus envoient en même temps des valeurs dans ce canal, l'ordre dans lequel l'affichage de ces valeurs par la fonction `print_int` sera fait n'est pas spécifié. Étant donné que seules les déclarations et les expressions sont autorisées à *top-level*, une nouvelle construction est nécessaire pour utiliser ces canaux en tant qu'expressions ; le mot clé `spawn` (qui signifie *engendrer*) permet d'envoyer des valeurs dans ces canaux et ce de façon asynchrone :

```
# spawn {echo 1 | echo 2};;
- : unit = ()
21
```

La fonction `spawn` envoie ici *en parallèle* (ce qui est indiqué par le mot clé `|`) les valeurs `1` et `2` sur le canal `echo`. Il important de bien noter que l'ordre dans lequel ces `echo` vont « réagir » est indéterministe et le programme aurait très bien pu afficher `12`.

La définition des canaux synchrones se fait de la même manière mais en omettant le point d'exclamation à la fin du nom de canal. Les canaux synchrones, eux,

renvoient une valeur. En première approche rien ne semble distinguer les canaux synchrones des fonctions habituelles, sauf peut-être la syntaxe (en particulier le mot clé `reply` pour donner la valeur de retour du canal) :

```
# let def print_and_double x = print_int x; reply (2 * x);;
val print_and_double : int -> int = <fun>
# print_and_double 3;;
3- : int = 6
```

et, en effet, l'on aurait très bien pu définir `print_and_double` comme une fonction OCaml habituelle :

```
# let print_and_double x = print_int x; (2 * x);;
val print_and_double : int -> int = <fun>
# print_and_double 3;;
3- : int = 6
```

Ce qui différencie fondamentalement les canaux synchrones des simples fonctions, est le fait qu'on puisse les utiliser dans des *join patterns* (l'implémentation de la construction `def` $x\,\langle u\rangle\,|y\,\langle v\rangle \triangleright P_1$ `in` P_2). Un exemple de leur utilisation est celui du compteur :

```
# let def count! n | get () = count n | reply n to get
       or count! n | inc () = count (n + 1) | reply () to inc
  ;;
val get : unit -> int = <fun>
val count : <<int>> = chan
val inc : unit -> unit = <fun>
# spawn {count 0}
  ;;
- : unit = ()
# inc (); inc (); inc (); get ()
  ;;
- : int = 3
```

En termes chimiques, la première ligne peut se lire : « si j'ai à la fois une molécule `count` n et une molécule `get ()` en solution alors je réémet une molécule `count` n et l'appel à la fonction `get` doit renvoyer le n sus-mentionné ». Si j'ai un compteur (représenté par le canal asynchrone `count`) et que je veux en connaître la valeur (par un appel au canal synchrone `get`) alors `get` doit renvoyer cette valeur et je dois remettre un compteur `count` en solution car les *join patterns* « consomment » les canaux sur lesquels ils se synchronisent. Mais je peux aussi vouloir l'incrémenter : si j'ai un compteur en solution et une demande d'incrémentation (par le canal synchrone `get`) alors je remets un compteur en solution dont la valeur sera la valeur de l'ancien compteur incrémentée de 1 et le canal synchrone `inc` renvoie la valeur `()`.

La syntaxe de JoCaml permet d'être plus concis en rendant implicite le fait que la réponse est faite par le canal synchrone s'il n'y en a qu'un seul dans la définition. De plus la valeur `()` est renvoyée si rien n'est précisé :

```
let def count! n | get () = count n | reply n
     or count! n | inc () = count (n + 1) | reply
;;
```

Bien sûr la couche OCaml permet toujours de gérer les portées des canaux de même qu'elle gérait les portées des variables. On peut donc définir un générateur de compteurs de la façon suivante :

```
# let def new_counter () =
    let def count! n | get () = count n | reply n
         or count! n | inc () = count (n + 1) | reply
    in
      count 0 | reply (get, inc)
  ;;
val new_counter : unit -> (unit -> int) * (unit -> unit) = <fun>
# let g, i = new_counter ()
  ;;
val g : unit -> int = <fun>
val i : unit -> unit = <fun>
# i (); i (); g ()
  ;;
- : int = 2
```

Le canal synchrone `new_counter` lorsqu'il est appelé crée comme précédemment un canal asynchrone `count` et deux canaux synchrones `get` et `inc` et met dans la solution une molécule `count 0`. Mais il ne renvoie que les canaux `get` et `inc`. Le canal `count` n'est pas exporté. Il est donc garanti que seuls des appels aux canaux synchrones `get` et `inc` pourront le modifier ; les autres canaux n'y ont pas accès et ne peuvent pas émettre dessus.

JoCaml contient aussi des primitives qui peuvent permettre de distribuer les calculs. Supposons que j'aie un ordinateur 1 qui soit très rapide pour calculer les carrés. Sur cet ordinateur, je vais exécuter le programme suivant :

```
# let def f x =
      print_string "Squaring...\n";
      reply x * x
    in
      Ns.register "square" f vartype
    ;;
Warning: VARTYPE replaced by type
( int ->  int) metatype
```

L'avertissement nous indique simplement que `vartype` a été remplacé par le type inféré pour `f` (c'est justement l'utilité de ce mot clé qui permet de ne pas avoir à explicitement donner le type des identifiants que l'on enregistre).

La fonction `Ns.register` va enregistrer la fonction `f` sous le nom `square` auprès d'un serveur de noms (qui est spécifié par la variable shell `JNSNAME`). Ensuite un autre ordinateur (disons l'ordinateur 0), peut utiliser cette fonction, qui sera exécutée sur l'ordinateur 1 et dont le résultat lui sera renvoyé et ce, de façon transparente :

```
# let sqr = Ns.lookup "square" vartype in
    print_int (sqr 13)
    ;;
Warning: VARTYPE replaced by type
( int ->  int) metatype
169- : unit = ()
```

Il est à noter que l'affichage de `Squaring...` se fait bien sur l'ordinateur 1 : l'appel au canal synchrone `sqr` se fait *à distance*, sur l'ordinateur qui l'a enregistré.

JoCaml offre de plus un ensemble de constructions qui permettent de faire migrer du code et ce de manière *subjective*. Une *localité* est un bloc de code qui est localisé : une localité peut *migrer* c'est à dire elle même aller dans une autre localité (éventuellement physiquement située sur un autre ordinateur) ou encore être la destination d'une migration. Ainsi un ordinateur peut exporter une localité (`here`) qu'il vient de déclarer :

```
# let loc here do {}
  ;;
val here : Join.location = <abstr>
# Ns.register "here" here vartype
  ;;
Warning: VARTYPE replaced by type
 Join.location metatype
- : unit = ()
```

Et un second ordinateur peut déclarer à son tour une localité qu'il va faire migrer dans la première et ainsi faire exécuter son code par le premier ordinateur :

```
# let loc mobile
    do
      {
        let here = Ns.lookup "here" vartype in
          go here;
          print_string "Computing a lot of complicated things...";
          let x = 5 in
          let y = x * x in
            print_int y;
      }
Warning: VARTYPE replaced by type
 Join.location metatype
val mobile : Join.location = <abstr>
```

Ici par exemple la localité `mobile` va migrer dans la localité `here` et va afficher `Computing a lot of complicated things...` non pas sur l'ordinateur local mais sur l'ordinateur sur lequel se trouve la localité `here`. La migration se fait grâce à l'instruction `go` qui est à rapprocher de la construction `in` du calcul des ambients si l'on assimile les localités à des ambients.

2.2 Un exemple : $\sum_{i=1}^{n} i$

L'intérêt de JoCaml se fait véritablement sentir lorsque l'on cherche à distribuer des calculs. Supposons souhaitions calculer la somme des entiers de 1 à $n = kp$ et que nous avons à notre disposition plusieurs machines. Une idée naturelle, l'addition étant associative et commutative, va être de découper le calcul en sous sous-sommes $\sum_{i=jp+1}^{(j+1)p} i$ et de distribuer ces sous-tâches aux différentes machines disponibles (oui, il existe une formule qui nous donnerait directement le résultat mais là n'est pas le propos). En appelant n `size` et p `chunk`, on obtient :

```
# let size = 1000
  let chunk = 200

  let def join! (name, there) =
    let loc mobile
    do
      {
        let def start! (i, finished) =
        let def loop! (u, s) =
          if u < (i + 1) * chunk then
            loop (u + 1, s + u)
          else
            finished s
        in
          loop (i * chunk, 0)
        in
        go there;
        worker (name, mobile, start)
      }
    in
      print_string (name ^ " joins the party\n");
      flush stdout;

  and job! i | worker! (name, there, start) =
    print_string (name ^ ", " ^ string_of_int (i * chunk) ^ "\n"); flush stdout;
    let def once! () | finished! s = add s | worker (name, there, start)
         or once! () | failed! () = print_string (name ^ " went down\n"); job i
    in
      once () | start (i, finished) | fail there; failed ()

  and result! (n, s) | add! ds =
    let s' = s + ds in
      if n > 0 then
        result (n - 1, s')
      else
        { print_string ("The sum is " ^ string_of_int s' ^ "\n"); exit 0; }
  ;;
val size : int = 1000
val chunk : int = 200
val join : <<(string * Join.location)>> = chan
val worker : <<(string * Join.location * <<(int * <<int>>)>>)>> = chan
val job : <<int>> = chan
val add : <<int>> = chan
val result : <<(int * int)>> = chan

# spawn { result (size / chunk - 1, 0) | let def jobs! n = job n
        | if n > 0 then jobs (n - 1) in jobs (size / chunk - 1) }
  ;;
- : unit = ()
```

```
# Ns.register "join" join vartype;
  Join.server ()
  ;;
Warning: VARTYPE replaced by type
<<( string *  Join.location)>> metatype
- : unit = ()
```

Le canal asynchrone `join` permet à un agent de signaler qu'il est disponible, les deux paramètres de ce canal sont le nom de l'agent (qui ne sert que pour les messages de debogage) ainsi qu'une localité qui se trouve sur l'agent. Lorsque ce canal est appelé, il va y faire migrer une localité appelée `mobile` qui va y émettre sur un canal `worker`. Ce dernier canal indique véritablement que l'agent est prêt à faire un calcul. Lorsqu'un calcul reste à faire, son numéro est émis sur le canal `job` et si un agent est disponible, le calcul lui est attribué. De plus les pannes sont gérées : si la localité `there` tombe en panne (ou est déconnectée) alors la fonction `fail` – qui est une fonction prédéfinie par JoCaml – retourne et une molécule `failed ()` est émise. La molécule `failed ()` permet de savoir que le calcul a échoué et de le relancer en réémettant le numéro du calcul qui a échoué sur le canal `job`. Lorsque la somme d'un bloc de taille p est terminée on l'ajoute à la somme totale qui est stockée dans le canal asynchrone `result`. L'initialisation consiste bien sûr à émettre tous les `jobs` qui représentent les sous-calculs qui sont à effectuer.

Voici un exemple de code d'un agent qui va proposer sa puissance de calcul au programme précédent :

```
# let loc worker do
    {
      let join = Ns.lookup "join" vartype in
        join ("reliable", worker)
    }
  ;;
Warning: VARTYPE replaced by type
<<( string *  Join.location)>> metatype
val worker : Join.location = <abstr>
```

Il ne fait que définir une localité locale et l'enregistrer auprès du programme principal par l'intermédiaire du canal synchrone `join`.

2.3 Limites

Un très gros frein au choix de JoCaml pour l'implémentation de programmes nécessitant de la concurrence est le fait que JoCaml n'est absolument pas maintenu. En particulier, il est fondé sur OCaml 1.07 (si si, vous avez bien lu) alors que OCaml en est actuellement à sa version 3.07 ; il contient donc potentiellement un grand nombre de bugs de OCaml qui ont été corrigés dans les versions ultérieures de OCaml. De plus, par expérience, il est loin d'être évident qu'il va compiler sans problème sur n'importe quelle machine.

Un inconvénient de l'implémentation est qu'elle est centralisée en ce qui concerne l'exportation des valeurs Caml : il y a un unique serveur de nom auxquels tous les programmes qui veulent importer ou exporter des canaux ou des localités doivent se connecter et la centralisation est toujours critiquable en

ce qui concerne les applications distribuées car le serveur central représente à la fois une pièce critique du programme et parfois un goulot d'étranglement en termes de communications.

Enfin, l'implémentation est naïve d'un point de vue « algorithmique ». En effet, si le programme déclare un join-pattern de la forme $A|B \triangleright C$, dès que deux molécules a_1 et c_1 avec a_1 qui vérifie A et b_1 qui vérifie B, ces molécules sont consommées et la molécule C correspondante est créée. Dans certains cas il aurait été algorithmiquement préférable d'attendre une molécule b_2 vérifiant B et de faire réagir a_1 et b_2 puis de faire réagir b_1, « plus tard », avec une autre molécule vérifiant A. Ceci est en effet compatible avec la sémantique de JoCaml et est un choix qui ne concerne que l'implémentation. Cependant il est très dur au sens algorithmique – voire impossible – pour un compilateur de prévoir et d'optimiser la mise en présence des molécules pour qu'elles réagissent. On peut donc légitimement se demander si cela n'est pas une limite du join-calcul lui-même qui n'est pas assez fin pour exprimer la façon dont les molécules doivent s'agiter dans la solution ou si l'on ne pourrait pas faire un calcul, dans lequel on pourrait projeter le join-calcul, qui permettrait d'exprimer plus finement ce genre de propriétés.