École Normale Supérieure
Département d'Informatique

Rapport de Stage de L3

Conception et Implémentation d'une Machine Abstraite pour HOcore

Lionel Zoubritzky
lionel.zoubritzky@ens.fr

Juin - Juillet 2017

Dirigé par
Alan Schmitt

IRISA / INRIA Rennes

Table des matières

1 Introduction

Pour étudier les propriétés de programmes ou de langages de programmation, il est souvent nécessaire de les modéliser par des systèmes formels plus faciles à manipuler. Le lien à établir entre ces systèmes formels, comme le λ-calcul, et le langage étudié, par exemple OCaml, peut être cependant assez éloigné : on a alors recours à des machines abstraites pour fournir une modélisation intermédiaire du système formel, et le rapprocher du fonctionnement de la machine concrète.

La notion de machine abstraite est en soi assez vague, puisqu'elle peut recouvrir des niveaux d'abstractions très divers. Parmi les plus connues, on trouve notamment la machine de Turing qui sert de référence dans la théorie de la calculabilité. Une liste des machines abstraites conçues pour des langages de programmation explicites est donnée dans [4]. On peut noter en particulier la *Java Virtual Machine* qui fait partie des plus utilisées puisqu'elle sert de langage intermédiaire lors de la compilation du code Java.

Dans le cadre des langages concurrents, un des objectifs est de pouvoir simuler de façon réaliste la distribution, c'est-à-dire la capacité pour plusieurs unités de calcul indépendantes de pouvoir effectuer des tâches en interne, ou de les communiquer sur un réseau. Pour ce faire, plusieurs systèmes formels ont été créés pour les modéliser, qu'on appelle calculs de processus : ils sont en majorité inspiré du Calcul de Système de Communication (on pourra se référer à [10] pour une introduction complète sur ce calcul). Tous disposent d'un moyen d'émettre et de recevoir des messages, ce qui, avec une certaine notion de parallélisme, forme le cœur des calculs de processus.

Il existe une machine abstraite très générale pour les calculs de processus, la *Chemical Abstract Machine* [1] qui modélise ces émissions et réceptions de messages par des constituants d'une réaction chimique. Sa généralité lui permet de modéliser plusieurs calculs comme le π-calcul [6] ou le *Distributed Join-Calculus* [7]. Cependant, sa nature est très éloignée de l'implémentation qui est faite de ces calculs. D'autres machines abstraites existent, chacune spécifique à un calcul comme [2] pour le π-calcul, [13] pour *Safe Ambient* ou [8] pour le M-calcul par exemple.

Le M-calcul est en particulier un calcul d'ordre supérieur, c'est-à-dire que les messages transmis sont eux-mêmes des processus de M-calcul. Ceci modélise la réalité des processus distribués dans lesquels du code exécutable peut-être transmis sur le réseau. Le π-calcul possède aussi un pendant d'ordre supérieur, HOπ [12]. HOπ ne dispose cependant pas encore de machine abstraite permettant de le modéliser, alors qu'il s'agit d'un calcul de processus d'ordre supérieur de référence.

Ce stage se situe dans le cadre du développement d'une machine abstraite pour HOπ. Plus précisément, j'ai conçu une machine abstraite pour HOcore [9], un calcul de processus d'ordre supérieur minimal, inspiré de HOπ mais simplifié de façon à ne contenir que les constructions nécessaires à un calcul de processus d'ordre supérieur. J'ai ensuite prouvé que la machine abstraite simulait de façon adéquate le calcul, selon des notions de correction

et de complétude développées en 3.2.2.[1] Enfin, j'ai réalisé deux implémentations de la machine, une en OCaml afin de visualiser son action, et l'autre en HOcore afin de mieux comprendre ce calcul et pour montrer son expressivité.

2 HOcore

2.1 Définition

2.1.1 Syntaxe

La syntaxe d'un processus HOcore est définie par récurrence avec

$$P ::= \quad P \parallel P \;\middle|\; \bar{a}\,\langle P\rangle \;\middle|\; a(x).P \;\middle|\; x \;\middle|\; \star$$

— $P \parallel Q$ désigne la composition en parallèle des deux processus P et Q ;
— $\bar{a}\,\langle P\rangle$ désigne l'émission du processus P sur le canal a. Comme le message émis est lui-même un processus, le calcul est d'ordre supérieur ;
— $a(x).P$ désigne la réception d'un message sur le canal a avec pour continuation le processus P, dans lequel x désigne le message reçu.
— x est une variable ;
— $\star$ désigne le processus vide, qui n'agit pas.

L'opération $\parallel$ est commutative, associative et possède $\star$ comme élément neutre.

2.1.2 Réduction

HOcore dispose d'une unique règle de réduction, consistant en la communication d'un message entre une émission et une réception.

Formellement, cette règle s'écrit

$$P \parallel \bar{a}\,\langle Q\rangle \parallel a(x).R \;\rightarrow\; P \parallel R\{x \leftarrow Q\}$$

Le terme $R\{x \leftarrow Q\}$ correspond à R dans lequel toutes les occurrences de x ont été remplacées par Q.

Exemples :

— $\star$ n'admet pas de réduction.
— $\bar{a}\,\langle y\rangle \parallel a(x).x \rightarrow y$.
— Pour $P = \bar{a}\,\langle y\rangle \parallel \bar{a}\,\langle z\rangle \parallel a(x).x$, il y a deux réductions possibles : $P \rightarrow \bar{a}\,\langle y\rangle \parallel z$ et $P \rightarrow \bar{a}\,\langle z\rangle \parallel y$.
— Pour $\Omega = \bar{a}\,\langle a(x).(x \parallel \bar{a}\,\langle x\rangle)\rangle \parallel a(x).(x \parallel \bar{a}\,\langle x\rangle)$, on a : $\Omega \rightarrow \Omega$.

1. La preuve complète est disponible à l'adresse :
`http://people.rennes.inria.fr/Alan.Schmitt/research/AbstractMachineHOCoreProofs.pdf`

2.1.3 Indices de de Bruijn

La syntaxe initiale est sujette au problème de l'α-renommage, c'est-à-dire que, quitte à renommer entièrement les variables, un même processus peut s'écrire de plusieurs façons différentes. Ainsi, $a(x).x$ et $a(y).y$ désignent deux réceptions équivalentes, mais qui ne sont pas écrites de la même façon.

Pour y remédier, on utilise donc les variables de de Bruijn, c'est-à-dire qu'une variable ne sera plus nommée, mais sera représentée par le nombre de réceptions intermédiaires de la forme $a(x)$ se trouvant entre sa déclaration et sa position. Formellement, la syntaxe devient

$$P ::= \quad P \parallel P \mid \bar{a}\langle P\rangle \mid a.P \mid i \mid \star$$

avec i un indice de de Bruijn, c'est-à-dire un entier.

Exemples :

- $\star$ s'écrit de la même façon dans les deux syntaxes.
- $a(x).x$ se réécrit en $a.0$ car il n'y a aucun liant entre la position du x et sa déclaration dans $a(x)$.
- $a(x).b(y).x$ se réécrit $a.b.1$ car il y a un seul liant, $b(y)$, entre x et sa déclaration dans $a(x)$.
- $\Omega = \bar{a}\langle a(x).(x \parallel \bar{a}\langle x\rangle)\rangle \parallel a(x).(x \parallel \bar{a}\langle x\rangle)$ se réécrit $\bar{a}\langle a.(0 \parallel \bar{a}\langle 0\rangle)\rangle \parallel a.(0 \parallel \bar{a}\langle 0\rangle)$.

Un dernier problème se pose, pour traiter le cas des variables libres, c'est-à-dire des variables qui n'ont jamais été déclarées par un liant. Pour simplifier le opérations sur la machine, j'ai décidé d'utiliser la convention localement anonyme [3], consistant à dire qu'une variable liée est représentée par son indice de de Bruijn, tandis qu'une variable libre est nommée. La syntaxe que j'ai utilisée est donc finalement

$$P ::= \quad P \parallel P \mid \bar{a}\langle P\rangle \mid a.P \mid i \mid x \mid \star$$

où x désigne une variable libre. Par exemple, $\bar{a}\langle y\rangle \parallel a(x).x$ se réécrit en $\bar{a}\langle y\rangle \parallel a.0$.

2.2 Propriétés du calcul

HOcore est un calcul non-déterministe car un terme peut avoir plusieurs réductions différentes possibles en même temps. Il n'est pas confluent, c'est-à-dire que si un processus P admet deux réductions distinctes ayant pour résultat P_1 et P_2, il est possible que P_1 et P_2 ne puissent pas se réduire en un nombre quelconque d'étapes vers un même processus Q. C'est par exemple le cas pour $P = \bar{a}\langle y\rangle \parallel \bar{a}\langle z\rangle \parallel a(x).x$ avec $P_1 = \bar{a}\langle y\rangle \parallel z$ et $P_2 = \bar{a}\langle z\rangle \parallel y$. Cette propriété est l'équivalent de la notion de course critique dans un programme concurrent : deux exécutions d'un même processus peuvent ne pas avoir la même issue en fonction de l'entrelacement des sous-processus mis en parallèle.

La différence principale entre HOcore et HOπ dont il est issu est l'absence d'opérateur de restriction de nom. En effet, de nombreux calculs de processus disposent d'un moyen plus ou moins direct de créer des nouveaux nom de canaux ou de variables, ce qui permet d'effectuer en parallèle un nombre arbitraire de tâches. Les opérations de ce genre sont cependant impossibles en HOcore. En particulier, le nombre de canaux différents qui peuvent exister au cours de l'exécution d'un processus est toujours majoré par le nombre de canaux initiaux.

Cette contrainte fait que HOcore dispose d'une expressivité moindre que beaucoup d'autres calculs. Il reste cependant Turing-complet [9].

3 Machine abstraite

3.1 De la machine de Krivine à HOcore

3.1.1 Machine de Krivine

Une machine abstraite classique est la machine de Krivine, qui sert à modéliser le λ-calcul. Je m'en suis inspiré pour construire celle pour HOcore. Pour rappel, la syntaxe du λ-calcul en utilisant des indices de de Bruijn est la suivante :

$$u ::= \lambda.u \mid u\,u \mid i$$

La machine de Krivine est alors définie comme un triplet formé d'un λ-terme, d'une pile π et d'un environnement e. La pile et l'environnement sont des listes dont les éléments sont définis par récurrence comme étant des paires formée d'un λ-terme et d'un environnement. La pile contient les prochains termes à évaluer et l'environnement, les arguments successifs des λ-abstractions.

La machine possède plusieurs réductions possibles, en fonction de la forme du λ-terme :

$$\langle u\,v, \pi, e\rangle \to \langle u, (v, e) :: \pi, e\rangle \qquad \langle \lambda.u, p :: \pi, e\rangle \to \langle u, \pi, p :: e\rangle$$
$$\langle i+1, \pi, p :: e\rangle \to \langle i, \pi, e\rangle \qquad \big\langle 0, \pi, (u, e') :: e\big\rangle \to \big\langle u, \pi, e'\big\rangle$$

La machine de Krivine est déterministe puisqu'à un λ-terme ne peut correspondre qu'une seule réduction. Cela vient du fait qu'elle simule une stratégie d'évaluation particulière, l'appel par nom (ou stratégie d'évaluation externe gauche) : dans un terme de la forme $(\lambda.u)v$, c'est d'abord u qui est évalué puis v. Cela se justifie car le λ-calcul est confluent, et si un terme dispose d'une forme normale (c'est-à-dire s'il peut se réduire en un terme qui ne se réduit plus ensuite), alors cette stratégie d'évaluation mène à cette unique forme normale.

3.1.2 Machine abstraite pour HOcore

Contrairement au λ-calcul, HOcore n'est pas un calcul confluent donc j'ai choisi de ne pas implémenter de stratégie d'évaluation spécifique dans la machine abstraite. Par

conséquent, celle-ci est non-déterministe, puisqu'elle doit pouvoir simuler n'importe quelle réduction du processus HOcore initial.

La machine est inspirée de celle de Krivine par l'utilisation d'environnements e, récursivement définis comme des listes de paires formées d'un processus HOcore et d'un environnement. Le i-ème élément d'un environnement e, noté $e\,[i]$ est donc de la forme (P, e). La syntaxe de la machine est :

$$\mathbf{M} ::= \mathbf{M} + \mathbf{M} \;\big|\; (\bar{a}\,\langle P\rangle\,, e) \;\big|\; (a.P, e) \;\big|\; x \;\big|\; \star$$

— $+$ est le pendant de $\|$ pour les machines. Il est de même commutatif, associatif et admet $\star$ comme élément neutre ;
— $(\bar{a}\,\langle P\rangle\,, e)$ et $(a.P, e)$ sont des processus annotés d'un environnement, respectivement une émission et une réception ;
— x désigne une variable libre ;
— $\star$ est la machine vide, qui ne peut réaliser aucune action.

Pour pouvoir définir la transition disponible pour la machine, on utilise une traduction des processus annotés (P, e) en machine, notée $[\![(P, e)]\!]_{\mathcal{M}}$ et récursivement définie comme

$$\begin{aligned}
[\![(P \parallel Q, e)]\!]_{\mathcal{M}} &= [\![(P, e)]\!]_{\mathcal{M}} + [\![(Q, e)]\!]_{\mathcal{M}} & [\![(\star, e)]\!]_{\mathcal{M}} &= \star \\
[\![(\bar{a}\,\langle P\rangle\,, e)]\!]_{\mathcal{M}} &= (\bar{a}\,\langle P\rangle\,, e) & [\![(x, e)]\!]_{\mathcal{M}} &= x \\
[\![(a.P, e)]\!]_{\mathcal{M}} &= (a.P, e) & [\![(i, e)]\!]_{\mathcal{M}} &= [\![e\,[i]]\!]_{\mathcal{M}}
\end{aligned}$$

La transition d'une machine est alors

$$(\bar{a}\,\langle Q\rangle\,, e) + (a.R, f) + \mathbf{M} \;\to\; [\![(R, (Q, e) :: f)]\!]_{\mathcal{M}} + \mathbf{M}$$

Ceci signifie que le message émis est simplement mis dans l'environnement de la réception, sauf dans le cas où la réception est une variable liée, auquel cas sa valeur est évaluée.

Cette transition n'est pas tout à fait élémentaire. En effet, l'évaluation récursive de $[\![e\,[i]]\!]_{\mathcal{M}}$ dans le cas d'une variable n'est pas immédiate : on aurait pu définir à la place $[\![(i, e)]\!]_{\mathcal{M}} = (i, e)$ où (i, e) serait une nouvelle construction possible pour la machine, et rajouter la transition spontanée $(i, e) + \mathbf{M} \;\to\; [\![e\,[i]]\!]_{\mathcal{M}} + \mathbf{M}$. Les réductions possibles de cette alternative sont plus élémentaires, et on peut démontrer que toute transition de la première forme correspond à une série de réductions de la deuxième.

L'intérêt de la transition retenue par rapport à cette alternative est qu'une réduction du processus modélisé correspond exactement à une transition de la machine correspondante, ce qui simplifie grandement les preuves faites sur la machine.

Exemples

— $\star$ n'admet pas de transition.
— $(\bar{a}\,\langle y\rangle\,, [\,]) + (a.0, [\,]) \;\to\; [\![(0, (y, [\,]) :: [\,])]\!]_{\mathcal{M}} = y$.
— $(\bar{a}\,\langle 0\rangle\,, [(x, [\,])]) + \big(a.(1 \parallel \bar{b}\,\langle\star\rangle), [(y, [\,])]\big) \;\to\; y + \big(\bar{b}\,\langle\star\rangle\,, [(x, [\,])\,; (y, [\,])]\big)$.

3.1.3 Bonne formation

L'utilisation de la convention localement anonyme permet l'existence de terme qui sont mal formés, si une variable de de Bruijn n'est pas liée. Par exemple, dans le processus $\bar{a}\,\langle 0\rangle$, la variable 0 est en réalité libre, et devrait donc être nommée et non pas écrite comme un indice de de Bruijn.

La correction de la machine est donc soumise à la contrainte que tous les indices de de Bruijn du processus initial se réfèrent à des liants valides, propriété dite de bonne formation. Il est intuitif, et on vérifie bien, que si $P \to Q$ et P est bien formé, alors Q l'est aussi.

Une condition équivalente a été développée sur la machine de façon à ce que si $\mathbf{M} \to \mathbf{N}$ et $\mathbf{M}$ est bien formée, alors $\mathbf{N}$ aussi.

3.2 Correction et complétude

3.2.1 Traduction

L'essentiel de la preuve consiste à montrer qu'il existe une équivalence entre les transitions de la machine et les réductions des processus. Pour montrer ceci, il est nécessaire de disposer d'une traduction entre les machines et les processus.

Un processus P est simplement traduit en une machine abstraite qui le représente et notée $[\![P]\!]_{\mathcal{M}}$, définie par $[\![P]\!]_{\mathcal{M}} = [\![(P, [\,])]\!]_{\mathcal{M}}$.

La traduction inverse est plus complexe. Elle consiste à remplacer toutes les variables liées à une réception qui a déjà reçu un message par le message lui-même, que l'on peut trouver dans un environnement. Pour définir formellement cette traduction inverse, on a besoin d'utiliser un paramètre annexe, la profondeur, qui sert à compter le nombre de liants sous lequel le terme à traduire est placé. Cette nécessité vient de l'utilisation des variables de de Bruijn. On définit donc la traduction partielle d'un processus annoté (P, e) avec profondeur d, notée $[\![(P, e)]\!]^d_{\mathcal{P}}$, définie par

$$
\begin{array}{ll}
[\![(P \parallel Q, e)]\!]^d_{\mathcal{P}} = [\![(P, e)]\!]^d_{\mathcal{P}} \parallel [\![(Q, e)]\!]^d_{\mathcal{P}} & [\![(\star, e)]\!]^d_{\mathcal{P}} = \star \\
[\![(\bar{a}\,\langle P\rangle\,, e)]\!]^d_{\mathcal{P}} = \bar{a}\left\langle [\![(P, e)]\!]^d_{\mathcal{P}} \right\rangle & [\![(x, e)]\!]^d_{\mathcal{P}} = x \\
[\![(a.P, e)]\!]^d_{\mathcal{P}} = a.\left([\![(P, e)]\!]^{d+1}_{\mathcal{P}}\right) & [\![(i, e)]\!]^d_{\mathcal{P}} = \begin{cases} i & \text{si } i < d \\ [\![e\,[i-d]]\!]^0_{\mathcal{P}} & \text{sinon} \end{cases}
\end{array}
$$

On peut alors définir la traduction inverse d'une machine, notée $[\![\mathbf{M}]\!]_{\mathcal{P}}$, simplement par

$$
[\![\mathbf{M} + \mathbf{N}]\!]_{\mathcal{P}} = [\![\mathbf{M}]\!]_{\mathcal{P}} \parallel [\![\mathbf{N}]\!]_{\mathcal{P}} \qquad [\![(P, e)]\!]_{\mathcal{P}} = [\![(P, e)]\!]^0_{\mathcal{P}} \qquad [\![x]\!]_{\mathcal{P}} = x \qquad [\![\star]\!]_{\mathcal{P}} = \star
$$

Une première propriété de ces traductions dont on dispose est que pour tout processus P bien formé, on a $[\![[\![P]\!]_{\mathcal{M}}]\!]_{\mathcal{P}} = P$, ce qui est attendu et donne un sens à la traduction.

L'autre égalité, $[\![[\![\mathbf{M}]\!]_{\mathcal{P}}]\!]_{\mathcal{M}} = \mathbf{M}$, est en revanche fausse dans le cas général d'une machine $\mathbf{M}$ bien formée. En effet, tous les environnements apparaissant dans la traduction directe $[\![P]\!]_{\mathcal{M}}$ sont vides, ce qui n'est pas nécessairement le cas chez la machine initiale $\mathbf{M}$.

Ce phénomène est une conséquence du fait qu'on peut obtenir un même processus HOcore à travers des suites de réductions différentes en partant de termes différents. En effet, la machine garde dans ses environnements la trace des différentes communications qui ont eu lieu, c'est-à-dire des réductions successives qui se sont produites.

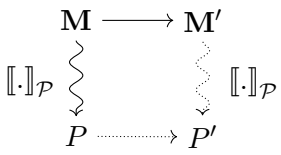
3.2.2 Propriétés requises

Deux propriétés sont requises pour la machine : la correction et la complétude.

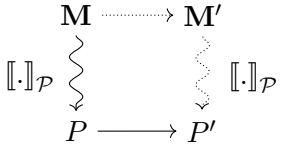
La correction de la machine signifie que toute suite de transitions partant d'une machine bien formée correspond à une suite de réductions du processus en lequel elle est traduit.

La complétude est la propriété duale, consistant à dire que toute suite de réductions d'un processus bien formé correspond à une suite de transitions de la machine en lequel il est traduit.

Ces deux propriétés se visualisent plus facilement à l'aide de diagrammes :



Correction



Complétude

Sur ces diagrammes, les flèches pleines sont les conditions et celles en pointillées, les conséquences. Les flèches horizontales désignent généralement un nombre quelconque de réductions, mais ici, du fait de la transition retenue pour les machines, une transition au sein des machines correspond exactement à une réduction pour les processus.

3.2.3 Résumé de la preuve

La preuve de la correction et de la complétude de la machine abstraite passe par de nombreuses étapes intermédiaires, nécessaires notamment pour la manipulation correcte des indices de de Bruijn. Elle consiste en trois grandes étapes :

1. Équivalence de machines

   Raisonner directement sur une machine abstraite quelconque pour établir la correction est délicat, car les environnements peuvent être de n'importe quelle forme. On réduit donc le problème à l'étude de certaines machines particulières.

   Deux machines $\mathbf{M}$ et $\mathbf{N}$ sont dites équivalentes lorsque $[\![\mathbf{M}]\!]_{\mathcal{P}} = [\![\mathbf{N}]\!]_{\mathcal{P}}$, ce qu'on note $\mathbf{M} \sim \mathbf{N}$. Le représentant standard de la classe d'équivalence de $\mathbf{M}$, que l'on note $\widehat{\mathbf{M}}$, est défini comme $\widehat{\mathbf{M}} = [\![[\![\mathbf{M}]\!]_{\mathcal{P}}]\!]_{\mathcal{M}}$.

   C'est ce $\widehat{\mathbf{M}}$ qui va servir à établir la correction de la machine, car tous ses environnements sont vides, ce qui le rend plus manipulable. Parmi les premières propriétés de $\widehat{\mathbf{M}}$ on trouve la stabilité par $+$, c'est-à-dire $\widehat{\mathbf{M} + \mathbf{N}} = \widehat{\mathbf{M}} + \widehat{\mathbf{N}}$ et la bonne définition,

soit $\widehat{\widehat{\mathbf{M}}} = \widehat{\mathbf{M}}$. Cette dernière propriété est d'ailleurs un simple corollaire du fait que $[\![[\![P]\!]_{\mathcal{M}}]\!]_{\mathcal{P}} = P$, ce que l'on a déjà vu.

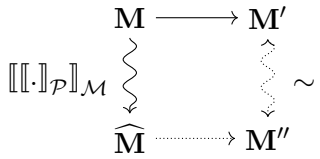
2. Lemme principal

   Le lemme principal à établir correspond à montrer la correction de la machine lors d'une unique transition, en partant d'un représentant standard.

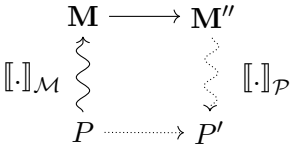
   Formellement, on montre que si $\mathbf{M} \to \mathbf{M}'$, alors $\widehat{\mathbf{M}} \to \mathbf{M}''$ avec $\mathbf{M}' \sim \mathbf{M}''$.

   Pour conclure la preuve de la correction, il faut enfin prouver que si $[\![P]\!]_{\mathcal{M}} \to \mathbf{M}''$ alors $P \to P'$ avec $[\![\mathbf{M}'']\!]_{\mathcal{P}} = P'$.

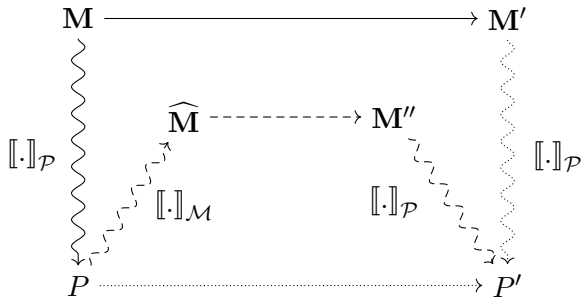
   Ces deux théorèmes sont décrits par les deux diagrammes :



et



   Mis bout à bout, on obtient donc la forme générale de la preuve :



3. De la correction à la complétude

   La complétude se déduit d'une propriété de correction un peu plus forte que celle que l'on vient de voir. On peut en effet étiqueter les transitions d'un processus : si P se réduit en P' alors P est de la forme $Q \parallel (a.R) \parallel \bar{a}\langle S \rangle$. On étiquette alors sa transition par le couple émission / réception : $P \xrightarrow{(\bar{a}\langle S \rangle, a.R)} P'$.

   On peut de même étiqueter les transitions au sein des machines : pour $\mathbf{M} = (\bar{a}\langle Q \rangle, e) + (a.R, f) + \mathbf{N} \to \mathbf{M}'$, on note $\mathbf{M} \xrightarrow{\left([\![(\bar{a}\langle Q \rangle, e)]\!]_{\mathcal{P}}, [\![(a.R,f)]\!]_{\mathcal{P}}\right)} \mathbf{M}'$.

   On a alors la nouvelle propriété de correction : Si $\mathbf{M} \xrightarrow{t} \mathbf{M}'$ alors $[\![\mathbf{M}]\!]_{\mathcal{P}} \xrightarrow{t} P'$ avec $[\![\mathbf{M}']\!]_{\mathcal{P}} = P'$.

   On peut alors prouver la complétude en remarquant que si $P \xrightarrow{t} P'$, alors, du fait de la structure de P, pour tout $\mathbf{M}$ tel que $[\![\mathbf{M}]\!]_{\mathcal{P}} = P$, $\mathbf{M}$ admet une transition étiquetée

par t vers un $\mathbf{M}'$. Or, par correction, P admet alors une transition étiquetée par t vers un P'' et $[\![\mathbf{M}']\!]_{\mathcal{P}} = P''$. Enfin, comme $P \xrightarrow{t} P'$ et $P \xrightarrow{t} P''$, on obtient $P' = P''$ donc $[\![\mathbf{M}']\!]_{\mathcal{P}} = P'$.

4 Implémentation

L'implémentation de la machine abstraite permet de visualiser l'exécution de la machine sur un processus donné. Étant donné sa nature non-déterministe, j'ai implémenté plusieurs versions de la machine abstraite, nécessitant ou non un dialogue avec l'utilisateur pour le choix des réductions.

4.1 En OCaml

4.1.1 Interpréteur

La première étape de la réalisation d'une machine abstraite pour HOcore est la compréhension du calcul lui-même. J'ai donc commencé par créer un interpréteur pour HOcore, écrit en OCaml.

La première version, très naïve, modélise les processus comme des listes d'atomes, les atomes pouvant être des émissions, des réceptions ou des variables. Cette version convient pour une utilisation pas-à-pas, où la communication à choisir entre une émission et une réception est demandée à l'utilisateur à chaque réduction.

Elle n'est cependant pas suffisamment efficace pour une utilisation cherchant à explorer toutes les réductions possibles d'un processus initial donné. Dans ce cadre, j'ai amélioré l'interpréteur en raffinant le modèle de HOcore choisi, de façon à ce que l'exploration d'un terme possédant un nombre non borné de réductions prenne un temps raisonnable, même pour plusieurs milliers de réductions au maximum.

1. Regroupement par canal

   La première amélioration consiste à regrouper les émissions et les réceptions en fonction du canal sur lequel elles ont lieu. En utilisant des variables nommées, on obtient donc la structure suivante pour les processus :

   ```
   module SMap  = Map.Make(String)
   type process = Proc of ((send list * recv list) SMap.t) * var list
   and send = process
   and recv = var * process
   and var  = string
   ```

   L'intérêt de ce regroupement est de pouvoir identifier facilement les communications possibles, et de ne pas prendre en compte les canaux sur lesquels il n'y a pas de communication.

2. Compilation vers une fonction

   Une seconde amélioration possible consiste à transformer les réceptions en fonctions de la forme `type recv = process -> process`, les autres types étant conservés par ailleurs.

   En effet, la communication se fait alors très simplement en appliquant la réception (qui est une fonction) à l'émission (qui est un processus). Cependant, la création de telles fonctions ne peut pas se faire au sein du même programme qui fait l'analyse syntaxique du processus HOcore entré, car les différentes réceptions sont mutuellement récursives. Par exemple, la réception $a(x).b(y).x$ devra être transformé à terme en un objet dont la structure ressemblera à `fun x -> fun y -> x` qui ne peut pas être créée lors d'un parcours de l'arbre de syntaxe abstrait du processus.

   La solution consiste donc à compiler le processus HOcore entré en un code OCaml contenant un objet de type `process`, qui sera ensuite compilé puis exécuté pour rechercher toutes ses réductions. On profite alors aussi de l'efficacité du compilateur pour que les fonctions constituant les réceptions soient optimisées.

   L'implémentation de cette méthode n'a cependant pas montré une amélioration significative des performances de l'interpréteur, sans doute à cause de la difficulté pour le compilateur d'optimiser les fonctions à travers l'appel au module `SMap`.

3. Structure de multi-ensemble

   Une dernière amélioration importante de l'interpréteur consiste à ne plus utiliser des listes, mais des multi-ensembles pour identifier plus facilement les processus identiques. L'intérêt de cet ajout réside dans l'exploration du graphe des réductions possibles, où l'on évite alors d'effectuer de nombreuses fois la même réduction. Cela requiert cependant l'utilisation de modules mutuellement récursifs pour pouvoir continuer à utiliser le foncteur `Map.Make`.

   Les performances sont effectivement nettement accrues par cet ajout.

4.1.2 Machine abstraite

L'implémentation de la machine abstraite se fait par-dessus celle de l'interpréteur. Il suffit d'y ajouter les types suivants :

```
type 'a annot = 'a * env
and env = Env of process annot list
type machine = ((send annot) list * (recv annot) list) SMap.t * var list
```

Les transitions de la machine se font alors comme définies, par insertion de l'émission dans l'environnement de la réception et évaluation du tout.

4.2 En HOcore

4.2.1 Motivations

Afin d'avoir une autre perspective sur le calcul, j'ai décidé d'implémenter la machine abstraite en HOcore.

L'intérêt d'une telle démarche est avant tout de mieux comprendre ce calcul. Plus spécifiquement, je souhaitais voir comment manipuler des éléments de programmation élémentaires comme les boucles ou les nombres dans HOcore, qui est Turing-complet et devait donc pouvoir les exprimer. Cela permet par ailleurs de donner un aperçu de l'expressivité du langage, et en particulier de la façon d'échapper à la contrainte de n'utiliser qu'un nombre fixé de canaux, ainsi que la gestion du non-déterminisme.

Le programme à implémenter, en l'occurrence un interpréteur pour la machine abstraite, est suffisamment compliqué pour permettre d'explorer en détail ces aspects, mais reste réalisable dans le cadre très contraint de la programmation en HOcore. Enfin, cette implémentation m'a obligé à beaucoup simplifier la machine par rapport à ce qu'elle était à l'origine, ce qui a permis en retour de clarifier les preuves et rendre la machine sans doute mieux adaptable à HOπ.

4.2.2 Éléments de programmation

Voici un aperçu des différents éléments de programmation en HOcore que j'ai développés, et qui m'ont été très utiles pour cette implémentation.

1. Entiers naturels

   Les entiers naturels sont représentés en HOcore d'une façon assez similaire à leur représentation en λ-calcul par les entiers de Church. En effet, on représente un nombre n comme une réception sur le canal `repeat`, qui attend un processus et le répète n fois, puis envoie le signal `zero<*>` pour signaler la fin de son exécution. On définit ainsi, dans la syntaxe du préprocesseur C :

   ```
   #define ZERO       (repeat.zero<*>)
   #define SUCC(n)    (repeat(P).(P || ACK_rep.(repeat<P> || n)))
   ```

   La convention choisie ici est que le processus `P` doit envoyer à la fin de son exécution un signal sur le canal `ACK_rep`.

   Parmi les opérations élémentaires, on peut ainsi définir l'addition par exemple :

   ```
   #define ADD(n,m)   temp<n> || m || zero.temp(x).RET_add<x> ||       \
                      repeat< temp(x).(temp<SUCC(x)> || ACK_rep<*>) >
   ```

   Le résultat de l'opération peut être obtenu sur le canal `RET_add`.

2. Listes

   Les listes chaînées ont une représentation intuitive, la tête de la liste étant dans le canal `hd` et la queue, dans le canal `tl`. On garde dans `len` la taille de la liste, afin de pouvoir la parcourir plus simplement :

```
#define NIL        hd<*> || tl<*> || len<ZERO>
#define push(x,l)  (l || hd.tl.len(n).(RET_push<hd<x> || tl<l> ||   \
                   len<SUCC(n)>>))
```

L'accès au `i`^{e} élément de la liste `l` se fait alors par :

```
#define nth(i,l)  l || i || repeat<hd.len.tl(s).(s || ACK_rep<*>)>  \
               || (zero.tl.len.hd(x).RET_nth<x>)
```

3. Booléens

   Les booléens sont représentés de la façon suivante :

   ```
   #define TRUE      (false.true(x).x)
   #define FALSE     (true.false(x).x)
   ```

   Une conditionnelle de la forme "*if* `e` *then* `x` *else* `y`" où `e` est un booléen s'écrit alors :

   ```
   e || true<x> || false<y>
   ```

   On peut remarquer que cette structure est en fait très adaptée à un raisonnement par cas avec un nombre quelconque de cas, en remplaçant `true` et `false` par `case_1`, `case_2`, ..., `case_n` avec un pseudo-booléen de la forme `(case_1. ... .case_{i-1}.case_{i+1}. ... .case_n.case_i(x).x)`.

4. Boucles

   Les boucles enfin s'écrivent par analogie avec le terme Ω vu plus haut. Une boucle de la forme "*while* `e` *do* `x`" où `x` se termine par une émission sur `ACK_loop` s'écrit :

   ```
   INIT_loop< loop(o).(e || false<ACK_endloop<*>>
                        || true< x || ACK_loop.(o || loop<o>) >)
            > ||
   INIT_loop(o).(o || loop<o>)
   ```

   Cette boucle se terminera alors par une émission sur `ACK_endloop`.

5 Développements possibles

Ce travail pourrait être étendu de plusieurs façons.

Une première consisterait en une formalisation de la preuve, en Coq par exemple. J'ai travaillé la preuve de façon à ce qu'elle soit le plus détaillé possible, ce qui est la première étape de cette formalisation

Par ailleurs, une simplification possible de la machine consisterait en une division de la transition en sous-transitions plus élémentaires, comme expliqué en 3.1.2. Les preuves de correction et complétude avec cette alternative peuvent se déduire de celles faites avec la transition retenue, donc l'essentiel du travail est déjà fait. Cela rendrait par ailleurs la machine plus flexible, et peut-être plus adaptée pour représenter des processus en cours d'exécution.

Sur un autre plan, il serait intéressant de définir une notion de bisimulation pour la machine abstraite qui corresponde à celle sur HOcore. Les bisimulations sont des relations d'équivalences adaptées à la comparaison de processus distribués [11]. HOcore dispose de plusieurs propriétés très fortes par rapport aux bisimulations, notamment la décidabilité de celles-ci et leur équivalence avec la congruence barbue, une autre relation de comparaison importante [5]. La conception d'une notion de bisimulation simple sur la machine abstraite qui corresponde à celle des processus n'est cependant pas immédiate à cause des environnements de la machine qui n'ont pas de contrepartie dans le processus HOcore.

6 Conclusion

Pour conclure, ce stage m'a permis de découvrir de l'intérieur le milieu de la recherche théorique, ce qui a constitué une expérience très intéressante. J'ai ainsi eu l'occasion de concevoir une machine abstraite pour un calcul de processus d'ordre supérieur, chose qui n'apparaît quasiment nulle part dans la littérature en tant que telle, ce qui m'a donc donné un aperçu assez significatif des difficultés mais aussi de l'intérêt qu'il y a dans l'exploration d'une branche encore peu exploitée de la recherche.

Ce stage fut aussi pour moi l'occasion de découvrir la programmation concurrente, même si le cadre très restreint de HOcore est loin d'être suffisant pour en appréhender toutes ses spécificités. La programmation en HOcore reste une expérience tout à fait instructive et m'a permis de mieux comprendre en retour certaines constructions utilisées en λ-calcul.

Enfin, j'ai beaucoup apprécié de devoir établir entièrement la preuve de la correction de la machine abstraite à partir de rien, faute d'existence d'autres machines abstraites suffisamment proches sur lesquels de telles preuves aient été faites. Cela m'a fait comprendre comment prévoir le plan général d'une preuve, et notamment l'intérêt d'adapter les définitions intermédiaires aux lemmes correspondants, ce qui permet de clarifier parfois considérablement le cheminement de la démonstration.

La suite logique de ce travail consiste désormais en une généralisation de la machine abstraite à HOπ, en ajoutant les restrictions de nom et en adaptant la machine de façon adéquate. Cela pourrait encore s'étendre en ajoutant une notion de localité afin de simuler la distribution physique des opérations au sein d'un réseau réel.

Références

[1] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1) :217 – 248, 1992.

[2] Philippe Bidinger and Adriana Compagnoni. Pict correctness revisited. *Theor. Comput. Sci.*, 410(2-3) :114–127, February 2009.

[3] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3) :363–408, October 2012.

[4] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7) :739 – 751, 2000.

[5] Martín Escarrá, Petar Maksimović, and Alan Schmitt. HOCore in Coq. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, Le Val d'Ajol, France, January 2015.

[6] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM.

[7] Cédric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier Rémy. *A calculus of mobile agents*, pages 406–421. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[8] Florence Germain, Marc Lacoste, and Jean-Bernard Stefani. An abstract machine for a higher-order distributed process calculus. *Electronic Notes in Theoretical Computer Science*, 66(3) :145 – 169, 2002. F-WAN, Foundations of Wide Area Network Computing (ICALP 2002 Satellite Workshop).

[9] Ivan Lanese, Jorge A. Peréz, Davide Sangiorgi, and Alan Schmitt. On the Expressiveness and Decidability of Higher-Order Process Calculi. In *23rd Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*, Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS 2008), pages 145–155, Pittsburgh, Pennsylvania, United States, June 2008.

[10] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[11] Damien Pous. Using bisimulation proof techniques for the analysis of distributed abstract machines, 2008.

[12] Davide Sangiorgi. *Expressing mobility in process algebras : first-order and higher-order paradigms*. 1993.

[13] Davide Sangiorgi and Andrea Valente. A distributed abstract machine for safe ambients. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, ICALP '01, pages 408–420, London, UK, UK, 2001. Springer-Verlag.