février 1999 – Journées Francophones des Langages Applicatifs – JFLA99

Objets Concurrents dans un π-calcul Applicatif

S. Dal-Zilio

INRIA Sophia-Antipolis,
BP 93, 2004 route des Lucioles.
F-06902 Sophia Antipolis cedex
Silvano.Dal_Zilio@sophia.inria.fr

Résumé

Nous présentons un modèle des objets concurrents basé sur le calcul bleu ($\pi^\star$), qui est une variante applicative et fortement typée du π-calcul asynchrone. L'étude de l'expressivité et du typage de ce modèle d'objets est basée sur la définition d'un calcul « basé objets ». Ce calcul, introduit ici, est obtenu par une interprétation directe dans le calcul bleu. À titre d'exemple, nous proposons une interprétation directe et adéquate du calcul d'objets d'Abadi et Cardelli : $\mathbf{Ob}_{1<:}$, qui préserve le sous-typage.

1. Introduction

Dans [21], Milner remarque que la relation de réduction du π-calcul (π) est basée sur le *paradigme objet* dans le sens où « ce qui est transmis et lié n'est jamais l'objet lui-même, mais plutôt le moyen d'*accéder* à l'objet ». Le lien entre π-calcul et langage orienté objet est fort : les processus sont les objets (ainsi que leurs états !) et les canaux de communication sont les références utilisées pour accéder/nommer les objets. Mais le concept de programmation orienté objets est plus complexe que la simple notion de « calcul par références ». Il ne peut se comprendre sans les notions *d'encapsulation*, de *liaison dynamique* et de *sous-typage*. Le but de cet article est de fournir un modèle des objets concurrents basé sur leur codage dans un calcul de processus. À cette fin, nous codons dans le calcul bleu (augmenté avec des enregistrements) [6], les opérateurs d'un calcul d'objets que nous introduisons ici et nous étudions les propriétés de ce codage. Pour le typage des « processus bleus », nous utiliserons un système du premier ordre avec un opérateur de récursion et le sous-typage sur les enregistrements.

L'expressivité du modèle d'objets proposé est démontrée par l'interprétation de $\mathbf{Ob_{1<:}}$, un calcul d'objets fonctionnel typé d'Abadi et Cardelli [2], qui préserve le sous-typage. Ce résultat était prévisible puisque D. Sangiorgi [24] a déjà fournit un codage de $\mathbf{Ob_{1<:}}$ dans le π-calcul. Mais le codage et le système de type utilisé ici sont très différents. En particulier, dans notre codage, une modification de méthode[1] ne crée pas un nouvel objet, mais est plus naturellement modélisée comme un changement d'état. De plus, la preuve de correction opérationnelle du codage ne repose pas sur l'utilisation du système de type et notre codage est « direct » (c.-à-d. ne fait pas appel à des transformations du type *continuation passing style*). Une autre différence est que le modèle d'objets étudié est impératif puisqu'un objet peut être cloné. Ceci n'est pas surprenant car, bien que $\pi^\star$ soit un calcul applicatif, dans le sens où les fonctions sont directement incluse dans la syntaxe du calcul, il est possible d'y coder des constructions impératives. Un autre intérêt de cet article, est qu'il est le premier à introduire un calcul d'objets basé sur les références et le nommage des objets, alors que les calculs de [2, 13] utilisent la substitution de code. Dans ces calculs, par exemple, si o est l'objet[2] : $[\, m = \varsigma(x)e\,]$, la sélection de m dans o produit comme résultat $e\{[\, m = \varsigma(x)e\,]/x\}$ (où $e\{f/x\}$ est la substitution de f à x dans e). On obtient donc un terme dans lequel le code de l'objet est dupliqué au lieu d'être partagé. De plus, notre calcul est fortement typé et il est concurrent.

Après une présentation de la syntaxe du calcul bleu, de son système de type et de sa sémantique opérationnelle, nous introduisons, section 3, un calcul d'objets concurrent. Dans le reste de cette section, nous démontrons que les opérateurs de ce calcul peuvent être dérivés dans $\pi^\star$ et nous donnons leurs types dérivés. Dans la section 4, nous donnons un codage adéquat de $\mathbf{Ob_{1<:}}$ et nous étudions quelques exemples d'équivalence entre objets.

1.1. Liens avec d'autres travaux.

Plusieurs études théoriques traitent de la modélisation des langages de programmation orientés objets dans des langages procéduraux [16, 1]. Mais peu de codages ont réussit à préserver des caractéristiques importantes, telle que le sous-typage par exemple. Dans [3], les auteurs proposent une interprétation compositionelle d'un calcul d'objets avec sous-typage dans un calcul fonctionnel. Mais le calcul cible,

[1] « method update » dans le vocabulaire définit par Abadi et Cardelli

[2] x représente l'objet dans le corps de la méthode m. C'est l'équivalent de la notation *this* dans certains langages de programmation tel que C++

Tab. 1 La Syntaxe du Calcul Bleu : $\pi^\star$

$$
\begin{array}{lll}
a & ::= & x \mid u \\
P & ::= & a \mid (\lambda x)P \mid (P\ a) \mid \mathtt{0} \mid (P \mid P) \mid (\nu u)P \mid \\
 & & \langle u \Leftarrow P \rangle \mid \mathbf{def}\, D\, \mathbf{in}\, P \mid [\,] \mid [\,l = P\,,\ P\,] \mid (P\ \cdot l)
\end{array}
$$

$\mathbf{F}_{<:\mu}$, utilise les types polymorphes du second ordre, alors que notre interprétation utilise un système du premier ordre. Très récemment, R. Viswanathan [26] a proposé une interprétation dans un λ-calcul du premier ordre étendu avec des références et des enregistrements non-extensible. Une autre définition de calcul d'objets typé est donnée dans [13]. Mais aucun de ces calculs ne permet de modéliser plusieurs objets actifs simultanément.

Dans le domaine de la concurrence, Jones [18] et Walker [27] ont utilisé le π-calcul pour coder POOL, un langage parallèle orienté objets et pour prouver la validité de certaines transformations. Mais POOL est non typé et très restreint. Dans [24], Sangiorgi donne la première interprétation du calcul d'objets d'Abadi et Cardelli avec sous-typage dans π. Dans [19], les auteurs s'intéresse au calcul impératif. Il faut aussi noter des travaux récents donnant la définition de calcul d'objets concurrents obtenu en étendant un langage séquentiel avec des « opérateurs concurrents » : [12, 14].

2. Le calcul bleu

Le calcul bleu est une variante du mini π-calcul asynchrone [5] dans lequel les fonctions (ou abstractions pour utiliser la terminologie du λ-calcul) sont directement incluses. Le calcul n'a pas d'opérateur de choix, de matching ou de garde pour les émissions et, alors que π favorise un style de programmation indirect, $\pi^\star$ fourni une meilleure notation pour programmer des comportements concurrents de haut-niveau. Un exemple est le codage des fonctions en π, pour lequel on doit explicitement s'occuper des canaux de communications transportant le résultat.

Les termes de $\pi^\star$ utilisent trois catégories distinctes de noms : les variables : $x, y, z \cdots \in \mathcal{V}$, les références : $u, v, w \cdots \in \mathcal{R}$ et les labels : $k, l, m \cdots \in \mathcal{L}$. Dans la syntaxe, donnée Tab. 1, D (dans $\mathbf{def}\, D\, \mathbf{in}\, P$[3]) est une séquence de définitions mutuellement récursives où les x_i sont deux à deux distincts : $D =_{\text{def}} (x_1 = P_1, \ldots, x_n = P_n)$.

[3] une définition peut être vide et on confond $\mathbf{def}\ \ \mathbf{in}\, P$ avec P

Par soucis de simplicité, nous séparons la description du calcul en trois catégories syntaxique. Dans chaque cas, nous donnons les règles de réductions ($\rightarrow$), d'équivalence structurelle ($\equiv$) et de typage. La description utilise la notion standard de *contextes d'évaluation*. Les contextes sont définis à partir de la syntaxe de $\pi^\star$ en ajoutant la constante : $[.]$. Les contextes d'évaluation, $\mathrm{E}\,[.]$, étant les contextes tel que $[.]$ n'est pas gardé par une abstraction, un enregistrement ou une déclaration :

$$\mathrm{E}\,[.] ::= \ [.] \mid (\mathrm{E}\,[.]\ a) \mid (\mathrm{E}\,[.] \mid P) \mid (\nu u)(\mathrm{E}\,[.]) \mid \mathbf{def}\, D\, \mathbf{in}\, \mathrm{E}\,[.] \mid (\mathrm{E}\,[.]\ \cdot l)$$

Deux règles générales peuvent être données concernant la réduction :

$$\frac{P \rightarrow P' \qquad P \equiv Q}{Q \rightarrow P'}\ \text{(red equiv)}$$

$$\frac{P \rightarrow P' \qquad \mathrm{E}\,[.]\ \text{contexte d'évaluation}}{\mathrm{E}\,[P] \rightarrow \mathrm{E}\,[P']}\ \text{(red context)}$$

En ce qui concerne l'équivalence structurelle, deux processus α-convertible seront considéré égaux. Le système de type utilisé ici est le système à la Curry présenté dans [6] et étendu pour supporter les enregistrements, la récursion et le sous-typage. Nous définissons deux constantes de type, $\bot$ et $\top$, qui sont, respectivement, le plus petit et le plus grand des types :

$$\tau, \sigma, \omega \ldots ::= \bot \mid \top \mid \alpha \mid \mu\alpha.\tau \mid (\tau \rightarrow \tau) \mid [\,] \mid [\,\tau\,,\, l : \tau\,]$$

2.1. Description du fragment fonctionnel de $\pi^\star$

La partie fonctionnelle de $\pi^\star$ est celle définie par les constructeurs :

$$P, Q, R \ldots ::= \ \ldots \mid a \mid (\lambda x)P \mid (P\ a) \mid \mathbf{def}\, D\, \mathbf{in}\, P$$

C'est le « petit » λ-calcul étendu avec l'opérateur de définitions : $\mathbf{def}\, D\, \mathbf{in}\, P$, où petit signifie qu'un processus ne peut être appliqué qu'à un nom et non pas à un terme quelconque. On dit également que le calcul bleu est : *name passing*. Néanmoins on peut obtenir l'équivalent du λ-calcul « d'ordre supérieur » en définissant l'application par :

$$(P\ Q) =_{\text{def}} \mathbf{def}\ x = Q\ \mathbf{in}\ (P\ x) \qquad (x \notin \mathrm{fn(P)} \cup \mathrm{fn(Q)})$$

Il existe une différence avec la présentation originelle du calcul faite dans [6]. Nous remplaçons les définitions « flottantes » : $\langle u\ =\ P\rangle$ (l'équivalent d'une infinité de déclarations en parallèles : $!\langle u \Leftarrow P\rangle$) par une construction mêlant à la fois restriction, réplication et déclaration :

$\mathbf{def}\, x = R\, \mathbf{in}\, P =_{\mathrm{def}} (\nu x)(\langle x = R\rangle \mid P)$.

Nous notons $P\{a/x\}$ la substitution de x par a dans P. L'ensemble des variables définies par D est noté : def(D) et fn(P) représente l'ensemble des noms libres de P. les « lieurs » sont ici l'abstraction : $(\lambda x)P$ et la définition : $\mathbf{def}\, D\, \mathbf{in}\, P$. Nous notons $\tilde{x}$ le tuple $x_1, \ldots, x_n$ et $(\lambda \tilde{x})P$ représente le terme $(\lambda x_1) \ldots (\lambda x_n)P$. L'équivalence structurelle est définie par les axiomes suivant [7] :

$$\begin{array}{rcll}
(\mathbf{def}\, D\, \mathbf{in}\, P) \mid Q & \equiv & \mathbf{def}\, D\, \mathbf{in}\, (P \mid Q) & \mathrm{def(D)} \cap \mathrm{fn(Q)} = \emptyset \\
\mathbf{def}\, D\, \mathbf{in}\, (\mathbf{def}\, D'\, \mathbf{in}\, P) & \equiv & \mathbf{def}\, D, D'\, \mathbf{in}\, P & \mathrm{def(D')} \cap \mathrm{fn(D)} = \emptyset \\
\mathbf{def}\, D\, \mathbf{in}\, ((\nu u)P) & \equiv & (\nu u)(\mathbf{def}\, D\, \mathbf{in}\, P) & u \notin \mathrm{fn(D)} \\
(\mathbf{def}\, D\, \mathbf{in}\, P)\ a & \equiv & \mathbf{def}\, D\, \mathbf{in}\, (P\ a) & a \notin \mathrm{def(D)} \\
((\nu u)P)\ a & \equiv & (\nu u)(P\ a) & a \neq u
\end{array}$$

Pour le fragment fonctionnel du calcul, la réduction est définie par deux règles. Les règles de typage sont données dans le tableau Types 1.

$$((\lambda x)P\ a) \rightarrow P\{a/x\} \text{ (red beta)}$$

$$\frac{D =_{\mathrm{def}} D_1, x = R, D_2 \qquad \{x\} \cup \mathrm{fn(R)} \text{ non liés dans } \mathrm{E}\,[.]}{\mathbf{def}\, D\, \mathbf{in}\, \mathrm{E}\,[x\ a_1 \ldots a_n] \rightarrow \mathbf{def}\, D\, \mathbf{in}\, \mathrm{E}\,[R\ a_1 \ldots a_n]} \text{ (red def)}$$

Types 1 Typage du fragment λ-calcul

$$\Gamma, x : \tau \vdash x : \tau \text{ (type ax)} \qquad \frac{\Gamma, x : \tau \vdash P : \tau'}{\Gamma \vdash (\lambda x)P : \tau \rightarrow \tau'} \text{ (type abs)}$$

$$\frac{\Gamma \vdash P : \tau \rightarrow \tau' \qquad \Gamma \vdash a : \tau}{\Gamma \vdash (P\ a) : \tau'} \text{ (type app)}$$

$$\frac{\Gamma, \{x_i : \tau_i{}^{1 \leq i \leq n}\} \vdash R_i : \tau_i \qquad \Gamma, \{x_i : \tau_i{}^{1 \leq i \leq n}\} \vdash P : \omega}{\Gamma \vdash \mathbf{def}\, x_1 = R_1, \ldots, x_n = R_n\, \mathbf{in}\, P : \omega} \text{ (type def)}$$

2.2. Description du fragment π-calcul de $\pi^\star$

Nous considérons ici les opérateurs directement empruntés au π-calcul :

$$P, Q, R \ldots ::= \ldots \;\big|\; \mathtt{0} \;\big|\; \langle u \Leftarrow P\rangle \;\big|\; (P \mid P) \;\big|\; (\nu u)P$$

Le seul constructeur original est la déclaration : $\langle u \Leftarrow P\rangle$, qui peut s'interpréter comme une ressource, localisée en u, accessible seulement une fois[4]. Ce constructeur est nécessaire à la modélisation de processus ayant un état mutable.

Il faut noter que la différence faîte entre variables et références implique qu'il n'est pas possible d'abstraire une référence. Ainsi $(\lambda u)\langle u \Leftarrow P\rangle$, par exemple, n'est pas un processus valide. Ceci est l'équivalent, dans π, de la restriction qui interdit de créer un receveur à partir d'un nom reçu. Les règles définissant l'équivalence structurelle sur ce fragment sont la règle de « scope extrusion » de π, les règles usuelles de monoïde commutatif pour $(P, \mid, \mathtt{0})$ et les règles concernant l'application :

$$\begin{array}{rclrcl} P \mid Q & \equiv & Q \mid P & (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) \\ \langle u \Leftarrow P\rangle\ a & \equiv & \langle u \Leftarrow P\rangle & (P \mid Q)\ a & \equiv & (P\ a) \mid (Q\ a) \\ \mathtt{0} \mid P & \equiv & P & ((\nu u)P) \mid Q & \equiv & (\nu u)(P \mid Q)\ (\text{si } u \notin \text{fn(Q)}) \end{array}$$

La réduction consomme une déclaration et exécute le processus déclaré à l'emplacement de l'émetteur :

$$\langle u \Leftarrow P\rangle \mid (u\ a_1 \ldots a_n) \rightarrow (P\ a_1 \ldots a_n)\ \text{(red decl)}$$

Types 2 Typage du fragment π-Calcul

$$\Gamma \vdash \mathtt{0} : \bot\ \text{(type nil)} \qquad \frac{\Gamma, u : \tau \vdash P : \tau}{\Gamma, u : \tau \vdash \langle u \Leftarrow P\rangle : \bot}\ \text{(type decl)}$$

$$\frac{\Gamma \vdash P : \tau}{\Gamma_{|u} \vdash (\nu u)P : \tau}\ \text{(type new)} \qquad \frac{\Gamma \vdash P : \tau \quad \Gamma \vdash Q : \tau}{\Gamma \vdash P \mid Q : \tau}\ \text{(type par)}$$

2.3. Description des enregistrements dans $\pi^\star$

Les enregistrements sont construit incrémentalement à partir de l'enregistrement vide : $[\,]$, en utilisant l'opération d'extension/modification : $[\,Q\,,\,l = P\,]$ qui ajoute/modifie le champ l dans l'enregistrement Q. Intuitivement, un enregistrement est une fonction des champs vers les processus et la sélection est une application. Suivant cette intuition, nous notons la sélection $(P\ \cdot l)$ et nous choisissons comme règles d'équivalence structurelle pour la sélection celles de l'application.

$$(P \mid Q)\ \cdot l \ \equiv\ (P\ \cdot l \mid Q\ \cdot l) \qquad \langle u \Leftarrow P\rangle\ \cdot l \ \equiv\ \langle u \Leftarrow P\rangle \qquad \ldots$$

[4]la déclaration $\langle u \Leftarrow (\lambda x)P\rangle$ est l'équivalent de la réception : $u(x).P$ du π-calcul.

Tab. 2 Règles pour le sous-typage

$$\bot <: \tau \ (\text{sub } \bot) \qquad \tau <: \top \ (\text{sub } \top) \qquad [\,\omega\,,\, l : \tau\,] <: [\,] \ (\text{sub } [\,])$$

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ (\text{sub trans}) \qquad \frac{\alpha <: \beta \Rightarrow \tau <: \omega}{\mu\alpha.\tau <: \mu\beta.\omega}\ (\text{sub rec})$$

$$\frac{k \neq l}{[\,[\,\upsilon\,,\, k : \omega\,]\,,\, l : \tau\,] <: [\,[\,\upsilon\,,\, l : \tau\,]\,,\, k : \omega\,]}\ (\text{sub swap})$$

$$\frac{\omega_1 <: \tau_1 \qquad \tau_2 <: \omega_2}{\tau_1 \rightarrow \tau_2 <: \omega_1 \rightarrow \omega_2}\ (\text{sub app}) \qquad \frac{\omega <: \omega' \qquad \tau <: \tau'}{[\,\omega\,,\, l : \tau\,] <: [\,\omega'\,,\, l : \tau'\,]}\ (\text{sub over})$$

Nous notons : $[\, l_i = P_i{}^{1 \leq i \leq n}\,]$ le terme : $[\,[\,[\,]\,,\, l_1 = P_1\,]\,, \ldots, l_n = P_n\,]$, si les l_i sont distincts. Il existe deux règles de réduction sur les enregistrements :

$$([\,Q\,,\, l = P\,] \ \cdot l) \rightarrow P \ (\text{red sel})$$

$$\frac{k \neq l}{([\,Q\,,\, l = P\,] \ \cdot k) \rightarrow (Q \ \cdot k)}\ (\text{red over})$$

Il faut remarquer que notre formalisation des enregistrements est proche de celle proposée par Wand [28]. Ainsi il existe une unique opération pour modifier ou pour ajouter un champ à un enregistrement. Nous pouvons donc définir une relation de sous-typage (Tab. 2) de manière standard. Par exemple, la notion de sous-typage en largeur peut se dériver avec notre définition.

Proposition 1 (Sous-typage en largeur)
$[\, l_1 : \tau_1, \ldots, l_{n+m} : \tau_{n+m}\,] \ <: \ [\, l_1 : \tau_1, \ldots, l_n : \tau_n\,]$
La preuve de cette proposition utilise les règles (sub swap), (sub [ ]) et (sub over). Par exemple, le jugement : $[\, k : \tau, l : \sigma\,] <: [\, k : \tau\,]$ est une conséquence de : $[\,[\, k : \tau\,]\,,\, l : \sigma\,] <: [\,[\, l : \sigma\,]\,,\, k : \tau\,]$ et de $[\, l : \sigma\,] <: [\,]$ On peut également prouver la propriété de conservation du typage.

Proposition 2 (Conservation du typage) *Si* $\Gamma \vdash P : \tau$ *et* $P \rightarrow P'$, *alors* $\Gamma \vdash P' : \tau$.

2.4. Équivalence opérationnelle

Nous définissons une relation d'équivalence opérationnelle entre termes de $\pi^\star$ ($\approx$) qui est utilisée pour démontrer la correction de l'interprétation de $\mathbf{Ob}_{\mathbf{1}<:}$ (voir Th. 2). Cette relation est la plus grande bisimulation qui préserve un critère d'observabilité, appelé barbe, et qui soit une congruence [17] (c'est une variante de la congruence barbue

Types 3 Typage des enregistrement

$$\Gamma \vdash [\,] : [\,] \ (\text{type } [\,]) \qquad \frac{\Gamma \vdash P : \tau \qquad \Gamma \vdash Q : \omega}{\Gamma \vdash [\, Q \,,\, l = P \,] : [\, \omega \,,\, l : \tau \,]} \ (\text{type over})$$

$$\frac{\Gamma \vdash P : [\, l : \tau \,]}{\Gamma \vdash (P \ \cdot l) : \tau} \ (\text{type sel}) \qquad \frac{\Gamma \vdash P : \tau \qquad \tau <: \omega}{\Gamma \vdash P : \omega} \ (\text{type sub})$$

faible [20]). Cependant, alors que les actions observables de CCS et de π sont les émissions sur un canal libre[5], nous choisissons au contraire d'observer la présence de valeurs, c.-à-d. d'abstractions, suivant en cela la définition des équivalences « à la Morris » pour le λ-calcul [4, ex. 16.5.5]. Nous disons que P est une valeur de $\pi^\star$ (et donc que P est observable, noté : $P\!\downarrow$) si $P \equiv (\nu \tilde{u})((\lambda x)Q \mid R)$. La version faible des barbes, qui est celle utilisée pour définir $\approx$, est $P\!\Downarrow \; =_{\text{def}} \exists P', P \to^* P'\!\downarrow$.

Définition 1 *Une relation S est une* simulation faible barbue *si pour chaque couple $(P, Q) \in S$:*

(1) $P \to P'$ *implique* $Q \to^* Q'$ *et* $(P', Q') \in S$ *;*

(2) $P\!\downarrow$ *implique* $Q\!\Downarrow$.

S est une bisimulation barbue (faible) si S et S^{-1} sont des simulations barbues (faibles). P et Q sont observationnellement équivalent, noté $P \approx Q$, ssi il existe une bisimulation barbue faible S, qui soit une congruence et telle que $(P, Q) \in S$.

L'étude des propriétés de cette équivalence n'est pas le but de cet article, et nous renvoyons le lecteur à [6], où une équivalence similaire est définie.

3. Les objets dans le calcul bleu

Avant de présenter la définition des opérateurs du calcul d'objets concurrent et leur codage dans $\pi^\star$, nous étudions l'exemple de la *cellule mutable*, qui est le prototype des objets introduit ci-après. En effet, un objet sera modélisé par une cellule qui encapsule un état (représenté par un enregistrement de méthode comme dans l'interprétation classique des objets par des enregistrements récursifs [8]) et le nom de l'objet sera la référence à laquelle son état est mémorisé. Soit $\mathrm{R}_s(b)$ l'enregistrement :

$$\mathrm{R}_s(b) \ =_{\text{def}} \ [\, get = (s \ b \mid b), \ put = (\lambda x)(s \ x) \,]$$

[5] l'équivalent, dans $\pi^\star$, est d'observer les messages : $(u \ a_1 \ldots a_n)$

La cellule de « nom » o est le processus $(\text{CELL}(o)\ a_0)$ tel que :

$$\begin{array}{rcl} \text{CELL}(o) & =_{\text{def}} & \mathbf{def}\ s = (\lambda b)\langle o \Leftarrow \mathrm{R}_s(b)\rangle\ \mathbf{in}\ o \\ (\text{CELL}(o)\ a_0) & \approx & \mathbf{def}\ s = (\lambda b)\langle o \Leftarrow \mathrm{R}_s(b)\rangle\ \mathbf{in}\ \langle o \Leftarrow \mathrm{R}_s(a_0)\rangle \end{array}$$

L'application : $(\text{CELL}(o)\ a_0)$ permet d'initialiser la cellule à la valeur a_0, tandis que $P_{get} =_{\text{def}} (\text{CELL}(o)\ a_0) \mid (o\ \cdot get)$ et $P_{put} =_{\text{def}} (\text{CELL}(o)\ a_0) \mid (o\ \cdot put\ a)$ se réduisent déterministiquement en :

$$\begin{array}{rcl} P_{get} & \rightarrow & \mathbf{def}\ s = (\lambda b)\langle o \Leftarrow \mathrm{R}_s(b)\rangle\ \mathbf{in}\ (\langle o \Leftarrow \mathrm{R}_s(a_0)\rangle \mid o\ \cdot get) \\ & \rightarrow & \mathbf{def}\ s = (\lambda b)\langle o \Leftarrow \mathrm{R}_s(b)\rangle\ \mathbf{in}\ (\mathrm{R}_s(a_0)\ \cdot get) \\ & \rightarrow & \mathbf{def}\ s = (\lambda b)\langle o \Leftarrow \mathrm{R}_s(b)\rangle\ \mathbf{in}\ (s\ a_0 \mid a_0)\ \equiv\ (\text{CELL}(o)\ a_0) \mid a_0 \end{array}$$

$$P_{put} \rightarrow^* \mathbf{def}\ s = (\lambda b)\langle o \Leftarrow \mathrm{R}_s(b)\rangle\ \mathbf{in}\ (s\ a)\ \equiv\ (\text{CELL}(o)\ a)$$

Il est important de noter l'utilisation linéaire qui est faîte de la référence o dans $(\text{CELL}(o)\ a)$. À chaque invocation de la cellule, l'unique déclaration $\langle o \Leftarrow \mathrm{R}_s(a)\rangle$ qui est présente, est consommée et un unique message $(s\ a')$ (qui tient le rôle de sémaphore) est libéré. Ce message libère à son tour une unique déclaration $\langle o \Leftarrow \mathrm{R}_s(a')\rangle$. Ainsi, on peut démontrer qu'il existe une seule ressource disponible en o et que celle-ci mémorise la dernière valeur transmise dans un appel : $(o\ \cdot put)$. Une extension possible de l'exemple de la cellule est la cellule à n valeurs. Ce processus utilise un enregistrement de $2n$ champs : $(put_1, \ldots, put_n)$ et $(get_1, \ldots, get_n)$ à la place de $\mathrm{R}_s(b)$. Soit $\tilde{b}$ le tuple $(b_1, \ldots, b_n)$:

$$\mathrm{R}_s(\tilde{b}) \quad =_{\text{def}} \quad \left[\begin{array}{ll} \ldots & {\scriptstyle 1 \le i \le n} \\ get_i = (s\ b_1 \ldots b_n \mid b_i), & \\ put_i = (\lambda x_i)(s\ b_1 \ldots x_i \ldots b_n), & \\ \ldots & \end{array} \right]$$

$$\text{NCELL}(o) =_{\text{def}} \mathbf{def}\ s = (\lambda \tilde{b})\langle o \Leftarrow \mathrm{R}_s(\tilde{b})\rangle\ \mathbf{in}\ o$$

$$(\text{NCELL}(o)\ a_1 \ldots a_n) \mid o\ \cdot get_i \rightarrow^* (\text{NCELL}(o)\ a_1 \ldots a_n) \mid a_i$$

$$(\text{NCELL}(o)\ a_1 \ldots a_n) \mid o\ \cdot put_i\ a \rightarrow^* (\text{NCELL}(o)\ a_1 \ldots a \ldots a_n)$$

3.1. Un calcul d'objets concurrents

Dans cette section, nous présentons un calcul d'objets concurrents en définissant un ensemble d'opérateurs ainsi que leurs comportements opérationnel. Par la suite, nous prouvons que ces opérateurs (et les règles de réduction associées) sont dérivable dans $\pi^\star$. Nous étudions également les règles de typage dérivés. Dans la spécification du calcul d'objets (Tab. 3), nous utilisons un sous-ensemble de références $(o, a, b, \dots \in \mathcal{R})$ pour nommer les objets. Nous utiliserons L pour représenter le « corps

Tab. 3 Syntaxe et réduction dans $\pi^\star$avec objets

$\varsigma(x)P$	méthode
$\mathbf{obj}\, o = \{l_i = \varsigma(x_i)P_i{}^{1\leq i\leq n}\}\mathbf{in}\ P$	objet avec n méthodes
$P \Leftarrow l$	invocation de la méthode l
$P \leftarrow l = \varsigma(x)Q$	modification de la méthode l
$\mathbf{clone}(o)$	clonage de l'objet o

Soit $L =_{\mathrm{def}} (l_i = \varsigma(x_i)P_i{}^{1\leq i\leq n})$. On suppose que $\mathrm{E}[.]$ ne lie aucune variable libre :

$$\mathbf{obj}\, o = \{L\}\mathbf{in}\ \mathrm{E}[o \Leftarrow l_j] \rightarrow_\varsigma \mathbf{obj}\, o = \{L\}\mathbf{in}\ \mathrm{E}[P_j\{o/x_j\}]$$

$$\begin{pmatrix} \mathbf{obj}\, o = \{L\} \\ \mathbf{in}\ \mathrm{E}[o \leftarrow l_j = \varsigma(x)P] \end{pmatrix} \rightarrow_\varsigma \begin{pmatrix} \mathbf{obj}\, o = \{l_j = \varsigma(x)P, l_i = \varsigma(x_i)P_i{}^{i\neq j}\} \\ \mathbf{in}\ \mathrm{E}[o] \end{pmatrix}$$

$$\mathbf{obj}\, o = \{L\}\mathbf{in}\ \mathrm{E}[\mathbf{clone}(o)] \rightarrow_\varsigma \begin{pmatrix} \mathbf{obj}\, o = \{L\} \\ \mathbf{in}\ (\mathrm{E}[\mathbf{obj}\, a = \{L\}\mathbf{in}\ a]) \end{pmatrix}$$

d'un objet » : $L =_{\mathrm{def}} (l_1 = \varsigma(x_1)P_1, \ldots, l_n = \varsigma(x_n)P_n)$. Un exemple d'objets est celui qui produit récursivement une infinité de copie de lui-même en parallèle. Soit $L =_{\mathrm{def}} (rec = \varsigma(x)(\mathbf{clone}(x) \mid x \Leftarrow rec))$, alors :

$$\begin{aligned} \mathbf{obj}\, a = \{L\}\mathbf{in}\ (a \Leftarrow rec) &\rightarrow_\varsigma \mathbf{obj}\, a = \{L\}\mathbf{in}\ (\mathbf{clone}(a) \mid a \Leftarrow rec) \\ &\rightarrow_\varsigma \begin{pmatrix} \mathbf{obj}\, a = \{L\} \\ \mathbf{in}\ (\mathbf{obj}\, b = \{L\}\mathbf{in}\ (b \mid a \Leftarrow rec)) \end{pmatrix} \end{aligned}$$

Il faut noter que, moyennant une syntaxe plus compliquée, il est possible de définir des objets mutuellement dépendant.

3.2. Interprétation des objets dans $\pi^\star$

Le processus modélisant un objet est inspiré de l'exemple de la cellule mutable. Dans le codage proposé dans la table 4, un objet : $(\mathbf{obj}\, o = \{L\}\mathbf{in}\ P)$ (avec $L =_{\mathrm{def}} (l_i = \varsigma(x_i)P_i{}^{1\leq i\leq n})$), est une « cellule n-aire » augmentée d'un champ *clone* pour le clonage. Les méthodes $\varsigma(x_i)P_i$ sont codées par des abstractions $(\lambda x_i)P_i$. Les champs get_{l_i} permettent d'invoquer une méthode de l'objet (dans laquelle le paramètre x_i est remplacé par le nom de l'objet) et les champs put_{l_i} permettent de modifier ces méthodes. Il faut noter que la portée du nom de l'objet est contrôlée par une restriction et que la modification d'une méthode retourne une référence à l'objet modifié. Une remarque supplémentaire

Tab. 4 Codage des opérateurs du calcul objets

Soit $L =_{\text{def}} (l_i = \varsigma(x_i)P_i{}^{1\leq i\leq n})$.

$$\mathrm{T}_s(o,\tilde{b}) \quad =_{\text{def}} \quad \left[\begin{array}{l} \ldots \\ get_{l_i} = (s\ b_1 \ldots b_n \mid b_i\ o), \\ put_{l_i} = (\lambda x_i)(s\ b_1 \ldots x_i \ldots b_n \mid o), \\ \ldots \\ clone = (s\ b_1 \ldots b_n \mid x_{clone}\ b_1 \ldots b_n) \end{array}\right]$$

$$\text{OBJ}(o) =_{\text{def}} \mathbf{def}\, o = (\lambda\tilde{b})\langle o \Leftarrow \mathrm{T}_s(o,\tilde{b})\rangle\, \mathbf{in}\, o$$

$$\mathbf{obj}\, o = \{L\}\mathbf{in}\ P =_{\text{def}} \left(\begin{array}{l} \mathbf{def}\ x_{clone} = (\lambda\tilde{f})(\nu a)(\text{OBJ}(a)\ f_1 \ldots f_n \mid a) \\ \mathbf{in}\ (\nu o)((\text{OBJ}(o)\ (\lambda x_1)P_1 \ldots (\lambda x_n)P_n) \mid P) \end{array}\right)$$

$$P \Leftarrow l_j =_{\text{def}} (P\ \cdot get_{l_j}) \qquad\qquad \mathbf{clone}(o) =_{\text{def}} (o\ \cdot clone)$$

$$(P \leftarrow l_j = \varsigma(x)Q) =_{\text{def}} (P\ \cdot put_{l_j}\ (\lambda x)Q)$$

est que, dans la définition du clonage, de l'invocation et de la modification de méthodes, nous n'utilisons que la sélection et l'application. En particulier : $\mathbf{obj}\, o = \{L\}\mathbf{in}\ ([.] \Leftarrow l)$ est un contexte d'évaluation. Il est donc possible de dériver, pour ces opérations, les mêmes règles d'équivalence structurelle que pour l'application. Par exemple, on peut montrer que $(\mathbf{def}\, D\, \mathbf{in}\, P \mid Q) \Leftarrow l \ \equiv\ \mathbf{def}\, D\, \mathbf{in}\, (P \Leftarrow l \mid Q \Leftarrow l)$. Le théorème 1 établi la correspondance opérationnelle entre $\rightarrow_\varsigma$ (Tab. 3) et $\rightarrow$.

Théorème 1 (Équivalence opérationnelle) *La définition des règles de réduction des objets est* complète *vis-à-vis du codage dans* $\pi^\star$ *:* $P \rightarrow_\varsigma P' \Rightarrow P \rightarrow^* P'$. *Le codage est aussi* correct *:* $P \rightarrow Q$ *implique* $Q \rightarrow^* Q'$ *et* $P \rightarrow_\varsigma Q'$.

3.3. Système de type pour les objets

Nous établissons, dans cette section, le type dérivé pour les objets à partir de leur codage dans $\pi^\star$. Soit $(\mathbf{obj}\ [\,l_i : \rho_i^{1\leq i\leq n}\,])$ le type récursif :

$$(\mathbf{obj}\ [\,l_i : \rho_i^{1\leq i\leq n}\,]) \ =_{\text{def}}\ \mu t. \left[\begin{array}{l} \ldots \qquad\qquad\qquad\quad {}^{1\leq i\leq n} \\ get_{l_i} = \rho_i, \\ put_{l_i} = (t \rightarrow \rho_i) \rightarrow t, \\ \ldots \\ clone = t \end{array}\right]$$

Nous notons ρ le type $[\, l_i : \rho_i^{1 \leq i \leq n} \,]$. On montre que le type de l'enregistrement $\mathrm{T}_s(o, \tilde{b})$, définit Tab. 4, est $(\mathbf{obj}\ \rho)$ si les $(b_i)_{1 \leq i \leq n}$ on le type $(t \to \rho_i)\{(\mathbf{obj}\ \rho)/t\}$. La preuve de cette propriété se fait en remarquant que :

$$\Gamma \vdash \mathrm{T}_s(o, \tilde{b}) : \begin{bmatrix} get_{l_i} = \rho_i, & {}^{1 \leq i \leq n} \\ put_{l_i} = (\tau \to \rho_i) \to \tau, & \\ clone = \tau & \end{bmatrix}$$

$$\begin{aligned} \text{où } \Gamma = \ & o : \tau,\ b_i : (\tau \to \rho_i), \ldots, o : (\tau \to \rho_1) \to \ldots \to (\tau \to \rho_n) \to \bot, \\ & x_{clone} : (\tau \to \rho_1) \to \ldots \to (\tau \to \rho_n) \to \tau \end{aligned}$$

Donc $(\mathbf{obj}\ \rho)$ est aussi le type de la référence o, utilisée pour nommer l'objet. Le système de type dérivés est présenté dans le tableau Types 4. On peut remarquer la similitude avec le système de type de $\mathbf{Ob_{1<:}}$ (Tab. 5). Il faut aussi noter que le type d'un objet n'est pas covariant,

Types 4 Système de types dérivés pour les objets

Soit $\rho = [\, l_i : \rho_i^{1 \leq i \leq n} \,]$

$$\frac{\Gamma, x_i : (\mathbf{obj}\ \rho) \vdash P_i : \rho_i \qquad \forall i, 1 \leq i \leq n \qquad \Gamma, o : (\mathbf{obj}\ \rho) \vdash P : \tau}{\Gamma \vdash \mathbf{obj}\, o = \{l_i = \varsigma(x_i) P_i{}^{1 \leq i \leq n}\} \mathbf{in}\ P : \tau}$$

$$\frac{\Gamma, x : (\mathbf{obj}\ \rho) \vdash Q : \rho_j \qquad \Gamma \vdash P : (\mathbf{obj}\ \rho) \qquad 1 \leq j \leq n}{\Gamma \vdash P \Leftarrow l_j = \varsigma(x) Q : (\mathbf{obj}\ \rho)}$$

$$\frac{\Gamma \vdash o : (\mathbf{obj}\ \rho)}{\Gamma \vdash \mathbf{clone}(o) : (\mathbf{obj}\ \rho)} \qquad \frac{\Gamma \vdash P : (\mathbf{obj}\ \rho) \qquad 1 \leq j \leq n}{\Gamma \vdash P \Leftarrow\!\!| l_j : \rho_j}$$

c.-à-d. que $\rho <: \sigma$ n'implique pas $(\mathbf{obj}\ \rho) <: (\mathbf{obj}\ \sigma)$. En effet, le type ρ_i de la méthode l_i apparaît de manière covariante dans le type du champ get_{l_i} et de manière contravariante dans le type de put_{l_i}. Néanmoins, soit ρ_1 et ρ_2 les types $[\, l_i : \rho_i{}^{1 \leq i \leq n} \,]$ et $[\, l_i : \rho_i{}^{1 \leq i \leq n+m} \,]$. On montre qu'un objet o_2 de type $(\mathbf{obj}\ \rho_2)$ peut être utilisé partout où o_1, de type $(\mathbf{obj}\ \rho_1)$, peut l'être. On dit que o_2 peut se substituer à o_1.

Proposition 3 (Substituivité)
$(\mathbf{obj}\ [\, l_1 : \rho_1, \ldots, l_{n+m} : \rho_{n+m} \,]) \ <: \ (\mathbf{obj}\ [\, l_1 : \rho_1, \ldots, l_n : \rho_n \,])$

Ce résultat découle de la proposition 1 et de la règle (sub rec) (Tab. 2). La substituivité, comme le polymorphisme dans ML ou le sous-typage pour les enregistrements, est importante pour permettre la *réutilisation de code* car elle permet de définir des fonctions se comportant uniformément sur des entrées de types différents. Un exemple est la fonction : $(\lambda a)(a \Leftarrow\!\!| repaint)$, de type : $((\mathbf{obj}\ [\, repaint : \bot \,]) \to \bot)$, qui peut

Tab. 5 Objets, réduction et types dans $\mathbf{Ob_{1<:}}$

x	variable
$\varsigma(x)b$	méthode
$[\,l_i = \varsigma(x_i)b_i{}^{1 \leq i \leq n}\,]$	objet avec n méthodes
$o{\cdot}l$	invocation de la méthode l sur l'objet o
$o{\cdot}l \Leftarrow \varsigma(x)b$	modification de l dans o

$$\frac{o \rightsquigarrow v = [\,l_i = \varsigma(x_i)b_i{}^{1 \leq i \leq n}\,] \qquad 1 \leq j \leq n}{o{\cdot}l_j \rightsquigarrow b_j\{v/x_j\}}\ \text{(select)}$$

$$\frac{o \rightsquigarrow [\,l_i = \varsigma(x_i)b_i{}^{1 \leq i \leq n}\,] \qquad 1 \leq j \leq n}{o{\cdot}l_j \Leftarrow \varsigma(x)b \rightsquigarrow [\,l_j = \varsigma(x)b,\ l_i = \varsigma(x_i)b_i{}^{i \neq j}\,]}\ \text{(update)}$$

Soit A le type $[\,l_i : B_i{}^{1 \leq i \leq n}\,]$. La relation de sous-typage est telle que $[\,l_i : B_i{}^{1 \leq i \leq n+m}\,] <: [\,l_i : B_i{}^{1 \leq i \leq n}\,]$.

$$\frac{E \vdash_\varsigma a : A \qquad A <: B}{E \vdash_\varsigma a : B}\ \text{(subsumption)}$$

$$\frac{E, x_i : A \vdash_\varsigma b_i : B_i \qquad \forall i, 1 \leq i \leq n}{E \vdash_\varsigma [\,l_i = \varsigma(x_i)b_i{}^{1 \leq i \leq n}\,] : A}\ \text{(object)}$$

$$\frac{E \vdash_\varsigma a : A \qquad E, x : A \vdash_\varsigma b : B_j \qquad 1 \leq j \leq n}{E \vdash_\varsigma a{\cdot}l \Leftarrow \varsigma(x)b : A}\ \text{(update)}$$

$$E, x : A \vdash_\varsigma x : A\ \text{(axiom)} \qquad \frac{E \vdash_\varsigma a : A \qquad 1 \leq j \leq n}{E \vdash_\varsigma a{\cdot}l_j : B_j}\ \text{(select)}$$

s'appliquer à tout les objets ayant au moins la méthode *repaint* avec le type $\perp$.

4. Interprétation de $\mathbf{Ob_{1<:}}$

Abadi et Cardelli définissent dans [1] un calcul fonctionnel : ς, dans lequel les objets sont primitifs. Ce calcul formalise des concepts clefs des langage orientés objets tel que la notion de *self* (une méthode peut se référer à l'objet qui la contient par l'intermédiaire de self), la *substituivité* (un objet peut en émuler un autre avec moins de méthodes) ou la *modification de méthodes*. Nous donnons une interprétation de $\mathbf{Ob_{1<:}}$, une variante de ς qui possède un système de type du premier ordre

Tab. 6 Interprétation de $\mathbf{Ob_{1<:}}$

interprétation des termes :

$$[\![\, [\, l_i = \varsigma(x_i) b_i{}^{1 \le i \le n} \,] \,]\!] \;=\; \mathbf{obj}\, o = \{ l_i = \varsigma(x_i) [\![b_i]\!]^{1 \le i \le n} \} \mathbf{in}\; o$$

$$[\![o{\cdot}l \Leftarrow \varsigma(x) b]\!] \;=\; ([\![o]\!] \leftarrow l = \varsigma(x) [\![b]\!])$$

$$[\![x]\!] \;=\; \mathbf{clone}(x) \qquad\qquad [\![o{\cdot}l]\!] \;=\; [\![o]\!] \Leftarrow\!\!| l$$

interprétation des types :

$$[\![\, [\, l_i : B_i{}^{1 \le i \le n} \,] \,]\!] = \; (\mathbf{obj}\; [\, l_i : [\![B_i]\!]^{1 \le i \le n} \,])$$

$$[\![\emptyset]\!] = \; \emptyset \qquad\qquad [\![E, x : A]\!] = \; [\![E]\!],\; x : [\![A]\!]$$

avec sous-typage. Bien qu'il n'y ait aucune notion de nom ou d'identité dans ς, ni de notion de parallélisme, les constructions introduites dans la table 3 sont de manière évidente inspirée de celle d'Abadi et Cardelli. Cette filiation se reflète dans la simplicité de l'interprétation de $\mathbf{Ob_{1<:}}$. Remarquons toutefois que, alors que la réduction de $\mathbf{Ob_{1<:}}$ est basée sur la substitution de code (on substitue au paramètre self le code de l'objet), la réduction des objets dans $\pi^\star$ est basé sur le nommage des objets (on substitue à self le nom de l'objet). Ceci est plus proche du comportement des langages de programmation.

La sémantique de $\mathbf{Ob_{1<:}}$ est donnée Tab. 5. Il y a un unique lieur, ς, qui lie les occurrences de x dans la méthode : $\varsigma(x)b$. Nous considérons uniquement la stratégie de réduction faible (c.-à-d. telle que la réduction sous le lieur est interdite). Dans ce calcul, les termes : $[\, l_i = \varsigma(x_i) b_i{}^{1 \le i \le n} \,]$ sont les valeurs. Nous supposons aussi que $o \rightsquigarrow o'$ implique $\mathrm{E}\,[o] \rightsquigarrow \mathrm{E}\,[o']$, où $\mathrm{E}\,[.]$ est un contexte d'évaluation : $[.] \;\big|\; (\mathrm{E}\,[.]{\cdot}l_i) \;\big|\; (\mathrm{E}\,[.]{\cdot}l_i \Leftarrow \varsigma(x)b)$.

L'interprétation de $\mathbf{Ob_{1<:}}$ est donnée Tab. 6. Le théorème 2 énonce la correspondance entre $(\mathbf{Ob_{1<:}}, \rightsquigarrow)$ et $(\pi^\star, \,\rightarrow)$.

Théorème 2 ($\pi^\star$ simule $\mathbf{Ob_{1<:}}$) *Le codage de* $\mathbf{Ob_{1<:}}$ *est* complet *(i) et* correct *(ii) et l'interprétation des types est correcte (iii). On montre également que $A <: B$ implique $[\![A]\!] <: [\![B]\!]$.*

(i) $o \rightsquigarrow o' \Rightarrow [\![o]\!] \rightarrow^* \approx [\![o']\!]$;

(ii) $[\![o]\!] \rightarrow P \Rightarrow o \rightsquigarrow o'$ *et* $[\![o]\!] \rightarrow^* \approx [\![o']\!]$;

(iii) $E \vdash_\varsigma a : A \Rightarrow [\![E]\!] \vdash_{\pi^\star} [\![a]\!] : [\![A]\!]$

La preuve du théorème 2 utilise le Lemme 1 comme résultat intermédiaire pour prouver que $\mathbf{obj}\, o = \{L\} \mathbf{in}\; [\![b]\!] \{o/x\}$ équivaut à $[\![b\{v/x\}]\!]$ dès que

$[\![v]\!]$ est le processus $\mathbf{obj}\, o = \{L\}\mathbf{in}\ o$:

Lemme 1 (« object-replication ») *Soit* M [.] *un multi-contexte. Si o n'apparaît pas dans* M [.], *alors :*

$$\mathbf{obj}\, o = \{L\}\mathbf{in}\ \mathrm{M}\,[\mathbf{clone}(o)] \ \approx\ \mathrm{M}\,[\mathbf{obj}\, o = \{L\}\mathbf{in}\ o]$$

ceci implique que, si o apparaît dans P et Q uniquement sous la forme $\mathbf{clone}(o)$, *alors :*

$$\begin{array}{l} (1)\ \mathbf{obj}\, o = \{L\}\mathbf{in}\ (P \mid Q) \ \approx\ (\mathbf{obj}\, o = \{L\}\mathbf{in}\ P) \mid (\mathbf{obj}\, o = \{L\}\mathbf{in}\ Q) \\ (2)\ \mathbf{obj}\, o = \{L\}\mathbf{in}\ \mathbf{clone}(o) \ \approx\ \mathbf{obj}\, o = \{L\}\mathbf{in}\ o \end{array}$$

Notre interprétation peut être aisément étendue à une variante impérative de $\mathbf{Ob_{1<:}}$ en définissant $[\![\mathbf{clone}(o)]\!] =_{\mathrm{def}} \mathbf{clone}([\![o]\!])$ et $[\![\mathbf{let}\, x = o\, \mathbf{in}\, b]\!] =_{\mathrm{def}} \mathbf{def}\, x = [\![o]\!]\, \mathbf{in}\, [\![b]\!]$. Ceci nous donne l'équivalent du ς-calcul impératif de [1] dans lequel l'opérateur *let* serait « call-by-name ». Il est intéressant de remarquer que des équivalences vrai dans le calcul impératif [15] sont préservé par la traduction.

Lemme 2 (Quelques équivalences sur $[\![\mathbf{Ob_{1<:}}]\!]$) *Si v est l'objet* $[\, l_i = \varsigma(x_i)b_i{}^{1 \leq i \leq n}\,]$, *alors* $[\![\mathbf{clone}(v)]\!] \approx [\![v]\!]$ *et :*

$$\begin{cases} [\![v \cdot l_j]\!] \approx [\![\mathbf{let}\, x_j = v\, \mathbf{in}\, b_j]\!] \\ [\![(v \cdot l_j \Leftarrow \varsigma(x)b) \cdot l_j]\!] \approx [\![\mathbf{let}\, x = (v \cdot l_j \Leftarrow \varsigma(x)b)\, \mathbf{in}\, b]\!] \end{cases}$$

5. Conclusion et futurs développements

Un des résultats de cet article est la définition d'un calcul d'objets impératif et concurrent dérivé du calcul bleu. Son intérêt étant, en particulier, qu'il se fonde sur la notion plus naturelle de nommage des objets plutôt que sur la substitution de code (comme dans les calculs de [2, 13]). Les calculs de processus ont déjà été utilisé pour raisonner sur la fondation des langages orientés objets, mais notre but est différent ici. Plutôt que de se fixer sur la formalisation de la sémantique de langages déjà existant, nous avons tenté de prouver que $\pi^\star$ était la base idéale pour la conception d'un langage de programmation distribué de haut-niveau possédant certaines des caractéristiques des langages à objets. Ce but à été atteint grâce à une interprétation des objets comme un cas spécial de processus. Dans ce sens, nous suivons la trace des travaux de Pierce et Turner sur le π-calcul asynchrone qui était motivé par la conception du langage Pict [25].

Le codage des objets d'Abadi et Cardelli a été un test réussit qui a démontré l'expressivité de $\pi^\star$puisque nous avons pu donner un codage typée, simple et direct (c.-à-d. sans utilisation de transformations CPS) de $\mathbf{Ob_{1<:}}$ avec un système de type du premier ordre. Ceci nous conduit à conjecturer que notre codage peut être étendu au calcul impératif d'objets et à $\mathbf{Ob_{1<:\mu}}$, une extension de $\mathbf{Ob_{1<:}}$ avec des types récursifs.

Plusieurs développements des travaux présentés ici sont menés en ce moment. Par exemple, reprenant les résultats obtenus dans [10], nous avons définis un système de type polymorphe « à la ML » pour le calcul d'objets. Des recherches sont également poursuivie pour ajouter des primitives de distribution/migration à $\pi^\star$ [11]. Il serait alors intéressant de donner une interprétation d'un langage distribué orienté objets tel que Obliq [9], qui n'a pas de définition formelle. On pourrait également étudier la modélisation des *Object Request Brokers*, comme Corba. En effet les types enregistrements sont réminiscent des langages de définition d'interface utilisés dans les ORBs [22]. Nous pensons donc que $\pi^\star$ est un calcul approprié pour la vérification d'applications basées sur les objets distribués et bâties avec ces outils. Enfin, il serait intéressant de savoir quel équivalence est induite sur ς par $\approx$ et le codage de ς dans $\pi^\star$ (cf. section 2.4). Quelques exemples ont été fournis par le lemme 2.

Remerciements

Je tiens à remercier Gérard Boudol pour son aide. Il est à la fois le créateur du calcul bleu et à l'origine de plusieurs des idées contenues dans cet article.

Références

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects : Untyped and first-order systems. *Information and Computation*, 2(125) :78–102, March 1996.

[3] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Proceedings POPL'96*, pages 396–409, 1996.

[4] H. P. Barendregt. *The Lambda Calculus, Its syntax and Semantics*. North Holland, 1981.

[5] G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, INRIA, 1992.

[6] Gérard Boudol. The π-calculus in direct style. In *Conference Record of POPL '97 : The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, Paris, France, 15–17 January 1997.

[7] Gérard Boudol. Typing the use of resources in a concurrent calculus. In *ASIAN'97, the Asian Computing Science Conference*, Lecture Notes in Computer Science, Kathmandu, December 1997.

[8] L. Cardelli and J. C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1) :3–48, 1991.

[9] Luca Cardelli. A language with distributed scope. *Computer Systems*, 8 :27–59, 1995.

[10] Silvano Dal-Zilio. Implicit polymorphic type system for the blue calculus. Technical Report 3244, INRIA, September 1997.

[11] Silvano Dal-Zilio. Quiet and bouncing objects : Two migration abstractions in a simple distributed blue calculus. In *1st International Workshop on Semantics of Objects as Processes.* BRICS Notes Series, July 1998.

[12] P. Di Blasio and K. Fisher. A calculus for concurrent objects. In *7th International Conference on Concurrency Theory CONCUR'96*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1996.

[13] Kathleen Fisher and John C. Mitchell. A delegation-based object calculus with subtyping. In *proceedings of FCT'95*, volume 965 of *Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag, 1995.

[14] Andrew Gordon and D. Hankin. A concurrent object calculus. In *HLCL'98*, Elsevier ENTCS, 1998.

[15] Andrew Gordon, D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In *Proceedings of FST & TCS'97.* Lecture Notes in Computer Science, December 1997.

[16] C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, 1994.

[17] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2) :437–486, 1995.

[18] Cliff B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proceedings of CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.

[19] Josva Kleist and Davide Sangiorgi. Imperative objects and mobile processes, 1998. (to appear in Proceedings of PROCOMET'98).

[20] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[21] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2 :119–141, 1992.

[22] E. Najm and J.B. Stefani. A formal semantics for the ODP computational model. Technical Report PAA/3527, CNET, May 1993.

[23] Didier Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, INRIA, May 1991.

[24] Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report 3000, INRIA, 1996.

[25] David N. Turner. *The polymorphic pi-calculus : theory and implementation*. PhD thesis, University of Edinburgh, 1995.

[26] Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. to appear in proceedings of LICS'98.

[27] D. Walker. π-calculus semantics of object-oriented programming languages. In *Proceedings TACS'91*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991. Proc. TACS'91.

[28] Mitchell Wand. Complete type inference for simple objects. In *2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in LICS 1988.