N° d'ordre:

UNIVERSITÉ de NICE – SOPHIA ANTIPOLIS

École Doctorale des Sciences Pour l'Ingénieur
Laboratoire : INRIA Sophia Antipolis

THÈSE

présentée pour obtenir le titre de
Docteur en SCIENCES
discipline : Informatique

par

Silvano DAL-ZILIO

le calcul bleu: types et objets

Soutenue le 7 juillet 1999 à 10h devant le jury composé de :

Président	Denis CAROMEL	Professeur à l'UNSA (I3S)
Rapporteurs	Martín ABADI	Senior Researcher (Compaq SRC)
	Didier RÉMY	Directeur de Recherches (INRIA)
Directeur	Gérard BOUDOL	Directeur de Recherches (INRIA)
Examinateurs	Jean-Jacques LÉVY	Directeur de Recherches (INRIA)
	Matthew HENNESSY	Professeur (University of Sussex)

à l'Institut National de Recherche en Informatique et Automatiques de Sophia Antipolis

Remerciements

Je remercie vivement M. Denis Caromel de présider mon jury de thèse et MM. Jean-Jacques Lévy et Matthew Hennessy de participer à ce jury. Mes interactions avec Jean-Jacques et les membres du projet PARA ont été très fructueuses, et leur travail sur le calcul Join a été un exemple pour moi. Quant à Matthew, ces nombreux séjours à Sophia Antipolis font que je le considère un peu comme le «septième homme» de notre équipe.

Je remercie MM. Didier Rémy et Martìn Abadi d'avoir accepté d'être mes rapporteurs. J'ai appris à estimer Martìn en découvrant ses travaux sur les calculs d'objets, mais aussi la pluralité des sujets de recherches qu'il a abordé. Mais c'est avant tout sa gentillesse au cours des six mois qui viennent de s'écouler qui m'a marqué. Je remercie aussi, tout spécialement, Didier pour le temps qu'il m'a consacré durant ces trois dernières années, et pour avoir répondu à mes nombreuses questions sur les sytèmes de types et les enregistrements.

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 ce manuscrit. En tant que directeur de thèse, sa patience et sa pédagogie m'ont permis de passer à travers les difficultés de la thèse.

Je remercie mes amis et collègues du CMA et de l'INRIA Sophia Antipolis que j'ai eu la chance de cotoyer: Gérard Berry, Amar Bouali, Michel Bourdellès, Frédéric Boussinot, Robert De simone, Xavier Fornari, Olivier Ploton, Annie Ressouche, Valérie Roy, Jean-Ferdinand Susini, Horia Toma, ... et plus particulièrement l'équipe «théorie du parallélisme»: Roberto Amadio, Ilaria Castellani, Massimo Merro et Davide Sangiorgi. Chacun d'eux mériteraient une page complète de remerciements. Grazie mille a tutti...

Ce travail a été financé par une bourse du Ministère de l'Éducation nationale, de la Recherche et de la Technologie. J'ai également bénéficié d'un soutien financier de France Telecom (CTI-CNET 95-1B-182, Modélisations de Sytèmes Mobiles), du projet RNRT MARVEL, et du centre franco-indien de promotion de la recherche avancée (projet de recherche 1502-1).

Table des matières

CHAPITRE 1

Introduction

Les tâches dévolues à un programmeur peuvent, de façon schématique, se découper en trois catégories qui sont: comprendre ce qu'il doit programmer, écrire des programmes, et corriger des erreurs. Chacune de ces activités peut bénéficier de l'apport des méthodes formelles et, en particulier, de l'apport des outils fournis par la sémantique des langages de programmation, qui permet de définir une modélisation mathématiques des programmes grâce, par exemple, à l'utilisation d'un formalisme algébrique comme le λ-calcul.

Tout d'abord, l'existence d'un modèle du programme – ou du système informatique – que l'on cherche à construire est un élément qui permet de simplifier la compréhension d'un cahier des charges. On peut l'interpréter comme une «super documentation» du programme. En effet, un modèle formel fourni une description non équivoque, qui autorise des traitements automatisés par des outils de génie logiciel. Ce modèle est également utile dans la tâche de validation d'un programme, son «debuggage». En effet, on peut espérer utiliser ce modèle dans des preuves basées sur des méthodes rigoureuses de vérifications et de raisonnements. Nous pouvons essayer de faire comprendre l'utilité d'associer modèle théorique et programme, à travers une métaphore assez simple. On peut comparer un programme construit sans base théorique à de la fausse monnaie. Intuitivement, la valeur de ce programme n'est liée qu'à la confiance que ces utilisateurs lui donnent. Le jour où ce programme ne se comportera pas comme prévu – où qu'il faudra confier à un nouveau venu la tâche de le modifier –, il aura perdu toute sa valeur.

L'idée – qui n'est pas neuve – qui motive notre étude est que l'activité d'écriture du code peut, elle aussi, bénéficier des outils de la sémantique. On peut imaginer utiliser les concepts et les outils développés en sémantique pour définir un langage de programmation qui «n'encourage pas» les fautes. Cette application de la sémantique se motive très facilement. Il suffit, par exemple, de comparer les différences de productivité – pour le codage et la maintenance – entre un programme écrit en assembleur, et un programme écrit dans un langage de plus haut niveau, tel que Ada par exemple. Cette idée de se servir des concepts issus de la sémantique pour les appliquer à la conception de langage de programmation a d'ailleurs déjà été exploité. Elle a, par exemple, conduit à la création des langages fonctionnels tel que ML.

Aujourd'hui, il nous faut employer les outils de la sémantique à l'étude des systèmes distribuées, qui sont encore mal modélisés, mais qui néanmoins connaissent un développement effréné, poussé par la croissance continue des applications utilisant les réseaux. De même, il nous faut pouvoir raisonner sur les paradigmes de programmation qui se développent, comme la programmation à base de composants ou d'agents, la mobilité du code, ... Un outil mathématique prometteur pour modéliser ces nouveaux concepts, comparable à ce qu'a été le λ-calcul pour les applications

séquentielles, est constitué par les algèbres de processus – on dit aussi calcul de processus – comme le π-calcul par exemple. Dans cette thèse, nous étudions un calcul de processus nommé le *calcul bleu*, qui est basé à la fois sur le λ-calcul et le π-calcul, en nous concentrant sur l'expressivité de ce calcul et sur le «modèle de programmation» qu'il définit, plus que sur l'aspect vérification. En particulier, nous étudions comment la programmation fonctionnelle et la programmation orientée objet peuvent se modéliser dans ce calcul, et s'il est possible d'y généraliser les notions de typage de ces deux modèles.

Calcul de processus mobile

Le π-calcul est un calcul de processus basé sur le paradigme de l'interaction par échange de messages [91], qui représente une évolution du calcul CCS [95]. Cette évolution consiste en la possibilité d'échanger les noms de canaux comme valeurs au cours d'une communication. L'expressivité du π-calcul est donc supérieure à celle de CCS et il est possible, par exemple, d'y modéliser les systèmes pour lesquels la structure des communications évolue au cours du temps. On parle de *systèmes mobiles*.

Tout en étant défini à partir d'un nombre réduit d'opérateurs de base – réception et émission sur un canal, composition parallèle, création de canal privé et récursion – le π-calcul est suffisamment expressif pour modéliser une vaste gamme de systèmes concurrents, et son intérêt a déjà été démontré dans plusieurs utilisations, en particulier comme support pour formaliser des spécifications de systèmes distribués, et aussi comme outil pour la vérification formelle. Très récemment, il a été utilisé pour valider des concepts de programmation pour des langages reposant sur la mobilité. Nous faisons référence ici aussi bien à la mobilité de code, qu'à la mobilité au sens du π-calcul, c'est-à-dire la reconfiguration dynamique de la «géométrie des interactions». Ainsi, des langages de programmation parallèle directement basés sur le π-calcul ont été proposés. Par exemples le langage Pict [105, 130], développé par B. Pierce et D. Turner, et le langage Join [54, 52, 55], développé par le projet PARA de l'INRIA Rocquencourt.

Le π-calcul est donc passé du rôle d'outil mathématique qui permet de comprendre un programme, à celui de modèle mathématique pour le fondement de langages de programmation. C'est le chemin emprunté 20 ans plus tôt par son «grand-père», le λ-calcul, qui a été utilisé dans l'étude de la sémantique des langages séquentiels, avant de servir de fondement à l'élaboration de langages fonctionnels tels que ML par exemple [93]. Le parallèle entre ces deux modèles de programmation ne s'arrête pas là. Ainsi, chacun de ces modèles est basé sur un paradigme de calcul et de composition défini mathématiquement. Le λ-calcul est basé sur l'application et les fonctions d'ordre supérieur [71], tandis que le π-calcul est basé sur l'échange de messages et la composition parallèle. Est-il possible de combiner ces deux approches?

Dans cette thèse, nous étudions une extension directe du π-calcul et du λ-calcul que nous appelons le calcul bleu, et qui a été définie par G. Boudol dans [23]. La programmation fonctionnelle offre l'avantage de mettre en oeuvre des concepts mathématiquement très bien compris, qui apportent des éléments de sécurité, comme l'analyse statique par typage par exemple. On pense aussi à des notions de vérification, comme l'analyse abstraite, ou à des notions d'optimisation, comme l'évaluation partielle. Or ces concepts sont loin d'être aussi bien maîtrisés dans le monde concurrent, en particulier à cause du non-déterminisme et de l'absence d'une bonne notion de type. Aussi, notre but est de réitérer avec le calcul bleu certains des succès rencontrés avec le λ-calcul. En quelque sorte, si il devait n'y avoir qu'une seule thèse à défendre dans ce travail, elle serait que le calcul bleu fournit une base mathématique adéquate pour la conception d'un langage de programmation distribué de haut-niveau, fortement typé. Pour accréditer cette thèse, nous montrons dans ce manuscrit de quelle manière les différents modèles de programmation – et de typage – des langages fonctionnels, impératifs, concurrents et à objets peuvent s'intégrer dans le calcul bleu.

Dans la suite de cette introduction, nous présentons le schéma de la thèse en reprenant le découpage en quatre partie distinctes que nous avons choisit de suivre. Dans la partie I, nous

présentons le calcul bleu. Cette présentation se base sur des notions issues du λ-calcul et sur l'idée de machine abstraite d'exécution. Dans la partie II, nous présentons la notion d'équivalence choisie entre les processus et nous décrivons les principales égalités qui nous intéressent. Les parties III et IV sont respectivement dédiées à l'étude du typage dans le calcul bleu, et à l'interprétation du modèle de programmation à objets dans ce calcul.

Partie I: présentation du calcul bleu

Nous allons tenter de donner une intuition du calcul bleu en partant d'un autre calcul, dont nous supposons la sémantique bien maîtrisée par le lecteur, le λ-calcul. Pour ce faire, nous définissons une machine abstraite pour le λ-calcul, comparable à la machine de Krivine mais sans utilisation des fermetures, et nous modifions cette machine pour lui ajouter des «caractéristiques concurrentes». Le résultat final fournit une idée assez fidèle du calcul bleu.

Dans l'approche fonctionnelle, un programme est une fonction, ou de façon idéalisée, un terme donné par la grammaire suivante:

$$M,N ::= x \mid \lambda x.M \mid (MN)$$

où x est une variable (on suppose qu'il en existe un ensemble dénombrable); $\lambda x.M$ représente la fonction de paramètre x et de corps M; et (MN) représente l'application de la fonction M à l'argument N. Nous enrichissons ce calcul avec un ensemble de constantes $a_1,\ldots,a_n$, que nous appelons des *noms*, et qui peuvent intuitivement s'interpréter comme étant des adresses de ressources ou d'emplacements mémoires. Nous pouvons décrire un modèle d'exécution très simple des «programmes» écrit avec cette syntaxe, qui se base sur la notion de machine abstraite. Nous montrons ensuite comment, en modifiant notre machine abstraite fonctionnelle, nous aboutissant à une *machine chimique concurrente* pour le calcul bleu.

Notre machine abstraite pour le λ-calcul (avec noms) est définie par un ensemble de *configurations* $\mathcal{K}$, et de *règles de transitions* $\mathcal{K} \to \mathcal{K}'$. Dans notre formalisme, une configuration est un triplet $\{\mathcal{E};M;\mathcal{S}\}$ tel que: $\mathcal{E}$ représente la mémoire – le *store* en anglais, on dira aussi un *environnement* – qui est une association entre noms et programmes; M est le programme à exécuter, c'est-à-dire un λ-terme; $\mathcal{S}$ est une pile qui contient les arguments des appels aux fonctions. La mémoire de la machine abstraite est initialement vide, on note cet état par le symbole ϵ, et elle peut-être modifiée en ajoutant une nouvelle *déclaration*, c'est l'opération $(\mathcal{E} \mid \langle n = M\rangle)$. La pile à une structure similaire, mis à part qu'on mémorise des noms au lieu de déclarations.

$$\mathcal{E} ::= \epsilon \mid (\mathcal{E} \mid \langle a = N\rangle) \qquad\qquad \mathcal{S} ::= \epsilon \mid (a,\mathcal{S})$$

Une exécution de la machine abstraite, dans le cas où on cherche à évaluer le terme M par exemple, commence dans la configuration initiale $\mathcal{K}_0$, qui contient une mémoire et une pile vide: $\mathcal{K}_0 = \{\epsilon;M;\epsilon\}$. L'exécution de la machine est alors décrite comme une suite de pas de calculs élémentaires: $\mathcal{K}_0 \to \mathcal{K}_1 \to \ldots$

Définition 1.1 (Machine abstraite fonctionnelle) Les transitions de la machine abstraite fonctionnelle sont définies par les relations suivantes. L'opération $M\{^a/_x\}$, qui dénote la substitution de la variable x par le nom a, est définie de la manière usuelle.

$$\begin{array}{rcll} \{\mathcal{E};a;\mathcal{S}\} & \to & \{\mathcal{E};N;\mathcal{S}\} & (\text{si } \mathcal{E} = \cdots \mid \langle a = N\rangle \mid \cdots) \\ \{\mathcal{E};\lambda x.M;(a,\mathcal{S})\} & \to & \{\mathcal{E};M\{^a/_x\};\mathcal{S}\} & \\ \{\mathcal{E};(MN);\mathcal{S}\} & \to & \{(\mathcal{E} \mid \langle a = N\rangle);M;(a,\mathcal{S})\} & (\text{où } a \text{ est un nouveau nom}) \end{array}$$

Ainsi, pour évaluer une application, nous mémorisons l'argument dans un nouvel emplacement mémoire, et nous ajoutons le nom de cet emplacement dans la pile.

Nous pouvons modifier notre machine pour obtenir un «modèle d'exécution» plus riche. Pour commencer, nous pouvons faire quelques aménagements d'ordre syntaxique. Ainsi, à chaque pas de calcul la machine est dans une configuration de la forme:

$$\mathcal{K}_n = \{(\langle \mathrm{a}_0 = N_0 \rangle \mid \cdots \mid \langle a_n = N_n \rangle); M; (a_{i_1}, \ldots, a_{i_k})\} \tag{1.1}$$

où les noms $a_0, \ldots, a_n$ sont tous distincts. Si nous remplaçons la pile par une suite d'applications, la configuration générique $\mathcal{K}_n$ peut se réécrire:

$$\langle a_0 = N_0 \rangle \mid \cdots \mid \langle a_n = N_n \rangle \mid (M a_{i_1} \ldots a_{i_k}) \tag{1.2}$$

et les règles de transition de la définition 1.1 peuvent être redéfinies de la manière suivante.

$$\begin{array}{rcll} \langle a = N \rangle \mid \cdots \mid (a\, a_1 \ldots a_n) & \to & \langle a = N \rangle \mid \cdots \mid (N a_1 \ldots a_n) & (\rho) \\ \mathcal{K} \to \mathcal{K}' & \Rightarrow & (\langle a = N \rangle \mid \mathcal{K}) \to (\langle a = N \rangle \mid \mathcal{K}') & (\,\mid\,) \\ (\lambda x.M) a_1 \ldots a_n & \to & M\{a_1/x\} a_2 \ldots a_n & (\beta) \\ (MN) a_1 \ldots a_n & \to & \langle a = N \rangle \mid M a\, a_1 \ldots a_n & (\varpi) \end{array}$$

La règle (β) correspond à une forme simplifiée de béta-réduction, où on substitue un nom – plutôt qu'un terme – à une variable, tandis que la règle (ρ) peut s'interpréter comme une forme de communication. Cependant, à la place d'un modèle basé sur l'envoi de message comme le π-calcul, la communication est représentée ici comme un cas particulier d'accès à une ressource. Pour obtenir un modèle équivalent à celui de la «machine fonctionnelle», il nous faut aussi supposer que, dans la règle (ϖ), le nom a est nouveau.

Chaque configuration de la machine correspond à un terme du λ-calcul. La configuration $\mathcal{K}_n$ donnée dans (1.1), par exemple, correspond au terme $(M a_{i_1} \ldots a_{i_k})\{N_0/a_0\} \ldots \{N_n/a_n\}$. Par conséquent, chaque extension de la machine d'exécution correspond à une généralisation du λ-calcul. Dans la suite de cette partie, nous modifions la machine pour nous rapprocher du «modèle d'exécution» du π-calcul.

Ajout du non-déterminisme. Une première extension possible de notre modèle est de considérer le constructeur $\mid$ comme un opérateur de composition associatif, proche de la composition parallèle du π-calcul, et d'autoriser plusieurs configurations en parallèles. En particulier, nous modifions la notion de configuration pour inclure des exemples tel que $(\mathcal{K}_1 \mid \mathcal{K}_2)$. Nous nous libérons aussi de la contrainte (implicite) que les déclarations $\langle a \Leftarrow N \rangle$, présentent dans la mémoire, sont toutes de noms différents. Néanmoins nous ne choisissons pas un opérateur de composition parallèle commutatif car, pour chaque configuration, nous voulons conserver la possibilité de différencier la partie exécution de la partie environnement. Intuitivement, nous voulons permettre certaine commutation, avec la restriction que le terme M – qu'on cherche à évaluer – reste toujours à droite de la composition parallèle «la plus extérieure». La solution choisie est de définir les règles suivantes pour l'opérateur de composition parallèle, dans lesquelles $M \leftrightarrows N$ signifie que $M \to N$ et $N \to M$ sont vraies, et qui énoncent que $\mid$ est associatif et semi-commutatif. On dira aussi *commutatif gauche*.

$$\begin{array}{rcl} (M_1 \mid M_2) \mid M_3 & \leftrightarrows & M_1 \mid (M_2 \mid M_3) \\ (M_1 \mid M_2) \mid M_3 & \leftrightarrows & (M_2 \mid M_1) \mid M_3 \end{array}$$

Nous obtenons alors un opérateur de composition parallèle dissymétrique, comme dans la définition de CML [45] par exemple, tel que:

$$(\cdots \mid \langle a_1 = N_1 \rangle \mid \langle a_2 = N_2 \rangle \mid \cdots \mid M) \leftrightarrows (\cdots \mid \langle a_2 = N_2 \rangle \mid \langle a_1 = N_1 \rangle \mid \cdots \mid M)$$

L'ajout de la composition parallèle nous permet alors de simplifier la règle (ρ) de la manière suivante:

$$\langle a = N \rangle \mid (a a_1 \ldots a_n) \to \langle a = N \rangle \mid (N a_1 \ldots a_n) \qquad (\rho)$$

Nous avons donc transformé notre machine d'exécution fonctionnelle, en une machine chimique. Nous faisons ici référence à la notion de *machine abstraite chimique*, définie dans [11], qui a été utilisée par les auteurs pour «implanter» différents modèles de calculs concurrents et asynchrones. De plus, comme nous ajoutons la possibilité d'avoir plusieurs déclarations sur le même nom, flottants en solution, le modèle d'exécution que nous obtenons est plus puissant que celui du λ-calcul, puisque nous avons des réductions non-déterministes.

$$\begin{array}{lcl} \langle a = N_1\rangle \mid \langle a = N_2\rangle \mid a & \stackrel{*}{\to} & \langle a = N_1\rangle \mid \langle a = N_2\rangle \mid N_1 \\ \langle a = N_1\rangle \mid \langle a = N_2\rangle \mid a & \stackrel{*}{\to} & \langle a = N_1\rangle \mid \langle a = N_2\rangle \mid N_2 \end{array}$$

Pour l'instant, le modèle d'exécution que nous obtenons peut-être comparé à celui de Prolog [125]. Intuitivement, on peut comparer les termes de la forme $(Na_1 \dots a_n)$ à des buts, et les déclarations $\langle a = N\rangle$ à des règles. On se retrouve alors dans la situation où plusieurs règles différentes existent pour le même but. Néanmoins, si nous voulons obtenir un modèle d'exécution aussi riche que celui du π-calcul, il manque deux mécanismes importants à notre système.

Ajout des ressources. Le premier mécanisme correspond à la possibilité pour une ressource de disparaître, de la même manière qu'un récepteur du π-calcul est consommé par une communication. Nous pouvons modéliser ce mécanisme en ajoutant un nouveau type de déclarations, $\langle a \Leftarrow P\rangle$, et une nouvelle règle de communication:

$$\langle a \Leftarrow N\rangle \mid (aa_1 \dots a_n) \to (Na_1 \dots a_n)$$

La déclaration $\langle a = N\rangle$ s'interprète alors comme une infinité de déclarations $\langle a \Leftarrow P\rangle$ en parallèle, et nous obtenons un modèle qui se rapproche du λ-calcul avec ressources [19, 79, 18], introduit par G. Boudol afin de capturer les phénomènes de blocage de l'évaluation, ou «deadlock», qui sont propres aux exécutions concurrentes. Intuitivement, la déclaration $\langle a \Leftarrow P\rangle$ fournit un moyen de contrôler explicitement le nombre d'accès aux ressources de nom a.

Ajout de la restriction. Le second mécanisme correspond à la possibilité de créer dynamiquement des nouveaux noms. Il peut être facilement introduit dans notre machine chimique grâce à l'ajout de l'opérateur de restriction du π-calcul : $(\boldsymbol{\nu} a)\mathcal{K}$, et à l'ajout de nouvelles règles de réduction.

$$\begin{array}{rcll} (MN)a_1 \dots a_n & \to & (\boldsymbol{\nu} a)(\langle a = N\rangle \mid Maa_1 \dots a_n) & (a \text{ nouveau nom}) \\ \langle a_2 = N\rangle \mid (\boldsymbol{\nu} a_1)\mathcal{K} & \to & (\boldsymbol{\nu} a_1)(\langle a_2 = N\rangle \mid \mathcal{K}) & (a_1 \text{ non libre dans } \langle a_2 = N\rangle) \\ \mathcal{K} \to \mathcal{K}' & \Rightarrow & (\boldsymbol{\nu} a)\mathcal{K} \to (\boldsymbol{\nu} a)\mathcal{K}' & \end{array}$$

Le calcul bleu résulte de ces ajouts à l'interprétation «chimique» de la machine fonctionnelle, avec la possibilité supplémentaire de définir plusieurs configurations en parallèles. En particulier il inclut le λ-calcul de façon directe. Il possède aussi les caractéristiques du π-calcul : il inclut le non-déterminisme et le parallélisme, il possède un opérateur de restriction, et son modèle de communication est basé sur l'échange de noms. Cependant il existe aussi des différences entre ces deux calculs. Ainsi, dans le calcul bleu, il est possible de définir des fonctions d'ordre supérieur et d'appliquer des processus à des noms.

Dans la première partie de cette thèse, nous introduisons le calcul bleu plus formellement, et nous montrons qu'il est aussi expressif que le π-calcul. Dans la suite, nous étudions le modèle de programmation associé au calcul bleu. Le plan de cette thèse se divise idéalement en quatre parties. Comme nous l'avons dit, la première partie est consacrée à la présentation du calcul bleu, nous détaillons les trois autres parties dans les sections suivantes.

Partie II: Équivalence

Dans la seconde partie de cette thèse, nous définissons une relation d'équivalence comportementale ($\approx_b$) entre termes du calcul bleu, qui est basée sur la notion classique de *congruence à*

barbes [92]. Dans cette partie nous démontrons la validité des *lois de réplication*. Intuitivement, ces lois énoncent qu'une ressource privée et répliquée peut être distribuée sans danger entre plusieurs «acteurs». Par exemple, si nous notons $(\mathbf{def}\, a = P \,\mathbf{in}\, Q)$ l'opérateur dérivé défini par le processus $(\boldsymbol{\nu} a)(\langle a = P\rangle \mid Q)$, nous montrons que:

$$\mathbf{def}\, a = P \,\mathbf{in}\, (Q \mid R) \;\approx_b\; (\mathbf{def}\, a = P \,\mathbf{in}\, Q) \mid (\mathbf{def}\, a = P \,\mathbf{in}\, R)$$

Intuitivement, on montre que l'opérateur **def** vérifie des lois équivalentes à celles des substitutions explicites dans le λ-calcul [1]. Néanmoins, les preuves directes pour la congruence à barbes sont très complexes, car elles demandent de considérer une quantification sur tous les contextes. Aussi, pour éviter cet écueil, nous définissons un système de transitions étiquetées qui simule la réduction dans le calcul bleu, et une bisimulation associée à ce système ($\sim_d$).

Pour simplifier les preuves avec la bisimulation étiquetée, nous définissons et nous prouvons la validité des techniques de preuve «up-to» [113]. En employant ces techniques de preuve, nous montrons trois propriétés de la bisimulation: (i) c'est une congruence; (ii) elle contient l'équivalence structurelle; et (iii) elle vérifie les lois de réplication. Ces propriétés permettent de prouver, dans le chapitre 8, que la bisimulation est incluse dans la congruence à barbes. Comme la bisimulation $\sim_d$ est plus fine que la congruence à barbes, les lois valides pour $\sim_d$ sont donc aussi des lois de $\approx_b$.

Partie III: Types

Nous cherchons à prouver dans cette thèse que le calcul bleu fournit un bon modèle pour la conception d'un langage de programmation distribué de haut-niveau, de la même manière que le λ-calcul sert de fondement à la conception de certains langages séquentiels, comme ML par exemple. Or les systèmes de types du λ-calcul sont à la base des systèmes de types pour les langages séquentiel. Il est donc naturel de s'intéresser aux types que nous pouvons donner aux processus.

Dans cette partie, nous définissons quatre systèmes de types pour le calcul bleu. Le premier de ces systèmes `B`, est similaire au système de types simples du λ-calcul défini par Curry. En particulier le typage des termes est *implicite*, dans le sens où les types n'apparaissent pas dans la syntaxe des termes. Les trois autres systèmes sont obtenus par extensions successives de `B`. Ainsi, nous étudions un système avec sous-typage $\mathtt{B}_{\leqslant}$; un système avec *polymorphisme paramétrique*, comparable au système de types de Hindley-Milner pour ML ; et un système avec *polymorphisme contraint* $\mathtt{BF}_{\leqslant}$, c'est-à-dire un système avec récursion, types d'ordre supérieur et une forme particulière de quantification bornée, qui est approprié à l'étude du typage des langages à objets.

Chacun de ces systèmes de types possède la propriété de *conservation du typage* – appelée aussi *subject reduction* en anglais – qui énonce que le type d'un terme est conservé après une réduction. Cette propriété, même si elle n'est pas suffisante en elle même, permet de prouver qu'un processus bien typé ne peut pas provoquer d'erreur à l'exécution. Nous prouvons la propriété de conservation du typage pour le système $\mathtt{BF}_{\leqslant}$, qui représente le cas le plus complexe. Nous étudions aussi le problème de la décidabilité du problème de l'inférence de type dans le système avec polymorphisme paramétrique.

Partie IV: Objets

Dans la dernière partie, nous étudions les liens entre le calcul bleu et le modèle de programmation à objets. Nous étudions le calcul d'objets fonctionnel de M. Abadi et L. Cardelli, et plus précisément $\mathbf{Ob}_{1<:}$, qui est une version de ce calcul avec un système de types simples et du sous-typage. Une présentation plus approfondie de ce calcul d'objets est faîte au chapitre 16.

Dans le chapitre 17, nous nous inspirons des opérateurs de ce calcul pour définir un calcul d'objets concurrents. Nous prouvons que ce calcul se dérive simplement dans le calcul bleu, et

nous donnons un système de types dérivés pour les objets. Le typage des objets est basé sur le système de type du premier ordre avec sous-typage ($\mathtt{B}_{\leqslant}$), que nous avons introduit dans la partie III.

Dans le chapitre 18, nous vérifions l'expressivité de nos opérateurs d'objets concurrents en donnant une interprétation de $\mathbf{Ob}_{1<:}$ qui préserve le typage et le sous-typage. Cette interprétation est fondamentalement basée sur l'opérateur de dénomination: $\langle a \mapsto [\, l_i = (\boldsymbol{\lambda} x_i) M_i{}^{i\in[1..n]} \,]\rangle$, qui est l'équivalent pour les objets, de ce que la déclaration $\langle a = M\rangle$ est aux fonctions du λ-calcul. Intuitivement nous montrons que, alors que les fonctions correspondent à des ressources non-modifiables et infiniment disponible ($\langle a = M\rangle$), et alors que les récepteurs du π-calcul correspondent à des ressources consomables ($\langle a \Leftarrow M\rangle$), les objets correspondent à des ressources «gérées linéairement». Nous donnons aussi une interprétation du calcul d'objets concurrents défini par A. Gordon et P. Hankin dans [60], qui est une version concurrente et impérative de $\mathbf{Ob}_{1<:}$.

Dans le chapitre 20, nous nous intéressons à un autre modèle des objets, introduit par K. Fisher *et al.* [46, 49], dans lequel on peut étendre l'ensemble des méthodes d'un objet. L'intérêt du calcul d'objets avec extension, dénoté $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$, est qu'il permet de simuler le mécanisme *d'héritage* des langages orientés objets: l'ajout de méthode permet de «réutiliser du code» déjà existant pour définir de nouveaux objets plus complexes. Nous appliquons au calcul $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ un programme similaire à celui réservé au calcul $\mathbf{Ob}_{1<:}$, en donnant une interprétation – opérationnellement correcte – qui préserve les notions de typage et de sous-typage.

Informations utiles

Tous les éléments numérotés de cette thèse – figures, définitions, théorèmes, ... – sont numérotés selon le schéma *c.n*, où *c* représente le chapitre dans lequel l'élément se trouve. Nous avons choisit de numéroter les conjectures, lemmes, propositions et théorèmes, en utilisant un compteur commun. Les chapitres, quant à eux, sont numérotés de manières uniforme sur l'ensemble des quatre parties.

Première partie

Présentation du calcul bleu

CHAPITRE 2

Définition du calcul bleu

> Le langage fournit les outils abstraits utilisés pour résoudre des problèmes. Ainsi, les avancées de la technique rendent possible la résolution des problèmes par les machines, alors que les avancées du langage rendent possible la résolution des problèmes par l'homme.

DANS CE CHAPITRE nous présentons la syntaxe ainsi que la sémantique opérationnelle du calcul bleu. Comme le π-calcul, le calcul bleu est un modèle des exécutions concurrentes basé sur la notion de nommage. C'est-à-dire que l'élément de base, dans la construction des processus, est le nom. Les noms représentent à la fois l'endroit où se trouvent les ressources: leurs localisations, ainsi que les valeurs échangées au cours d'une communication. Nous serons amenés, tout au long de cette thèse, à considérer plusieurs catégories de noms selon l'usage qui en est fait. Par exemple, nous différencierons les noms de canaux de communications, ou références, des constantes utilisées pour nommer les champs d'un enregistrement.

Nous commençons en donnant la syntaxe du calcul bleu, qui est dénoté par le symbole $\boldsymbol{\pi}^\star$. La syntaxe que nous avons choisie n'est pas gravée dans la pierre, et il existe diverses possibilités dans le choix des opérateurs du calcul. Ces choix sont discutés dans la section 2.3. Disons cependant que, quelque soit les choix envisagés, l'expressivité du calcul est conservée. Dans la section 2.2, nous donnons la sémantique opérationnelle de $\boldsymbol{\pi}^\star$. Celle-ci est définie en terme de réduction, mais une définition utilisant une relation de transitions étiquetées est donnée dans la partie II. Nous terminons ce chapitre par quelques exemples de processus.

2.1 Syntaxe

On admet l'existence d'un ensemble dénombrable $\mathcal{N}$, de *noms*, dont les éléments sont désignés par a,b,c.... Pour faciliter la présentation du calcul bleu, et plus précisément de sa version locale qui est introduite dans la section 2.3.2, il est nécessaire de distinguer différentes catégories de noms. En particulier, nous distinguons un sous-ensemble de noms, qui représente les *étiquettes* et qui sont utilisés pour nommer les champs d'un enregistrement. Cet ensemble est noté $\mathcal{L}$ par la suite, et nous utilisons les lettres k,l,m... pour désigner les éléments de $\mathcal{L}$.

Nous emploierons aussi les lettres x,y,z... pour désigner un nom qui est utilisé comme une variable: c'est le cas des noms liés par une abstraction par exemple, et les lettres u,v,w..., lorsqu'un nom est utilisé comme un nom de canal. Dans ce cas, nous parlons aussi de *référence*. Ainsi, les noms liés par une restriction sont des références. On pourra supposer l'existence d'un système de types, ou de contraintes syntaxiques, pour vérifier que l'usage d'un nom est cohérent avec l'une

de ces trois catégories. Nous ne donnons pas les détails d'un tel système ici, et nous ne ferons pas, pour le moment, de différence entre variables et références.

La syntaxe du calcul bleu est donnée dans la figure 2.1. On remarque la présence des opérateurs du λ-calcul : l'abstraction $(\boldsymbol{\lambda}a)P$, et l'application (Pa), cette dernière étant restreinte à l'application d'un terme à un nom. Le calcul bleu contient aussi des opérateurs du π-calcul [91, 97]: la composition parallèle $(P \mid Q)$, permet d'exécuter les deux processus P et Q en parallèle; la restriction $(\boldsymbol{\nu}a)P$, permet de restreindre la portée du nom a à P, une notion semblable à la définition des variables locales dans les langages de programmation.

Dans le calcul bleu, les opérateurs de réception et de réplication du π-calcul sont absents, à la faveur des opérateurs de déclarations: la déclaration $\langle a \Leftarrow P\rangle$, est une ressource qui attend un message sur le nom a, et qui permet de libérer le processus P. Une fois consommée, la déclaration disparaît; la déclaration répliquée $\langle a = P\rangle$ a le même comportement, mis à part pour l'accès, qui n'est pas destructif. Finalement, le reste des opérateurs introduits dans la figure 2.1, sont les constructeurs usuels de définition des enregistrements. On peut trouver quelques commentaires sur ces opérateurs à la section 2.1.3.

Par la suite, nous employons des abréviations usuelles dans les algèbres de processus. Ainsi, lorsque la taille d'une séquence de noms est connue (ou insignifiante), nous notons $\tilde{a}$ le tuple $(a_1,\ldots,a_n)$. Par abus, on pourra considérer $\tilde{a}$ aussi bien comme un tuple, que comme un ensemble, et la concaténation de tuples: $\tilde{a}$,$\tilde{b}$, comme une union ensembliste. Nous notons $(Pa_1 \ldots a_n)$, ou encore $(P\tilde{a})$, la suite d'applications $(\ldots(Pa_1)\ldots a_n)$. Nous notons $(\boldsymbol{\lambda}\tilde{a})P$ la suite d'abstractions $(\boldsymbol{\lambda}a_1)\ldots(\boldsymbol{\lambda}a_n)P$, et $(\boldsymbol{\nu}\tilde{a})P$ le processus $(\boldsymbol{\nu}a_1)\ldots(\boldsymbol{\nu}a_k)P$. Le symbole ϵ est utilisé pour noter le tuple vide.

Remarque (Conventions) Par souci de lisibilité, on utilisera des mots, plutôt que des lettres minuscules, pour désigner certaines références ou étiquettes. On choisira alors de les écrire en caractères sans sérif: `put`, `get`, ... De même, lorsqu'on voudra nommer certains processus, on choisira des caractères gras: **POR**, **BUFF**, ... ■

On peut remarquer l'absence dans $\boldsymbol{\pi}^\star$ d'un constructeur équivalent au processus inerte, souvent noté $\mathbf{0}$, du π-calcul. Toutefois, on peut exhiber des exemples de processus inertes. Pour simplifier la présentation de certains résultats, nous introduisons un opérateur dérivé pour $\mathbf{0}$ dans le calcul bleu.

Définition 2.1 (Nil) On note $\mathbf{0}$, prononcez nil, le processus $(\boldsymbol{\nu}u)\langle u = u\rangle$.

Une différence avec le π-calcul, est l'introduction des enregistrements comme opérateurs primitifs de notre calcul. Dans les calculs de processus, on considère généralement que les enregistrements sont obtenus par codage. Nous dérogeons ici à cette habitude, car la présence des enregistrements est un moyen de faciliter l'étude des objets: la partie IV est totalement dévolue à cette étude. C'est aussi un choix évident, dès lors qu'on s'intéresse à une application de nos travaux à l'étude des langages de programmation.

L'introduction des enregistrements a comme conséquence d'introduire une notion d'erreur d'exécution: par exemple si on cherche à sélectionner un champ manquant dans un enregistrement, comme dans le terme: $([\,]\cdot l)$. Cette situation est en partie comparable à l'ajout des communications polyadique à $\boldsymbol{\pi}$, et comme pour le π-calcul, ce problème est résolu par l'addition d'un système de types au calcul. C'est ce qui constitue la partie III de cette thèse.

2.1.1 Noms libres

Le calcul bleu possède deux lieurs. Le premier est l'abstraction $(\boldsymbol{\lambda}a)P$, comme dans le λ-calcul, et le second est la restriction $(\boldsymbol{\nu}a)P$, comme dans le π-calcul. Ainsi, nous définissons pour chaque

P,Q	::=	a	nom
	$\mid$	$(\boldsymbol{\lambda} a)P$	abstraction
	$\mid$	(Pa)	application
	$\mid$	$(P \mid Q)$	composition parallèle
	$\mid$	$(\boldsymbol{\nu} a)P$	restriction
	$\mid$	$\langle a \Leftarrow P \rangle$	déclaration
	$\mid$	$\langle a = P \rangle$	déclaration répliquée
	$\mid$	$(P \cdot l)$	sélection
	$\mid$	$[\,]$	enregistrement vide
	$\mid$	$[\,P\,, l = Q\,]$	extension/modification

Fig. 2.1: Syntaxe des processus bleus

processus P, les ensembles $\mathbf{fn}(P)$ et $\mathbf{bn}(P)$ qui sont, respectivement, les noms libres et liés de P.

Définition 2.2 (Noms libres et noms liés) Pour chaque processus P, on peut calculer l'ensemble $\mathbf{fn}(P)$ de ses noms libres, et l'ensemble $\mathbf{bn}(P)$ de ses noms liés. Par analogie avec la définition usuelle de ces ensembles, on ne tient pas compte des noms d'étiquettes.

$$\mathbf{fn}(a) =_{\text{def}} \{a\} \qquad \mathbf{fn}([\,]) =_{\text{def}} \emptyset$$
$$\mathbf{fn}(Pa) =_{\text{def}} \mathbf{fn}(\langle a \Leftarrow P \rangle) =_{\text{def}} \mathbf{fn}(\langle a = P \rangle) =_{\text{def}} \mathbf{fn}(P) \cup \{a\}$$
$$\mathbf{fn}((\boldsymbol{\nu} a)P) =_{\text{def}} \mathbf{fn}((\boldsymbol{\lambda} a)P) =_{\text{def}} \mathbf{fn}(P) \setminus \{a\}$$
$$\mathbf{fn}(P \mid Q) =_{\text{def}} \mathbf{fn}([\,P\,, l = Q\,]) =_{\text{def}} \mathbf{fn}(P) \cup \mathbf{fn}(Q)$$
$$\mathbf{fn}(P \cdot l) =_{\text{def}} \mathbf{fn}(P)$$

$$\mathbf{bn}(a) =_{\text{def}} \mathbf{bn}([\,]) =_{\text{def}} \emptyset$$
$$\mathbf{bn}(Pa) =_{\text{def}} \mathbf{bn}(\langle a \Leftarrow P \rangle) =_{\text{def}} \mathbf{bn}(\langle a = P \rangle) =_{\text{def}} \mathbf{bn}(P \cdot l) =_{\text{def}} \mathbf{bn}(P)$$
$$\mathbf{bn}((\boldsymbol{\nu} a)P) =_{\text{def}} \mathbf{bn}((\boldsymbol{\lambda} a)P) =_{\text{def}} \mathbf{bn}(P) \cup \{a\}$$
$$\mathbf{bn}(P \mid Q) =_{\text{def}} \mathbf{bn}([\,P\,, l = Q\,]) =_{\text{def}} \mathbf{bn}(P) \cup \mathbf{bn}(Q)$$

Notez que le nom a n'est pas lié par les déclarations $\langle a \Leftarrow P \rangle$ et $\langle a = P \rangle$. Nous supposons définie l'opération de substitution d'un nom a par un nom b, qui est noté $P\{^b/_a\}$. Les détails de la définition de $P\{^b/_a\}$ sont omis ici. La substitution peut nécessiter le renommage de certains noms liés, une opération appelée α-conversion. On dénote $P =_\alpha Q$ la congruence engendrée par cette forme de renommage, et par la suite, nous omettrons souvent de distinguer les termes α-convertible. Nous utiliserons aussi parfois une notion de substitution d'un terme à une variable, opération notée $P\{\!\{^Q/_x\}\!\}$. Encore une fois, nous ne donnons pas les détails de cette opération. Parfois, on aura également besoin de connaître l'ensemble des noms déclarés par un processus. C'est-à-dire les noms qui apparaissent en partie objet d'une déclaration:

Définition 2.3 (Noms déclarés) Pour chaque processus P, on peut définir l'ensemble $\mathbf{decl}(P)$ des noms déclarés par P, qui représente l'ensemble des références utilisées comme sujet d'une déclaration. Cet ensemble est donné par la définition suivante.

$$\mathbf{decl}(\langle a \Leftarrow P \rangle) =_{\text{def}} \mathbf{decl}(\langle a = P \rangle) =_{\text{def}} \mathbf{decl}(P) \cup \{a\}$$
$$\mathbf{decl}(a) =_{\text{def}} \mathbf{decl}([\,]) =_{\text{def}} \emptyset$$
$$\mathbf{decl}(Pa) =_{\text{def}} \mathbf{decl}(P \cdot l) =_{\text{def}} \mathbf{decl}(P)$$
$$\mathbf{fn}((\boldsymbol{\nu} a)P) =_{\text{def}} \mathbf{decl}((\boldsymbol{\lambda} a)P) =_{\text{def}} \mathbf{decl}(P) \setminus \{a\}$$
$$\mathbf{fn}(P \mid Q) =_{\text{def}} \mathbf{decl}([\,P\,, l = Q\,]) =_{\text{def}} \mathbf{decl}(P) \cup \mathbf{decl}(Q)$$

2.1.2 Contextes

Un contexte du calcul bleu est un terme bâti à partir de la syntaxe de $\boldsymbol{\pi}^\star$, étendu avec la constante $[\_]$. Cette constante représente le «trou» dans lequel on peut placer un processus. On utilise les caractères $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ ... pour les contextes, et on note $\mathbf{C}[P]$ le résultat obtenu en comblant le trou de $\mathbf{C}$ par P. Dans cette opération, certains noms de P peuvent se retrouver liés. Si on étend la définition de $\mathbf{fn}(.)$ et $\mathbf{bn}(.)$ aux contextes, il est facile de voir que l'ensemble des noms de P, capturés par $\mathbf{C}$, est inclus dans $\mathbf{fn}(P) \cap \mathbf{bn}(\mathbf{C})$.

Dans les sections suivantes, nous utiliserons un type particulier de contextes que nous appellerons *contextes d'évaluation*. Ce sont les contextes tels que le trou n'apparaît pas sous une λ-abstraction, une déclaration ou une extension. Plus formellement, on a la définition suivante.

Définition 2.4 (Contextes d'évaluation) L'ensemble des contextes d'évaluation, noté $\mathcal{E}$, est l'ensemble engendré par la grammaire suivante:

$$\mathbf{E} ::= [\_] \mid (\mathbf{E}a) \mid (\mathbf{E} \cdot l) \mid (\mathbf{E} \mid P) \mid (P \mid \mathbf{E}) \mid (\boldsymbol{\nu} a)\mathbf{E}$$

où P est un processus quelconque.

Nous définissons aussi quelques conventions de notation pour les contextes. Ainsi, on désigne par les symboles $\mathbf{E}$, $\mathbf{F}$ ... les contextes d'évaluation, et on pourra noter $(\mathbf{C} \circ \mathbf{D})$ la composition des deux contextes $\mathbf{C}$ et $\mathbf{D}$, c'est-à-dire que: $(\mathbf{C} \circ \mathbf{D})[P] =_{\text{def}} \mathbf{C}[\mathbf{D}[P]]$.

2.1.3 Enregistrements

Les enregistrements sont construits incrémentalement à partir de l'enregistrement vide $[\,]$, en utilisant une opération: $[\,P\,, l = Q\,]$, qui peut être interprétée comme l'extension / la modification du champ l dans l'enregistrement P. Ainsi, notre modèle est comparable à celui défini par M. Wand [140], puisque nous combinons dans une seule construction ces deux opérations.

Une notation classique, que nous adoptons ici, est d'écrire $[\,l_1 = P_1, \ldots, l_n = P_n\,]$, le terme $[\,[\,[\,]\,, l_1 = P_n\,], \ldots l_n = P_n\,]$. Ceci seulement lorsque les champs $(l_i)_{i \in [1..n]}$ sont distincts. Parfois, nous noterons (Pl) la sélection $(P \cdot l)$. Cet abus correspond à une intuition très simple: un enregistrement est une fonction partielle des champs vers les processus, et la sélection correspond à l'application. Ainsi, dans la suite, le terme (Pa) pourra dénoter aussi bien une application (Pu), qu'une sélection $(P \cdot l)$.

2.2 Sémantique opérationnelle

La sémantique opérationnelle du calcul bleu est présentée suivant le modèle de la *machine chimique* [11, 20]. La réduction est alors factorisée en deux notions: une relation de réduction, $P \to P'$, qui décrit l'évolution du processus P vers le processus P', et qui représente la partie calcul d'une réduction; une relation d'équivalence structurelle, $P \equiv P'$, qui permet de réarranger les termes afin de mettre en contact les parties «réactives» d'un processus. Cette dernière relation permet de se représenter chaque terme comme une «solution chimique», ou une soupe, de processus élémentaires.

L'utilisation de ce type de présentation pour les calculs de processus, a été popularisée par R. Milner [97], qui l'utilise pour donner la sémantique du π-calcul. C'est aussi la présentation choisie par G. Boudol, dans son article [22] qui a introduit le calcul bleu.

2.2.1 Équivalence structurelle

La relation $\equiv$, est la plus petite congruence vérifiant les axiomes de la figure 2.2. Cette définition partage des points communs avec celle donné pour $\boldsymbol{\pi}$, en particulier la règle d'extension de la

portée, dite de «*scope extrusion*» en anglais. Il faut noter, cependant, que la composition parallèle n'est pas commutative dans notre calcul.

Nous ne sommes ni les premiers, ni les seuls, à avoir fait le choix d'un opérateur de composition parallèle dissymétrique. On retrouve ce choix, par exemple, dans la description de la sémantique de CML [45], et dans la définition du calcul d'objets concurrents de A. Gordon et P. Hankin [60]. On retrouve également cette idée dans l'implantation de la composition parallèle dans Pict [130, chap. 7] et JoCaml [85, 84].

Intuitivement, dans la composition parallèle $(P \mid Q)$, le processus Q est l'acteur principal du calcul. C'est lui qui peut interagir avec l'environnement extérieur. Ce rôle est visible, par exemple, dans la règle de distributivité. Nous reprenons en cela les conventions prises dans les exemples de calculs que nous venons de citer: l'idée est que Q représente le «thread» principal d'exécution, alors que P représente l'environnement. Cependant, contrairement au calculs séquentiels, la partie environnement est active: on peut y trouver des messages et des applications.

Nous reparlerons du choix d'un parallèle dissymétrique dans la section 2.3. En particulier, nous présenterons les arguments motivant notre choix. Arguments qui sont principalement ceux exposés par G. Boudol dans la conclusion de [22, sect. 7].

$P =_\alpha Q \implies P \equiv Q$		α-conversion
$((P_1 \mid P_2) \mid P_3) \equiv (P_1 \mid (P_2 \mid P_3))$		associativité
$((P_1 \mid P_2) \mid P_3) \equiv ((P_2 \mid P_1) \mid P_3)$		commutativité gauche
$(\boldsymbol{\nu} a)P \mid Q \equiv (\boldsymbol{\nu} a)(P \mid Q)$	$(a \notin \mathbf{fn}(Q))$	extension de la portée
$Q \mid (\boldsymbol{\nu} a)P \equiv (\boldsymbol{\nu} a)(Q \mid P)$	$(a \notin \mathbf{fn}(Q))$	
$(\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P \equiv (\boldsymbol{\nu} b)(\boldsymbol{\nu} a)P$		
$(P \mid Q)a \equiv P \mid (Qa)$		distributivité (sur le parallèle)
$(P \mid Q) \cdot l \equiv P \mid (Q \cdot l)$		
$((\boldsymbol{\nu} a)P)b \equiv (\boldsymbol{\nu} a)(Pb)$	$(a \neq b)$	distributivité (sur la restriction)
$((\boldsymbol{\nu} a)P) \cdot l \equiv (\boldsymbol{\nu} a)(P \cdot l)$		
$(\boldsymbol{\nu} a)P \equiv P$	$(a \notin \mathbf{fn}(P))$	«garbage collection»
$P \equiv Q \implies \mathbf{C}[P] \equiv \mathbf{C}[Q]$		congruence

Fig. 2.2: Équivalence structurelle

2.2.2 Réduction

Afin de simplifier notre raisonnement, on peut séparer les processus du calcul bleu en quatre catégories syntaxiques distinctes.

Définition 2.5 (Catégories syntaxique)

processus (ou termes)	P,Q	$::= A \mid D \mid R \mid (P \mid Q) \mid (\boldsymbol{\nu} a)P$
agents	A	$::= u \mid (\boldsymbol{\lambda} a)P \mid (Pa) \mid (P \cdot l)$
déclarations	D	$::= \langle a \Leftarrow P \rangle \mid \langle a = P \rangle$
enregistrements	R	$::= [\,] \mid [\,P\,, l = Q\,]$

Il est facile de montrer que les règles exposées dans la figure 2.2, permettent de réécrire tout processus P sous une forme particulière

$$P \equiv (\boldsymbol{\nu}\, \tilde{u})(P_1 \mid \cdots \mid P_n) \tag{2.1}$$

telle que la portée des noms privés a été étendue au maximum et que chaque application a été distribuée au maximum. On se retrouve alors avec une «soupe» de molécules élémentaires, qui sont soit des agents, soit des déclarations, soit des enregistrements. De plus on peut choisir de mettre les agents sous une forme simple[1]: $(\boldsymbol{\lambda} u)P)\tilde{a}$, ou $(R\,\tilde{a})$, ou $(u\,\tilde{a})$. Dans le dernier cas, on dit également que l'agent est un *message sur le canal* u. On peut aussi trouver des termes de la forme $(\langle u \Leftarrow P\rangle\, a)$, que nous considérons comme des termes dégénérés: ne se réduisant pas, et qui seront interdits par l'utilisation d'un système de types.

Définition 2.6 (Messages) Un message M, est un nom appliqué à une suite de noms ou de sélections. C'est un élément de l'ensemble généré par la grammaire suivante.

$$M ::= a \mid (Ma) \mid (M \cdot l)$$

On peut découper la définition de la relation de réduction, donné dans la figure 2.3, en trois parties, qui correspondent aux trois types de molécules élémentaires que l'on vient de définir.

Béta-réduction le premier type de réduction possible est la «petite» béta-réduction, qui permet de réduire un agent.

$$(\boldsymbol{\lambda} x)P)a \rightarrow_{(\beta)} P\{a/x\}$$

L'adjectif petite s'emploie ici pour signifier que l'on substitue seulement un nom à une variable. Ceci est à comparer au λ-calcul, où un terme est substitué à une variable: $(\lambda x.M)N \rightarrow M\{\!\{N/x\}\!\}$. On dit aussi que le λ-calcul est d'ordre supérieur.

Sélection la sélection est aux enregistrements, ce que la béta-réduction est aux agents. Cependant il existe deux cas, selon que le champ considéré est présent ou absent. Supposons que l soit différent de k, on a:

$$[\,P\,, l = Q\,] \cdot l \rightarrow_{(\beta)} Q \qquad\qquad [\,P\,, l = Q\,] \cdot k \rightarrow_{(\beta)} P \cdot k$$

Communication appelée aussi «*ressource fetching*» dans [22]. Comme dans la réduction de la machine abstraite définie dans l'introduction, une déclaration répliquée $\langle u = P\rangle$, est une *ressource* localisée en u qui peut réagir avec un message au cours d'une communication:

$$\begin{array}{l}\langle u = P\rangle \mid ua_1 \ldots a_n \rightarrow_{(\rho)} \langle u = P\rangle \mid Pa_1 \ldots a_n \\ ua_1 \ldots a_n \mid \langle u = P\rangle \rightarrow_{(\rho)} Pa_1 \ldots a_n \mid \langle u = P\rangle\end{array}$$

Dans le cas de la déclaration $\langle u \Leftarrow P\rangle$, la ressource est détruite au cours de la communication. Plus exactement, elle est remplacée par un processus inactif.

$$\begin{array}{l}\langle u \Leftarrow P\rangle \mid ua_1 \ldots a_n \rightarrow_{(\rho)} \mathbf{0} \mid Pa_1 \ldots a_n \\ ua_1 \ldots a_n \mid \langle u \Leftarrow P\rangle \rightarrow_{(\rho)} Pa_1 \ldots a_n \mid \mathbf{0}\end{array}$$

Il est facile de voir que l'ensemble des couples $((\mathbf{0} \mid P),P)$ forme une relation de bisimulation forte. Une relation $\mathcal{D}$ est une bisimulation forte si c'est une équivalence telle que, si $(P,Q) \in \mathcal{D}$ et $P \rightarrow P'$, alors il existe un processus Q' tel que $Q \rightarrow Q'$ et $(P',Q') \in \mathcal{D}$. On peut donc considérer que $(\mathbf{0} \mid P)$ est équivalent à P, ce qui permet de simplifier la première règle (cf. règle (red decl) de la figure 2.3).

$$\frac{P \to P' \qquad \mathbf{E} \in \mathcal{E}}{\mathbf{E}[P] \to \mathbf{E}[P']}\ \text{(red context)} \qquad \frac{P \to P' \qquad P \equiv Q}{Q \to P'}\ \text{(red struct)}$$

$$\frac{}{(\boldsymbol{\lambda} x)P)a \to P\{a/x\}}\ \text{(red beta)}$$

$$\frac{}{\langle u \Leftarrow P\rangle \mid (u\tilde{a}) \to (P\tilde{a})}\ \text{(red decl)} \qquad \frac{}{\langle u = P\rangle \mid (u\tilde{a}) \to \langle u = P\rangle \mid (P\tilde{a})}\ \text{(red mdecl)}$$

$$\frac{}{(u\tilde{a}) \mid \langle u \Leftarrow P\rangle \to (P\tilde{a}) \mid \mathbf{0}}\ \text{(red decl)} \qquad \frac{}{(u\tilde{a}) \mid \langle u = P\rangle \to (P\tilde{a}) \mid \langle u = P\rangle}\ \text{(red mdecl)}$$

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

Fig. 2.3: Réduction

Les règles de réduction sont rassemblées dans la figure 2.3. Nous excluons cependant de donner le symétrique des règles de communication des déclarations. Les règles qui s'ajoutent à celles déjà données, précisent que la réduction est compatible avec l'équivalence structurelle, et les contextes d'évaluations.

Les deux relations $\to_{(\beta)}$ et $\to_{(\rho)}$, qui correspondent respectivement à la béta-réduction plus la sélection, et à la communication, vérifient des propriétés simples.

Lemme 2.1 *La béta-réduction satisfait la «propriété du losange»: si $P \to_{(\beta)} P_0$ et si $P \to_{(\beta)} P_1$, alors soit $P_0 \equiv P_1$, soit il existe P' tel que: $P_0 \to_{(\beta)} P'$ et $P_1 \to_{(\beta)} P'$. De plus cette relation est* fortement normalisante*: pour tout terme P, il existe un terme Q tel que: $P \overset{*}{\to}_{(\beta)} Q$ et Q est* béta-irréductible*, c'est-à-dire $\{Q' \mid Q \to_{(\beta)} Q'\} = \emptyset$.*

On peut donc définir une *béta-forme normale* pour chaque processus. Il n'y a pas de résultat équivalent pour la communication, mais nous montrons qu'on peut toujours effectuer toutes les béta-réductions avant de communiquer.

Lemme 2.2 *La communication et la béta-réduction sont deux relations «indépendantes»: si $P \to_{(\rho)} P_0$ et $P \to_{(\beta)} P_1$, alors il existe un terme P' tel que: $P_0 \to_{(\beta)} P'$ et $P_1 \to_{(\rho)} P'$.*

Preuve (Lemmes 2.1 et 2.2) La preuve de ces résultats peut être trouvée dans [22]. Elle utilise le fait que la béta-réduction n'introduit jamais de nouveau rédex. □

Il est clair que c'est la communication qui est responsable de la «puissance» du calcul bleu. En effet elle est responsable du *non-déterminisme* et de la *non terminaison* du calcul. Le plus simple exemple de calcul infini est fourni, par exemple, par le terme suivant.

$$\Omega =_{\text{def}} (\boldsymbol{\nu} u)(\langle u = u\rangle \mid u) \to_{(\rho)} \Omega$$

Le non-déterminisme provient de deux sources de conflit possibles. Ainsi on peut avoir une «paire critique», faisant intervenir deux ressources localisées au même endroit. C'est le cas dans le codage du choix interne par exemple

$$(P \oplus Q) =_{\text{def}} (\boldsymbol{\nu} u)((\langle u \Leftarrow P\rangle \mid \langle u \Leftarrow Q\rangle) \mid u) \qquad \text{avec } u \notin \mathbf{fn}(P,Q)$$

En effet, on montre qu'après une réduction, le processus $(P \oplus Q)$ peut se comporter soit comme P, soit comme Q, c'est-à-dire que $(P \oplus Q) \to \approx_b P$ et que $(P \oplus Q) \to \approx_b Q$, où $\approx_b$ est une certaine équivalence comportementale définie sur $\boldsymbol{\pi}^\star$. Une autre source de non-déterminisme,

1. Nous utilisons la convention qui consiste à désigner par (Pa), aussi bien une sélection qu'une application.

est le conflit entre deux messages destinés à une même ressource. Comme dans le terme $(\langle u \Leftarrow P\rangle) \mid ub_1 \dots b_k \mid ua_1 \dots a_n$ par exemple.

On peut remarquer que la définition de la relation de réduction ne comporte pas de notion d'erreurs d'exécution. Or, comme nous l'avons indiqué en introduction, certains processus sont de manières évidente des erreurs. Une solution classique pour traiter le cas des erreurs, est de définir une constante, disons **err**, et des règles de réduction supplémentaires [128]: par exemple $((\boldsymbol{\lambda} x)P) \cdot l \to \mathbf{err}$. Pour garder notre présentation simple, nous préférons ne pas introduire ces règles annexes. On remarque toutefois que le processus $((\boldsymbol{\lambda} x)P) \cdot l$ ne se réduit pas.

2.3 Variantes du calcul bleu

Comme pour le π-calcul, il existe presque autant de versions du calcul bleu, que d'articles publié à son sujet. Dans ce chapitre nous donnons les définitions de deux versions du calcul bleu. La première version étudiée, que nous appelons symétrique, correspond à la définition initiale de $\boldsymbol{\pi}^\star$: c'est le calcul avec un opérateur de composition parallèle commutatif. L'étude de cette version nous permet d'éclairer certains choix fait sur la sémantique du calcul, elle a aussi un intérêt «historique», puisque la première définition du calcul bleu [22] utilisait un opérateur de composition parallèle commutatif.

Nous définissons, dans la section 2.3.2, une seconde variante du calcul bleu, appelée calcul *local*, d'après le nom de «π-calcul local» ($\mathbf{L}\boldsymbol{\pi}$) donné par D. Sangiorgi et M. Merro [89], qui identifie une variante de $\boldsymbol{\pi}$ dans laquelle un nom reçu ne peut être utilisé que pour émettre.

2.3.1 Calcul bleu symétrique

Dans toutes les versions du π-calcul connu par l'auteur, l'opérateur de composition parallèle est commutatif. C'est-à-dire qu'on a la règle d'équivalence structurelle suivante: $(P \mid Q) \equiv (Q \mid P)$. Ce choix est également celui fait dans les premières définitions du calcul bleu [22, 36], que nous appellerons aussi *calcul classique*.

Pour obtenir la définition du calcul bleu symétrique, il suffit de modifier les règles d'équivalence structurelle données dans la figure 2.2. Ainsi, en plus de la règle de commutativité du parallèle, on doit remplacer la règle de distributivité de l'application $(P \mid Q)a \equiv P \mid (Qa)$, par la règle: $(P \mid Q)a \equiv (Pa) \mid (Qa)$. En effet, dans le cas contraire, l'utilisation de la relation $\equiv$, pourrait introduire une nouvelle forme de non-déterminisme dans les réductions. Comme dans le cas suivant, par exemple:

$$\begin{array}{ccc} ((\boldsymbol{\lambda} x)P \mid (\boldsymbol{\lambda} x)Q)a & \to & ((\boldsymbol{\lambda} x)P \mid Q\{a/x\}) \\ \equiv & & \not\equiv \\ ((\boldsymbol{\lambda} x)Q \mid (\boldsymbol{\lambda} x)P)a & \to & ((\boldsymbol{\lambda} x)Q \mid P\{a/x\}) \end{array}$$

Ce phénomène est gênant car l'équivalence structurelle n'est plus un bisimulation. De même, il nous faut ajouter des règles permettant de distribuer l'application sur les déclarations:

$$(\langle u \Leftarrow P\rangle a) \equiv \langle u \Leftarrow P\rangle \qquad (\langle u = P\rangle a) \equiv \langle u = P\rangle$$

À titre indicatif, nous donnons dans la figure 2.4 le récapitulatif des règles d'équivalence structurelle dans le calcul classique.

Le calcul classique peut sembler plus simple que $\boldsymbol{\pi}^\star$, néanmoins cette simplification s'accompagne de l'introduction de nouvelles règles d'équivalence, qui violent les intuitions que nous avons donné du calcul. En particulier, on perd la possibilité de désigner la partie fonctionnelle «active» d'un processus. De plus, l'utilisation du parallèle dissymétrique permet de simplifier les règles de typage du calcul.

Les systèmes de types pour le calcul bleu, ne seront introduits que dans la partie III. Disons simplement que, dans le calcul classique, on peut trouver des exemples tels que $(Pa) \mid (Qa)$ est

$$
\begin{array}{rcll}
P =_\alpha Q & \Longrightarrow & P \equiv Q & \\
((P_1 \mid P_2) \mid P_3) & \equiv & (P_1 \mid (P_2 \mid P_3)) & \\
(P \mid Q) & \equiv & (Q \mid P) & \\
((\boldsymbol{\nu} u)P) \mid Q & \equiv & (\boldsymbol{\nu} u)(P \mid Q) & (u \notin \mathbf{fn}(Q)) \\
(\boldsymbol{\nu} u)(\boldsymbol{\nu} v)P & \equiv & (\boldsymbol{\nu} v)(\boldsymbol{\nu} u)P & \\
(P \mid Q)a & \equiv & (Pa) \mid (Qa) & \\
((\boldsymbol{\nu} u)P)a & \equiv & (\boldsymbol{\nu} u)(Pa) & (a \neq u) \\
(\langle u \Leftarrow P \rangle)a & \equiv & \langle u \Leftarrow P \rangle & (\text{de même pour } \langle u = P \rangle) \\
P \equiv Q & \Longrightarrow & \mathbf{C}[P] \equiv \mathbf{C}[Q] &
\end{array}
$$

Fig. 2.4: Équivalence structurelle dans le calcul bleu symétrique

bien typé, alors que $(P \mid Q)a$ ne l'est pas. On considère, dans ce cas, un système avec des types polymorphes, cf. section 13.

2.3.2 Calcul bleu local

Si on se base sur le codage informel du λ-calcul donné en introduction de cette thèse, on peut interpréter le processus $(\boldsymbol{\nu} u)(\langle u = N \rangle \mid (Mu))$ comme étant équivalent à l'application d'ordre supérieur $(M\,N)$. Nous introduisons un opérateur dérivé pour simplifier les idées développées dans cette section. Soit $(\mathbf{def}\, u = Q \,\mathbf{in}\, P)$ l'opérateur définit par le processus $(\boldsymbol{\nu} u)(\langle u = Q \rangle \mid P)$. Alors, si u est un nom nouveau, on choisit de noter par $(P\,Q)$ le processus:

$$(P\,Q) \;=_{\text{def}}\; (\mathbf{def}\, u = Q \,\mathbf{in}\, (Pu)) \;=_{\text{def}}\; (\boldsymbol{\nu} u)(\langle u = Q \rangle \mid Pu)$$

Avec cette notation, on obtient une règle dérivée pour la béta-réduction, qui est équivalente à celle que l'on trouve dans le λ-calcul avec substitutions explicites [1]Ėn supposant que $(\mathbf{def}\, u = Q \,\mathbf{in}\, P)$ est la substitution explicite du nom u, par le processus Q, dans P. Soit x une variable qui n'apparaît pas libre dans Q, alors:

$$((\boldsymbol{\lambda} x)P)\,Q \;\rightarrow\; (\boldsymbol{\nu} u)(\langle u = Q \rangle \mid P\{u/x\}) \;=\; (\mathbf{def}\, u = Q \,\mathbf{in}\, P\{u/x\})$$

Cependant, cette règle n'est pas toujours compatible avec la définition de la béta-réduction donnée par la règle (red beta). En particulier, si on applique cette règle dans le cas suivant:

$$((\boldsymbol{\lambda} x)\langle x \Leftarrow R \rangle)v \;\rightarrow\; \mathbf{def}\, u = v \,\mathbf{in}\, \langle u \Leftarrow R\{u/x\} \rangle$$

on obtient comme résultat un processus inerte, alors que l'on s'attend à obtenir un processus équivalent à la déclaration: $\langle u \Leftarrow R\{u/x\} \rangle$.

Pour se débarrasser de ce cas problématique, une solution est d'interdire l'abstraction des noms qui apparaissent en position objet d'une déclaration. Nous disons d'un processus qui vérifie cette propriété, qu'il vérifie la condition de localité. Dans le calcul local, un nom reçu pendant une communication ne peut pas servir à créer une déclaration, c'est-à-dire en terme de π-calcul : il ne peut pas servir à créer un récepteur. On dit aussi que seule la «*capacité d'émission*» est communiquée.

Nous nommons *calcul local* le calcul obtenu en restreignant l'abstraction, ceci en référence au π-calcul local de [89], et au calcul π^i_a de M. Boreale [16]. Cette restriction ne nuit pas à l'expressivité du calcul: en appliquant les résultats de [16], on peut exhiber un codage de $\boldsymbol{\pi}^\star$ dans le calcul bleu local.

La condition de localité peut être facilement imposée. Il suffit de considérer deux catégories de noms distinctes: une pour les variables (les noms qui peuvent être abstraits) et une pour les références (les noms qui peuvent apparaître dans une déclaration). De plus, la condition de localité est cohérente avec les hypothèses généralement admises dans les langages de programmation: on

P,Q	::=	$x \mid u$	variable ou référence
	$\mid$	$(\boldsymbol{\lambda} x)P$	abstraction (sur les variables)
	$\mid$	(Pa)	application
	$\mid$	$(P \mid Q)$	composition parallèle
	$\mid$	$(\boldsymbol{\nu} u)P$	restriction (sur les références)
	$\mid$	$\langle u \Leftarrow P \rangle \mid \langle u = P \rangle$	déclarations (sur les références)
	$\mid$	$(P \cdot l) \mid [\,] \mid [\,P\,, l = Q\,]$	enregistrement

Fig. 2.5: Syntaxe pour le calcul bleu local

peut interpréter le nom u, dans le processus $\langle u \Leftarrow P \rangle$, comme étant une référence à un objet dont le code est P. La contrainte de localité implique alors que, si l'on transmet une référence à un objet, personne ne peut créer d'objet accessible à la même référence. Ainsi, moyennant la définition d'un système de type assez simple [8] pour le calcul local, on peut assurer qu'à un nom correspond un seul objet (une seule déclaration).

En effet, dans le calcul local, on peut connaître statiquement, c'est-à-dire en examinant la définition d'un terme, quels sont les déclarations possibles sur un nom. Cette propriété est utile pour le typage, c'est ce que nous verrons dans la partie III (en particulier à la section 13.4), elle est aussi utile au niveau du raisonnement: elle permet de maintenir un invariant important lors des réductions d'un terme, celui que l'ensemble des noms déclarés d'un terme décroît. Pour conclure cette section, signalons que le premier calcul de processus ayant imposé l'hypothèse de localité est le calcul Join [52, 55]. En effet, dans ce calcul, l'hypothèse de localité découle du fait que le nom d'un récepteur est toujours restreint. On peut également trouver cette hypothèse employée dans le calcul TyCO [135].

CHAPITRE 3

Expressivité du calcul bleu

Le calcul bleu, que nous avons introduit au chapitre précédent, inclut à la fois les constructeurs du π-calcul et du λ-calcul. Il semble donc fournir un modèle d'exécution plus expressif que le π-calcul, et il se pose le problème de comparer l'expressivité de ces deux modèles. Cette question est traitée dans la section 3.1. Le problème se pose aussi de comparer la simplicité de ces deux modèles, dans le sens de la simplicité d'expression de certains exemples. Aussi, nous donnons dans la section 3.5, quelques exemples de processus empruntés au π-calcul. L'étude du codage du λ-calcul, dans la section 3.3, fournit un autre exemple de la simplicité et de l'expressivité du calcul bleu.

Dans ce chapitre, nous considérons que le lecteur est familier avec la théorie du π-calcul et du λ-calcul. Bien que nous introduisions le π-calcul à la section 3.1, nous invitons les «non-spécialistes» à lire l'introduction écrite par R. Milner [97]. La thèse de D. Turner [130] est aussi une lecture conseillée pour comprendre le modèle de programmation associé au π-calcul. Pour ce qui concerne le problème de l'interprétation du λ-calcul dans le π-calcul, le lecteur profitera de la lecture de [98]. Ce problème est traité avec plus de détails dans [121], et dans la thèse de D. Sangiorgi [114].

3.1 Interprétation du pi-calcul

Parmi les nombreuses définitions de π-calcul existantes, la version qui se rapproche le plus de notre calcul, est celle du mini π-calcul asynchrone et polyadique. C'est-à-dire le calcul sans opérateur de choix, ni de matching, utilisé comme base du langage de programmation Pict [105, 130]. Dans la suite, nous dénotons ce calcul $\boldsymbol{\pi}$. La syntaxe de $\boldsymbol{\pi}$ est donnée par la grammaire suivante:

$$P,Q ::= \bar{u}\langle\tilde{v}\rangle \mid u(\tilde{v}).P \mid !u(\tilde{v}).P \mid (P \mid Q) \mid (\boldsymbol{\nu} u)P \tag{3.1}$$

où $\tilde{v}$ est un tuple de noms, appelés aussi canaux de communications. La sémantique de $\boldsymbol{\pi}$ est donnée dans la figure 3.1. Ce calcul, à priori très simple, est suffisamment expressif pour coder certaines stratégies de réduction du λ-calcul [98, 116]. Il est aussi possible d'y coder le π-calcul d'ordre supérieur [115, 114], c'est-à-dire le calcul dans lequel on échange des processus, plutôt que des noms, au cours d'une communication. On suppose que les noms du π-calcul sont inclus dans $\mathcal{N}$. Le codage de $\boldsymbol{\pi}$ dans $\boldsymbol{\pi}^\star$, donné dans la définition suivante, est simple et direct.

Définition 3.1 (Codage de π dans $\pi^\star$)

$$\begin{aligned}
[\![\bar{u}\langle v_1 \dots v_n\rangle]\!] &= uv_1 \dots v_n \\
[\![u(\tilde{v}).P]\!] &= \langle u \Leftarrow (\boldsymbol{\lambda}\tilde{v})P\rangle \\
[\![!u(\tilde{v}).P]\!] &= \langle u = (\boldsymbol{\lambda}\tilde{v})P\rangle \\
[\![P \mid Q]\!] &= ([\![P]\!] \mid [\![Q]\!]) \\
[\![(\boldsymbol{\nu}u)P]\!] &= (\boldsymbol{\nu}u)[\![P]\!]
\end{aligned}$$

syntaxe:

$P,Q ::= \bar{u}\langle\tilde{v}\rangle \mid u(\tilde{v}).P \mid !u(\tilde{v}).P \mid (P \mid Q) \mid (\boldsymbol{\nu}u)P$

équivalence structurelle:

$$\begin{aligned}
P =_\alpha Q &\implies P \equiv Q \\
((P \mid Q) \mid R) &\equiv (P \mid (Q \mid R)) \\
(P \mid Q) &\equiv (Q \mid P) \\
((\boldsymbol{\nu}u)P) \mid Q &\equiv (\boldsymbol{\nu}u)(P \mid Q) \qquad (u \notin \mathbf{fn}(Q)) \\
(\boldsymbol{\nu}u)(\boldsymbol{\nu}v)P &\equiv (\boldsymbol{\nu}v)(\boldsymbol{\nu}u)P \\
(\boldsymbol{\nu}u)P &\equiv P \qquad (u \notin \mathbf{fn}(P)) \\
P \equiv Q &\Rightarrow \mathbf{C}[P] \equiv \mathbf{C}[Q]
\end{aligned}$$

réduction: on suppose que les tuples $\tilde{v}$ et $\tilde{w}$ sont de taille égale, et qu'il n'existe pas deux noms égaux dans $\tilde{w}$.

$$\begin{aligned}
&\bar{u}\langle\tilde{v}\rangle \mid u(\tilde{w}).P \rightarrow_\pi P\{\tilde{v}/\tilde{w}\} \\
&\bar{u}\langle\tilde{v}\rangle \mid !u(\tilde{w}).P \rightarrow_\pi P\{\tilde{v}/\tilde{w}\} \mid !u(\tilde{w}).P \\
&P \rightarrow_\pi P' \Rightarrow (P \mid Q) \rightarrow_\pi (P' \mid Q) \;\&\; (\boldsymbol{\nu}u)P \rightarrow_\pi (\boldsymbol{\nu}u)P' \\
&P \rightarrow_\pi P' \;\&\; P \equiv Q \Rightarrow Q \rightarrow_\pi P'
\end{aligned}$$

Fig. 3.1: Le π-calcul: syntaxe et sémantique opérationnelle

Il est simple de montrer que les réductions de $\boldsymbol{\pi}$ sont mimées de la manière suivante.

Lemme 3.1 ($\boldsymbol{\pi^\star}$ simule $\boldsymbol{\pi}$) *Si P est un processus de $\boldsymbol{\pi}$ tel que $P \rightarrow_\pi P'$, alors*

$$([\![P]\!] \mid \mathbf{0}) \rightarrow_{(\rho)} \stackrel{*}{\rightarrow}_{(\beta)} ([\![P']\!] \mid \mathbf{0})$$

Ainsi, par exemple, la communication du π-calcul se traduit de la manière suivante. Soit $\tilde{v}$ et $\tilde{w}$ deux tuples de même taille – on suppose que les noms de $\tilde{w}$ sont distincts – nous utilisons la notation $P\{\tilde{v}/\tilde{w}\}$ pour dénoter la substitutions des noms w_i par v_i.

$$\begin{aligned}
[\![\bar{u}\langle\tilde{v}\rangle \mid u(\tilde{w}).P]\!] \mid \mathbf{0} \quad &=_{\text{def}} \quad u\tilde{v} \mid \langle u \Leftarrow (\boldsymbol{\lambda}\tilde{w})[\![P]\!]\rangle \mid \mathbf{0} \\
&\equiv \quad (\langle u \Leftarrow (\boldsymbol{\lambda}\tilde{w})[\![P]\!]\rangle \mid u\tilde{v}) \mid \mathbf{0} \\
&\rightarrow_{(\rho)} \quad ((\boldsymbol{\lambda}\tilde{w})[\![P]\!])\tilde{v} \mid \mathbf{0} \\
&\stackrel{*}{\rightarrow}_{(\beta)} \quad [\![P]\!]\{\tilde{v}/\tilde{w}\} \equiv ([\![P\{\tilde{v}/\tilde{w}\}]\!] \mid \mathbf{0})
\end{aligned}$$

Preuve La preuve se fait par induction sur l'inférence de $P \rightarrow_\pi P'$. Nous ne donnerons pas les détails ici, le lecteur pouvant trouver les grandes lignes de la preuve dans [22]. Disons simplement que la présence de nil est nécessaire dans les cas utilisant la commutativité du parallèle dans $\boldsymbol{\pi}$.□

Nous montrons que le codage de $\boldsymbol{\pi}$ dans $\boldsymbol{\pi}^\star$ est complet – plus précisément, le lemme 3.1 implique que le π-calcul est inclus dans $\boldsymbol{\pi}^\star$– ce qui implique que $\boldsymbol{\pi}^\star$ est au moins aussi expressif que $\boldsymbol{\pi}$. La propriété inverse est aussi vraie. La preuve de la correction du codage du π-calcul dans le calcul bleu a été donnée par G. BOUDOL dans [22], où l'auteur utilise des transformations similaires aux transformations «continuation passing style» (CPS) du λ-calcul. Une conclusion que nous pouvons tirer de ce résultat, est que le π-calcul, comparé au calcul bleu, est un langage assembleur qui encourage l'utilisation d'un style indirect pour modéliser les comportements.

A titre d'information, on peut dire que la propriété inverse du lemme 3.1 n'est pas toujours vérifiée. C'est le cas, dans l'exemple précédent, si les tuples $\tilde{v}$ et $\tilde{w}$ ne sont pas de la même taille par exemple. En effet on peut avoir une réduction dans $\boldsymbol{\pi}^\star$, alors qu'on a une erreur dans $\boldsymbol{\pi}$. Cependant, comme le montre G. BOUDOL [22, Lemma 5.2], le résultat est vrai si P est bien typé: où la notion de type considérée sur $\boldsymbol{\pi}$, est le système de sortes de MILNER (cf. section 11.3).

Avant de s'intéresser au codage du λ-calcul, on introduit un opérateur dérivé dans $\boldsymbol{\pi}^\star$, pour traiter les définitions récursives.

3.2 L'opérateur de définition

En utilisant la déclaration répliquée, il est facile de construire un opérateur dérivé, noté $(\mathbf{def}\, u = R \,\mathbf{in}\, P)$, qui a une sémantique semblable aux opérateurs de définitions récursives des langages de programmation fonctionnel, tel que ML ou ISWIM [78] par exemple.

Définition 3.2 (Opérateur de définition) Soit $\tilde{u}$ le tuple de noms $(u_1, \ldots, u_n)$, sans aucune répétition. On utilise la lettre D pour désigner une liste de définitions mutuellement récursives: $u_1 = P_1, \ldots, u_n = P_n$, et on dénote $(\mathbf{def}\, D \,\mathbf{in}\, Q)$ le processus

$$\mathbf{def}\, D \,\mathbf{in}\, Q =_{\text{def}} (\boldsymbol{\nu}\, \tilde{u})(\langle u_1 = P_1 \rangle \mid \cdots \mid \langle u_n = P_n \rangle \mid Q)$$

à la condition qu'aucun nom u_i ne soit déclaré dans Q, c'est-à-dire que $(\tilde{u} \cap \mathbf{decl}(Q)) = \emptyset$.

Il est facile de montrer que cet opérateur obéit aux règles d'équivalence structurelle donnée dans la figure [21]. Il faut noter que la première de ces règles est encore valide si on remplace v par une sélection, ainsi on a: $(\mathbf{def}\, D \,\mathbf{in}\, P) \cdot l \equiv \mathbf{def}\, D \,\mathbf{in}\, (P \cdot l)$. On verra que ces règles correspondent, en partie, aux règles d'équivalence structurelle de $\boldsymbol{\Lambda}^\star$, le «$\lambda$-calcul avec passage de noms» défini à la section suivante.

En ce qui concerne la réduction, il est facile de montrer que la règle (red def) définie dans la figure 3.2 peut être dérivée. On retrouve donc l'équivalent de la réduction dans $\boldsymbol{\Lambda}^\star$, puisque cette règle implique que:

$$\mathbf{def}\, D, u = P, D' \,\mathbf{in}\, (u\tilde{v}) \quad \rightarrow\!\equiv \quad \mathbf{def}\, D, u = P, D' \,\mathbf{in}\, (P\tilde{v})$$

Le lecteur peut noter que, dans la définition de $(\mathbf{def}\, D \,\mathbf{in}\, Q)$, la condition qui impose aux $(u_i)^{i \in [1..n]}$ d'être distincts, et différents des noms de $\mathbf{decl}(Q)$, implique que le canal u_i correspond à une unique déclaration répliquée. Dans ce cas, il s'agit de $\langle u_i = P_i \rangle$. Ceci correspond à un cas particulier de l'*hypothèse du récepteur unique*, souvent employée dans $\boldsymbol{\pi}$: c'est-à-dire qu'un canal est toujours associé à un seul récepteur.

Dans la suite de cette thèse, il nous arrivera de considérer une version du calcul bleu dans laquelle l'opérateur de déclaration répliquée $\langle u = P \rangle$ est remplacé par l'opérateur de définition. Ceci ne modifie pas l'expressivité de notre calcul, puisqu'il existe un codage simple de l'un vers l'autre. En effet, soit x un nouveau nom, on a:

$$\langle u = P \rangle \;\simeq\; \mathbf{def}\, x = \langle u \Leftarrow (x \mid P) \rangle \,\mathbf{in}\, x \;\simeq\; \mathbf{rec}\, x.\langle u \Leftarrow (x \mid P) \rangle$$

équivalence structurelle: on note $\tilde{u}$ le tuple des noms définis par D.

$$\begin{array}{rcll}
(\mathbf{def}\, D \,\mathbf{in}\, Q)v & \equiv & \mathbf{def}\, D \,\mathbf{in}\, (Qv) & (v \notin \tilde{u}) \\
\mathbf{def}\, D \,\mathbf{in}\, (\mathbf{def}\, v = P \,\mathbf{in}\, Q) & \equiv & \mathbf{def}\, D, v = P \,\mathbf{in}\, Q & (v \notin \tilde{u}) \\
(\mathbf{def}\, D \,\mathbf{in}\, P) \mid Q & \equiv & \mathbf{def}\, D \,\mathbf{in}\, (P \mid Q) & (\tilde{u} \cap \mathbf{fn}(Q) = \emptyset) \\
\mathbf{def}\, D \,\mathbf{in}\, ((\boldsymbol{\nu} v)P) & \equiv & (\boldsymbol{\nu} v)(\mathbf{def}\, D \,\mathbf{in}\, P) & (v \notin \tilde{u})
\end{array}$$

réduction:

$$\frac{D =_{\mathrm{def}} D_1, u = R, D_2 \qquad (\{u\} \cup \mathbf{fn}(R)) \cap \mathbf{bn}(\mathbf{E}) = \emptyset \qquad \mathbf{E} \in \mathcal{E}}{\mathbf{def}\, D \,\mathbf{in}\, \mathbf{E}[u] \rightarrow \mathbf{def}\, D \,\mathbf{in}\, \mathbf{E}[R]} \text{ (red def)}$$

Fig. 3.2: Règles dérivées pour l'opérateur de définition

où $\simeq$ représente n'importe quelle équivalence «sensée» pour le calcul bleu, et où $\mathbf{rec}\, x.P$ est l'opérateur dérivé de récursion défini par $\mathbf{rec}\, x.P = (\mathbf{def}\, x = P \,\mathbf{in}\, x)$. Dans la section suivante, nous nous intéressons au codage du λ-calcul dans le calcul bleu.

3.3 Interprétation du lambda-calcul

Une conséquence du lemme 3.1, qui énonce que le codage de $\boldsymbol{\pi}$ dans $\boldsymbol{\pi}^\star$ est complet, est qu'il est possible de remplacer le mécanisme de la communication asynchrone du π-calcul, par deux mécanismes encore plus élémentaires: l'accès aux ressources (représenté par $\rightarrow_{(\rho)}$), et la béta-réduction (représentée par $\rightarrow_{(\beta)}$). Ce phénomène est caractéristique du calcul bleu, où la communication est vue comme une forme spéciale d'accès aux données. Nous montrons dans cette section comment, en utilisant le même paradigme, on peut interpréter le λ-calcul dans le calcul bleu.

Les termes du λ-calcul sont obtenus à partir de trois constructeurs: les variables: x; l'abstraction: $\lambda x.M$, qui permet de construire une fonction; l'application: (MN), qui permet d'appliquer une fonction M à un argument N.

$$M, N ::= x \mid \lambda x.M \mid (MN)$$

On considère ici le λ-calcul faible en *appel par nom* [108], noté $\boldsymbol{\Lambda}$. Dans ce λ-calcul, la réduction est localisée à l'application la plus à gauche. Plus formellement, on a les deux règles de réduction suivantes.

$$(\lambda x.M)N \rightarrow_l M\{\!\{N/x\}\!\} \qquad\qquad M \rightarrow_l M' \Rightarrow (MN) \rightarrow_l (M'N)$$

On suppose que les variables de $\boldsymbol{\Lambda}$ sont des noms de $\boldsymbol{\pi}^\star$. Une première étape dans le codage du λ-calcul est la définition d'un calcul intermédiaire, noté $\boldsymbol{\Lambda}^\star$, qu'on appelle le «$\lambda$-calcul avec passage de noms». La sémantique de $\boldsymbol{\Lambda}^\star$ est donnée dans la figure 3.3. Ce calcul correspond à la fois à une restriction et à une extension de $\boldsymbol{\Lambda}$:

- La restriction est que, comme dans le calcul bleu, on ne peut appliquer un terme qu'à un nom. En particulier, on utilise la notation $(\boldsymbol{\lambda} x)M$ pour l'abstraction, au lieu de $\lambda x.M$;
- l'extension est réalisée par l'ajout d'un opérateur de définition: $\mathbf{def}\, x = L \,\mathbf{in}\, I$, comme celui défini à la section 3.2

Généralement, l'opérateur de définition ajouté au λ-calcul est noté **let**, c'est le cas dans ML [93] par exemple. Le choix d'un nom différent ici, est motivé par le fait que le **let** de ML est associé à la stratégie d'évaluation en *appel par valeur*, alors que notre stratégie est avec *appel par nom*. De plus l'opérateur **def** est récursif.

Dans $\mathbf{\Lambda}^\star$, au lieu de remplacer une variable par un terme, on échange des noms. On utilise alors les définitions pour mémoriser la «valeur» associée à chaque nom, exactement comme on a utilisé les déclarations dans la machine chimique décrite en introduction de cette thèse. Avec cette approche, on ne remplace un nom par une valeur que lorsque ceci est nécessaire, c'est-à-dire lorsque la variable apparaît en position de tête. On retrouve donc le mécanisme de $\boldsymbol{\pi}^\star$, où les définitions sont des ressources répliquées, et les formes «normales de têtes» sont des agents. Cependant, dans le λ-calcul avec passage de noms, et contrairement à $\boldsymbol{\pi}^\star$, un seul message à la fois peut être actif. En effet il n'y a pas de composition parallèle dans $\mathbf{\Lambda}^\star$.

syntaxe: on suppose les x_i distincts

$$\begin{array}{rcl} D & ::= & x_1 = L_1, \ldots, x_n = L_n \\ I,L & ::= & x \mid (\boldsymbol{\lambda} x) L \mid (Lx) \mid \mathbf{def}\, D\, \mathbf{in}\, I \end{array}$$

équivalence structurelle: on suppose que x n'est pas défini dans D

$$(\mathbf{def}\, D\, \mathbf{in}\, L)x \equiv \mathbf{def}\, D\, \mathbf{in}\, (Lx)$$
$$\mathbf{def}\, D\, \mathbf{in}\, (\mathbf{def}\, x = I\, \mathbf{in}\, L) \equiv \mathbf{def}\, D, x = I\, \mathbf{in}\, L$$

réduction:

$$(\boldsymbol{\lambda} x) L)y \to L\{y/x\}$$
$$\mathbf{def}\, D, x = L, D'\, \mathbf{in}\, x y_1 \ldots y_n \to \mathbf{def}\, D, x = L, D'\, \mathbf{in}\, L y_1 \ldots y_n$$
$$L \to L' \Rightarrow (Lz) \to (L'z)\ \&\ \mathbf{def}\, D\, \mathbf{in}\, L \to \mathbf{def}\, D\, \mathbf{in}\, L'$$
$$L \to L'\ \&\ L \equiv I \Rightarrow I \to L'$$

Fig. 3.3: Le λ-calcul avec passage de noms: $\mathbf{\Lambda}^\star$

Dans son article où il étudie le λ-calcul paresseux [78], J. LAUNCHBURY a donné une interprétation très simple de $\mathbf{\Lambda}$ dans $\mathbf{\Lambda}^\star$. On note $M^\star$ la traduction d'un terme de $\mathbf{\Lambda}$.

Définition 3.3 (Codage de $\mathbf{\Lambda}$ dans $\mathbf{\Lambda}^\star$)

$$\begin{array}{rcl} x^\star & = & x \\ (\lambda x.M)^\star & = & (\boldsymbol{\lambda} x)(M^\star) \\ (MN)^\star & = & \mathbf{def}\, u = N^\star\, \mathbf{in}\, (M^\star u) \quad (\text{où } u \text{ est choisit non libre dans } (MN)) \end{array}$$

Il faut noter que la définition ($\mathbf{def}\, u = N^\star\, \mathbf{in} \ldots$), qui apparaît dans le codage de l'application, n'est pas récursive, car le nom u n'apparaît pas dans $N^\star$. Dans le cas où une définition n'est pas récursive, on montre qu'il est possible de définir une transformation inverse de $\mathbf{\Lambda}^\star$ vers $\mathbf{\Lambda}$, telle que ($\mathbf{def}\, u = I\, \mathbf{in}\, L$) devient $L\{\!\{I/u\}\!\}$. Un résultat de correction prouvé par J. LAUNCHBURY, nous montre que $\mathbf{\Lambda}$ et $\mathbf{\Lambda}^\star$ sont équivalents (ce résultat est formellement défini dans le théorème 3.2). Nous donnons d'abord quelques définitions. Soit M un terme de $\mathbf{\Lambda}$. On dit que M converge si $M \xrightarrow{*}_l \lambda x.N$. Cette propriété est notée $M \Downarrow \lambda x.N$. De la même façon, on définit une notion de valeur dans $\mathbf{\Lambda}^\star$: on dit que L' est une fermeture si $L' \equiv (\mathbf{def}\, D\, \mathbf{in}\, (\boldsymbol{\lambda} x) I)$, un terme L converge, noté $L \Downarrow L'$, si L peut se réduire en une fermeture L' après plusieurs pas de calculs. Le résultat suivant énonce que $\mathbf{\Lambda}^\star$ simule $\mathbf{\Lambda}$.

Théorème 3.2 (Launchbury [78]) *Soit M un terme clos de $\mathbf{\Lambda}$. Alors le terme M converge, si et seulement si $M^\star$ converge: $M \Downarrow M' \iff M^\star \Downarrow L'$. De plus, on montre que M' s'obtient depuis L', en utilisant la «transformation inverse» de $\mathbf{\Lambda}^\star$ vers $\mathbf{\Lambda}$.*

On montre maintenant comment coder $\boldsymbol{\Lambda}^\star$ dans $\boldsymbol{\pi}^\star$. En utilisant l'opérateur dérivé de définition de $\boldsymbol{\pi}^\star$, on obtient une interprétation directe de $\boldsymbol{\Lambda}^\star$ dans $\boldsymbol{\pi}^\star$, donnée dans la définition ci-dessous. L'interprétation de $\boldsymbol{\Lambda}$ dans $\boldsymbol{\pi}^\star$, s'obtient alors par composition des deux codages $(.)^\star$ et $[\![.]\!]$ (définition 3.3 et 3.4). On peut trouver cette interprétation dans la définition 8.1.

Définition 3.4 (Codage de $\boldsymbol{\Lambda}^\star$ dans $\boldsymbol{\pi}^\star$)

$$\begin{aligned}
[\![x_i = L_1, \ldots, x_n = L_n]\!] &= x_i = [\![L_1]\!], \ldots, x_n = [\![L_n]\!] \\
[\![x]\!] &= x \\
[\![(\boldsymbol{\lambda} x) L]\!] &= (\boldsymbol{\lambda} x)[\![L]\!] \\
[\![Lx]\!] &= ([\![L]\!] x) \\
[\![\mathbf{def}\, D \,\mathbf{in}\, L]\!] &= \mathbf{def}\, [\![D]\!] \,\mathbf{in}\, [\![L]\!]
\end{aligned}$$

Le résultat de correction du codage de $\boldsymbol{\Lambda}^\star$ dans $\boldsymbol{\pi}^\star$ est plus fort que celui énoncé dans le théorème 3.2. En effet, il est facile de montrer la propriété suivante.

Proposition 3.3 ($\boldsymbol{\pi}^\star$ simule $\boldsymbol{\Lambda}^\star$) *Soit L un terme de $\boldsymbol{\Lambda}^\star$. Si L réduit: $L \to L'$, alors il existe un processus P' tel que $[\![L]\!] \to P'$ et $P' \equiv [\![L']\!]$. Réciproquement, si $[\![L]\!] \to P'$, alors il existe L' tel que $L \to L'$ et $P' \equiv [\![L']\!]$.*

Associé au théorème 3.2, cette propriété implique la correction de l'interprétation de $\boldsymbol{\Lambda}$ dans $\boldsymbol{\pi}^\star$. Il suffit, pour ce faire, de fournir une définition cohérente de la convergence dans $\boldsymbol{\pi}^\star$, c'est-à-dire une définition de la relation $P \Downarrow P'$. Cette notion de convergence est donnée dans la définition 4.1, à la section 4.3. Nous ne donnerons donc pas plus de détails ici.

Dans le codage du λ-calcul, la composition parallèle est utilisée uniquement dans le codage de l'application (implicitement dans le codage de $\mathbf{def}\, D \,\mathbf{in}\, P$), c'est-à-dire pour coder les fermetures. Elle ne sert qu'à ajouter des déclarations (répliquées) à un agent. Ceci motive, en partie, le choix d'un opérateur parallèle dissymétrique pour notre calcul, ainsi que la règle de distribution de l'application sur le parallèle: $(P \mid Q)a \equiv P \mid (Qa)$. En effet, avec un parallèle dissymétrique, on peut désigner la «partie fonction» d'un processus, c'est-à-dire les expressions du λ-calcul dans notre codage, comme étant la partie à droite du parallèle le plus externe.

Une autre conséquence du codage de $\boldsymbol{\Lambda}$, est qu'il nous permet de mieux cerner l'ensemble des processus du calcul bleu , ayant un «comportement fonctionnel»: il contient bien sûr les termes engendrés par les constructeurs du λ-calcul, mais il contient aussi des termes de la forme $\mathbf{def}\, D \,\mathbf{in}\, P$. C'est ce calcul que nous étudions dans la section suivante.

3.4 Un lambda-calcul avec définitions récursives

Dans cette section, nous définissons un nouveau calcul fonctionnel, dénoté $\boldsymbol{\lambda}\mathcal{D}$ef. Il s'agit, intuitivement, du calcul obtenu à partir de $\boldsymbol{\pi}^\star$ en se privant de la restriction, de la composition parallèle et des déclarations, et en ajoutant les définitions. On peut donc voir $\boldsymbol{\lambda}\mathcal{D}$ef comme la «partie fonctionnelle» du calcul bleu, ou comme un sous-calcul déterministe du calcul bleu. Ce calcul nous sera utile dans la partie IV, pour étudier l'interprétation des objets extensibles.

Définition 3.5 (Syntaxe de $\boldsymbol{\lambda}\mathcal{D}$ef) On suppose l'existence d'un nombre dénombrable de variable x,$y\ldots$ incluses dans $\mathcal{N}$. Les termes de $\boldsymbol{\lambda}\mathcal{D}$ef sont définis de la manière suivante.

M,N	::=	$x \mid (\boldsymbol{\lambda} x) M \mid (MN)$	λ-calcul
	$\mid$	$\mathbf{def}\, x = N \,\mathbf{in}\, M$	définition récursive
	$\mid$	$(M \cdot l)$	sélection du champ l
	$\mid$	$[\,]$	enregistrement vide
	$\mid$	$[M, l = N]$	extension/modification

On distingue un sous-ensemble de termes qui représente les valeurs de $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$.

$$V \;::=\; (\boldsymbol{\lambda}x)M \mid [\,M\,,l = N\,] \mid \mathbf{def}\,x = N\,\mathbf{in}\,V$$

Comme nous allons le voir maintenant, les opérateurs de $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$ ont pratiquement la même sémantique que dans le calcul bleu. D'ailleurs tout terme de ce calcul peut s'interpréter directement comme un processus du calcul bleu. On peut alors définir les notions de variables libres et liés, comme au chapitre 2. De même pour les notions de contexte et de substitution. En s'inspirant des relations d'équivalence structurelle et de réduction de $\boldsymbol{\pi}^\star$, on peut définir une relation de réduction de la manière suivante.

$$\begin{array}{rcll}
((\boldsymbol{\lambda}x)M)N & \to & \mathbf{def}\,x = N\,\mathbf{in}\,M & (x \notin \mathbf{fn}(N)) \\
[\,R\,,l = M\,]\cdot l & \to & M & \\
[\,R\,,l = M\,]\cdot k & \to & R\cdot k & (k \neq l) \\
(\mathbf{def}\,x = N\,\mathbf{in}\,M)L & \to & \mathbf{def}\,x = N\,\mathbf{in}\,(ML) & (x \notin \mathbf{fn}(L)) \\
(\mathbf{def}\,x = N\,\mathbf{in}\,M)\cdot l & \to & \mathbf{def}\,x = N\,\mathbf{in}\,(M\cdot l) & \\
(\mathbf{def}\,x = N\,\mathbf{in}\,\mathbf{C}[x]) & \to & (\mathbf{def}\,x = N\,\mathbf{in}\,\mathbf{C}[N]) & ((\{x\}\cup\mathbf{fn}(N))\cap\mathbf{bn}(\mathbf{C}) = \emptyset) \\
\multicolumn{4}{c}{M \to N \;\Rightarrow\; \mathbf{C}[M] \to \mathbf{C}[M']}
\end{array}$$

On pourra noter cette relation $\to_{\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}}$, pour la distinguer de la relation de réduction du calcul bleu, et on note $M \Downarrow$ le fait que le terme M converge, c'est-à-dire qu'il existe une valeur V telle que $M \xrightarrow{*}_{\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}} V$. Nous définissons aussi une relation de conversion entre termes, notée $\approx_{\mathcal{D}}$, qui est la plus petite congruence qui contient la relation de réduction, et qui obéit aux équations suivantes:

$$\begin{array}{rcll}
(\mathbf{def}\,x = N\,\mathbf{in}\,M) & \approx_{\mathcal{D}} & M\{\!\{\mathbf{rec}\,x.N/x\}\!\} & (\text{avec } \mathbf{rec}\,x.N =_{\mathrm{def}} (\mathbf{def}\,x = N\,\mathbf{in}\,x)) \\
[\,[\,R\,,l = M\,]\,,k = N\,] & \approx_{\mathcal{D}} & [\,[\,R\,,k = N\,]\,,l = M\,] & (k \neq l) \\
[\,[\,R\,,l = M\,]\,,l = N\,] & \approx_{\mathcal{D}} & [\,R\,,l = N\,] &
\end{array}$$

Bien qu'inspiré par la partie fonctionnelle du calcul bleu, le calcul $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$ ne s'obtient pas directement de $\boldsymbol{\pi}^\star$. Une première différence est l'utilisation de l'application d'ordre supérieur: (MN). Cependant, comme dans le cas du codage du λ-calcul dans $\boldsymbol{\Lambda}^\star$, on peut interpréter ce terme par le processus $(\mathbf{def}\,x = N\,\mathbf{in}\,(Mx))$, où x est une variable fraîche. Une seconde différence, beaucoup plus profonde, se trouve dans la définition de la réduction. Ainsi, dans $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$, on peut réduire un terme dans tout contexte:

$$(M \to_{\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}} M') \;\Rightarrow\; (\mathbf{C}[M] \to_{\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}} \mathbf{C}[M'])$$

Alors que dans le cas du calcul bleu, la règle (red context) restreint la réduction au cas où $\mathbf{C}$ est un contexte d'évaluation. Il en est de même pour la réduction de la définition. Dans la partie II, nous prouvons des propriétés qui permettent de montrer que $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$ peut s'interpréter dans le calcul bleu, et que la notion de conversion ($\approx_{\mathcal{D}}$) est induite par notre relation d'équivalence sur $\boldsymbol{\pi}^\star$. C'est-à-dire qu'on a la propriété suivante. Soit $\approx_b$ la relation d'équivalence définie dans la partie II, et soit M et N deux termes de $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$. On note $M^\bullet$ le processus de $\boldsymbol{\pi}^\star$ obtenu à partir de M par l'isomorphisme tel que $(MN)^\bullet = (\mathbf{def}\,u = N\,\mathbf{in}\,(Mu))$ – pour un u nouveau –, et tel que $(.)^\bullet$ est un homomorphisme pour les autres constructeurs de $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$.

Proposition 3.4 ($\boldsymbol{\pi}^\star$ simule $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$) *Si $M \approx_{\mathcal{D}} N$, alors $M^\bullet \approx_b N^\bullet$.*

3.5 Quelques exemples de processus

Pour terminer ce chapitre, nous donnons quelques exemples de processus du calcul bleu. On a montré dans la section 2.2.2, comment coder le choix interne dans notre calcul: l'opérateur $(P \oplus Q)$, nous donnons maintenant plusieurs exemples, classé en trois catégories: des exemples tirés du λ-calcul ; des exemples tirés du π-calcul, et en particulier les structures de données définies par R. Milner dans [97]; des exemples qui correspondent aux opérateurs des langages séquentiels.

3.5.1 Exemples de fonctions

Comme nous l'avons vu dans les sections 3.3 et 3.5, on peut définir un opérateur pour «l'application d'ordre supérieur» dans le calcul bleu.

Définition 3.6 (Application) Soit u un nom nouveau, on note $(P\,Q)$ le processus:

$$(P\,Q) \;=_{\text{def}}\; (\mathbf{def}\, u = Q\, \mathbf{in}\, (Pu)) \;=_{\text{def}}\; (\boldsymbol{\nu} u)(\langle u = Q\rangle \mid Pu)$$

Avec cette notation, on obtient une règle dérivée pour la béta-réduction, qui est équivalente à celle de $\boldsymbol{\lambda}\mathcal{D}$ef. Soit x une variable qui n'apparaît pas libre dans Q, alors:

$$((\boldsymbol{\lambda} x)P)Q \rightarrow (\boldsymbol{\nu} u)(\langle u = Q\rangle \mid P\{u/x\}) =_{\alpha} \mathbf{def}\, x = Q\, \mathbf{in}\, P$$

On peut aussi définir les combinateurs usuels du λ-calcul, comme les entiers de Church: $\underline{n} = (\boldsymbol{\lambda} f x)(f \ldots fx)$, les booléens vrai: $\mathsf{T} =_{\text{def}} (\boldsymbol{\lambda} xy)x$, et faux: $\mathsf{F} =_{\text{def}} (\boldsymbol{\lambda} xy)y$, ou encore la récursion:

Définition 3.7 (Récursion) On dénote $\mathbf{rec}\, u.P$ le processus $(\boldsymbol{\nu} u)(\langle u = P\rangle \mid u)$, c'est-à-dire le terme $(\mathbf{def}\, u = P\, \mathbf{in}\, u)$.

Dans la partie II, nous montrerons que cet opérateur a bien la sémantique d'un opérateur de récursion. En particulier, on peut démontrer que la loi classique de «dépliage» de la récursion est valide, c'est-à-dire que: $\mathbf{rec}\, u.P \;\approx_b\; P\{\!\{\mathbf{rec}\, u.P/u\}\!\}$. On peut aussi définir des fonctions non λ-définissables, comme le «ou parallèle»: **POR**, qui renvoie la valeur T dès que l'un de ses arguments est vrai, ceci même si l'autre argument diverge.

$$\mathsf{POR} =_{\text{def}} (\boldsymbol{\lambda} xy)(\boldsymbol{\nu} rtf)(\langle t \Leftarrow \langle r \Leftarrow \mathsf{T}\rangle\rangle \mid \langle f \Leftarrow \langle f \Leftarrow \langle r \Leftarrow \mathsf{F}\rangle\rangle\rangle \mid xtf \mid ytf \mid r)$$

Intuitivement, si l'un des arguments de **POR** est équivalent à T, alors, après un certains nombres de réductions déterministes, une déclaration $\langle r \Leftarrow \mathsf{T}\rangle$ est mise en solution. Cette déclaration peut alors interagir avec le message sur le nom r pour donner comme résultat un processus de la forme $(\boldsymbol{\nu} rtf)(\cdots \mid \mathsf{T})$, qui – dans ce cas précis – est équivalent à T.

Que **POR** soit fidèle à la spécification informelle que nous en avons donné, dépend de la sémantique choisie pour le calcul. Cela dépend en particulier de l'équivalence choisie sur les termes. Soit Ω le processus $(\boldsymbol{\nu} u)(\langle u = u\rangle \mid u)$ introduit à la section 2.2.2. On peut montrer par exemple que:

$$\begin{array}{lll}
(\mathsf{POR}\,\Omega\,\mathsf{T}) & \stackrel{*}{\rightarrow} \approx_b & (\boldsymbol{\nu} rtf)(\langle t \Leftarrow \langle r \Leftarrow \mathsf{T}\rangle\rangle \mid \langle f \Leftarrow \langle f \Leftarrow \langle r \Leftarrow \mathsf{F}\rangle\rangle\rangle \mid (\Omega tf) \mid t \mid r) \\
 & \rightarrow & (\boldsymbol{\nu} rtf)(\langle f \Leftarrow \langle f \Leftarrow \langle r \Leftarrow \mathsf{F}\rangle\rangle\rangle \mid (\Omega tf) \mid \langle r \Leftarrow \mathsf{T}\rangle \mid r) \\
 & \rightarrow & (\boldsymbol{\nu} rtf)(\langle f \Leftarrow \langle f \Leftarrow \langle r \Leftarrow \mathsf{F}\rangle\rangle\rangle \mid (\Omega tf) \mid \mathsf{T}) \\
 & \approx_b & \mathsf{T} \\
(\mathsf{POR}\,\mathsf{T}\,\Omega) & \stackrel{*}{\rightarrow} \approx_b & \mathsf{T} \\
(\mathsf{POR}\,\mathsf{F}\,\mathsf{F}) & \stackrel{*}{\rightarrow} \approx_b & \mathsf{F}
\end{array}$$

3.5.2 Exemples de processus

Comme le π-calcul est inclus dans le calcul bleu, on peut aussi donner des exemples de processus non séquentiels, comme des structures de données mutables, c'est-à-dire modifiables en place, par exemple. Un exemple de ce type est le buffer à une place: **BUFF**, qui peut alternativement réagir aux messages «put» et «get».

$$\mathbf{BUFF} =_{\text{def}} \mathbf{rec}\, b.\langle \mathsf{put} \Leftarrow (\boldsymbol{\lambda} x)\langle \mathsf{get} \Leftarrow (b \mid x)\rangle\rangle$$

Comme dans l'exemple de **POR**, on peut noter l'utilisation de déclarations imbriquées ($\langle f \Leftarrow \langle f \Leftarrow \ldots\rangle\rangle$ par exemple), comme moyen de synchroniser les messages. Le lecteur est invité, à titre d'exercice, à montrer que le terme $((\mathbf{BUFF} \mid \mathsf{put}\, f) \mid \mathsf{get}\, \tilde{v})$ se réduit en $(\mathbf{BUFF} \mid f\, \tilde{v})$.

Nous donnons également un exemple qui montre la différence avec la réduction dans $\boldsymbol{\pi}$. On peut coder dans le calcul bleu, un équivalent de la structure de donnée liste [97]. On définit deux références, nil pour la liste vide et cons pour le constructeur de listes. On déclare ces noms dans l'environnement global en mettant les deux ressources suivantes en parallèle avec tout processus

$$\langle \mathsf{nil} = (\boldsymbol{\lambda} nc) n\rangle \qquad\qquad \langle \mathsf{cons} = (\boldsymbol{\lambda} ht)(\boldsymbol{\lambda} nc) cht\rangle$$

On rajoute également une référence cond, pour le test conditionnel à vrai.

$$\langle \mathsf{cond} = (\boldsymbol{\lambda} bxy) bxy\rangle$$

Il faut noter qu'on peut considérer ces déclarations comme des fonctions. En effet, l'on fait attention à ne pas définir plusieurs déclarations sur le nom cond, et la déclaration est répliquée. Un appel à cond produira donc toujours le même résultat.

Le codage des listes se comprend de la manière suivante: une liste est une fonction à deux arguments, la «cellule nil»: n, et la «cellule cons»: c, qui retourne n si elle est la liste vide et qui retourne le «cons» de sa tête h et de sa queue t sinon. Ainsi, si on utilise la fonction de test sur une liste l en calculant $(\mathsf{cond}\, lxy)$, on obtient après réduction et élimination des déclarations inaccessibles

$$\begin{cases} x & \text{si } l = \mathsf{nil} \\ yht & \text{si } l = \mathsf{cons}\, ht \end{cases}$$

En utilisant cette remarque, on peut définir une ressource map qui permet d'appliquer une fonction à tous les éléments d'une liste

$$\langle \mathsf{map} = (\boldsymbol{\lambda} fu)(\boldsymbol{\nu} v)(\langle v \Leftarrow (\boldsymbol{\lambda} ht)(\boldsymbol{\nu} h't')(\langle h' \Leftarrow fh\rangle \mid \langle t' \Leftarrow \mathsf{map}\, ft\rangle \mid \mathsf{cons}\, h't')\rangle \mid \mathsf{cond}\, uuv\ )\rangle$$

En utilisant l'application d'ordre supérieur, on peut simplifier la lecture de ce processus. On obtient alors une nouvelle définition pour map, qui est très proche de celle usuellement donnée dans un langage fonctionnel.

$$\langle \mathsf{map} = (\boldsymbol{\lambda} fu)\mathsf{cond}\, uu((\boldsymbol{\lambda} ht)\mathsf{cons}\,(fh)(\mathsf{map}\, ft))\rangle$$

On peut encore simplifier cette définition en ajoutant une nouvelle construction

$$\left(\begin{array}{ll} \mathbf{case}\, u\, \mathbf{of} & \mathsf{nil} \Rightarrow P \\ & \mathsf{cons}(h,t) \Rightarrow Q \end{array}\right) =_{\text{def}} (\boldsymbol{\nu} pq)(\langle p \Leftarrow P\rangle \mid \langle q \Leftarrow (\boldsymbol{\lambda} ht) Q\rangle \mid \mathsf{cond}\, upq)$$

on peut alors réécrire la fonction map en

$$\langle \mathsf{map} = (\boldsymbol{\lambda} fu)(\mathrm{case}\, u\, \mathbf{of}\ \begin{array}{l} \mathsf{nil} \Rightarrow u \\ \mathsf{cons}(h,t) \Rightarrow \mathsf{cons}\,(fh)(\mathsf{map}\, ft))\rangle \end{array}$$

Supposons que, suivant le codage de Church, nous ayons défini les entiers et la fonction successeur suc. Il est maintenant possible d'utiliser des fonctions telle que $\langle f = (\mathsf{map}\,\mathsf{suc})\rangle$, et des messages «renvoyant» des fonctions, comme $(\mathsf{map}\,\mathsf{suc})$. On peut aussi utiliser ces fonctions avec un appel, comme dans le terme $(f\ u)$. Ce mécanisme, très proche de celui des langages fonctionnels, est moins rigide que le style de réduction de $\boldsymbol{\pi}$, qui est «par passage de canal». Ainsi, G. Boudol montre dans [22] que le π-calcul est un calcul par continuations, dans le sens où l'on n'a jamais qu'un accès indirect aux valeurs à travers leur nom. De plus il est nécessaire de manipuler explicitement les canaux chargés de porter les résultats/continuations.

On peut observer ces restrictions dans le cas du codage des listes dans $\boldsymbol{\pi}$. Dans cet exemple, la fonction map, par exemple, est codée comme le processus répliqué: $!\mathsf{map}(f,u)\ldots$ Le canal map attend donc deux valeurs lors de chaque communication. Ainsi le message $(\mathsf{map}\,\mathsf{suc})$, qui est valide dans le calcul bleu, est une erreur dans le π-calcul, qui est rejetée au typage des processus. En effet le message $\overline{\mathsf{map}}\langle\mathsf{suc}\rangle$ contient une erreur flagrante d'arité.

3.5.3 Exemples d'opérateurs impératif

Dans les langages fonctionnels tel que ML, la composition séquentielle de deux expressions est un opérateur dérivé obtenu à partir de l'application: soit y une variable non libre dans M, alors $(M\ ;\ N) =_{\text{def}} (({\boldsymbol{\lambda}}y)M)\,N$, ce qui s'écrit aussi: $(M\ ;\ N) =_{\text{def}} (\mathbf{let}\,y = N\,\mathbf{in}\,M)$. Cependant cette définition n'est pas correcte dans le calcul bleu, où l'évaluation est en appel par nom. Dans cette section, nous définissons deux opérateurs dérivés pour traiter le cas des évaluations séquentielles.

Définition 3.8 (Opérateurs séquentiels) Nous définissons les processus $(\mathbf{set}\,x = Q\ \mathbf{in}\,P)$ et $\mathbf{return}(P)$, par les termes suivants. Soit u une variable non libre dans P et Q.

$$\begin{array}{rcl} \mathbf{set}\,x = Q\ \mathbf{in}\,P & =_{\text{def}} & ({\boldsymbol{\nu}}u)(\langle u \Leftarrow ({\boldsymbol{\lambda}}x)P\rangle \mid Qu) \\ \mathbf{return}(P) & =_{\text{def}} & ({\boldsymbol{\lambda}}r)(rP) \end{array}$$

Dans la dernière définition, si P n'est pas un nom, on utilise implicitement notre notation pour l'application d'ordre supérieur. C'est-à-dire que: $\mathbf{return}(P) =_{\text{def}} \mathbf{def}\,u = P\,\mathbf{in}\,(({\boldsymbol{\lambda}}r)(ru))$.

L'opérateur $(\mathbf{set}\,x = Q\ \mathbf{in}\,P)$, permet de coder le mécanismes d'évaluation de l'opérateur let des langages fonctionnel. Ainsi, dans ce terme le processus Q peut s'évaluer, tandis que P reste bloqué. On considère que Q termine de s'évaluer lorsqu'il émet un nom sur le canal privé u, qui lui est passé dans l'application (Qu). Ce nom est alors lié dans P à l'argument x, et le terme P peut se réduire. L'opérateur $\mathbf{return}(P)$, est utile pour retourner un terme à un nom de continuation passé en argument. Par exemple, on a la réduction suivante:

$$\begin{array}{rcl} \mathbf{set}\,x = (Q \mid \mathbf{return}(v))\ \mathbf{in}\,P & =_{\text{def}} & ({\boldsymbol{\nu}}u)(\langle u \Leftarrow ({\boldsymbol{\lambda}}x)P\rangle \mid (Q \mid ({\boldsymbol{\lambda}}r)(rv))u) \\ & \to & ({\boldsymbol{\nu}}u)(\langle u \Leftarrow ({\boldsymbol{\lambda}}x)P\rangle \mid Q \mid (uv)) \\ & \overset{*}{\to} & Q \mid P\{v/x\} \end{array}$$

Ces opérateurs seront utilisés dans la partie IV, pour donner une interprétation d'un calcul d'objets ayant une sémantique en appel par valeur.

Discussion

Dans cette première partie, nous avons défini le calcul bleu, et nous avons étudié différents choix possibles pour la définition des ces opérateurs. Nous avons également introduit un calcul fonctionnel, $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$, qui représente un sous-ensemble séquentiel de termes du calcul bleu.

La définition du calcul bleu est entièrement due à G. Boudol, et on peut retrouver une présentation des différents papiers sur ce calcul dans [23]. La plus grande partie des résultats et des exemples présentés dans cette partie sont extraits de [22]. Cet article est un peu l'équivalent, pour le calcul bleu, du «tutorial» sur le π-calcul polyadique de R. Milner [97]. Le choix du parallèle dissymétrique s'est imposé assez récemment. En particulier parce qu'il règle des problèmes de typage. Il règle aussi des problèmes dans l'interprétation du λ-calcul, et du langage d'objets concurrent de P. Hankin et A. Gordon [60] (cf. chapitre 19). La condition de localité a été utilisée dans la totalité de mes articles [36, 37, 38], elle apparaît aussi en annexe de [22]. Il en est de même en ce qui concerne le choix de remplacer la déclaration répliquée: $\langle u = P \rangle$, par l'opérateur de définition: $(\mathbf{def}\, D\, \mathbf{in}\, P)$.

Dans la partie suivante, dévolue à l'étude des équivalences pour $\boldsymbol{\pi}^\star$, on utilise une version du calcul qui est symétrique et locale pour simplifier la présentation des résultats. Les définitions données pour ce calcul se transposent facilement au cas du calcul dissymétrique. On remplace également les déclarations répliquées par des définitions. C'est par exemple le calcul utilisé dans [21, 38].

Deuxième partie

Équivalence

CHAPITRE 4

Équivalence comportementale

> La quantité de sens compressé dans un petit espace par les symboles algébriques, sont une des conditions qui facilitent le raisonnement que nous sommes habitués à mener avec leur aide.
>
> – *Charles Babbage*

Dans la partie précédente, nous avons défini la syntaxe d'un nouveau calcul de processus. Se pose alors le problème de savoir quelle sémantique nous choisissons pour ce calcul, plus précisément, quelle notion d'équivalence choisissons nous entre processus bleus. Le problème se pose également de trouver des lois raisonnables et des techniques de preuve puissantes pour cette équivalence.

4.1 Choix de l'équivalence

Ce chapitre présente une définition d'équivalence comportementale entre termes du calcul bleu, à savoir la *congruence à barbes*, dénotée $\approx_b$. En l'absence de la définition d'une sémantique dénotationelle pour notre calcul, la sémantique de $\boldsymbol{\pi}^\star$ est donnée par cette équivalence.

Le choix d'une «bonne» équivalence est un problème important qui, dans le cas du π-calcul par exemple, a été à l'origine de multiples recherches. Nous choisissons de considérer la congruence à barbes qui, comme la relation homonyme de $\boldsymbol{\pi}$ [92], est une bisimulation [96] qui préserve un critère d'observabilité, appelé *barbes*, (cf. définition 4.1). Plus précisément, $\approx_b$ est la plus grande congruence qui préserve les barbes, et qui soit une bisimulation.

Dans cette partie, nous démontrons la validité des *lois de réplication* pour la congruence à barbes. Intuitivement, ces lois énoncent qu'une ressource privée et répliquée peut être distribuée sans danger entre plusieurs «acteurs». Par exemple, nous montrons que l'équivalence suivante est valide.

$$\mathbf{def}\, u = P \,\mathbf{in}\, (Q \mid R) \;\approx_b\; (\mathbf{def}\, u = P \,\mathbf{in}\, Q) \mid (\mathbf{def}\, u = P \,\mathbf{in}\, R)$$

Nous renvoyons le lecteur a l'énoncé du théorème 8.7, dans lequel nous donnons la liste des lois de réplication. Une propriété équivalente a été démontré pour le π-calcul par R. Milner [97, sect. 5.4]. Cependant, l'équivalence employée par l'auteur est la «*strong ground congruence*» [91], et la loi de réplication n'est valide que si le nom de la ressource (le nom u dans notre exemple)

n'apparaît ni en argument d'une émission – on dit que u n'apparaît pas en *position objet* d'une émission – ni en *position sujet* d'une réception.

Cette restriction est nécessaire: intuitivement, en l'absence de l'hypothèse de localité, il est possible de définir des contextes d'exécution qui différencient plus finement les noms reçus. Seulement récemment, D. Sangiorgi et M. Merro [89] ont prouvé la validité d'équivalences similaires pour la congruence à barbes dans le π-calcul local: $\mathbf{L}\boldsymbol{\pi}$, avec la condition annexe (moins restrictive) que u n'apparaît pas en sujet d'une réception.

Le choix de la congruence à barbes comme équivalence pour notre calcul est motivé par deux raisons principales. Tout d'abord, les équivalences à barbes constituent «*une base uniforme pour définir des équivalences comportementales entre différent calculs*» [15]. On peut donc espérer établir des comparaisons avec les équivalences de $\boldsymbol{\pi}$ et du *calcul join* [52, 55, 15]. Deuxièmement, le choix d'une équivalence «très générale» qui est à la fois une bisimulation et une congruence, facilite les preuves de correction des transformations de termes et des interprétations (des codages) que nous définissons dans cette thèse. Un exemple est donné dans la section 8.2, où nous donnons une interprétation du λ-calcul complet.

Néanmoins il y a un inconvénient majeur dans le choix de la congruence à barbes: les preuves d'équivalence nécessitent de considérer des quantifications sur tous les contextes. Un exemple de ce phénomène est donné dans la section 4.3.1, où nous donnons une preuve «directe» d'une loi de $\approx_b$. Pour éviter cet écueil, nous définissons un système de transitions étiquetées qui simule la réduction dans $\boldsymbol{\pi}^\star$, et une bisimulation associée à ce système. Nous montrons ensuite que cette bisimulation est plus fine que la congruence à barbes. Les lois valides pour la bisimulation sont donc aussi des lois de la congruence à barbes. Notre but, c'est-à-dire prouver les lois de réplication, est atteint en montrant que la bisimulation étiquetée vérifie les lois de réplications (cf. lemme 8.2).

La preuve du théorème de réplication ne constitue pas l'unique résultat de cette partie. La définition du système de transitions étiquées est intéressante en elle-même, car elle conduit à une meilleure compréhension du calcul bleu, et de ses caractères fonctionnels et «d'ordre supérieur».

Le reste de cette partie est organisé de la manière suivante. Dans la section 4.3 nous définissons l'équivalence à barbes. Dans le chapitre 5, nous définissons le sytème de transitions étiquées et la bisimulation associée. Dans les chapitres 6 et 7, nous prouvons la validité des techniques de preuve «up-to» [113] pour cette bisimulation. En employant ces techniques de preuve, nous montrons trois propriétés de la bisimulation: (*i*) c'est une congruence; (*ii*) elle contient l'équivalence structurelle; et (*iii*) elle vérifie les lois de réplication. Ces propriétés permettent de prouver, dans la section 8.1, que la bisimulation est incluse dans la congruence à barbes. Dans le chapitre 10, Nous définissons la relation d'expansion et la technique de preuve up-to associée. Nous concluons cette partie par quelques remarques générales.

4.2 Conventions

Dans cette partie, nous considérons une variante du calcul bleu local et symétrique. Dans le chapitre 9, nous montrons comment les définitions et les résultats que nous donnons dans cette partie, peuvent être transposés au cas du calcul bleu dissymétrique.

Nous apportons quelques modifications légères à la définition donnée dans la première partie. Ainsi nous remplaçons les déclarations répliquées par les définitions. En particulier, nous considérons les règles d'équivalence structurelle et de réduction données dans la figure 3.2. Nous considérons un sous-ensemble de références $p,q,\ldots$, qui sont utilisées dans les définitions. Nous désignons par le symbole D une association entre références et processus: $(p_1 = R_1, \ldots, p_n = R_n)$. C'est l'équivalent d'un environnement. Afin de simplifier la présentation de nos résultats, nous ne considérons que des définitions simple, et nous supposons que le terme $(\mathbf{def}\, D\, \mathbf{in}\, P)$ désigne le processus $\mathbf{def}\, p_1 = R_1\, \mathbf{in}\, (\ldots \mathbf{def}\, p_n = R_n\, \mathbf{in}\, P)$. Nous notons $\emptyset$ l'environnement vide et nous considérons que la définition vide $(\mathbf{def}\, \emptyset\, \mathbf{in}\, P)$ est identique à P. Nous introduisons aussi le pro-

cessus $\mathbf{0}$ directement dans la syntaxe des termes de $\boldsymbol{\pi}^\star$, ainsi que les deux règles d'équivalence structurelle suivantes:

$$(\mathbf{0} \mid P) \equiv P \qquad\qquad (\mathbf{0}\ a) \equiv \mathbf{0}$$

4.3 Congruence à barbes

La congruence à barbes, dénotée $\approx_b$, est la plus grande bisimulation qui préserve une notion de convergence entre processus et qui est une congruence [68]. Comme pour le λ-calcul ou $\boldsymbol{\lambda}\mathcal{D}$ef, on note $P \Downarrow$ le fait que le processus P converge (cf. définition 4.1), on dit aussi que P a une barbe, ou qu'il est *observable*. Néanmoins, contrairement au cas du λ-calcul, les valeurs ne sont pas uniquement les abstractions.

Pour les lecteurs familiers avec les bisimulations définies pour $\boldsymbol{\pi}$, disons que $\approx_b$ est une variante de la congruence à barbes faible [92]. On peut aussi rapprocher la définition de $\approx_b$, donnée dans la définition 4.3, de l'équivalence définie pour le calcul JOIN dans [15]. En particulier, comme pour le JOIN, nous considérons la plus grande bisimulation à barbes qui est une congruence, plutôt que la congruence la plus grande incluse dans la bisimulation à barbes.

Néanmoins, la définition de $\approx_b$ ne suit pas exactement celle de ses homonymes de $\boldsymbol{\pi}$. En effet, alors que les comportements observables considérés dans CCS ou $\boldsymbol{\pi}$ sont les émissions visibles (un choix équivalent est d'observer les noms libres en tête de messages dans $\boldsymbol{\pi}^\star$) nous choisissons à la place d'observer la présence de valeurs.

Notre intuition est qu'une valeur de base doit être une abstraction, comme dans le λ-calcul, ou comme dans $\boldsymbol{\lambda}\mathcal{D}$ef. De plus, comme nous sommes dans un calcul concurrent, nous considérons aussi qu'une valeur en parallèle avec un processus arbitraire est une valeur.

Définition 4.1 (Barbes) Une *barbe*, on dit aussi une *valeur*, est un processus engendré par la grammaire suivante.

$$V ::= (\boldsymbol{\lambda} x)P \mid [\,Q\,, l = P\,] \mid (V \mid P) \mid \ \mid (P \mid V) \mid (\boldsymbol{\nu}\, u)V \mid \mathbf{def}\, D\, \mathbf{in}\, V$$

Les valeurs sont les éléments «observables» de $\boldsymbol{\pi}^\star$. Nous dénotons $P\downarrow$ le fait que P soit une valeur. La version faible des barbes $P \Downarrow$, indique que P peut «converger» vers une valeur, c'est-à-dire qu'il existe une valeur V telle que $P \xrightarrow{*} V$.

Avec cette définition, une valeur est un processus qui peut se réduire quand elle est appliquée à un nom ou à une sélection. En particulier, l'enregistrement vide [ ] n'est pas une valeur. On remarque que la notion de valeur est préservée par modification structurelle.

La définition des barbes s'étend facilement aux contextes: nous disons que $\mathbf{C}\downarrow$ si et seulement si $\mathbf{C}[\mathbf{0}]\downarrow$. Il est facile de montrer que ceci implique $\mathbf{C}[P]\downarrow$ pour tout processus P. Avant de définir la relation de congruence à barbes, nous définissons la notion de relation *compositionelle*.

Définition 4.2 (fermeture sous tout contexte) La relation $\mathcal{D}$ est compositionelle, ou close sous tout contexte, si pour tout contexte $\mathbf{C}$, et couple $(P,Q) \in \mathcal{D}$, on a $(\mathbf{C}[P],\mathbf{C}[Q]) \in \mathcal{D}$.

La congruence à barbes est alors définie comme la plus grande des bisimulations compositionelles qui préservent les barbes.

Définition 4.3 (Congruence à barbes) La relation $\mathcal{D}$ est une *simulation à barbes faible*, si pour tout couple $(P,Q) \in \mathcal{D}$ on a:

(i) $P \to P'$ implique $Q \xrightarrow{*} Q'$ et $(P',Q') \in \mathcal{D}$;

(*ii*) $P\downarrow$ implique $Q\Downarrow$.
La relation $\mathcal{D}$ est une bisimulation à barbes faible si $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont des simulations à barbes faibles. Les processus P et Q sont équivalents: $P \approx_b Q$, si et seulement s'il existe une bisimulation à barbes faible $\mathcal{D}$ qui soit compositionelle, et telle que $P\,\mathcal{D}\,Q$.

4.3.1 Propriétés de la congruence à barbes

Dans cette section, nous présentons un exemple de preuve directe – et brutale – d'une loi de la congruence à barbes. Plus précisément, nous montrons que la béta-conversion est une loi «valide» de la congruence à barbes, c'est-à-dire que $(\lambda x)P)a \;\approx_b\; P\{^a/_x\}$ (cf. théorème 4.3). Cet exemple est intéressant, car il montre comment la preuve directe d'une propriété de la congruence à barbes demande de raisonner avec une quantification sur tous les contextes. Avant de prouver le théorème 4.3, nous commençons par définir deux propriétés intermédiaires. Dans le lemme 4.1, on étudie les interactions entre les barbes et les contextes. Dans la conjecture 4.2, nous indiquons comment on peut simplifier le raisonnement sur les réductions qu'un rédex peut exécuter sous un contexte. Nous ne faisons qu'ébaucher une preuve de cette dernière propriété, qui demanderait un développement plus important que ce que nous sommes prêts à lui dévouer dans cette partie. En effet, cette section a comme unique but de démontrer que les preuves directes pour $\approx_b$ sont compliquées. De plus, le théorème 4.3 peut être prouvé directement en utilisant la bisimulation étiquetée définie dans les chapitres 5 et suivants.

Lemme 4.1 (Barbes et Contextes) *Si le processus* $\mathbf{C}[P]$ *est une valeur, alors il y a deux cas. Soit:* (*i*) *le contexte* $\mathbf{C}$ *contient une valeur, c'est-à-dire* $\mathbf{C}\downarrow$*; soit* (*ii*) *le contexte* $\mathbf{C}$ *est un contexte d'évaluation et* P *est une valeur.*

Preuve Par induction sur la taille de $\mathbf{C}$. □

On définit la *profondeur* du contexte $\mathbf{C}$, dénotée $h(\mathbf{C})$, comme le nombre d'abstractions et de déclarations rencontrées lorsqu'on traverse $\mathbf{C}$ pour atteindre le trou $[\_]$. On incrémente aussi la profondeur si le trou est dans la partie objet d'une définition: $h(\mathbf{def\,p = C\,in\,P}) = h(\mathbf{C}) + 1$. Il est simple de voir que la profondeur est une grandeur préservée par manipulation structurelle, c'est-à-dire par les lois définissant $\equiv$. De plus $h(\mathbf{C})$ est nulle si et seulement si $\mathbf{C}$ est un contexte d'évaluation.

Conjecture 4.2 (Rédex et Contextes) *Si* $\mathbf{C}[(\lambda x)P)a] \to Q$*, alors il y a deux possibilités*

(*i*) ***la réduction est externe:*** *il existe un contexte* $\mathbf{D}$ *tel que* $Q \equiv \mathbf{D}[((\lambda x)P)a)\{^b/_y\}]$*, où* $\{^b/_y\}$ *peut être la substitution identité et où* $\mathbf{D}$ *peut être un contexte ayant deux trous, et tel que pour tout processus* R*, on a* $\mathbf{C}[R] \to \mathbf{D}[R\{^b/_y\}]$ *et* $h(\mathbf{D}) \leq h(\mathbf{C})$*;*

(*ii*) ***la réduction vient du rédex:*** $Q \equiv \mathbf{C}[P\{^a/_x\}]$.

Nous nous contentons de donner un schéma de preuve possible pour la conjecture 4.2. On peut montrer [22] que tout processus est structurellement équivalent à un processus de la forme suivante, que nous appelons *forme normale*:

$$(\nu\,\tilde{u})(\mathbf{def}\,D\,\mathbf{in}\,(V_1 \mid \cdots \mid V_m \mid M_1 \mid \cdots \mid M_s \mid \langle v_1 \Leftarrow R_1\rangle \mid \cdots \mid \langle v_r \Leftarrow R_r\rangle)) \tag{4.1}$$

où les V_i sont des abstractions ou des enregistrements ($(\lambda x)P$, $[\,]$, ou $[\,Q\,, l = P\,]$), et les M_i sont des applications ou des sélections ($(P\ a)$, avec P et Q en forme normale). On peut toujours supposer que la relation $\equiv$ est utilisée seulement au début et à la fin d'une preuve de réduction. C'est-à-dire que, pour chaque réduction $P \to P'$, il existe deux processus Q et Q' tels que: $P \equiv Q$, et $Q \to Q'$, et $Q' \equiv P'$, et tels que l'inférence $Q \to Q'$ n'utilise pas la règle (red struct). De plus, on peut supposer que Q est dans la forme normale définie par l'équation (4.1). Dans le cas de notre preuve, on choisit le contexte $\mathbf{C}$ sous forme normale, ce qui est suffisant pour prouver le cas plus général. Pour prouver notre conjecture, on peut utiliser une induction sur la taille de $\mathbf{C}$. Nous

n'étudions que deux des cas de cette induction. Le premier est le cas $\mathbf{C} =_{\text{def}} (\mathbf{D}b)$. Supposons que $\mathbf{C}[(\!(\boldsymbol{\lambda}x)\!)P)a] \to Q$. Il y a deux possibilités:

- **cas $\mathbf{D}[(\!(\boldsymbol{\lambda}x)\!)P)a]$ se réduit:** on utilise l'hypothèse d'induction;
- **cas il y a une béta-réduction impliquant b et le rédex est dans D:** si le trou $[\_]$ n'est pas dans le rédex, alors nous sommes dans le cas (i), avec $b = y$. Si $[\_]$ est dans le rédex, puisque $\mathbf{D}$ est en forme normale, on a $\mathbf{D}[(\!(\boldsymbol{\lambda}x)\!)P)a] \equiv (\!(\boldsymbol{\lambda}y)\!)\mathbf{B}[(\!(\boldsymbol{\lambda}x)\!)P)a])$, et par conséquent $Q \equiv \mathbf{B}[(\!(\boldsymbol{\lambda}x)\!)P)a]\{b/y\}$. C'est le cas (i).

Le second cas est tel que $\mathbf{C} =_{\text{def}} (\mathbf{def}\, p = \mathbf{D}\, \mathbf{in}\, \mathbf{E}[p])$, et tel que la réduction est:

$$\mathbf{C}[(\!(\boldsymbol{\lambda}x)\!)P)a] \to (\mathbf{def}\, p = \mathbf{D}[(\!(\boldsymbol{\lambda}x)\!)P)a]\, \mathbf{in}\, \mathbf{E}[\mathbf{D}[(\!(\boldsymbol{\lambda}x)\!)P)a]])$$

On est alors dans le cas (i) de la proposition – avec $b = y$ – et le contexte résultant a «deux trous». On remarque que la réduction de la définition est le seul cas qui duplique le trou dans un contexte.

Nous prouvons maintenant le résultat principal de cette section.

Théorème 4.3 (Béta-conversion) *la règle de béta-réduction est une loi de notre système, c'est-à-dire que* $(\!(\boldsymbol{\lambda}x)\!)P)a \approx_b P\{a/x\}$.

Preuve (Théorème 4.3) Soit $\mathcal{D}$ la relation définie par:

$$\mathcal{D} = \mathcal{I}d \cup \{(\mathbf{C}[(\!(\boldsymbol{\lambda}x)\!)P)a], \mathbf{C}[P\{a/x\}]) \mid \text{pour tout } \mathbf{C}, P \text{ et } a\}$$

La relation $\mathcal{D}$ est compositionelle par construction. Nous prouvons que $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont des simulations à barbes faibles. Soient P_1 et P_2 les processus $P_1 =_{\text{def}} \mathbf{C}[(\!(\boldsymbol{\lambda}x)\!)P)a]$ et $P_2 =_{\text{def}} \mathbf{C}[P\{a/x\}]$. Supposons que P_1 se réduise en P_1'. On montre qu'il existe un processus P_2' tel que P_2 se réduit en P_2' et $(P_1' \,\mathcal{D}\, P_2')$. En utilisant la conjecture 4.2, il existe deux cas. Soit $P_1' = P_2$, et on utilise le fait que $P_2 \,\mathcal{D}\, P_2$; soit $P_1' =_{\text{def}} \mathbf{D}[((\!(\boldsymbol{\lambda}x)\!)P)a)\{b/y\}]$. Dans ce cas P_2 peut se réduire en $P_2' =_{\text{def}} \mathbf{D}[P\{a/x\}\{b/y\}]$ et $P_1' \,\mathcal{D}\, P_2'$. La preuve est semblable dans le cas symétrique ou P_2 se réduit.

Nous prouvons maintenant que $(P_1, P_2) \in \mathcal{D}$ et $P_1 \downarrow$ (respectivement $P_2 \downarrow$) impliquent $P_2 \Downarrow$ (resp. $P_1 \Downarrow$).

- **cas $P_1 \downarrow$:** comme $(\!(\boldsymbol{\lambda}x)\!)P)a$ n'est pas une valeur, on a nécessairement (cf. lemme 4.1) $\mathbf{C}[R] \downarrow$ pour tout processus R, et donc $P_2 \downarrow$;
- **cas $P_2 \downarrow$:** il y a deux cas. (i) pour tout R, $\mathbf{C}[R] \downarrow$, et en particulier $P_1 \downarrow$; (ii) $\mathbf{C}$ est un contexte d'évaluation et $P\{a/x\} \downarrow$. Dans ce cas: $P_1 \to P_2$ et $P_2 \downarrow$, et par conséquent $P_1 \Downarrow$.

Par conséquent $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont des bisimulations faibles, et donc: $(P \,\mathcal{D}\, Q)$ implique que $P \approx_b Q$. Le résultat du théorème 4.3 s'obtient en choisissant $\mathbf{C} = [\_]$. □

On peut aussi montrer un résultat équivalent pour la communication – la proposition 4.4 ci-dessous – en utilisant une technique de preuve «brutale». Dans le chapitre 10, on pourra démontrer le même résultat plus simplement en utilisant la technique de preuve modulo expansion avec la bisimulation étiquetée.

Proposition 4.4 (Communication) *La communication avec une ressource privée est une loi de notre système:* $(\mathbf{def}\, p = R\, \mathbf{in}\, p) \approx_b (\mathbf{def}\, p = R\, \mathbf{in}\, R)$.

CHAPITRE 5

Transitions étiquetées

DANS CETTE SECTION, nous définissons un système de transitions étiquetées pour le calcul bleu. Nous rappelons que le terme $(P\ a)$ dénote soit une application $(P\ u)$, soit une sélection $(P{\cdot}l)$. Nous définissons quatre différents types d'actions dans le système de transitions étiquetées.

$$\mu ::= \tau \mid \lambda\, a \mid \mathbf{out}\, u.(\tilde{v};\tilde{a}) \mid \mathbf{in}\, u.(\tilde{b};\tilde{a})$$

Ces actions sont respectivement:

l'action silencieuse (τ): elle correspond à la synchronisation interne. Comme dans le π-calcul, deux processus en parallèle peuvent se synchroniser pendant une communication. Ce qui correspond à la règle (par com) du système de transitons étiquetées (s.t.e. en français, et l.t.s. en anglais). Nous considérons aussi deux autres mécanismes de synchronisation: la béta-réduction, qui correspond à la règle de réduction (red beta) et à la transition (app com); la communication avec une définition, qui correspond à la transition (def com);

l'action lambda ($\lambda\, a$): c'est l'action exécutée par les valeurs, λ-abstraction ou enregistrement, qui correspond aux transitions (lambda) du s.t.e. ;

l'action in ($\mathbf{in}\, u.(\tilde{b};\tilde{a})$): utilisée pour la réception des valeurs $\tilde{a}$ sur le canal u. Le tuple $\tilde{b}$ est utilisé pour mémoriser les applications/sélections qui encadrent le récepteur. Intuitivement, pour dériver la transition $(P\,\tilde{b}) \overset{\mathbf{in}\, u.(\epsilon;\tilde{a})}{\longmapsto} P'$, il faut d'abord dériver $P \overset{\mathbf{in}\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'$;

l'action out ($\mathbf{out}\, u.(\tilde{v};\tilde{a})$): utilisée pour l'émission des valeurs $\tilde{a}$ sur le canal u. Contrairement aux trois autres actions – pour lesquelles les transitions sont de la forme $P \overset{\mu}{\longmapsto} P'$ – les transitions de type **out** sont de la forme $P \overset{\mathbf{out}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')$. Dans cette transition, le tuple $\tilde{v}$ représente l'ensemble des noms de $\tilde{a}$ qui étaient restreints, et qui ont été ouverts (scope extrusion). L'environnement D, qui est de la forme $(p_1 = R_1),\ldots,(p_n = R_n)$, correspond aux définitions ouvertes.

Pour distinguer les différentes actions, nous employons la notion de *sorte de l'action* μ, dénoté $\kappa(\mu)$, et qui appartient à l'ensemble $\{\tau,\lambda,\mathbf{out},\mathbf{in}\}$. Les transitions de notre système, définies dans la section 5.1, se partagent de façon grossière en deux catégories. La première concerne les transitions de la forme $P \overset{\mu}{\longmapsto} P'$, avec $\kappa(\mu) \in \{\tau,\lambda,\mathbf{in}\}$. La seconde concerne les émissions: $P \overset{\mathbf{out}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')$. Dans cette transition, D désigne un environnement $(p_1 = R_1),\ldots,(p_n = R_n)$, et P' est un processus. Nous utiliserons le symbole $\emptyset$ lorsque D est l'environnement vide, et ϵ pour le tuple vide.

Remarque Dans l'action **out** $u.(\tilde{v};\tilde{a})$, nous considérons implicitement que le tuple des noms restreints $\tilde{v}$ n'est pas ordonné, c'est-à-dire qu'on le considère comme un ensemble. Ceci est important pour prouver, par exemple, que les processus $(\boldsymbol{\nu} u)(\boldsymbol{\nu} v)P$ et $(\boldsymbol{\nu} v)(\boldsymbol{\nu} u)P$ sont bisimilaires. ■

Pour simplifier la présentation des règles de transitions, et les preuves, nous omettons de donner le symétrique des règles de transitions pour la composition parallèle. Nous considérerons aussi que les noms liés sont toujours uniques et différents les uns des autres. C'est-à-dire que nous considérons les actions modulo α-conversion. Il serait suffisant de choisir une représentation implicite des noms basées sur les indices de De Bruijn [41], par exemple, pour obtenir que les noms liés soient uniques. Néanmoins nous préférons ne pas considérer explicitement ce mécanisme. En effet, l'utilisation des indices à la De Bruijn est déjà compliquée dans les preuves pour le λ-calcul, et cela devient encore pire pour le π-calcul. Il suffit de lire les travaux de D. Hirschkoff [67] sur les preuves de bisimulation assistée par ordinateur pour le voir.

Afin de donner une intuition des règles de transitions présentées dans la section 5.1, on peut donner des relations entre actions et transitions étiquetées. Ces résultats sont extraits du lemme 8.3, qui est prouvé au chapitre 8. Ainsi, (i) si $P \stackrel{\tau}{\longmapsto} P'$ et $\mathbf{E}$ est un contexte d'évaluation, alors $\mathbf{E}[P] \stackrel{\tau}{\longmapsto} \mathbf{E}[P']$; (ii) si $P \stackrel{\lambda\, a}{\longmapsto} P'$, alors $(Pa) \to P'$; (iii) si $P \stackrel{\mathbf{in}\ u.(\tilde{b};\tilde{a})}{\longmapsto} P'$, alors $(P\tilde{b}) \mid (u\tilde{a}) \to P'$. Et finalement (iv) si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')$, alors il existe un processus R tel que $P \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R \mid u\tilde{a}))$, et $(\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, R) \sim_d (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, P')$, où $\sim_d$ est la relation de bisimulation forte définie à la fin de ce chapitre.

Les règles de transitions présentées dans la section suivante ne sont pas usuelles, même si l'on met de côté les règles concernant l'action λ. Une partie de la complexité du système de transition résulte de la nature «polyadique» du calcul: plus d'un nom peut être échangé pendant une communication. Une autre source de complexité vient de la nature d'ordre supérieur des actions **out**. Ainsi, dans une transition $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')$ par exemple, l'ensemble D fait d'une certaine manière partie de l'action. Par conséquent nous sommes dans une situation comparable aux systèmes de transitions où les processus apparaissent dans les actions [117]. Notre s.t.e. se distingue des systèmes de transitions du π-calcul pour d'autres raisons: les τ-transitions ne sont pas équivalentes aux réductions obtenues avec la relation $\to$. Nous avons par exemple les relations suivantes:

$$\langle u \Leftarrow Q\rangle \mid \mathbf{def}\, p = R\, \mathbf{in}\, (u\ p \mid P) \quad \begin{cases} \to & \mathbf{def}\, p = R\, \mathbf{in}\, ((Q\ p) \mid P) \\ \stackrel{\tau}{\longmapsto} \equiv & \mathbf{def}\, p = R\, \mathbf{in}\, ((Q\ p) \mid (\mathbf{def}\, p = R\, \mathbf{in}\, P)) \end{cases} \tag{5.1}$$

La source de la différence entre les deux relations: transition et réduction, réside dans les règles de transitions des actions **out** et, plus particulièrement, la règle (def open) concernant la définition. Intuitivement, nous interdisons que le nom d'une ressource privée soit transmis à l'extérieur de sa portée, à la place nous créons une copie de cette ressource avec un nouveau nom, et nous transmettons ce nouveau nom.

5.1 Définition du système de transitions étiquetées

Axiomes

$$(\boldsymbol{\lambda} x)P \stackrel{\lambda\, u}{\longmapsto} P\{u/_x\} \quad \text{(lambda)} \qquad a \stackrel{\mathbf{out}\ u.(\epsilon;\epsilon)}{\longmapsto} (\emptyset; \mathbf{0}) \quad \text{(out)}$$

$$[\, R\, , l = N\,] \stackrel{\lambda\, l}{\longmapsto} N \quad \text{(lambda)} \qquad [\, R\, , l = N\,] \stackrel{\lambda\, k}{\longmapsto} (R \cdot k) \ (\text{if } k \neq l) \quad \text{(lambda)}$$

$$\langle u \Leftarrow P\rangle \stackrel{\mathbf{in}\ u.(\tilde{b};\tilde{a})}{\longmapsto} (P\ \tilde{a}) \quad \text{(decl)}$$

Règles pour les actions τ, λ et in

$$\frac{P \vdash^{\lambda\, a}\rightarrow P'}{P \mid Q \vdash^{\lambda\, a}\rightarrow P' \mid (Q\ a)}\ \text{(par lambda)} \qquad \frac{P \vdash^{\lambda\, a}\rightarrow P'}{P\ a \vdash^{\tau}\rightarrow P'}\ \text{(app com)}$$

$$\frac{P \vdash^{\tau}\rightarrow P'}{P \mid Q \vdash^{\tau}\rightarrow P' \mid Q}\ \text{(par tau)} \qquad \frac{P \vdash^{\tau}\rightarrow P'}{(P\ a) \vdash^{\tau}\rightarrow (P'\ a)}\ \text{(app tau)}$$

$$\frac{P \vdash^{\mu}\rightarrow P' \quad (\kappa(\mu) \in \{\tau,\lambda,\mathbf{in}\}\ \&\ u \notin \mathbf{fn}(\mu))}{(\boldsymbol{\nu} u)P \vdash^{\mu}\rightarrow (\boldsymbol{\nu} u)P'}\ \text{(new mu)}$$

$$\frac{P \vdash^{\mu}\rightarrow P' \quad (\kappa(\mu) \in \{\tau,\lambda,\mathbf{in}\}\ \&\ p \notin \mathbf{fn}(\mu))}{\mathbf{def}\, p = R\, \mathbf{in}\, P \vdash^{\mu}\rightarrow \mathbf{def}\, p = R\, \mathbf{in}\, P'}\ \text{(def mu)}$$

Règles pour l'action in

$$\frac{P \vdash^{\mathbf{in}\ u.(\tilde{b};\tilde{a})}\rightarrow P'}{P \mid Q \vdash^{\mathbf{in}\ u.(\tilde{b};\tilde{a})}\rightarrow P' \mid (Q\ \tilde{b})}\ \text{(par in)} \qquad \frac{P \vdash^{\mathbf{in}\ u.((c,\tilde{b});\tilde{a})}\rightarrow P'}{(P\ c) \vdash^{\mathbf{in}\ u.(\tilde{b};\tilde{a})}\rightarrow P'}\ \text{(app in)}$$

Règles pour l'action out

$$\frac{P \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow (D; P') \quad (\tilde{v},\mathbf{decl}(D)) \cap \mathbf{fn}(Q) = \emptyset}{P \mid Q \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow (D; (P' \mid Q))}\ \text{(par out)}$$

$$\frac{P \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow (D; P') \quad c \notin (\tilde{v},\mathbf{decl}(D))}{(P\ c) \vdash^{\mathbf{out}\ u.(\tilde{v};(\tilde{a},c))}\rightarrow (D; (P'\ c))}\ \text{(app out)}$$

$$\frac{P \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow (D; P') \quad (v \notin u,\tilde{v},\tilde{a},\mathbf{fn}(D))}{(\boldsymbol{\nu} v)P \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow (D; (\boldsymbol{\nu} v)P')}\ \text{(new out)}$$

$$\frac{P \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow (D; P') \quad (u \neq v)}{(\boldsymbol{\nu} v)P \vdash^{\mathbf{out}\ u.((v,\tilde{v});\tilde{a})}\rightarrow (D; P')}\ \text{(new open)}$$

$$\frac{P \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow (D; P') \quad (p \notin u,\tilde{a},\tilde{v},\mathbf{fn}(D))}{\mathbf{def}\, p = R\, \mathbf{in}\, P \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow (D; \mathbf{def}\, p = R\, \mathbf{in}\, P')}\ \text{(def out)}$$

$$\frac{P \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow (D; P') \quad (p \neq u\ \&\ p \notin (\tilde{v},\mathbf{decl}(D)))}{\mathbf{def}\, p = R\, \mathbf{in}\, P \vdash^{\mathbf{out}\ u.(\tilde{v};\tilde{a})}\rightarrow ((p = R,D); \mathbf{def}\, p = R\, \mathbf{in}\, P')}\ \text{(def open)}$$

Synchronisation

$$\frac{P \overset{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;P') \qquad Q \overset{\mathbf{in}\ u.(\epsilon;\tilde{a})}{\longmapsto} Q'}{P \mid Q \overset{\tau}{\longmapsto} (\boldsymbol{\nu}\,\tilde{v})(\mathbf{def}\,D\,\mathbf{in}\,(P' \mid Q'))}\ \text{(par com)}$$

$$\frac{P \overset{\mathbf{out}\ p.(\tilde{v};\tilde{a})}{\longmapsto} (D;P')}{\mathbf{def}\,p = R\,\mathbf{in}\,P \overset{\tau}{\longmapsto} \mathbf{def}\,p = R\,\mathbf{in}\,(\boldsymbol{\nu}\,\tilde{v})(\mathbf{def}\,D\,\mathbf{in}\,(R\ \tilde{a} \mid P'))}\ \text{(def com)}$$

5.2 Définition de la def-bisimulation

En utilisant le système de transitions étiquetées, il est possible de définir une *bisimulation étiquetée*, que nous notons $\sim_d$. Nous appelons cette équivalence la *def-bisimulation*. Nous commençons par donner quelques définitions.

Définition 5.1 (Tuples de noms def-similaires) Pour chaque relation binaire entre processus $\mathcal{D}$, on note $(\mathbf{def}\,D\,\mathbf{in}\,\tilde{a})\,\mathcal{D}\,(\mathbf{def}\,D'\,\mathbf{in}\,\tilde{b})$ le fait que les noms des tuples $\tilde{a}$ et $\tilde{b}$ soient dans $\mathcal{D}$, c'est-à-dire, plus précisément, que les deux propriétés suivantes soient vraies.

- (i) $\tilde{a}$ et $\tilde{b}$ ont la même taille, disons n, *i.e.*: $|\tilde{a}| = |\tilde{b}| = n$;
- (ii) pour tout indice i dans l'intervalle $[1..n]$, les noms a_i et b_i sont liés à des ressources en relation $\mathcal{D}$: $(\mathbf{def}\,D\,\mathbf{in}\,a_i)\,\mathcal{D}\,(\mathbf{def}\,D'\,\mathbf{in}\,b_i)$.

La relation de transition faible $\Rightarrow$ est la fermeture réflexive et transitive de la relation $\overset{\tau}{\longmapsto}$. Nous utilisons la notation $P \overset{\mu}{\Rightarrow} Q$, s'il existe deux processus R et S tels que $P \Rightarrow R \overset{\mu}{\longmapsto} S \Rightarrow Q$. Nous notons $P \overset{\widehat{\mu}}{\longmapsto} Q$ la relation telle que: $P \overset{\mu}{\longmapsto} Q$ si $\kappa(\mu) \neq \tau$, et $P = Q$ ou $P \overset{\tau}{\longmapsto} Q$ si $\kappa(\mu) = \tau$. Nous utilisons aussi la relation $\overset{\hat{\mu}}{\Longrightarrow}$ pour désigner $P \overset{\mu}{\Rightarrow} Q$ si $\kappa(\mu) \neq \tau$, et $P \Rightarrow Q$ sinon.

La def-bisimulation, que nous définissons dans la définition 5.2, est une équivalence qui égalise les processus dans lesquels on a «répliqué» les définitions. Par exemple, nous montrons que la def-bisimulation égalise les deux processus obtenus par réduction et τ-transition dans l'équation (5.1):

$$\mathbf{def}\,p = R\,\mathbf{in}\,((Q\ p) \mid P)\ \sim_d\ \mathbf{def}\,p = R\,\mathbf{in}\,((Q\ p) \mid (\mathbf{def}\,p = R\,\mathbf{in}\,P))$$

Dans les sections 6 et 8, nous montrons que cette équivalence est incluse dans la congruence à barbes.

Définition 5.2 (*d*-simulation et def-bisimulation) La relation d'équivalence $\mathcal{D}$ est une *d*-simulation forte, si pour tout couple $(P,Q) \in \mathcal{D}$, les trois propriétés suivantes sont vérifiées.

- (i) si $P \overset{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \overset{\mu}{\longmapsto} Q'$ et $P'\,\mathcal{D}\,Q'$;
- (ii) si $P \overset{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D;P')$, alors il existe un processus Q' et un environnement D' tels que: $Q \overset{\mathbf{out}\ u.(\tilde{v};\tilde{c})}{\longmapsto} (D';Q')$, et $P'\,\mathcal{D}\,Q'$, et $(\mathbf{def}\,D\,\mathbf{in}\,\tilde{b})\,\mathcal{D}\,(\mathbf{def}\,D'\,\mathbf{in}\,\tilde{c})$
- (iii) pour toute substitution σ – de variables par des noms – on a $P\sigma\,\mathcal{D}\,Q\sigma$.

Une équivalence qui vérifie les propriétés (i) et (ii) est une simulation *ground*. La relation $\mathcal{D}$ est une *d*-bisimulation forte si $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont des *d*-simulations fortes. La def-bisimulation forte $\sim_d$, est la relation telle que $P \sim_d Q$ s'il existe une *d*-bisimulation forte $\mathcal{D}$ telle que $P\,\mathcal{D}\,Q$. On peut aussi définir la relation de *d*-simulation faible de la manière usuelle, il suffit de remplacer les propriétés (i) et (ii) par:

- (i') si $P \overset{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \overset{\hat{\mu}}{\Longrightarrow} Q'$ et $P'\,\mathcal{D}\,Q'$;
- (ii') si $P \overset{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D;P')$, alors il existe un processus Q' et un environnement D' tels que: $Q \overset{\mathbf{out}\ u.(\tilde{v};\,\tilde{c})}{\Longrightarrow} (D';Q')$, et $P'\,\mathcal{D}\,Q'$, et $(\mathbf{def}\,D\,\mathbf{in}\,\tilde{b})\,\mathcal{D}\,(\mathbf{def}\,D'\,\mathbf{in}\,\tilde{c})$

On dénote $\approx_d$ la def-bisimulation faible associée à cette notion de d-simulation.

On montre de manière simple pourquoi la def-bisimulation ne caractérise pas la congruence à barbes. Une loi algébrique importante du π-calcul asynchrone [68, 6], qui est fausse dans le calcul synchrone, est: $u(x).\bar{u}x = \mathbf{0}$. Dans le calcul bleu, on montre la loi équivalente, c'est-à-dire : $\langle u \Leftarrow u \rangle \approx_b \mathbf{0}$. Or de manière évidente $\langle u \Leftarrow u \rangle \not\approx_d \mathbf{0}$, puisque le premier processus peut faire une action **in**, et que $\mathbf{0}$ est inerte. Par conséquent, un premier perfectionnement de notre définition serait de définir une bisimulation asynchrone, comme il a été fait dans [6].

Avec une bisimulation asynchrone, la relation $\langle u \Leftarrow u \rangle \approx \mathbf{0}$ est vérifiée. Néanmoins nous n'avons pas besoin de cette équivalence dans le cadre de notre étude. En effet, le processus $\langle u \Leftarrow u \rangle$ crée un lien inutile entre un nom et lui-même, et c'est un processus que nous ne rencontrons nulle part dans les exemple pratiques. A l'opposé, on montre que l'exemple du processus (**def** $p = u$ **in** P): «l'égalisateur» de p et u, est tel que (**def** $p = u$ **in** P) $\approx_d P\{u/p\}$.

Il y a d'autre exemples de processus non def-bisimilaires – bien qu'équivalents – selon $\approx_b$. Soient P et Q les deux processus

$$\begin{array}{lll} P & =_{\text{def}} & (\boldsymbol{\nu}\, d)(\mathbf{def}\, p = (\boldsymbol{\nu}\, c)\langle c = d\rangle \,\mathbf{in}\, (\langle d \Leftarrow R_1\rangle \mid up)) \\ Q & =_{\text{def}} & (\boldsymbol{\nu}\, d)(\mathbf{def}\, p = \mathbf{0} \,\mathbf{in}\, (\langle d \Leftarrow R_2\rangle \mid up)) \end{array}$$

Il est clair que personne ne peut communiquer sur le nom c avec le processus $(\boldsymbol{\nu}\, c)\langle c = d\rangle$, et donc $(\boldsymbol{\nu}\, c)\langle c = d\rangle \approx_b \mathbf{0}$. Par conséquent, puisque le nom d reste privé dans P, personne ne peut communiquer avec la déclaration $\langle d \Leftarrow R\rangle$. Néanmoins le processus P peut réaliser l'action **out** $u.(d; p)$, alors que la transition similaire dans Q utilise l'action **out** $u.(\epsilon; p)$, c'est-à-dire qu'on peut observer l'extrusion d'un nom privé dans la transition de P, mais pas dans celle de Q.

Ce dernier exemple est un cas typique de ce qui arrive dans les calculs d'ordre supérieur, où des processus au lieu des noms sont échangés durant une communication. Par exemple, le processus P peut s'interpréter comme le terme du π-calcul d'ordre supérieur [114] suivant: $(\boldsymbol{\nu}\, d)(\bar{u}\langle(\boldsymbol{\nu}\, c)(c().\bar{d}\langle\rangle)\rangle \mid \dots)$, et Q comme le terme $(\boldsymbol{\nu}\, d)(\bar{u}\langle\mathbf{0}\rangle \mid \dots)$. Cependant, comme pour le problème de l'extension asynchrone de $\sim_d$, nous ne rencontrerons pas de processus de ce type dans notre étude.

CHAPITRE 6

Bisimulations up-to

DANS CE CHAPITRE, nous étudions la technique de preuve modulo contextes, nous disons aussi «*up-to contextes*» [113], et nous montrons qu'une bisimulation modulo contextes est une def-bisimulation. Un des intérêts de cette technique de preuve, est qu'elle permet d'éviter une preuve directe du fait que $\sim_d$ est une congruence. D'autres propriétés découlent de la preuve du théorème 6.1, elles sont énumérées dans le chapitre 8.

Nous commençons par définir la notion de preuve modulo contextes. Cette notion nécessite de définir la fermeture sous tout contextes $\mathcal{D}_{\rm C}$, d'une relation $\mathcal{D}$. Plus exactement, $\mathcal{D}_{\rm C}$ est la plus petite relation compositionelle qui contient la fermeture transitive et réflexive de $\mathcal{D}$, c'est-à-dire que pour tout processus P,Q,R,S, on a:

- si $P\,\mathcal{D}_{\rm C}\,Q$, alors $(\boldsymbol{\nu} u)P\,\mathcal{D}_{\rm C}\,(\boldsymbol{\nu} u)Q$, et $(P\,a)\,\mathcal{D}_{\rm C}\,(Q\,a)$, et $(\boldsymbol{\lambda} x)P\,\mathcal{D}_{\rm C}\,(\boldsymbol{\lambda} x)Q$, et $\langle u \Leftarrow P\rangle\,\mathcal{D}_{\rm C}\,\langle u \Leftarrow Q\rangle$;
- si $P\,\mathcal{D}_{\rm C}\,Q$ et $R\,\mathcal{D}_{\rm C}\,S$, alors $(P \mid R)\,\mathcal{D}_{\rm C}\,(Q \mid S)$, et $(\mathbf{def}\,p = P\,\mathbf{in}\,R)\,\mathcal{D}_{\rm C}\,(\mathbf{def}\,p = Q\,\mathbf{in}\,S)$, et $[\,P\,, l = R\,]\,\mathcal{D}_{\rm C}\,[\,Q\,, l = S\,]$;
- si $P\,\mathcal{D}\,Q$, alors $P\,\mathcal{D}_{\rm C}\,Q$;
- si $P\,\mathcal{D}_{\rm C}\,Q$ et $Q\,\mathcal{D}_{\rm C}\,R$, alors $P\,\mathcal{D}_{\rm C}\,R$;
- $P\,\mathcal{D}_{\rm C}\,P$.

Définition 6.1 (Simulation ground modulo contextes) La relation $\mathcal{D}$ est une simulation ground *modulo contextes*, si pour tout couple $(P,Q) \in \mathcal{D}$ on a:

(i) si $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \stackrel{\mu}{\longmapsto} Q'$ et $P'\,\mathcal{D}_{\rm C}\,Q'$;

(ii) si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')$, alors il existe un processus Q' et un environnement D' tels que: $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D'; Q')$ avec $P'\,\mathcal{D}_{\rm C}\,Q'$ et $(\mathbf{def}\,D\,\mathbf{in}\,\tilde{a})\,\mathcal{D}_{\rm C}\,(\mathbf{def}\,D'\,\mathbf{in}\,\tilde{b})$.

De manière identique, on peut définir une notion de simulation modulo équivalence structurelle. Nous notons $\mathcal{D}_{\equiv}$ la relation définie par: $\mathcal{D}_{\equiv} =_{\rm def} \{(P,Q)\,|\exists (P',Q') \in \mathcal{D}\,.P \equiv P'\ \&\ Q \equiv Q'\}$.

Définition 6.2 (Simulation ground modulo équivalence structurelle) La relation $\mathcal{D}$ est une simulation ground *modulo* $\equiv$, si pour tout couple $(P,Q) \in \mathcal{D}$ on a:

(i) si $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \stackrel{\mu}{\longmapsto} Q'$ et $P'\,\mathcal{D}_{\equiv}\,Q'$;

(*ii*) si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D \mid P')$, alors il existe un processus Q' et un environnement D' tels que: $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D';Q')$, avec $P'\,\mathcal{D}_{\equiv}\,Q'$ et $(\mathbf{def}\ D\ \mathbf{in}\ \tilde{a})\,\mathcal{D}_{\equiv}\,(\mathbf{def}\ D'\ \mathbf{in}\ \tilde{b})$.

Nous définissons une troisième notion de simulation «up-to», la *simulation modulo réplication*. Pour toute relation $\mathcal{D}$, nous définissons $\mathcal{D}_R$ comme la plus petite relation d'équivalence qui contient $\mathcal{D}$, et qui est close par les lois de réplication (cf. lemme 8.2), c'est-à-dire que pour tout processus P,Q,R,S, on a:

- $P\,\mathcal{D}_R\,P$, et si $P\,\mathcal{D}_R\,Q$, alors $Q\,\mathcal{D}_R\,P$, et si $P\,\mathcal{D}_R\,Q$ et $Q\,\mathcal{D}_R\,R$, alors $P\,\mathcal{D}_R\,R$;
- si $P\,\mathcal{D}\,Q$, alors $P\,\mathcal{D}_R\,Q$;
- si $(\mathbf{def}\ D\ \mathbf{in}\ a)\,\mathcal{D}_R\,(\mathbf{def}\ D'\ \mathbf{in}\ b)$, et si $\mathbf{fn}(P) \cap (\mathbf{decl}(D) \cup \mathbf{decl}(D')) = \emptyset$, alors: $(\mathbf{def}\ D\ \mathbf{in}\ (P a))\,\mathcal{D}_R\,(\mathbf{def}\ D'\ \mathbf{in}\ (P b))$;
- pour tout processus P et Q:

$$
\begin{array}{rcll}
(\mathbf{def}\ p = R\ \mathbf{in}\ (P \mid Q)) & \mathcal{D}_R & (\mathbf{def}\ p = R\ \mathbf{in}\ ((\mathbf{def}\ p = R\ \mathbf{in}\ P) \mid Q)) & \\
(\mathbf{def}\ p = R\ \mathbf{in}\ (P \mid Q)) & \mathcal{D}_R & (\mathbf{def}\ p = R\ \mathbf{in}\ (P \mid (\mathbf{def}\ p = R\ \mathbf{in}\ Q))) & \\
(\mathbf{def}\ p = R\ \mathbf{in}\ (P p)) & \mathcal{D}_R & (\mathbf{def}\ p = R\ \mathbf{in}\ ((\mathbf{def}\ p = R\ \mathbf{in}\ P) p)) & \\
(\mathbf{def}\ p = R\ \mathbf{in}\ (\lambda x) P)) & \mathcal{D}_R & ((\boldsymbol{\lambda} x)(\mathbf{def}\ p = R\ \mathbf{in}\ P)) & (x \notin \mathbf{fn}(R)) \\
(\mathbf{def}\ p = R\ \mathbf{in}\ (\langle u \Leftarrow P\rangle)) & \mathcal{D}_R & (\langle u \Leftarrow (\mathbf{def}\ p = R\ \mathbf{in}\ P)\rangle) & \\
(\mathbf{def}\ p = R,q = S\ \mathbf{in}\ P) & \mathcal{D}_R & (\mathbf{def}\ p = R,q = S\ \mathbf{in}\ (\mathbf{def}\ p = R\ \mathbf{in}\ P)) & (q \notin \mathbf{fn}(R)) \\
(\mathbf{def}\ p = R,q = S\ \mathbf{in}\ P) & \mathcal{D}_R & (\mathbf{def}\ p = R,q = (\mathbf{def}\ p = R\ \mathbf{in}\ S)\ \mathbf{in}\ P) & (q \notin \mathbf{fn}(R))
\end{array}
$$

La relation $\mathcal{D}_R$ permet de définir la notion de simulation modulo réplication.

Définition 6.3 (Simulation ground modulo réplication) La relation $\mathcal{D}$ est une simulation ground modulo réplication, si pour tout couple $(P,Q) \in \mathcal{D}$ on a:

(*i*) si $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \stackrel{\mu}{\longmapsto} Q'$ and $P'\,\mathcal{D}_R\,Q'$;

(*ii*) si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;P')$, alors il existe un processus Q' et un environnement D' tels que: $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D';Q')$ avec $P'\,\mathcal{D}_R\,Q'$ et $(\mathbf{def}\ D\ \mathbf{in}\ \tilde{a})\,\mathcal{D}_R\,(\mathbf{def}\ D'\ \mathbf{in}\ \tilde{b})$.

Nous montrons que les relations de bisimulations up-to sont aussi des def-bisimulations.

Théorème 6.1 (Preuve «up-to») *Soit $\mathcal{D}$ une relation close par substitution des variables par des noms. (i) : si $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont des simulation ground modulo contextes, alors $\mathcal{D}$ est une def-bisimulation, c'est-à-dire que $\mathcal{D} \subseteq \sim_d$. ($ii$) : si $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont des simulation ground modulo équivalence structurelle, alors $\mathcal{D}$ est une def-bisimulation. (iii) : si $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont des simulation ground modulo réplication, alors $\mathcal{D}$ est une def-bisimulation.*

La preuve complète du théorème 6.1 est donné dans le chapitre 7. Cette preuve est longue et assez technique, aussi, dans ce chapitre, nous ne faisons que donner le schéma de la preuve. Disons simplement que cette preuve utilise une fonction annexe $F(X)$, que nous définissons ci-dessous.

$$
\begin{array}{rcl}
F(X) & = & X \cup \sim_d \cup \{(P,Q) \mid \exists R.(P,R) \in X \text{ and } (R,Q) \in X\} \\
 & & \cup \{((\boldsymbol{\nu} u)P,(\boldsymbol{\nu} u)Q) \mid (P,Q) \in X\} \\
 & & \cup\ \{((P\ a),(Q\ a)) \mid (P,Q) \in X\} \\
 & & \cup\ \{(\langle u \Leftarrow P\rangle,\langle u \Leftarrow Q\rangle) \mid (P,Q) \in X\} \\
 & & \cup\ \{((\boldsymbol{\lambda} x)P,(\boldsymbol{\lambda} x)Q) \mid (P,Q) \in X\} \\
 & & \cup\ \{([P_1, l = P_2],[Q_1, l = Q_2]) \mid (P_1,Q_1) \text{ and } (P_2,Q_2) \in X\} \\
 & & \cup\ \{(P_1 \mid P_2,Q_1 \mid Q_2) \mid (P_1,Q_1) \text{ and } (P_2,Q_2) \in X\} \\
 & & \cup\ \{(\mathbf{def}\ p = P_1\ \mathbf{in}\ P_2,\mathbf{def}\ p = Q_1\ \mathbf{in}\ Q_2) \mid (P_1,Q_1) \text{ and } (P_2,Q_2) \in X\} \\
 & & \cup\ D(X) \cup \mathcal{E}
\end{array}
$$

Dans ce chapitre, nous ne définirons pas la relation $\mathcal{E}$, ni la fonction D. On peut trouver ces définitions dans le chapitre 7.

On remarque que $F(X)$ est une fonction qui, étant donné une relation X, construit la fermeture de X «sous tout les opérateurs de $\boldsymbol{\pi}^\star$». En particulier, il est clair que pour toute relation $\mathcal{D}$, la relation $\bigcup_{n\geq 0} F^n(\mathcal{D})$ (existe et) est compositionelle et qu'elle vérifie la relation:

$$\mathcal{D}_{\mathrm{C}} \subseteq \bigcup_{n\geq 0} F^n(\mathcal{D})$$

On peut voir dans F un exemple de fonction «*respectful*», telle que définie par D. Sangiorgi dans [122]. Les relations $D(X)$ et $\mathcal{E}$, que nous ne définissons pas ici, permettent quant à elles de construire la fermeture de X sous les lois de réplication et d'équivalence structurelle. Disons simplement que la présence de la fonction D, dans la définition de F, permet de prouver que $\mathcal{D}_R \subseteq \bigcup_{n\geq 0} F^n(\mathcal{D})$, tandis que $\mathcal{E}$ est une relation telle que:

$$\mathcal{D}_{\equiv} \subseteq \bigcup_{n\geq 0} F^n(\mathcal{D}) \qquad \text{et} \qquad \equiv \subseteq \bigcup_{n\geq 0} F^n(\mathcal{E})$$

Nous dénotons $\mathcal{D}_\infty$ la relation définie par $\mathcal{D}_\infty =_{\mathrm{def}} \bigcup_{n\geq 0} F^n(\mathcal{D})$. En particulier on a $(\mathcal{D}_{\mathrm{C}} \cup \mathcal{D}_{\equiv} \cup \mathcal{D}_R) \subseteq \mathcal{D}_\infty$. Plutôt que de prouver directement le théorème 6.1, nous prouvons que $\mathcal{D}_\infty$ est une def-simulation. Plus précisément, nous prouvons le lemme 6.2 qui est plus général.

Définition 6.4 (Simulation ground modulo *F*) Soit $\mathcal{D}_\infty$ la relation définie par $\mathcal{D}_\infty =_{\mathrm{def}} \bigcup_{n\geq 0} F^n(\mathcal{D})$. La relation $\mathcal{D}$ est une simulation ground modulo F, si pour tout couple $(P,Q) \in \mathcal{D}$ on a:

(i) si $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \stackrel{\mu}{\longmapsto} Q'$ et $P' \,\mathcal{D}_\infty\, Q'$;

(ii) si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;P')$, alors il existe un processus Q' et un environnement D' tels que: $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D';Q')$ avec $P' \,\mathcal{D}_\infty\, Q'$ et $(\mathbf{def}\ D\ \mathbf{in}\ \tilde{a})\,\mathcal{D}_\infty\,(\mathbf{def}\ D'\ \mathbf{in}\ \tilde{b})$.

Il est clair que les trois notions de simulation «up-to» définies en début de ce chapitre sont des cas particuliers de simulation modulo F. Aussi, pour prouver le théorème 6.1, il est suffisant de prouver la propriété suivante.

Lemme 6.2 (Preuve modulo *F*) *Si $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont des simulations ground modulo F, et si $\mathcal{D}$ est close par substitutions des variables par des noms, alors $\mathcal{D}_\infty$ est une def-bisimulation, c'est-à-dire que $\mathcal{D}_\infty \subseteq \sim_d$.*

Preuve Si $\mathcal{D}$ est close par substitution des variables par des noms, alors, par construction, c'est aussi vrai pour $F^n(\mathcal{D})$, et donc pour $\mathcal{D}_\infty$. Par conséquent, il reste seulement à prouver que $\mathcal{D}_\infty$ est une simulation ground. Pour ce faire, on montre la propriété ($\sharp$) suivante: pour tout entier k, la relation $F^k(\mathcal{D})$ est une simulation modulo F. Plus précisément, on montre que si $(P,Q) \in F^k(\mathcal{D})$, alors :

(i) si $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \stackrel{\mu}{\longmapsto} Q'$ et $P' \,\mathcal{D}_\infty\, Q'$;

(ii) si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;P')$, alors il existe un processus Q' et un environnement D' tels que: $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D';Q')$ avec $P' \,\mathcal{D}_\infty\, Q'$ et $(\mathbf{def}\ D\ \mathbf{in}\ \tilde{a})\,\mathcal{D}_\infty\,(\mathbf{def}\ D'\ \mathbf{in}\ \tilde{b})$.

La preuve de ce résultat est par induction sur l'entier k. Le cas $k = 0$ est très simple, puisque $F^0(\mathcal{D})$ est la relation identité; le cas général est prouvé dans le chapitre 7.

Munis d'une preuve de ($\sharp$), il est facile de voir que $\mathcal{D}_\infty = \bigcup_{n\geq 0} F^n(\mathcal{D})$ vérifie la définition de ground simulation. Par conséquent $\mathcal{D}_\infty$ est une def-bisimulation, ce qui implique aussi que $\mathcal{D}$ est une def-bisimulation, en effet:

$$\mathcal{D} \subseteq F(\mathcal{D}) \subseteq \bigcup_{n\geq 0} F^n(\mathcal{D}) = \mathcal{D}_\infty \subseteq \sim_d$$

□

On remarque que, dans la définition de $F(X)$, on utilise la relation $\sim_d$: on a $F(X) = X \cup \sim_d \cup \ldots$ Par conséquent, un corollaire du lemme 6.2 est que les bisimulations modulo «bisimulation forte» sont aussi des def-bisimulation. Le lemme 6.2 permet aussi de prouver la validité des trois techniques de preuve énumérées dans le théorème 6.1. Par exemple, supposons que $\mathcal{D}$ soit close par substitutions, et que $\mathcal{D}$ et $\mathcal{D}^{-1}$ soient des ground simulations modulo contextes. Par conséquent, ce sont aussi des simulations modulo F, puisque $\mathcal{D}_{\mathrm{C}} \subseteq \mathcal{D}_\infty$. En utilisant le lemme 6.2, on prouve alors que $\mathcal{D}$ est une def-bisimulation.

Dans le chapitre suivant, nous nous prouvons la validité des techniques de preuves up-to. Nous nous servirons ensuite de ces méthodes de preuve pour prouver plusieurs propriétés de la def-bisimulation.

CHAPITRE 7

Preuve de la validité des techniques up-to

Dans ce chapitre, nous prouvons le résultat principal de la partie II, c'est-à-dire que nous prouvons le lemme 6.2. Nous rappelons que ce lemme implique le théorème 6.1. Soit $F(X)$ la fonction entre relations que nous avons brièvement décrite dans la section 6: $F(X) =_{\text{def}} G(X) \cup D(X) \cup \mathcal{E}$, où $G(X)$ est la fonction qui construit la fermeture de X pour tous les opérateurs de $\boldsymbol{\pi}^\star$, $D(X)$ est la fonction qui construit la fermeture de X pour les «lois de réplication», et $\mathcal{E}$ est la relation qui nous permet d'obtenir «les lois de l'équivalence structurelle».

$$
\begin{aligned}
G(X) \;=\;& X \cup \sim_d \cup \{(P,Q) \,|\exists R.(P,R) \in X \text{ et } (R,Q) \in X\} \\
& \cup \{((\boldsymbol{\nu} u)P,(\boldsymbol{\nu} u)Q) \,|(P,Q) \in X\} \\
& \cup\ \{((P\ a),(Q\ a)) \,|(P,Q) \in X\} \\
& \cup\ \{(\langle u \Leftarrow P\rangle,\langle u \Leftarrow Q\rangle) \,|(P,Q) \in X\} \\
& \cup\ \{((\boldsymbol{\lambda} x)P,(\boldsymbol{\lambda} x)Q) \,|(P,Q) \in X\} \\
& \cup\ \{([P_1\,, l = P_2\,],[\,Q_1\,, l = Q_2\,]) \,|(P_1,Q_1) \text{ et } (P_2,Q_2) \in X\} \\
& \cup\ \{(P_1 \mid P_2,Q_1 \mid Q_2) \,|(P_1,Q_1) \text{ et } (P_2,Q_2) \in X\} \\
& \cup\ \{(\mathbf{def}\, p = P_1 \,\mathbf{in}\, P_2,\mathbf{def}\, p = Q_1 \,\mathbf{in}\, Q_2) \,|(P_1,Q_1) \text{ et } (P_2,Q_2) \in X\}
\end{aligned}
$$

$$
\begin{aligned}
D(X) \;=\;& \{(\mathbf{def}\, D \,\mathbf{in}\, (P\ a),\mathbf{def}\, D' \,\mathbf{in}\, (P\ b))| \ (\mathbf{def}\, D \,\mathbf{in}\, u,\mathbf{def}\, D' \,\mathbf{in}\, v) \in X \text{ et} \\
& \qquad\qquad \mathbf{fn}(P) \cap (\mathbf{decl}(D) \cup \mathbf{decl}(D')) = \emptyset\ \} \\
& \cup \{(\mathbf{def}\, p = R \,\mathbf{in}\, (P \mid Q),\mathbf{def}\, p = R \,\mathbf{in}\, ((\mathbf{def}\, p = R \,\mathbf{in}\, P) \mid Q))\} \\
& \cup \{(\mathbf{def}\, p = R \,\mathbf{in}\, (P\ p),\mathbf{def}\, p = R \,\mathbf{in}\, ((\mathbf{def}\, p = R \,\mathbf{in}\, P)\ p))\} \\
& \cup\ \{(\mathbf{def}\, p = R \,\mathbf{in}\, ((\boldsymbol{\lambda} x)P),(\boldsymbol{\lambda} x)(\mathbf{def}\, p = R \,\mathbf{in}\, P)) \,|x \notin \mathbf{fn}(R)\} \\
& \cup \{(\mathbf{def}\, p = R \,\mathbf{in}\, (\langle u \Leftarrow P\rangle),\langle u \Leftarrow (\mathbf{def}\, p = R \,\mathbf{in}\, P)\rangle)\} \\
& \cup\ \{(\mathbf{def}\, p = R,q = S \,\mathbf{in}\, P,\mathbf{def}\, p = R,q = S \,\mathbf{in}\, (\mathbf{def}\, p = R \,\mathbf{in}\, P)) \,|q \notin \mathbf{fn}(R)\} \\
& \cup\ \{(\mathbf{def}\, p = R,q = S \,\mathbf{in}\, P,\mathbf{def}\, p = R,q = (\mathbf{def}\, p = R \,\mathbf{in}\, S) \,\mathbf{in}\, P) \,|q \notin \mathbf{fn}(R)\}
\end{aligned}
$$

$$
\begin{array}{rcl}
\mathcal{E} & = & \{(P \mid \mathbf{0},P)\} \cup \{(P \mid Q,Q \mid P)\} \cup \{(P \mid (Q \mid R),(P \mid Q) \mid R)\} \\
& & \cup\ \{(\mathbf{def}\, p = R\, \mathbf{in}\, P,P) \,|p \notin \mathbf{fn}(P)\,\} \ \cup\ \{((\boldsymbol{\nu} u)P,P) \,|u \notin \mathbf{fn}(P)\,\} \\
& & \cup\ \{(\mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu} u)P),(\boldsymbol{\nu} u)(\mathbf{def}\, p = R\, \mathbf{in}\, P))\} \\
& & \cup\ \{(\mathbf{def}\, p = R,q = S\, \mathbf{in}\, P,\mathbf{def}\, q = S,p = R\, \mathbf{in}\, P) \,|q \neq p,p \notin \mathbf{fn}(S),q \notin \mathbf{fn}(R)\,\} \\
& & \cup\ \{((\boldsymbol{\nu} u)(\boldsymbol{\nu} v)P,(\boldsymbol{\nu} v)(\boldsymbol{\nu} u)P) \,|u \neq v\,\} \\
& & \cup\ \{((\boldsymbol{\nu} u)P \mid Q,(\boldsymbol{\nu} u)(P \mid Q)) \,|u \notin \mathbf{fn}(Q)\,\} \\
& & \cup\ \{((\mathbf{def}\, p = R\, \mathbf{in}\, P) \mid Q,\mathbf{def}\, p = R\, \mathbf{in}\, (P \mid Q)) \,|p \notin \mathbf{fn}(Q)\,\} \\
& & \cup\ \{((\boldsymbol{\nu} u)P\ a,(\boldsymbol{\nu} u)(P\ a)) \,|a \neq u\,\} \\
& & \cup\ \{((\mathbf{def}\, p = R\, \mathbf{in}\, P)\ a,\mathbf{def}\, p = R\, \mathbf{in}\, (P\ a)) \,|a \neq p\,\} \\
& & \cup\ \{((P \mid Q)\ a,(P\ a) \mid (Q\ a))\} \\
& & \cup\ \{(\langle u \Leftarrow P\rangle\ a,\langle u \Leftarrow P\rangle)\} \cup \{(\mathbf{0}\ a,\mathbf{0})\}
\end{array}
$$

La fonction F est monotone et pour tout $n \geq 1$, nous avons $X \subseteq F(X) \subseteq F^n(X)$. Par conséquent, pour toute relation $\mathcal{D}$, nous pouvons définir la relation $\mathcal{D}_\infty =_{\text{def}} \bigcup_{n\geq 0} F^n(\mathcal{D})$. Il est simple de voir que si $\mathcal{D}$ est close par substitution des variables par les noms, alors la même propriété est vraie pour $F^n(\mathcal{D})$, et donc pour $\mathcal{D}_\infty$. De plus, il est simple de voir que, pour chaque relation $\mathcal{D}$, la relation $\bigcup_{n\geq 0} G^n(\mathcal{D})$ est une congruence, et que la relation $\bigcup_{n\geq 0} G^n(\mathcal{E})$ contient $\equiv$. En ce qui concerne la fonction $D(X)$, il est facile de voir que $\mathcal{D}_R \subseteq \mathcal{D}_\infty$. De plus, si $(\mathbf{def}\, D\, \mathbf{in}\, u)\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, v)$ et $(\mathbf{decl}(D) \cup \mathbf{decl}(D')) \cap \mathbf{fn}(P) = \emptyset$, alors $(\mathbf{def}\, D\, \mathbf{in}\, P\{u/x\})\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, P\{v/x\})$.

Dans ce chapitre, nous prouvons que si $\mathcal{D}$ est une bisimulation up-to F (cf. lemme 6.2), alors $\mathcal{D}_\infty$ est une def-simulation. Nous avons déjà noté que $\mathcal{D}_\infty$ était close par substitution, par conséquent il est suffisant de prouver que $\mathcal{D}_\infty$ est une ground simulation. Avant d'établir ce résultat, nous prouvons deux propriétés intermédiaires.

Lemme 7.1 *Soit $\mathcal{D}$ une bisimulation up-to F, et soit $\mathcal{D}_\infty$ la relation $\bigcup_{n\geq 0} F^n(\mathcal{D})$ définie ci-dessus. Soit $\tilde{a},\tilde{c}$ deux tuples, et D,D' deux définitions telles que: $(\mathbf{def}\, D\, \mathbf{in}\, \tilde{a})\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, \tilde{c})$, alors:*

(i) si $P\, \mathcal{D}_\infty\, Q$ et $(\mathbf{fn}(P) \cap \mathbf{decl}(D)) = (\mathbf{fn}(Q) \cap \mathbf{decl}(D')) = \emptyset$, alors $(\mathbf{def}\, D\, \mathbf{in}\, (P\ \tilde{a}))\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, (Q\ \tilde{c}))$;

(ii) si $P \overset{\mathbf{in}\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'$ et $\mathbf{fn}(P) \cap (\mathbf{decl}(D) \cup \mathbf{decl}(D')) = \emptyset$, alors il existe un processus P'' tel que: $P \overset{\mathbf{in}\, u.(\tilde{b};\tilde{c})}{\longmapsto} P''$ et $(\mathbf{def}\, D\, \mathbf{in}\, P')\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, P'')$. Et si $P \overset{\mathbf{in}\, u.(\tilde{a};\tilde{b})}{\longmapsto} P'$ et $\mathbf{fn}(P) \cap (\mathbf{decl}(D) \cup \mathbf{decl}(D')) = \emptyset$, alors il existe un processus P'' tel que: $P \overset{\mathbf{in}\, u.(\tilde{c};\tilde{b})}{\longmapsto} P''$ et $(\mathbf{def}\, D\, \mathbf{in}\, P')\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, P'')$;

Soit a,c deux noms tels que $(\mathbf{def}\, D\, \mathbf{in}\, a)\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, c)$.

(iii) si $P \overset{\lambda\, a}{\longmapsto} P'$ et $\mathbf{fn}(P) \cap (\mathbf{decl}(D) \cup \mathbf{decl}(D')) = \emptyset$, alors il existe un processus P'' tel que: $P \overset{\lambda\, c}{\longmapsto} P''$ avec $(\mathbf{def}\, D\, \mathbf{in}\, P')\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, P'')$.

Preuve Soit $\tilde{a},\tilde{c}$ deux tuples de la même taille, et P,Q deux processus tels que $P\, \mathcal{D}_\infty\, Q$. La première propriété est prouvée par induction sur la taille de $\tilde{a}$.

- **cas $|\tilde{a}| = 0$:** comme $\{(\mathbf{def}\, p = R\, \mathbf{in}\, P,P) \,|p \notin \mathbf{fn}(P)\,\}$ est un sous-ensemble de $\mathcal{D}_\infty$, et comme $(\mathbf{fn}(P) \cap \mathbf{decl}(D)) = (\mathbf{fn}(Q) \cap \mathbf{decl}(D')) = \emptyset$, nous avons:

$$(\mathbf{def}\, D\, \mathbf{in}\, P)\, \mathcal{D}_\infty\, P\, \mathcal{D}_\infty\, Q\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, Q)$$

- **cas $|\tilde{a}| = n + 1$:** supposons que $(\mathbf{def}\, D\, \mathbf{in}\, (P\ \tilde{a}))\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, (Q\ \tilde{c}))$. Le couple $((\mathbf{def}\, D\, \mathbf{in}\, (Pa)),(\mathbf{def}\, D'\, \mathbf{in}\, (Pc)))$ est dans $\mathcal{D}_\infty$, par conséquent, en utilisant la transitivité de la relation $\mathcal{D}_\infty$, et le fait que $\mathcal{D}_\infty$ est une congruence, nous obtenons que:

$$
\begin{array}{rcl}
\mathbf{def}\, D\, \mathbf{in}\, (P\ \tilde{a}\ a_{n+1}) & = & \mathbf{def}\, D\, \mathbf{in}\, ((\mathbf{def}\, D\, \mathbf{in}\, P\ \tilde{a})\ a_{n+1}) \\
& \mathcal{D}_\infty & \mathbf{def}\, D'\, \mathbf{in}\, ((\mathbf{def}\, D\, \mathbf{in}\, P\ \tilde{a})c_{n+1}) \\
& \mathcal{D}_\infty & \mathbf{def}\, D'\, \mathbf{in}\, ((\mathbf{def}\, D'\, \mathbf{in}\, Q\ \tilde{c})\ c_{n+1}) \\
& \mathcal{D}_\infty & \mathbf{def}\, D'\, \mathbf{in}\, (Q\ \tilde{c}\ c_{n+1})
\end{array}
$$

La seconde propriété est prouvée par induction sur l'inférence de $P \stackrel{\mathbf{in}\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'$. Soit μ_a et μ_c les deux actions: $\mu_a = \mathbf{in}\, u.(\tilde{b};\tilde{a})$ et $\mu_c = \mathbf{in}\, u.(\tilde{b};\tilde{c})$, alors:

- **cas (decl):** nous avons $P = \langle u \Leftarrow P_1 \rangle \stackrel{\mu_a}{\longmapsto} P' = (P_1\ \tilde{a})$. Par conséquent: $P \stackrel{\mu_c}{\longmapsto} (P_1\ \tilde{c})$ et comme $P_1\, \mathcal{D}_\infty\, P_1$, le résultat attendu, c'est-à-dire $(\mathbf{def}\, D\, \mathbf{in}\, (P_1\ \tilde{a}))\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, (P_1\ \tilde{c}))$, suit du lemme 7.1-$(i)$;
- **cas (new mu):** nous avons $P = (\boldsymbol{\nu} u)P_1$ et $P_1 \stackrel{\mu_a}{\longmapsto} P_1'$. Par conséquent $P \stackrel{\mu_a}{\longmapsto} (\boldsymbol{\nu} u)P_1'$, et en utilisant l'hypothèse d'induction, il suit qu'il existe un processus P_1'' tel que $P_1 \stackrel{\mu_c}{\longmapsto} P_1''$, avec $(\mathbf{def}\, D\, \mathbf{in}\, P_1')\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, P_1'')$. Le résultat attendu suit du fait que $\mathcal{D}_\infty$ est une congruence. La preuve est similaire dans les cas (def mu) et (par in);
- **cas (app in):** nous avons: $P = (P_1\ b)$ et $P_1 \stackrel{\mathbf{in}\, u.((b,\tilde{b});\tilde{a})}{\longmapsto} P_1'$. Par conséquent $(P\ b) \stackrel{\mu_a}{\longmapsto} P_1'$, et en utilisant l'hypothèse d'induction, il suit qu'il existe un processus P_1'' tel que $P_1 \stackrel{\mathbf{in}\, u.((b,\tilde{b});\tilde{c})}{\longmapsto} P_1''$ et $(\mathbf{def}\, D\, \mathbf{in}\, P_1')\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, P_1'')$. Le résultat suit du fait que $(P\ b) \stackrel{\mu_c}{\longmapsto} P_1''$.

Nous prouvons la troisième propriété par induction sur l'inférence de $P \stackrel{\lambda\, u}{\longmapsto} P'$. Cette propriété est triviale si a est une étiquette, ou si $a = b$. Par conséquent, dans la suite de la preuve, nous considérons que a est un nom lié dans D:

- **cas (lambda):** nous avons $P = (\boldsymbol{\lambda} x)P_1$. Le résultat suit du fait que

$$(\mathbf{def}\, D\, \mathbf{in}\, P\{^a/_x\})\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, P\{^c/_x\})$$

  Plus précisément, nous obtenons ce résultat en utilisant les "lois de $D(X)$" pour distribuer les définitions sous chaque constructeur de P. Ensuite, nous utilisons le fait que, pour tout Q tel que $(\mathbf{fn}(Q) \cap (\mathbf{decl}(D) \cup \mathbf{decl}(D'))) = \emptyset$, la relation $(\mathbf{def}\, D\, \mathbf{in}\, (Q\ a))\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, (P\ c))$ est valide.
- **cas (par lambda):** Le résultat suit de la propriété (i) et de la règle de distribution des définitions sur la composition parallèle;
- **cas (new mu) et (def mu):** le résultat suit du fait que $\mathcal{D}_\infty$ est une congruence.

□

Lemme 7.2 (Actions in et application) *Si* $P \stackrel{\mathbf{in}\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'$, *alors il existe une transition* **in** *telle que* $P \stackrel{\mathbf{in}\, u.((\tilde{b},a);(\tilde{a},a))}{\longmapsto} P''$ *avec* $P'' \equiv (P'\ a)$.

En utilisant la règle (app in), nous obtenons comme corollaire que si $P \stackrel{\mathbf{in}\, u.(\epsilon;\tilde{a})}{\longmapsto} P'$, alors $(P\ a) \stackrel{\mathbf{in}\, u.(\epsilon;(\tilde{a},a))}{\longmapsto} P''$ avec $P'' \equiv (P'\ a)$.

Preuve La preuve est faite par induction sur l'inférence de $P \stackrel{\mathbf{in}\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'$. Pour simplifier la présentation de la preuve, nous utiliserons le symbole μ pour désigner l'action $\mathbf{in}\, u.(\tilde{b};\tilde{a})$, et μ_a pour désigner l'action $\mathbf{in}\, u.((\tilde{b},a);(\tilde{a},a))$. La preuve se réduit à une analyse par cas sur la dernière règle de l'inférence de $P \stackrel{\mathbf{in}\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'$.

- **cas (decl):** nous avons $\langle u \Leftarrow P \rangle \stackrel{\mu}{\longmapsto} (P\ \tilde{a})$ pour tout tuples $\tilde{b}$ et $\tilde{a}$. Par Conséquent $\langle u \Leftarrow P \rangle \stackrel{\mu_a}{\longmapsto} (P\ \tilde{a}\ a)$;
- **cas (new mu):** nous avons $P = (\boldsymbol{\nu} u)P_1$ et $P_1 \stackrel{\mu}{\longmapsto} P_1'$. Par conséquent $P \stackrel{\mu}{\longmapsto} P' = (\boldsymbol{\nu} u)P_1'$, et comme $a \neq u$, nous pouvons utiliser l'hypothèse d'induction et la règle (new mu) pour prouver que $(\boldsymbol{\nu} u)P \stackrel{\mu_a}{\longmapsto} (\boldsymbol{\nu} u)P_1''$ avec $P_1'' \equiv (P_1'\ a)$. Le résultat suit du fait que $(\boldsymbol{\nu} u)P_1'' \equiv (\boldsymbol{\nu} u)(P_1'\ a) \equiv (\boldsymbol{\nu} u)P_1'\ a$. La preuve dans le cas (def mu) est similaire;
- **cas (par in):** nous avons $P = (P_1 \mid Q_1)$ et $P_1 \stackrel{\mu}{\longmapsto} P_1'$. Par conséquent $P \stackrel{\mu}{\longmapsto} P' = (P_1' \mid (Q_1\ \tilde{b}))$, et en utilisant l'hypothèse d'induction et la règle (par in), il suit que $P \stackrel{\mu_a}{\longmapsto} (P_1'' \mid (Q_1\ \tilde{b}\ a))$ avec $P_1'' \equiv (P_1'\ a)$. Le résultat suit du fait que $(P_1'' \mid (Q_1\ \tilde{b}\ a)) \equiv ((P_1'\ a) \mid (Q_1\ \tilde{b}\ a)) \equiv (P_1' \mid (Q_1\ \tilde{b}))\ a$;
- **cas (app in):** soit ξ l'action $\mathbf{in}\, u.((c,\tilde{b});\tilde{a})$. Nous avons $P = (P_1\ c)$ et $P_1 \stackrel{\xi}{\longmapsto} P_1'$. Par conséquent $P \stackrel{\xi}{\longmapsto} P' = P_1'$, et en utilisant l'hypothèse d'induction et la règle (app in), il suit que $P \stackrel{\mu_a}{\longmapsto} P_1''$ avec $P_1'' \equiv (P_1'\ a)$, ce qui est le résultat attendu.

□

Nous prouvons maintenant le résultat principal de ce chapitre, c'est-à-dire que si $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont deux bisimulations up-to F, alors $\mathcal{D}_\infty$ est une bisimulation ground. Plus précisément, nous prouvons simultanément deux propriétés: si $(P,Q) \in F^k(\mathcal{D})$, alors

- si $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \stackrel{\mu}{\longmapsto} Q'$, et $P' \,\mathcal{D}_\infty\, Q'$;
- si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;P')$, alors il existe un processus Q' et un environnement D' tel que: $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{c})}{\longmapsto} (D';Q')$, et $P' \,\mathcal{D}_\infty\, Q'$, et $|\tilde{a}| = |\tilde{c}|$ (disons n), et pour tout indice i dans l'intervalle $[1..n]$ nous avons $(\mathbf{def}\ D\ \mathbf{in}\ a_i) \,\mathcal{D}_\infty\, (\mathbf{def}\ D'\ \mathbf{in}\ c_i)$.

La preuve est faite par induction sur k et sur la définition de $F(X)$.

Comme $F^0(X) = \mathcal{I}d$, le cas $k = 0$ est trivial. En effet l'identité est une bisimulation. Pour le cas $k = (n+1)$, la preuve est plus technique. Nous divisons celle-ci en suivant les trois «principaux sous-ensembles» de $F(X)$. Il faut noter que, en utilisant les propriétés de $\mathcal{E}$ et $G(X)$ données en introduction de ce chapitre, il est facile de voir que si $P \equiv Q$, alors $P \,\mathcal{D}_\infty\, Q$, et que pour tout contexte $\mathbf{C}$, si $P \,\mathcal{D}_\infty\, Q$, alors $\mathbf{C}[P] \,\mathcal{D}_\infty\, \mathbf{C}[Q]$.

7.1 Règles pour la congruence

cas $P \,\mathcal{D}\, Q$ ou $P \sim_d Q$: par définition, les relations $\mathcal{D}$ et $\sim_d$ sont incluses dans $\mathcal{D}_\infty$. Par conséquent le résultat suit de la définition de bisimulation up-to F.

cas $(P,Q) \in F^n(\mathcal{D})$ et $(Q,R) \in F^n(\mathcal{D})$: supposons que $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$. Nous utilisons l'hypothèse d'induction pour prouver que $R \stackrel{\mu}{\longmapsto} R'$ et $P' \,\mathcal{D}_\infty\, R'$, et l'hypothèse d'induction encore une fois, pour prouver que $Q \stackrel{\mu}{\longmapsto} Q'$ et $R' \,\mathcal{D}_\infty\, Q'$. Le résultat suit de la transitivité de $\mathcal{D}_\infty$. La preuve est similaire dans le cas des actions **out**.

cas $P = (\boldsymbol{\nu} u)P_1$ et $Q = (\boldsymbol{\nu} u)Q_1$:

- **cas $\kappa(\mu) \neq \mathbf{out}$:** la dernière règle de l'inférence $P \stackrel{\mu}{\longmapsto} P'$ est (new mu), et $P' = (\boldsymbol{\nu} u)P_1'$ avec $P_1 \stackrel{\mu}{\longmapsto} P_1'$. En utilisant l'hypothèse d'induction, il suit que $Q_1 \stackrel{\mu}{\longmapsto} Q_1'$ avec $(P_1',Q_1') \in \mathcal{D}_\infty$, et en utilisant la règle (new mu), il existe une transition $Q \stackrel{\mu}{\longmapsto} (\boldsymbol{\nu} u)Q_1'$. Le résultat suit du fait que $(P_1',Q_1') \in \mathcal{D}_\infty$ implique que $(\boldsymbol{\nu} u)P_1' \,\mathcal{D}_\infty\, (\boldsymbol{\nu} u)Q_1'$;
- **cas $\kappa(\mu) = \mathbf{out}$:** si la dernière règle est (new out), alors il existe un environnement D et un processus P_1' tels que: $P' = (D;(\boldsymbol{\nu} u)P_1')$ et $P_1 \stackrel{\mu}{\longmapsto} (D;P_1')$. En utilisant l'hypothèse d'induction, il suit que $Q_1 \stackrel{\mu}{\longmapsto} (D';Q_1')$ avec $(P_1',Q_1') \in \mathcal{D}_\infty$. Comme dans le cas précédent, nous concluant en utilisant la règle (new out) et le fait que $(P_1',Q_1') \in \mathcal{D}_\infty$ implique que $((\boldsymbol{\nu} u)P_1',(\boldsymbol{\nu} u)Q_1') \in \mathcal{D}_\infty$;

  Si la dernière règle est (new open), alors $P_1 \stackrel{\mathbf{out}\ v.(\tilde{v};\tilde{a})}{\longmapsto} (D;P_1')$ et $u \neq v$. En utilisant l'hypothèse d'induction, il suit que $Q_1 \stackrel{\mathbf{out}\ v.(\tilde{v};\tilde{c})}{\longmapsto} (D';Q_1')$ avec: $(P_1' \,\mathcal{D}_\infty\, Q_1')$ et $(\mathbf{def}\ D\ \mathbf{in}\ \tilde{a}) \,\mathcal{D}_\infty\, (\mathbf{def}\ D'\ \mathbf{in}\ \tilde{c})$, ce qui est le attendu résultat.

cas $P = (P_1\ a)$ et $Q = (Q_1\ a)$: nous faisons une analyse par cas sur la dernière règle de l'inférence de $P \stackrel{\mu}{\longmapsto} P'$. Supposons que $P_1 \stackrel{\mu_1}{\longmapsto} P_1'$ (avec $\kappa(\mu_1) \neq \mathbf{out}$):

- **cas (app tau):** nous avons $\mu = \mu_1 = \tau$. Nous utilisons l'hypothèse d'induction pour prouver que $Q_1 \stackrel{\tau}{\longmapsto} Q_1'$ avec $P_1' \,\mathcal{D}_\infty\, Q_1'$. Le résultat suit en appliquant la règle (app tau);
- **cas (app com):** nous avons $\mu = \tau$ et $\mu_1 = \lambda\, a$. En utilisant l'hypothèse d'induction, on obtient qu'il existe une transition $Q_1 \stackrel{\mu_1}{\longmapsto} Q_1'$ avec $P_1' \,\mathcal{D}_\infty\, Q_1'$. Le résultat suit de la règle (app com);
- **cas (app in):** nous avons $\mu_1 = \mathbf{in}\, u.((a,\tilde{b});\tilde{a})$ et $\mu = \mathbf{in}\, u.(\tilde{b};\tilde{a})$. En utilisant l'hypothèse d'induction, on obtient qu'il existe une transition $Q_1 \stackrel{\mu_1}{\longmapsto} Q_1'$ avec $P_1' \,\mathcal{D}_\infty\, Q_1'$. Le résultat suit de la règle (app in);

– **cas (app out):** nous avons $P_1 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_1')$ et $P \stackrel{\mathbf{out}\ u.(\tilde{v};(\tilde{a},c))}{\longmapsto} (D; (P_1'\ c))$. En utilisant l'hypothèse d'induction, il suit que $Q_1 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{c})}{\longmapsto} (D'; Q_1')$ avec $P_1' \,\mathcal{D}_\infty\, Q_1'$ et $(\mathbf{def}\ D\ \mathbf{in}\ \tilde{a})\,\mathcal{D}_\infty\,(\mathbf{def}\ D'\ \mathbf{in}\ \tilde{c})$. Le résultat suit de l'application de la règle (app out), et du fait que $a \notin (\mathbf{decl}(D) \cup \mathbf{decl}(D'))$ implique que $(\mathbf{def}\ D\ \mathbf{in}\ a)\,\mathcal{D}_\infty\, a\,\mathcal{D}_\infty\,(\mathbf{def}\ D'\ \mathbf{in}\ a)$.

cas $P = (\boldsymbol{\lambda} x)P_1$ **et** $Q = (\boldsymbol{\lambda} x)Q_1$**:** la dernière règle est nécessairement (lambda). Par conséquent, le résultat suit du fait que $\mathcal{D}_\infty$ est close par instantiation. La preuve est similaire dans le cas $P = [\,P_1\,, l = P_2\,]$ et $Q = [\,Q_1\,, l = Q_2\,]$ et dans le cas $P = \langle u \Leftarrow P_1\rangle$ et $Q = \langle u \Leftarrow Q_1\rangle$.

cas $P = (P_1 \mid P_2)$ **et** $Q = (Q_1 \mid Q_2)$**:** Si la dernière règle utilisée dans l'inférence de la transition $P \stackrel{\mu}{\longmapsto} P'$ est (par tau), (par lambda), (par in) ou (par out), le résultat suit de l'hypothèse d'induction et du fait que, quelque soit le tuple de noms $\tilde{b}$ on a: $(P,Q) \in \mathcal{D}_\infty$ implique $(P\ \tilde{b}, Q\ \tilde{b}) \in \mathcal{D}_\infty$. Le seul cas intéressant est celui tel que la dernière règle utilisée est (par com). Supposons que $P_1 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_1')$ et $P_2 \stackrel{\mathbf{in}\ u.(\epsilon;\tilde{a})}{\longmapsto} P_2'$ et $P' = (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (P_1' \mid P_2'))$. Alors, en utilisant l'hypothèse d'induction, il suit que $Q_1 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{c})}{\longmapsto} (D'; Q_1')$ avec $(\dagger) : P_1' \,\mathcal{D}_\infty\, Q_1'$ et $(\mathbf{def}\ D\ \mathbf{in}\ \tilde{a})\,\mathcal{D}_\infty\,(\mathbf{def}\ D'\ \mathbf{in}\ \tilde{c})$. En utilisant le lemme 7.1, cette dernière propriété implique qu'il existe un processus P_2'' tel que: $P_2 \stackrel{\mathbf{in}\ u.(\epsilon;\tilde{c})}{\longmapsto} P_2''$ avec $(\mathbf{def}\ D\ \mathbf{in}\ P_2')\,\mathcal{D}_\infty\,(\mathbf{def}\ D'\ \mathbf{in}\ P_2'')$. Finalement, nous utilisons l'hypothèse d'induction sur le couple (P_2,Q_2) pour prouver que: $Q_2 \stackrel{\mathbf{in}\ u.(\epsilon;\tilde{c})}{\longmapsto} Q_2'$ avec $Q_2' \,\mathcal{D}_\infty\, P_2''$. Ceci implique que:

$$(\ddagger) : (\mathbf{def}\ D'\ \mathbf{in}\ Q_2')\,\mathcal{D}_\infty\,(\mathbf{def}\ D'\ \mathbf{in}\ P_2'')\,\mathcal{D}_\infty\,(\mathbf{def}\ D\ \mathbf{in}\ P_2')$$

Par conséquent il existe une transition τ à partir de Q telle que: $Q \stackrel{\tau}{\longmapsto} Q' = (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D'\ \mathbf{in}\ (Q_1' \mid Q_2'))$, et comme les variables dans $\mathbf{decl}(D)$ (resp. $\mathbf{decl}(D')$) ne sont pas libres dans P_1' (resp. Q_1'), nous avons que:

$$\begin{array}{lll} P' = (\mathbf{def}\ D\ \mathbf{in}\ (P_1' \mid P_2')) & \mathcal{D}_\infty & (P_1' \mid \mathbf{def}\ D\ \mathbf{in}\ P_2') \\ & \mathcal{D}_\infty & (Q_1' \mid \mathbf{def}\ D'\ \mathbf{in}\ Q_2')\,\mathcal{D}_\infty\,(\mathbf{def}\ D'\ \mathbf{in}\ (Q_1' \mid Q_2')) = Q' \end{array}$$

Dans ces relations, nous avons aussi utilisé les propriétés (†) et (‡), et le fait que $\mathcal{D}_\infty$ est une congruence.

cas $P = (\mathbf{def}\ p = P_1\ \mathbf{in}\ P_2)$ **et** $Q = (\mathbf{def}\ p = Q_1\ \mathbf{in}\ Q_2)$**:** les cas intéressants correspondent à l'utilisation des règles (def open) et (def com).

– **cas (def open):** supposons que $P_2 \stackrel{\mathbf{out}\ p.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_2')$ et que: $P \stackrel{\mathbf{out}\ p.(\tilde{v};\tilde{a})}{\longmapsto} ((p = P_1, D) \mid P_2')$. Par conséquent, en utilisant l'hypothèse d'induction, il existe une transition $Q_2 \stackrel{\mathbf{out}\ p.(\tilde{v};\tilde{c})}{\longmapsto} (D'; Q_2')$ telle que:

$$(i) : (\mathbf{def}\ D\ \mathbf{in}\ \tilde{a})\,\mathcal{D}_\infty\,(\mathbf{def}\ D'\ \mathbf{in}\ \tilde{c}) \qquad (ii) : P_2' \,\mathcal{D}_\infty\, Q_2' \qquad (iii) : P_1 \,\mathcal{D}_\infty\, Q_1$$

Le résultat suit alors du fait que (i) et (iii) impliquent que:

$$(\mathbf{def}\ p = P_1, D\ \mathbf{in}\ \tilde{a})\,\mathcal{D}_\infty\,(\mathbf{def}\ p = Q_1, D'\ \mathbf{in}\ \tilde{c})$$

et que (ii) et (iii) impliquent que: $(\mathbf{def}\ p = P_1\ \mathbf{in}\ P_2')\,\mathcal{D}_\infty\,(\mathbf{def}\ p = Q_1\ \mathbf{in}\ Q_2')$;

– **cas (def com):** supposons que $P_2 \stackrel{\mathbf{out}\ p.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_2')$, et que:

$$P \stackrel{\tau}{\longmapsto} \mathbf{def}\ p = P_1\ \mathbf{in}\ (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (P_1\ \tilde{a} \mid P_2'))$$

Par conséquent, en utilisant l'hypothèse d'induction, il existe une transition à partir de Q_2 telle que $Q_2 \stackrel{\mathbf{out}\ p.(\tilde{v};\tilde{c})}{\longmapsto} (D'; Q_2')$. De plus cette transition vérifie les conditions (i), (ii) et (iii) données dans le cas précédent. Par conséquent nous pouvons dériver une transition à partir de Q, telle que:

$$Q \stackrel{\tau}{\longmapsto} \mathbf{def}\ p = Q_1\ \mathbf{in}\ (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D'\ \mathbf{in}\ (Q_1\ \tilde{c} \mid Q_2'))$$

Comme les variables de $\mathbf{decl}(D)$ (resp. $\mathbf{decl}(D')$) ne sont pas libres dans P_1 et P_2' (resp. Q_1 et Q_2'), nous avons que:

$$\begin{array}{rcl}(\mathbf{def}\, D\, \mathbf{in}\, (P_1\ \tilde{a} \mid P_2')) & \mathcal{D}_\infty & ((\mathbf{def}\, D\, \mathbf{in}\, (P_1\ \tilde{a})) \mid P_2') \\ (\mathbf{def}\, D'\, \mathbf{in}\, (Q_1\ \tilde{c} \mid Q_2')) & \mathcal{D}_\infty & ((\mathbf{def}\, D'\, \mathbf{in}\, (Q_1\ \tilde{a})) \mid Q_2')\end{array}$$

Nous pouvons alors conclure, en utilisant le lemme 7.1-(i), que:
$(\mathbf{def}\, D'\, \mathbf{in}\, (Q_1\ \tilde{c}))\, \mathcal{D}_\infty\, (\mathbf{def}\, D\, \mathbf{in}\, (P_1\ \tilde{a}))$. Le résultat suit du fait que $\mathcal{D}_\infty$ est a congruence.

7.2 Lois de réplication

cas $P = \mathbf{def}\, D\, \mathbf{in}\, (P_1\ a)$ **et** $Q = \mathbf{def}\, D'\, \mathbf{in}\, (P_1\ b)$**:** nous avons comme hypothèses que $(\mathbf{def}\, D\, \mathbf{in}\, u)\, F^n(\, \mathcal{D}\,)\, (\mathbf{def}\, D'\, \mathbf{in}\, v)$, et que $\mathbf{fn}(P_1) \cap (\mathbf{decl}(D) \cup \mathbf{decl}(D')) = \emptyset$. Il est facile de voir que, avec ces conditions, on a $P\, \mathcal{D}_\infty\, Q$ implique $(\mathbf{def}\, D\, \mathbf{in}\, (P\ a))\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, (Q\ b))$. Supposons que $P_1 \stackrel{\mu_1}{\longmapsto} P_1'$, nous faisons une étude par cas sur la règle utilisée pour la donner la transition de $(P_1\ a)$:

- **cas (app tau):** le résultat suit du fait que $(\mathbf{def}\, D\, \mathbf{in}\, (P_1'\ a), \mathbf{def}\, D'\, \mathbf{in}\, (P_1'\ b)) \in \mathcal{D}_\infty$ et que $(\dagger) : \mathbf{fn}(\mathbf{def}\, D\, \mathbf{in}\, (P_1'\ a)) \cap \mathbf{decl}(D) = \mathbf{fn}(\mathbf{def}\, D'\, \mathbf{in}\, (P_1'\ b)) \cap \mathbf{decl}(D') = \emptyset$;
- **cas (app com):** nous avons $\mu_1 = \lambda\, a$ et $(P_1\ a) \stackrel{\tau}{\longmapsto} P_1'$. Par conséquent, en utilisant le lemme 7.1-(iii), nous pouvons conclure qu'il existe un processus P_1'' tel que $(\ddagger) : P_1 \stackrel{\lambda\, b}{\longmapsto} P_1''$ et $((\mathbf{def}\, D\, \mathbf{in}\, P_1')\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, P_1''))$. Le résultat suit du fait que $(\ddagger)$ implique que $(P_1\ b) \stackrel{\tau}{\longmapsto} P_1''$, et de la propriété $(\dagger)$;
- **cas (app in) et (app out):** dans le premier cas, nous avons $\mu_1 = \mathbf{in}\, c.(a,\tilde{b};\tilde{a})$ et $(P_1\ a) \stackrel{\mathbf{in}\, c.(\tilde{b};\tilde{a})}{\longmapsto} P_1'$. Par conséquent, en utilisant le lemme 7.1-(ii), on montre qu'il existe un processus P_1'' tel que $P_1 \stackrel{\mathbf{in}\, c.(b,\tilde{b};\tilde{a})}{\longmapsto} P_1''$ et $(\mathbf{def}\, D\, \mathbf{in}\, P_1')\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, P_1'')$. Le résultat suit de la propriété $(\dagger)$. La preuve est similaire dans le cas de la règle (app out): dans ce case nous utilisons aussi le fait que si $\mathbf{fn}(P_1) \cap \mathbf{decl}(D) = \emptyset$ et $P_1 \stackrel{\mathbf{out}\, c.(\tilde{v};\tilde{a})}{\longmapsto} P_1'$, alors $\tilde{a} \cap \mathbf{decl}(D) = \emptyset$.

cas $P = \mathbf{def}\, p = R\, \mathbf{in}\, (P_1 \mid Q_1)$ **et** $Q = \mathbf{def}\, p = R\, \mathbf{in}\, ((\mathbf{def}\, p = R\, \mathbf{in}\, P_1) \mid Q_1)$**:** nous supposons que $P_1 \stackrel{\mu_1}{\longmapsto} P_1'$ et nous faisons une analyse par cas sur les deux dernières règles de l'inférence de la transition $P \stackrel{\mu}{\longmapsto} P'$. Comme les rôles de P_1 et Q_1 sont symétriques, nous nous intéressons uniquement au cas où la transition «vient de» Q_1.

- **cas (par tau) suivie par (def mu):** nous avons $\mu_1 = \tau$ et $P' = \mathbf{def}\, p = R\, \mathbf{in}\, (P_1' \mid Q_1)$. Par conséquent, en utilisant les règles (def mu), (par tau) et (def mu), nous pouvons dériver la transition suivante:

$$Q \stackrel{\tau}{\longmapsto} \mathbf{def}\, p = R\, \mathbf{in}\, (\mathbf{def}\, p = R\, \mathbf{in}\, P_1' \mid Q_1)$$

  La preuve est similaire dans le cas (par lambda) suivie par (def mu);
- **cas (par in) suivie par (def mu):** nous avons $\mu_1 = \mathbf{in}\, u.(\tilde{b};\tilde{a})$ et $P' = \mathbf{def}\, p = R\, \mathbf{in}\, (P_1' \mid Q_1\ \tilde{b})$. Comme la dernière règle utilisée est (def mu), les noms de $\tilde{a}$ sont tous différent de p (ce qui est aussi vrai pour $\tilde{b}$). Par conséquent, en utilisant les règles (def mu), (par in) et (def mu), nous pouvons dériver la transition suivante:

$$Q \stackrel{\mu}{\longmapsto} \mathbf{def}\, p = R\, \mathbf{in}\, (\mathbf{def}\, p = R\, \mathbf{in}\, P_1' \mid Q_1\ \tilde{b})$$

  La preuve est similaire dans le cas (par out) suivie par (def mu);
- **cas (par com) suivie par (def mu):** le cas le plus intéressant correspond à une «scope extrusion», c'est-à-dire à la transition $P_1 \stackrel{\mathbf{out}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_1')$ avec $p \in \tilde{a}$. Dans ce cas, nous pouvons dériver une transition similaire à partir de Q en utilisant les règles (def open), (par com) et (def mu). En particulier nous avons:

$$\left\{\begin{array}{l} P' = \mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu}\, \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (P_1 \mid Q_1))) \\ Q' = \mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu}\, \tilde{v})(\mathbf{def}\, p = R, D\, \mathbf{in}\, ((\mathbf{def}\, p = R\, \mathbf{in}\, P_1) \mid Q_1)))\end{array}\right.$$

Le résultat suit du fait que $P' \mathcal{D}_\infty Q'$. Dans le cas $p \notin \tilde{a}$, nous pouvons construire une transition à partir de Q en utilisant les règles (def out), (par com) et (def mu);

– **cas (par out) suivie par (def open):** nous avons $(P_1 \mid Q_1) \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;(P_1' \mid Q_1))$ et

$$P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} P' = ((p = R,D) \mid \mathbf{def}\, p = R\, \mathbf{in}\, (P_1' \mid Q_1))$$

Par conséquent, en utilisant les règles (def open), (par out) et (def mu), nous pouvons dériver la transition suivante à partir de Q:

$$Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} Q' = ((p = R,D) \mid \mathbf{def}\, p = R\, \mathbf{in}\, ((\mathbf{def}\, p = R\, \mathbf{in}\, P_1') \mid Q_1))$$

Le résultat suit du fait que $P' \mathcal{D}_\infty Q'$ et du fait que, pour tout couple (P,Q) dans $\mathcal{D}_\infty$, on a:

$$(\mathbf{def}\, p = R\, \mathbf{in}\, P)\, \mathcal{D}_\infty\, (\mathbf{def}\, p = R\, \mathbf{in}\, Q)$$

– **cas (par out) suivie par (def com):** nous avons $\mu = \tau$ et $\mu_1 = \mathbf{out}\, p.(\tilde{v};\tilde{a})$ et $P' = \mathbf{def}\, p = R\, \mathbf{in}\, (R\ \tilde{a} \mid (P_1' \mid Q))$. Par conséquent, en utilisant les règles (def com), (par tau) et (def mu) nous pouvons dériver la transition suivante:

$$\begin{array}{rcl} Q \stackrel{\tau}{\longmapsto} Q' & = & \mathbf{def}\, p = R\, \mathbf{in}\, (\mathbf{def}\, p = R\, \mathbf{in}\, (R\ \tilde{a} \mid P_1') \mid Q) \\ & \mathcal{D}_\infty & \mathbf{def}\, p = R\, \mathbf{in}\, ((\mathbf{def}\, p = R\, \mathbf{in}\, (R\ \tilde{a}) \mid \mathbf{def}\, p = R\, \mathbf{in}\, (P_1')) \mid Q) \\ & \mathcal{D}_\infty & P' \end{array}$$

La preuve dans les cas où la transition «provient de Q_1», c'est-à-dire les cas tels que $Q_1 \stackrel{\mu_1}{\longmapsto} Q_1'$, est similaire. Nous utilisons simplement le fait que $((\mathbf{def}\, p = R\, \mathbf{in}\, P)\ a)\, \mathcal{D}_\infty\, (\mathbf{def}\, p = R\, \mathbf{in}\, (P\ a))$.

cas $P = \mathbf{def}\, p = R\, \mathbf{in}\, (P_1\ p)$ **et** $Q = \mathbf{def}\, p = R\, \mathbf{in}\, ((\mathbf{def}\, p = R\, \mathbf{in}\, P_1)\ p)$**:** nous supposons que $P_1 \stackrel{\mu_1}{\longmapsto} P_1'$ et nous faisons une analyse par cas sur les deux dernières règles de l'inférence de $P \stackrel{\mu}{\longmapsto} P'$:

– **cas (app tau) suivie par (def mu):** nous avons $\mu = \mu_1 = \tau$ et $P' = \mathbf{def}\, p = R\, \mathbf{in}\, (P_1'\ p)$. Par conséquent, en utilisant les règles (def mu), (par tau) et (def mu), nous pouvons dériver la transition suivante:

$$Q \stackrel{\tau}{\longmapsto} \mathbf{def}\, p = R\, \mathbf{in}\, (\mathbf{def}\, p = R\, \mathbf{in}\, P_1'\ p)$$

– **cas (app com) suivie par (def mu):** nous avons $\mu = \tau$, $\mu_1 = \lambda\, p$ et $P' = \mathbf{def}\, p = R\, \mathbf{in}\, P_1'$. Nous omettons de donner la preuve qu'il existe un processus P_1'' tel que, pour tout nom v, on a: $P_1 \stackrel{\lambda\, v}{\longmapsto} P_1''\{v/x\}$, ce qui implique, en particulier, que $P_1' = P_1''\{p/x\}$. Pour éviter la capture des noms libres, nous utilisons un nom neuf q et nous utiliserons parfois l'α-équivalence pour convertir le processus $(\mathbf{def}\, p = R\, \mathbf{in}\, P)$ en $(\mathbf{def}\, q = R\{q/p\}\, \mathbf{in}\, P\{q/p\})$. Cette transformation est valide car l'α-équivalence sur les actions est déjà considérée implicitement. En utilisant les règles (def mu), (app tau) et (def mu), nous pouvons dériver les transitions suivantes:

$$\begin{array}{l} P \stackrel{\tau}{\longmapsto} P' = \mathbf{def}\, p = R\, \mathbf{in}\, (P_1''\{p/x\}) \\ Q \stackrel{\tau}{\longmapsto} Q' = \mathbf{def}\, p = R\, \mathbf{in}\, (\mathbf{def}\, q = R\{q/p\}\, \mathbf{in}\, (P_1''\{q/p\}\{p/x\})) \end{array}$$

Le résultat suit alors du fait que $\mathcal{D}_\infty$ contient l'ensemble des couples: $((\mathbf{def}\, D\, \mathbf{in}\, P\{u/x\}), (\mathbf{def}\, D'\, \mathbf{in}\, P\{v/x\}))$ tels que $(\mathbf{def}\, D\, \mathbf{in}\, u)\, \mathcal{D}_\infty\, (\mathbf{def}\, D'\, \mathbf{in}\, v)$;

– **cas (app in) suivie par (def mu):** nous avons $\mu_1 = \mathbf{in}\, u.(p,\tilde{b};\tilde{a})$ et $P' = \mathbf{def}\, p = R\, \mathbf{in}\, P_1'$. Comme la dernière règle utilisée est (def mu), les noms du tuple $\tilde{a}$ sont différent de p (ceci est aussi vrai pour $\tilde{b}$). Par conséquent, en utilisant règle (def mu), (app in) et (def mu), nous pouvons dériver la transition suivante à partir de Q:

$$Q \stackrel{\mu}{\longmapsto} \mathbf{def}\, p = R\, \mathbf{in}\, (\mathbf{def}\, p = R\, \mathbf{in}\, P_1')\, \mathcal{D}_\infty\, (\mathbf{def}\, p = R\, \mathbf{in}\, P_1') = P'$$

– **cas (app out) suivie par (def open):** nous avons $P_1 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_1')$ et $P \stackrel{\mathbf{out}\ u.(\tilde{v};(\tilde{a},p))}{\longmapsto} P' = ((p = R,D); \mathbf{def}\ p = R\ \mathbf{in}\ (P_1'\ p))$. Le cas intéressant est celui où p est déjà contenu dans $\tilde{a}$. Dans ce cas, en utilisant les règles (def open), (app out) et (def open), nous pouvons dériver une transition à partir de Q telle que (nous choisissons un nouveau nom q pour remplacer le nom lié p dans P_1 et nous désignons par R_q le processus $R\{q/p\}$)

$$Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a},q)}{\longmapsto} Q' = ((q = R_q,p = R,D) \mid \mathbf{def}\ q = R_q\ \mathbf{in}\ (\mathbf{def}\ p = R\ \mathbf{in}\ P_1'))$$

Le résultat suit du fait que $(\mathbf{def}\ q = R_q\ \mathbf{in}\ (\mathbf{def}\ p = R\ \mathbf{in}\ P_1'))\ \mathcal{D}_\infty\ (\mathbf{def}\ p = R\ \mathbf{in}\ P_1')$ et du fait que $(\mathbf{def}\ p = R,D\ \mathbf{in}\ (\tilde{a},p))\ \mathcal{D}_\infty\ (\mathbf{def}\ q = R_q,p = R,D\ \mathbf{in}\ (\tilde{a},q))$

La preuve est similaire dans le cas $P = \mathbf{def}\ p = R,q = S\ \mathbf{in}\ P$ et $Q = \mathbf{def}\ p = R,q = S\ \mathbf{in}\ (\mathbf{def}\ p = R\ \mathbf{in}\ P)$, et dans le cas $P = \mathbf{def}\ p = R,q = S\ \mathbf{in}\ P$ et $Q = \mathbf{def}\ p = R,q = (\mathbf{def}\ p = R\ \mathbf{in}\ S)\ \mathbf{in}\ P$.

cas $P = \mathbf{def}\ p = R\ \mathbf{in}\ ((\boldsymbol{\lambda} x)P_1)$ **et** $Q = (\boldsymbol{\lambda} x)(\mathbf{def}\ p = R\ \mathbf{in}\ P_1)$**:** nous avons la condition annexe que $x \notin \mathbf{fn}(R)$. Les deux dernières règles de l'inférence de $P \stackrel{\mu}{\longmapsto} P'$ sont (lambda) suivie par (def mu). Par conséquent $\mu = \lambda\, u$ et $P' = \mathbf{def}\ p = R\ \mathbf{in}\ (P_1\{u/x\})$. Le résultat suit alors par application de la règle (lambda) et du fait que $x \notin \mathbf{fn}(R)$ implique $(\mathbf{def}\ p = R\ \mathbf{in}\ P_1)\{u/x\} = \mathbf{def}\ p = R\ \mathbf{in}\ (P_1\{u/x\})$. Dans le cas symétrique, nous utilisons le fait que la transition $Q \stackrel{\mu}{\longmapsto} Q'$ utilise la règle (lambda).

cas $P = \mathbf{def}\ p = R\ \mathbf{in}\ (\langle u \Leftarrow P_1\rangle)$ **et** $Q = \langle u \Leftarrow (\mathbf{def}\ p = R\ \mathbf{in}\ P_1)\rangle$**:** nous utilisons le fait que les deux dernières règles de la transition $P \stackrel{\mu}{\longmapsto} P'$ sont (decl) suivie par (def mu). Par conséquent $\langle u \Leftarrow P_1\rangle \stackrel{\mathbf{in}\ u.(\tilde{b};\tilde{a})}{\longmapsto} (P\ \tilde{a})$, et comme la dernière règle est (def mu), $p \notin \tilde{b} \cup \tilde{a}$. Le résultat suit du fait que $p \notin \tilde{a}$ implique $(\mathbf{def}\ p = R\ \mathbf{in}\ P_1)\ \tilde{a} \equiv \mathbf{def}\ p = R\ \mathbf{in}\ (P_1\ \tilde{a})$.

7.3 Règles pour l'équivalence structurelle

cas $P = P_1 \mid \mathbf{0}$ **et** $Q = P_1$**:** si $\kappa(\mu) \neq \mathbf{out}$, alors la dernière règle de la transition $P \stackrel{\mu}{\longmapsto} P'$ est (par tau), (par lambda) ou (par in), la règle est (par out) dans les autres cas. Le résultat suit alors du fait que $(\mathbf{0}\ \tilde{a})\ \mathcal{D}_\infty\ \mathbf{0}$ et que $(P \mid \mathbf{0})\ \mathcal{D}_\infty\ P$. La preuve est similaire dans le cas symétrique.

cas $P = P_1 \mid Q_1$ **et** $Q = Q_1 \mid P_1$**:** le cas intéressant correspond à l'utilisation de la règle (par com). Supposons que $P_1 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_1')$, que $Q_1 \stackrel{\mathbf{in}\ u.(\epsilon;\tilde{a})}{\longmapsto} Q_1'$, et que $P \stackrel{\tau}{\longmapsto} (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (P_1' \mid Q_1'))$. Par conséquent nous pouvons dériver une transition à partir de Q telle que $Q \stackrel{\tau}{\longmapsto} (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (Q_1' \mid P_1'))$. Le résultat suit du fait que $P\ \mathcal{D}_\infty\ Q$ implique $(\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ P)\ \mathcal{D}_\infty\ (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ Q)$. La preuve est similaire dans le cas $P = P_1 \mid (Q_1 \mid R_1)$ et $Q = (P_1 \mid Q_1) \mid R_1$.

cas $P = \mathbf{def}\ p = R\ \mathbf{in}\ Q$ **et** $p \notin \mathbf{fn}(Q)$**:**

– **cas** $\kappa(\mu) \neq \mathbf{out}$**:** la dernière règle de l'inférence de $P \stackrel{\mu}{\longmapsto} P'$ est (def mu), et nous avons $Q \stackrel{\mu}{\longmapsto} Q'$ et $P \stackrel{\mu}{\longmapsto} \mathbf{def}\ p = R\ \mathbf{in}\ Q'$. Comme $p \notin \mathbf{fn}(Q)$ implique que $p \notin \mathbf{fn}(Q')$, le résultat suit du fait que $(\mathbf{def}\ p = R\ \mathbf{in}\ Q')\ \mathcal{D}_\infty\ Q'$;

– **cas** $\kappa(\mu) = \mathbf{out}$**:** soit μ l'action $\mathbf{out}\ u.(\tilde{v};\tilde{a})$). Comme $p \notin \mathbf{fn}(Q)$, la dernière règle utilisée ne peut pas être (def com). Supposons que la dernière règle soit (def out), nous avons: $Q \stackrel{\mu}{\longmapsto} (D'; Q')$ et $P \stackrel{\mu}{\longmapsto} (D'; \mathbf{def}\ p = R\ \mathbf{in}\ Q')$. Le résultat suit alors du fait que $(\mathbf{def}\ p = R\ \mathbf{in}\ Q')\ \mathcal{D}_\infty\ Q'$ et que $(\mathbf{def}\ D'\ \mathbf{in}\ \tilde{a})\ \mathcal{D}_\infty\ (\mathbf{def}\ D'\ \mathbf{in}\ \tilde{a})$. Supposons que la dernière règle soit (open def). Alors nous avons $P \stackrel{\mu}{\longmapsto} ((p = R,D') \mid Q')$, et comme $p \notin \mathbf{fn}(Q)$, nous pouvons conclure que $p \notin \tilde{a}$ et $p \notin \mathbf{fn}(D')$. Par conséquent $(\mathbf{def}\ p = R,D'\ \mathbf{in}\ \tilde{a})\ \mathcal{D}_\infty\ (\mathbf{def}\ D'\ \mathbf{in}\ \tilde{a})$, ce qui est le résultat attendu.

La preuve est similaire dans le case $P = (\boldsymbol{\nu} u)Q$ et $u \notin \mathbf{fn}(Q)$.

cas $P = \mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu} u)P_1)$ **et** $Q = (\boldsymbol{\nu} u)(\mathbf{def}\, p = R\, \mathbf{in}\, P_1)$**:** nous avons la condition annexe que $u \notin \mathbf{fn}(R)$. Supposons que $P_1 \stackrel{\mu_1}{\longmapsto} P_1'$ (avec $\kappa(m_1) \neq \mathbf{out}$). Par conséquent les deux dernières règles de l'inférence de $P \stackrel{\mu}{\longmapsto} P'$ sont (new mu) suivie par (def mu) et nous pouvons donc dériver une transition similaire à partir de Q en utilisant les règles (def mu) et (new mu). La preuve est similaire dans le cas où $P_1 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_1')$. La preuve est aussi similaire dans le cas le symétrique, c'est-à-dire si $P = (\boldsymbol{\nu} u)(\mathbf{def}\, p = R\, \mathbf{in}\, P_1)$ et $Q = \mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu} u)P_1)$, et dans les cas où: $P = (\mathbf{def}\, p = R, q = S\, \mathbf{in}\, P)$ et $Q = (\mathbf{def}\, q = S, p = R\, \mathbf{in}\, P)$, ou: $P = (\boldsymbol{\nu} u)((\boldsymbol{\nu} v)P)$ et $Q = (\boldsymbol{\nu} v)((\boldsymbol{\nu} u)P)$.

cas $P = ((\boldsymbol{\nu} u)P_1 \mid Q_1)$ **et** $Q = (\boldsymbol{\nu} u)(P_1 \mid Q_1)$**:** le cas le plus intéressant correspond à l'utilisation de la règle (par com) dans la transition $P \stackrel{\mu}{\longmapsto} P'$. Nous considérons le cas tel que P_1 fait une transition **out** (et que la dernière règle est (new out)), et que Q_1 fait une transition **in**. La preuve est similaire si l'inférence de la transition de P_1 se termine par la règle (open new). Tous les autres cas sont similaire au cas $P = ((\boldsymbol{\nu} u)P_1\ a)$. Supposons que:

$$(\boldsymbol{\nu} u)P_1 \stackrel{\mathbf{out}\ p.(\tilde{v};\tilde{a})}{\longmapsto} (D; (\boldsymbol{\nu} u)(P_1')) \qquad P' = (\boldsymbol{\nu} \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, ((\boldsymbol{\nu} u)(P_1') \mid Q_1'))$$

Par conséquent, nous pouvons dériver une transition similaire à partir de Q telle que:

$$Q \stackrel{\tau}{\longmapsto} Q' = (\boldsymbol{\nu} u)((\boldsymbol{\nu} \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (P_1' \mid Q_1')))$$

Le résultat suit alors du fait que $P' \equiv Q'$ implique $P\, \mathcal{D}_\infty\, Q'$. La preuve est similaire dans le cas symétrique.

cas $P = \mathbf{def}\, p = R\, \mathbf{in}\, P_1 \mid Q_1$ **et** $Q = \mathbf{def}\, p = R\, \mathbf{in}\, (P_1 \mid Q_1)$**:** le cas intéressant est tel que les deux dernière règles utilisées dans l'inférence de $P \stackrel{\mu}{\longmapsto} P'$ sont (def com) suivie par (par tau). Dans les autres cas, nous sommes dans une situation comparable à la preuve du cas précédent. Nous supposons que: $\mu = \tau$ et que:

$$P_1 \stackrel{\mathbf{out}\ p.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_1') \qquad P' = \mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu} \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R\ \tilde{a} \mid P_1'))) \mid Q_1$$

Par conséquent nous pouvons dériver la transition suivante à partir de Q:

$$Q \stackrel{\tau}{\longmapsto} Q' = \mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu} \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R\ \tilde{a} \mid P_1' \mid Q_1)))$$

Le résultat suit du fait que $P' \equiv Q'$ implique $P'\, \mathcal{D}_\infty\, Q'$. La preuve est similaire dans le cas symétrique et dans le cas: $P = \mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu} u)P_1)$ et $Q = (\boldsymbol{\nu} u)(\mathbf{def}\, p = R\, \mathbf{in}\, P_1)$.

cas $P = (\mathbf{def}\, p = R\, \mathbf{in}\, P_1)\ a$ **et** $Q = \mathbf{def}\, p = R\, \mathbf{in}\, (P_1\ a)$**:** nous avons la condition annexe que $a \neq p$. Le cas intéressant est tel que les deux dernières règles utilisées dans l'inférence de $P \stackrel{\mu}{\longmapsto} P'$ sont (tau def) suivie par (app tau). Dans les autres cas, nous sommes dans une situation comparable à la preuve du cas précédent. Nous supposons que: $\mu = \tau$ et que:

$$P_1 \stackrel{\mathbf{out}\ p.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_1') \qquad P' = \mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu} \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R\ \tilde{a} \mid P_1')))\ a$$

Par conséquent nous pouvons dériver la transition suivante: $Q \stackrel{\tau}{\longmapsto} Q''$ avec
$Q'' = \mathbf{def}\, p = R\, \mathbf{in}\, ((\boldsymbol{\nu} \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R\ \tilde{a}\ a \mid P_1'\ a)))$. Le résultat suit du fait que $P' \equiv Q'$ implique $P'\, \mathcal{D}_\infty\, Q'$. La preuve est similaire dans le cas symétrique.

cas $P = (\boldsymbol{\nu} u)P_1\ a$ **et** $Q = (\boldsymbol{\nu} u)(P_1\ a)$**:** nous étudions les deux dernières règles de l'inférence de $P \stackrel{\mu}{\longmapsto} P'$. Si $\kappa(\mu) \neq \mathbf{out}$, ces règles sont (new mu) suivie par (app lambda), (app tau) ou (app in). Dans les autres cas, ce sont (new out) ou (new open) suivies par (app out).

- **cas (new mu) suivie par (app com):** dans ce cas, on a: $\mu = \tau$, et $P_1 \stackrel{\lambda\, a}{\longmapsto} P_1'$, et $P' = (\boldsymbol{\nu} u)P_1'$ et nous pouvons dériver la transition suivante: $Q \stackrel{\tau}{\longmapsto} (\boldsymbol{\nu} u)P_1'$. Le résultat suit du fait que $(\boldsymbol{\nu} u)P_1'\, \mathcal{D}_\infty\, (\boldsymbol{\nu} u)P_1'$. La preuve est similaire dans le cas (new mu) suivie de (app in);

- **cas (new mu) suivie par (app tau):** dans ce cas, on a: $\mu = \tau$, et $P_1 \stackrel{\tau}{\longmapsto} P_1'$, et $P' = (\boldsymbol{\nu} u)P_1'\ a$ et nous pouvons dériver la transition suivante: $Q \stackrel{\tau}{\longmapsto} (\boldsymbol{\nu} u)(P_1'\ a)$. Le résultat suit du fait que $((\boldsymbol{\nu} u)P_1'\ a)\ \mathcal{D}_\infty\ (\boldsymbol{\nu} u)(P_1'\ a)$;
- **cas (new out) suivie par (app out):** dans ce cas, on a: $\kappa(\mu) = \mathbf{out}$, et $P_1 \stackrel{\mu}{\longmapsto} (D; P_1')$, et $P' = (D; ((\boldsymbol{\nu} u)P_1'\ a))$ et nous pouvons dériver la transition suivante: $Q \stackrel{\mu}{\longmapsto} (D; (\boldsymbol{\nu} u)(P_1'\ a))$. Le résultat suit su fait que $((\boldsymbol{\nu} u)P_1'\ a)\ \mathcal{D}_\infty\ (\boldsymbol{\nu} u)(P_1'\ a)$. La preuve est similaire dans le cas (new open) suivie par (app out).

La preuve est similaire dans le cas symétrique.

cas $P = (P_1 \mid Q_1)\ a$ **et** $Q = (P_1\ a) \mid (Q_1\ a)$**:** nous étudions les deux dernières règles de l'inférence de $P \stackrel{\mu}{\longmapsto} P'$. Le premier cas est tel que la transition provient «uniquement» de P_1 (resp. de Q_1), c'est-à-dire que la transition est de la forme $P_1 \stackrel{\mu'}{\longmapsto} P_1'$ implique $(P_1 \mid Q_1) \stackrel{\mu'}{\longmapsto} (P_1' \mid Q_1\ \tilde{a})$.

- **cas (par tau) suivie par (app tau):** nous avons $\mu = \tau$, et $P_1 \stackrel{\mu}{\longmapsto} P_1'$, et $(P_1 \mid Q_1) \stackrel{\mu}{\longmapsto} (P_1' \mid Q_1)$ et $P \stackrel{\tau}{\longmapsto} (P_1' \mid Q_1)$. Par conséquent, en utilisant les règles (app tau) et (par tau), nous pouvons dériver une transition similaire pour Q, c'est-à-dire : $Q \stackrel{\tau}{\longmapsto} Q' = (P_1' \mid Q_1)$;
- **cas (par lambda) suivie par (app tau):** nous avons: $\mu = \tau$, et $P_1 \stackrel{\lambda u}{\longmapsto} P_1'$, et $(P_1 \mid Q_1) \stackrel{\lambda u}{\longmapsto} (P_1' \mid Q_1\ a)$ et $P \stackrel{\tau}{\longmapsto} (P_1' \mid Q_1\ a)$. Par conséquent, en utilisant les règles (app tau) et (par lambda), nous pouvons dériver une transition similaire pour Q, c'est-à-dire : $Q \stackrel{\tau}{\longmapsto} Q' = (P_1' \mid Q_1\ a)$;
- **cas (par in) suivie par (app in):** dans ce cas, on a: $\mu = \mathbf{in}\ v.(\tilde{b}; \tilde{a})$. Soit μ' l'action $\mathbf{in}\ v.((a,\tilde{b}); \tilde{a})$, nous avons: $P_1 \stackrel{\mu'}{\longmapsto} P_1'$, et $(P_1 \mid Q_1) \stackrel{\mu'}{\longmapsto} (P_1' \mid Q_1\ a\ \tilde{b})$ et $P \stackrel{\mu}{\longmapsto} (P_1' \mid Q_1\ a\ \tilde{b}))$. Par conséquent, en utilisant les règles (app in) et (par in), nous pouvons dériver une transition similaire pour Q, c'est-à-dire : $Q \stackrel{\mu}{\longmapsto} Q' = (P_1' \mid (Q_1\ a)\ \tilde{b})$;
- **cas (par out) suivie par (app out):** dans ce cas on a: $\mu = \mathbf{out}\ v.(\tilde{v}; (\tilde{a}, a))$. Soit μ' l'action $\mathbf{out}\ v.(\tilde{v}; \tilde{a})$, nous avons: $P_1 \stackrel{\mu'}{\longmapsto} (D; P_1')$, et $(P_1 \mid Q_1) \stackrel{\mu'}{\longmapsto} (D; (P_1' \mid Q_1))$ et $P \stackrel{\mu}{\longmapsto} (D; ((P_1' \mid Q_1)\ a))$. Par conséquent nous pouvons dériver une transition similaire pour Q, c'est-à-dire $Q \stackrel{\mu}{\longmapsto} Q' = (D; ((P_1'\ a) \mid (Q_1\ a)))$.

Le dernier cas est tel que les deux dernières règles de la transition $P \stackrel{\mu}{\longmapsto} P'$ sont (par com) suivie par (app tau). Par exemple le cas tel que $P_1 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P_1')$, et $Q_1 \stackrel{\mathbf{in}\ v.(\epsilon;\tilde{a})}{\longmapsto} Q_1'$, et $(P_1 \mid Q_1) \stackrel{\tau}{\longmapsto} (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (P_1' \mid Q_1'))$ et $P \stackrel{\tau}{\longmapsto} (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (P_1' \mid Q_1'))\ a$. Par conséquent, en utilisant règle (app out) et le lemme 7.2, nous pouvons dériver les transitions suivantes: $(P_1\ a) \stackrel{\mathbf{out}\ u.(\tilde{v};(\tilde{a},a))}{\longmapsto} (D; P_1'\ a)$ et $(Q_1\ a) \stackrel{\mathbf{in}\ v.(\epsilon;(\tilde{a},a))}{\longmapsto} Q_1''$, avec $Q_1'' \equiv (Q_1'\ a)$. En utilisant la règle (par com), il suit que $Q \stackrel{\tau}{\longmapsto} Q' = (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ ((P_1'\ a) \mid Q_1'')) \equiv P'$, ce qui est le résultat attendu.

cas $P = \langle u \Leftarrow P_1 \rangle\ a$ **et** $Q = \langle u \Leftarrow P_1 \rangle$**:** la seule possibilité est que $\langle u \Leftarrow P_1 \rangle$ «fasse» une action **in**. Dans ce cas, nous avons $Q \stackrel{\mathbf{in}\ v.(\tilde{b};\tilde{a})}{\longmapsto} (P_1\ \tilde{a})$, et comme nous pouvons aussi deriver la transition $\langle u \Leftarrow P_1 \rangle \stackrel{\mathbf{in}\ v.((a,\tilde{b});\tilde{a})}{\longmapsto} P_1\ \tilde{a}$, il suit, en utilisant la règle (app in), que $P \stackrel{\mathbf{in}\ v.(\tilde{b};\tilde{a})}{\longmapsto} P_1\ \tilde{a}$. La preuve est similaire dans le cas symétrique $P = (\mathbf{0}\ a)$.

□

CHAPITRE 8

Propriétés de la def-bisimulation et de la congruence à barbes

DANS CE CHAPITRE, nous prouvons plusieurs propriétés de la def-bisimulation. Nous montrons que cette relation est une congruence, qu'elle contient l'équivalence structurelle et qu'elle vérifie les lois de réplication. Nous montrons ensuite que la def-bisimulation est incluse dans la congruence à barbes, ceci permet de prouver que l'interprétation du λ-calcul, donnée dans la partie I, est adéquate.

Théorème 8.1 *La def-bisimulation est une congruence qui contient la relation d'équivalence structurelle, c'est-à-dire que* $\equiv \subset \sim_d$, *et pour tout contexte* $\mathbf{C}$, *on a* $P \sim_d Q$ *implique* $\mathbf{C}[P] \sim_d \mathbf{C}[Q]$.

Preuve La relation identité $\mathcal{I}d$ est un exemple particulier de bisimulation modulo contextes, et donc de bisimulation modulo F. Par conséquent, en utilisant le lemme 6.2, on obtient que $\bigcup_{n\geq 0} F^n(\mathcal{I}d) \subseteq \sim_d$, et donc: $\equiv \subseteq \bigcup_{n\geq 0} F^n(\mathcal{I}d) \subseteq \sim_d$. Pour montrer que $\sim_d$ est une congruence, on utilise le fait que $\sim_d$ est un exemple de bisimulation modulo contextes. Par conséquent $\bigcup_{n\geq 0} F^n(\sim_d)$ est une def-bisimulation, et donc $\bigcup_{n\geq 0} F^n(\sim_d) \subseteq \sim_d$. Ceci implique que $F(\sim_d) \subseteq \sim_d$, et donc que $\sim_d$ est compositionelle. Comme c'est aussi une relation d'équivalence, c'est donc une congruence. □

Le résultat suivant implique que les définitions peuvent être «distribuées» sous n'importe quels contextes, et donc qu'elles agissent comme des substitutions explicites [1].

Lemme 8.2 (Les lois de réplication) *Les lois suivantes sont vraies:*

(*i*) *Si* $p \notin \mathbf{fn}(P)$, *alors* $\mathbf{def}\, p = R \,\mathbf{in}\, P \sim_d P$;

(*ii*) $\mathbf{def}\, p = R \,\mathbf{in}\, (P \mid Q) \sim_d \mathbf{def}\, p = R \,\mathbf{in}\, ((\mathbf{def}\, p = R \,\mathbf{in}\, P) \mid Q)$;

(*iii*) *si* $p \neq q$ *et* $q \notin \mathbf{fn}(R)$, *alors:*

$$\begin{array}{lcl} \mathbf{def}\, p = R \,\mathbf{in}\, (\mathbf{def}\, q = S \,\mathbf{in}\, P) & \sim_d & \mathbf{def}\, p = R, q = S \,\mathbf{in}\, (\mathbf{def}\, p = R \,\mathbf{in}\, P) \\ \mathbf{def}\, p = R \,\mathbf{in}\, (\mathbf{def}\, q = S \,\mathbf{in}\, P) & \sim_d & \mathbf{def}\, q = (\mathbf{def}\, p = R \,\mathbf{in}\, S) \,\mathbf{in}\, (\mathbf{def}\, p = R \,\mathbf{in}\, P) \end{array}$$

(*iv*) $\mathbf{def}\, p = R \,\mathbf{in}\, (\langle u \Leftarrow P \rangle) \sim_d \langle u \Leftarrow (\mathbf{def}\, p = R \,\mathbf{in}\, P) \rangle$;

(*v*) *si* x *n'est pas libre dans* R, *alors:* $(\mathbf{def}\, p = R \,\mathbf{in}\, (\boldsymbol{\lambda} x) P)) \sim_d (\boldsymbol{\lambda} x)(\mathbf{def}\, p = R \,\mathbf{in}\, P)$;

(*vi*) *pour tout contexte* $\mathbf{C}$ *qui ne capture pas un nom libre de* R, *et pour tout processus* P, *la loi suivante est vraie* $(\mathbf{def}\, p = R \,\mathbf{in}\, (\mathbf{C}[P])) \sim_d (\mathbf{def}\, p = R \,\mathbf{in}\, (\mathbf{C}[\mathbf{def}\, p = R \,\mathbf{in}\, P]))$.

On montre qu'un corollaire des lois (i) et (ii) est:

$$\mathbf{def}\, p = R \,\mathbf{in}\, (P \mid Q) \sim_d (\mathbf{def}\, p = R \,\mathbf{in}\, P) \mid (\mathbf{def}\, p = R \,\mathbf{in}\, Q)$$

qui est la loi que nous annoncions en introduction de cette partie.

Preuve Les lois de (i) à (v) sont des conséquences directes du fait que $F(\sim_d) \subseteq \sim_d$. Pour prouver la loi 8.2-(vi), on montre que les deux processus P_1 et P_2 de la forme:

$$P_1 =_{\text{def}} \mathbf{D}[P] \qquad \text{et} \qquad P_2 =_{\text{def}} \mathbf{D}[\mathbf{def}\, p = R \,\mathbf{in}\, P]$$

où $\mathbf{D}$ est le contexte $\mathbf{def}\, p = R \,\mathbf{in}\, (\mathbf{C}[.])$, sont bisimilaires. Soit $[\![.]\!]$ la fonction sur les processus définie par les règles suivantes (dans chaque cas, nous utilisons l'α-conversion pour éviter la capture des noms libre de R):

$$\begin{array}{rcl}
[\![\mathbf{def}\, p = R \,\mathbf{in}\, u]\!] & = & \mathbf{def}\, p = [\![R]\!] \,\mathbf{in}\, u \\
[\![\mathbf{def}\, p = R \,\mathbf{in}\, (\boldsymbol{\lambda} x)P)]\!] & = & \mathbf{def}\, p = [\![R]\!] \,\mathbf{in}\, (\boldsymbol{\lambda} x)([\![\mathbf{def}\, p = R \,\mathbf{in}\, P]\!]) \\
[\![\mathbf{def}\, p = R \,\mathbf{in}\, (P\ a)]\!] & = & \mathbf{def}\, p = [\![R]\!] \,\mathbf{in}\, ([\![\mathbf{def}\, p = R \,\mathbf{in}\, P]\!]\ a) \\
[\![\mathbf{def}\, p = R \,\mathbf{in}\, (\boldsymbol{\nu} u)P)]\!] & = & \mathbf{def}\, p = [\![R]\!] \,\mathbf{in}\, (\boldsymbol{\nu} u)([\![\mathbf{def}\, p = R \,\mathbf{in}\, P]\!]) \\
[\![\mathbf{def}\, p = R \,\mathbf{in}\, (P \mid Q)]\!] & = & \mathbf{def}\, p = R \,\mathbf{in}\, (([\![\mathbf{def}\, p = R \,\mathbf{in}\, P]\!]) \mid ([\![\mathbf{def}\, p = R \,\mathbf{in}\, Q]\!])) \\
[\![\mathbf{def}\, p = R \,\mathbf{in}\, (\langle u \Leftarrow P\rangle)]\!] & = & \mathbf{def}\, p = [\![R]\!] \,\mathbf{in}\, \langle u \Leftarrow ([\![\mathbf{def}\, p = R \,\mathbf{in}\, P]\!])\rangle \\
[\![\mathbf{def}\, p = R \,\mathbf{in}\, (\mathbf{def}\, q = S \,\mathbf{in}\, Q)]\!] & = & \mathbf{def}\, p = [\![R]\!] \,\mathbf{in} \left(\begin{array}{l} \mathbf{def}\, q = [\![\mathbf{def}\, p = R \,\mathbf{in}\, S]\!] \\ \mathbf{in}\, [\![\mathbf{def}\, p = R \,\mathbf{in}\, Q]\!] \end{array} \right)
\end{array}$$

et telle que $[\![.]\!]$ est un homomorphisme dans les autres cas. La fonction $[\![.]\!]$ est utilisée pour distribuer au maximum les définitions dans un processus. Si nous ajoutons la constante $[\_]$, telle que $[\![\mathbf{def}\, p = R \,\mathbf{in}\, [\_]\,]\!] = [\_]$, cette fonction devient aussi une transformation de contextes en contextes. En utilisant les lois de réplication, on vérifie que pour tous processus P on a: $[\![P]\!] \sim_d P$ et $[\![\mathbf{D}]\!][P] \sim_d \mathbf{D}[P]$. Il est donc immédiat que:

$$P_1 \sim_d [\![P_1]\!] = [\![\mathbf{D}]\!][[\![\mathbf{def}\, p = R \,\mathbf{in}\, P]\!]] \qquad \text{et que} \qquad P_2 \sim_d \mathbf{D}[[\![\mathbf{def}\, p = R \,\mathbf{in}\, P]\!]]$$

Le résultat attendu, c'est-à-dire $P_1 \sim_d P_2$, suit par transitivité de la bisimulation. □

8.1 Relation entre transitions étiquetées et réductions

Dans cette section, nous montrons que le système de transitions étiquetées est «fidèle» à la notion de réduction, c'est-à-dire que toute réduction $P \to Q$, peut être associée à une τ-transition $P \stackrel{\tau}{\longmapsto} Q'$, du système de transitions étiquetées. Ce résultat est formellement donné dans le lemme 8.4. Néanmoins, une différence principale avec les systèmes de transitions étiquetées classiques définis pour le π-calcul, est que dans notre cas, les τ-transitions ne sont pas directement équivalentes aux réductions. Ainsi nous avons déjà vu l'exemple suivant dans la section 5.2.

$$\langle u \Leftarrow Q\rangle \mid \mathbf{def}\, p = R \,\mathbf{in}\, (u\, p \mid P) \begin{cases} \to & \\ \stackrel{\tau}{\longmapsto} \equiv & \end{cases} \begin{array}{l} \mathbf{def}\, p = R \,\mathbf{in}\, ((Q\, p) \mid P) \\ \mathbf{def}\, p = R \,\mathbf{in}\, ((Q\ p) \mid (\mathbf{def}\, p = R \,\mathbf{in}\, P)) \end{array}$$

L'intuition est qu'une définition est dupliquée, dès lors qu'un message peut rendre public la référence utilisée pour la nommer. Pour établir ce résultat formellement, nous prouvons d'abord une propriété préliminaire.

Lemme 8.3 (Relation entre actions et transitions étiquetées)

actions tau: *Si* $P \stackrel{\tau}{\longmapsto} P'$ *et* $\mathbf{E}$ *est un contexte d'évaluation, alors* $\mathbf{E}[P] \stackrel{\tau}{\longmapsto} \mathbf{E}[P']$*;*
actions lambda: *si* $P \stackrel{\lambda\, a}{\longmapsto} P'$*, alors* $(Pa) \to P'$*;*
actions in: *si* $P \stackrel{\mathbf{in}\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'$*, alors* $(P\tilde{b}) \mid (u\tilde{a}) \to P'$*;*

actions out: *si* $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;P')$*, alors il existe un processus* R *tel que:*

$$P \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (R \mid u\tilde{a})) \quad et \quad (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ R) \sim_d (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ P')$$

Preuve On démontre chaque propriété séparément.

actions tau on montre que si $P \stackrel{\tau}{\longmapsto} P'$ et $\mathbf{E}$ est un contexte d'évaluation, alors $\mathbf{E}[P] \stackrel{\tau}{\longmapsto} \mathbf{E}[P']$. La preuve est faite par induction sur la taille de $\mathbf{E}$. Supposons que $P \stackrel{\tau}{\longmapsto} P'$, le cas $\mathbf{E} = [\_]$ est trivial.

- **cas** $(\mathbf{E}u)$ **et** $(\mathbf{E} \cdot l)$**:** le résultat est impliqué par la règle (app tau);
- **cas** $(\mathbf{E} \mid P)$ **et** $(P \mid \mathbf{E})$**:** le résultat est impliqué par la règle (par tau);
- **cas** $(\boldsymbol{\nu}u)(\mathbf{E})$**:** le résultat est impliqué par la règle (new mu);
- **cas** $(\mathbf{def}\ D\ \mathbf{in}\ \mathbf{E})$**:** le résultat est impliqué par la règle (def mu).

actions lambda on montre que si $P \stackrel{\lambda\, a}{\longmapsto} P'$, alors $(Pa) \rightarrow P'$. Dans cette preuve, on suppose être dans le cas d'une application à une valeur. Le cas de la sélection est similaire. La preuve est une analyse par cas sur la dernière règle de la transition $P \stackrel{\lambda\, a}{\longmapsto} P'$.

- **cas (lambda):** on a $P = (\boldsymbol{\lambda}x)Q$ et $P' = Q\{^a/_x\}$, par conséquent le résultat découle de la règle (red beta);
- **cas (par lambda):** on a $P \stackrel{\lambda\, a}{\longmapsto} P'$ implique $(P \mid Q) \stackrel{\lambda\, a}{\longmapsto} (P' \mid (Qa))$. Or, d'après l'hypothèse d'induction $(Pa) \rightarrow P'$. Par conséquent le résultat est impliqué par les règles (red struct) et (red context): $(P \mid Q)a \equiv (Pa) \mid (Qa) \rightarrow P' \mid (Qa)$;
- **cas (new mu) et (def mu):** dans le premier cas, on a $P =_{\mathrm{def}} (\boldsymbol{\nu}v)Q$, et $P' =_{\mathrm{def}} (\boldsymbol{\nu}v)Q'$, et $Q \stackrel{\lambda\, a}{\longmapsto} Q'$. On utilise l'hypothèse d'induction pour montrer que $(Qa) \rightarrow Q'$. De plus, les pré-conditions de la règle (new mu) impliquent que b est différent de v. Donc $(Pa) \equiv (\boldsymbol{\nu}v)(Qa) \rightarrow (\boldsymbol{\nu}v)Q'$. La preuve est similaire dans le cas (def mu).

actions in on montre que si $P \stackrel{\mathbf{in}\ a.(\tilde{b};\tilde{a})}{\longmapsto} P'$, alors $(P\tilde{b}) \mid (u\tilde{a}) \rightarrow P'$. Un corollaire de cette propriété est que $P \stackrel{\mathbf{in}\ u.(\epsilon;\tilde{a})}{\longmapsto} P'$ implique $P \mid (u\tilde{a}) \rightarrow P'$. Soit $\tilde{a}$ (respectivement $\tilde{b}$) le tuple $(a_1,\ldots,a_n)$ (resp. $(b_1,\ldots,b_m)$). On fait une analyse par cas sur la dernière règle de la transition $P \stackrel{\mathbf{in}\ u.(\tilde{b};\tilde{a})}{\longmapsto} P'$.

- **cas (decl):** on a $P = \langle u \Leftarrow Q \rangle \stackrel{\mathbf{in}\ u.(\tilde{b};\tilde{a})}{\longmapsto} (Q\tilde{a}) = P'$ et $(P\tilde{b}) \equiv P$. Par conséquent, en utilisant l'équivalence structurelle et la règle (red decl), on obtient $(P\tilde{b}) \mid (u\tilde{a}) \rightarrow P'$;
- **cas (new mu) and (def mu):** dans le premier cas on a $P = (\boldsymbol{\nu}v)Q$, et $P' = (\boldsymbol{\nu}v)Q'$, et $Q \stackrel{\mu}{\longmapsto} Q'$. La condition annexe de la règle implique que v n'est pas dans l'ensemble $u,\tilde{a},\tilde{b}$. Par conséquent

  $$((\boldsymbol{\nu}v)Q)\ \tilde{b}) \mid (u\tilde{a}) \equiv (\boldsymbol{\nu}v)(Q\tilde{b} \mid (u\tilde{a}))$$

  le résultat découle de l'hypothèse d'induction et de la règle (red context). La preuve est similaire dans le cas (def mu);
- **cas (par in):** on a $P \stackrel{\mathbf{in}\ u.(\tilde{b};\tilde{a})}{\longmapsto} P'$ implique $(P \mid Q) \stackrel{\mathbf{in}\ u.(\tilde{b};\tilde{a})}{\longmapsto} (P' \mid (Q\tilde{b}))$ et $(P \mid Q)\tilde{b} \equiv (P\tilde{b}) \mid (Q\tilde{b})$. Le résultat est impliqué par l'hypothèse d'induction et la règle (red context);
- **cas (app in):** dans ce cas on utilise le fait que, avec nos notations, $((Pd)\tilde{b})$ est égal à $(P(d,\tilde{b}))$.

actions out on montre que si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;P')$, alors il existe un processus R tel que

$$P \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (R \mid u\ \tilde{a})) \ \text{ et } \ (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ R) \sim_d (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ P')$$

L' intuition est que l'on peut toujours augmenter la portée des définitions se trouvant sous un contexte d'évaluation. La preuve se fait par une analyse par cas de la dernière règle de $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;P')$.

- **cas (out):** on a $P = u \stackrel{\mathbf{out}\ u.(\epsilon;\epsilon)}{\longmapsto} (\emptyset;\mathbf{0})$. Or $u \equiv (\boldsymbol{\nu}\epsilon)(\mathbf{def}\ \emptyset\ \mathbf{in}\ (\mathbf{0} \mid u))$ et $\mathbf{0} \sim_d (\boldsymbol{\nu}\epsilon)(\mathbf{def}\ \emptyset\ \mathbf{in}\ \mathbf{0})$. Le résultat est donc obtenu en choisissant $\mathbf{0}$ pour R;

– **cas (par out), (app out), (new out) et (def out):** dans le cas (par out), par exemple, on a $P \xmapsto{\mathbf{out}\ u.(\tilde{v};\tilde{a})} (D; P')$. Par conséquent il existe un processus R tel que $P \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R \mid u\ \tilde{a}))$. De plus la condition $\mathbf{fn}(Q) \cap (\mathbf{decl}(D) \cup \tilde{v}) = \emptyset$ implique que

$$(P \mid Q) \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, ((R \mid Q) \mid u\ \tilde{a}))$$

Le résultat est donc obtenu en remarquant que $\equiv \subset \sim_d$ et que $\sim_d$ est une congruence (cf. théorème 8.1);

– **cas (new open) et (def open):** dans le cas (new open), on utilise le fait que $P \sim_d Q$ implique $(\boldsymbol{\nu}v)P \sim_d (\boldsymbol{\nu}v)Q$. Dans le cas (def open), on doit aussi utiliser le lemme 8.2 pour prouver que

$$(\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, p = S, D\, \mathbf{in}\, R) \sim_d (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, p = S, D\, \mathbf{in}\, (\mathbf{def}\, p = S\, \mathbf{in}\, R))$$

□

Dans le lemme 8.3, on a relié chaque type d'actions à une règle de réduction. Par exemple les actions lambda sont associées à la règle (red beta). Cette propriété permet de prouver que le s.t.e. simule les réductions, c'est-à-dire qu'on a la propriété suivante.

Lemme 8.4 ($\xmapsto{\tau}$ simule $\rightarrow$) *S'il existe une réduction d'un processus P à un processus P', alors il existe une transition étiquetée équivalente modulo équivalence structurelle, c'est-à-dire que:*

(i) si $P \rightarrow P'$, alors il existe un processus R tel que $P \equiv R$, et $R \xmapsto{\tau} R'$, et $R' \equiv P'$.

Réciproquement, l'existence d'une τ-transition, implique l'existence d'une réduction. Dans ce cas, les «résiduels» sont bisimilaires, c'est-à-dire que:

(ii) si $P \xmapsto{\tau} P'$, alors il existe un processus R tel que $P \rightarrow R$ et $R \sim_d P'$.

Preuve (Lemme 8.4) La preuve de la propriété (i) est faite par induction sur l'inférence de: $P \rightarrow P'$.

– **cas (red struct):** on a $P \equiv Q$ et $Q \rightarrow P'$. Aussi, en utilisant l'hypothèse d'induction, on montre qu'il existe deux processus R et R' tels que $Q \equiv R$, et $R \xmapsto{\tau} R'$, et $R' \equiv P'$. Le résultat suit de la transitivité de l'équivalence structurelle;

– **cas (red context):** on a $P = \mathbf{E}[Q]$, et $Q \rightarrow Q'$ et $P' = \mathbf{E}[Q']$. Par conséquent il existe un processus R tel que $Q \equiv R$ et $R \xmapsto{\tau} R$; et $R' \equiv Q'$. Il suffit alors d'utiliser le lemme 8.3 et le fait que $\equiv$ soit une congruence, pour prouver que $P \equiv \mathbf{E}[R] \xmapsto{\tau} \mathbf{E}[R'] \equiv P'$;

– **cas (red beta) et (red sel):** dans le premier cas, nous avons $P =_{\text{def}} (\boldsymbol{\lambda}x)Q)a$ et $P' =_{\text{def}} Q\{a/x\}$. On utilise alors la règle (lambda), qui implique que $(\boldsymbol{\lambda}x)Q) \xmapsto{\lambda\, a} Q\{a/x\}$, et règle (app com), qui implique que $P \xmapsto{\tau} P'$. La preuve est similaire dans le deuxième cas;

– **cas (red decl):** on a $P =_{\text{def}} (\langle u \Leftarrow Q\rangle \mid u\tilde{a})$ et $P' =_{\text{def}} (Q\tilde{a})$. La règle (decl) implique que $\langle u \Leftarrow Q\rangle \xmapsto{\mathbf{in}\ u.(\epsilon;\tilde{a})} (Q\tilde{a})$. Par conséquent, en utilisant les règles (out) et (app out), on obtient: $(u\tilde{a}) \xmapsto{\mathbf{out}\ u.(\epsilon;\tilde{a})} (\emptyset; (\mathbf{0}\tilde{a}))$. Le résultat suit de l'application de la règle (par com), puisqu'elle permet de montrer que $P \xmapsto{\tau} (P' \mid (\mathbf{0}\tilde{a})) \equiv P'$;

– **cas (red def):** on a $P =_{\text{def}} (\mathbf{def}\, p = T\, \mathbf{in}\, \mathbf{E}[p])$ et $P' =_{\text{def}} (\mathbf{def}\, p = T\, \mathbf{in}\, \mathbf{E}[T])$ avec comme hypothèse que p et les noms libre de T ne sont pas liés par $\mathbf{E}$. En utilisant l'équivalence structurelle, on peut toujours supposer que $\mathbf{E}$ est mis dans la forme normale suivante.

$$\mathbf{E} = (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (\, [\_]\, \tilde{a} \mid Q))$$

Aussi, en utilisant les règles (out), (app out), (new open) et (def open), on montre que $\mathbf{E}[P] \xmapsto{\mathbf{out}\ p.(\tilde{v};\tilde{a})} (D; (\mathbf{0}\tilde{a} \mid Q))$. Finalement on utilise la règle (def com) pour montrer que

$$P \xmapsto{\tau} (\mathbf{def}\, p = T\, \mathbf{in}\, ((\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (T\tilde{a} \mid (\mathbf{0}\tilde{a} \mid Q))))) \equiv P$$

La preuve de la deuxième propriété est faite par induction sur l'inférence de $P \stackrel{\tau}{\longmapsto} P'$.

- **cas (par tau), (app tau), (new mu) et (def mu):** le résultat suit de la règle (red context) et du fait que $\sim_d$ soit une congruence;
- **cas (app com):** supposons que nous ayons une béta-réduction (le cas de la sélection est similaire). Alors $P =_{\text{def}} (Qa)$ et $Q \stackrel{\lambda\, a}{\longmapsto} P'$. Le résultat suit du lemme 8.3;
- **cas (par com) et (def com):** dans le cas de la composition parallèle, on a $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')$ et $Q \stackrel{\mathbf{in}\ u.(\epsilon;\tilde{a})}{\longmapsto} Q'$ impliquent $(P \mid Q) \stackrel{\tau}{\longmapsto} (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (P' \mid Q'))$. En utilisant le lemme 8.3, on montre que ces hypothèses impliquent qu'il existe un processus R tel que

$$\begin{cases} Q \mid (u\tilde{a}) \to Q' \\ P \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (u\tilde{a} \mid R)) \\ (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, R) \sim_d (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, P') \end{cases}$$

En combinant ces trois propriétés, nous obtenons la suite de relations suivantes:

$$\begin{array}{lll} (P \mid Q) \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (a\ \tilde{\beta} \mid R)) \mid Q & & \\ \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R \mid (Q \mid a\ \tilde{\beta}))) & \text{(d'après les hypothèses de (par com))} \\ \to (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R \mid Q')) & \text{(d'après les hypothèses de (red context))} \\ \sim_d (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (P' \mid Q')) & \text{(d'après les hypothèses du théorème 8.1)} \end{array}$$

Dans la dernière égalité, nous utilisons le fait que $(\mathbf{fn}(Q) \cap (\tilde{v}, \mathbf{decl}(D)) = \emptyset)$ implique que $(\mathbf{fn}(Q') \cap (\tilde{v}, \mathbf{decl}(D)) = \emptyset)$. La preuve est similaire dans le cas (def com).

□

Nous donnons maintenant une caractérisation des valeurs en nous aidant du système de transitions étiquetées.

Lemme 8.5 (Caractérisation des barbes) *Le processus P est une valeur, c'est-à-dire $P\downarrow$, si et seulement si il existe une valeur a telle que $P \stackrel{\lambda\, a}{\longmapsto} P'$.*

Preuve La première implication est prouvée par induction sur la taille de P, la réciproque est prouvée par induction sur l'inférence de $P \stackrel{\lambda\, a}{\longmapsto} P'$. □

Finalement, nous prouvons le résultat principal de cette partie, qui est que la def-bisimulation forte est incluse dans la congruence à barbes.

Théorème 8.6 *Si $P \sim_d Q$, alors $P \approx_b Q$.*

Preuve Nous montrons que la relation $\sim_d$ est une bisimulation à barbes. Dans le chapitre 6, nous avons déjà prouvé que $\sim_d$ est une congruence. Par conséquent il suffit de montrer que (1) : la def-bisimulation est une simulation, et que (2) : la def-bisimulation préserve les barbes.

(1) supposons que $P \sim_d Q$ et $P \to P'$. Nous montrons qu'il existe un processus Q' tel que $Q \to Q'$ et $P' \sim_d Q'$. En utilisant le lemme 8.4-(i), on montre qu'il existe un processus R tel que $(\sharp) : P \equiv R$, et $R \stackrel{\tau}{\longmapsto} R'$, et $(\natural) : R' \equiv P'$. En utilisant le théorème 8.1 et la propriété $(\sharp)$, on en déduit que $Q \sim_d R$. Par conséquent, puisque $P \sim_d Q$, il existe un processus S tel que $Q \stackrel{\tau}{\longmapsto} S$ avec $R \sim_d S$. De plus, d'après le théorème 8.1 et la propriété $(\natural)$, on a $P' \sim_d S$. Finalement, en utilisant le lemme 8.4-(ii), on prouve l'existence d'un processus Q' tel que $Q \to Q'$ et $Q' \sim_d S$, et donc tel que $Q' \sim_d P'$;

(2) supposons que $P\downarrow$. D'après le lemme 8.5, il existe un nom a tel que $P \stackrel{\lambda\, a}{\longmapsto} P'$. Or, par hypothèse, $P \sim_d Q$. Donc il existe un processus Q' tel que $Q \stackrel{\lambda\, a}{\longmapsto} Q'$ et $P' \sim_d Q'$. Par conséquent, en utilisant la propriété réciproque du lemme 8.5, on obtient que $Q\downarrow$.

□

Ce dernier résultat implique que les lois de réplication (cf. lemme 8.2) sont également vraies pour la congruence à barbes, c'est-à-dire qu'on a les propriétés suivantes.

Théorème 8.7 (Lois de réplication) *Les lois suivantes sont vraies:*

$$\begin{array}{lrcl}
(i) & \mathbf{def}\, p = R \,\mathbf{in}\, P & \approx_b & P \qquad (u \notin \mathbf{fn}(P)) \\
(ii) & \mathbf{def}\, p = R \,\mathbf{in}\, (P \mid Q) & \approx_b & (\mathbf{def}\, p = R \,\mathbf{in}\, P) \mid (\mathbf{def}\, p = R \,\mathbf{in}\, Q) \\
(iii) & \mathbf{def}\, p = R \,\mathbf{in}\, (Pa) & \approx_b & \mathbf{def}\, p = R \,\mathbf{in}\, ((\mathbf{def}\, p = R \,\mathbf{in}\, P)a) \\
(iv) & \mathbf{def}\, p = R \,\mathbf{in}\, (\mathbf{def}\, q = S \,\mathbf{in}\, P) & \approx_b & \mathbf{def}\, q = (\mathbf{def}\, p = R \,\mathbf{in}\, S) \,\mathbf{in}\, (\mathbf{def}\, p = R \,\mathbf{in}\, P) \\
(v) & \mathbf{def}\, p = R \,\mathbf{in}\, (\langle u \Leftarrow P \rangle) & \approx_b & \langle u \Leftarrow (\mathbf{def}\, p = R \,\mathbf{in}\, P) \rangle \\
(vi) & (\mathbf{def}\, p = R \,\mathbf{in}\, (\lambda x) P)) & \approx_b & (\lambda x)(\mathbf{def}\, p = R \,\mathbf{in}\, P) \qquad (x \notin \mathbf{fn}(R))
\end{array}$$

Nous conjecturons que la relation de def-bisimulation faible est aussi un congruence à barbes.

Conjecture 8.8 *Si $P \approx_d Q$, alors $P \approx_b Q$.*

8.2 Interprétation du lambda-calcul

Dans cette section, nous étudions le λ-calcul complet, c'est-à-dire sans stratégie de réduction particulière. En particulier on autorise la réduction sous les abstractions. On rappelle que la relation de béta-conversion, notée $\leftrightarrow_\beta$, est la plus petite congruence telle que $(\lambda x.M)N \leftrightarrow_\beta M\{\!\{N/x\}\!\}$.

Nous utilisons les lois de réplication – pour la congruence à barbes, cf. théorème 8.6 — pour prouver la correction de la traduction du λ-calcul vers $\boldsymbol{\pi}^\star$ donnée dans la partie I. En reprenant les notations de la section 3.3, on interprète un terme M par le processus $(\!|M|\!) =_{\text{def}} [\![(M)^\star]\!]$, où $(.)^\star$ est l'interprétation de $\boldsymbol{\Lambda}$ dans $\boldsymbol{\Lambda}^\star$ donnée dans la définition 3.3, et $[\![.]\!]$ est l'interprétation de $\boldsymbol{\Lambda}^\star$ dans $\boldsymbol{\pi}^\star$ donnée dans la définition 3.4. Nous commençons par définir une construction dérivée:

$$P[p{:=}Q] \;=_{\text{def}}\; \mathbf{def}\, p = Q \,\mathbf{in}\, P \qquad (p \notin \mathbf{fn}(Q))$$

qui nous permet d'établir formellement un lien entre l'opérateur dérivé **def**, et les substitutions explicites [1]. En utilisant $[p{:=}Q]$, la fonction $(\!|.|\!)$ peut se réécrire plus simplement. Nous rappelons que nous ne faisons pas de distinction entre les variables de λ et les noms de $\boldsymbol{\pi}^\star$.

Définition 8.1 (Codage du $\boldsymbol{\lambda}$-calcul complet)

$$\begin{array}{rcl}
(\!|x|\!) & = & x \\
(\!|\lambda x.M|\!) & = & (\lambda x)(\!|M|\!) \\
(\!|MN|\!) & = & (\!|Mp|\!)[p{:=}(\!|N|\!)]
\end{array}$$

On remarque que la traduction d'un rédex du λ-calcul est un rédex du calcul bleu. De plus, en utilisant le théorème 4.3, on obtient que:

$$(\!|(\lambda x.M)N|\!) \;=\; ((\lambda x)(\!|M|\!))p)[p{:=}(\!|N|\!)] \;\approx_b\; (\!|M|\!)\{p/x\})[p{:=}(\!|N|\!)] \tag{8.1}$$

En utilisant les lois de réplication, on montre que $[p{:=}(\!|N|\!)]$ vérifie les mêmes lois que la substitution $\{\!\{N/p\}\!\}$, et en particulier $(\!|M|\!)[x{:=}(\!|N|\!)] \approx_b (\!|M\{\!\{N/x\}\!\}|\!)$, ce qui implique de manière immédiate la propriété suivante.

Théorème 8.9 *Si $M \leftrightarrow_\beta N$, alors $(\!|M|\!) \approx_b (\!|N|\!)$.*

Plus précisément, on montre que les lois de réplication sont l'équivalent, pour $[p{:=}(\!|N|\!)]$, des transformations définies dans le λ-calcul avec substitutions explicites. Plus précisément, si nous

ajoutons l'opérateur de substitution explicite $M[x{:=}N]$ au λ-calcul, et si nous définissons le terme $(\!|M[x{:=}N]|\!)$ par $(\!|M|\!)[x{:=}(\!|N|\!)]$. Alors, soit p,q deux références telles que $p \neq q$ et $q \notin \mathbf{fn}(R)$, on a:

$$\begin{array}{rcll}
(P[q{:=}Q])[p{:=}R] & \approx_b & (P[p{:=}R])[q{:=}(Q[p{:=}R])] & \text{(théorème 8.7-}(iv)\text{)} \\
p[p{:=}R] & \approx_b & R & \text{(proposition 4.4)} \\
q[p{:=}R] & \approx_b & q & \text{(théorème 8.7-}(i)\text{)} \\
(\!|\lambda y.M|\!)[p{:=}(\!|N|\!)] & \approx_b & (\boldsymbol{\lambda} y)((\!|M|\!)[p{:=}(\!|N|\!)]) & \text{(théorème 8.7-}(vi)\text{)} \\
(\!|ML|\!)[p{:=}(\!|N|\!)] & \approx_b & (\!|(M[p{:=}N])\ (L[p{:=}N])|\!) & \text{(théorème 8.7-}(iv)\text{)}
\end{array}$$

Par conséquent, en utilisant la relation (8.1), nous pouvons montrer que:

$$(\!|(\lambda x.M)N|\!) \ \approx_b\ ((\!|M|\!)\{p/x\})[p{:=}(\!|N|\!)] \ \approx_b\ (\!|M|\!)\{\!\{(\!|N|\!)/x\}\!\} \ =\ (\!|M\{\!\{N/x\}\!\}|\!)$$

Ce dernier résultat doit être comparé à la remarque de R. Milner [98, section 4] qui écrit que «*la correspondance entre le λ-calcul avec substitutions explicites et le π-calcul, est peut être plus étroite qu'entre le λ-calcul et le π-calcul* ».

Dans le chapitre suivant, nous étudions comment les définitions et les résultats donnés pour le calcul bleu symétrique, peuvent être transposés au cas du calcul bleu dissymétrique.

CHAPITRE 9

Équivalence dans le calcul dissymétrique

DANS LA PARTIE PRÉCÉDENTE, nous avons défini la congruence à barbes $\approx_b$ et la bisimulation «étiquetée» $\sim_d$, pour le calcul symétrique. Dans cette partie, nous définissons ces relations pour le calcul dissymétrique, et nous montrons comment transposer les résultats prouvés dans les chapitres précédents, en particulier en ce qui concerne les techniques de preuve up-to.

9.1 Conventions

Dans cette partie, nous considérons une variante du calcul bleu local et dissymétrique, dans laquelle les déclarations répliquées sont remplacées par les définitions. Nous choisissons également de suivre les conventions données dans la section 4.2. En particulier, nous introduisons le processus $\mathbf{0}$ directement dans la syntaxe des termes de $\boldsymbol{\pi}^\star$, et nous considérons une règle d'équivalence structurelle supplémentaire:

$$(\mathbf{0} \mid P) \equiv P$$

9.2 Congruence à barbes

La congruence à barbes, dénotée $\approx_b$, est la plus grande bisimulation qui préserve une notion de convergence entre processus et qui est une congruence [68]. Comme dans la définition du chapitre 4, notre intuition est qu'une valeur est un processus qui peut se réduire lorsqu'il est appliqué à un nom ou à une sélection. Cette intuition nous donne une définition des barbes pour le calcul symétrique, qui est différente de de la définition 4.1.

Définition 9.1 (Barbes) Une *barbe* est un processus engendré par la grammaire suivante.

$$V ::= (\boldsymbol{\lambda} x)P \mid [\,Q\,, l = P\,] \mid (P \mid V) \mid (\boldsymbol{\nu} u)V \mid \mathbf{def}\, D\, \mathbf{in}\, V$$

Nous dénotons $P \downarrow$ le fait que P soit une valeur, et $P \Downarrow$ lorsqu'il existe une valeur V telle que $P \xrightarrow{*} V$.

La différence avec la définition précédente est que, si V est une barbe, alors le processus $(V \mid P)$ ne l'est pas nécessairement. En effet, dans le calcul dissymétrique, la règle d'équivalence structurelle pour la distribution de l'application sur la composition parallèle est $(P \mid Q)\ a \equiv P \mid (Q\ a)$, alors que dans le calcul symétrique on a: $(P \mid Q)\ a \equiv (P\ a) \mid (Q\ a)$.

9.3 Système de transitions étiquetées

Dans cette section, nous définissons un système de transitions étiquetées pour le calcul bleu dissymétrique. Ce système est très proche de celui utilisé dans les chapitres précédents, la principale différence étant que l'on doit mémoriser si les actions proviennent de la gauche ou de la droite d'un parallèle. Pour conserver cette information au cours d'une transition, on ajoute des annotations aux actions **in** et **out**:

$$b ::= \blacktriangleleft \mid \blacktriangleright \qquad \mu ::= \tau \mid \lambda\, a \mid \mathbf{out}^b\, u.(\tilde{v};\tilde{a}) \mid \mathbf{in}^b\, u.(\tilde{b};\tilde{a})$$

Ces actions ont la même interprétation que dans le chapitre 5.1, à la différence que $\mathbf{out}^{\blacktriangleleft}\, u.(\tilde{v};\tilde{a})$ signifie que l'émission provient d'un processus qui est à la gauche d'au moins une composition parallèle (cf. règle (par out 1)), et que $\mathbf{out}^{\blacktriangleright}\, u.(\tilde{v};\tilde{a})$ signifie que l'émission provient du processus «le plus à droite» (cf. règles (out) et (par out 2)).

Intuitivement, un message situé à droite de toutes les compositions parallèles peut être étendu par une application. Ce mécanisme correspond à la règle (app out 2) du s.t.e. défini à la section suivante. Symétriquement, on peut remarquer que dans les transitions $P \overset{\mathbf{in}^{\blacktriangleleft}\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'$, le tuple $\tilde{b}$ est toujours égal à ϵ. Ce qui correspond au fait qu'un processus situé à gauche d'une composition parallèle, ne peut pas recevoir un nom fourni par l'environnement.

9.3.1 Définition du système de transitions étiquetées

Axiomes

$$(\boldsymbol{\lambda} x)P \overset{\lambda\, u}{\longmapsto} P\{u/x\} \quad \text{(lambda)} \qquad a \overset{\mathbf{out}^{\blacktriangleright}\, u.(\epsilon;\epsilon)}{\longmapsto} (\emptyset;\mathbf{0}) \quad \text{(out)}$$

$$[\,R\,,l = N\,] \overset{\lambda\, l}{\longmapsto} N \quad \text{(lambda)} \qquad [\,R\,,l = N\,] \overset{\lambda\, k}{\longmapsto} (R \cdot k) \ (\text{if } k \neq l) \quad \text{(lambda)}$$

$$\langle u \Leftarrow P\rangle \overset{\mathbf{in}^{\blacktriangleright}\, u.(\epsilon;\tilde{a})}{\longmapsto} (\mathbf{0} \mid P\ \tilde{a}) \quad \text{(decl 1)} \qquad \langle u \Leftarrow P\rangle \overset{\mathbf{in}^{\blacktriangleleft}\, u.(\epsilon;\tilde{a})}{\longmapsto} (P\ \tilde{a} \mid \mathbf{0}) \quad \text{(decl 2)}$$

Règles pour les actions τ, λ et in

$$\frac{P \overset{\lambda\, a}{\longmapsto} P'}{Q \mid P \overset{\lambda\, a}{\longmapsto} Q \mid P'} \ \text{(par lambda)} \qquad \frac{P \overset{\lambda\, a}{\longmapsto} P'}{(P\ a) \overset{\tau}{\longmapsto} P'} \ \text{(app com)}$$

$$\frac{P \overset{\tau}{\longmapsto} P'}{P \mid Q \overset{\tau}{\longmapsto} P' \mid Q} \ \text{(par tau)} \qquad \frac{P \overset{\tau}{\longmapsto} P'}{Q \mid P \overset{\tau}{\longmapsto} Q \mid P'} \ \text{(par tau)} \qquad \frac{P \overset{\tau}{\longmapsto} P'}{(P\ a) \overset{\tau}{\longmapsto} (P'\ a)} \ \text{(app tau)}$$

$$\frac{P \overset{\mu}{\longmapsto} P' \quad (\kappa(\mu) \in \{\tau,\lambda,\mathbf{in}\}\ \&\ u \notin \mathbf{fn}(\mu))}{(\boldsymbol{\nu} u)P \overset{\mu}{\longmapsto} (\boldsymbol{\nu} u)P'} \ \text{(new mu)}$$

$$\frac{P \overset{\mu}{\longmapsto} P' \quad (\kappa(\mu) \in \{\tau,\lambda,\mathbf{in}\}\ \&\ p \notin \mathbf{fn}(\mu))}{\mathbf{def}\, p = R\, \mathbf{in}\, P \overset{\mu}{\longmapsto} \mathbf{def}\, p = R\, \mathbf{in}\, P'} \ \text{(def mu)}$$

Règles pour les actions in

$$\frac{P \overset{\mathbf{in}^{\blacktriangleleft}\, u.(\epsilon;\tilde{a})}{\longmapsto} P'}{P \mid Q \overset{\mathbf{in}^{\blacktriangleleft}\, u.(\epsilon;\tilde{a})}{\longmapsto} P' \mid Q} \ \text{(par in 1)} \qquad \frac{P \overset{\mathbf{in}^{\blacktriangleright}\, u.(\epsilon;\tilde{a})}{\longmapsto} P'}{P \mid Q \overset{\mathbf{in}^{\blacktriangleright}\, u.(\tilde{b};\tilde{a})}{\longmapsto} (Q\ \tilde{b}) \mid P'} \ \text{(par in 2)}$$

$$\frac{P \overset{\mathbf{in}^b\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'}{Q \mid P \overset{\mathbf{in}^b\, u.(\tilde{b};\tilde{a})}{\longmapsto} Q \mid P'} \text{ (par in 3)}$$

$$\frac{P \overset{\mathbf{in}^{\blacktriangleleft}\, u.(\epsilon;\tilde{a})}{\longmapsto} P'}{(P\ c) \overset{\mathbf{in}^{\blacktriangleleft}\, u.(\epsilon;\tilde{a})}{\longmapsto} (P' c)} \text{ (app in 1)} \qquad \frac{P \overset{\mathbf{in}^{\blacktriangleright}\, u.((c,\tilde{b});\tilde{a})}{\longmapsto} P'}{(P\ c) \overset{\mathbf{in}^{\blacktriangleright}\, u.(\tilde{b};\tilde{a})}{\longmapsto} P'} \text{ (app in 2)}$$

Règles pour l'action out

$$\frac{P \overset{\mathbf{out}^b\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad (\tilde{v},\mathbf{decl}(D)) \cap \mathbf{fn}(Q) = \emptyset}{P \mid Q \overset{\mathbf{out}^{\blacktriangleleft}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; (P' \mid Q))} \text{ (par out 1)}$$

$$\frac{P \overset{\mathbf{out}^b\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad (\tilde{v},\mathbf{decl}(D)) \cap \mathbf{fn}(Q) = \emptyset}{Q \mid P \overset{\mathbf{out}^b\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; (Q \mid P'))} \text{ (par out 2)}$$

$$\frac{P \overset{\mathbf{out}^{\blacktriangleleft}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad c \notin (\tilde{v},\mathbf{decl}(D))}{(P\ c) \overset{\mathbf{out}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; (P'\ c))} \text{ (app out 1)}$$

$$\frac{P \overset{\mathbf{out}^{\blacktriangleright}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad c \notin (\tilde{v},\mathbf{decl}(D))}{(P\ c) \overset{\mathbf{out}\, u.(\tilde{v};(\tilde{a},c))}{\longmapsto} (D; P')} \text{ (app out 2)}$$

$$\frac{P \overset{\mathbf{out}^b\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad (v \notin u,\tilde{a},\tilde{v},\mathbf{fn}(D))}{(\boldsymbol{\nu} v)P \overset{\mathbf{out}^b\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; (\boldsymbol{\nu} v)P')} \text{ (new out)}$$

$$\frac{P \overset{\mathbf{out}^b\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad (u \neq v)}{(\boldsymbol{\nu} v)P \overset{\mathbf{out}\, u.((v,\tilde{v});\tilde{a})}{\longmapsto} (D; P')} \text{ (new open)}$$

$$\frac{P \overset{\mathbf{out}^b\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad (p \notin u,\tilde{a},\tilde{v},\mathbf{fn}(D))}{\mathbf{def}\, p = R\, \mathbf{in}\, P \overset{\mathbf{out}^b\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; \mathbf{def}\, p = R\, \mathbf{in}\, P')} \text{ (def out)}$$

$$\frac{P \overset{\mathbf{out}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad (p \neq u\ \&\ p \notin (\tilde{v},\mathbf{decl}(D)))}{\mathbf{def}\, p = R\, \mathbf{in}\, P \overset{\mathbf{out}^b\, u.(\tilde{v};\tilde{a})}{\longmapsto} ((p = R,D); \mathbf{def}\, p = R\, \mathbf{in}\, P')} \text{ (def open)}$$

Synchronisation

$$\frac{P \overset{\mathbf{out}^{\blacktriangleright}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad Q \overset{\mathbf{in}^{\blacktriangleright}\, u.(\epsilon;\tilde{a})}{\longmapsto} Q'}{Q \mid P \overset{\tau}{\longmapsto} (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (P' \mid Q'))} \text{ (par com 1)}$$

$$\frac{P \overset{\mathbf{out}^{\blacktriangleright}\, u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad Q \overset{\mathbf{in}^{\blacktriangleleft}\, u.(\epsilon;\tilde{a})}{\longmapsto} Q'}{P \mid Q \overset{\tau}{\longmapsto} (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (P' \mid Q'))} \text{ (par com 2)}$$

$$\frac{P \overset{\mathbf{out}^{\blacktriangleleft}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad Q \overset{\mathbf{in}^{\blacktriangleleft}\ u.(\epsilon;\tilde{a})}{\longmapsto} Q'}{P \mid Q \overset{\tau}{\longmapsto} (\boldsymbol{\nu}\,\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (P' \mid Q'))} \text{ (par com 3)}$$

$$\frac{P \overset{\mathbf{out}^{\blacktriangleleft}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P') \qquad Q \overset{\mathbf{in}^{\blacktriangleleft}\ u.(\epsilon;\tilde{a})}{\longmapsto} Q'}{Q \mid P \overset{\tau}{\longmapsto} (\boldsymbol{\nu}\,\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (Q' \mid P'))} \text{ (par com 4)}$$

$$\frac{P \overset{\mathbf{out}^{\blacktriangleleft}\ p.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')}{\mathbf{def}\ p = R\ \mathbf{in}\ P \overset{\tau}{\longmapsto} \mathbf{def}\ p = R\ \mathbf{in}\ (\boldsymbol{\nu}\,\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (R\ \tilde{a} \mid P'))} \text{ (def com 1)}$$

$$\frac{P \overset{\mathbf{out}^{\blacktriangleright}\ p.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')}{\mathbf{def}\ p = R\ \mathbf{in}\ P \overset{\tau}{\longmapsto} \mathbf{def}\ p = R\ \mathbf{in}\ (\boldsymbol{\nu}\,\tilde{v})(\mathbf{def}\ D\ \mathbf{in}\ (P' \mid R\ \tilde{a}))} \text{ (def com 2)}$$

La présentation du système de transitions étiquetées est compliquée par le fait que l'opérateur de composition parallèle n'est pas symétrique. Les règles de transitions sont donc elles aussi dissymétriques. On peut remarquer plusieurs différences avec le système défini dans le chapitre 5.

- il n'y a pas de symétrique à la règle (par lambda), c'est-à-dire que $P \overset{\lambda\, a}{\longmapsto} P'$, n'implique pas que $(P \mid Q)$ puisse faire une action lambda. Cette règle est cohérente avec la règle de distribution de l'application sur la composition parallèle;
- il y a deux règles de transitions pour $\langle u \Leftarrow P \rangle$. La règle (decl 1) correspond à une communication avec un message situé à droite de toutes les compositions parallèles (cf. règle (par com 1)).

$$\begin{array}{rcl} (\langle u \Leftarrow P\rangle \mid Q) & \overset{\mathbf{in}^{\blacktriangleright}\ u.(\epsilon;\epsilon)}{\longmapsto} & Q \mid (\mathbf{0} \mid P) \\ ((\langle u \Leftarrow P\rangle \mid Q) \mid u) & \overset{\tau}{\longmapsto} & (\mathbf{0} \mid (Q \mid (\mathbf{0} \mid P))) \equiv (Q \mid P) \end{array} \tag{9.1}$$

  la règle (decl 2) correspond à une communication avec un message situé à gauche d'au moins une composition parallèle, ce qui correspond aux règles (par com 2), (par com 3) et (par com 4).

$$\begin{array}{c} (u \mid \langle u \Leftarrow P\rangle) \overset{\tau}{\longmapsto} (\mathbf{0} \mid (P \mid \mathbf{0})) \equiv (P \mid \mathbf{0}) \\ (\langle u \Leftarrow P\rangle \mid (u \mid Q)) \overset{\tau}{\longmapsto} ((P \mid \mathbf{0}) \mid (\mathbf{0} \mid Q)) \equiv (P \mid Q) \end{array}$$

- dans la règle (par in 2), on commute la position des deux processus P et Q. Ceci permet de positionner le résultat de la réception à droite de la composition parallèle, comme dans la transition (9.1) ci-dessus. On peut remarquer que, en utilisant cette règle, on a que $((\langle u \Leftarrow P\rangle \mid Q)\ a \mid u) \overset{\tau}{\longmapsto} (Q\ a \mid P)$, alors que le processus $(\langle u \Leftarrow P\rangle\ a \mid u)$ ne se réduit pas. Dans la règle (par in 3), on n'applique pas le processus Q au nom de $\tilde{b}$, car Q se trouve à gauche d'une composition parallèle;

À partir de ce système de transitions étiquetées, on peut utiliser les définitions des chapitres 5 et 6 pour définir la relation de bisimulation étiquetée $\sim_d$, et les relations de bisimulation modulo contextes, équivalence structurelle et réplication. On peut aussi définir des relations entre transitions étiquetées et réductions, comme nous l'avons fait pour le calcul symétrique dans la section 8.1, en particulier, on montre que les relations concernant les actions τ et λ sont celles données dans le lemme 8.3. Nous prouvons que les relations pour les actions in et out sont similaires à celles données dans le lemme 8.3.

Lemme 9.1 (Relation entre actions et transitions étiquetées)

(*i*) *si* $P \overset{\mathbf{in}^{\blacktriangleright}\ u.(\tilde{b};\tilde{a})}{\longmapsto} P'$, *alors* $(P\tilde{b}) \mid (u\tilde{a}) \to P'$;

(*ii*) *si* $P \overset{\mathbf{in}^{\blacktriangleleft}\ u.(\epsilon;\tilde{a})}{\longmapsto} P'$, *alors* $(u\tilde{a}) \mid P \to P'$;

(*iii*) *si* $P \stackrel{\mathbf{out}^{\blacktriangleright}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')$, *alors il existe un processus* R *tel que:*

$$P \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R \mid u\tilde{a})) \quad \textit{et} \quad (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (R \mid \mathbf{0})) \sim_d (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, P')$$

(*iv*) *si* $P \stackrel{\mathbf{out}^{\blacktriangleleft}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D; P')$, *alors il existe un processus* R *tel que:*

$$P \equiv (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (u\tilde{a} \mid R)) \quad \textit{et} \quad (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, R) \sim_d (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, P')$$

Preuve On prouve chaque propriété séparément. La propriété (*i*) est prouvée par induction sur l'inférence de la transition $P \stackrel{\mathbf{in}^{\blacktriangleright}\ u.(\tilde{b};\tilde{a})}{\longmapsto} P'$. Soit $\tilde{a}$ (respectivement $\tilde{b}$) le tuple $(a_1,\ldots,a_n)$ (resp. $(b_1,\ldots,b_m)$). On fait une analyse par cas sur la dernière règle de la transition $P \stackrel{\mathbf{in}^{\blacktriangleright}\ u.(\tilde{b};\tilde{a})}{\longmapsto} P'$.

- **cas (decl 1):** on a $P = \langle u \Leftarrow Q\rangle \stackrel{\mathbf{in}^{\blacktriangleright}\ u.(\epsilon;\tilde{a})}{\longmapsto} (\mathbf{0} \mid Q\tilde{a}) = P'$, et, en utilisant la règle (red decl 1), on obtient que $P \mid (u\tilde{a}) \to P'$;
- **cas (new mu) and (def mu):** dans le premier cas on a $P = (\boldsymbol{\nu}v)Q$, et $P' = (\boldsymbol{\nu}v)Q'$, et $Q \stackrel{\mu}{\longmapsto} Q'$. La condition annexe de la règle implique que v n'est pas dans l'ensemble $u,\tilde{a},\tilde{b}$. Par conséquent

  $$((\boldsymbol{\nu}v)Q)\ \tilde{b}) \mid (u\tilde{a}) \equiv (\boldsymbol{\nu}v)(Q\tilde{b} \mid (u\tilde{a}))$$

  le résultat découle de l'hypothèse d'induction et de la règle (red context). La preuve est similaire dans le cas (def mu);
- **cas (par in 2):** on a $P \stackrel{\mathbf{in}^{\blacktriangleright}\ u.(\epsilon;\tilde{a})}{\longmapsto} P'$ implique $(P \mid Q) \stackrel{\mathbf{in}^{\blacktriangleright}\ u.(\tilde{b};\tilde{a})}{\longmapsto} ((Q\tilde{b}) \mid P')$ et $(P \mid Q)\tilde{b} \mid (u\tilde{a}) \equiv (Q\tilde{b}) \mid (P \mid (u\tilde{a}))$. Le résultat est impliqué par l'hypothèse d'induction et la règle (red context);
- **cas (par in 3):** on a $P \stackrel{\mathbf{in}^{\blacktriangleright}\ u.(\tilde{b};\tilde{a})}{\longmapsto} P'$ implique $(Q \mid P) \stackrel{\mathbf{in}^{\blacktriangleright}\ u.(\tilde{b};\tilde{a})}{\longmapsto} (Q \mid P')$ et $(Q \mid P)\tilde{b} \mid (u\tilde{a}) \equiv Q \mid (P\tilde{b} \mid u\tilde{a})$. Le résultat est impliqué par l'hypothèse d'induction et la règle (red context);
- **cas (app in 2):** dans ce cas on utilise le fait que, avec nos notations, $((P\ c)\ \tilde{b})$ est égal à $(P(c,\tilde{b}))$.

La preuve de la propriété (*ii*) est similaire, on étudie uniquement le cas des règles (par in 1) et (app in 1).

- **cas (par in 1):** on a $P \stackrel{\mathbf{in}^{\blacktriangleleft}\ u.(\epsilon;\tilde{a})}{\longmapsto} P'$ implique $(P \mid Q) \stackrel{\mathbf{in}^{\blacktriangleleft}\ u.(\epsilon;\tilde{a})}{\longmapsto} (P' \mid Q)$ et $(u\tilde{a}) \mid (P \mid Q) \equiv ((u\tilde{a} \mid P) \mid Q)$. Le résultat est impliqué par l'hypothèse d'induction et la règle (red context);
- **cas (app in 1):** dans ce cas on utilise la règle (red struct) et le fait que, avec nos notations, $(u\tilde{a} \mid (P\ c)) \equiv (u\tilde{a} \mid P)\ c$;

La preuve des propriétés (*iii*) et (*iv*) est similaire à celle donnée dans le lemme 8.3. □

En utilisant les mêmes schémas de preuve que dans les chapitres précédents, nous conjecturons qu'il est possible de prouver, pour la version dissymétrique du calcul bleu, des résultats similaires à ceux des théorèmes 8.1, 8.6 et 8.7. Aussi, dans la suite de cette thèse, nous utiliserons ces théorèmes indifféremment pour le calcul bleu symétrique et dissymétrique.

Dans le chapitre suivant, nous étudions la relation d'expansion et la technique de preuve up-to associée.

CHAPITRE 10

Preuves modulo expansion

UNE TECHNIQUE GÉNÉRALEMENT UTILISÉE POUR DÉMONTRER des résultats de bisimulation, est de fournir une relation et de vérifier, pour chaque couple de processus, que leurs dérivées sont encore en relation. Comme nous l'avons déjà montré dans le chapitre 6, les techniques de preuve up-to permettent de réduire la taille des relations misent en jeu. Elles sont donc d'une aide précieuse pour simplifier les preuves complexes. Dans ce chapitre nous définissons la technique de preuve modulo expansion [113, 119] et nous montrons comment celle-ci peut-être utilisée pour prouver de nouvelles lois. La définition de l'expansion ne dépend pas de la variante du calcul bleu utilisée (symétrique ou dissymétrique), nous n'avons donc pas besoin de préciser la variante que nous considérons dans les résultats donnés ici.

La relation d'expansion, noté $\lesssim_d$, est une version asymétrique de $\approx_d$. Intuitivement, la relation $P \lesssim_d Q$ signifie que P est équivalent à Q, mais aussi que Q doit faire plus de transitions internes pour pouvoir simuler P, ou encore que Q est moins efficace que P [9].

Définition 10.1 (Expansion) La relation $\mathcal{D}$ est une *expansion* si pour tout couple $(P,Q) \in \mathcal{D}$, on a:

- (i) si $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \stackrel{\mu}{\Longrightarrow} Q'$ et $P' \,\mathcal{D}\, Q'$;
- (ii) si $Q \stackrel{\mu}{\longmapsto} Q'$ et $\kappa(\mu) \notin \mathbf{out}$, alors il existe un processus P' tel que: $P \stackrel{\widehat{\mu}}{\longmapsto} P'$ et $P' \,\mathcal{D}\, Q'$;
- (i') si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D; P')$, alors il existe un processus Q' et un environnement D' tels que: $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\,\tilde{c})}{\Longrightarrow} (D'; Q')$, et: $P' \,\mathcal{D}\, Q'$, et $(\mathbf{def}\ D\ \mathbf{in}\ \tilde{b}) \,\mathcal{D}\, (\mathbf{def}\ D'\ \mathbf{in}\ \tilde{c})$;
- (ii') si $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D; Q')$, alors il existe un processus P' et un environnement D' tels que: $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{c})}{\longmapsto} (D'; P')$, et: $P' \,\mathcal{D}\, Q'$, et $(\mathbf{def}\ D\ \mathbf{in}\ \tilde{b}) \,\mathcal{D}\, (\mathbf{def}\ D'\ \mathbf{in}\ \tilde{c})$;
- (iii) pour toute substitution σ, substitution des valeurs pour les variables, on a $P\sigma \,\mathcal{D}\, Q\sigma$.

La relation d'expansion $\lesssim_d$ est telle que $P \lesssim_d Q$ s'il existe une expansion $\mathcal{D}$ telle que $P \,\mathcal{D}\, Q$. On note $\gtrsim_d$ la relation inverse de l'expansion, que nous appelons *contraction*.

Il faut noter que la relation $\lesssim_d$ n'est pas symétrique. Un exemple particulier d'expansion sont les def-bisimulations. Ainsi, on prouve le résultat suivant.

Proposition 10.1 *Si $P \sim_d Q$, alors $P \lesssim_d Q$. Si $P \lesssim_d Q$, alors $P \approx_d Q$.*

Ce qui implique aussi que l'équivalence structurelle est incluse dans la relation d'expansion. On conjecture que les résultats prouvés pour l'expansion dans le π-calcul [113] sont aussi vrais pour le calcul bleu. Ainsi, nous supposons qu'une expansion «modulo expansion» est aussi une expansion.

Définition 10.2 (Expansion modulo $\lesssim_d$) La relation $\mathcal{D}$ est une expansion modulo $\lesssim_d$, si pour tout couple $(P,Q) \in \mathcal{D}$ on a:

(i) si $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \stackrel{\mu}{\Longrightarrow} Q'$ et $P' \sim_d \mathcal{D} \lesssim_d Q'$;

(ii) si $Q \stackrel{\mu}{\longmapsto} Q'$ et $\kappa(\mu) \notin \mathbf{out}$, alors il existe un processus P' tel que: $P \stackrel{\widehat{\mu}}{\longmapsto} P'$ et $P' \lesssim_d \mathcal{D} \lesssim_d Q'$;

Et la relation similaire dans le cas où $\mu = \mathbf{out}\ u.(\tilde{v};\tilde{c})$.

Conjecture 10.2 *Si $\mathcal{D}$ est une expansion modulo $\lesssim_d$, alors $\mathcal{D}$ est une expansion: $\mathcal{D} \subseteq \lesssim_d$.*

La relation d'expansion permet de lier des processus obtenus par «réduction déterministe», c'est-à-dire par des τ-transitions qui vérifient la propriété de Church-Rosser. La béta-réduction et la sélection sont des exemples de ce type de transitons.

Lemme 10.3 *Pour tout processus P, on a: $P\{u/x\} \lesssim_d (\boldsymbol{\lambda}x)P)u$. Les lois équivalentes pour la sélection sont également vraies:*

$$\begin{cases} P & \lesssim_d & [\,Q\,, l = P\,] \cdot l \\ (Q \cdot k) & \lesssim_d & [\,Q\,, l = P\,] \cdot k \end{cases}$$

De même pour la communication avec une définition. Soit $\mathbf{E}$ un contexte qui ne capture ni le nom p, ni les noms libres de R, alors:

$$\mathbf{def}\, p = R \,\mathbf{in}\, \mathbf{E}[R] \ \lesssim_d\ \mathbf{def}\, p = R \,\mathbf{in}\, \mathbf{E}[p]$$

La preuve de ces relations est faite en exhibant une relation qui est une expansion. Pour la première propriété par exemple, on montre que la relation $\{(P\{u/x\},((\boldsymbol{\lambda}x)P)u)\}$ est une expansion. Nous omettons de donner les autres relations. Un autre exemple de réduction déterministe est la communication impliquant un «émetteur unique» et une déclaration.

Lemme 10.4 *Pour tout processus P on a: $P \lesssim_d (\boldsymbol{\nu}u)(\langle u \Leftarrow P\rangle \mid u)$.*

Cette propriété peut s'interpréter comme une loi pour l'opérateur de définition $\mathbf{set}\, x = P\ \mathbf{in}\, Q$, défini dans la définition 3.8. En effet ce lemme permet de prouver que:

$$(Pa) \lesssim_d (\mathbf{set}\, x = P\ \mathbf{in}\, (\mathbf{return}(a)))$$

Nous conjecturons que la technique de preuves modulo expansion est correcte, c'est-à-dire qu'on a le résultat suivant.

Définition 10.3 (Simulation ground modulo expansion) La relation $\mathcal{D}$ est une simulation ground modulo expansion, si pour tout couple $(P,Q) \in \mathcal{D}$ on a:

(i) si $P \stackrel{\mu}{\longmapsto} P'$ et $\kappa(\mu) \neq \mathbf{out}$, alors il existe un processus Q' tel que: $Q \stackrel{\mu}{\Longrightarrow} Q'$ et $P' \gtrsim_d \mathcal{D} \lesssim_d Q'$;

(ii) si $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{a})}{\longmapsto} (D;P')$, alors il existe un processus Q' et un environnement D' tels que: $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\Longrightarrow} (D';Q')$, et $(\mathbf{def}\, D \,\mathbf{in}\, \tilde{a})\, \mathcal{D}\, (\mathbf{def}\, D' \,\mathbf{in}\, \tilde{b})$, et $P' \gtrsim_d \mathcal{D} \lesssim_d Q'$.

Conjecture 10.5 (Preuve modulo expansion) *Si $\mathcal{D}$ et $\mathcal{D}^{-1}$ sont des simulations ground modulo expansion, et si $\mathcal{D}$ est close par substitutions des variables par des noms, alors $\mathcal{D}$ est une bisimulation faible: $\mathcal{D} \subseteq \approx_d$.*

On remarque que, en utilisant ce dernier résultat et le lemme 10.3, on peut fournir une autre preuve pour les résultats donnés dans le théorème 4.3 et la proposition 4.4.

Discussion

À la connaissance de l'auteur, il existe peu de preuve directe de lois non triviales pour la congruence à barbes. Dans son article sur l'interprétation du λ-calcul dans $\boldsymbol{\pi}$ [98], R. Milner donne une raison intuitive à cet état de fait: «la relation de réduction nous informe uniquement sur le comportement interne d'un processus P; elle décrit comment les composants de P peuvent interagir entre eux, mais pas comment P peut interagir avec son environnement.»

Dans cette partie, la solution que nous avons utilisée pour simplifier les preuves d'équivalence, a été (i) de définir un système de transitions étiquetées et la bisimulation associée; (ii) de prouver que cette bisimulation était incluse dans la congruence à barbes; et (iii) de prouver, avec cette bisimulation, les propriétés qui nous intéressait. Nous avons ainsi prouvé une version étendue du théorème de réplication. La méthode employée pour prouver ce résultat est elle aussi intéressante. En effet l'utilisation d'une bisimulation «spécialisée» pour les définitions a simplifiée nos preuves. Un autre ingrédient important a été l'utilisation de la méthode de preuve «up-to» pour prouver que $\sim_d$ est une congruence. Ce qui a fourni une méthode plus simple que d'exhiber une bisimulation différente pour chaque constructeur du calcul bleu.

On peut décrire l'approche consistant à utiliser une bisimulation étiquetée comme étant une approche classique. Ce qui est moins usuel, est la définition d'une bisimulation telle que les «transitions τ» ne correspondent pas directement avec les réductions (cf. lemme 8.4). En effet l'approche traditionnelle est de modifier – et de compliquer – la définition de la bisimulation, plutôt que le système de transitions. Un exemple de cette approche plus traditionnelle est donné par la bisimulation asynchrone de [6].

Pour conclure, je présente quelques applications possible des résultats de cette partie. Les lois de réplication pour le π-calcul ont été utilisées pour prouver la validité des interprétations du λ-calcul [121] et des langages à objets [139]. Dans la section 8.2, nous avons utilisé ces mêmes lois pour prouver notre interprétation du λ-calcul dans le calcul bleu était correcte.

Un résultat similaire est prouvé dans la partie IV, dans laquelle le calcul source considéré est un calcul d'objets fonctionnel [3]. Dans cette partie, nous définissons un ensemble de constructeurs dérivés pour modéliser les objets. En particulier, nous définissons un processus noté $(\mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i) P_i{}^{i\in I}\,]\, \mathbf{in}\ P)$, qui représente un objet de nom e, tel que l'invocation de la méthode l_j déclenche l'exécution de la méthode P_j, où le paramètre x_j est lié à e:

$$\mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i) P_i{}^{i\in I}\,]\, \mathbf{in}\ \mathbf{E}[e \Leftarrow l_j] \ \stackrel{*}{\rightarrow}\ \mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i) P_i{}^{i\in I}\,]\, \mathbf{in}\ \mathbf{E}[P_j\{{}^{e}/_{x_j}\}]$$

Les lois de réplication permettent de prouver certaines lois sur les objets, comme par exemple,

avec certaines hypothèses sur les occurrences de e dans P et Q:

$$\begin{cases} \mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in I}\,]\, \mathbf{in}\ (\mathbf{clone}(e)) & \approx_b & \mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in I}\,]\, \mathbf{in}\ e \\ \mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in I}\,]\, \mathbf{in}\ (P \mid Q) & \approx_b & (\mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in I}\,]\, \mathbf{in}\ P) \mid \\ & & (\mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in I}\,]\, \mathbf{in}\ Q) \end{cases}$$

Une autre application du théorème de réplication, qui n'est pas présenté dans cette thèse, est dans la définition d'une version distribué du calcul bleu [37]. Ce calcul étend $\boldsymbol{\pi}^\star$ par l'ajout de localités $[s :: P]$, et d'un opérateur de migration explicite: **go** $s.Q$, dans la syntaxe. Le terme $[s :: P]$ représente le processus P, situé dans le site s, tandis que **go** $s.Q$ est utilisé pour lancer l'exécution du processus Q sur le site s. Dans ce calcul, nous considérons une règle d'équivalence particulière ($\star$):

$$(\star) \qquad \mathbf{def}\, p = R\, \mathbf{in}\, (P \mid Q) \equiv (\mathbf{def}\, p = R\, \mathbf{in}\, P) \mid (\mathbf{def}\, p = R\, \mathbf{in}\, Q)$$

pour distribuer les définitions sur la composition parallèle. Ce choix permet de simplifier la définition de la relation de réduction. En effet on peut utiliser cette règle pour dériver la réduction suivante.

$$\begin{array}{rcl} \left(\begin{array}{l} [s_1 :: \mathbf{def}\, p = R\, \mathbf{in}\, (\mathbf{go}\ s_2.p \mid P)] \mid \\ [s_2 :: Q] \end{array} \right) & \equiv & \left(\begin{array}{l} [s_1 :: (\mathbf{def}\, p = R\, \mathbf{in}\, P)] \mid \\ \mathbf{go}\ s_2.(\mathbf{def}\, p = R\, \mathbf{in}\, p) \mid [s_2 :: Q] \end{array} \right) \\ & \rightarrow & \left(\begin{array}{l} [s_1 :: \mathbf{def}\, p = R\, \mathbf{in}\, P] \mid \\ [s_2 :: (\mathbf{def}\, p = R\, \mathbf{in}\, p) \mid Q] \end{array} \right) \end{array}$$

Ainsi, la règle ($\star$) permet de considérer les définitions comme des «briques de bases» des applications distribuées, que l'on peut copier sans danger au cours d'une communication distante. Une autre interprétation est que, si on met de coté le problème du coût des transmissions, une communication RPC est équivalente à envoyer le code de la fonction appelé sur le site du client. C'est cette intuition que l'on trouve au coeur du paradigme de «*l'évaluation distante*» [124].

Troisième partie

Types

CHAPITRE 11

Système de types simples

> Imaginez que, tout les jeudi, vos chaussures explosent si vous les lacez de la manière habituelle. Cela nous arrive tout les jours avec les ordinateurs, et personnes ne pense à se plaindre.
>
> – *Jeff Raskin*, interviewé par le magazine *Doctor Dobb's*

La grande majorité des langages de programmation de haut niveau possèdent un mécanisme de typage. L'utilité du typage est de fournir un outil simple de *vérification statique* de la cohérence des programmes, où statique signifie que la vérification est faite à la compilation, et donc avant le lancement du programme. Ainsi, pour tout programme correctement typé, on peut certifier certaines propriétés de son exécution.

La propriété qui nous intéresse ici est l'absence d'erreurs d'exécution, comme par exemple l'addition entre un entier et une chaîne de caractères, ou l'appel d'une méthode inconnue d'un objet. Un système de types qui assure cette propriété est dit *correct*, et un langage de programmation possédant un système de types avec cette propriété est dit *fortement typé*. C'est le cas des langages fonctionnels dans la lignée de ML par exemple, et c'est d'ailleurs un de leurs points forts.

Dans cette thèse, nous cherchons à développer un parallèle entre les langages fonctionnel et le calcul bleu, il est donc naturel de s'intéresser aux types que nous pouvons donner aux processus. Ainsi, dans les chapitres suivants, nous définissons quatre systèmes de types pour le calcul bleu. Ces systèmes sont obtenus par extension successives d'un système de types simples, noté $\mathtt{B}$, qui est introduit à la section 11.2. Le système $\mathtt{B}$ est comparable au système de types simples du λ-calcul défini par Curry. Les systèmes de types que nous étudions sont respectivement:

- un système avec sous-typage, $\mathtt{B}_{\leqslant}$, défini dans le chapitre 12;
- un système avec *polymorphisme paramétrique*, $\mathtt{B}_{\forall}$, comparable au système de types de Hindley-Milner pour ML, qui est introduit au chapitre 13;
- un système avec *polymorphisme contraint*, $\mathtt{BF}_{\leqslant}$, défini dans le chapitre 14.

Chacun de ces systèmes de types partage la propriété de *conservation du typage*, appelée aussi *subject reduction* en anglais, qui énonce que le type d'un terme est conservé après une réduction. Un énoncé formel de cette propriété est donné dans la proposition 12.4, au chapitre suivant. Dans la section 14.4 nous montrons qu'une erreur d'exécution correspond à un terme qui n'est pas typable dans le système $\mathtt{BF}_{\leqslant}$, qui est le complexe des systèmes étudiés ici. La propriété de conservation du typage implique donc la correction des systèmes de types présentés ici. Comme la preuve de conservation du typage a grossièrement la même structure pour chacun des systèmes

étudiés, nous avons choisi de ne fournir qu'une seule preuve, celle pour le système $\mathtt{BF}_{\leqslant}$. Cette preuve est donnée au chapitre 15. Le lecteur peut aussi trouver une preuve de la propriété de conservation du typage pour le système `B` dans [22], et pour le système $\mathtt{B}_{\forall}$ dans [36, théorème 4.1]. Cependant, il faut noter que dans les deux cas, le calcul étudié ne possède pas d'enregistrements.

Dans cette thèse, c'est aussi l'étude des objets qui a motivé la définition du système $\mathtt{BF}_{\leqslant}$. La partie IV est entièrement consacrée à ce problème. Ainsi, l'auteur partage le point de vue de J. Eifrig [44] quand il écrit: «*nous pensons que les caractéristiques de base nécessaires à la modélisation de la programmation orientée objets avec inférence de type comportent une notion de sous-typage [28], et une notion de polymorphisme récursivement contraint*», c'est-à-dire une généralisation du polymorphisme, telle que donné dans le chapitre 14. Il faut toutefois noter l'approche originale choisie par D. Rémy pour le typage du langage OCaml [83, 110], qui démontre qu'on peut ajouter une couche objets à un langage de programmation fortement typé, basé sur le polymorphisme à la ML.

11.1 Conventions

Dans notre présentation, un système de types est défini par un ensemble de règles de la forme:

$$\frac{\text{prémisse}_1 \quad \ldots \quad \text{prémisse}_n}{\text{résultat}} \text{ (nom de la règle)}$$

C'est, par exemple, la présentation utilisée par R. Milner [93] pour définir le système de types du langage ML. À la différence de certaines autres présentations de règles d'inférence, nous choisissons d'écrire les conditions auxiliaires des règles de typage avec les prémisses. Les jugements du système de types sont notés $\Gamma \vdash P : \tau$, où Γ est un environnement qui contient des associations entre noms et types, ainsi qu'entre *variables de types* et *sortes*. Dans le séquent $\Gamma \vdash P : \tau$, la lettre P désigne un terme du calcul bleu, et τ est un type. On adopte une présentation du système «à la Curry», c'est-à-dire qu'il n'y a aucune annotation ni indication de types dans la syntaxe des termes. On considère aussi d'autres jugements: le séquent $\Gamma \vdash *$ pour signifier que l'environnement Γ est bien formé; le séquent $\Gamma \vdash \tau :: \kappa$, pour signifier que le type τ à la sorte κ. On suppose que les notions de variables libres et de substitution sont étendues aux environnements et aux types.

11.2 Système de types simples

On présente dans cette section, un système de types simples pour le calcul bleu, que l'on dénote `B`. Ce système est essentiellement celui donné dans [22], augmenté pour tenir compte des records.

On admet l'existence d'un ensemble dénombrables $\mathcal{T}$, de variables de types. Les éléments de $\mathcal{T}$ sont désignés par les symboles α,β,γ,... La syntaxe des expressions de types est donnée dans la définition suivante. Notez que l'on ne définit pas de types de base, comme `int`,`bool`,... par exemple. Cependant, il existe une constante `o`, qui représente le «type des processus».

Définition 11.1 (Expressions de type) Les types sont générés par la grammaire suivante. Comme pour la syntaxe des termes, on prend comme convention d'écrire $[\, l_1 : \tau_1, \ldots, l_n : \tau_n \,]$ plutôt

que $[\,[\,[\,]\,, l_1 : \tau_n\,]\,, \ldots l_n : \tau_n\,]$, lorsque les champs $(l_i)_{i\in[1..n]}$ sont distincts.

τ,ϑ,ϱ	::=	α,β,γ	variables de type
	|	$\mathtt{o}$	type des processus
	|	$(\tau \to \vartheta)$	type fonctionnel
	|	$(\mu\alpha.\tau)$	type récursif
	|	$[\,]$	type enregistrement vide
	|	$[\,\varrho\,, l : \tau\,]$	extension/modification

Dans la suite, nous considérons que tous les types sont correctement formés, c'est-à-dire qu'ils sont conformes à un système de sortes. Cette notion est formellement définie par le jugement $\Gamma \vdash \tau :: \kappa$, dont les règles sont données dans la figure 11.1. On considère ici deux sortes de bases: $\mathbb{R}$, la sorte des enregistrements; et $\mathbb{T}$, la sorte des types. On utilise aussi le terme de *rangée* pour désigner les types enregistrements.

Définition 11.2 (Sortes) $\kappa,\chi ::= \mathbb{R} \mid \mathbb{T}$

Dans notre système, la rangée vide $[\,]$ a la sorte $\mathbb{R}$, tandis que $\mathtt{o}$ a la sorte $\mathbb{T}$. Intuitivement, le système de sortes est utilisé pour imposer la contrainte que, dans une extension $[\,\varrho\,, l : \tau\,]$, le type ϱ correspond à une rangée. Ainsi, des types tel que $[\,(\tau \to \vartheta)\,, \varrho\,]$ sont rejetés.

Remarque L'utilisation des sortes dans les systèmes de types définis dans les trois premiers chapitres de cette partie peut sembler inutilement compliquée. En effet le même résultat peut être obtenu en considérant une catégorie syntaxique distincte pour les rangées. Cependant nous aurons besoin des sortes pour définir le système du chapitre 14. En introduisant cette notion dès maintenant, nous pouvons conserver une présentation uniforme des systèmes de types introduit dans ce manuscrit. ■

Afin d'éviter la définition de types «mal formés», nous avons imposé une discipline sur la création des types. De la même manière, nous considérons une discipline particulière sur la création des environnements de typage.

Définition 11.3 (Environnements de typage) Un environnement est une expression engendrée par la grammaire suivante.

$$\Gamma,\Delta ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, \alpha :: \kappa$$

Dans ce chapitre, on ne considérera que des variables de sorte $\mathbb{T}$. On note $\Gamma_{|u}$ l'environnement obtenu à partir de Γ, en effaçant toute association sur le nom u, et on note $\mathbf{dom}(\Gamma)$ le domaine de Γ, c'est-à-dire l'ensemble de variables récursivement définit par les relations: $\mathbf{dom}(\emptyset) = \emptyset$, et $\mathbf{dom}(\Gamma, x : \tau) = \mathbf{dom}(\Gamma) \cup \{x\}$, et $\mathbf{dom}(\Gamma, \alpha :: \kappa) = \mathbf{dom}(\Gamma)$.

Ainsi le jugement $\Gamma \vdash *$, défini dans la figure 11.1, signifie que Γ associe de manière unique un type à chaque nom.

Les règles du système $\mathtt{B}$ sont données dans la figure 11.2. Dans ce système, on suppose que les types sont donnés modulo α-conversion, autrement on ne pourrait pas typer des termes tel que $(\boldsymbol{\lambda}x)(\boldsymbol{\lambda}x)P$ par exemple. On suppose également que les types récursifs sont définis «modulo μ-conversion», c'est-à-dire qu'on suppose l'existence d'une relation d'égalité entre les types, noté $\sim$, telle que $\mu\alpha.\tau \sim \tau\{\mu\alpha.\tau/\alpha\}$. Cette relation est formalisée dans le système avec sous-typage,

$$\emptyset \vdash *$$

$$\frac{\Gamma \vdash * \quad \Gamma \vdash \tau :: \mathbb{T} \quad x \notin \mathbf{dom}(\Gamma)}{\Gamma,x : \tau \vdash *} \qquad \frac{\Gamma \vdash * \quad \alpha \notin \mathbf{fn}(\Gamma)}{\Gamma,\alpha :: \kappa \vdash *}$$

$$\frac{\Gamma \vdash \tau :: \mathbb{R}}{\Gamma \vdash \tau :: \mathbb{T}} \qquad \frac{\Gamma \vdash *}{\Gamma \vdash [\,] :: \mathbb{R}} \qquad \frac{\Gamma \vdash *}{\Gamma \vdash \mathtt{o} :: \mathbb{T}} \qquad \frac{\Gamma \vdash * \quad (\alpha :: \kappa) \in \Gamma}{\Gamma \vdash \alpha :: \kappa}$$

$$\frac{\Gamma,\alpha :: \mathbb{T} \vdash \tau :: \mathbb{T}}{\Gamma \vdash \mu\alpha.\tau :: \mathbb{T}} \qquad \frac{\Gamma \vdash \tau :: \mathbb{T} \quad \Gamma \vdash \vartheta :: \mathbb{T}}{\Gamma \vdash (\tau \to \vartheta) :: \mathbb{T}} \qquad \frac{\Gamma \vdash \varrho :: \mathbb{R} \quad \Gamma \vdash \tau :: \mathbb{T}}{\Gamma \vdash [\,\varrho\,, l : \tau\,] :: \mathbb{R}}$$

Fig. 11.1: Environnement bien formés, et système de sortes pour les types simples

$$\frac{\Gamma \vdash * \quad (u : \tau) \in \Gamma}{\Gamma \vdash u : \tau}\ \text{(type ax)} \qquad \frac{\Gamma,x : \tau \vdash P : \vartheta}{\Gamma \vdash (\boldsymbol{\lambda} x)P : \tau \to \vartheta}\ \text{(type abs)}$$

$$\frac{\Gamma \vdash P : \tau \to \vartheta \quad \Gamma \vdash u : \tau}{\Gamma \vdash (Pu) : \vartheta}\ \text{(type app)} \qquad \frac{\Gamma \vdash P : [\,l_i : {\tau_i}^{i \in I}\,] \quad j \in I}{\Gamma \vdash (P \cdot l_j) : \tau_j}\ \text{(type sel)}$$

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

$$\frac{\Gamma,u : \tau \vdash P : \vartheta}{\Gamma \vdash (\boldsymbol{\nu} u)P : \vartheta}\ \text{(type new)} \qquad \frac{\Gamma \vdash P : \mathtt{o} \quad \Gamma \vdash Q : \tau}{\Gamma \vdash (P \mid Q) : \tau}\ \text{(type par)}$$

$$\frac{\Gamma \vdash P : \tau \quad (u : \tau) \in \Gamma}{\Gamma \vdash \langle u \Leftarrow P \rangle : \mathtt{o}}\ \text{(type decl)} \qquad \frac{\Gamma \vdash P : \tau \quad (u : \tau) \in \Gamma}{\Gamma \vdash \langle u = P \rangle : \mathtt{o}}\ \text{(type mdecl)}$$

Fig. 11.2: Système de types simples: B

avec les règles (sub rec fold) et (sub rec unfold) de la figure 12.1. On note $\Gamma \vdash P : \tau$ [B] les séquents inférés en utilisant les règles du système B. Une règle de typage est dite admissible dans B, si toute preuve utilisant cette règle peut être transformée en preuve qui ne l'utilise pas. La *règle d'affaiblissement*, par exemple, est admissible dans B.

Lemme 11.1 (Affaiblissement) *Si on a un jugement $\Gamma \vdash P : \tau$ [B], alors on peut affaiblir les hypothèses utilisées pour typer P. C'est-à-dire que la règle (type weak), ci-dessous, est admissible.*

$$\frac{\Gamma \vdash P : \tau \qquad \Gamma,\Gamma' \vdash *}{\Gamma,\Gamma' \vdash P : \tau} \text{ (type weak)}$$

Preuve Par induction sur la structure de $\Gamma \vdash P : \tau$. □

Cette règle sera explicitement insérée dans les systèmes de type introduit plus loin.

On remarque que le système B est *dirigé par la syntaxe*. C'est-à-dire qu'on peut faire correspondre chaque règle de typage à une unique construction de $\boldsymbol{\pi}^\star$. On remarque aussi que les règles de typage pour les agents, c'est-à-dire pour la partie fonctionnelle de $\boldsymbol{\pi}^\star$, sont celles du système de types simples du λ-calcul. Les règles de typage pour les processus sont moins usuels.

La règle pour la composition parallèle s'interprète ainsi: dans un processus $(P \mid Q)$, il y a un «thread» d'exécution principal, Q, qui peut-être une fonction. En particulier Q peut avoir un type fonctionnel $(\tau \to \vartheta)$. Le terme P représente alors l'environnement de Q, c'est-à-dire une soupe de processus de type o qui peuvent se réduire en déclarations ou en messages.

Les règles concernant les déclarations: (type decl) et (type mdecl), méritent plus de commentaires. Un séquent $\Gamma,u : \tau,\Gamma' \vdash P : \vartheta$, signifie que si le nom u est utilisé dans P, alors il l'est avec le type τ. Par conséquent, si u apparaît en sujet d'une déclaration $\langle u \Leftarrow Q\rangle$ dans le terme $(\langle u \Leftarrow Q\rangle \mid P)$, on doit pouvoir typer Q avec le type τ. C'est en gros le même raisonnement que pour le typage de l'opérateur **let** de ML– mis à part le polymorphisme –, c'est-à-dire qu'on doit typer une ressource avec le type du nom qui permet d'y accéder. De plus, la ressource doit être «silencieuse», dans le sens où elle doit pouvoir être composée avec n'importe quel terme (P dans notre cas) sans que son type interfère avec le type du résultat. L'intuition, ici, est que la présence d'une ressource doit être invisible tant qu'elle n'est pas activée. On donne alors le type processus o aux déclarations. Il faut noter que ce choix implique que, dans un terme bien typé, une déclaration n'est jamais appliquée à un nom. Le jugement suivant, qui illustre notre discussion, est valide dans le système B.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash P : \tau} \quad (u : \tau) \in \Gamma}{\Gamma \vdash \langle u \Leftarrow P\rangle : \mathtt{o}} \qquad \dfrac{\vdots}{\Gamma \vdash Q : \vartheta}}{\Gamma \vdash (\langle u \Leftarrow P\rangle \mid Q) : \vartheta} \tag{11.1}$$

et de même pour $(\langle u = Q\rangle \mid P)$. On peut donc fournir une règle de typage admissible, (type rec), pour l'opérateur de récursion $\mathbf{rec}\, u.P$. En utilisant les règles (type mdecl) et (type new), il est aussi possible de montrer que le processus $\mathbf{0}$, c'est-à-dire $(\boldsymbol{\nu} u)\langle u = u\rangle$, a le type des processus.

$$\frac{\Gamma,u : \tau \vdash P : \tau}{\Gamma \vdash \mathbf{rec}\, u.P : \tau} \text{ (type rec)} \qquad \frac{\Gamma,u : \tau \vdash P : \tau \qquad \Gamma,u : \tau \vdash Q : \sigma}{\Gamma \vdash \mathbf{def}\, u = P \,\mathbf{in}\, Q : \sigma} \text{ (type def)}$$

$$\frac{}{\Gamma \vdash (\boldsymbol{\nu} u)\langle u = u\rangle : \mathtt{o}} \text{ (type nil)}$$

Ces dernières règles ne sont pas dénuées d'intérêt. En effet, on a vu dans la première partie (cf. section 3.2), qu'il est possible de remplacer l'opérateur de déclaration répliquée $\langle u = P\rangle$, par

l'opérateur de récursion ou de définition, ceci sans modifier l'expressivité du calcul bleu. Nous rappelons ici le codage:

$$\langle u = P \rangle \;\simeq\; \mathbf{def}\, x = \langle u \Leftarrow (x \mid P) \rangle \,\mathbf{in}\, x \;\simeq\; \mathbf{rec}\, x.\langle u \Leftarrow (x \mid P) \rangle$$

Les règles de typage (type rec) et (type def) impliquent quand à elles, que ce remplacement ne modifie rien au niveau du typage. En effet on peut typer ces trois processus de la même manière. Il suffit de choisir x de type $\mathtt{o}$. On peut donc conclure qu'il n'y a pas, pour le moment, de choix nécessaire à faire entre ces trois constructeurs. Cependant, nous verrons dans le chapitre 13, que l'opérateur de définition est plus avantageux.

11.3 Relation avec le système de sortes du pi-calcul

Nous considérons dans cette section le système de sortes définit par R. Milner dans [97], que nous dénotons $\mathtt{M}$. Il s'agit d'un système simple, sans récursion, étudié par exemple par S. Gay, V. Vasconcelos et K. Honda [58, 133, 69, 134], ainsi que dans la thèse de D. Turner [130]. La présentation choisie pour ce système a été modifiée pour s'accommoder des notations déjà introduites. Les *sortes*: $\delta_1, \delta_2, \ldots$ à ne pas confondre avec leur homonymes de la section précédente, sont engendrées par la grammaire suivante.

Définition 11.4 (Sortes du π-calcul)

$$\begin{array}{rcll} \delta & ::= & \alpha & \text{variable de type} \\ & \mid & \updownarrow[\delta_1,\ldots,\delta_n] & \text{sorte des canaux} \end{array}$$

La sorte $\updownarrow[\delta_1,\ldots,\delta_n]$, est associée aux canaux qui transportent des n-uplets de noms $(v_1,\ldots,v_n)$, de sortes respectives $\delta_1,\ldots,\delta_n$. Le système $\mathtt{M}$ est défini dans la figure 11.3.

$$\frac{(u :\updownarrow[\tilde{\delta}]) \in \Delta \qquad \forall i \in [1..n], (v_i : \delta_i) \in \Delta}{\Delta \vdash \bar{u}\langle \tilde{v} \rangle : \mathtt{o}}$$

$$\frac{\Delta, \tilde{v} : \tilde{\delta} \vdash P : \mathtt{o} \qquad (u :\updownarrow[\tilde{\delta}]) \in \Delta}{\Delta \vdash u(\tilde{v}).P : \mathtt{o}} \qquad \frac{\Delta, \tilde{v} : \tilde{\delta} \vdash P : \mathtt{o} \qquad (u :\updownarrow[\tilde{\delta}]) \in \Delta}{\Delta \vdash !u(\tilde{v}).P : \mathtt{o}}$$

$$\frac{\Delta \vdash P : \mathtt{o} \qquad \Delta \vdash Q : \mathtt{o}}{\Delta \vdash (P \mid Q) : \mathtt{o}} \qquad \frac{\Delta, u : \delta \vdash P : \mathtt{o}}{\Delta \vdash (\boldsymbol{\nu} u) P : \mathtt{o}}$$

Fig. 11.3: Système de sortes simples pour $\boldsymbol{\pi} : \mathtt{M}$

Dans un jugement $\Delta \vdash P : \mathtt{o}$ $[\mathtt{M}]$, l'environnement Δ est une fonction partielle des noms vers les sortes et P est un processus de $\boldsymbol{\pi}$. On définit une interprétation des sortes dans les types de $\mathtt{B}$ de la manière suivante.

Définition 11.5 (Codage des sortes de π)

$$\begin{array}{rcl} [\![\alpha]\!] & = & \alpha \\ [\![\updownarrow[]]\!] & = & \mathtt{o} \\ [\![\updownarrow[\delta_1,\ldots,\delta_n]]\!] & = & ([\![\delta_1]\!] \to \ldots ([\![\delta_n]\!] \to \mathtt{o}) \ldots) \end{array}$$

On montre alors qu'on peut associer un jugement de B à chaque jugement de M. Dans la propriété suivante, l'interprétation d'un environnement $[\![\Delta]\!]$, est définie de manière évidente.

Théorème 11.2 *Si* $\Delta \vdash P : \mathtt{o}$ [M]*, alors* $[\![\Delta]\!] \vdash [\![P]\!] : \mathtt{o}$ [B].

Preuve Une simple induction sur la structure du séquent $\Delta \vdash P : \mathtt{o}$. Il faut noter que le choix du parallèle dissymétrique n'intervient à aucun moment dans la preuve. □

Par exemple, on associe à la sorte: $\updownarrow[\updownarrow[\,],\updownarrow[\,]]$, qui est la sorte des «canaux booléens» dans $\boldsymbol{\pi}$, le type $(\mathtt{o} \to \mathtt{o} \to \mathtt{o})$, qui est le type de T et F, cf. section 3.5.

On remarque que le type donné aux processus est toujours o. Ce qui implique que les informations données par le typage, sont toutes entières contenues dans l'environnement. Ce fait est déjà connu, en effet on aurait pu se contenter de noter le séquent précédent: $\Delta \vdash P$, comme le fait D. Turner dans sa thèse par exemple. Ceci nous montre que, en utilisant notre codage de $\boldsymbol{\pi}$ dans $\boldsymbol{\pi}^\star$, on peut typer les π-termes de manière plus générale que dans le système M. Par exemple, on peut définir un agent récursif dans le π-calcul, $\mathbf{rec}\, r.(\boldsymbol{\lambda} x)P$, par le processus $(\boldsymbol{\nu} r)(!r(x).P \mid \bar{r}\langle\rangle)$. Ce processus n'est pas bien «typé dans M»: l'émission sur le nom r est vide alors que la réception attend un argument, alors que dans $\boldsymbol{\pi}^\star$, il a le type de l'agent $(\boldsymbol{\lambda} x)P$. D'autres exemples de ce phénomène sont donnés dans [22].

11.4 Remarque sur le typage des processus

Avant de nous intéresser aux extensions possibles du système de types présenté dans ce chapitre, nous pouvons ajouter quelques remarques d'ordre général.

L'étude des systèmes de types pour les calculs de processus, particulièrement les calculs dérivé de $\boldsymbol{\pi}$, a suscité beaucoup d'attention ces dernières années. Ainsi, on a cherché a développer des système de types permettant de vérifier d'autres propriétés que la correction, comme par exemple l'absence de deadlocks [126, 21], l'existence d'un receveur unique [8], ou «uniformément disponible» [120] ... On peut également penser à des systèmes permettant la détection de «trous de sécurité», dans le style de [138, 5]. Ces problèmes ne font pas partie des axes de recherche exposés ici, et nous n'aborderons pas plus avant ces questions dans ce manuscrit.

Une autre généralisation possible est d'annoter le type des noms par une information sur la directionalité des communications. On parle aussi de *polarité*. C'est ce qui est fait pour le π-calcul dans [107]. On exprime ainsi qu'un canal peut être utilisé pour émettre: il a la sorte $\uparrow[\delta]$ dans $\boldsymbol{\pi}$; pour recevoir: il a la sorte $\downarrow[\delta]$; ou bien pour faire les deux: il a alors la sorte $\updownarrow[\delta]$. On peut ainsi exprimer qu'un canal de communication, reçoit des canaux sur lequel il ne peut qu'émettre: il suffit de lui donner la sorte $\downarrow[\uparrow[\,]]$. Cette approche peut être adaptée au calcul bleu: on dira qu'une référence possède la capacité de réception, si elle a le type $\downarrow(\tau \to \vartheta)$. Cependant ceci ne permet pas de raffiner notre système, car l'hypothèse de localité impose une contrainte très forte sur les types. C'est-à-dire que tout les noms liés par une λ-abstraction doivent avoir le type $\uparrow(\tau \to \vartheta)$.

On peut noter une autre utilisation des systèmes de types, plus spécifique aux calculs de processus. Le π-calcul a souvent joué le rôle de calcul cible, pour donner l'interprétation de langages de programmation, ou des interprétations d'autres calculs. C'est ce que nous avons fait dans la section 3.3, et c'est ce que nous ferons dans la partie IV. Lorsqu'on cherche à prouver des propriétés sur ces codages, comme par exemple la propriété de «full abstraction», on tombe rapidement sur des problèmes. En effet le π-calcul est très expressif, il est donc possible de coder des contextes modélisant des comportements qui ne sont pas toujours exprimable dans le calcul source. Ce problème apparaît, par exemple, dans le codage du λ-calcul dans le π-calcul. Si on ajoute un système de types au π-calcul, et si on demande à l'environnement de respecter le typage des

processus[1], on peut alors démontrer des propriétés qui sont fausses dans un modèle non typé. Intuitivement, le système de types va permettre d'établir un contrat entre un processus et son environnement. Ce contrat empêche alors un processus de mal se comporter.

Dans cette thèse, nous n'utiliserons pas notre système de types à cette fin. Cependant, on pourrait interpréter notre restriction sur la localité des déclarations, comme une contrainte de typage, et pas comme une contrainte syntaxique. Pour interdire l'abstraction des références, il suffit par exemple d'utiliser des indications sur la polarité des communications, comme dans [107]. C'est d'ailleurs ce que fait D. Sangiorgi dans [118]. Le choix d'étudier la version locale de $\boldsymbol{\pi}^\star$, nous permet donc de simplifier la présentation de nos résultats.

1. Plus précisément l'environnement doit respecter le type des références.

CHAPITRE 12

Sous-typage

Aucun système de types n'est assez fin pour typer l'ensemble des programmes s'exécutant sans erreurs, et seulement ceux-ci. Ainsi, il existe des termes corrects qui ne sont pas typables. On peut donner un exemple assez simple de ce phénomène dans notre système.

Supposons que l'on dispose de la ressource $\langle \mathsf{sel}_l = (\boldsymbol{\lambda} x)(x \cdot l) \rangle$, qui est la fonction permettant de sélectionner le champ l d'un enregistrement. Le type de la référence sel_l est donc de la forme $([\, \varrho\, , l : \mathtt{o}\,] \to \mathtt{o})$, où ϱ est n'importe quel type de sorte $\mathbb{R}$. Supposons aussi disposer de deux ressources $\langle r_1 = [\, m = \mathbf{0}, l = \mathbf{0}\,] \rangle$ et $\langle r_2 = [\, k = \mathbf{0}, l = \mathbf{0}\,] \rangle$, en parallèle avec les autres termes. Il est facile de montrer que le terme $P =_{\text{def}} (\mathsf{sel}_l r_1 \mid \mathsf{sel}_l r_2)$ n'est pas typable dans notre système. L'argument est qu'il faut répondre aux deux contraintes $\varrho = [\, m : \mathtt{o}, l : \mathtt{o}\,]$ et $\varrho = [\, k : \mathtt{o}, l : \mathtt{o}\,]$, qui correspondent aux types de r_1 et r_2. Cependant P et ses dérivés ne comportent aucune erreur.

L'introduction du *sous-typage*, que l'on trouve formalisé dans les travaux de J. Mitchell [99] par exemple, permet d'augmenter le nombre des termes corrects que l'on sait typer. L'idée est de considérer qu'un enregistrement peut être utilisé là ou un enregistrement contenant moins de champs peut l'être: «qui peut le plus, peut le moins». On se donne une relation d'ordre entre les types: $\Gamma \vdash \tau \leqslant \vartheta$, pour dire que τ peut se substituer à ϑ. On dit que τ est un sous-type de ϑ. Par la suite, nous pourrons omettre de noter l'environnement, et écrire simplement $\tau \leqslant \vartheta$. On ajoute alors une règle supplémentaire, (type sub), au système B.

Définition 12.1 (Sous-typage) Soit $\leqslant$ la relation définie dans la figure 12.1. Si un processus P à le type τ et si τ est un sous-type de ϑ, alors P à le type ϑ

$$\frac{\Gamma \vdash P : \tau \qquad \Gamma \vdash \tau \leqslant \vartheta}{\Gamma \vdash P : \vartheta} \text{ (type sub)}$$

Dans notre exemple, on peut alors typer les références r_1 et r_2 avec le type moins précis $[\, l : \mathtt{o}\,]$, et typer P avec le type $\mathtt{o}$. On note $\mathtt{B}_{\leqslant}$ le système B augmenté de la règle de sous-typage.

La relation de sous-typage est donnée dans la figure 12.1, on omet cependant de donner la règle qui implique que $\leqslant$ est une relation réflexive. La partie concernant les types fonctionnel est usuelle. En particulier $\tau \to \vartheta$ est *contravariant* selon le premier paramètre et *covariant* selon le second.

$$\frac{\Gamma \vdash [\,\varrho\,, l : \tau\,] :: \mathbb{R}}{[\,\varrho\,, l : \tau\,] \leqslant [\,]}\ \text{(sub void)} \qquad \frac{\Gamma \vdash \tau_1 \leqslant \tau_2 \quad \Gamma \vdash \tau_2 \leqslant \tau_3}{\Gamma \vdash \tau_1 \leqslant \tau_3}\ \text{(sub trans)}$$

$$\frac{\Gamma,\alpha :: \mathbb{T} \vdash \tau :: \mathbb{T} \quad \Gamma,\beta :: \mathbb{T} \vdash \vartheta :: \mathbb{T} \quad \alpha \leqslant \beta \Rightarrow \Gamma \vdash \tau \leqslant \vartheta}{\Gamma \vdash \mu\alpha.\tau \leqslant \mu\beta.\vartheta}\ \text{(sub rec)}$$

$$\frac{\Gamma \vdash \mu\alpha.\tau :: \mathbb{T}}{\Gamma \vdash \tau\{\mu\alpha.\tau/\alpha\} \leqslant \mu\alpha.\tau}\ \text{(sub rec fold)} \qquad \frac{\Gamma \vdash \mu\alpha.\tau :: \mathbb{T}}{\Gamma \vdash \mu\alpha.\tau \leqslant \tau\{\mu\alpha.\tau/\alpha\}}\ \text{(sub rec unfold)}$$

$$\frac{\Gamma \vdash [\,[\,\varrho\,, k : \vartheta\,], l : \tau\,] :: \mathbb{R} \quad k \neq l}{[\,[\,\varrho\,, k : \vartheta\,], l : \tau\,] \leqslant [\,[\,\varrho\,, l : \tau\,], k : \vartheta\,]}\ \text{(sub swap)}$$

$$\frac{\Gamma \vdash \vartheta_1 \leqslant \tau_1 \quad \Gamma \vdash \tau_2 \leqslant \vartheta_2}{\Gamma \vdash \tau_1 \to \tau_2 \leqslant \vartheta_1 \to \vartheta_2}\ \text{(sub arrow)} \qquad \frac{\Gamma \vdash \varrho \leqslant \varrho' \quad \Gamma \vdash \tau \leqslant \tau'}{\Gamma \vdash [\,\varrho\,, l : \tau\,] \leqslant [\,\varrho'\,, l : \tau'\,]}\ \text{(sub over)}$$

Fig. 12.1: Sous-typage

Définition 12.2 (Covariance, contravariance et invariance) Soit $\sim$ la relation d'équivalence telle que $\tau \sim \vartheta$, si et seulement si $\tau \leqslant \vartheta$ et $\vartheta \leqslant \tau$. On dit qu'un opérateur de type typop(.), est *covariant* si typop(τ) $\leqslant$ typop(ϑ) dès que $\tau \leqslant \vartheta$. On dit qu'il est *contravariant* si la même condition implique typop(ϑ) $\leqslant$ typop(τ). On dit qu'il est *invariant* si typop(τ) $\leqslant$ typop(ϑ) implique que $\tau \sim \vartheta$.

Ces notions de variance s'étendent à l'occurrence des variables dans un type. Ainsi, par exemple, la variable α apparaît en position *covariante* dans le type $(\alpha \to \tau) \to \tau$.

Dans la figure 12.1, les règles concernant les enregistrements sont moins usuelles: leur présentation reflète la construction incrémentale des enregistrements. En particulier la règle (sub swap) permet de s'affranchir de l'ordre dans lequel est formé un enregistrement, puisqu'elle implique que $[\ldots, l : \tau, k : \vartheta, \ldots] \sim [\ldots, k : \vartheta, l : \tau, \ldots]$. On retrouve aussi les notions de sous-typage en largeur et en profondeur.

Lemme 12.1 (Sous-typage en largeur) $[\,l_1 : \tau_1, \ldots, l_{n+k} : \tau_{n+k}\,] \leqslant [\,l_1 : \tau_1, \ldots, l_n : \tau_n\,]$

Preuve Par induction sur k. On donne le cas où $k = 1$. La propriété, dans le cas plus général, est obtenue facilement en utilisant la règle (sub over). En utilisant n fois la règle (sub swap), on obtient que:

$$[\,[\,[\,]\,, l_1 : \tau_1\,], \ldots, l_n : \tau_n\,, l_{n+1} : \tau_{n+1}\,] \sim [\,[\,[\,]\,, l_{n+1} : \tau_{n+1}\,], l_1 : \tau_1, \ldots, l_n : \tau_n\,]$$

Et en utilisant la règle (sub void) on obtient que: $[\,[\,]\,, l_{n+1} : \tau_{n+1}\,] \leqslant [\,]$. Le résultat découle donc en appliquant n fois la règle (sub over):

$$[\,[\,], l_1 : \tau_1, \ldots, l_{n+1} : \tau_{n+1}\,] \leqslant [\,[\,], l_1 : \tau_1, \ldots, l_n : \tau_n\,]$$

□

Lemme 12.2 (Sous-typage en profondeur) *Si pour tout entier i de l'intervalle $[1..n]$, on a $\tau_i \leqslant \vartheta_i$, alors* $[\,l_1 : \tau_1, \ldots, l_n : \tau_n\,] \leqslant [\,l_1 : \vartheta_1, \ldots, l_n : \vartheta_n\,]$.

Preuve Par induction sur n, on utilise uniquement la règle (sub over). □

Ces deux résultats sont des instances d'une propriété plus générale, qui caractérise les cas dans lesquels deux enregistrements, écrits sous la forme $[\, l_i : \tau_i{}^{i\in I}\,]$, sont en relation de sous-typage. Soit I une famille d'indices $i_1,\ldots,i_k$. On note $[\, l_i : \tau_i{}^{i\in I}\,]$ l'enregistrement $[\, l_{i_1} : \tau_{i_1},\ldots,l_{i_k} : \tau_{i_k}\,]$.

Lemme 12.3 (Sous-typage) $[\, l_i : \tau_i{}^{i\in I}\,] \leqslant [\, l_i : \vartheta_i{}^{i\in J}\,]$, *si et seulement si* J *est inclus dans* I *et si, pour tout* $i \in J$, *on a* $\tau_i \leqslant \vartheta_i$.

Preuve Par induction sur l'inférence de $[\, l_i : \tau_i{}^{i\in I}\,] \leqslant [\, l_i : \vartheta_i{}^{i\in J}\,]$. Cette propriété est une conséquence d'une propriété plus générale prouvée à la section 15.2.2 (cf. corollaire 15.14). En particulier, on montre que si $[\, \varrho\,, l : \tau\,] \leqslant [\, \varrho'\,, k : \vartheta\,]$ et $k \neq l$, alors $\varrho \leqslant [\, k : \vartheta\,]$. □

Avec l'ajout de la règle de sous-typage, il est possible de simplifier la règle (type sel), de typage de la sélection, comme suit. C'est cette règle qu'on considérera par la suite.

Définition 12.3 (Nouvelle règle (type sel)) Dans le système de types avec sous-typage, on peut remplacer la règle (type sel) par la règle suivante.

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

Comme nous l'avons annoncé en introduction de ce chapitre, une propriété qu'on veut voir vérifiée par un système de types est la correction. Ce résultat est généralement obtenu en deux étapes. On montre d'abord que le type est préservé au cours d'une réduction. C'est la propriété 12.4. On montre ensuite que les termes «erreurs» ne sont pas typable. Ce qui est simple dans notre cas.

Proposition 12.4 (Conservation du typage) *Si* $\Gamma \vdash P : \tau\ [\mathtt{B}_\leqslant]$ *et* $P \to P'$, *alors* $\Gamma \vdash P' : \tau\ [\mathtt{B}_\leqslant]$.

On ne donne pas la preuve de cette propriété dans cette thèse, disons simplement qu'elle est semblable (en plus simple), à la preuve donnée dans le chapitre 15.

En plus d'être correct, un système de types doit être pratique. Ainsi, une propriété qu'on peut rechercher pour un système de types est, par exemple, que le problème de l'*inférence de type* soit décidable: on veut avoir un algorithme qui fournit pour chaque terme et son environnement, son type. On discute de ce problème dans la section suivante.

12.1 Choix de la relation de sous-typage et inférence de type

La notion de sous-typage considérée ici n'est pas la seule définie dans la littérature. En particulier, certains systèmes considèrent une notion de *sous-typage atomique* [99], qui est basé sur des relations entre types de bases. On a alors une relation d'ordre $\subseteq$ entre types de bases, avec des jugements tel que `int` $\subseteq$ `float`, par exemple. Cette équation signifiant qu'un entier peut être utilisé là où on attend un flottant. Ceci correspond, en gros, à de l'*overloading*.

Une première généralisation consiste à ajouter deux constantes: $\bot$ et $\top$, qui représentent, respectivement, le plus petit et le plus grand des types. Dans ce système, on peut supposer pour type τ les relation suivantes: $\bot \subseteq \tau \subseteq \top$. Alors qu'en l'absence de sous-typage, l'inférence se réduit à la résolution d'un ensemble de contraintes d'unification[1] [94, 40]: $\tau = \vartheta$, on doit maintenant résoudre un ensemble d'inégalitées: $\tau \subseteq \vartheta$, ce qui est un problème beaucoup plus difficile. On adresse le lecteur à [109] pour un état de l'art sur le problème de la résolution (et de

1. On ne discutera pas ici du choix des relations derrière les symboles $\subseteq$ et $=$.

la simplification) de ces contraintes.

Contrairement au sous-typage dit atomique, la relation de sous-typage considérée ici est dite *structurelle*, car elle est basée sur la syntaxe des types et non pas sur une relation d'ordre annexe. En particulier, on remarque qu'il n'existe pas d'équivalent aux constantes $\bot$ et $\top$. Le problème de l'inférence de type dans ce système se réduit aussi à la résolution d'un système de contraintes [44], de la forme $\tau \leqslant \vartheta$ dans notre cas.

Comme nous l'avons dit, le problème de l'inférence de type en présence de sous-typage a été beaucoup étudié, et particulièrement en présence de sous-typage structurel. Cet intérêt est dû au rôle central joué par le sous-typage dans la définition de systèmes de typage pour les langages orientés objets. Ce problème n'a jamais été étudié dans le cadre d'un calcul de processus. Cependant, comme l'indiquent les résultats obtenus dans cette thèse, on peut conjecturer qu'il est possible d'adapter les résultats du λ-calcul typé au calcul bleu. On pourrait, par exemple, chercher à adapter les résultat de J. Eifrig, S. Smith et V. Trifonov [44] et de F. Pottier [109].

La correspondance entre $\mathtt{B}_\leqslant$ et les systèmes de types des calculs cités plus haut n'est pas totale. En effet, pour pouvoir définir un algorithme d'inférence de type, il faut vérifier la propriété de *typage principal* [39, 72]: un jugement de type $\Gamma \vdash P : \tau$ est dit principal, si pour tout jugement $\Gamma \vdash P : \vartheta$, on peut montrer que $\tau \leqslant \vartheta$. C'est-à-dire qu'il existe un type «le plus précis», représentant tous les types possibles de P dans Γ. Il n'est pas clair qu'un tel type existe dans $\mathtt{B}_\leqslant$, le problème venant de notre utilisation des types récursifs et enregistrements. En particulier, on remarque qu'il n'est pas possible d'exprimer le fait qu'un champ est absent d'un enregistrement. Cette propriété est importante si on cherche à obtenir un algorithme d'inférence de types, en particulier si on veut pouvoir ajouter le polymorphisme [111]: C'est ce qui manquait, par exemple, au système de M. Wand [140] pour avoir la propriété de type principal.

12.2 Absence d'un champ dans un enregistrement

On étudie dans cette section deux approches possibles, pour exprimer qu'un champ n'est pas dans un enregistrement.

La première approche est l'utilisation des *variables de rangées* [111]. On ajoute un nouveau symbole `abs`, pour signifier qu'un champ est absent, avec la règle de formation suivante.

$$\frac{\Gamma \vdash \varrho : \mathbb{R}}{\Gamma \vdash [\,\varrho\,, l : \mathtt{abs}\,] : \mathbb{R}}$$

Ainsi, on peut comparer le type $[\,\varrho\,, l : \mathtt{abs}\,]$ avec la fonction $\varrho \setminus l$ de [27, 86], qui restreint le champ l dans l'enregistrement ϱ. Si on utilise la règle (sub void) de la figure 12.1, on obtient alors que $[\,l : \mathtt{abs}\,] \leqslant [\,]$. Par conséquent, la règle (sub abs) implique que la privation d'un champ accroît l'information qu'on possède sur un enregistrement, ce qui est conforme à notre intuition. On ajoute alors la relation inverse, qui signifie que l'enregistrement vide est l'enregistrement tel que tous les champs sont absents.

Définition 12.4 (Variable de rangée) On considère un nouveau constructeur `abs`, pour les types, qui n'apparaît que sous la forme $[\,\varrho\,, l : \mathtt{abs}\,]$. Et on ajoute la règle de sous-typage suivante.

$$\frac{}{[\,] \leqslant [\,l : \mathtt{abs}\,]}\ \text{(sub abs)}$$

On dit alors que le champ l est absent de τ, si on peut montrer que $\tau \sim [\,\varrho\,, l : \mathtt{abs}\,]$. En particulier, il est facile de montrer que l'enregistrement vide n'a aucun champ présent, c'est-à-dire que $[\,] \sim [\,l_1 : \mathtt{abs}, \ldots, l_n : \mathtt{abs}\,]$.

On peut par exemple typer la fonction sel_l, utilisée comme exemple dans l'introduction de ce chapitre, avec le type $([\,k : \mathtt{abs}, l : \mathtt{o}\,] \to \mathtt{o})$, puisque $([\,l : \mathtt{o}\,] \to \mathtt{o}) \leqslant ([\,k : \mathtt{abs}, l : \mathtt{o}\,] \to \mathtt{o})$.

Une autre solution consiste à introduire la constante de type: $\top$, qui représente le type le plus grand. Comme nous l'avons dit, ce type est usuel lorsqu'on considère le sous-typage, ce qui est moins usuel, c'est qu'on peut l'utiliser – à la manière de F. Cardone et M. Coppo dans [30] – comme le type des erreurs. Si on considère cette utilisation de $\top$, il faut également ajouter de nouvelles règles de typage et de sous-typage. Ainsi il faut ajouter la règle (type error) ci-dessous, qui permet de typer tout les processus, même les erreurs. On ajoute aussi une règle de sous-typage pour pouvoir supposer que tout type est plus petit que $\top$, et qu'un champ de type erreur doit être «substituable» à un champ absent. Intuitivement, ceci souligne le fait qu'un champ absent n'est pas accessible.

$$\frac{\Gamma \vdash *}{\Gamma \vdash P : \top}\ \text{(type error)} \qquad \frac{}{\tau \leqslant \top}\ \text{(sub top)} \qquad \frac{}{[\,] \leqslant [\,l : \top\,]}\ \text{(sub error)}$$

On est donc dans une situation où $[\,l : \tau\,] \leqslant [\,l : \top\,] \sim [\,]$ (règles (sub over) et (sub void)). Dans ce système, le résultat de correction est légèrement différent. En effet il ne suffit pas de demander qu'un terme soit typable, mais plutôt qu'il soit d'un type différent de $\top$.

Il est facile de voir que, mis à part l'énoncé de la propriété de correction, ces deux approches sont équivalentes. D'ailleurs D. Rémy fait à peu près la même remarque dans [112], où il montre qu'on peut concilier variables de rangée et sous-typage en considérant que $[\,\varrho\,, l : \tau\,] \leqslant [\,\varrho\,, l : \mathtt{abs}\,]$. Plus précisément, il utilise la relation $[\,\varrho\,, l : \tau\,\mathtt{pre}\,] \leqslant [\,\varrho\,, l : \mathtt{abs}\,]$, où $(l : \tau\,\mathtt{pre})$ signifie que le champ l est présent dans l'enregistrement avec le type τ. En particulier, ces deux approches donnent des types similaires sur l'exemple de la fonction d'extension ext_l, qui étend tout enregistrement r avec le champ l:

$$\langle \mathsf{ext}_l = (\boldsymbol{\lambda} r x)[\,r\,, l = x\,]\rangle$$

Dans la dernière approche – avec ajout du type $\top$ –, on peut typer ext_l avec le type $[\,\alpha\,, l : \top\,] \to \beta \to [\,\alpha\,, l : \beta\,]$, où α est une variable de sorte $\mathbb{R}$, implicitement quantifiée. Ce type indique que le premier argument de ext_l peut-être un enregistrement qui contient, ou non, le champ l, sans se soucier de son type. C'est le même type qu'on donne dans l'approche avec variables de rangées, ainsi on peut typer ext_l par $[\,\alpha\,, l : \mathtt{abs}\,] \to \beta \to [\,\alpha\,, l : \beta\,]$. On pourrait aussi typer cette fonction dans le système de départ en utilisant les sortes. En effet ext_l a le type $\alpha \to \beta \to [\,\alpha\,, l : \beta\,]$ avec la contrainte que α est un enregistrement. Plus exactement, si on anticipe sur les chapitres suivants, on peut donner un type polymorphe à la fonction ext_l:

$$\forall(\alpha^{\mathbb{R}}, \beta^{\mathbb{T}}).(\alpha \to \beta \to [\,\alpha\,, l : \beta\,])$$

où l'exposant dans $\alpha^{\mathbb{R}}$, indique que la sorte de α est $\mathbb{R}$. Il faut noter que, dans les systèmes de types du chapitre 13, on considère que les variables quantifiées ont toutes la sorte $\mathbb{T}$.

Le système d'enregistrements que nous avons présenté ici regroupe des caractéristiques qu'on retrouve dans d'autres systèmes d'enregistrements fonctionnels et extensibles [111, 86, 57, 56]. On retrouve, en particulier, une présentation similaire dans les premier articles sur les calculs d'enregistrements [140, 27].

Le choix des opérateurs du «calcul d'enregistrements» que nous avons inclus dans le calcul bleu, a été motivé par l'utilisation d'un codage très simple, qui utilise les opérateurs de *matching* et *mismatching* – généralement noté $[a = b]P$ et $[a \neq b]P$ – des algèbres de processus, c'est-à-dire le test d'égalité (resp. de d'inégalité) entre les noms a et b. Il faut noter que, pour représenter des enregistrements, il est possible de restreindre ce test à une catégorie spéciale de noms, qui sont par exemple les étiquettes dans notre cas. C'est l'existence d'un tel codage, qui reste implicite dans notre présentation, qui explique pourquoi nous n'avons pas réutilisé – sur l'armoire dirait les anglophones! – un calcul d'enregistrements déjà existant.

CHAPITRE 13

Polymorphisme paramétrique

On définit dans ce chapitre, un système de types polymorphes à la ML pour le calcul bleu. Ce type de polymorphisme, appelé *polymorphisme paramétrique*, est une autre généralisation possible de `B`, qui permet de typer de nouveaux termes.

Il s'agit ici de typer les fonctions qui calculent de manière uniforme sur leurs arguments, c'est-à-dire sans prendre en compte leurs types. Un exemple est la fonction qui renverse l'ordre des éléments d'une liste. Sa définition est indépendante du type des éléments de la liste. On type cette fonction avec le *schéma de types* $\forall\alpha.(\text{list}(\alpha) \to \text{list}(\alpha))$, et on peut *spécialiser* son type pour obtenir une fonction des listes d'entiers vers les listes d'entiers: $(\text{list}(\texttt{int}) \to \text{list}(\texttt{int}))$, ou des listes de fonctions vers les listes de fonctions: $(\text{list}(\texttt{int} \to \texttt{bool}) \to \text{list}(\texttt{int} \to \texttt{bool}))$, ...

L'ajout de la quantification des types ne va pas sans poser de problèmes. Ainsi, si on ne pose aucune restriction à l'usage de la quantification, ce qui correspond au système F, le problème de l'inférence de type est indécidable [77].

On décide alors de se restreindre au polymorphisme de *premier ordre*, dans lequel la quantification ne peut apparaître qu'en tête d'un type. C'est l'hypothèse choisie dans le système de Hindley-Milner, qui est à la base du système utilisé dans le langage ML. On ne considère pas non plus les types récursifs et les types enregistrements, car leur présence complique le système, sans apporter de grande différence aux propriétés données dans cette section. Ainsi les types de la forme $(\forall\alpha.\tau) \to \vartheta$ ou $\mu\alpha.\forall\beta.(\alpha \to \beta)$, par exemple, sont rejetés.

Définition 13.1 (Schéma de types) On définit une nouvelle catégorie d'expressions de types, appelé *schéma de types*, générés par la grammaire suivante.

$$\sigma,\eta ::= \tau \mid \forall\alpha.\sigma$$

où τ est un type de `B`, appelé aussi *type simple*, et α est une variable. On pourra noter $\forall\tilde{\alpha}.\tau$, le schéma de type $(\forall\alpha_1 \ldots \forall\alpha_n.\tau)$. On utilise les symboles σ et η pour désigner les schémas de types.

On définit la sorte $\mathbb{S}$ des schémas de types. Cette sorte nous permet de formaliser la contrainte sur l'utilisation de la quantification, directement dans le système de sortes. En particulier, on ajoute deux règles de formation des types aux systèmes introduits dans le chapitre III, figure 11.1:

$$\frac{\Gamma \vdash \sigma :: \mathbb{T}}{\Gamma \vdash \sigma :: \mathbb{S}} \qquad \frac{\Gamma,\alpha :: \mathbb{T} \vdash \sigma :: \mathbb{S}}{\Gamma \vdash \forall\alpha.\sigma :: \mathbb{S}}$$

La première règle signifie qu'un type est aussi un schéma de type, la seconde permet de construire un schéma de types $\forall\alpha.\tau$. Il est facile de montrer que $\Gamma \vdash \tau :: \mathbb{T}$, implique que τ ne contient aucune quantification. Il faut aussi modifier la règle de formation des environnements de la manière suivante:

$$\frac{\Gamma \vdash * \qquad \Gamma \vdash \sigma :: \mathbb{S} \qquad x \notin \mathbf{dom}(\Gamma)}{\Gamma,x : \sigma \vdash *}$$

On note $\mathtt{B}_\forall$ le système de types polymorphes basé sur B. Ce système est donné dans la figure 13.1. Comme on choisit de nommer les règles du système $\mathtt{B}_\forall$, d'après les règles correspondantes de B, on utilisera la notation $\Gamma \vdash P : \tau\ [\mathtt{B}_\forall]$, ou $\Gamma \vdash P : \tau\ [\mathtt{B}]$, pour séparer les jugements de ces deux systèmes. La définition de $\mathtt{B}_\forall$ nécessite d'introduire quelques notations, qui sont données dans la définition 13.3. Intuitivement, le système $\mathtt{B}_\forall$ est équivalent au système formé par B et étendu par deux règles de typage qui permettent, respectivement, de *spécialiser* et de *généraliser* le type d'un terme.

Définition 13.2 (Généralisation et spécialisation) Un terme P de type σ, dans un environnement où la variable α n'apparaît pas, a aussi le type $\forall\alpha.\sigma$. On dit qu'on généralise le type de P.

$$\frac{\Gamma,\alpha :: \mathbb{T} \vdash P : \sigma}{\Gamma \vdash P : \forall\alpha.\sigma}\ \text{(type gen)}$$

Un terme de type $\forall\alpha.\sigma$, a aussi le type $\sigma\{\tau/\alpha\}$. On dit qu'on spécialise, ou encore instancie, son type.

$$\frac{\Gamma \vdash P : \forall\alpha.\sigma \qquad \Gamma \vdash \tau :: \mathbb{T}}{\Gamma \vdash P : \sigma\{\tau/\alpha\}}\ \text{(type inst)}$$

On aura souvent besoin de généraliser le type d'un terme au maximum, c'est-à-dire d'utiliser la règle (type gen) le plus possible. Pour cela, on défini le schéma de type $\mathrm{Gen}_\Gamma(\tau)$, qui associe à un type et un environnement, son type le plus quantifié.

Définition 13.3 (Clôture d'un type et substitutivité) Soit Γ un environnement et τ un type tel que $\Gamma \vdash \tau :: \mathbb{T}$. On note $\mathrm{Gen}_\Gamma(\tau)$ le type $\forall\tilde{\alpha}.\tau$, où $\tilde{\alpha}$ est l'ensemble des variables libres de τ qui n'apparaissent dans aucun type de Γ. Plus formellement, si on note $\mathbf{dv}(\Gamma)$ l'ensemble récursivement défini par:

$$\mathbf{dv}(\Gamma,x : \tau) = \mathbf{dv}(\Gamma) \cup \mathbf{fn}(\tau) \qquad\qquad \mathbf{dv}(\Gamma,\alpha :: \kappa) = \mathbf{dv}(\Gamma)$$

On définit $\mathrm{Gen}_\Gamma(\tau)$ comme étant le type $\forall\tilde{\alpha}.\tau$, avec $\tilde{\alpha} = (\mathbf{fn}(\tau) \cap \mathbf{dv}(\Gamma))$. La relation de substitutivité (ou subsumption), notée $\prec$, est telle que $\Gamma \vdash \forall\tilde{\alpha}.\tau \prec \tau\{\tilde{\vartheta}/\tilde{\alpha}\}$, si pour tout i de l'intervalle $[1..n]$, on a: $\Gamma \vdash \vartheta_i :: \mathbb{T}$. On utilisera la notation $\sigma \prec \tau$, lorsque l'environnement est connu.

Plutôt que de choisir un système de types avec une règle d'instantiation et de généralisation, on a préféré considérer un système de types orienté par la syntaxe, dans lequel ces deux règles sont intégrées aux règles de typage pour les constructeurs du calcul. C'est, par exemple, l'approche choisie dans [31], par rapport à l'approche de [39].

Dans ce système, donné dans la figure 13.1, on donne toujours un type simple aux processus. C'est-à-dire que les séquents sont de la forme $\Gamma \vdash P : \tau\ [\mathtt{B}_\forall]$. Ceci correspond, implicitement, à typer P avec le schéma de types $\mathrm{Gen}_\Gamma(\tau)$. Les différences entre les systèmes $\mathtt{B}_\forall$ et B se trouvent dans les règles (decl) et (mdecl), qui «intègrent» la règle de généralisation, et dans la règle (type ax), qui intègre la règle d'instantiation.

Les règles (type abs), (type app), (type new) et (type par) sont les mêmes que dans le système B (cf. figure 11.2).

$$\frac{(u:\sigma)\in\Gamma \qquad \Gamma\vdash\sigma\prec\tau}{\Gamma\vdash u:\tau}\ \text{(type ax)}$$

$$\frac{\Gamma,u:\tau,\Gamma'\vdash P:\tau}{\Gamma,u:\mathrm{Gen}_\Gamma(\tau),\Gamma'\vdash\langle u\Leftarrow P\rangle:\mathtt{o}}\ \text{(type decl)}$$

$$\frac{\Gamma,u:\tau,\Gamma'\vdash P:\tau}{\Gamma,u:\mathrm{Gen}_\Gamma(\tau),\Gamma'\vdash\langle u=P\rangle:\mathtt{o}}\ \text{(type mdecl)}$$

Fig. 13.1: Système de types avec polymorphisme paramétrique: $\mathtt{B}_\forall$

13.1 Récursion polymorphe

Dans cette section, nous comparons le système $\mathtt{B}_\forall$ avec le système de type de ML. Ceci nous permet d'étudier deux choix possibles dans le typage des déclarations. On étudie aussi la complexité du problème de l'inférence de types pour ces deux choix.

Pour cette comparaison, seul un fragment réduit de ML, appelé mini-ML [31], est nécessaire. Il s'agit du λ-calcul augmenté des opérateurs de définition: $(\mathbf{let}\,x=M\,\mathbf{in}\,N)$, et de point fixe: $(\mathbf{rec}\,x.M)$. On rappelle qu'on distingue l'opérateur $(\boldsymbol{\lambda}x)P$: la «petite» abstraction du calcul bleu, de $\lambda x.M$: l'équivalent d'ordre supérieur de $\boldsymbol{\Lambda}$ qui permet la substitution d'une variable par un terme. Les termes de mini-ML sont générés par la grammaire suivante.

$$M,N ::= x \mid (MN) \mid \lambda x.M \mid \mathbf{let}\,x=N\,\mathbf{in}\,M \mid \mathbf{rec}\,x.M$$

Un système de types pour ce calcul, basé sur la définition donnée par L. Damas dans sa thèse [40], est donné à la figure 13.2.

Remarque Comme pour le système B, on note $\Gamma\vdash M:\tau$ les séquents du système de types de mini-ML. Le système présenté dans la figure 13.2, est une version de celui donné dans [31], aménagé pour se plier à nos conventions. En particulier, le système donné dans [31] n'a pas de système de sortes. Notre choix de considérer un système de sortes nous permet de conserver une présentation uniforme des systèmes de types introduit dans ce manuscrit. On ne discute pas du fait que cette présentation puisse modifier l'ensemble des termes bien typé de mini-ML. ■

Une caractéristique du calcul de Damas-Milner est que l'occurrence d'une fonction à l'intérieur du corps de sa définition (récursive), ne peut utiliser qu'un type simple. Pour dépasser cette limitation, A. Mycroft [102] a suggéré une règle de typage «polymorphe» pour la récursion.

Définition 13.4 (Récursion polymorphe) La règle de typage de l'opérateur de récursion, dans le système de Damas-Milner: DM , est la règle (rec) ci-dessous. A. Mycroft propose de généraliser cette règle pour permettre à x d'avoir un type polymorphe dans le corps de la récursion. Le système de types de la figure 13.2, augmenté de la règle (rec+), donne le système de Milner-Mycroft: MM

$$\frac{\Gamma,x:\tau\vdash M:\tau}{\Gamma\vdash\mathbf{rec}\,x.M:\tau}\ \text{(rec)} \qquad \frac{\Gamma,x:\mathrm{Gen}_\Gamma(\tau)\vdash M:\tau}{\Gamma\vdash\mathbf{rec}\,x.M:\tau}\ \text{(rec+)}$$

On différencie ces deux systèmes en notant $\Gamma\vdash M$ [DM], les séquents du premier système, utilisant la règle (rec), et $\Gamma\vdash M$ [MM] les séquents qui utilisent la règle (rec+).

$$\frac{(x:\sigma)\in\Gamma \qquad \sigma \prec \tau}{\Gamma \vdash x:\tau}\ (\text{taut}) \qquad \frac{\Gamma \vdash N:\tau \qquad \Gamma,x:\mathrm{Gen}_\Gamma(\tau)\vdash M:\vartheta}{\Gamma \vdash \mathbf{let}\, x = N\, \mathbf{in}\, M:\vartheta}\ (\text{let})$$

$$\frac{\Gamma,x:\tau \vdash M:\vartheta}{\Gamma \vdash \lambda x.M:\tau\to\vartheta}\ (\text{abs}) \qquad \frac{\Gamma \vdash M:\tau\to\vartheta \qquad \Gamma\vdash N:\tau}{\Gamma\vdash (MN):\vartheta}\ (\text{app})$$

Fig. 13.2: Système de types pour mini-ML, sans la règle pour la récursion

On peut remarquer que la règle de typage de la récursion dans le système de Damas-Milner (rec) est compatible avec l'interprétation de la récursion, dans laquelle le terme $\mathbf{rec}\, x.M$ est codé par l'application d'un opérateur de point fixe, qu'on notera $\mathbf{Y}$, à la fonction $\lambda x.M$.

$$\mathbf{rec}\, x.M \simeq \mathbf{Y}(\lambda x.M)$$

Dans cette interprétation, le type de $\mathbf{Y}$ est $\forall\alpha.(\alpha\to\alpha)\to\alpha$. Cependant, comme nous l'avons déjà dit, cette règle est trop contraignante, car elle n'autorise pas la définition d'une fonction, à contenir des appels récursifs à elle-même sous des types différents. C'est ce qu'on appelle la *récursion polymorphe*.

On introduit un nouveau système de types pour le calcul bleu, $\mathtt{B}_{\mathtt{MM}}$, qui permet de typer la récursion polymorphe.

Définition 13.5 (Systèmes de types polymorphes) On dénote $\mathtt{B}_\forall$ le système définit par les règles de la figure 13.1. Le système $\mathtt{B}_{\mathtt{MM}}$ est le système de types obtenu à partir de $\mathtt{B}_\forall$, en remplaçant les règles (type decl) et (type mdecl), par les règles (type decl+) et (type mdecl+) données dans la figure 13.3.

Ces règles permettent de dériver la règle (type rec+) pour le typage de l'opérateur de récursion dans $\mathtt{B}_{\mathtt{MM}}$, au lieu de la règle(type rec), qui est moins générale.

Dans la première partie, on a montré comment dériver l'opérateur de récursion dans le calcul bleu: $\mathbf{rec}\, x.P \ =_{\mathrm{def}}\ (\boldsymbol{\nu} x)(\langle x = P\rangle \mid x)$. Ce codage induit des règles de typage dérivées pour la récursion, qui sont aussi données dans la figure 13.3. Par comparaison avec le système de types de ML, il apparaît que le système $\mathtt{B}_\forall$ est proche, dans l'esprit, de celui de Damas et que le système $\mathtt{B}_{\mathtt{MM}}$ est proche de celui de Mycroft.

L'ajout de la récursion polymorphe à ML permet de typer plus de termes, plus précisément, elle permet de typer des définitions mutuellement récursives non typable dans le système DM. On montre que le système de types de Milner-Mycroft est correct et qu'il vérifie la propriété de *type principal* [102]. Cependant cette amélioration a un prix: au contraire du système de Damas, le problème de l'inférence de type dans le système MM n'est pas décidable.

Un algorithme d'inférence de type a été disponible dès les premières implantation du langage de programmation ML [94, 39]. Ce qui a été un avantage important vis-à-vis des autres langages de programmation fonctionnel. Cependant, les résultats concernant la complexité du problème de l'inférence de types sont plus récents. Pendant presque 10 ans, un certain nombre de personnes ont pensé que ce problème avait une complexité polynomiale. Ceci jusqu'à ce que H. Mairson prouve que le problème de l'inférence de type polymorphe était de complexité exponentielle [87]. De même, le système de Milner-Mycroft fut tout d'abord «implanté» dans une version de ML, avant que le problème de l'inférence de type dans ce système, soit prouvé indécidable 6 ans après. F. Henglein [65], et indépendamment A. Kfoury, J. Tiuryn et P. Urzyczyn [75], montrèrent que le problème de la typabilité est «log-space» équivalent au problème de la semi-unification, qui

$$\frac{\Gamma,u:\tau,\Gamma' \vdash P:\tau}{\Gamma,u:\mathrm{Gen}_\Gamma(\tau),\Gamma' \vdash \langle u \Leftarrow P\rangle : \mathtt{o}} \text{ (type decl)} \qquad \frac{\Gamma,u:\tau \vdash P:\tau}{\Gamma \vdash \mathbf{rec}\, u.P : \tau} \text{ (type rec)}$$

$$\frac{\Gamma,u:\tau,\Gamma' \vdash P:\tau}{\Gamma,u:\mathrm{Gen}_\Gamma(\tau),\Gamma' \vdash \langle u = P\rangle : \mathtt{o}} \text{ (type mdecl)}$$

$$\frac{\Gamma,u:\mathrm{Gen}_\Gamma(\tau),\Gamma' \vdash P:\tau}{\Gamma,u:\mathrm{Gen}_\Gamma(\tau),\Gamma' \vdash \langle u \Leftarrow P\rangle : \mathtt{o}} \text{ (type decl+)} \qquad \frac{\Gamma,u:\mathrm{Gen}_\Gamma(\tau) \vdash P:\tau}{\Gamma \vdash \mathbf{rec}\, u.P : \tau} \text{ (type rec+)}$$

$$\frac{\Gamma,u:\mathrm{Gen}_\Gamma(\tau),\Gamma' \vdash P:\tau}{\Gamma,u:\mathrm{Gen}_\Gamma(\tau),\Gamma' \vdash \langle u = P\rangle : \mathtt{o}} \text{ (type mdecl+)}$$

Fig. 13.3: Typage des déclarations et règles dérivées pour la récursion

est le problème de résoudre un système d'inéquations, pour la relation $\prec$, entre termes du premier ordre.

On a montré que le problème de la semi-unification est récursivement indécidable. Par conséquent, nous ne pouvons pas raisonnablement attendre que le problème de la typabilité soit décidable pour $\mathtt{B}_{\mathtt{MM}}$. Résultat plus surprenant, nous montrons aussi – dans la section suivante – l'indécidabilité du problème de la typabilité dans le système $\mathtt{B}_\forall$.

13.2 Complexité du problème de l'inférence de type

On montre que les systèmes de types avec polymorphisme sont corrects. La preuve est semblable à celle donnée dans le chapitre 15. On peut aussi trouver cette preuve dans [36]. Dans ce dernier papier, on trouve également une preuve de la propriété de type principal. Dans cette section, nous montrons que le problème de l'inférence de type est indécidable pour les deux systèmes $\mathtt{B}_\forall$ et $\mathtt{B}_{\mathtt{MM}}$.

Avant de montrer cette propriété, nous montrons quelques résultats intermédiaires. Soit $(\!|.|\!)$ l'interprétation de mini-ML dans $\boldsymbol{\pi}^\star$ donnée ci-dessous.

Définition 13.6 (Codage de mini-ML)

$$\begin{array}{rcl}
(\!|x|\!) & = & x \\
(\!|\lambda x.M|\!) & = & (\boldsymbol{\lambda} x)(\!|M|\!) \\
(\!|MN|\!) & = & \mathbf{def}\, u = (\!|N|\!)\, \mathbf{in}\, ((\!|M|\!)\, u) \quad u \notin \mathbf{fn}(M,N) \\
(\!|\mathbf{let}\, x = N\, \mathbf{in}\, M|\!) & = & \mathbf{def}\, x = (\!|N|\!)\, \mathbf{in}\, (\!|M|\!) \\
(\!|\mathbf{rec}\, x.M|\!) & = & \mathbf{def}\, x = (\!|N|\!)\, \mathbf{in}\, x
\end{array}$$

Ce codage est obtenu à partir de la définition 8.1, généralisée pour tenir compte des opérateurs de définition et de récursion. On peut noter que, dans ce codage, on retrouve l'équivalence opérationnelle entre $(\mathbf{let}\, x = N\, \mathbf{in}\, M)$ et $(\lambda x.M)N$. Nous montrons qu'il est possible d'associer un séquent de $\mathtt{B}_{\mathtt{MM}}$ à tout séquent de $\mathtt{MM}$, de la manière suivante.

Théorème 13.1 (Équivalence du typage) *Si le terme M de mini-*ML *est typable avec le type $\Gamma \vdash M : \tau$ [MM], alors on peut typer $(\!|M|\!)$ avec le même type dans $\boldsymbol{\pi}^\star$, c'est-à-dire $\Gamma \vdash (\!|M|\!) : \tau$ [$\mathtt{B}_{\mathtt{MM}}$]. Inversement, si $\Gamma \vdash (\!|M|\!) : \tau$ [$\mathtt{B}_{\mathtt{MM}}$], alors $\Delta \vdash M : \tau$ [MM], où $\Delta =_{\mathrm{def}} \Gamma_{|\mathbf{fn}(\Gamma)\setminus\mathbf{fn}(M)}$.*

Preuve (Théorème 13.1) La première propriété se prouve par induction sur l'inférence de $\Gamma \vdash M : \tau$ [MM]. Dans les cas de l'application, du «let», et de la récursion, on utilise une règle de typage

dérivée pour les définitions

$$\frac{\Gamma,u : \mathrm{Gen}_\Gamma(\tau) \vdash N : \tau \qquad \Gamma,u : \mathrm{Gen}_\Gamma(\tau) \vdash M : \vartheta}{\Gamma \vdash \mathbf{def}\, u = N \,\mathbf{in}\, M : \vartheta}$$

Cette règle est obtenue par l'inférence suivante.

$$\dfrac{\dfrac{\dfrac{\Gamma,u : \mathrm{Gen}_\Gamma(\tau) \vdash N : \tau}{\Gamma,u : \mathrm{Gen}_\Gamma(\tau) \vdash \langle u = N\rangle : \mathtt{o}}\ \text{(type mdecl+)} \qquad \Gamma,u : \mathrm{Gen}_\Gamma(\tau) \vdash M : \vartheta}{\Gamma,u : \mathrm{Gen}_\Gamma(\tau) \vdash (\langle u = N\rangle \mid M) : \vartheta}\ \text{(type par)}}{\Gamma \vdash \mathbf{def}\, u = N \,\mathbf{in}\, M : \vartheta}\ \text{(type new)}$$

Pour la propriété inverse, on tient compte de la règle d'affaiblissement (type weak), ce qui introduit l'environnement $\Gamma_{|\mathbf{fn}(\Gamma)\setminus\mathbf{fn}(M)}$. En effet, on peut retirer du domaine de l'environnement Γ, les variables qui ne sont pas libres dans M: si $\Gamma \vdash (\!|M|\!) : \tau\ [\mathtt{B_{MM}}]$, alors $\Gamma_{|\mathbf{fn}(\Gamma)\setminus\mathbf{fn}(M)} \vdash M : \tau\ [\mathtt{B_{MM}}]$, et cette inférence n'utilise pas la règle (type weak). La propriété se prouve alors par induction sur l'inférence de $\Gamma_{|\mathbf{fn}(\Gamma)\setminus\mathbf{fn}(M)} \vdash M : \tau\ [\mathtt{B_{MM}}]$. □

Comme nous l'avons dit à la section 13.1, le problème de la typabilité dans le système MM a été prouvé indécidable [75, 65]. Par conséquent, une conséquence immédiate du théorème 13.1 est:

Théorème 13.2 *Le problème de l'inférence de type dans* $\mathtt{B_{MM}}$ *est indécidable.*

On peut d'ailleurs prouver une propriété plus fine: pour citer F. Henglein [65]: «*pour tout système d'inéquations* $\mathcal{I}$*, il existe une expression de la forme* $\mathbf{rec}\, x.M$*, calculable en log-space, où* M *ne comporte pas de récursion ni de let, c'est-à-dire* M *est un terme pur de* $\mathbf{\Lambda}$*, et tel que* $\mathcal{I}$ *est semi-unifiable, si et seulement si* $\mathbf{rec}\, x.M$ *est typable dans* MM »[1]. Dans le cas du calcul bleu, on dit que le terme P n'a pas de définition récursive, s'il ne comporte aucune déclaration $\langle u = Q\rangle$ ou $\langle u \Leftarrow Q\rangle$, telle que $u \in \mathbf{fn}(Q)$. Comme on a prouvé que le codage de mini-ML vers $\boldsymbol{\pi}^\star$ préserve le typage, on a prouvé qu'on peut réduire le problème de la semi-unification, au problème du typage dans $\mathtt{B_{MM}}$.

Lemme 13.3 (Semi-unification et typabilité dans $\mathtt{B_{MM}}$**)** *Pour tout système d'inéquations* $\mathcal{I}$*, on peut calculer en «log-space», un processus du calcul bleu de la forme* $\mathbf{rec}\, x.P$*, où* P *n'a pas de définitions récursives et tel que* $\mathcal{I}$ *est semi-unifiable, si et seulement si* $\mathbf{rec}\, x.P$ *est typable dans* $\mathtt{B_{MM}}$.

Ce lemme implique le théorème 13.2, puisque le problème de la semi-unification a été prouvé récursivement indécidable [74, 131].

Nous montrons maintenant que la typabilité dans le système $\mathtt{B}_\forall$ est aussi un problème indécidable. Cependant, avant de prouver le théorème 13.5, nous avons besoin d'une propriété intermédiaire.

Lemme 13.4 *Soit* N *un terme de* $\mathbf{\Lambda}$*. Si* $\Delta \vdash (\!|N|\!) : \varrho\ [\mathtt{B_{MM}}]$*, alors* $\Delta \vdash (\!|N|\!) : \varrho\ [\mathtt{B}_\forall]$.

Preuve La preuve est faite par induction sur l'inférence de $\Delta \vdash (\!|N|\!) : \varrho\ [\mathtt{B_{MM}}]$ et utilise le fait qu'il n'y a pas de définitions récursives dans $(\!|N|\!)$. □

Théorème 13.5 *Le problème de l'inférence de type dans* $\mathtt{B}_\forall$ *est indécidable.*

Preuve Pour démontrer le théorème 13.5, il est suffisant de montrer que le problème de la semi-unification se réduit au problème de la typabilité dans $\mathtt{B}_\forall$. On commence par prouver le premier sens de cette équivalence.

1. On peut trouver un résultat équivalent de A. J. Kfoury *et al.* dans [75].

Soit $\mathcal{I}$ un système d'inéquations, et $\mathbf{rec}\, u.M_u$ le terme qui lui est associé dans le lemme 13.3. En particulier M_u ne contient pas de définitions récursives. Si $\mathcal{I}$ est unifiable, alors $\Gamma \vdash \mathbf{rec}\, u.M_u : \tau\ [\mathtt{B}_{\mathtt{MM}}]$. Soit v une nouvelle référence, et soit M_v le processus $M_u\{v/u\}$. On montre que

$$\Gamma \vdash (\boldsymbol{\nu} u)(\mathbf{def}\, v = u\, \mathbf{in}\, (\langle u = M_v\rangle \mid v)) : \tau\ [\mathtt{B}_{\forall}]$$

On rappelle que $\mathbf{def}\, v = u\, \mathbf{in}\, (\langle u = M_v\rangle \mid v)$, est égal à $(\boldsymbol{\nu} v)(\langle v = u\rangle \mid \langle u = M_v\rangle \mid v)$. Notre hypothèse est que $\Gamma \vdash \mathbf{rec}\, u.M_u : \tau\ [\mathtt{B}_{\mathtt{MM}}]$. On montre alors qu'il existe un type ϑ tel que

$$\frac{\Gamma,u : \mathrm{Gen}_\Gamma(\vartheta) \vdash M_u : \vartheta\ [\mathtt{B}_{\mathtt{MM}}] \qquad \mathrm{Gen}_\Gamma(\vartheta) \prec \tau}{\Gamma \vdash \mathbf{rec}\, u.M_u : \tau\ [\mathtt{B}_{\mathtt{MM}}]}$$

Dans la suite, on note η le schéma de type $\mathrm{Gen}_\Gamma(\vartheta)$. Par hypothèse, M_u (et donc M_v) n'a pas de définition récursive. De plus, M_v est – par construction – la traduction d'un terme de $\boldsymbol{\Lambda}$. Par conséquent, en appliquant le lemme 13.4 au cas du terme M_u et au jugement $\Gamma,u : \eta \vdash M_u : \vartheta\ [\mathtt{B}_{\mathtt{MM}}]$, on obtient que $\Gamma,u : \eta \vdash M_u : \vartheta\ [\mathtt{B}_{\forall}]$. De plus v est un nom nouveau, et par conséquent on a aussi $\Gamma,v : \eta \vdash M_v : \vartheta\ [\mathtt{B}_{\forall}]$. On utilise pour cela une propriété équivalente au lemme 15.5, sur le renommage des variables. On montre alors que le jugement suivant est valide dans $\mathtt{B}_{\forall}$.

$$\dfrac{\dfrac{\dfrac{\Gamma,v : \eta \vdash M_v : \vartheta \qquad u \notin \mathbf{fn}(M_v)}{\Gamma,u : \vartheta,v : \eta \vdash M_v : \vartheta}\ (\text{type weak})}{\Gamma,u : \eta,v : \eta \vdash \langle u = M_v\rangle : \mathtt{o}}\ (\text{type decl}) \qquad \dfrac{\eta \prec \tau}{\Gamma,u : \eta,v : \eta \vdash v : \tau}\ (\text{type ax})}{\Gamma,u : \eta,v : \eta \vdash (\langle u = M_v\rangle \mid v) : \tau}$$

On montre aussi que

$$\dfrac{\dfrac{\dfrac{\eta \prec \tau}{\Gamma,u : \eta \vdash u : \tau}\ (\text{type ax})}{\Gamma,u : \eta,v : \tau \vdash u : \tau}\ (\text{type weak})}{\Gamma,u : \eta,v : \eta \vdash \langle v = u\rangle : \mathtt{o}}\ (\text{type decl})$$

Par conséquent, en appliquant les règles (type par) et (type new), on obtient que:

$$\Gamma \vdash (\boldsymbol{\nu} u)(\mathbf{def}\, v = u\, \mathbf{in}\, (\langle u = M_v\rangle \mid v)) : \tau\ [\mathtt{B}_{\forall}] \tag{13.1}$$

Réciproquement, si le processus dans (13.1) est typable dans $\mathtt{B}_{\forall}$, alors le système d'inéquations $\mathcal{I}$ est semi-unifiable. Par conséquent, pour tout système d'inéquations $\mathcal{I}$, on peut exhiber un terme de $\boldsymbol{\pi}^\star$– calculable en log-space –, disons P, tel que $\mathcal{I}$ est semi-unifiable si et seulement si P est typable dans le système $\mathtt{B}_{\forall}$. □

En conclusion, nous avons montré que les deux généralisations du calcul bleu aux types polymorphes données ici ont la même complexité vis-à-vis du problème de l'inférence de type. Ce résultat semble paradoxal, en effet $\mathtt{B}_{\forall}$ est inspiré par le système de Damas-Milner, pour lequel le problème de l'inférence de type est décidable. Le problème est que, dans $\boldsymbol{\pi}^\star$, une déclaration ne lie pas le nom qu'elle définit. On peut alors avoir plusieurs déclarations sur le même nom, ce qui crée des «contraintes récursives» entre plusieurs variables de type de l'environnement: c'est la limitation classique du typage des fonctions mutuellement récursives.

13.3 Polymorphisme et définitions

Dans cette section, nous proposons un système de types avec polymorphisme, pour lequel le problème de la typabilité est décidable. L'idée est de restreindre l'utilisation du polymorphisme au définitions: $\mathbf{def}\, D\, \mathbf{in}\, P$, de la même manière que le polymorphisme de ML est limité à la construction $\mathbf{let}\, x = N\, \mathbf{in}\, M$. C'est aussi cette limitation qui a été choisie pour le typage des termes du calcul Join [53]. On note $\mathtt{B}_{\mathrm{ML}}$ ce nouveau système de types, dont la définition est donnée dans la figure 13.4.

Les règles (type abs), (type app), (type new) et (type par) sont les mêmes que dans le système B (cf. figure 11.2).

$$\frac{(u:\sigma)\in\Gamma \qquad \sigma \prec \tau}{\Gamma \vdash u : \tau}\ \text{(type ax)}$$

$$\frac{\Gamma \vdash P : \tau \qquad (u:\tau)\in\Gamma}{\Gamma \vdash \langle u \Leftarrow P\rangle : \mathtt{o}}\ \text{(type decl)} \qquad \frac{\Gamma \vdash P : \tau \qquad (u:\tau)\in\Gamma}{\Gamma \vdash \langle u = P\rangle : \mathtt{o}}\ \text{(type mdecl)}$$

$$\frac{\Gamma,\{u_i : \tau_i{}^{i\in[1..n]}\} \vdash R_i : \tau_i \qquad \Gamma,\{u_i : \mathrm{Gen}_{\Delta_i}(\tau_i)^{i\in[1..n]}\} \vdash P : \tau}{\Gamma \vdash \mathbf{def}\, u_1 = R_1, \ldots, u_n = R_n \,\mathbf{in}\, P : \tau}\ \text{(type def)}$$

$$(\text{avec } \Delta_i =_{\text{def}} \Gamma,\{u_j : \tau_j{}^{j\neq i}\})$$

Fig. 13.4: Système de types avec polymorphisme restreint aux définitions: $\mathtt{B}_{\mathrm{ML}}$

Notre choix rejoint le choix fait dans ML : l'opérateur de définition n'augmente pas l'expressivité de notre calcul, de même que l'opérateur $\mathbf{let}\, x = N \,\mathbf{in}\, M$ peut être codé par le terme $(\lambda x.M)N$. Par contre, on peut utiliser cet opérateur pour typer différemment des termes opérationnellement équivalent. On retrouve alors les deux mécanismes liés dans l'opérateur let:

partage la définition permet de partager l'utilisation d'une ressource. Il faut remarquer que, alors que le let de ML est associé à une stratégie de réduction en appel par valeur, le calcul bleu est associé à une stratégie en appel par nom [108];

polymorphisme la généralisation est restreinte aux types des noms liés par une définition.

Il est intéressant de comparer cette remarque avec la distinction faite par X. Leroy dans [80, 82], entre un constructeur «let val», pour le partage des valeurs après réductions, et un constructeur «let name», pour la généralisation des types. On retrouve la même distinction faite dans [64]. Il montre aussi que, si on choisit une stratégie avec appel par nom pour le second constructeur, alors l'ajout des références polymorphe ne pose pas de problème. Une remarque que nous faisons dans la section 13.4.

Remarque Dans la règle de typage (type def) de la figure 13.4, on utilise des définitions mutuellement récursives. Afin d'interdire «d'augmenter» dynamiquement le nombre de déclarations d'une définition, on suppose qu'on n'utilise jamais l'équivalence structurelle pour obtenir la relation suivante (cf. figure 3.2): $\mathbf{def}\, D \,\mathbf{in}\, (\mathbf{def}\, D' \,\mathbf{in}\, P) \equiv \mathbf{def}\, D, D' \,\mathbf{in}\, P$. Cette restriction ne modifie pas l'expressivité de notre calcul, mais elle est importante: sans elle, les jugements de types ne sont pas conservé par modification structurelle, et donc on perd la propriété de conservation du typage. ■

On montre dans [36] que les propriétés de préservation du typage et de type principal sont vraies pour le système $\mathtt{B}_{\mathrm{ML}}$. Nous montrons aussi qu'on peut associer un jugement de $\mathtt{B}_{\mathrm{ML}}$, à tout jugement de DM. C'est-à-dire qu'on a un résultat équivalent à celui déjà montré entre $\mathtt{B}_{\mathtt{MM}}$ et MM (cf. théorème 13.1).

Théorème 13.6 (Équivalence du typage) *Soit* $(\!|.|\!)$ *le codage de mini-*ML *dans* $\boldsymbol{\pi}^\star$ *donné à la définition 13.6. Si le terme* M *de mini-*ML *est typable avec le type* $\Gamma \vdash M : \tau$ $[\mathtt{DM}]$*, alors on peut typer* $(\!|M|\!)$ *avec le même type dans* $\mathtt{B}_{\mathrm{ML}}$*, c'est-à-dire* $\Gamma \vdash (\!|M|\!) : \tau$ $[\mathtt{B}_{\mathrm{ML}}]$*. Inversement, si* $\Gamma \vdash (\!|M|\!) : \tau$ $[\mathtt{B}_{\mathrm{ML}}]$*, alors* $\Delta \vdash M : \tau$ $[\mathtt{DM}]$*, où* $\Delta =_{\text{def}} \Gamma_{|\mathbf{fn}(\Gamma)\setminus\mathbf{fn}(M)}$.

Preuve la preuve est similaire à celle du théorème 13.1. □

Ce qui est nouveau, est que l'on peut fournir une procédure décidable qui, étant donné le type des variables libres d'un processus P, infère le type de P. Pour prouver cette propriété, il suffit

de donner un algorithme qui utilise uniquement l'unification entre termes du premier ordre, puis de montrer que cet algorithme calcule les bons types. Nous ne donnons pas les détails de cet algorithme ici. Disons simplement qu'il peut être obtenu simplement en étendant l'algorithme d'inférence de ML, noté W dans [40].

Cependant, il faut noter que, dans les systèmes de types étudiés dans ce chapitre, nous n'avons considéré ni les enregistrements, ni la récursion sur les types. L'ajout des enregistrements complique le système. Comme avec l'ajout du sous-typage étudié au chapitre 12, il est nécessaire de pouvoir marquer qu'un champ est absent pour avoir la propriété de type principal, et donc pour pouvoir espérer avoir un algorithme d'inférence de type. Quant à l'ajout d'un opérateur de récursion pour les types, il semble assez délicat.

13.4 Cas du calcul non local

Dans ce chapitre, nous avons considéré le cas du calcul bleu local. Dans cette section, nous nous intéressons au typage du calcul bleu sans restriction sur l'abstraction des déclarations.

Dans l'hypothèse où le calcul n'est pas local, on peut montrer avec un exemple très simple, que la propriété de préservation du typage (pour les systèmes $\mathtt{B}_{\mathrm{ML}}$, $\mathtt{B}_{\forall}$ et $\mathtt{B}_{\mathtt{MM}}$) est fausse. Ceci nous donne donc une raison supplémentaire d'interdire l'abstraction des références qui apparaissent en sujet d'une déclaration. Soit R un processus du calcul non-local, défini de la manière suivante:

$$R =_{\mathrm{def}} \langle u \Leftarrow P\rangle \mid ((\boldsymbol{\lambda} v)\langle v \Leftarrow Q\rangle)u \qquad (u \text{ n'est pas libre dans } P \text{ et } Q)$$

et Γ un environnement tel que $(u : \sigma) \in \Gamma$, et $\Gamma \vdash P : \tau_p$, et $\Gamma \vdash Q : \tau_q$. Par exemple on peut choisir:

$$\begin{cases} \sigma =_{\mathrm{def}} \forall\alpha.(\alpha \to \alpha) & \\ P =_{\mathrm{def}} (\boldsymbol{\lambda} x)x & \text{et} \quad \Gamma \vdash P : \tau_p =_{\mathrm{def}} (\alpha \to \alpha) \\ Q =_{\mathrm{def}} (\boldsymbol{\lambda} x)(x+1) & \text{et} \quad \Gamma \vdash Q : \tau_q =_{\mathrm{def}} (\mathtt{int} \to \mathtt{int}) \end{cases}$$

Une condition suffisante pour que R soit bien typé est que $\sigma = \mathrm{Gen}_\Gamma(\tau_p)$ et $\sigma \prec \tau_q$. Ces contraintes sont induites par l'utilisation des règles (type decl), (type abs) et (type app). Or R se réduit, en un pas, en le processus R' égal à $(\langle u \Leftarrow P\rangle \mid \langle u \Leftarrow Q\rangle)$, c'est-à-dire à $(\langle u \Leftarrow (\boldsymbol{\lambda} x)x\rangle \mid \langle u \Leftarrow (\boldsymbol{\lambda} x)(x+1)\rangle)$. Les contraintes nécessaires pour typer R' sont plus fortes que pour R. En effet on doit montrer que $\sigma = \mathrm{Gen}_\Gamma(\tau_p) = \mathrm{Gen}_\Gamma(\tau_q)$. Or cette équation n'est clairement pas vérifiée dans notre exemple, puisque $\forall\alpha.(\alpha \to \alpha) \neq (\mathtt{int} \to \mathtt{int})$.

Ce problème n'apparaît pas avec le système de types simples. En particulier, on peut montrer que le système $\mathtt{B}$ est correct pour le calcul non-local. Nous sommes donc dans une situation comparable a l'extension «naïve» de ML avec des références polymorphes. En fait, comme le montre X. Leroy dans sa thèse [81], c'est un problème qu'on rencontre pour chaque extension de ML avec des opérateurs impératifs: références, continuations, canaux de communications. Plusieurs systèmes de types ont été proposés pour «typer» les références polymorphes: M. Tofte [129], D. Macqueen, A. Wright, X. Leroy ... Un état de l'art sur ces systèmes est donné dans [80]. On montre, sur un simple exemple, comment on peut adapter une de ces solutions pour résoudre notre problème. Ce qui renforce notre proposition qu'il existe un lien entre le problème de l'ajout de «références polymorphe» et l'ajout de la «possibilité d'abstraire les références».

Une approche communément choisie dans ML pour contraindre l'utilisation du polymorphisme, consiste à annoter les variables de types qui apparaissent libres dans le type des références. On dit souvent que ces variables sont *faibles*, ou encore *dangereuses*. On réserve alors un traitement spécial aux variables faibles au moment de généraliser les types. Par exemple, la solution la plus simple (mais aussi la plus restrictive) consiste à interdire la généralisation des variables faibles. Dans cette section, nous ne définissons pas formellement un nouveau système de types pour le

calcul bleu, mais nous nous contentons de donner une intuition des changements à apporter au système $\mathtt{B}_\forall$, afin de typer le calcul non-local.

On annote avec une étoile les variables faibles: $\alpha^\star$. Cette notation s'étend aux expressions de types de manière naturelle. Par exemple on a: $(\tau \to \vartheta)^\star =_{\text{def}} \tau^\star \to \vartheta^\star$. En se basant sur le parallèle que nous avons tiré avec ML, on redéfinit alors le système $\mathtt{B}_\forall$, pour «affaiblir» les variables génériques potentiellement dangereuses. En particulier, on considère la règle spéciale suivante, pour typer les abstractions sur une référence. Notez que, avec nos conventions, u est une référence et τ est un type simple.

$$\frac{\Gamma, u : \tau^\star \vdash P : \vartheta}{\Gamma \vdash (\boldsymbol{\lambda} u) P : \tau^\star \to \vartheta} \text{ (type abs}^\star\text{)}$$

On modifie aussi la relation de substitutivité $\prec$, afin de ne pas faire arbitrairement apparaître des types annotés $\tau^\star$. Par exemple, on interdit les cas de la forme $(\forall\alpha.\alpha) \prec \tau^\star$. Il suffit pour cela d'interdire l'introduction de variables faibles au moment de l'instanciation d'un schéma de types.

On étudie maintenant ce qui se passe avec le typage de l'exemple précédent. Dans le typage de R, on doit utiliser la règle (type abs*) pour typer l'abstraction $(\boldsymbol{\lambda} v)\langle v \Leftarrow (\boldsymbol{\lambda} x)(x+1)\rangle$, ce qui nous donne le jugement suivant:

$$\frac{\dfrac{v : (\mathtt{int} \to \mathtt{int})^\star \vdash (\boldsymbol{\lambda} x)(x+1) : (\mathtt{int} \to \mathtt{int})^\star}{v : (\mathtt{int} \to \mathtt{int})^\star \vdash \langle v \Leftarrow (\boldsymbol{\lambda} x)(x+1)\rangle : \mathtt{o}} \text{ (type decl)}}{\emptyset \vdash (\boldsymbol{\lambda} v)\langle v \Leftarrow (\boldsymbol{\lambda} x)(x+1)\rangle : (\mathtt{int} \to \mathtt{int})^\star \to \mathtt{o}} \text{ (type abs}^\star\text{)}$$

Avec la définition modifiée de $\prec$, il est alors impossible d'inférer que $\forall\alpha.(\alpha \to \alpha) \prec \mathtt{int}^\star \to \mathtt{int}^\star$. En effet il est devenu interdit de substituer $\mathtt{int}^\star$ à la variable générique α. Par conséquent, on montre que les typages valides de R sont de la forme $u : \mathtt{int}^\star \to \mathtt{int}^\star \vdash R : \mathtt{o}$; typages qui sont préservés par réduction.

L'étude de cet exemple, nous permet de comprendre comment réutiliser les nombreux résultats existants sur le typage de ML avec références polymorphes, pour les appliquer au problème du calcul bleu non-local. Comme dans le cas de l'ajout du sous-typage au calcul bleu, nous n'avons pas besoin d'inventer de nouveaux concepts, et il suffit de reprendre des outils déjà étudiés et bien compris. Cette étude a aussi pour conséquence de fournir des intuitions pour la définition d'un système de types polymorphes implicites pour le π-calcul.

L'auteur ne connaît qu'une seule définition d'un système de types polymorphes pour $\boldsymbol{\pi}$. Ce système, décrit par D. Turner dans sa thèse [130], utilise un typage explicite, on parle aussi de «style à la Church», c'est-à-dire que dans une communication, on doit échanger le type des données qui sont envoyées: les types deviennent des valeurs qui comme les noms, peuvent être émises ou reçues. En utilisant notre système de types polymorphes implicite, ainsi que notre codage de $\boldsymbol{\pi}$ dans $\boldsymbol{\pi}^\star$, on peut alors définir un nouveau système de types pour $\boldsymbol{\pi}$.

13.5 Discussion pour les systèmes de types avec polymorphisme

Dans les sections précédentes, nous avons présenté un système de types avec récursion polymorphe pour $\boldsymbol{\pi}^\star$, adapté des systèmes de types développés pour le langage de programmation ML. On s'est également intéressé à la propriété de conservation du typage et à la propriété de type principal.

Dans le cadre de notre étude, qui vise à étudier le calcul bleu comme modèle de la programmation concurrente, les résultats obtenus ici sont intéressants. En effet ils permettent de cerner la définition d'un système de types pour un langage basé sur $\boldsymbol{\pi}^\star$. Cependant un système de types n'est réellement utile que si on peut fournir une procédure (un algorithme) permettant d'inférer le type d'un programme, ou au moins de vérifier si un terme est bien typé.

Une première approche consiste à limiter l'utilisation du polymorphisme aux définitions. C'est ce qui a été avec l'introduction du système $\mathtt{B}_{\mathrm{ML}}$. En effet, le problème de la reconstruction de type dans $\mathtt{B}_{\mathrm{ML}}$ se réduit trivialement au problème de l'unification entre termes du premier ordre et, par conséquent, il est décidable. Une deuxième approche consisterait à utiliser le système de types $\mathtt{B}_{\mathtt{MM}}$, qui permet de typer plus de termes, et à fournir le type des références dans chaque déclarations: ce qui revient, en ML, à typer les fonctions à l'endroit de leur définition. On peut également se baser sur des semi-algorithmes pour l'inférence de type [102] ou pour la semi-unification [65, section 5], c'est-à-dire des procédures qui terminent dès lors qu'une solution existe.

Il est aussi intéressant de comprendre pourquoi le problème de l'inférence de type est plus compliqué que dans le cas de ML. Contrairement à l'opérateur let de ML, ou au JOIN de [51], la portée des déclarations dans le calcul bleu n'est pas fixée syntaxiquement. Plus précisément, on a une portée lexicale pour la définition des noms, qui est celle de la restriction, mais on a aussi la «scope extrusion». Par conséquent, si on se représente $\boldsymbol{\pi}^\star$ comme un langage de programmation, et les déclarations comme la définition de fonctions, nous sommes dans le cas où toutes les fonctions sont «top-level» et mutuellement récursives. De plus, une référence peut apparaître en sujet de plusieurs déclarations: une situation comparable à l'existence de multiples définitions pour une fonction. Ce problème a été étudié par A. MYCROFT [103], qui l'a nommé *inférence incrémentale de types polymorphes avec mise à jour*. Sa motivation pour étudier ce problème, a été qu'il s'agit du cas du langage de programmation PROLOG, dans lequel les *prédicats* sont «incrémentalement et mutuellement» définis. Les systèmes de types pour PROLOG ont été beaucoup étudiés [101, 10] et il devrait être possible de les adapter à notre besoin.

CHAPITRE 14

Polymorphisme contraint

Dans le chapitre précédent, nous avons introduit un système de types avec polymorphisme paramétrique, et montré en quoi il permet d'augmenter l'ensemble des termes typables. Dans ce chapitre, nous généralisons la notion de type polymorphe, et nous étudions un langage de types dans lequel l'usage de la quantification n'est pas limité à apparaître en tête d'un type.

Plus exactement, nous nous intéressons à une variante de la notion de *polymorphisme contraint*, ou «*bounded polymorphism*» en anglais, introduite dans la définition du langage FUN [28] et que l'on trouve dans le système $\mathtt{F}_{\leq}$ [35], prononcer «F-sub». Cependant, au lieu de choisir la présentation traditionnelle de $\mathtt{F}_{\leq}$, dans laquelle on décore les termes avec des types, on choisit de conserver le mode de présentation de cette partie et de donner un système de types avec *polymorphisme implicite*. C'est-à-dire que nous continuons à considérer séparément les termes et leurs (possibles) types. En particulier, nous ne rajoutons pas les opérateurs de $\mathtt{F}_{\leq}$ qui permettent d'abstraire un type dans un terme ($\Lambda X \leqslant \tau.M$), et d'appliquer un terme à un type ($M\tau$).

Dans le système que nous définissons ici, noté $\mathtt{BF}_{\leqslant}$, nous considérons deux relations de sous-typage. La première, notée $\sqsubseteq$, représente le sous-typage en largeur des enregistrements. La seconde, notée $\leqslant$, est équivalente à la relation donnée au chapitre 12. C'est pour cette raison qu'on conserve la même notation entre ces deux chapitres. Dans $\mathtt{BF}_{\leqslant}$, on remplace le schéma de types $\forall\alpha.\vartheta$, par le type $(\forall t \sqsubseteq \tau \,.\, \vartheta)$, qui exprime la contrainte que, lors d'une instantiation, le type qui remplace la variable t doit être un sous-type de τ pour la relation $\sqsubseteq$. Il faut noter que, afin de respecter les conventions de notations des systèmes de types «d'ordre supérieur», nous utiliserons les lettres r,s,t,... pour désigner les variables de types, à la place de α,β,γ.... De façon formelle, l'opérateur de quantification bornée $(\forall t \sqsubseteq \tau \,.\, \vartheta)$ est associé à la règle d'instantiation suivante.

$$\frac{\Gamma \vdash P : (\forall t \sqsubseteq \tau \,.\, \vartheta) \qquad \Gamma \vdash \tau' \sqsubseteq \tau}{\Gamma \vdash P : \vartheta\{\tau'/t\}} \text{ (type inst)}$$

Dans ce système, par exemple, la fonction sélecteur: $(\boldsymbol{\lambda}x)(x \cdot l)$, a le type $(\forall t \sqsubseteq [\, l : \tau \,] \,.\, (t \to \tau))$. C'est-à-dire un type équivalent à $(t \to \tau)$, mais avec la contrainte que l'argument soit un enregistrement ayant un champ l de type τ.

14.1 Expressions de types

Dans ce chapitre, les opérateurs de types sont ceux définis dans le système de types simples, auxquels on ajoute des constructeurs pour l'application: $(\tau\sigma)$, et l'abstraction des types: $(\Lambda t.\tau)$,

ainsi qu'un constructeur de quantification bornée: $((\forall t \sqsubseteq \tau \,.\, \sigma))$. En particulier, on ne fait plus de distinction entre schéma de types et types. De plus, on a un système avec récursion, et dans lequel les types dépendent des types [59].

Définition 14.1 (Expression de type) Les types sont engendrés par la grammaire suivante. Dans ce chapitre, on ne différencie plus les types et les schémas de types, et on emploie les symboles $\tau,\sigma,\varrho\ldots$ pour les désigner. Pour désigner certains type, on pourra utiliser la même convention que pour les références et utiliser des mots en caractères sans sérif, comme obj par exemple.

$\tau,\sigma,\varrho,\ldots$	::=	r,s,t	variables de type
	|	o	type des processus
	|	$(\tau \to \sigma)$	type fonctionnel
	|	$[\,]$	type enregistrement vide
	|	$[\,\varrho\,, l : \tau\,]$	extension/modification
	|	$(\mu t^{\kappa}.\tau)$	type récursif
	|	$(\Lambda t^{\kappa}.\tau)$	abstraction
	|	$(\tau\sigma)$	application
	|	$(\forall t \sqsubseteq \tau \,.\, \sigma)$	quantification bornée

---

Dans le type $(\forall t \sqsubseteq \tau \,.\, \sigma)$, comme dans le type $\Lambda t.\sigma$, la variable t est liée dans σ. Cependant elle n'est pas liée dans τ. C'est-à-dire que nous n'utilisons pas le *polymorphisme «F-bounded»* [26, 34]. Dans les expressions, les variables de types liées sont annotées avec leurs sortes. Ces annotations seront omises quand la sorte des variables est claire d'après le contexte.

Comme on peut définir des opérateurs de types, comme $\Lambda t.(t \to t)$ par exemple, c'est-à-dire des types d'ordre supérieur, il nous faut augmenter la définition des sortes.

Définition 14.2 (Sortes)

κ,χ	::=	$\mathbb{T}$	sorte des types
	|	$\mathbb{R}$	sorte des rangées
	|	$(\kappa \to \chi)$	sortes des opérateurs

Les sortes de base sont celles données dans la définition 11.2, et on ajoute les sortes d'ordre supérieur: $(\kappa \to \chi)$. Ainsi le type $(\Lambda t^{\mathbb{T}}.[\,put : t, get : t \to \mathsf{o}\,])$, qu'on appelle aussi une *interface*, a la sorte $(\mathbb{T} \to \mathbb{R})$. On introduit une notation spéciale pour la sorte des interfaces $\mathbb{I} = (\mathbb{T} \to \mathbb{R})$. Le choix du nom interface se comprend puisqu'il s'agit d'une fonction qui, étant donné un type, renvoie une association entre champs et types (plus exactement un enregistrement). On utilisera la notion d'interface dans le codage des objets à la partie IV.

En ce qui concerne les environnements de typage, on rajoute deux types d'associations entre variables de types et types: $(t \sqsubseteq \tau)$ et $(s \leqslant t :: \kappa)$. Le premier type d'association permet d'ajouter une contrainte de sous-type à une variable: la variable t ne pourra être instanciée que par des sous-types de τ. Le second type d'association permet de sous-typer la récursion. Dans le sens où elle évite de faire appel à un méta-jugement: $s \leqslant t \Rightarrow \sigma \leqslant \tau$, comme on l'a fait dans la règle (sub rec) de la figure 12.1.

Définition 14.3 (Environnements) On rajoute deux associations aux environnements de typage définis dans la définition 11.3

$$\Gamma,\Delta ::= \ldots \mid \Gamma, t \sqsubseteq \tau \mid \Gamma, s \leqslant t :: \kappa$$

Intuitivement, les hypothèses de la forme $t \sqsubseteq \tau$ servent à construire les types de la forme $(\forall t \sqsubseteq \tau.\sigma)$, tandis que les hypothèses de la forme $s \leqslant t :: \kappa$ servent à montrer les jugements $\mu s.\tau \leqslant \mu t.\sigma$. On omet les informations de sortes quand celles-ci sont claires dans le contexte. La définition du domaine d'un environnement $\mathbf{dom}(\Gamma)$, est celle de la définition 11.3 augmentée de la relation $\mathbf{dom}(\Gamma,t \sqsubseteq \tau) =_{\text{def}} \mathbf{dom}(\Gamma,s \leqslant t :: \kappa) =_{\text{def}} \mathbf{dom}(\Gamma)$.

On peut imaginer remplacer la contrainte $(s \leqslant t :: \kappa)$, par la contrainte plus générale $(t \leqslant \tau)$, et rajouter un opérateur de quantification $(\forall t \leqslant \tau.\sigma)$ à la syntaxe des types. Ce système aurait l'avantage de faire jouer un rôle symétrique aux deux relations de sous-typage $\sqsubseteq$ et $\leqslant$. Néanmoins cette complication du système n'est pas nécessité par notre étude, aussi nous n'introduisons pas ces modifications dans notre système de types.

La définition de la relation $\sqsubseteq$ est donnée dans la section suivante, ainsi que les définitions des jugements indiquant qu'un environnement est bien formé: $\Gamma \vdash *$, qu'un type est bien sorté: $\Gamma \vdash \tau :: \kappa$, et qu'un terme est bien typé: $\Gamma \vdash P : \tau$.

14.2 Système de types et sous-typage

Dans la figure 14.1, nous présentons les règles qui définissent les environnements et les types bien formés. Ces règles sont assez usuelles et ne diffèrent pas beaucoup de celles données dans la figure 11.1. Il faut noter que, dans l'environnement $\Gamma,x : \tau,\Gamma'$, la variable x a toujours un type de

$$\emptyset \vdash * \qquad \frac{\Gamma \vdash \tau :: \mathbb{T} \quad x \notin \mathbf{dom}(\Gamma)}{\Gamma,x : \tau \vdash *} \qquad \frac{\Gamma \vdash * \quad t \notin \mathbf{fn}(\Gamma)}{\Gamma,t :: \kappa \vdash *}$$

$$\frac{\Gamma \vdash * \quad s \neq t \quad s,t \notin \mathbf{fn}(\Gamma)}{\Gamma,s \leqslant t :: \kappa \vdash *} \qquad \frac{\Gamma \vdash \tau :: \kappa \quad t \notin \mathbf{fn}(\Gamma)}{\Gamma,t \sqsubseteq \tau \vdash *}$$

$$\frac{\Gamma \vdash \tau :: \mathbb{R}}{\Gamma \vdash \tau :: \mathbb{T}} \qquad \frac{\Gamma \vdash *}{\Gamma \vdash [\,] :: \mathbb{R}} \qquad \frac{\Gamma \vdash *}{\Gamma \vdash \mathsf{o} :: \mathbb{T}} \qquad \frac{\Gamma \vdash * \quad (t :: \kappa) \in \Gamma}{\Gamma \vdash t :: \kappa}$$

$$\frac{\Gamma \vdash \tau :: \kappa \quad (t \sqsubseteq \tau) \in \Gamma}{\Gamma \vdash t :: \kappa} \qquad \frac{\Gamma \vdash * \quad (s \leqslant t :: \kappa) \in \Gamma}{\Gamma \vdash s :: \kappa} \qquad \frac{\Gamma \vdash * \quad (s \leqslant t :: \kappa) \in \Gamma}{\Gamma \vdash t :: \kappa}$$

$$\frac{\Gamma \vdash \tau :: \mathbb{T} \quad \Gamma \vdash \sigma :: \mathbb{T}}{\Gamma \vdash (\tau \to \sigma) :: \mathbb{T}} \qquad \frac{\Gamma \vdash \varrho :: \mathbb{R} \quad \Gamma \vdash \tau :: \mathbb{T}}{\Gamma \vdash [\,\varrho\,,l : \tau\,] :: \mathbb{R}}$$

$$\frac{\Gamma,t :: \kappa \vdash \tau :: \kappa \quad t \notin \mathbf{fn}(\Gamma)}{\Gamma \vdash (\mu t^{\kappa}.\tau) :: \kappa} \qquad \frac{\Gamma,t :: \kappa \vdash \tau :: \chi \quad t \notin \mathbf{fn}(\Gamma)}{\Gamma \vdash (\Lambda t^{\kappa}.\tau) :: \kappa \to \chi}$$

$$\frac{\Gamma \vdash \tau :: \kappa \to \chi \quad \Gamma \vdash \sigma :: \kappa}{\Gamma \vdash (\tau\sigma) :: \chi} \qquad \frac{\Gamma,t \sqsubseteq \tau \vdash \sigma :: \mathbb{T} \quad t \notin \mathbf{fn}(\Gamma)}{\Gamma \vdash (\forall t \sqsubseteq \tau\,.\,\sigma) :: \mathbb{T}}$$

Fig. 14.1: Environnement bien formés et système de sortes

sorte $\mathbb{T}$. De plus, dans la règle de formation de l'environnement $\Gamma,t \sqsubseteq \tau$, les conditions $\Gamma \vdash \tau :: \kappa$ et $t \notin \mathbf{fn}(\Gamma)$, impliquent que t n'est pas libre dans τ. Une contrainte de sous-typage n'est donc jamais récursive.

Il est clair que notre système de types est «équivalent» au λ-calcul simplement typé avec récursion (et enregistrements), et que le système de sortes est similaire au système de types simples de Curry. Il existe une différence cependant, qui est l'existence de deux constantes de sortes $\mathbb{R}$ et $\mathbb{T}$, en relation de sous typage: $\Gamma \vdash \tau :: \mathbb{R}$ implique $\Gamma \vdash \tau :: \mathbb{T}$.

Il est possible d'utiliser cette analogie entre notre système de types et $\mathbf{\Lambda}$, pour prouver certains résultats sur $\mathtt{BF}_{\leqslant}$. On peut par exemple définir une relation d'égalité entre types $\sim$, qui correspond à la béta-conversion, et prouver une propriété équivalente à la préservation du typage pour le système de sortes.

Définition 14.4 (Égalité des types) On note $\rightarrow_{\beta\mu}$ la relation de réduction sur les types telle que

$$((\Lambda t^{\kappa}.\tau)\sigma) \rightarrow_{\beta\mu} \tau\{\sigma/t\} \qquad (\mu t^{\kappa}.\tau) \rightarrow_{\beta\mu} \tau\{\mu t^{\kappa}.\tau/t\}$$

On note $\sim$ la plus petite congruence qui contient la $\beta\mu$-conversion. On montre, par analogie avec le λ-calcul simplement typé, que cette notion d'équivalence préserve la sorte des types. C'est-à-dire que $\Gamma \vdash \tau :: \kappa$ et $\tau \sim \sigma$, implique $\Gamma \vdash \sigma :: \kappa$.

On peut également prouver que cette relation de réduction est Church-Rosser.

Lemme 14.1 (La conversion entre types est Church-Rosser) *Soit τ et σ deux expressions de types, alors $\tau \sim \sigma$ si et seulement s'il existe un type ϑ tel que $\tau \xrightarrow{*}_{\beta\mu} \vartheta$ et $\sigma \xrightarrow{*}_{\beta\mu} \vartheta$.*

Preuve On prouve ce résultat par analogie avec PCF [12]. Une preuve directe, basée sur la technique de preuve de ce résultat dans le λ-calcul pur, n'est pas très compliquée. □

Cette dernière propriété implique par exemple que si $\tau \sim (\forall t \sqsubseteq \sigma\,.\,\tau)$, alors $\tau \not\sim [\,\varrho\,, l : \eta\,]$ et $\tau \not\sim (\varrho \rightarrow \eta)$. Plus précisément, on ne peut pas identifier entre eux les types fonctionnels, les types enregistrements et les types quantifiés. Nous nous servirons de cette propriété dans la preuve de conservation du typage, dans le chapitre 15.

14.2.1 Sous-typage en largeur

La relation $\sqsubseteq$, est définie dans la figure 14.2. On qualifie cette relation de sous-typage en largeur, car elle vérifie une propriété équivalente à la propriété 12.1.

Proposition 14.2 *Soit I et J deux ensembles finis d'indices tels que $I \subseteq J$ et $\Gamma \vdash [\,l_i : {\tau_i}^{i\in J}\,] :: \mathbb{R}$, alors: $\Gamma \vdash [\,l_i : {\tau_i}^{i\in J}\,] \sqsubseteq [\,l_i : {\tau_i}^{i\in I}\,]$.*

Preuve La preuve se fait par induction sur la taille de $J \setminus I$. □

Nous montrons aussi, dans la section 15.2.2, que le jugement $\Gamma \vdash [\,\varrho\,, l : \tau\,] \sqsubseteq [\,l : \sigma\,]$, implique que $\tau \sim \sigma$. Ainsi la règle (wsubt updt), qui est la règle la moins usuelle dans la définition de la figure 14.2, peut s'interpréter de la manière suivante: si l'enregistrement σ à un champ l de type τ, alors on peut l'étendre avec $\{l : \tau\}$ sans le modifier.

Intuitivement, la relation de sous-typage en largeur est équivalente, sur les types enregistrements de la forme $[\,l_i : {\tau_i}^{i\in J}\,]$, à la relation de *matching* introduite par K. Bruce dans [25] (voir équation 20.10 (p. 174)). Cependant ces deux relations ne sont pas comparables car, dans notre formalisme, nous pouvons comparer des types de la forme $[\,t\,, l : \tau\,]$ – où t est une variable de sorte $\mathbb{R}$ – ou $[\,[\,\varrho\,, l : \tau\,]\,, l : \tau'\,]$.

14.2.2 Sous-typage général

Si on compare la relation $\sqsubseteq$, avec la relation de sous-typage $\leqslant$ définie au chapitre 12, on peut faire deux remarques.

- la relation $\sqsubseteq$ permet de poser des contraintes plus précises. Ainsi, supposons que P soit un terme de type τ. La contrainte $\tau \sqsubseteq [\,l : \sigma\,]$, signifie que P est un enregistrement qui possède un champ l de type σ. Alors que la contrainte $\tau \leqslant [\,l : \sigma\,]$, signifie que P est un enregistrement qui possède un champ l dont le type est «plus précis» que σ;

$$\frac{\Gamma \vdash * \quad (s \sqsubseteq t) \in \Gamma}{\Gamma \vdash s \sqsubseteq t} \text{ (wsub ax)} \qquad \frac{\Gamma \vdash \tau :: \kappa \quad \tau \sim \sigma}{\Gamma \vdash \tau \sqsubseteq \sigma} \text{ (wsub eq)}$$

$$\frac{\Gamma \vdash \tau \sqsubseteq \vartheta \quad \Gamma \vdash \vartheta \sqsubseteq \sigma}{\Gamma \vdash \tau \sqsubseteq \sigma} \text{ (wsub trans)} \qquad \frac{\Gamma \vdash \sigma :: \mathbb{R}}{\Gamma \vdash \sigma \sqsubseteq [\,]} \text{ (wsub void)}$$

$$\frac{\Gamma \vdash \sigma \sqsubseteq [\,l : \tau\,]}{\Gamma \vdash [\,\sigma\,, l : \tau\,] \sqsubseteq \sigma} \text{ (wsub updt)} \qquad \frac{\Gamma \vdash [\,\sigma\,, l : \tau\,] :: \mathbb{R} \quad \Gamma \vdash \sigma \sqsubseteq \sigma'}{\Gamma \vdash [\,\sigma\,, l : \tau\,] \sqsubseteq [\,\sigma'\,, l : \tau\,]} \text{ (wsub cong)}$$

$$\frac{\Gamma \vdash [\,[\,\varrho\,, k : \vartheta\,]\,, l : \tau\,] :: \mathbb{R} \qquad k \neq l}{\Gamma \vdash [\,[\,\varrho\,, k : \vartheta\,]\,, l : \tau\,] \sqsubseteq [\,[\,\varrho\,, l : \tau\,]\,, k : \vartheta\,]} \text{ (wsub swap)}$$

$$\frac{\Gamma, t :: \kappa \vdash \tau \sqsubseteq \sigma}{\Gamma \vdash \Lambda t^{\kappa}.\tau \sqsubseteq \Lambda t^{\kappa}.\sigma} \text{ (wsub lam)} \qquad \frac{\Gamma \vdash (\tau\sigma) :: \kappa \quad \Gamma \vdash \tau \sqsubseteq \tau'}{\Gamma \vdash (\tau\sigma) \sqsubseteq (\tau'\sigma)} \text{ (wsub app)}$$

Fig. 14.2: Sous-typage en largeur: $\sqsubseteq$

– la relation $\sqsubseteq$ n'est pas intéressante pour le sous-typage des termes, car elle n'est pas assez générale.

Pour ces raisons, on utilise $\sqsubseteq$ dans la quantification ($\forall t \sqsubseteq \tau.\sigma$), car on veut imposer les contraintes les plus fines possible sur les types, et on définit une relation $\leqslant$ plus générale pour sous-typer les termes. Cette relation est définie dans la figure 14.3. La différence principale avec la relation

$$\frac{\Gamma \vdash * \quad (t \leqslant \tau) \in \Gamma}{\Gamma \vdash t \leqslant \tau} \text{ (sub ax)} \qquad \frac{\Gamma \vdash \tau \sqsubseteq \sigma}{\Gamma \vdash \tau \leqslant \sigma} \text{ (sub width)}$$

$$\frac{\Gamma \vdash \tau \leqslant \vartheta \quad \Gamma \vdash \vartheta \leqslant \sigma}{\Gamma \vdash \tau \leqslant \sigma} \text{ (sub trans)} \qquad \frac{\Gamma \vdash [\,\sigma\,, l : \tau\,] :: \mathbb{R} \quad \Gamma \vdash \sigma \leqslant \sigma' \quad \Gamma \vdash \tau \leqslant \tau'}{\Gamma \vdash [\,\sigma\,, l : \tau\,] \leqslant [\,\sigma'\,, l : \tau'\,]} \text{ (sub over)}$$

$$\frac{\Gamma, s :: \kappa \vdash \sigma :: \kappa \quad \Gamma, t :: \kappa \vdash \tau :: \kappa \quad \Gamma, t \leqslant s :: \kappa \vdash \tau \leqslant \sigma}{\Gamma \vdash \mu t^{\kappa}.\tau \leqslant \mu s^{\kappa}.\sigma} \text{ (sub rec)} \qquad \frac{\Gamma \vdash \tau \to \sigma :: \mathbb{T} \quad \Gamma \vdash \tau' \leqslant \tau \quad \Gamma \vdash \sigma \leqslant \sigma'}{\Gamma \vdash \tau \to \sigma \leqslant \tau' \to \sigma'} \text{ (sub arrow)}$$

$$\frac{\Gamma, t :: \kappa \vdash \tau \leqslant \sigma}{\Gamma \vdash \Lambda t^{\kappa}.\tau \leqslant \Lambda t^{\kappa}.\sigma} \text{ (sub lam)} \qquad \frac{\Gamma \vdash (\tau\sigma) :: \kappa \quad \Gamma \vdash \tau \leqslant \tau'}{\Gamma \vdash (\tau\sigma) \leqslant (\tau'\sigma)} \text{ (sub app)}$$

Fig. 14.3: Sous-typage: $\leqslant$

homonyme du chapitre 12, consiste en l'ajout de conditions annexes qui permettent d'assurer que les types sont bien sortés. On remarque que les hypothèses de la règle (sub rec) impliquent que, dans le jugement $\Gamma \vdash \mu t^{\kappa}.\tau \leqslant \mu s^{\kappa}.\sigma$, la variable t n'apparaît pas dans σ et s n'apparaît pas dans τ.

14.3 Typage des termes

La base du système $\mathtt{BF}_{\leqslant}$, qui est défini dans la figure 14.4, est le système de types simples `B`, donné à la section 11.2. Pour obtenir le système $\mathtt{BF}_{\leqslant}$, il suffit d'ajouter quatre nouvelles règles à `B` :

– la règle d'affaiblissement, cf. proposition 11.1;
– une règle pour sous-typer les termes, comme dans $\mathtt{B}_{\leqslant}$, qui utilise la relation $\leqslant$;

– deux règles pour généraliser et instancier un type, qui utilisent les contraintes $t \sqsubseteq \tau$.

$$\frac{\Gamma \vdash *\quad (u:\sigma)\in\Gamma}{\Gamma \vdash u:\sigma}\ \text{(type ax)} \qquad \frac{\Gamma,x:\tau \vdash P:\sigma\quad x\notin \mathbf{dom}(\Gamma)}{\Gamma \vdash (\boldsymbol{\lambda} x)P:\tau\to\sigma}\ \text{(type abs)}$$

$$\frac{\Gamma \vdash P:\tau\to\sigma\quad \Gamma\vdash u:\tau}{\Gamma\vdash (Pu):\sigma}\ \text{(type app)} \qquad \frac{\Gamma\vdash P:[\,l:\sigma\,]}{\Gamma\vdash (P\cdot l):\sigma}\ \text{(type sel)}$$

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

$$\frac{\Gamma,u:\tau\vdash P:\sigma\quad u\notin\mathbf{dom}(\Gamma)}{\Gamma\vdash (\boldsymbol{\nu} u)P:\sigma}\ \text{(type new)} \qquad \frac{\Gamma\vdash P:\mathtt{o}\quad \Gamma\vdash Q:\sigma}{\Gamma\vdash (P\mid Q):\sigma}\ \text{(type par)}$$

$$\frac{\Gamma\vdash P:\tau\quad (u:\tau)\in\Gamma}{\Gamma\vdash \langle u\Leftarrow P\rangle:\mathtt{o}}\ \text{(type decl)} \qquad \frac{\Gamma\vdash P:\tau\quad (u:\tau)\in\Gamma}{\Gamma\vdash \langle u=P\rangle:\mathtt{o}}\ \text{(type mdecl)}$$

$$\frac{\Gamma\vdash \tau\leqslant\sigma\quad \Gamma\vdash P:\tau}{\Gamma\vdash P:\sigma}\ \text{(type sub)} \qquad \frac{\Gamma\vdash P:\sigma\quad \Gamma,\Gamma'\vdash *}{\Gamma,\Gamma'\vdash P:\sigma}\ \text{(type weak)}$$

$$\frac{\Gamma\vdash \tau'\sqsubseteq\tau\quad \Gamma\vdash P:(\forall t\sqsubseteq\tau\,.\,\sigma)}{\Gamma\vdash P:\sigma\{\tau'/t\}}\ \text{(type inst)} \qquad \frac{\Gamma,t\sqsubseteq\tau\vdash P:\sigma}{\Gamma\vdash P:(\forall t\sqsubseteq\tau\,.\,\sigma)}\ \text{(type gen)}$$

Fig. 14.4: Système de types avec polymorphisme contraint: $\mathtt{BF}_{\leqslant}$

Il existe dans la littérature (informatique) de nombreux systèmes de types similaires à $\mathtt{BF}_{\leqslant}$, utilisés dans le typage de langages à objets. On parle ici du système restreint à la partie fonctionnelle du calcul. On est donc dans un cas similaire à celui de $\mathtt{F}_{\leq}$. Mais, contrairement à $\mathtt{F}_{\leq}$, le système $\mathtt{BF}_{\leqslant}$ n'a jamais été étudié en lui-même, ni formalisé. En particulier il n'existe pas de preuves de préservation du typage. On trouve beaucoup de systèmes de types qui allient types d'ordre supérieur et récursion, et qui sont utilisés pour donner des considérations générale sur le typage des objets: comme comparer différents systèmes de types entre eux, par exemple. Ainsi, $\mathtt{BF}_{\leqslant}$ est similaire au système de types utilisé dans [24], pour comparer divers codages des calculs d'objets. Une description plus complète de ce système, sans la récursion, peut être trouvée dans [106]. Il y a quelques différences cependant: nous n'avons pas de type le plus grand $\top(\kappa)$, ni de type existentiel, par contre nous utilisons la récursion sans restriction sur la sorte des types. Le système $\mathtt{BF}_{\leqslant}$ est aussi très similaire à celui utilisé par J. Mitchell dans [100], mis à part que nous utilisons la relation de sous-typage en largeur dans la quantification bornée.

Dans la section suivante, nous montrons la propriété de préservation du typage pour $\mathtt{BF}_{\leqslant}$.

Théorème 14.3 (Préservation du typage) *Si* $\Gamma\vdash P:\tau$ *et* $P\to P'$, *alors* $\Gamma\vdash P':\tau$.

La preuve de ce théorème est longue et assez technique, de plus la méthode de preuve employée diffère sensiblement de la technique utilisée, pour la même propriété, dans les systèmes avec types d'ordre supérieur. Plus précisément, dans les systèmes qui possèdent une règle de conversion entre types, comme (wsub eq).

Dans ces preuves, comme par exemple pour $\mathtt{F}^{\omega}_{\wedge}$ [33] et les systèmes de types pour les calculs d'objets [3, chapitre 20] [49, chapitre 7], on utilise un système intermédiaire simplifié, dans lequel les types sont sous forme normale. Ceci permet de se passer de la règle – problématique – de béta-réduction.

Comme nous avons un opérateur de récursion dans les types de $\mathtt{BF}_{\leqslant}$, cette méthode de preuve n'est pas utilisable: il n'y a pas de résultat de *normalisation forte* pour le λ-calcul simplement typé avec récursion. Néanmoins, comme nous n'avons pas de règle de sous-typage entre les type

quantifiés (l'équivalent de la règle de FUN [28]) il nous est possible de donner une preuve directe, qui utilise un raisonnement sur la forme des termes.

14.4 Sûreté du typage

Dans cette section, nous définissons formellement la notion d'erreurs d'exécution pour notre calcul, et nous prouvons qu'un terme bien typé dans $\mathtt{BF}_{\leqslant}$ ne peut pas provoquer d'erreur.

Comme nous l'avons dit précédemment, la présence des enregistrements dans la syntaxe des termes de $\boldsymbol{\pi}^\star$ suffit pour introduire une notion de processus erreurs. Ainsi nous n'avons pas besoin, comme dans le λ-calcul, d'introduire de nouvelles constantes de types – comme `int`,`bool`,... – et des opérateurs primitifs sur les structures de données. Ces erreurs sont, par exemple, des processus qui contiennent une sélection à un champ manquant dans un enregistrement, comme dans le terme $([\,]\cdot l)$. D'autres formes d'erreurs sont, par exemple, la sélection sur une fonction: $(\!(\boldsymbol{\lambda} x)\!)P)\cdot l$, l'application sur un enregistrement: $[\,P\,, l = Q\,]a$, l'application ou la sélection d'une ressource.

Définition 14.5 (Erreurs) Une *erreur élémentaire* est un processus engendré par la grammaire suivante, où P est un processus quelconque:

$$\begin{array}{rcl} E & ::= & ([\,]\cdot l) \mid ([\,]\,a) \mid ([\,P\,, l = Q\,]\,a) \mid (\!(\boldsymbol{\lambda} x)\!)P)\cdot l \\ & \mid & (\langle u \Leftarrow P\rangle\, a) \mid (\langle u \Leftarrow P\rangle \cdot l) \mid (\langle u = P\rangle\, a) \mid (\langle u = P\rangle \cdot l) \end{array}$$

Une *erreur* est un processus qui contient au moins une erreur élémentaire. Ainsi P est une erreur si il existe un contexte $\mathbf{C}$, et une erreur élémentaire E, tels que $P \equiv \mathbf{C}[E]$. Finalement, on dit que le processus P est *sûr*, si pour tout processus Q tel que $P \xrightarrow{*} Q$, on a Q n'est pas une erreur.

Il est facile de montrer qu'une erreur élémentaire ne peut jamais être typée, c'est-à-dire que pour toute erreur élémentaire E, il n'existe pas d'environnement Γ, et de type τ tel que $\Gamma \vdash E : \tau$. Par conséquent, une erreur n'est pas un processus typable. Or, d'après le théorème 14.3, un processus conserve son type au cours d'une réduction. Par conséquent un processus bien typé ne peut pas se réduire en un processus erreur

Théorème 14.4 (Sûreté du typage) *Si $\Gamma \vdash P : \tau$ et $P \xrightarrow{*} Q$, alors Q n'est pas une erreur.*

Avant de passer à la preuve de la propriété de conservation du typage pour $\mathtt{BF}_{\leqslant}$, il est intéressant de voir comment ce système peut être modifié. Tout d'abord, il est possible d'étendre la syntaxe des types avec un type «le plus grand» $\top$. On ne considère cependant pas des types $\top$ pour chaque sortes, contrairement aux types $\top(\kappa)$ introduit par B. Pierce [106], mais un seul type, qui à la sorte $\mathbb{T}$. Il faut noter que dans les systèmes de types d'ordre supérieur, on ne considère pas de type le plus petit – qu'on peut interpréter par $(\forall t \sqsubseteq \top\,.\,t)$ – car ceci peut introduire des inconsistances dans le système de types.

Nous ne montrons pas dans cette thèse que le système $\mathtt{BF}_{\leqslant}$, étendu avec $\top$, vérifie la propriété de conservation du typage. Disons simplement que l'ajout du type $\top$ ne complique par outre mesure la preuve donnée au chapitre suivant. La différence principale est qu'on peut avoir un sous-typage non trivial avec les types quantifiés, c'est-à-dire qu'on peut montrer que $(\forall t \sqsubseteq \tau\,.\,\sigma) \leqslant \top$.

Une deuxième généralisation possible est d'ajouter un opérateur de quantification borné pour la relation de sous-typage la plus générale $\leqslant$, c'est-à-dire un opérateur de la forme $(\forall t \leqslant \tau\,.\,\sigma)$. Encore une fois, ceci ne bouleverse pas la preuve de la propriété de conservation du typage. Cependant, comme nous allons utiliser le système $\mathtt{BF}_{\leqslant}$– dans cette thèse – uniquement pour définir l'interprétation du type des objets extensibles (cf. chapitre 20), nous nous restreignons à l'étude des opérateurs qui sont directement nécessaires à cette tâche.

CHAPITRE 15

Preuve de la préservation du typage

Ce chapitre est entièrement dévolu à la preuve du théorème 14.3. On utilisera le symbole $\trianglelefteqslant$, pour désigner soit $\leqslant$, soit $\sqsubseteq$, dans les contextes où on prouve des résultats qui sont vrais pour ces deux relations. On utilisera la notation $\Gamma \vdash \mathcal{J}$, pour représenter les jugements de la forme $\Gamma \vdash *$, ou $\Gamma \vdash \tau :: \kappa$, ou $\Gamma \vdash \sigma \trianglelefteqslant \tau$, ou $\Gamma \vdash P : \tau$.

Nous supposons que tous les jugements, les environnements et les types utilisés sont bien formés. Cette hypothèse n'est pas exagérée, en effet on montre le résultat suivant.

Lemme 15.1 *Si $\Gamma \vdash \mathcal{J}$, alors Γ est bien formé, c'est-à-dire que $\Gamma \vdash *$. De plus, si $\Gamma \vdash \sigma \trianglelefteqslant \tau$, alors σ et τ ont la même sorte, c'est-à-dire qu'il existe une sorte κ telle que $\Gamma \vdash \sigma :: \kappa$ et $\Gamma \vdash \tau :: \kappa$.*

On peut également montrer que la règle d'affaiblissement est valide pour tous les jugements. C'est-à-dire que, si $\Gamma \vdash \mathcal{J}$ et $\Gamma,\Gamma' \vdash *$, alors $\Gamma,\Gamma' \vdash \mathcal{J}$. La preuve de ce résultat nécessite de considérer les jugements modulo α-conversion des types et de montrer une propriété de renommage des variables. Nous omettons cette preuve ici.

Avant de s'intéresser à la preuve de la propriété de préservation du typage, nous montrons un ensemble de propriétés préliminaires. Ces lemmes, et particulièrement les lemmes de substitutions, sont classiques dans les preuves pour les systèmes de types avec polymorphisme. Par contre, les propriétés «d'analyse», prouvées à la section 15.2 et 15.3, sont moins usuelles.

15.1 Propriétés de substitutions

Nous commençons par montrer que la sorte d'un type est préservée par substitution. Nous ne définissons pas la notion de substitution pour les types et les environnements: $\sigma\{\tau/t\}$ et $\Gamma\{\tau/t\}$. Les détails de ces opérations sont standards. Disons néanmoins qu'on suppose avoir renommé les variables de types liées par une récursion, une abstraction, ou une quantification bornée, afin d'éviter les captures.

Lemme 15.2 (Lemme de substitution pour les sortes)

- *Si $\Gamma,t :: \kappa,\Gamma' \vdash *$ et $\Gamma \vdash \tau :: \kappa$, alors $\Gamma,\Gamma'\{\tau/t\} \vdash *$;*
- *si $\Gamma,t :: \kappa,\Gamma' \vdash \sigma :: \chi$ et $\Gamma \vdash \tau :: \kappa$, alors $\Gamma,\Gamma'\{\tau/t\} \vdash \sigma\{\tau/t\} :: \chi$;*
- *si $\Gamma,t \sqsubseteq \vartheta,\Gamma' \vdash \sigma :: \chi$, et $\Gamma \vdash \vartheta :: \kappa$, et $\Gamma \vdash \tau :: \kappa$, alors $\Gamma,\Gamma'\{\tau/t\} \vdash \sigma\{\tau/t\} :: \chi$.*

Preuve La preuve est faite par induction sur l'inférence de $\Gamma,t :: \kappa,\Gamma' \vdash \mathcal{J}$ et de $\Gamma,t \sqsubseteq \vartheta,\Gamma' \vdash \sigma :: \chi$. □

Nous prouvons ensuite une résultat équivalent pour les autres jugements.

Lemme 15.3 (Lemme de substitution)

(*i*) *Si* $\Gamma,t \sqsubseteq \tau,\Gamma' \vdash \mathcal{J}$, *et* $\Gamma \vdash \tau' \sqsubseteq \tau$, *alors* $\Gamma,\Gamma'\{\tau'/t\} \vdash \mathcal{J}\{\tau'/t\}$;
(*ii*) *si* $\Gamma,s \leqslant t :: k,\Gamma' \vdash \mathcal{J}$, *et* $\Gamma \vdash \tau' \leqslant \tau$, *et* $\Gamma \vdash \tau :: k$, *alors* $\Gamma,\Gamma'\{\tau'/s\}\{\tau/t\} \vdash \mathcal{J}\{\tau'/s\}\{\tau/t\}$;
(*iii*) *si* $\Gamma,x : \tau,\Gamma' \vdash P : \sigma$ *et* $\Gamma,\Gamma' \vdash y : \tau$, *alors* $\Gamma,\Gamma' \vdash P\{y/x\} : \sigma$.

Preuve On prouve la première propriété par induction sur l'inférence du jugement $\Gamma,t \sqsubseteq \tau,\Gamma' \vdash \mathcal{J}$. Nous laissons de coté la preuve pour les jugements de sortes et de types, et nous n'étudions que les jugements de sous-typage. Pour la suite de cette preuve, Δ désigne l'environnement $\Gamma,t \sqsubseteq \tau,\Gamma'$, et Δ' dénote $\Gamma,\Gamma'\{\tau'/t\}$. On suppose que $\Delta \vdash \vartheta \trianglelefteqslant \sigma$ et que $\Gamma \vdash \tau' \sqsubseteq \tau$, et on prouve que $\Delta \vdash \vartheta\{\tau'/t\} \trianglelefteqslant \sigma\{\tau'/t\}$. Cette preuve se ramène à une étude de cas sur la dernière règle utilisée dans l'inférence de $\Delta \vdash \vartheta \trianglelefteqslant \sigma$.

- **cas (wsub ax):** le type ϑ est égal à la variable s, et on a $(s \sqsubseteq \sigma) \in \Delta$ implique $\Delta \vdash s \sqsubseteq \sigma$. Le cas intéressant est celui où s est égal à t. Par hypothèse, on a $\Gamma \vdash \tau' \sqsubseteq \tau$ et par conséquent, en utilisant l'affaiblissement, on peut prouver que $\Gamma,\Gamma'\{\tau'/t\} \vdash \tau' \sqsubseteq \tau$. La preuve est similaire dans le cas (sub ax);
- **cas (wsub eq):** il existe une sorte κ telle que $\Delta \vdash \vartheta :: \kappa$, et $\vartheta \sim \sigma$. Par analogie avec le λ-calcul, on peut montrer que $\vartheta \sim \sigma$ implique $\vartheta\{\tau'/t\} \sim \sigma\{\tau'/t\}$. Le résultat est obtenu en utilisant la règle (wsub eq), et le lemme 15.2, qui sert à prouver que $\vartheta\{\tau'/t\}$ est bien sorté;
- **cas (wsub trans) et (sub trans):** il suffit d'utiliser l'hypothèse d'induction deux fois;
- **cas (wsub void):** le type σ est égal à $[\,]$ et, avec l'hypothèse $\Delta \vdash \vartheta :: \mathbb{R}$, on a $\Delta \vdash \vartheta \sqsubseteq [\,]$. Il suffit d'utiliser le lemme 15.2 pour prouver que $\Delta' \vdash \vartheta\{\tau'/t\} :: \mathbb{R}$ et la règle (wsub void). La preuve est similaire dans les cas (wsub updt), (wsub cong), (wsub swap), (wsub lam), (wsub app), (sub over), (sub arrow) et (sub rec).

Les propriétés (*ii*) et (*iii*) sont prouvées de manière similaire. □

Dans cette preuve on démontre un résultat plus fort pour un cas particulier de la propriété (*i*): si ($\sharp$) $\Gamma,t \sqsubseteq \tau,\Gamma' \vdash P : \tau$, et $\Gamma \vdash \tau' \sqsubseteq \tau$, alors ($\flat$) $\Gamma,\Gamma'\{\tau'/t\} \vdash P : \tau$, et on a une preuve pour le jugement ($\flat$), qui a moins d'occurences des règles (type inst) et (type gen) que la preuve de ($\sharp$).

Le lemme de substitutions permet de simplifier les jugements d'inférence de type. En effet, si on compose les règles (type gen) et (type inst), on obtient une règle dérivée (type cut), telle que:

$$\frac{\Gamma,t \sqsubseteq \tau \vdash P : \sigma \qquad \Gamma \vdash \tau' \sqsubseteq \tau}{\Gamma \vdash P : \sigma\{\tau'/t\}} \text{ (type cut)}$$

Nous montrons que, pour tout jugement de types, il est possible de trouver une inférence de type «sans cut». C'est un résultat qui ressemble à l'élimination des coupures dans les systèmes logiques, bien que sa preuve soit beaucoup plus simple.

Corollaire 15.4 *Tout jugement de type valide possède une preuve où aucune utilisation de (type inst) ne suit immédiatement une utilisation de (type gen).*

Preuve Il suffit d'utiliser le lemme 15.3-(*i*) pour remplacer l'utilisation de la règle dérivée (type cut). Ce processus termine, en effet on n'introduit pas de nouvelles utilisation des règles (type inst) et (type gen). □

On montre en fait un résultat plus fort: à partir d'une preuve de $\Gamma \vdash P : \tau$, on peut construire une preuve qui vérifie la condition du corollaire 15.4, et qui utilise moins les règles (type inst) et (type gen). On peut aussi montrer diverses propriétés sur les environnements.

Lemme 15.5 (Renommage des variables) *On peut renommer les variables définies dans un environnement: si* $\Gamma \vdash P : \sigma$, *et* $y \notin \mathbf{dom}(\Gamma)$, *alors* $\Gamma\{y/x\} \vdash P\{y/x\} : \sigma$.

Preuve La preuve est faite par induction sur l'inférence de $\Gamma \vdash P : \sigma$. Elle nécessite de montrer tout d'abord que $\Gamma \vdash *$ et $y \notin \mathbf{fn}(\Gamma)$, implique $\Gamma\{y/x\} \vdash *$. □

Lemme 15.6 (Permutation)

- *Si* $\Gamma,t \sqsubseteq \tau,\Gamma' \vdash P : \sigma$, *et* $\Gamma,\Gamma' \vdash *$, *et* $t \notin \mathbf{fn}(\Gamma')$, *alors* $\Gamma,\Gamma',t \sqsubseteq \tau \vdash P : \sigma$;
- *si* $\Gamma,x : \tau,\Gamma' \vdash P : \sigma$, *et* $\Gamma,\Gamma' \vdash *$, *et* $x \notin \mathbf{dom}(\Gamma')$, *alors* $\Gamma,\Gamma',x : \tau \vdash P : \sigma$.

Preuve La preuve est faite par induction sur l'inférence de $\Gamma,t \sqsubseteq \tau,\Gamma' \vdash P : \sigma$ (resp. $\Gamma,x : \tau,\Gamma' \vdash P : \sigma$). On doit montrer préliminairement que $\Gamma,x : \tau,\Gamma' \vdash *$, et $\Gamma,\Gamma' \vdash *$, implique $\Gamma,\Gamma',x : \tau \vdash *$.□

15.2 Analyse des jugements de sous-typage

On peut simplifier l'utilisation des règles (type inst), (type gen), (type weak) et (type sub), dans un jugement de types, de plusieurs manières. Par exemple, on peut substituer plusieurs utilisations de la règle (type sub) par une seule utilisation de cette règle: ceci découle de la transitivité de la relation $\leqslant$. De même pour (type weak). On peut également échanger l'utilisation des règles (type inst) et (type weak), et des règles (type sub) et (type weak).

Une autre simplification est donnée par le corollaire 15.4: on peut simplifier une utilisation de (type gen) suivie par une utilisation de (type inst).

Toutes ces simplifications nous permettent d'écrire les preuves sous une «forme canonique». En particulier, dans toutes les preuves suivantes on considère que les conditions de la définition 15.1 sont vérifiées:

Définition 15.1 (Forme canonique d'une preuve $\Gamma \vdash P : \tau$)

- il n'y a jamais deux utilisations consécutive des règles (type sub) ou (type weak);
- il n'y a jamais utilisation de (type gen) suivie de (type inst);
- on remplace n utilisations successives de la règle (wsub app), par une utilisation de la règle dérivée (wsub n-app) suivante.

$$\frac{\Gamma \vdash (\tau\sigma_1 \ldots \sigma_n) :: \kappa \qquad \Gamma \vdash \tau \sqsubseteq \tau'}{\Gamma \vdash (\tau\sigma_1 \ldots \sigma_n) \sqsubseteq (\tau'\sigma_1 \ldots \sigma_n)} \text{ (wsub n-app)}$$

  et nous supposons qu'il n'y a jamais plusieurs utilisations successives de (wsub n-app). Nous faisons de même avec la règle (sub app).

Dans cette section, on étudie d'autres simplifications possibles. Nous commençons par montrer que les jugements de sous-typage entre types quantifiés sont inutiles. Ce résultat s'explique intuitivement en notant que, mis à part dans la règle (wsub eq), il est impossible de comparer des types quantifiés.

Lemme 15.7 (Sous-typage des types quantifiés)

(*i*) *Si* $\Gamma \vdash \eta_0 \sqsubseteq \eta_1$ *et* $\eta_0 \sim (\forall t \sqsubseteq \tau \,.\, \sigma)$, *alors* $\eta_1 \sim (\forall t \sqsubseteq \tau \,.\, \sigma)$;
(*ii*) *si* $\Gamma \vdash \eta_0 \leqslant \eta_1$ *et* $\eta_0 \sim (\forall t \sqsubseteq \tau \,.\, \sigma)$, *alors* $\eta_1 \sim (\forall t \sqsubseteq \tau \,.\, \sigma)$;
(*iii*) *si* $\Gamma \vdash (\forall t \sqsubseteq \tau \,.\, \sigma) \leqslant (\forall t \sqsubseteq \tau' \,.\, \sigma')$, *alors* $\tau \sim \tau'$ *et* $\sigma \sim \sigma'$.

Preuve On prouve la première propriété par induction sur l'inférence de $\Gamma \vdash \eta_0 \sqsubseteq \eta_1$. La preuve se ramène à une étude de cas sur la dernière règle utilisée dans cette inférence. En utilisant le lemme 14.1, on peut montrer que η_0 ne peut pas être une variable, un type fonctionnel ou un enregistrement. On a donc un nombre restreint de cas à étudier.

- **cas (wsub trans):** on utilise l'hypothèse d'induction deux fois, puis la transitivité de la relation $\sim$;

- **cas (wsub n-app):** il y a $n+2$ types: θ_0, θ_1, et $\phi_1, \ldots \phi_n$, tels que $\eta_0 = (\theta_0 \phi_1 \ldots \phi_n)$, et $\eta_1 = (\theta_1 \phi_1 \ldots \phi_n)$, et $\Gamma \vdash \theta_0 \sqsubseteq \theta_1$. On fait une étude de cas sur la règle utilisée avant (wsub n-app) dans l'inférence de $\Gamma \vdash \eta_0 \sqsubseteq \eta_1$.
  - **cas (wsub eq):** on a $\theta_0 \sim \theta_1$, et donc $\eta_0 \sim \eta_1$;
  - **cas (wsub ax):** on a $\eta_0 \sim (t\ \phi_1 \ldots \phi_n)$, où t est une variable de type dans le domaine de Γ. Ce cas est donc impossible, puisqu'il contredit l'hypothèse que $\eta_0 \sim (\forall t \sqsubseteq \tau \,.\, \sigma)$ (cf. lemme 14.1);
  - **cas (wsub trans):** il existe un type θ_2, tel que $\Gamma \vdash \theta_0 \sqsubseteq \theta_2 \sqsubseteq \theta_1$. Par Conséquent, en utilisant la règle (wsub n-app), on obtient deux dérivations plus petites: $\Gamma \vdash \eta_0 \sqsubseteq (\theta_2 \phi_1 \ldots \phi_n)$, et $\Gamma \vdash (\theta_2 \phi_1 \ldots \phi_n) \sqsubseteq \eta_1$. Le résultat suit de l'hypothèse induction;
  - **cas (wsub lam):** il existe une sorte κ telle que $\theta_0 = \Lambda t^k.\theta'_0$, et $\theta_1 = \Lambda t^k.\theta'_1$, et $\Gamma, t :: \kappa \vdash \theta'_0 \sqsubseteq \theta'_1$. Par conséquent, en utilisant le lemme 15.3-(i), on peut exhiber le jugement de sous-typage suivant (dont l'arbre d'inférence est plus petit que celui de $\Gamma \vdash \eta_0 \sqsubseteq \eta_1$): $\Gamma \vdash (\theta'_0\{\phi_1/t\}\phi_2 \ldots \phi_n) \sqsubseteq (\theta'_1\{\phi_1/t\}\phi_2 \ldots \phi_n)$. Le résultat suit en utilisant l'hypothèse d'induction et le fait que $\eta_0 = ((\Lambda t.\theta'_0)\phi_1 \ldots \phi_n) \sim (\theta'_0\{\phi_1/t\}\phi_2 \ldots \phi_n)$.

On prouve maintenant la seconde propriété en utilisant une méthode similaire. Supposons que $\Gamma \vdash \eta_0 \leqslant \eta_1$, et que $\eta_0 \sim (\forall t \sqsubseteq \tau \,.\, \sigma)$. Comme η_0 est équivalent à un type quantifié, la dernière règle ne peut pas être (sub ax), (sub arrow) ou (sub over): c'est une conséquence directe du lemme 14.1.

- **cas (sub trans):** on utilise l'hypothèse d'induction deux fois pour montrer le résultat;
- **cas (sub width):** c'est une conséquence directe de la propriété (i) montré plus haut;
- **cas (sub rec):** il existe deux types, η'_0 et η'_1, tel que $\eta_0 = \mu s^k.\eta'_0$ et $\eta_1 = \mu t^k.\eta'_1$, et $\Gamma, s \leqslant t :: \kappa \vdash \eta'_0 \leqslant \eta'_1$. Une autre condition est que s n'est pas libre dans η'_1 et que t n'est pas libre dans η'_0. En utilisant la règle de $\sim$ pour le dépliage de la récursion, on peut prouver alors que $\eta_0 \sim \eta'_0\{\eta_0/s\}$. on a une égalité similaire pour η_1. Par conséquent, en utilisant le lemme 15.3-(ii), il en suit qu'il existe un jugement $\Gamma \vdash \eta'_0\{\eta_0/s\} \leqslant \eta'_1\{\eta_1/t\}$, de la même taille que $\Gamma, s \leqslant t :: \kappa \vdash \eta'_0 \leqslant \eta'_1$. On peut alors prouver le résultat en utilisant l'hypothèse d'induction;
- **cas (sub n-app):** ce cas est exactement le même que dans la preuve de la propriété (i), ci-dessus, pour le cas (wsub n-app).

La propriété (iii) est une conséquence de la propriété (ii) et du lemme 14.1. □

Nous montrons qu'on peut généraliser le corollaire 15.4, en utilisant le lemme 15.7.

Corollaire 15.8 *Tout jugement de type valide possède une preuve où aucune utilisation de (type inst), ne suit immédiatement l'utilisation des règles (type sub) et (type weak), puis l'utilisation de (type gen).*

Preuve Une première remarque, est que l'on peut remplacer l'utilisation de la règle (type sub) suivie de (type weak), par l'utilisation de la règle (type weak) suivie de (type sub). Par permutation, on peut donc se ramener dans un cas tel qu'on a une utilisation de (type gen), suivie de (type weak), puis de (type sub), puis de (type inst). On se retrouve alors dans le cas suivant – on suppose l'existence d'un environnement Δ, bien formé, qui étend l'environnement Γ –:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, t \sqsubseteq \tau \vdash P : \sigma}}{\Gamma \vdash P : (\forall t \sqsubseteq \tau \,.\, \sigma)}\ (\text{type gen})}{\Delta \vdash P : (\forall t \sqsubseteq \tau \,.\, \sigma)}\ (\text{type weak}) \qquad \dfrac{\vdots}{\Delta \vdash (\forall t \sqsubseteq \tau \,.\, \sigma) \leqslant (\forall t \sqsubseteq \tau' \,.\, \sigma')}}{\dfrac{\Delta \vdash P : (\forall t \sqsubseteq \tau' \,.\, \sigma') \quad (\sharp)}{\Delta \vdash P : \sigma'\{\tau''/t\}}\ (\text{type inst})}\ (\text{type sub})$$

Où ($\sharp$) est un jugement de la forme: $\dfrac{\vdots}{\Delta \vdash \tau'' \sqsubseteq \tau'}$. En utilisant le lemme 15.7-(*iii*), on peut prouver que $\tau \sim \tau'$ et $\sigma \sim \sigma'$, et donc que $\Delta \vdash \tau'' \sqsubseteq \tau$ et $\Gamma, t \sqsubseteq \tau \vdash P : \sigma'$. De plus, si on utilise l'α-conversion pour choisir t différent des variables de Δ, on montre que ce dernier jugement implique que $\Delta, t \sqsubseteq \tau \vdash P : \sigma'$. C'est un équivalent du lemme de permutation (lemme 15.6). On peut donc remplacer l'inférence précédente par l'arbre de preuve suivant:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Delta, t \sqsubseteq \tau \vdash P : \sigma} \quad \sigma \sim \sigma'}{\Delta, t \sqsubseteq \tau \vdash P : \sigma'}}{\Delta \vdash P : (\forall t \sqsubseteq \tau \,.\, \sigma')}\ \text{(type gen)} \qquad \dfrac{(\sharp) \quad \tau \sim \tau'}{\Delta \vdash \tau'' \sqsubseteq \tau}}{\Delta \vdash P : \sigma'\{\tau''/t\}}\ \text{(type inst)}$$

Il suffit alors d'utiliser le corollaire 15.4, pour obtenir le résultat attendu. □

Comme dans le corollaire précédent, on montre en fait un résultat plus fort: à partir d'une preuve du jugement $\Gamma \vdash P : \tau$, on peut construire une preuve utilisant moins les règles (type gen) et (type inst), et qui vérifie les conditions du corollaire 15.8.

En nous basant sur ce dernier résultat, on peut ajouter une condition à la définition de preuve canonique (cf. définition 15.1).

Définition 15.2 (Addendum à la définition 15.1)

– il n'y a jamais utilisation de (type gen), suivie de (type weak) et (type sub), puis de l'utilisation de (type inst).

15.2.1 Analyse des types fonctionnels

Nous montrons que la relation de sous-typage en largeur est «triviale» lorsqu'on ne compare pas des types enregistrements: si $\Gamma \vdash (\sigma_0 \to \tau_0) \sqsubseteq (\sigma_1 \to \tau_1)$, alors $\sigma_0 \sim \sigma_1$, et $\tau_1 \sim \tau_1$. Nous montrons aussi un résultat pour la relation de sous-typage plus générale $\leqslant$: si $\Gamma \vdash (\sigma_0 \to \tau_0) \leqslant (\sigma_1 \to \tau_1)$, alors $\sigma_1 \leqslant \sigma_0$ et $\tau_0 \leqslant \tau_1$.

Lemme 15.9 *Si* $\Gamma \vdash \eta_0 \sqsubseteq \eta_1$, *et* $\eta_0 \sim (\sigma_0 \to \tau_0)$, *alors* $\eta_1 \sim \eta_0$. *Si* $\Gamma \vdash \eta_0 \leqslant \eta_1$, *et* $\eta_0 \sim (\sigma_0 \to \tau_0)$, *alors il existe deux types,* σ_1 *et* τ_1, *tel que* $\eta_1 \sim (\sigma_1 \to \tau_1)$, *et* $\sigma_1 \leqslant \sigma_0$, *et* $\tau_0 \leqslant \tau_1$.

Preuve La preuve de la première propriété est une analyse par cas sur la dernière règle du jugement $\Gamma \vdash \eta_0 \sqsubseteq \eta_1$. Elle est similaire à la preuve du lemme 15.7.

Dans cette preuve, on utilise le fait que la dernière règle ne peut pas être (wsub ax), (wsub void), (wsub cong), (wsub swap), (wsub updt) ou (wsub lam). Résultat qui est une conséquence du lemme 14.1. Dans le cas de la règle (wsub n-app), on utilise le fait que le type $(t\ \phi_1 \ldots \phi_n)$, où t est une variable de type, ne peut pas être équivalent à un type flèche.

La preuve de la seconde propriété est, elle aussi, très similaire à la preuve du lemme 15.7. L'unique cas nouveau, est que la dernière règle utilisée dans l'inférence du jugement peut être (sub arrow). Dans ce cas le résultat est trivialement vrai. □

Corollaire 15.10 *Si* $\Gamma \vdash (\sigma_0 \to \tau_0) \leqslant (\sigma_1 \to \tau_1)$, *alors* $\sigma_1 \leqslant \sigma_0$ *et* $\tau_0 \leqslant \tau_1$.

Preuve On prouve cette propriété en utilisant le lemme 15.9 avec l'hypothèse que $\eta_0 = (\sigma_0 \to \tau_0)$, et que $\eta_1 = (\sigma_2 \to \tau_2)$. On utilise aussi le fait que $(\sigma_1 \to \tau_1) \sim (\sigma_2 \to \tau_2)$ implique $\sigma_1 \sim \sigma_2$ et $\tau_1 \sim \tau_2$, qui est une conséquence du lemme 14.1. □

15.2.2 Analyse des types enregistrements

Nous montrons maintenant quelques propriétés sur le typage des enregistrements. Soit $\Gamma \vdash \varrho \lhd l : \tau$ le jugement défini par les règles suivantes.

Définition 15.3 (Sélection) Le jugement $\Gamma \vdash \varrho \lhd l : \tau$ permet d'associer à un enregistrement ϱ, le type de son champ l.

$$\frac{}{\Gamma \vdash [\,\sigma\,, l : \tau\,] \lhd l : \tau}\ \text{(sel ok)} \qquad \frac{\Gamma \vdash \varrho \lhd l : \tau \qquad k \neq l}{\Gamma \vdash [\,\varrho\,, k : \sigma\,] \lhd l : \tau}\ \text{(sel next)} \qquad \frac{\Gamma \vdash \sigma \lhd l : \tau \qquad \varrho \sim \sigma}{\Gamma \vdash \varrho \lhd l : \tau}\ \text{(sel eq)}$$

$$\frac{\Gamma \vdash (\sigma\ \phi_1 \ldots \phi_n) \lhd l : \tau \qquad (t \sqsubseteq \sigma) \in \Gamma}{\Gamma \vdash (t\ \phi_1 \ldots \phi_n) \lhd l : \tau}\ \text{(sel var)}$$

Lemme 15.11 (Quelques propriétés du jugement $\Gamma \vdash \varrho \lhd l : \tau$)

(*i*) *Il n'existe pas de types* ϱ,τ *et d'étiquette* l, *tels que* $\varrho \sim [\,]$ *et* $\Gamma \vdash \varrho \lhd l : \tau$;
(*ii*) *si* $\Gamma \vdash \varrho_0 \lhd l : \tau_0$, *et* $\Gamma \vdash \varrho_1 \lhd l : \tau_1$, *et* $\varrho_0 \sim \varrho_1$, *alors* $\tau_0 \sim \tau_1$;
(*iii*) *si* $\Gamma \vdash \varrho \lhd l : \tau$ *et* $\tau \sim \sigma$, *alors* $\Gamma \vdash \varrho \lhd l : \sigma$;
(*iv*) *si* $\Gamma \vdash \xi \lhd l : \tau$ *et* $k \neq l$ *et* $\xi \sim [\,\varrho\,, k : \sigma\,]$, *alors* $\Gamma \vdash \varrho \lhd l : \tau$.

Un corollaire de la propriété (*i*) est que le jugement $\Gamma \vdash [\,] \lhd l : \tau$ n'est pas prouvable. Un corollaire de la propriété (*ii*) est que $\Gamma \vdash \varrho \lhd l : \tau$ et $\Gamma \vdash \varrho \lhd l : \sigma$, implique $\tau \sim \sigma$. Un corollaire de la propriété (*iv*) est que $\Gamma \vdash [\,\varrho\,, k : \sigma\,] \lhd l : \tau$ et $k \neq l$, implique $\Gamma \vdash \varrho \lhd l : \tau$.

Preuve On prouve la première propriété par l'absurde. Supposons que $\Gamma \vdash \varrho \lhd l : \tau$. La dernière règle de cette dérivation ne peut pas être (sel var), ou (sel next) ou (sel ok). Si la dernière règle est (sel eq), on se retrouve à prouver la même propriété sur une inférence plus petite. Par conséquent il y a contradiction.

Soit (D0) et (D1), les dérivations $\Gamma \vdash \varrho_0 \lhd l : \tau_0$ et $\Gamma \vdash \varrho_1 \lhd l : \tau_1$. On prouve la seconde propriété par induction sur le couple: (taille de (D0); taille de (D1)), en utilisant l'ordre lexicographique. On fait une étude par cas sur la dernière règle de (D0).

- **cas (sel eq):** dans ce cas, il existe un type ϱ_2, tel que $\varrho_2 \sim \varrho_0$ et $\Gamma \vdash \varrho_2 \lhd l : \tau_0$. En utilisant la transitivité de $\sim$, il en suit que $\varrho_1 \sim \varrho_2$ et, par induction, $\tau_1 \sim \tau_0$. On peut utiliser la même preuve dans le cas où (D1) termine par (sel eq). Par conséquent, dans le reste de la preuve, nous ne considérons plus ce cas;
- **cas (sel ok):** on a $\varrho_0 = [\,\varrho'\,, l : \tau_0\,]$. Par conséquent (D1) se termine par la règle (sel ok) ou (sel eq). Dans le premier cas, on a $\tau_1 = \tau_0$, c'est-à-dire le résultat attendu. Il faut remarquer que la dernière règle de (D1) ne peut pas être (sel next) ou (sel var): ce résultat est encore une conséquence du lemme 14.1;
- **cas (sel next):** on a $\varrho_0 = [\,\varrho'\,, l : \tau'\,]$, et $\Gamma \vdash \varrho' \lhd l : \tau_0$. Par conséquent (D1) se termine par (sel next) ou (sel eq). Dans le premier cas, le résultat découle directement de l'hypothèse d'induction. Le second cas a été traité dans le premier item;
- **cas (sel var):** la preuve est la même que dans le cas précédent.

La propriété (*iii*) est prouvée par induction sur l'inférence de $\Gamma \vdash \varrho \lhd l : \tau$. Le seul cas intéressant, est celui où la dernière règle utilisée est (sel ok). Dans ce cas, on a $\varrho = [\,\varrho'\,, l : \tau\,]$. Or, comme $\sim$ est une congruence, on prouve que $\sigma \sim \tau$ implique $\varrho \sim [\,\varrho'\,, l : \sigma\,]$. Le résultat découle alors de l'emploi de la règle (sel eq).

On montre maintenant la propriété (*iv*). Supposons que $\Gamma \vdash \xi \lhd l : \tau$ et $\xi \sim [\,\varrho\,, k : \sigma\,]$. La propriété (*iv*) est prouvée par induction sur la structure de la dérivation $\Gamma \vdash \xi \lhd l : \tau$. Comme dans la preuve de la propriété (*ii*) (cf. cas (sel ok)), on montre que la dernière règle ne peut être que (sel next) ou (sel eq).

- **cas (sel next):** on a $\xi = [\,\varrho'\,, k : \sigma'\,]$, avec $\varrho \sim \varrho'$ et $\sigma \sim \sigma'$. En utilisant l'hypothèse d'induction, on montre alors que $\Gamma \vdash \varrho' \lhd l : \tau$. Le résultat suit alors de la règle (sel eq);

– **cas (sel eq):** on a $\xi \sim \xi'$ et $\Gamma \vdash \xi' \triangleleft l : \tau$. La première hypothèse implique que $\xi' = [\,\varrho\,, k : \sigma\,]$. Il suffit donc d'utiliser l'hypothèse d'induction pour montrer le résultat attendu.

□

Nous montrons que si $\Gamma \vdash \varrho \triangleleft l : \tau$, alors $l : \tau$ fait partie de ϱ. Plus exactement, on a le résultat suivant.

Lemme 15.12 *Si* $\Gamma \vdash \varrho \triangleleft l : \tau$ *et* $\Gamma \vdash \varrho :: \mathbb{R}$, *alors* $\Gamma \vdash \varrho \sqsubseteq [\,l : \tau\,]$.

Preuve Supposons que $\Gamma \vdash \varrho :: \mathbb{R}$, et $\Gamma \vdash \varrho \triangleleft l : \tau$. Nous prouvons le lemme 15.12 par induction sur la structure du jugement $\Gamma \vdash \varrho \triangleleft l : \tau$.

– **cas (sel ok):** on a $\varrho = [\,\sigma\,, l : \tau\,]$ et $\Gamma \vdash [\,\sigma\,, l : \tau\,] :: \mathbb{R}$. La dernière hypothèse implique que $\Gamma \vdash \sigma :: \mathbb{R}$. Par conséquent, en utilisant règle (wsub void), on a $\Gamma \vdash \sigma \sqsubseteq [\,]$, et en utilisant la règle (wsub cong): $\Gamma \vdash [\,\sigma\,, l : \tau\,] \sqsubseteq [\,l : \tau\,]$;
– **cas (sel next):** on a $\varrho = [\,\sigma\,, k : \theta\,]$, et $k \neq l$. On utilise l'hypothèse d'induction pour prouver que $\Gamma \vdash \sigma \sqsubseteq [\,l : \tau\,]$. Par conséquent, en utilisant les règles (wsub cong) et (sub swap), on obtient que: $\Gamma \vdash [\,\sigma\,, k : \theta\,] \sqsubseteq [\,l : \tau, k : \theta\,] \sqsubseteq [\,l : \tau\,]$;
– **cas (sel var):** on a $\varrho = (t\ \phi_1 \ldots \phi_n)$, et il existe un type θ tel que $(t \sqsubseteq \theta) \in \Gamma$. On utilise l'hypothèse d'induction pour prouver que $\Gamma \vdash (\theta\ \phi_1 \ldots \phi_n) \sqsubseteq [\,l : \tau\,]$. Or, en utilisant les règles (wsub ax) et (wsub n-app), il est possible de montrer que $\Gamma \vdash (t\ \phi_1 \ldots \phi_n) \sqsubseteq (\theta \phi_1 \ldots \phi_n)$. Le résultat suit alors de la règle (wsub trans);
– **cas (sel eq):** le résultat suit de la règle (wsub eq).

□

Nous montrons maintenant que si $\varrho_0 \sqsubseteq \varrho_1$, et si $l : \tau$ est dans ϱ_1, alors ce champ est aussi dans ϱ_0. On montre la même propriété avec la relation $\leqslant$. Cependant, dans ce dernier cas, il faut prendre en compte le sous-typage en profondeur.

Lemme 15.13 *Si* $\Gamma \vdash \varrho_0 \sqsubseteq \varrho_1$ *et* $\Gamma \vdash \varrho_1 \triangleleft l : \tau$, *alors* $\Gamma \vdash \varrho_0 \triangleleft l : \tau$. *De même, si* $\Gamma \vdash \varrho_0 \leqslant \varrho_1$ *et* $\Gamma \vdash \varrho_1 \triangleleft l : \tau$, *alors il existe un type* σ *tel que* $\Gamma \vdash \varrho_0 \triangleleft l : \sigma$, *et* $\Gamma \vdash \sigma \leqslant \tau$.

Preuve Supposons que $\Gamma \vdash \varrho_0 \sqsubseteq \varrho_1$ et $\Gamma \vdash \varrho_1 \triangleleft l : \tau$. La preuve est faite par induction sur l'inférence de $\Gamma \vdash \varrho_0 \sqsubseteq \varrho_1$.

– **cas (wsub ax):** il existe une variable t, dans le domaine de Γ, telle que $(t \sqsubseteq \varrho_1) \in \Gamma$ et $\varrho_0 = t$. Le résultat suit alors de la règle (sel var);
– **cas (wsub eq):** le résultat suit de la règle (sel eq);
– **cas (wsub trans):** on utilise l'hypothèse d'induction deux fois;
– **cas (wsub void):** ce cas n'est pas possible puisque, quelque soit le champ label l, on a montré que $\Gamma \vdash [\,] \triangleleft l : \tau$ n'est pas prouvable (cf. lemme 15.11-(i));
– **cas (wsub cong):** on a $\varrho_i = [\,\sigma_i\,, k : \theta\,]$ (pour i dans l'ensemble $\{0,1\}$), et $\Gamma \vdash \sigma_0 \sqsubseteq \sigma_1$. Si k est égal à l, alors $\Gamma \vdash \varrho_1 \triangleleft l : \theta$, et en utilisant le lemme 15.11, il en suit que $\theta \sim \tau$. De plus, en utilisant l'hypothèse d'induction, on montre que $\Gamma \vdash \varrho_0 \triangleleft l : \theta$. Par conséquent, en utilisant (sel eq), on montre que $\Gamma \vdash \varrho_0 \triangleleft l : \tau$. La preuve, dans le cas k différent de l, utilise la règle (sel next) et l'hypothèse d'induction. La preuve est similaire dans le cas (wsub swap);
– **cas (wsub updt):** on a $\varrho_0 = [\,\varrho_1\,, k : \theta\,]$, et $\Gamma \vdash \varrho_1 \sqsubseteq [\,k : \theta\,]$. Supposons que k égal l. En utilisant l'hypothèse d'induction et la règle (sel ok), on montre que $\Gamma \vdash \varrho_1 \triangleleft l : \theta$. Cette propriété implique que $\theta \sim \tau$ (cf. lemme 15.11-(ii)). On peut alors prouver le résultat attendu en utilisant le lemme 15.11-(iii). Si k est différent de l, on utilise la règle (sel next) qui permet de montrer que $\Gamma \vdash [\,\varrho_1\,, k : \theta\,] \triangleleft l : \tau$ dès que $\Gamma \vdash \varrho_1 \triangleleft l : \tau$;
– **cas (wsub lam):** ce cas est impossible puisque le lemme 14.1 implique que $\Lambda t^k.\sigma \nsim [\,\varrho\,, l : \tau\,]$ et que $\Lambda t^k.\sigma \nsim (t\ \phi_1 \ldots \phi_n)$. Par conséquent on ne peut pas utiliser la règle (sel eq), et donc pour tout type σ et champ l, le jugement $\Gamma \vdash \Lambda t^k.\sigma \triangleleft l : \tau$ n'est pas prouvable;

– **cas (wsub n-app):** on a $\varrho_0 = (\theta_0\ \phi_1 \dots \phi_n)$, et $\varrho_1 = (\theta_1\ \phi_1 \dots \phi_n)$, et $(\sharp) : \Gamma \vdash \theta_0 \sqsubseteq \theta_1$. On utilise une induction sur l'inférence de $(\sharp)$. Comme dans la preuve du lemme 15.9, il n'y a que quatre cas possible.
  – **cas (wsub ax):** il existe une variable, disons t, telle que $(t \sqsubseteq \theta_1) \in \Gamma$ et $\theta_0 = t$. Le résultat suit de la règle (sel var);
  – **cas (wsub eq):** Comme $\sim$ est «close par utilisation», on a $\varrho_0 \sim \varrho_1$. Le résultat suit de la règle (sel eq);
  – **cas (wsub lam):** on a $\Gamma,t :: \kappa \vdash \theta'_0 \sqsubseteq \theta'_1$, et $\theta_i = \Lambda t^k.\theta'_i$ (pour $i \in \{0,1\}$). Par conséquent, $\varrho_i \sim (\theta'_i\{\phi_1/t\}\ \phi_2 \dots \phi_n)$. Le résultat suit de l'hypothèse d'induction et du lemme 15.3-(i);
  – **cas (wsub trans):** il existe un type θ_2, tel que $\Gamma \vdash \theta_0 \sqsubseteq \theta_2$, et $\Gamma \vdash \theta_2 \sqsubseteq \theta_1$. Par conséquent, en utilisant la règle (wsub n-app), on obtient deux sous preuves: $\Gamma \vdash \varrho_0 \sqsubseteq (\theta_2\ \phi_1 \dots \phi_n)$, et $\Gamma \vdash (\theta_2\ \phi_1 \dots \phi_n) \sqsubseteq \varrho_1$. Le résultat s'obtient alors en utilisant l'hypothèse d'induction.

La preuve de la seconde propriété est faite d'une manière similaire.

□

Tout ces résultats, nous permettent de prouver une propriété équivalente à la proposition 12.3.

Corollaire 15.14 *Si* $\Gamma \vdash [\xi, l : \tau] \leqslant [\rho, l : \sigma]$, *alors* $\Gamma \vdash \tau \leqslant \sigma$. *Si* $\Gamma \vdash [\xi, l : \tau] \leqslant [k : \sigma]$ *et* $k \neq l$, *alors* $\Gamma \vdash \xi \leqslant [k : \sigma]$.

Preuve Pour la première propriété, on remarque que $\Gamma \vdash [\rho, l : \sigma] \lhd l : \sigma$, et que $\Gamma \vdash [\xi, l : \tau] \lhd l : \tau$. Par conséquent, en utilisant le lemme 15.13, on obtient que $\Gamma \vdash \tau \leqslant \sigma$. Pour la seconde propriété, on utilise l'hypothèse $\Gamma \vdash [\xi, l : \tau] \leqslant [k : \sigma]$, et le fait que $\Gamma \vdash [k : \sigma] \lhd k : \sigma$: en utilisant le lemme 15.13, on montre qu'il existe un type σ', tel que $\Gamma \vdash [\xi, l : \tau] \lhd k : \sigma'$ et $\Gamma \vdash \sigma' \leqslant \sigma$. On utilise alors le lemme 15.11, qui permet de prouver que $\Gamma \vdash \xi \lhd k : \sigma'$. Pour finir, on utilise le lemme 15.12, qui implique que $\Gamma \vdash \xi \leqslant [k : \sigma']$, et donc, par transitivité, que $\Gamma \vdash \xi \leqslant [k : \sigma]$. □

15.3 Analyse des jugements de type

Par analyse des jugements de types, on entend des résultats qui permettent de donner le type des composants d'un processus, à partir du type du processus. On peut, par exemple, donner des résultats qui utilisent les propriétés d'analyse des jugements de sous-typage.

Lemme 15.15 *S'il existe une preuve du jugement* $\Gamma \vdash (\boldsymbol{\nu} u)P : \sigma$, *telle que seules les règles (type inst), (type weak) et (type sub) sont utilisées après la dernière utilisation de (type new), alors il existe un type* ϱ *tel que* $\Gamma,u : \varrho \vdash P : \sigma$.

Preuve Soit $(\sharp)$ la preuve du jugement $\Gamma \vdash (\boldsymbol{\nu} u)P : \sigma$ qui utilise seulement (type inst), (type weak) et (type sub) après la dernière utilisation de (type new). La preuve est faite par induction sur l'inférence de $(\sharp)$. On étudie la dernière règle utilisée:
– **cas (type new):** le résultat est immédiat et découle de la définition de (type new);
– **cas (type sub):** il existe un type σ', tel que $\Gamma \vdash (\boldsymbol{\nu} u)P : \sigma'$, et $\Gamma \vdash \sigma' \leqslant \sigma$. Par conséquent, en utilisant l'hypothèse d'induction, on a $\Gamma,u : \varrho \vdash P : \sigma'$. Le résultat suit en utilisant la règle (type sub);
– **cas (type inst):** il existe trois types: τ_1, τ'_1, et σ', tels que $\Gamma \vdash (\boldsymbol{\nu} u)P : (\forall t \sqsubseteq \tau_1\ .\ \sigma')$ et $\Gamma \vdash \tau'_1 \sqsubseteq \tau_1$, et $\sigma = \sigma'\{\tau'_1/t_1\}$. Par conséquent, en utilisant l'hypothèse d'induction, on a $\Gamma,u : \varrho \vdash P : (\forall t \sqsubseteq \tau_1\ .\ \sigma')$. En utilisant l'affaiblissement on obtient que $\Gamma,u : \varrho \vdash \tau'_1 \sqsubseteq \tau_1$. Le résultat suit de l'utilisation de la règle (type inst);

– **cas (type weak):** il existe un environnement Γ', tel que $\Gamma,\Gamma' \vdash *$, et $\Gamma,\Gamma' \vdash (\boldsymbol{\nu} u)P : \sigma$. En utilisant l'hypothèse d'induction, on prouve que $\Gamma,u : \varrho \vdash P : \sigma$. De plus, on peut utiliser le lemme de renommage (lemme 15.5), pour choisir $u \notin \mathbf{dom}(\Gamma,\Gamma')$. Le résultat suit du lemme 15.6 et de la règle (type inst).

Lemme 15.16 (Analyse des jugements de types)

(*i*) *Si* $\Gamma \vdash (\boldsymbol{\nu} u)P : \sigma \to \tau$, *alors il existe un type* ϱ *tel que:* $\Gamma,u : \varrho \vdash P : \sigma \to \tau$. *De même, si* $\Gamma \vdash (\boldsymbol{\nu} u)P : [\,\sigma\,, l : \tau\,]$, *alors il existe un type* ϱ *tel que:* $\Gamma,u : \varrho \vdash P : [\,\sigma\,, l : \tau\,]$;

(*ii*) *si* $\Gamma \vdash (\boldsymbol{\lambda} x)P : \sigma \to \tau$, *alors il existe un type* ϱ *tel que:* $\Gamma,x : \varrho \vdash P : \tau$, *et* $\Gamma \vdash \sigma \leqslant \varrho$;

(*iii*) *si* $\Gamma \vdash [\,R\,, l = Q\,] : [\,\sigma\,, k : \tau\,]$, *alors il existe deux types,* ξ *et* ϱ, *tels que* $\Gamma \vdash R : \xi$ *et* $\Gamma \vdash Q : \varrho$, *et* $[\,\xi\,, l : \varrho\,] \leqslant [\,\sigma\,, k : \tau\,]$;

(*iv*) *si* $\Gamma \vdash (P \mid Q) : \tau$, *alors* $\Gamma \vdash P : \mathsf{o}$ *et* $\Gamma \vdash Q : \tau$;

(*v*) *si* $\Gamma \vdash \langle u \Leftarrow P \rangle : \mathsf{o}$, *alors il existe un type* τ *tel que* $(u : \tau) \in \Gamma$ *et* $\Gamma \vdash P : \tau$.

Preuve On prouve la propriété (*i*). Considérons une preuve canonique du jugement ($\sharp$) $\Gamma \vdash (\boldsymbol{\nu} u)P : \sigma \to \tau$ (cf. définitions 15.1 et 15.2). Nous prouvons par réfutation que cette preuve ne contient pas d'utilisation de la règle (type gen) après la dernière utilisation de (type new). Supposons le contraire. Comme ($\sharp$) est une preuve en forme canonique, on utilise la règle (type gen) puis les règles (type weak) et (type sub). Par conséquent il existe un type de la forme $(\forall t \sqsubseteq \eta\,.\,\vartheta)$ qui est sous-type de $(\sigma \to \tau)$, ce qui implique, d'après le lemme 15.7-(*ii*), que $(\sigma \to \tau) \sim (\forall t \sqsubseteq \eta\,.\,\vartheta)$, ce qui n'est pas possible (cf. lemme 14.1). La propriété (*i*) suit alors du lemme 15.15.

Les preuves des propriétés (*ii*) à (*v*) utilisent des propriétés similaires au lemme 15.15. Dans les cas (*ii*) et (*iii*), on utilise le fait qu'on peut trouver un jugement de type $\Gamma \vdash (\boldsymbol{\lambda} x)P : \sigma \to \tau$ (resp. $\Gamma \vdash [\,R\,, l = Q\,] : \sigma$), qui se termine par l'utilisation de la règle (type app) (resp. (type over)), suivi par l'utilisation des règles (type sub) et (type weak). Dans la preuve de la propriété (*ii*), on utilise aussi le corollaire 15.10. □

15.4 Preuve de la conservation du typage

On prouve, tout d'abord, une relation entre le typage d'un processus et les contextes.

Lemme 15.17 (Types et substitutions)

– *Soit* $\mathbf{C}$ *un contexte*[1] *qui ne lie pas le nom* u, *ni les noms libres de* P, *et soit* Γ *un environnement tel que* $(u : \tau) \in \Gamma$. *Si* $\Gamma \vdash \mathbf{C}[u] : \sigma$ *et* $\Gamma \vdash P : \tau$, *alors* $\Gamma \vdash \mathbf{C}[P] : \sigma$;

– *les jugements de types sont préservés par les contextes, c'est-à-dire que si* $\Gamma \vdash P : \tau$, *et* $\Gamma \vdash Q : \tau$, *et* $\Gamma \vdash \mathbf{C}[P] : \sigma$, *alors* $\Gamma \vdash \mathbf{C}[Q] : \sigma$.

Preuve La preuve de lemme 15.17 est faite par induction sur l'inférence de $\mathbf{C}$, puis sur l'inférence de $\Gamma \vdash \mathbf{C}[u] : \sigma$. Cette technique de preuve nous permet de ne pas considérer les preuves du jugement $\Gamma \vdash \mathbf{C}[u] : \sigma$ qui se terminent par l'utilisation des règles (type inst), (type gen), (type weak) ou (type sub). C'est-à-dire les règles – à termes constants – de la forme $\Gamma \vdash P : \tau_1 \Rightarrow \Delta \vdash P : \tau_2$.

– **cas** $\mathbf{C} = [\_]$ **et la dernière règle utilisée est (type ax):** le résultat suit directement de l'hypothèse $\Gamma \vdash P : \tau$;

– **cas** $\mathbf{C} = (\mathbf{D}u)$ **et la dernière règle utilisée est (type app):** on a $\Gamma,x : \tau,\Gamma' \vdash \mathbf{D}[x] : \varrho_1 \to \varrho_2$ et $\Gamma,x : \tau,\Gamma' \vdash u : \varrho_1$ et $\Gamma,x : \tau,\Gamma' \vdash P : \tau$. Par conséquent, en utilisant l'hypothèse d'induction, on obtient que $\Gamma,x : \tau,\Gamma' \vdash \mathbf{D}[P] : \varrho_1 \to \varrho_2$. Le résultat suit de l'utilisation de la règle (type app). La preuve est similaire dans les cas $\mathbf{C} = \langle u = \mathbf{D} \rangle$, et $\mathbf{C} = (Q \mid \mathbf{D})$, et $\mathbf{C} = [\,\mathbf{D}\,, l = Q\,]$, et $\mathbf{C} = [\,R\,, l = \mathbf{D}\,]$, et $\mathbf{C} = \mathbf{D} \cdot l$;

1. On suppose que le trou du contexte $\mathbf{C}$ n'apparaît pas en argument d'une application, ce qui permet de montrer que $\mathbf{C}[P]$ est un terme bien formé de $\boldsymbol{\pi}^\star$.

– **cas $\mathbf{C} = (\boldsymbol{\lambda} y)\mathbf{D}$ et la dernière règle utilisée est (type abs):** par hypothèse, y n'est pas dans l'ensemble $\{x\} \cup \mathbf{fn}(P)$, puisque aucune variable de P n'est capturée par $\mathbf{C}$. Par conséquent, on a $\Gamma,x : \tau,\Gamma',y : \varrho_1 \vdash \mathbf{D}[x] : \varrho_2$ et $\Gamma,x : \tau,\Gamma' \vdash P : \tau$. En utilisant la règle d'affaiblissement, on obtient $\Gamma,x : \tau,\Gamma',y : \varrho_1 \vdash P : \tau$. Le résultat suit alors de l'utilisation de la règle (type abs). La preuve est similaire dans le cas $\mathbf{C} = (\boldsymbol{\nu} u)\mathbf{D}$.

La preuve du second résultat est similaire à celle que l'on vient de donner. □

On montre maintenant que le typage est préservé par équivalence structurelle.

Théorème 15.18 (Préservation du typage par équivalence structurelle) *Si $\Gamma \vdash P : \tau$ et $P \equiv P'$, alors $\Gamma \vdash P' : \tau$.*

Preuve On prouve le résultat de préservation, par induction sur l'inférence de $P \equiv P'$, puis sur l'inférence de $\Gamma \vdash P : \tau$. Comme pour la preuve précédente, cette technique de preuve nous permet de supposer que la dernière règle de la preuve de $\Gamma \vdash P : \tau$ n'est pas (type inst), (type gen), (type weak) ou (type sub). On se ramène à une étude de cas sur la dernière règle de l'inférence de $P \equiv P'$. Nous utiliserons dans cette preuve la terminologie employée dans la figure 2.2. Dans le cas de l'α-conversion, le résultat est directement obtenu par notre hypothèse que deux termes α-équivalents ont le même type. Pour la règle qui implique que $\equiv$ est une congruence, on utilise le résultat précédent (lemme 15.17).

– **cas associativité:** on a $P = (P_1 \mid Q_1) \mid R_1$ et $P' = P_1 \mid (Q_1 \mid R_1)$, et la dernière règle du jugement $\Gamma \vdash P : \tau$ est (type par). Par conséquent, on a $\Gamma \vdash (P_1 \mid Q_1) : \mathtt{o}$ et $\Gamma \vdash R_1 : \tau$, et donc, en utilisant le lemme 15.16-(iv), $\Gamma \vdash P_1 : \mathtt{o}$ et $\Gamma \vdash Q_1 : \mathtt{o}$. On obtient le résultat attendu en utilisant deux fois la règle (type par). La preuve est similaire dans le cas symétrique: $P = P_1 \mid (Q_1 \mid R_1)$ et $P' = (P_1 \mid Q_1) \mid R_1$, et dans le cas de la règle de commutativité gauche;
– **cas extension de la portée:** on a $P = (Q_1 \mid (\boldsymbol{\nu} u)P_1)$ et $P' = (\boldsymbol{\nu} u)(Q_1 \mid P_1)$, et la dernière règle du jugement $\Gamma \vdash P : \tau$ est (type par). Par conséquent $\Gamma \vdash Q_1 : \mathtt{o}$, et $\Gamma \vdash (\boldsymbol{\nu} u)P_1 : \tau$. En utilisant une induction sur l'inférence de ($\sharp$) $\Gamma \vdash (\boldsymbol{\nu} u)P_1 : \tau$, il est possible de faire l'hypothèse que la preuve de ($\sharp$) se termine par une utilisation de (type new). Dans ce cas simple, il suffit d'échanger les utilisations des règles (type new) et (type app) pour obtenir un typage de P'. La preuve est similaire dans le cas symétrique: $P = ((\boldsymbol{\nu} u)P_1 \mid Q_1)$;
– **cas distributivité sur le parallèle:** on a $P = (P_1 \mid Q_1)u$ et $P' = P_1 \mid (Q_1 u)$, et la dernière règle du jugement $\Gamma \vdash P : \tau$ est (type app). Par conséquent, on a que ($\sharp$) $\Gamma \vdash (P_1 \mid Q_1) : \sigma \to \tau$ et que $\Gamma \vdash u : \sigma$. Le jugement ($\sharp$) et le lemme 15.16-(iv) permettent de montrer que $\Gamma \vdash P_1 : \mathtt{o}$ et $\Gamma \vdash Q_1 : \sigma \to \tau$. Il suffit alors d'utiliser la règle (type app), suivie de (type par), pour obtenir le résultat attendu. Le cas symétrique, ainsi que le cas de la distribution de la sélection sur la composition parallèle, sont similaires;
– **cas distributivité sur la restriction:** on a $P = ((\boldsymbol{\nu} u)P_1)v$, et $P' = (\boldsymbol{\nu} u)(P_1 v)$, et $u \neq v$, et la dernière règle du jugement $\Gamma \vdash P : \tau$ est (type app). Par conséquent, il existe un type σ tel que $\Gamma \vdash (\boldsymbol{\nu} u)P_1 : \sigma \to \tau$, et $\Gamma \vdash v : \sigma$. Le premier jugement et la propriété 15.16-(i), impliquent qu'il existe un type ϱ, tel que $\Gamma,u : \varrho \vdash P_1 : \sigma \to \tau$. De plus, en utilisant la règle d'affaiblissement, on montre que $\Gamma,u : \varrho \vdash v : \sigma$. Le résultat suit en utilisant la règle (type app) suivie de (type new).

□

On prouve finalement que les jugements de types sont préservé par réduction, c'est-à-dire la propriété donnée dans le théorème 14.3 – à la section 14.3 – que nous reproduisons ici.

Théorème 14.3 (Préservation du typage) Si $\Gamma \vdash P : \tau$ et $P \to P'$, alors $\Gamma \vdash P' : \tau$.

Preuve Le preuve est faite par induction sur l'inférence de $P \to P'$, puis sur l'inférence de $\Gamma \vdash P : \tau$. Comme pour la preuve précédente, on peut supposer que la dernière règle utilisée n'est

pas (type inst), (type gen), (type weak) ou (type sub). On se ramène alors à une étude de cas sur la dernière règle de l'inférence de $P \to P'$.

- **cas (red context):** on a comme hypothèse que $P \to P'$ et que $\mathbf{E}$ est un contexte d'évaluation, et comme résultat que $\mathbf{E}[P] \to \mathbf{E}[P']$. le résultat découle de l'hypothèse d'induction et du lemme 15.17.
- **cas (red struct):** on utilise le théorème 15.18;
- **cas (red beta):** on a $P = (\!(\boldsymbol{\lambda} x)\!) P_1) v$ et $P \to P_1\{v/x\}$. Par hypothèse la dernière règle utilisée dans le typage de P est (type app). Par conséquent, il existe un type σ tel que $\Gamma \vdash v : \sigma$ et $\Gamma \vdash (\!(\boldsymbol{\lambda} x)\!) P_1 : \sigma \to \tau$. En utilisant le lemme 15.16-(ii), on montre alors qu'on peut prouver le jugement $\Gamma, x : \varrho \vdash P_1 : \tau$, tel que $\Gamma \vdash \sigma \leqslant \varrho$. De plus, en utilisant l'α-conversion, on peut choisir x différent de v. Il suffit alors d'utiliser les règles (type sub) et (type weak) pour prouver le jugement: $\Gamma, x : \varrho \vdash v : \varrho$. Finalement, on utilise le lemme de substitution (lemme 15.3-(iii)) pour obtenir le résultat attendu. C'est-à-dire: $\Gamma \vdash P_1\{v/x\} : \tau$;
- **cas (red decl):** On a $P = (\langle u \Leftarrow P_1 \rangle \mid u\, a_1 \dots a_n)$ et $P \to P_1\, a_1 \dots a_n$. On peut supposer être dans le cas où la dernière règle utilisée pour typer P est (type par). Par conséquent, $(\sharp)$ $\Gamma \vdash \langle u \Leftarrow P_1 \rangle : \mathsf{o}$ et $\Gamma \vdash (u\, a_1 \dots a_n) : \tau$. En utilisant le lemme 15.16-(v) et le jugement $(\sharp)$, on montre qu'il existe un type σ tel que $(u : \sigma) \in \Gamma$ et $\Gamma \vdash P_1 : \sigma$. Par conséquent, en utilisant le lemme 15.17 avec $\mathbf{C} = (\,[\_]\; a_1 \dots a_n)$, on obtient que $\Gamma \vdash (P\, a_1 \dots a_n) : \tau$. Ce qui est le résultat attendu;
- **cas (red over):** on a $P = [\,R, l = Q\,] \cdot k$ et $k \neq l$, qui implique que $P \to R \cdot k$. On peut supposer être dans le cas où la dernière règle utilisée pour typer P est (type sel). Par conséquent $\Gamma \vdash [\,R, l = Q\,] : [\,\sigma, k : \tau\,]$. En utilisant le lemme 15.16-(iii), il en suit qu'il existe deux types, ξ et ϱ, tels que $\Gamma \vdash R : \xi$ et $\Gamma \vdash [\,\xi, l : \varrho\,] \leqslant [\,\sigma, k : \tau\,]$. Par transitivité de la relation $\leqslant$, on en déduit donc que $\Gamma \vdash [\,\xi, l : \varrho\,] \leqslant [\,k : \tau\,]$, et en utilisant le corollaire 15.14, que $\Gamma \vdash \xi \leqslant [\,k : \tau\,]$. on obtient le résultat attendu, c'est-à-dire : $\Gamma \vdash R \cdot k : \tau$, en utilisant la règle (type sub), puis (type sel);
- **cas (red sel):** on a $P = [\,R, l = Q\,] \cdot l$, qui implique $P \to Q$. La preuve, dans ce cas, est similaire au cas précédent. On utilise le lemme 15.16-(iii) pour prouver qu'il existe deux types, ξ et ϱ, tels que $\Gamma \vdash Q : \varrho$ et $\Gamma \vdash [\,\xi, l : \varrho\,] \leqslant [\,\sigma, l : \tau\,]$. Par conséquent, en utilisant le corollaire 15.14, on montre que: $\Gamma \vdash \varrho \leqslant \tau$ et donc, en utilisant la règle (type sub), que: $\Gamma \vdash Q : \tau$.

□

Discussion

Dans les deux derniers chapitres, nous avons étudié un système de types avec polymorphisme d'ordre supérieur pour le calcul bleu. Ce système, plus exactement la partie concernant les opérateurs du λ-calcul, a déjà été utilisé informellement dans quelques articles traitant du typage des langages à objets [24, 106, 47]. En effet il combine à la fois récursion, types d'ordre supérieur et une forme particulière de quantification bornée, qui sont trois éléments que l'on retrouve souvent dans l'étude du typage des objets. L'auteur ne connaît pas de travaux dans lequel ce système – qui s'apparente à une version «à la Curry» de $\mathtt{F}_{\leq\mu}$– est détaillé formellement. On trouve par contre de nombreuses références à des systèmes «à la Church» qui possèdent ces trois éléments [3], c'est-à-dire des systèmes avec types explicites et une opération d'application d'un terme à un type. c'est aussi la première fois, à notre connaissance, que l'on prouve la propriété de conservation du typage pour ce système.

Le système $\mathtt{BF}_{\leqslant}$ peut être encore généralisé. On peut par exemple introduire un opérateur de quantification bornée sur la relation de sous-typage la plus générale, $(\forall t \leqslant \tau \,.\, \sigma)$. On peut aussi ajouter une constante, $\top$, qui représente le type le plus grand. Cependant, nous nous sommes restreint à l'étude du système $\mathtt{BF}_{\leqslant}$ car il représente le système minimal nécessaire à la définition de l'interprétation du type des objets extensibles. En effet, dans la partie suivante, nous nous servons de ce système de type pour donner une interprétation du calcul d'objets extensible de K. Fisher *et al.* [46, 49], qui conserve la notion de typage et de sous-typage.

Quatrième partie

Objets

CHAPITRE 16

Les calculs d'objets

> J'ai un chat qui s'appelle Trash. Dans le climat politique actuel, j'ai l'impression que si je cherchais à le vendre (tout du moins à un informaticien), je ne soulignerais pas le fait qu'il est doux et auto-suffisant, se nourrissant principalement de souris des champs. Non, j'utiliserais plutôt comme argument qu'il est orienté objets.
>
> – *Roger King*

DANS SON ARTICLE SUR LE CODAGE DES FONCTIONS dans le π-calcul [98], R. MILNER remarque que la relation de réduction de $\boldsymbol{\pi}$ est basée sur le *paradigme objet*, dans le sens où «ce qui est transmis et lié [dans le π-calcul] n'est jamais l'objet lui-même, mais plutôt le moyen d'*accéder* à l'objet». Le lien entre le π-calcul et les langages à objets est fort: les processus sont les objets, 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 tardive* et de typage.

Dans cette partie, notre but n'est pas de donner une présentation du modèle de programmation à objets. Aussi, nous supposons que le lecteur est familier avec les concepts des langages orientés objets, ainsi qu'avec la notion de calcul d'objets. La lecture de [90, 3, 49] est une bonne introduction à ces concepts. Notre but est d'étudier les rapports entre le modèle de programmation à objet et le calcul bleu. Cette étude est menée grâce à la définition d'un modèle des objets concurrents, qui est défini par le codage dans $\boldsymbol{\pi}^\star$ des opérateurs d'un calcul d'objets. En ce qui concerne le typage des objets concurrents, nous utilisons les systèmes introduits dans la partie III.

Dans la section suivante, nous introduisons le calcul d'objets fonctionnel de M. ABADI et L. CARDELLI, que nous nommons *self* et qui est noté $\boldsymbol{\varsigma}$. Plus précisément, nous introduisons une version du calcul self avec un système de types simples et le sous-typage, qui est désigné par $\mathbf{Ob}_{1<:}$. En nous basant sur les opérateurs de $\boldsymbol{\varsigma}$, nous définissons dans le chapitre 17 un calcul d'objets concurrents. Nous démontrons que les opérateurs de ce calcul peuvent être dérivés dans $\boldsymbol{\pi}^\star$, et nous donnons un système de types dérivés pour ces objets en nous basant sur $\mathtt{B}_{\leqslant}$ (cf. chapitre 12), c'est-à-dire un système du premier ordre avec sous-typage. Dans le chapitre 18, nous utilisons ces opérateurs objets pour donner une interprétation de $\mathbf{Ob}_{1<:}$ qui préserve la notion de typage et de sous-typage. Nous donnons également une interprétation du calcul $\mathbf{conc}_{\boldsymbol{\varsigma}}$ [60, 61], qui est une version concurrente et impérative du calcul self.

Dans le chapitre 20, nous nous intéressons à un autre modèle des objets, introduit par K. FISHER *et al.* [46, 49], dans lequel on peut étendre l'ensemble des méthodes d'un objet. L'intérêt du calcul

$$
\begin{array}{rcll}
e,f,g & ::= & x & \text{variable} \\
& | & [\, l_i = \varsigma(x_i) f_i^{\,i\in[1..n]} \,] & \text{objet (les noms dans } (l_i)_{i\in[1..n]} \text{ sont distincts)} \\
& | & e{\cdot}l & \text{invocation de la méthode } l \text{ sur l'objet } e \\
& | & e{\cdot}l \Leftarrow \varsigma(x) f & \text{modification de } l \text{ dans } e
\end{array}
$$

$$
\frac{e \rightsquigarrow e' \qquad \mathbf{E} \in \mathcal{E}_\varsigma}{\mathbf{E}[e] \rightsquigarrow \mathbf{E}[e']}\ \text{(ac red context)} \qquad \frac{e = [\, l_i = \varsigma(x_i) f_i^{\,i\in[1..n]} \,] \qquad j \in [1..n]}{e{\cdot}l_j \rightsquigarrow f_j\{e/x_j\}}\ \text{(ac red invk)}
$$

$$
\frac{e = [\, l_i = \varsigma(x_i) f_i^{\,i\in[1..n]} \,] \qquad j \in [1..n]}{(e{\cdot}l_j \Leftarrow \varsigma(x) f) \rightsquigarrow [\, l_j = \varsigma(x) f, l_i = \varsigma(x_i) f_i^{\,i\neq j} \,]}\ \text{(ac red updt)}
$$

Fig. 16.1: Termes et réduction dans $\mathbf{Ob}_{1<:}$

d'objets avec extension, dénoté $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$, est qu'il permet de simuler le mécanisme de l'héritage des langages orientés objets: l'ajout de méthode permet de «réutiliser du code» déjà existant, pour définir de nouveaux objets plus complexes. Nous appliquons au calcul $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ le même programme qu'au calcul $\mathbf{Ob}_{1<:}$. Cependant, afin de simplifier notre présentation, nous choisissons $\boldsymbol{\lambda\mathcal{D}}$ef comme calcul cible à la place du calcul bleu. Comme pour chaque partie, nous concluons par une discussion générale.

16.1 Le calcul self

Nous introduisons le calcul d'objets fonctionnels $\mathbf{Ob}_{1<:}$ [3], ce qui nous permet d'introduire des notations et des concepts qui seront utilisés dans toute la partie IV.

Les premiers modèles des langages de programmation à objets qui ont été proposés, ont été donnés par un codage dans des calculs fonctionnels avec des primitives pour les enregistrements. Un travail représentatif est par exemple celui de L. Cardelli et J. Mitchell [27], où les auteurs introduisent un calcul d'enregistrements récursifs et extensibles.

Cependant les modèles basés sur une interprétation des objets soulèvent beaucoup de problèmes. S'il est facile de coder les objets et leur comportement opérationnel, il est plus difficile de donner une interprétation qui offre des notions de typage et de sous-typage satisfaisantes. L'introduction des calculs d'objets, c'est-à-dire des calculs dans lequel les objets sont primitifs, a été motivée par la difficulté posé par l'approche interprétative. Une autre motivation était d'étudier directement les systèmes de types des langages à objets.

Un exemple de calcul d'objets est celui de M. Abadi et L. Cardelli [4], dénoté $\boldsymbol{\varsigma}$. Ce calcul formalise des concepts clefs des langages à objets: l'invocation de méthodes; la notion de *self*: une méthode peut référer à l'objet appelé par l'intermédiaire de son argument self[1]; la *modification de méthodes*, ou *update*. Le calcul obtenu à partir de $\boldsymbol{\varsigma}$ en ajoutant un système de types du premier ordre et le sous-typage est nommé $\mathbf{Ob}_{1<:}$. Dans ce calcul, le sous-typage permet de formaliser la notion de *substitutivité*: un objet peut émuler un autre objet avec moins de méthodes.

La syntaxe et la relation de réduction de $\mathbf{Ob}_{1<:}$ sont donnés dans la figure 16.1. On peut noter que la relation de réduction présentée ici est «à petits pas» – c'est-à-dire *small-step* en anglais – et paresseuse. Pour reprendre les conventions des calculs d'objets [3, 48], on utilise des lettres minuscules: $e,f,g,\ldots$, pour désigner les termes de $\mathbf{Ob}_{1<:}$, et la lettre v lorsque le terme est une valeur, c'est-à-dire un terme de la forme: $[\, l_i = \varsigma(x_i) f_i^{\,i\in[1..n]} \,]$. Dans ce dernier cas, on suppose

1. On retrouve cette notion dans C++ par exemple, avec l'utilisation du mot clef *this*.

que les noms de méthodes $l_1, \ldots, l_n$ sont tous distincts.

Le calcul $\mathbf{Ob}_{1<:}$ ne possède qu'un seul lieur, dénoté ς, qui lie les occurrences de x dans la méthode $\varsigma(x)f$. Comme pour le calcul bleu, il est simple de définir les ensembles des noms libres et des noms liés d'un terme: $\mathbf{fn}(e)$ et $\mathbf{bn}(e)$. Nous ne donnons pas de détails ici. On peut aussi définir les notions de contexte et de contexte d'évaluation.

Définition 16.1 (Contextes d'évaluation) Les contextes d'évaluation, notés **E**, sont les termes appartenant à l'ensemble $\mathcal{E}_\varsigma$, engendré par la grammaire suivante.

$$\mathbf{E} ::= [\_] \mid (\mathbf{E}{\cdot}l_i) \mid (\mathbf{E}{\cdot}l_i \Leftarrow \varsigma(x)f)$$

Le système de types de $\mathbf{Ob}_{1<:}$ est donné dans la figure 16.2. On utilise les majuscules $A,B,\ldots$ pour désigner les types, et on note $E \vdash e : A$ [$\mathbf{Ob}_{1<:}$] les jugements de ce système. L'environnement de types, désigné par la lettre E dans le jugement, associe un type à chaque variable, c'est-à-dire que E appartient à l'ensemble engendré par la grammaire $E ::= \emptyset \mid E,x : A$.

Définition 16.2 (Types et sous-typage) La syntaxe des types est très simple. Un type est soit un type objet: $[\, l_i : {A_i}^{i\in[1..n]}\,]$, avec $n \geq 0$; soit le type maximal $\top$. En particulier, il n'y a pas de variable de types.

$$A,B ::= \top \mid [\, l_i : {A_i}^{i\in[1..n]}\,]$$

Le type $[\, l_i : {A_i}^{i\in[1..n]}\,]$ est le type des objets ayant les méthodes $l_1,\ldots,l_n$, retournant un résultat de type A_i. La relation de sous-typage $<:$ est la plus petite relation réflexive et transitive qui vérifie les deux règles suivantes.

$$\frac{I \subseteq J}{[\, l_i : {A_i}^{i\in I}\,] <: [\, l_i : {A_i}^{i\in J}\,]}\ \text{(ac sub obj)} \qquad \frac{}{A <: \top}\ \text{(ac sub top)}$$

La règle (ac sub obj) implique que A est un sous-type de B, si A possède toutes les méthodes de B et si, pour chacune de ces méthodes, leur type est égal dans A et B.

Soit A le type $[\, l_i : {B_i}^{i\in[1..n]}\,]$

$$\frac{(x : A) \in E}{E \vdash x : A}\ \text{(ac type ax)} \qquad \frac{E \vdash e : A \quad j \in [1..n]}{E \vdash e{\cdot}l_j : B_j}\ \text{(ac type sel)}$$

$$\frac{E \vdash e : A \quad A <: B}{E \vdash e : B}\ \text{(ac type sub)} \qquad \frac{\forall i, i \in [1..n] \quad E,x_i : A \vdash f_i : B_i}{E \vdash [\, l_i = \varsigma(x_i)f_i^{\,i\in[1..n]}\,] : A}\ \text{(ac type obj)}$$

$$\frac{E \vdash e : A \quad E,x : A \vdash f : B_j \quad j \in [1..n]}{E \vdash e{\cdot}l_j \Leftarrow \varsigma(x)f : A}\ \text{(ac type updt)}$$

Fig. 16.2: Système de types de $\mathbf{Ob}_{1<:}$

16.2 Un objet concurrent simple: la cellule

Avant de définir, dans le chapitre suivant, les opérateurs d'un calcul d'objets concurrents et leur codage dans $\boldsymbol{\pi}^\star$, nous étudions l'exemple de la *cellule mutable*, qui est le prototype des objets introduits ci-après.

Dans notre interprétation, une cellule mutable est une déclaration récursive, qui permet d'accéder et de modifier la valeur d'un nom qui est émis sur un canal privé. C'est un terme équivalent au processus nommé «channel-based reference cell» dans la thèse de D. Turner [130, section 3.7]. Soit $\mathbf{R}_{(s,x)}$ l'enregistrement suivant.

$$\mathbf{R}_{(s,x)} =_{\text{def}} [\, get = (sx \mid x), put = s \,] \tag{16.1}$$

La cellule de nom e, initialisée à la valeur v_0, est le processus $\mathbf{CELL}_e(v_0)$ donné par:

$$\mathbf{CELL}_e(v_0) \quad =_{\text{def}} \quad \mathbf{def}\ s = (\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{R}_{(s,x)}\rangle\ \mathbf{in}\ \langle e \Leftarrow \mathbf{R}_{(s,v_0)}\rangle$$

On voit que le processus $\mathbf{CELL}_e(v_0)$ est le résultat de l'application de la définition récursive: $\mathbf{rec}\, s.(\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{R}_{(s,x)}\rangle$, au nom v_0.

$$\begin{array}{rcl} (\mathbf{rec}\, s.(\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{R}_{(s,x)}\rangle\ v_0) & \equiv & \mathbf{def}\ s = (\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{R}_{(s,x)}\rangle\ \mathbf{in}\ (sv_0) \\ & \stackrel{*}{\rightarrow} & \mathbf{CELL}_e(v_0) \end{array}$$

Ainsi, intuitivement, une cellule est constituée de deux éléments. Une partie active, la déclaration $\langle e \Leftarrow \mathbf{R}_{(s,v_0)}\rangle$, qui peut interagir avec des messages émis sur le nom e. Une partie passive, la définition récursive sur s, qui permet de recréer une nouvelle déclaration. De plus, les processus P_{get} et P_{put}, ci-après

$$P_{get} =_{\text{def}} \mathbf{CELL}_e(v_0) \mid (e \cdot get) \qquad P_{put} =_{\text{def}} \mathbf{CELL}_e(v_0) \mid (e \cdot put\ v_1)$$

illustrent respectivement l'accès et l'écriture dans la cellule e. En effet on a les réductions suivantes:

$$\begin{array}{rcl} P_{get} & \equiv & \mathbf{def}\ s = (\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{R}_{(s,x)}\rangle\ \mathbf{in}\ (\langle e \Leftarrow \mathbf{R}_{(s,v_0)}\rangle \mid e \cdot get) \\ & \rightarrow & \mathbf{def}\ s = (\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{R}_{(s,x)}\rangle\ \mathbf{in}\ \big(\mathbf{R}_{(s,v_0)} \cdot get\big) \\ & \rightarrow & \mathbf{def}\ s = (\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{R}_{(s,x)}\rangle\ \mathbf{in}\ (sv_0 \mid v_0) \\ & \stackrel{*}{\rightarrow} & \mathbf{CELL}_e(v_0) \mid v_0 \\ \\ P_{put} & \stackrel{*}{\rightarrow} & \mathbf{def}\ s = (\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{R}_{(s,x)}\rangle\ \mathbf{in}\ (sv_1) \\ & \stackrel{*}{\rightarrow} & \mathbf{CELL}_e(v_1) \end{array}$$

Il est important de noter comment la référence e est utilisée «linéairement» dans le processus $\mathbf{CELL}_e(v_0)$. À chaque invocation de la cellule, l'unique déclaration $\langle e \Leftarrow \mathbf{R}_{(s,v_0)}\rangle$ qui est présente, est consommée et un unique message (sv_0), qui tient le rôle de sémaphore, est libéré. Ce message libère à son tour une unique déclaration $\langle e \Leftarrow \mathbf{R}_{(s,v_0)}\rangle$. Ainsi, on peut démontrer qu'il existe une seule déclaration disponible en e, et que celle-ci mémorise la dernière valeur transmise dans un message de la forme $(e \cdot put\ v)$.

16.2.1 Cellule n-aire

On peut proposer diverses extensions à l'exemple de la cellule. Par exemple on peut définir une «cellule à n valeurs», ou *cellule n-aire*. Ce processus utilise un enregistrement de $2n$ champs: $(put_1, \ldots, put_n)$ et $(get_1, \ldots, get_n)$, à la place de $\mathbf{R}_{(s,x)}$. On définit la cellule n-aire $\mathbf{NCELL}_e(\tilde{v})$ de

la manière suivante.

$$\mathbf{RN}_{(s,\tilde{x})} =_{\text{def}} \left[\begin{array}{l} \ldots \qquad\qquad\qquad\qquad\qquad\qquad i\in[1..n] \\ get_i = (sx_1 \ldots x_n \mid x_i), \\ put_i = (\boldsymbol{\lambda} y_i)(sx_1 \ldots y_i \ldots x_n), \\ \ldots \end{array}\right]$$

$$\mathbf{NCELL}_e(\tilde{v}) =_{\text{def}} \mathbf{def}\, s = (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathbf{RN}_{(s,\tilde{x})}\rangle \,\mathbf{in}\, \langle e \Leftarrow \mathbf{RN}_{(s,\tilde{v})}\rangle$$

La cellule n-aire nécessite un tuple de valeurs pour être initialisée. Sa réaction, lors de l'appel des méthodes *put* ou *get*, est similaire à celle de la cellule.

$$\mathbf{NCELL}_e(\tilde{v}) \mid e \cdot get_i \xrightarrow{*} \mathbf{NCELL}_e(\tilde{v}) \mid v_i$$

$$\mathbf{NCELL}_e(\tilde{v}) \mid e \cdot put_i\ v \xrightarrow{*} \mathbf{NCELL}_e(v_1, \ldots v, \ldots v_n)$$

16.2.2 Cellule clonable

On peut également définir une *cellule clonable*, c'est-à-dire une cellule munie d'un nouveau champ *clone*, telle que toute sélection sur ce champ, crée une nouvelle cellule ayant le même état. On nomme $\mathbf{CCELL}_e$ cette cellule. Dans la définition de ce processus, on utilise l'enregistrement $\mathbf{RC}_{(s,x,c)}$, défini ci-dessous, à la place de $\mathbf{R}_{(s,x)}$. On utilise aussi le processus $\mathbf{FCELL}_{(c,e)}$, qui peut s'interpréter comme une «fabrique de cellule».

$$\mathbf{RC}_{(s,x,c)} =_{\text{def}} \left[\begin{array}{l} get = (sx \mid x), \\ put = s \\ clone = (sx \mid cx) \end{array}\right]$$

$$\mathbf{FCELL}_{(c,e)} =_{\text{def}} \mathbf{rec}\, s.(\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{RC}_{(s,x,c)}\rangle$$

$$\mathbf{CCELL}_{(c,e)}(v_0) =_{\text{def}} \left(\begin{array}{l} \mathbf{def}\, c = (\boldsymbol{\lambda} x)(\boldsymbol{\nu} e')(\mathbf{FCELL}_{(c,e')}\ x \mid e') \\ \mathbf{in}\ \ \mathbf{def}\, s = (\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{RC}_{(s,x,c)}\rangle \\ \qquad \mathbf{in}\, \langle e \Leftarrow \mathbf{RC}_{(s,v_0)}\rangle \end{array}\right)$$

On voit que le processus $(\mathbf{FCELL}_{(c,e)} v_0)$ se réduit en $\mathbf{CELL'}_{(c,e)}(v_0)$, avec:

$$\mathbf{CELL'}_{(c,e)}(v_0) =_{\text{def}} \mathbf{def}\, s = (\boldsymbol{\lambda} x)\langle e \Leftarrow \mathbf{RC}_{(s,x,c)}\rangle \,\mathbf{in}\, \langle e \Leftarrow \mathbf{RC}_{(s,v_0,c)}\rangle$$

Le paramètre c de l'enregistrement $\mathbf{RC}_{(s,x,c)}$, représente le nom d'une définition récursive qui permet de créer une nouvelle cellule. C'est ce qui explique l'utilisation de la restriction $(\boldsymbol{\nu} e')(\ldots)$. Ainsi, lorsqu'on sélectionne le champ *clone* de la cellule de nom e, on obtient la réduction suivante.

$$\begin{array}{rcl} \mathbf{CCELL}_e(v_0) \mid (e \cdot clone) & \equiv & \mathbf{def}\, c = \cdots \,\mathbf{in}\, (\mathbf{def}\, s = \cdots \,\mathbf{in}\, \langle e \Leftarrow \mathbf{RC}_{(s,v_0,c)}\rangle \mid e \cdot clone) \\ & \xrightarrow{*} & \mathbf{def}\, c = \cdots \,\mathbf{in}\, (\mathbf{def}\, s = \cdots \,\mathbf{in}\, sv_0 \mid cv_0) \\ & \xrightarrow{*} & \mathbf{def}\, c = \cdots \,\mathbf{in}\, (\mathbf{CELL'}_{(c,e)}(v_0) \mid (\boldsymbol{\nu} e')(\mathbf{FCELL}_{(c,e')}\ v_0 \mid e')) \\ & \xrightarrow{*} & \mathbf{def}\, c = \cdots \,\mathbf{in}\, (\mathbf{CELL'}_{(c,e)}(v_0) \mid (\boldsymbol{\nu} e')(\mathbf{CELL'}_{(c,e')}(v_0) \mid e')) \end{array}$$

C'est-à-dire deux cellules en parallèle qui partagent le code de la «fonction fabrique cellule» $(\boldsymbol{\lambda} x)(\boldsymbol{\nu} e')(\mathbf{FCELL}_{e'}\ x \mid e')$. Si on utilise le théorème de réplication pour distribuer la définition sur le nom c, on peut simplifier le résultat de cette réduction. En effet, on peut alors montrer qu'on obtient deux cellules en parallèles.

$$(\mathbf{CCELL}_e\ v_0) \mid (e \cdot clone) \xrightarrow{*} \approx_b (\mathbf{CCELL}_e\ v_0) \mid (\boldsymbol{\nu} e')(\mathbf{CCELL}_{e'}\ v_0 \mid e')$$

On peut remarquer la présence du message e', en parallèle avec la cellule du même nom. Ce message nous permet d'interagir avec la cellule nouvellement crée: comme la cellule est crée avec un nouveau nom, c'est le seul moyen de pouvoir communiquer avec elle. Par exemple le processus

$(\mathbf{CCELL}_e\ v_0) \mid (e\ \cdot clone\ \cdot put\, v_1)$, clone la cellule e et écrit v_1 dans la cellule obtenue. Ceci à malheureusement comme effet secondaire, de nous faire perdre notre unique lien avec l'objet e'.

$$\begin{array}{lll} (\mathbf{CCELL}_e\ v_0) \mid (e \cdot clone\ \cdot put\ v_1) & \stackrel{*}{\rightarrow} \approx_b & (\mathbf{CCELL}_e\ v_0) \mid (\boldsymbol{\nu} e')(\mathbf{CCELL}_{e'}\ v_0 \mid e')\ \cdot put\ v_1 \\ & \equiv & (\mathbf{CCELL}_e\ v_0) \mid (\boldsymbol{\nu} e')(\mathbf{CCELL}_{e'}\ v_0 \mid e'\ \cdot put\ v_1) \\ & \stackrel{*}{\rightarrow} \equiv & (\mathbf{CCELL}_e\ v_0) \mid (\boldsymbol{\nu} e')(\mathbf{CCELL}_{e'}\ v_1) \end{array}$$

Dans le codage des objets, on utilise un processus qui est à la fois une cellule n-aire et une cellule clonable.

CHAPITRE 17

Un calcul d'objets concurrents

DANS CE CHAPITRE, nous définissons un ensemble de notations qui nous permettent de considérer que le calcul bleu est un calcul d'objets concurrents. Ces notations sont données dans la figure 17.1. Nous définissons dans la section 17.1 les processus du calcul bleu qui représentent ces opérateurs objets, et nous étudions leur comportement opérationnel. En particulier, nous montrons que les règles de réductions données dans la figure 17.1 correspondent à la définition des objets. Nous étudions aussi les règles de typage associées. Dans toute cette partie, nous considérons le calcul bleu dissymétrique et local. Cependant la majeure partie des codages sont encore valides dans le calcul symétrique.

Dans la définition de la syntaxe des objets, nous utilisons un sous-ensemble de références: $e,f,g,\cdots \in \mathcal{R}$, utilisées pour nommer les objets. Nous utilisons aussi la lettre L pour désigner le «corps d'un objet»: $L =_{\text{def}} [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in I}\,]$. Nous utilisons les mêmes notations pour les corps d'objets que pour les enregistrements et, en particulier, nous définissons une opération d'extension/modification d'un corps d'objet: $[\, L\,, l = (\boldsymbol{\lambda} x) P\,]$, qui est comparable à l'opération sur les enregistrements. L'opérateur le plus important que nous introduisons est la *dénomination*[1]: $\langle e \mapsto L\rangle$, qui est l'équivalent pour le codage des objets, de ce que la déclaration $\langle e = P\rangle$ est au codage des fonctions. Ces deux constructions sont très similaires, en particulier la règle (red invk), d'invocation d'une méthode, est proche de la règle de communication (red mdecl).

Bien qu'il n'y ait aucune notion de nom ou d'identité dans $\boldsymbol{\varsigma}$, ni de notion de parallélisme, les constructions introduites dans la figure 17.1 sont de manière évidente inspirées par les opérateurs du calcul de M. ABADI et L. CARDELLI. Cette filiation se reflète dans la simplicité de l'interprétation de $\mathbf{Ob}_{1<:}$, donnée dans le chapitre 18. 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 $\boldsymbol{\pi}^\star$ 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, dans lesquels on échange des références sur les objets, et pas du code. Nous verrons que c'est aussi le comportement du calcul d'objets impératif (cf. chapitre 19).

Un exemple d'objet que nous pouvons définir avec nos notations, est l'objet qui produit récursivement une infinité de copie de lui-même en parallèle. Soit L le corps

1. Ce nom est emprunté au calcul d'objets concurrents de [60], cf. chapitre 19.

opérateurs objets:

$\langle e \mapsto [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in[1..n]} \,]\rangle$	dénomination de nom e
$P \Leftarrow l$	invocation de la méthode l
$P \leftarrow l = (\boldsymbol{\lambda} x) Q$	modification de la méthode l
$\mathbf{clone}(P)$	clonage des objets dans P

réduction: soit L le corps d'objet $[\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in[1..n]} \,]$. Les règles de réduction données ici sont indicatives. Nous prouvons dans le théorème 17.1 que ces règles correspondent bien à la définition des opérateurs objets.

$$\frac{j \in [1..n]}{\langle e \mapsto L\rangle \mid e \Leftarrow l_j \to_\varsigma \langle e \mapsto L\rangle \mid P_j\{e/x_j\}} \text{ (red invk)}$$

$$\frac{j \in [1..n] \qquad L' =_{\text{def}} [\, L\,, l_j = (\boldsymbol{\lambda} x) P \,]}{\langle e \mapsto L\rangle \mid (e \leftarrow l_j = (\boldsymbol{\lambda} x) P) \to_\varsigma \langle e \mapsto L'\rangle \mid e} \text{ (red updt)}$$

$$\frac{f \notin \mathbf{fn}(L)}{\langle e \mapsto L\rangle \mid \mathbf{clone}(e) \to_\varsigma \langle e \mapsto L\rangle \mid (\boldsymbol{\nu} f)(\langle f \mapsto L\rangle \mid f)} \text{ (red clone)}$$

Fig. 17.1: Notations pour les objets dans $\boldsymbol{\pi}^\star$

$[\, rec = (\boldsymbol{\lambda} x)(x \Leftarrow rec \mid \mathbf{clone}(x)) \,]$, on a la réduction suivante (où f est un nouveau nom).

$$\begin{aligned} \langle e \mapsto L\rangle \mid e \Leftarrow rec &\to_\varsigma \langle e \mapsto L\rangle \mid (x \Leftarrow rec \mid \mathbf{clone}(x))\{e/x\} \\ &\to_\varsigma \langle e \mapsto L\rangle \mid e \Leftarrow rec \mid (\boldsymbol{\nu} f)(\langle f \mapsto L\rangle \mid f) \end{aligned}$$

La dénomination est très proche de l'intuition que nous avons des objets. Néanmoins il est possible de définir un processus dans lequel deux dénominations sur le même nom sont en parallèles, comme dans le processus $(\langle e \mapsto L\rangle \mid \langle e \mapsto L'\rangle \mid e \Leftarrow l)$. Ainsi, dans cet exemple, on ne peut pas statiquement prévoir laquelle des deux dénominations va répondre à la requête $e \Leftarrow l$. On ne peut donc pas assurer la propriété qu'un nom d'objet est associé à un unique processus, ce qui est une propriété très importante des langages à objets. On remarque qu'il s'agit d'une instance du problème plus général d'associer un «récepteur unique» à chaque nom dans $\boldsymbol{\pi}^\star$. Pour obtenir un processus plus proche de notre intuition de ce qu'est un objet, une solution est de restreindre la portée du nom d'une dénomination.

Définition 17.1 (Objet) Soit P un processus qui ne déclare pas le nom e, c'est-à-dire que $e \notin \mathbf{decl}(P)$, et qui n'utilise pas e pour nommer une dénomination. Un objet de nom e et de corps L, pour P, est le processus:

$$\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P \ =_{\text{def}}\ (\boldsymbol{\nu} e)(\langle e \mapsto L\rangle \mid P)$$

Ainsi, l'objet est à la dénomination, ce que la définition $(\mathbf{def}\, u = R\, \mathbf{in}\, P)$ est à la déclaration $\langle u = R\rangle$. Il faut noter que, moyennant une syntaxe plus compliquée, il est possible de définir des objets mutuellement dépendant. Un peu à la manière du processus $\mathbf{def}\, D\, \mathbf{in}\, P$, qui autorise les définitions mutuellement récursives.

Il existe une autre approche possible pour s'assurer de la propriété de récepteur unique, basée sur la définition d'un système de types spécifique. On peut retrouver cette approche, par exemple, dans les travaux de R. Amadio sur $\boldsymbol{\pi_1}$ [8], le π-calcul avec récepteur unique, ou dans ceux de A. Gordon et P. Hankin [60] sur une extension impérative et concurrente du calcul $\boldsymbol{\varsigma}$.

17.1 Interprétation des objets dans le calcul bleu

Le codage du processus modélisant une dénomination (cf. définition 17.3) est inspiré de l'exemple de la cellule mutable. On code une dénomination $\langle e \mapsto L \rangle$, où L est le corps $[\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\, i\in[1..n]}\,]$, par une cellule clonable à $2n$ valeurs. Le «champs d'accès»: get_{l_i} va permettre d'invoquer la méthode l_i, et le champ put_{l_i} va permettre de modifier cette méthode. Schématiquement, nous utilisons la technique nommée «*split-method*» dans [2]. Soit $\mathsf{E}_{(s,\tilde{x},c)}$ l'enregistrement suivant.

Définition 17.2 ($\mathsf{E}_{(s,\tilde{x},c)}$)

$$\mathsf{E}_{(s,\tilde{x},c)} =_{\text{def}} \left[\begin{array}{l} \ldots \qquad\qquad\qquad\qquad\qquad\qquad i\in[1..n] \\ get_{l_i} = (s\tilde{x} \mid x_i e), \\ put_{l_i} = (\boldsymbol{\lambda} y_i)(s x_1 \ldots y_i \ldots x_n \mid e), \\ \ldots \\ clone = (s\tilde{x} \mid c\tilde{x}) \end{array}\right]$$

Il est intéressant de comparer les enregistrements $\mathsf{E}_{(s,\tilde{x},c)}$ et $\mathsf{RC}_{(s,x)}$ (cf. section 16.2). La première différence est que, dans le premier enregistrement, on cherche à mémoriser n valeurs, et pas seulement une. On peut aussi noter des différences dans le champ d'accès: get_{l_i}, et le champ de modification: put_{l_i}.

- dans le champ get_{l_i}, on ne renvoie pas simplement la j^{e} valeur mémorisée, mais on l'applique au nom de l'objet. Ceci permet de coder le passage du nom de l'objet lors de l'invocation: $(\langle e \mapsto L\rangle \mid (e \Leftarrow\!\!\!\!\!\!\rightarrow l_j)) \xrightarrow{*} (\langle e \mapsto L\rangle \mid ((\boldsymbol{\lambda} x_j) P_j\ e))$;
- dans le champ put_{l_j}, on ne se contente pas de modifier la j^{e} valeur mémorisée, mais on renvoie également le nom de l'objet. Ceci permet de coder le comportement de la modification de méthode, qui retourne un «pointeur» vers le nom de l'objet modifié: $(\langle e \mapsto L\rangle \mid (e \leftarrow l_j = (\boldsymbol{\lambda} x) Q)) \xrightarrow{*} (\langle e \mapsto L'\rangle \mid e)$.

Comme dans les exemples de la section 16.2, on encapsule cet enregistrement dans une définition récursive qui produit une déclaration sur le nom e de l'objet.

$$\mathsf{Fobj}_{(c,e)} \quad =_{\text{def}} \quad \mathbf{def}\, s = (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)}\rangle\, \mathbf{in}\, s$$

Comme pour la définition de la cellule clonable, on a besoin de rajouter une définition sur le nom c, qui permet de dupliquer le code de l'objet. Pour simplifier les définitions suivantes, on décide de noter $\mathbf{Copy_n}$ le contexte d'évaluation suivant.

$$\mathbf{Copy_n} \ =_{\text{def}}\ \mathbf{def}\, c = (\boldsymbol{\lambda}\tilde{x})(\boldsymbol{\nu} f)(\mathsf{Fobj}_{(c,f)}\ x_1 \ldots x_n \mid f)\, \mathbf{in}\, [\_]$$

Une dénomination est alors une cellule initialisée avec «les valeurs» $((\boldsymbol{\lambda} x_i) P_i)_{i\in[1..n]}$.

Définition 17.3 (Dénomination) La dénomination $\langle e \mapsto [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\, i\in[1..n]}\,]\rangle$ est le processus du calcul bleu suivant.

$$\langle e \mapsto [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\, i\in[1..n]}\,]\rangle \ =\ \mathbf{Copy_n} \left[\begin{array}{l} \mathbf{def}\, u_1 = (\boldsymbol{\lambda} x_1) P_1, \ldots, u_n = (\boldsymbol{\lambda} x_n) P_n \\ \mathbf{in}\ \ \mathbf{def}\, s = (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)}\rangle \\ \qquad \mathbf{in}\, \langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)}\rangle \end{array}\right]$$

Où les noms $(u_i)_{i\in[1..n]}$ sont nouveaux.

La notation que nous avons choisi pour la dénomination n'est pas gratuite. En effet on verra dans le chapitre 19, que $\langle e \mapsto L \rangle$ est similaire à l'interprétation de la dénomination du calcul d'objets de A. Gordon et P. Hankin [60, 61].

Remarque En utilisant l'application d'ordre supérieure: $(PQ) =_{\text{def}} \mathbf{def}\, q = Q \,\mathbf{in}\, (Pq)$, on montre facilement que $\langle e \mapsto L \rangle$ s'obtient à partir du processus O, ci-dessous, par une suite de béta-réduction:

$$O =_{\text{def}} \mathbf{Copy_n}\left[\mathsf{Fobj}_{(c,e)}\ (\boldsymbol{\lambda} x_1)P_1 \ldots (\boldsymbol{\lambda} x_n)P_n\right]$$

En particulier, si on utilise la relation d'expansion définie dans le chapitre 10, on obtient que:

$$\langle e \mapsto L \rangle \ \lesssim_d\ \mathbf{Copy_n}[\mathsf{Fobj}_{(c,e)}\ (\boldsymbol{\lambda} x_1)P_1 \ldots (\boldsymbol{\lambda} x_n)P_n]$$

■

Une fois défini la dénomination, il est facile de définir les processus qui représentent les notations données dans la figure 17.1.

Définition 17.4 (Définition des opérateurs objets)

$$\begin{array}{rcl} (P \leftarrow l_j = (\boldsymbol{\lambda} x)Q) & = & (P \cdot put_{l_j}\ (\boldsymbol{\lambda} x)Q) \\ P \Leftarrow l_j & = & (P \cdot get_{l_j}) \\ \mathbf{clone}(P) & = & (P \cdot clone) \end{array}$$

Comme les objets sont dérivés de $\boldsymbol{\pi}^\star$, on obtient directement des règles d'équivalence structurelle pour les objets. En particulier, dans la définition de l'invocation, de la modification de méthodes et du clonage, nous n'utilisons que la sélection et l'application. Il est donc possible de dériver, pour ces opérations, les mêmes règles d'équivalence structurelle que pour l'application. Ainsi, on montre que les relations suivantes sont valides.

$$\begin{array}{rcl} (P \mid Q) \Leftarrow l & \equiv & P \mid (Q \Leftarrow l) \\ \mathbf{clone}(P \mid Q) & \equiv & P \mid \mathbf{clone}(Q) \\ (P \mid Q) \leftarrow l = (\boldsymbol{\lambda} x)R & \equiv & P \mid (Q \leftarrow l = (\boldsymbol{\lambda} x)R) \\ ((\boldsymbol{\nu} u)P) \Leftarrow l & \equiv & (\boldsymbol{\nu} u)(P \Leftarrow l) \\ \mathbf{clone}((\boldsymbol{\nu} u)P) & \equiv & (\boldsymbol{\nu} u)\mathbf{clone}(P) \\ ((\boldsymbol{\nu} u)P) \leftarrow l = (\boldsymbol{\lambda} x)Q & \equiv & (\boldsymbol{\nu} u)(P \leftarrow l = (\boldsymbol{\lambda} x)Q) \quad (u \notin \mathbf{fn}((\boldsymbol{\lambda} x)Q)) \end{array}$$

De plus, on montre que si $\mathbf{E}$ est un contexte d'évaluation, alors $(\mathbf{E} \Leftarrow l)$, et $(\mathbf{E} \leftarrow l = (\boldsymbol{\lambda} x)Q)$, et $\mathbf{clone}(\mathbf{E})$, sont aussi des contextes d'évaluation.

Nous montrons que les règles de réduction données dans la figure 17.1 sont dérivables à partir de la définition des notations objets dans le calcul bleu.

Théorème 17.1 *La définition des règles de réduction des objets est valide vis-à-vis du codage dans $\boldsymbol{\pi}^\star$: si $P \rightarrow_\varsigma P'$, alors il existe un terme Q tel que $P \xrightarrow{*} Q$ et $Q \sim_d P'$.*

Preuve On montre que les trois règles de réduction (red invk), (red updt) et (red clone), sont dérivables dans notre système. Dans cette preuve, on suppose que L désigne le corps d'objet $[\, l_i = (\boldsymbol{\lambda} x_i)P_i{}^{i \in [1..n]}\,]$.

- **cas (red invk):** soit P le processus $(\langle e \mapsto L \rangle \mid e \Leftarrow l_j)$. Nous montrons que si $j \in [1..n]$, alors $P \xrightarrow{*} \langle e \mapsto L \rangle \mid P_j\{e/x_j\}$. Pour simplifier le reste de la preuve, on définit le contexte

d'évaluation $\mathbf{F}$ par:

$$\mathbf{F} =_{\text{def}} \left(\begin{array}{l} \mathbf{def}\, c = (\boldsymbol{\lambda}\tilde{x})(\boldsymbol{\nu} f)(\mathsf{Fobj}_{(c,f)}\ \tilde{x} \mid f), \\ \mathbf{in} \left(\begin{array}{l} \mathbf{def}\, u_1 = (\boldsymbol{\lambda} x_1)P_1, \ldots, u_n = (\boldsymbol{\lambda} x_n)P_n \\ \mathbf{in}\, (\mathbf{def}\, s = (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)}\rangle\, \mathbf{in}\, [\_]\,) \end{array} \right) \end{array} \right)$$

Nous avons déjà noté que le terme $\mathbf{F}[s\tilde{u}]$ se réduisait vers $\langle e \mapsto L\rangle$. Aussi on dérive les réductions suivantes dans $\boldsymbol{\pi}^\star$.

$$\begin{array}{rcl} P & =_{\text{def}} & \mathbf{F}[\langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)}\rangle \mid e \cdot get_{l_j}] \\ & \to_{(\rho)} & \mathbf{F}[\mathsf{E}_{(s,\tilde{u},c)} \cdot get_{l_j}] \\ & \to_{(\beta)} & \mathbf{F}[s\tilde{u} \mid u_j e] \\ & \to_{(\rho)} & \mathbf{F}[s\tilde{u} \mid ((\boldsymbol{\lambda} x_j)P_j)e] \\ & \to_{(\beta)} & \mathbf{F}[s\tilde{u} \mid P_j\{e/x_j\}] \\ & \equiv & \mathbf{F}[s\tilde{u}] \mid P_j\{e/x_j\} \\ & \overset{*}{\to} & \langle e \mapsto L\rangle \mid P_j\{e/x_j\} \end{array}$$

– **cas (red updt):** soient L' le corps $[\, L\, , l_j = (\boldsymbol{\lambda} x)Q\,]$, et j un indice dans l'intervalle $[1..n]$, et P le processus $(\langle e \mapsto L\rangle \mid (e \leftarrow l_j = (\boldsymbol{\lambda} x)P))$:

$$\begin{array}{rcl} P & \equiv & \mathbf{F}[\langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)}\rangle \mid e \cdot put_{l_j}\ (\boldsymbol{\lambda} x)Q] \\ & \to_{(\rho)} & \mathbf{F}[\mathsf{E}_{(s,\tilde{u},c)} \cdot put_{l_j}\ (\boldsymbol{\lambda} x)Q] \\ & \to_{(\beta)} & \mathbf{F}[\mathbf{def}\, y = (\boldsymbol{\lambda} x)Q\, \mathbf{in}\, ((\boldsymbol{\lambda} y_i)(su_1 \ldots y_i \ldots u_n \mid e))y] \\ & \to_{(\beta)} & \mathbf{F}[\mathbf{def}\, y = (\boldsymbol{\lambda} x)Q\, \mathbf{in}\, (su_1 \ldots y \ldots u_n \mid e)] \\ & \sim_d & \langle e \mapsto L'\rangle \mid e \end{array}$$

Dans la dernière relation, on utilise la bisimulation $\sim_d$ pour éliminer la définition $\{u_j = (\boldsymbol{\lambda} x_j)P_j\}$ du contexte $\mathbf{F}$ (règle de «garbage collection», cf. lemme 8.2).

– **cas (red clone):** dans le cas du clonage, on suppose $f \notin \mathbf{fn}(L)$. Soit P le processus $(\langle e \mapsto L\rangle \mid \mathbf{clone}(e))$.

$$\begin{array}{rcl} P & \equiv & \mathbf{F}[\langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)}\rangle \mid e \cdot clone] \\ & \to_{(\rho)} & \mathbf{F}[\mathsf{E}_{(s,\tilde{u},c)} \cdot clone] \\ & \to_{(\beta)} & \mathbf{F}[s\tilde{u} \mid c\tilde{u}] \\ & \to_{(\rho)} & \mathbf{F}[s\tilde{u} \mid ((\boldsymbol{\lambda}\tilde{x})(\boldsymbol{\nu} f)(\mathsf{Fobj}_{(c,f)}\ \tilde{x} \mid f))\tilde{u}] \\ & \overset{*}{\to} & \mathbf{F}[s\tilde{u} \mid (\boldsymbol{\nu} f)(\mathsf{Fobj}_{(c,f)}\ \tilde{u} \mid f)] \\ & \sim_d & \mathbf{F}[s\tilde{u}] \mid (\boldsymbol{\nu} f)(\mathbf{F}[(\mathsf{Fobj}_{(c,f)}\ \tilde{u})] \mid f) \\ & \overset{*}{\to} & \langle e \mapsto L\rangle \mid (\boldsymbol{\nu} f)(\langle f \mapsto L\rangle \mid f) \end{array}$$

Dans l'avant-dernière relation, on utilise la bisimulation pour distribuer les définitions sur les noms $\tilde{u}$ et c (règle de réplication, cf. lemme 8.2).

□

Les notations objets sont des processus à part entière du calcul bleu, on peut donc utiliser ces notations dans des contextes qui ne correspondent pas à «l'utilisation normale» d'un objet. Par exemple, on peut modifier une méthode par un terme qui n'est pas une fonction du paramètre self: $(P \cdot put_{l_j}\ \mathbf{0})$. On ne peut donc pas formuler de résultat inverse pour le théorème 17.1.

17.2 Système de types simples

Dans cette section, nous établissons un typage dérivé pour les objets à partir de leur définition dans $\boldsymbol{\pi}^\star$. Afin de simplifier la présentation, nous commençons par définir une notation pour le

type des objets. Ce type est très similaire aux types des objets obtenus dans les codages de R. Viswanathan [137] et D. Sangiorgi [118].

Définition 17.5 (Type des objets) On dénote $(\mathbf{obj}\,\alpha.[\,l_i : \vartheta_i^{i\in[1..n]}\,])$, le type récursif:

$$(\mathbf{obj}\,\alpha.[\,l_i : \vartheta_i^{i\in[1..n]}\,]) =_{\text{def}} \mu\alpha.\left[\begin{array}{l} \ldots \\ get_{l_i} = \vartheta_i, \\ put_{l_i} = (\alpha \to \vartheta_i) \to \alpha, \\ \ldots \\ clone = \alpha \end{array}\right]^{i\in[1..n]}$$

Nous notons aussi ce type $(\mathbf{obj}\,[\,l_i : \vartheta_i^{i\in[1..n]}\,])$, lorsque la variable α – on utilisera le terme de *type Self* – n'apparaît pas dans les ϑ_i.

Soit α une variable de type et soit ϱ le type $[\,l_i : \vartheta_i^{i\in[1..n]}\,]$. La variable α peut être libre dans les types de $(\vartheta_i)_{i\in[1..n]}$. Nous montrons que le type de l'enregistrement $\mathsf{E}_{(s,\tilde{x},c)}$ est $(\mathbf{obj}\,\alpha.\varrho)$, sous l'hypothèse que les noms x_i sont du type: $(\alpha \to \vartheta_i)\{(\mathbf{obj}\,\alpha.\varrho)/\alpha\}$. Soit Γ l'environnement suivant.

$$\begin{array}{rl} \Gamma =_{\text{def}} & \Delta, e : \tau, c : (\tau \to \vartheta_1\{\tau/\alpha\}) \to \cdots \to (\tau \to \vartheta_n\{\tau/\alpha\}) \to \tau, \\ & s : (\tau \to \vartheta_1\{\tau/\alpha\}) \to \cdots \to (\tau \to \vartheta_n\{\tau/\alpha\}) \to \mathsf{o}, \\ & x_1 : (\tau \to \vartheta_1\{\tau/\alpha\}), \ldots, x_n : (\tau \to \vartheta_n\{\tau/\alpha\}) \end{array}$$

Dans cet environnement, on peut typer le message $(s\tilde{x})$ avec le type processus: $\Gamma \vdash s\tilde{x} : \mathsf{o}$, le message $(c\tilde{x})$ avec le type de e, c'est-à-dire le type τ, et le message $(x_i e)$ avec le type $\vartheta_i\{\tau/\alpha\}$. Ceci permet de montrer que:

$$\Gamma \vdash \mathsf{E}_{(s,\tilde{x},c)} : \left[\begin{array}{l} \ldots \\ get_{l_i} = \vartheta_i\{\tau/\alpha\}, \\ put_{l_i} = (\tau \to \vartheta_i\{\tau/\alpha\}) \to \tau, \\ \ldots \\ clone = \tau \end{array}\right]^{i\in[1..n]}$$

Ce raisonnement vaut si τ est le type $(\mathbf{obj}\,\alpha.\varrho)$, et donc:

$$\Gamma \vdash \mathsf{E}_{(s,\tilde{x},c)} : \left[\begin{array}{l} \ldots \\ get_{l_i} = \vartheta_i, \\ put_{l_i} = (\alpha \to \vartheta_i) \to \alpha, \\ \ldots \\ clone = \alpha \end{array}\right]^{i\in[1..n]} \{(\mathbf{obj}\,\alpha.\varrho)/\alpha\} \qquad (17.1)$$

c'est-à-dire que $\mathsf{E}_{(s,\tilde{x},c)}$ à le type $(\mathbf{obj}\,\alpha.\varrho)$. On se convainc facilement de la nécessité d'utiliser un type récursif pour typer les objets: dans la définition de $\langle e \mapsto L\rangle$, on utilise la déclaration $\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)}\rangle$ (cf. la définition de Fobj_e). Pour qu'un objet soit bien typé, il faut donc que $(\mathbf{obj}\,\alpha.\varrho)$ soit aussi le type de e.

17.2.1 Typage des objets

On s'occupe maintenant du typage des objets. On va étudier les règles dérivées de typage pour chaque opérateurs du calcul d'objets. Les règles que nous déduisons dans cette section sont réunies dans la figure 17.2. Dans ce système, on a choisi de ne pas utiliser de type objets récursifs, c'est-à-dire qu'on utilise le type $(\mathbf{obj}\,\varrho)$ et pas $(\mathbf{obj}\,\alpha.\varrho)$, ceci permet de souligner la similitude avec le système de types de $\mathbf{Ob}_{1<:}$ (cf. figure 16.2).

Soit ϱ le type enregistrement $[\, l_i : \vartheta_i^{\,i\in[1..n]} \,]$

$$\frac{\Gamma,x_i : (\mathbf{obj}\ \varrho) \vdash P_i : \vartheta_i \qquad (e : (\mathbf{obj}\ \varrho)) \in \Gamma}{\Gamma \vdash \langle e \mapsto [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in[1..n]} \,]\rangle : \mathsf{o}} \text{ (type denom)}$$

$$\frac{\Gamma,x_i : (\mathbf{obj}\ \varrho) \vdash P_i : \vartheta_i \qquad (e : (\mathbf{obj}\ \varrho)) \in \Gamma \qquad \Gamma \vdash P : \tau}{\Gamma \vdash \mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in[1..n]} \,]\, \mathbf{in}\ P : \tau} \text{ (type obj)}$$

$$\frac{\Gamma,x : (\mathbf{obj}\ \varrho) \vdash Q : \vartheta_j \qquad \Gamma \vdash P : (\mathbf{obj}\ \varrho) \qquad j \in [1..n]}{\Gamma \vdash P \leftarrow l_j = (\boldsymbol{\lambda} x) Q : (\mathbf{obj}\ \varrho)} \text{ (type updt)}$$

$$\frac{\Gamma \vdash P : (\mathbf{obj}\ \varrho)}{\Gamma \vdash \mathbf{clone}(P) : (\mathbf{obj}\ \varrho)} \text{ (type clone)} \qquad \frac{\Gamma \vdash P : (\mathbf{obj}\ \varrho) \qquad j \in [1..n]}{\Gamma \vdash P \Leftarrow l_j : \vartheta_j} \text{ (type invk)}$$

Fig. 17.2: Système de types pour les objets (sans le type Self)

Remarque Dans tout le reste de cette section, Υ est l'environnement:

$$\begin{array}{rl} \Upsilon =_{\text{def}} & \Gamma,c : ((\alpha \to \vartheta_1) \to \cdots \to (\alpha \to \vartheta_n) \to \alpha)\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}, \\ & s : ((\alpha \to \vartheta_1) \to \cdots \to (\alpha \to \vartheta_n) \to \mathsf{o})\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}, \\ & x_1 : (\alpha \to \vartheta_1)\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}, \ldots, x_n : (\alpha \to \vartheta_n)\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\} \end{array}$$

On suppose aussi que $(e : (\mathbf{obj}\ \alpha.\varrho)) \in \Gamma$, et que $\Upsilon \vdash *$. ■

Typage de la dénomination.

En utilisant le jugement de l'équation (17.1) avec la règle (type decl), on montre que $\Upsilon \vdash \langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)} \rangle : \mathsf{o}$. Par conséquent, en utilisant n fois la règle (type abs), nous montrons que:

$$\Upsilon_{|\tilde{x}} \vdash (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)} \rangle : ((\alpha \to \vartheta_1) \to \cdots \to (\alpha \to \vartheta_n) \to \mathsf{o})\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}$$

qui est le type de s. Par conséquent:

$$\Upsilon_{|s,\tilde{x}},u_1 : (\alpha \to \vartheta_1)\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}, \ldots,u_n : (\alpha \to \vartheta_n)\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\} \vdash \left(\begin{array}{l} \mathbf{def}\ s = (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)} \rangle \\ \mathbf{in}\ \langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)} \rangle \end{array} \right) : \mathsf{o}$$

et, en utilisant la règle dérivée (type rec) pour le typage des définitions récursives (cf. chapitre III 11.2), on montre aussi que:

$$\Upsilon_{|s,\tilde{x}} \vdash \mathsf{Fobj}_e : ((\alpha \to \vartheta_1) \to \cdots \to (\alpha \to \vartheta_n) \to \mathsf{o})\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}$$

Le dernier jugement permet de montrer que le terme $(\boldsymbol{\lambda}\tilde{x})(\boldsymbol{\nu} f)(\mathsf{Fobj}_f\ \tilde{x} \mid f)$ a le même type que c.

$$\frac{\Upsilon_{|s,\tilde{x}},f : (\mathbf{obj}\ \alpha.\varrho),x_1 : (\alpha \to \vartheta_1)\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}, \ldots \vdash (\mathsf{Fobj}_f\ \tilde{x}) : \mathsf{o}}{\frac{\Upsilon_{|s,\tilde{x}},f : (\mathbf{obj}\ \alpha.\varrho),x_1 : (\alpha \to \vartheta_1)\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}, \ldots \vdash (\mathsf{Fobj}_f\ \tilde{x} \mid f) : (\mathbf{obj}\ \alpha.\varrho)}{\Upsilon_{|s,\tilde{x}} \vdash (\boldsymbol{\lambda}\tilde{x})(\boldsymbol{\nu} f)(\mathsf{Fobj}_f\ \tilde{x} \mid f) : ((\alpha \to \vartheta_1) \to \cdots \to (\alpha \to \vartheta_n) \to \alpha)\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}}}$$

En particulier, si $\Upsilon_{|s,\tilde{x}} \vdash P : \tau$, on montre en utilisant la règle (type def) que: $\Gamma \vdash \mathbf{Copy_n}[P] : \tau$. Par conséquent, si pour chaque $i \in [1..n]$ on a $\Gamma \vdash (\boldsymbol{\lambda} x_i) P_i : (\alpha \to \vartheta_i)\{(\mathbf{obj}\ \alpha.\varrho)/\alpha\}$, nous pouvons conclure que:

$$\Gamma \vdash \mathbf{Copy_n} \left[\begin{array}{l} \mathbf{def}\ u_1 = (\boldsymbol{\lambda} x_1) P_1, \ldots, u_n = (\boldsymbol{\lambda} x_n) P_n \\ \mathbf{in}\ \ \mathbf{def}\ s = (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)} \rangle \\ \qquad \mathbf{in}\ \langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)} \rangle \end{array} \right] : \mathsf{o}$$

ce qui établit la règle (type denom). La règle (type obj), de typage des objets, se déduit simplement de (type denom) en utilisant les règles (type par) et (type new).

Typage de la modification.

On rappelle que $(P \leftarrow l_j = (\boldsymbol{\lambda} x) Q) =_{\text{def}} (P \cdot put_{l_j} (\boldsymbol{\lambda} x) Q)$. Supposons que j soit un indice de l'intervalle $[1..n]$ et que:

$$\begin{cases} \Gamma \vdash P : (\mathbf{obj}\, \alpha.\varrho) \\ \Gamma, x : (\mathbf{obj}\, \alpha.\varrho) \vdash Q : \vartheta_j\{(\mathbf{obj}\, \alpha.\varrho)/\alpha\} \end{cases}$$

Le type de Q implique que $\Gamma \vdash (\boldsymbol{\lambda} x) Q : (\alpha \to \vartheta_j)\{(\mathbf{obj}\, \alpha.\varrho)/\alpha\}$, et le type de P implique que $\Gamma \vdash P \cdot put_{l_j} : ((\alpha \to \vartheta_j) \to \alpha)\{(\mathbf{obj}\, \alpha.\varrho)/\alpha\}$. Par conséquent:

$$\Gamma \vdash P \cdot put_{l_j}\ (\boldsymbol{\lambda} x) Q : (\mathbf{obj}\, \alpha.\varrho)$$

ce qui établit la règle (type updt). On remarque qu'on ne peut pas étendre l'objet P, en effet l'ensemble des champs put_{l_j} est figé. On remarque aussi qu'on ne peut pas modifier une méthode par un processus ayant un type plus précis: on ne peut pas raffiner le type d'une méthode. En effet dans un type objet, le type ϑ de la méthode l_j, apparaît en position contravariante dans le type du champ put_{l_j}, et en position covariante dans le type du champ get_{l_j}. Nous reviendrons sur ce point dans l'étude du sous-typage des objets.

Typage de l'invocation.

On rappelle que $(P \Leftarrow\!\!\!\!\!\!\rightarrow l_j) =_{\text{def}} (P \cdot get_{l_j})$. Supposons que $\Gamma \vdash P : (\mathbf{obj}\, \alpha.\varrho)$, et que $j \in [1..n]$. Il est clair que $(\mathbf{obj}\, \alpha.\varrho) \sim [\ldots, get_{l_j} = \vartheta_j\{(\mathbf{obj}\, \alpha.\varrho)/\alpha\}]$. Il suffit alors d'utiliser la règle (type sel) pour montrer que $\Gamma \vdash (P \cdot get_{l_j}) : \vartheta_j\{(\mathbf{obj}\, \alpha.\varrho)/\alpha\}$, ce qui établit la règle (type invk).

Typage du clonage.

Si on utilise la relation d'équivalence entre types, on montre facilement que: $(\mathbf{obj}\, \alpha.\varrho) \sim [\ldots, clone = (\mathbf{obj}\, \alpha.\varrho)]$. Et par conséquent $\Gamma \vdash P : (\mathbf{obj}\, \alpha.\varrho)$, implique que $\Gamma \vdash P \cdot clone : (\mathbf{obj}\, \alpha.\varrho)$. Ce qui établit la règle (type clone).

17.2.2 Sous-typage pour les objets

La notion de substitutivité des objets, c'est-à-dire la possibilité d'utiliser un objet avec plus de méthodes là où un objet avec moins de méthodes est demandé, est un élément important pour permettre la *réutilisation du code* dans les langages orientés objets. Comme le polymorphisme dans ML ou le sous-typage pour les enregistrements, la substitutivité permet de définir des fonctions se comportant uniformément sur des entrées de types différents.

Comme nous l'avons déjà remarqué, il est clair que le type d'un objet n'est pas covariant, c'est-à-dire que $\varrho \leqslant \sigma$ n'implique pas $(\mathbf{obj}\, \varrho) \leqslant (\mathbf{obj}\, \sigma)$. En effet, le type 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 $[\, l_i : {\vartheta_i}^{i \in [1..n]}\,]$ et $[\, l_i : {\vartheta_i}^{i \in [1..n+m]}\,]$. On montre qu'un processus de type $(\mathbf{obj}\, \varrho_2)$ peut être utilisé partout où un processus de type $(\mathbf{obj}\, \varrho_1)$ peut l'être.

Lemme 17.2 (Substitutivité) $(\mathbf{obj}\, [\, l_i : {\vartheta_i}^{i \in [1..n+m]}\,]) \leqslant (\mathbf{obj}\, [\, l_i : {\vartheta_i}^{i \in [1..n]}\,])$

Preuve La preuve utilise le lemme 12.1 et la règle de sous-typage de la récursion (sub rec) (cf. figure 14.3). On montre la propriété dans le cas ou $m = 1$, les autres cas sont similaires.

Soit α_1 et α_2 deux variables de type. Avec l'hypothèse que $\alpha_1 \leqslant \alpha_2$, on peut montrer que pour tout type ϑ ne contenant ni α_1, ni α_2, on a: $((\alpha_1 \to \vartheta) \to \alpha_1) \leqslant ((\alpha_2 \to \vartheta) \to \alpha_2)$. Par conséquent,

en utilisant le lemme 12.1, on montre que:

$$\left[\begin{array}{l} get_{l_1} = \vartheta_1, \\ put_{l_1} = (\alpha_1 \to \vartheta_1) \to \alpha_1 \\ \ldots \\ get_{l_{n+1}} = \vartheta_{n+1}, \\ put_{l_{n+1}} = (\alpha_1 \to \vartheta_{n+1}) \to \alpha_1 \\ clone = \alpha_1 \end{array}\right] \leqslant \left[\begin{array}{l} get_{l_1} = \vartheta_2, \\ put_{l_1} = (\alpha_2 \to \vartheta_1) \to \alpha_1 \\ \ldots \\ get_{l_n} = \vartheta_n, \\ put_{l_n} = (\alpha_2 \to \vartheta_n) \to \alpha_2 \\ clone = \alpha_2 \end{array}\right]$$

et donc on montre que $(\mathbf{obj}\,[\,l_i : {\vartheta_i}^{i\in[1..n+1]}\,]) \leqslant (\mathbf{obj}\,[\,l_i : {\vartheta_i}^{i\in[1..n]}\,])$. □

En utilisant cette même preuve, on montre en fait un résultat plus précis. Si pour tout indice i de l'intervalle $[1..n]$ on a $\alpha_1 \leqslant \alpha_2$ implique $\vartheta_i\{\alpha_1/\alpha\} \leqslant \vartheta_i\{\alpha_2/\alpha\}$, c'est-à-dire si α apparaît en position covariante dans les ϑ_i, alors: $(\mathbf{obj}\,\alpha.[\,l_i : {\vartheta_i}^{i\in[1..n+m]}\,]) \leqslant (\mathbf{obj}\,\alpha.[\,l_i : {\vartheta_i}^{i\in[1..n]}\,])$. C'est la règle de sous-typage qu'on retrouve dans le calcul $\mathbf{Ob}_{1<:}$ étendu avec le type Self.

Proposition 17.3 *Si pour tout indice $i \in [1..n]$, la variable α est covariante ϑ_i, alors:* $(\mathbf{obj}\,\alpha.[\,l_i : {\vartheta_i}^{i\in[1..n+m]}\,]) \leqslant (\mathbf{obj}\,\alpha.[\,l_i : {\vartheta_i}^{i\in[1..n]}\,])$.

CHAPITRE 18

Interprétation des objets fonctionnels

DANS CE CHAPITRE, nous étudions plus formellement le rapport entre notre calcul d'objets concurrents et le calcul d'objets fonctionnel de M. ABADI et L. CARDELLI $\mathbf{Ob}_{1<:}$. En particulier nous donnons une interprétation de $\mathbf{Ob}_{1<:}$ dans le calcul bleu, et nous prouvons que la réduction et le typage des termes de $\mathbf{Ob}_{1<:}$ sont simulés dans $\boldsymbol{\pi}^\star$.

Le codage des termes de $\mathbf{Ob}_{1<:}$ est direct et simple, en particulier on n'a pas besoin d'utiliser la composition parallèle, et la restriction n'est utilisée que pour donner un nom distinct à chaque objet. Notre codage est plus simple que celui de [137], par exemple, car nous avons factorisé une partie de l'interprétation dans la définition des constructions pour les objets.

Définition 18.1 (Codage des termes de $\mathbf{Ob}_{1<:}$)

$$\begin{array}{rcl}
[\![[\, l_i = \varsigma(x_i) f_i{}^{i\in[1..n]} \,]]\!] & = & \mathbf{obj}\, e\, \mathbf{is}\, [\, l_i = (\boldsymbol{\lambda} x_i)[\![f_i]\!]^{i\in[1..n]} \,]\, \mathbf{in}\ e \\
[\![e{\cdot}l \Leftarrow \varsigma(x) f]\!] & = & ([\![e]\!] \leftarrow l = (\boldsymbol{\lambda} x)[\![f]\!]) \\
[\![x]\!] & = & \mathbf{clone}(x) \\
[\![e{\cdot}l]\!] & = & [\![e]\!] \Leftarrow l
\end{array}$$

Le codage des types et des jugements de type de $\mathbf{Ob}_{1<:}$ est tout aussi direct. On définit l'interprétation des environnements de $\mathbf{Ob}_{1<:}$ par les deux relations $[\![\emptyset]\!] = \emptyset$ et $[\![E,x : A]\!] = [\![E]\!],x : [\![A]\!]$, et le codage des types par:

Définition 18.2 (Codage des types de $\mathbf{Ob}_{1<:}$)

$$[\![[\, l_i : B_i{}^{i\in[1..n]} \,]]\!] = (\mathbf{obj}\, [\, l_i : [\![B_i]\!]^{i\in[1..n]} \,]) \qquad [\![\top]\!] = \top$$

Dans la suite, on suppose que les environnements $[\![E]\!]$ et les types $[\![A]\!]$ sont bien formés.

Avant de prouver que notre codage est complet: le résultat formel est donné dans le théorème 18.6, nous montrons un résultat intermédiaire. En effet, comme dans le cas de l'interprétation du λ-calcul, il nous faut montrer l'équivalence entre un mécanisme de réduction par substitution de code (le cas de $\mathbf{Ob}_{1<:}$), et un mécanisme de réduction par passage de noms (le cas

de $\boldsymbol{\pi}^\star$). Ainsi, dans le cas de l'interprétation de l'invocation de méthode par exemple, il nous faut montrer que la substitution d'un objet à une variable, est équivalent à fournir une référence à cet objet. Plus formellement, il nous faut montrer que si $[\![v]\!] = \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ e$, alors:

$$\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ ([\![f]\!]\{^e/_x\}) \ \approx_b\ [\![f\{^v/_x\}]\!]$$

On prouve cette relation en démontrant un équivalent pour les objets des lois de réplication pour les définitions que nous avons prouvées dans la partie II. Ces lois renforcent le parallèle déjà établi entre déclarations et dénominations, et entre définitions et objets.

18.1 Lois de réplication pour les objets

On remarque que dans les réductions qui impliquent les objets, on retrouve un certain nombre de «séquences de réduction déterministes». C'est le cas, par exemple, lorsqu'on émet un message sur le canal c qui commande le clonage d'un objet: voir la preuve du théorème 17.1. Plus formellement, on montre la propriété suivante.

Lemme 18.1 *On suppose que les arités des tuples sont correctes. Soit* $\mathbf{F}$ *un contexte d'évaluation qui ne capture pas le nom* c*, alors*[1]*:*

$$(\mathbf{Copy} \circ \mathbf{F})[(c\ \tilde{u})] \ \gtrsim_d\ (\mathbf{Copy} \circ \mathbf{F})[(\boldsymbol{\nu} f)(\mathbf{Fobj}_{(c,f)}\ \tilde{u} \mid f)] \tag{18.1}$$

Si $\mathbf{F}$ *est le contexte* $\mathbf{def}\, u_1 = (\boldsymbol{\lambda} x_1)P_1, \ldots, u_n = (\boldsymbol{\lambda} x_n)P_n\, \mathbf{in}\, \mathbf{E}'$*, où* $\mathbf{E}'$ *est un contexte d'évaluation, alors:*

$$(\mathbf{Copy} \circ \mathbf{F})[(\boldsymbol{\nu} f)(\mathbf{Fobj}_{(c,f)}\ \tilde{u} \mid f)] \ \gtrsim_d\ (\mathbf{Copy} \circ \mathbf{F})[\mathbf{obj}\, f\, \mathbf{is}\, L\, \mathbf{in}\ f] \tag{18.2}$$

Preuve Les processus des relations (18.1) et (18.2), sont obtenus par des béta-réductions et des communications avec des définitions. Le résultat découle du lemme 10.3. □

Nous montrons deux propriétés obtenues par application du lemme 8.2-(i) et de la proposition 4.4 aux objets.

Lemme 18.2 (Propriété générale de l'opérateur objet)

- ***garbage collection et objets:*** *si* P *est un processus qui ne contient pas le nom* e*, alors:* $\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P \sim_d P$.
- ***clonage et objets:*** *si* L *est un corps d'objet qui ne contient pas le nom* e*, alors:* $\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\, \mathbf{clone}(e) \ \approx_b\ \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ e$

Preuve La première propriété correspond à la loi de «garbage collection » des définitions. Elle se prouve, très simplement, en montrant que la relation $\{(\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P, P)\,|P \in \boldsymbol{\pi}^\star\}$ est une def-simulation. Pour prouver la seconde propriété, nous montrons que la relation $\mathcal{D}$, définie ci-dessous, est une expansion modulo $\lesssim_d$ (cf. conjecture 10.2).

$$\mathcal{D} = \left\{ (R_1, R_2) \;\middle|\; \begin{array}{l} R_1 = \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ e \\ R_2 = \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\, \mathbf{clone}(e) \\ L \text{ ne contient pas } e \end{array} \right\}$$

Il est clair que $\mathcal{D}$ est close par substitutions. Il nous reste donc à montrer que c'est une ground expansion. Avec les définitions données dans le chapitre précédent, le processus R_2 s'écrit

$$R_2 =_{\text{def}} (\boldsymbol{\nu} e)\left(\mathbf{Copy_n} \left[\left(\begin{array}{l} \mathbf{def}\, u_1 = (\boldsymbol{\lambda} x_1)P_1, \ldots, u_n = (\boldsymbol{\lambda} x_n)P_n \\ \mathbf{in}\ \ \mathbf{def}\, s = (\boldsymbol{\lambda} \tilde{x})\langle e \Leftarrow \mathbf{E}_{(s,\tilde{x},c)}\rangle \\ \qquad \mathbf{in}\, \langle e \Leftarrow \mathbf{E}_{(s,\tilde{u},c)}\rangle \end{array} \right) \right] \mid e \cdot clone \right)$$

1. Nous rappelons que $(\mathbf{F} \circ \mathbf{G})$ désigne le contexte $\mathbf{F}[\mathbf{G}]$.

Supposons que R_2 fasse une transition. La seule transition possible correspond à une communication entre le message $(e \cdot clone)$ et la déclaration $\langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)} \rangle$. Dans ce cas, le processus R_2 fait une τ-transition et on a:

$$R_2 \stackrel{\tau}{\longmapsto} R_2' \quad = \quad (\boldsymbol{\nu}\, e)\left(\mathbf{Copy_n} \left[\begin{array}{l} \mathbf{def}\ u_1 = (\boldsymbol{\lambda} x_1)P_1, \ldots, u_n = (\boldsymbol{\lambda} x_n)P_n \\ \mathbf{in}\ \ \mathbf{def}\ s = (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)} \rangle \\ \qquad \mathbf{in}\ (\mathsf{E}_{(s,\tilde{u},c)}\ \cdot clone) \end{array} \right] \mid \mathbf{0} \right)$$

Il existe donc un contexte d'évaluation $\mathbf{F}$ tel que

$$\begin{array}{lcll} R_2 \stackrel{\tau}{\longmapsto} R_2' & \equiv & (\mathbf{Copy} \circ \mathbf{F})[\mathsf{E}_{(s,\tilde{u},c)}\ \cdot clone] & \\ & \gtrsim_d & (\mathbf{Copy} \circ \mathbf{F})[(s\tilde{u}) \mid (c\tilde{u})] & \\ & \gtrsim_d & (\mathbf{Copy} \circ \mathbf{F})[((\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)} \rangle)\ \tilde{u} \mid (c\tilde{u})] & \text{(communicaton sur } s) \\ & \gtrsim_d & (\mathbf{Copy} \circ \mathbf{F})[\langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)} \rangle \mid (c\tilde{u})] & \text{(béta-réduction)} \\ & \gtrsim_d & (\mathbf{Copy} \circ \mathbf{F})[\langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)} \rangle \mid (\mathbf{obj}\, f\, \mathbf{is}\, L\, \mathbf{in}\ f)] & \text{(lemme 18.1)} \\ & \equiv & (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ \mathbf{0}) \mid (\mathbf{obj}\, f\, \mathbf{is}\, L\, \mathbf{in}\ f) & \\ & \sim_d & (\mathbf{obj}\, f\, \mathbf{is}\, L\, \mathbf{in}\ f) & \text{(«garbage collection»)} \\ & =_\alpha & R_1 & \end{array}$$

C'est-à-dire que $R_2' \gtrsim_d R_1$. Réciproquement, supposons que R_1 fasse une transition: $R_1 \stackrel{\mu}{\longmapsto} R_1'$. Il est clair que R_2 peut faire la même transition, en effet comme: $R_2 \stackrel{\tau}{\longmapsto} R_2' \gtrsim_d R_1$, on a par définition de l'expansion que: $R_2 \stackrel{\tau}{\longmapsto} \stackrel{\mu}{\Longrightarrow} R_1'$. Par conséquent $R_1 \lesssim_d R_2$, ce qui implique d'après la proposition 10.1 que $R_1 \approx_d R_2$, et donc que $R_1 \approx_b R_2$. □

Avant de montrer le résultat principal de cette section, nous définissons la notion de processus e-tabou. Intuitivement, les processus e-tabous sont les processus ne pouvant pas divulguer le nom e à leur environnement. Ce sont, par exemple, des processus qui ne peuvent pas émettre le nom e.

Définition 18.3 (Ensemble et processus e-tabous) L'ensemble $\mathcal{E}$ est e-tabou, s'il vérifie les conditions suivantes:

- si $P \in \mathcal{E}$ et $P \stackrel{\tau}{\longmapsto} P'$, alors $P' \in \mathcal{E}$;
- si $P \in \mathcal{E}$ et $P \stackrel{\lambda\, a}{\longmapsto} P'$, avec $a \neq e$, alors $P' \in \mathcal{E}$;
- si $P \in \mathcal{E}$ et $P \stackrel{\mathbf{in}\ u.(\tilde{c};\tilde{b})}{\longmapsto} P'$, avec $e \notin \tilde{b}$, alors $P' \in \mathcal{E}$;
- si $P \in \mathcal{E}$ et $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D; P')$, alors $P' \in \mathcal{E}$ et $e \notin (\{\tilde{b}\} \cup \mathbf{fn}(D))$.

On dit que P est un processus e-tabou, s'il existe un ensemble e-tabou $\mathcal{E}$ tel que $P \in \mathcal{E}$. On remarque que cette relation est définie de manière co-inductive, comme la bisimulation.

Munis de la notion de processus e-tabou, on montre une propriété similaire aux lois de réplication (cf. lemme 8.2).

Lemme 18.3 (Réplication des objets) *Soit L un corps d'objet qui ne contient pas le nom e. Soit P et Q deux processus e-tabous, tels que e apparaît dans P et Q uniquement sous la forme* $\mathbf{clone}(e)$*, alors:*

$$\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (P \mid Q) \approx_b (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P) \mid (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ Q)$$

Preuve On montre que la relation $\mathcal{D}$, définie ci-dessous, est une def-bisimulation modulo expansion.

$$\mathcal{D} = \left\{ (R_1, R_2) \;\middle|\; \begin{array}{l} R_1 = \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (P \mid Q), \\ R_2 = (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P) \mid (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ Q), \\ L \text{ ne contient pas } e, \text{ et } P \text{ et } Q \text{ sont } e\text{-tabous, et} \\ e \text{ apparaît uniquement sous la forme } \mathbf{clone}(e) \end{array} \right\}$$

On utilisera aussi la technique de preuve modulo équivalence structurelle et modulo contextes (cf. chapitre 6). Il est clair que $\mathcal{D}$ est close par substitution. De plus, dans les processus R_1 et R_2, les rôles de P et Q sont symétriques, et les transitions ne peuvent provenir que de P et Q, ce qui simplifie notre preuve. On commence par étudier les transitions possibles de R_1, en supposant que les transitions proviennent de P. Le cas intéressant est lorsque P émet un message. On omet ici de donner la preuve que la propriété «P et Q sont e-tabous et e apparaît uniquement sous la forme $\mathbf{clone}(e)$», est conservée après transitions.

- **cas** $P \stackrel{\lambda\, a}{\longmapsto} P'$**:** on a $R_1 \stackrel{\lambda\, a}{\longmapsto} R_1'$, et $R_2 \stackrel{\lambda\, a}{\longmapsto} R_2'$, avec

$$\begin{array}{rcl} R_1' & = & \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (P' \mid (Qa)) \\ R_2' & = & (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P') \mid (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ Q)a \\ & \equiv & (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P') \mid (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (Qa)) \end{array}$$

  On a donc $(R_1', R_2') \in \mathcal{D}$, modulo équivalence structurelle. La preuve est similaire si $P \stackrel{\tau}{\longmapsto} P'$;

- **cas** $P \stackrel{\mathbf{in}\ u.(\tilde{c};\tilde{b})}{\longmapsto} P'$**:** dans le cas où la transition de R_1 est une réception, la preuve est similaire au cas précédent. On peut aussi avoir une communication avec Q, c'est-à-dire que R_1 fait une τ transition. Dans ce cas: $\tilde{c} = \epsilon$, et $Q \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D; Q')$, et $R_1 \stackrel{\tau}{\longmapsto} R_1'$, avec

$$R_1' \quad = \quad \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ ((\boldsymbol{\nu}\, \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (P' \mid Q')))$$

  Les noms dans $\tilde{v}$ peuvent être considérés comme nouveaux. De plus, comme P est e-tabou, le nom e ne peut pas apparaître dans D. Par conséquent on a l'égalité suivante.

$$R_1' \quad \equiv \quad (\boldsymbol{\nu}\, \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (P' \mid Q')))$$

  Comme la communication ne peut pas se faire sur le canal e: on a $u \neq e$, et donc $(\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P) \stackrel{\mathbf{in}\ u.(\epsilon;\tilde{b})}{\longmapsto} (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P')$. De même pour Q. Aussi, il existe une transition $R_2 \stackrel{\tau}{\longmapsto} R_2'$ telle que

$$R_2' = (\boldsymbol{\nu}\, \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P' \mid \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ Q'))$$

  On a donc $(R_1', R_2') \in \mathcal{D}$, modulo équivalence structurelle et contextes, où le contexte utilisé est $(\boldsymbol{\nu}\, \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\ [\_])$;

- **cas** $P \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D; P')$**:** on peut avoir une communication entre P et Q: ce cas est similaire au cas précédent. Sinon, il y a deux cas.
  - **cas** $u \neq e$**:** comme P est e-tabou, les noms dans D et $\tilde{b}$ sont différents de e: il n'y a pas «scope extrusion» du nom e. Il est facile de montrer que dans ce cas, on a $R_1 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D \mid (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (P' \mid Q)))$ et $R_2 \stackrel{\mathbf{out}\ u.(\tilde{v};\tilde{b})}{\longmapsto} (D; (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P' \mid \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ Q))$;
  - **cas** $u = e$**:** comme le nom e est restreint dans R_1, la seule transition possible pour R_1 correspond à la communication avec la déclaration $\langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)} \rangle$, contenue dans $(\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (P \mid Q)$ (cf. définition 17.3). Dans ce cas on a:

$$R_1 \quad \stackrel{\tau}{\longmapsto} \quad R_1' = (\boldsymbol{\nu}\, e)\left(\mathbf{Copy_n}\left[\begin{array}{l} \mathbf{def}\, u_1 = (\boldsymbol{\lambda} x_1)P_1, \ldots, u_n = (\boldsymbol{\lambda} x_n)P_n \\ \mathbf{in}\ \ \mathbf{def}\, s = (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{E}_{(s,\tilde{x},c)}\rangle \\ \qquad \mathbf{in}\, (\boldsymbol{\nu}\, \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (\mathsf{E}_{(s,\tilde{u},c)}\ \tilde{b} \mid P' \mid Q)) \end{array}\right]\right)$$

    Comme par hypothèse e apparaît uniquement sous la forme $\mathbf{clone}(e)$, le tuple $\tilde{b}$ doit commencer par une sélection sur le champ *clone*, c'est-à-dire que $\tilde{b} = (\cdot clone, \tilde{b'})$. De plus P est e-tabou, et donc $e \notin \tilde{b'}$. Par conséquent il existe un contexte $\mathbf{F}$ tel que

$$\begin{array}{rcl} R_1 \stackrel{\tau}{\longmapsto} R_1' & \equiv & \mathbf{F}[\mathsf{E}_{(s,\tilde{u},c)}\ \cdot clone\ \tilde{b'} \mid P' \mid Q] \\ & \gtrsim_d & \mathbf{F}[(s\tilde{u} \mid c\tilde{u})\ \tilde{b'} \mid P' \mid Q] \\ & \equiv & \mathbf{F}[(s\tilde{u}) \mid (c\tilde{u}\tilde{b'}) \mid P' \mid Q] \\ & \gtrsim_d & \mathbf{F}[\langle e \Leftarrow \mathsf{E}_{(s,\tilde{u},c)}\rangle \mid (\boldsymbol{\nu} f)(\mathsf{Fobj}_{(c,f)}\ \tilde{u} \mid f)\ \tilde{b'} \mid P' \mid Q] \\ & \gtrsim_d & (\boldsymbol{\nu}\, \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (\mathbf{obj}\, f\, \mathbf{is}\, L\, \mathbf{in}\ (f\tilde{b'}) \mid P' \mid Q))) \\ & \equiv & (\boldsymbol{\nu}\, \tilde{v})(\mathbf{def}\, D\, \mathbf{in}\, ((\mathbf{obj}\, f\, \mathbf{is}\, L\, \mathbf{in}\ (f\tilde{b'})) \mid \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (P' \mid Q))) \end{array}$$

En utilisant le même schéma de transition, on montre aussi que

$$\begin{array}{rcl} R_2 \stackrel{\tau}{\longmapsto} R_2' & = & (\mathbf{F}[\mathsf{E}_{(s,\tilde{u},c)}\ \cdot clone\ \tilde{b} \mid P']) \mid (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ Q) \\ & \gtrsim_d & (\boldsymbol{\nu}\tilde{v})(\mathbf{def}\, D\, \mathbf{in} \left(\begin{array}{l} (\mathbf{obj}\, f\, \mathbf{is}\, L\, \mathbf{in}\ (f\tilde{b})) \mid \\ (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ P') \mid (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ Q) \end{array} \right)) \end{array}$$

On a donc $(R_1', R_2') \in \mathcal{D}$, modulo contextes et expansion.

La preuve est similaire si on s'intéresse aux transitions que peux faire R_2. □

On peut utiliser le schéma de la preuve précédente pour chaque constructeur du calcul bleu, ce qui permet de prouver qu'on peut distribuer la définition d'un objet sous tout contexte[2]. Néanmoins nous conjecturons qu'un résultat plus général est vrai, c'est-à-dire que les lois de réplication sont vraies même lorsque le nom de l'objet peut être extrudé.

Conjecture 18.4 (Lois de réplication pour les objets) *Soit L un corps d'objet qui ne contient pas e, et soit $\mathbf{C}$ un contexte qui ne capture ni e, ni les noms libre de L, alors:*

$$\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (\mathbf{C}[\mathbf{clone}(e)]) \ \approx_b\ \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ (\mathbf{C}[\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ e])$$

18.2 Correction de l'interprétation de self

Soit v la valeur de $\mathbf{Ob}_{1<:}$ définie par $[\, l_i = \varsigma(x_i) f_i{}^{i\in[1..n]}\,]$, et soit $(\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ e)$ l'interprétation de v dans le calcul bleu. Nous montrons que $(\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ [\![f]\!]\{e/x\})$ est équivalent à $[\![f\{v/x\}]\!]$. Ce résultat repose sur la conjecture 18.4.

Lemme 18.5 *Soit v une valeur de $\mathbf{Ob}_{1<:}$ telle que $[\![v]\!] = (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ e)$, alors:*

$$(\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ [\![f]\!]\{e/x\}) \approx_b [\![f\{v/x\}]\!]$$

Preuve La preuve est faite par induction sur la définition de l'objet f.

- **cas** $f =_{\text{def}} y$**:** on a $[\![f]\!] = \mathbf{clone}(y)$, et le résultat découle du lemme 18.2;
- **cas** $f =_{\text{def}} (e_1 \cdot l \Leftarrow \varsigma(y) e_2)$**:** on a $[\![f]\!] = ([\![e_1]\!] \leftarrow l = (\boldsymbol{\lambda} y)[\![e_2]\!])$. On peut supposer que $y \neq x$, et donc:

$$\begin{array}{rcll} (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ [\![f]\!]\{e/x\}) & \equiv & \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ ([\![e_1]\!]\{e/x\}\ \cdot put_{l_j}\ (\boldsymbol{\lambda} y)([\![e_2]\!]\{e/x\})) & \\ & \approx_b & \left(\begin{array}{l} \mathbf{def}\, p = (\boldsymbol{\lambda} y)(\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ ([\![e_2]\!]\{e/x\})) \\ \mathbf{in}\, (\mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\ ([\![e_1]\!]\{e/x\})\ \cdot put_{l_j}\ p) \end{array} \right) & (18.3) \\ & \approx_b & [\![e_1\{v/x\}]\!]\ \cdot put_{l_j}\ (\boldsymbol{\lambda} y)[\![e_2\{v/x\}]\!] & (18.4) \\ & = & [\![(e_1 \cdot l \Leftarrow \varsigma(y) e_2)\{v/x\}]\!] & \end{array}$$

  Dans la relation (18.3), on utilise la conjecture 18.4, et dans la relation (18.4), on utilise l'hypothèse d'induction. La preuve est similaire dans les cas $f =_{\text{def}} (e_1 \cdot l)$ et $f =_{\text{def}} [\, l_i = \varsigma(x_i) e_i{}^{i\in[1..n]}\,]$.

□

Nous nous aidons de ce résultat pour prouver que le codage de $\mathbf{Ob}_{1<:}$ dans le calcul bleu est complet. Ce résultat repose sur le lemme 18.5, et donc sur la conjecture 18.4.

Théorème 18.6 (Le calcul bleu simule $\mathbf{Ob}_{1<:}$) *Le codage de $\mathbf{Ob}_{1<:}$ est* complet*: si $e \rightsquigarrow e'$, alors il existe un processus Q tel que $[\![e]\!] \stackrel{*}{\rightarrow} Q \approx_b [\![e']\!]$.*

2. Moyennant l'hypothèse que les contextes soient e-tabous, et que la référence sur l'objet n'est utilisée que pour le cloner.

Preuve La preuve est faite par induction sur l'inférence de $e \rightsquigarrow e'$, elle se réduit à une étude par cas sur la dernière règle de cette inférence.

- **cas (ac red context):** il est évident que l'interprétation d'un contexte d'évaluation de $\mathbf{Ob}_{1<:}$ est un contexte d'évaluation de $\boldsymbol{\pi}^\star$. Le résultat suit donc par utilisation de la règle (red context).
- **cas (ac red invk):** soit v l'objet $[\, l_i = \varsigma(x_i) f_i^{\,i\in[1..n]}\,]$, et j un indice de l'intervalle $[1..n]$, alors:

$$\begin{array}{rcl} [\![v{\cdot}l_j]\!] & \equiv & (\boldsymbol{\nu} e)(\langle e \mapsto [\,(\boldsymbol{\lambda} x_i)[\![f_i]\!]^{i\in[1..n]}\,]\rangle \mid e \cdot get_{l_j}) \\ & \stackrel{*}{\rightarrow} & (\boldsymbol{\nu} e)(\langle e \mapsto [\,(\boldsymbol{\lambda} x_i)[\![f_i]\!]^{i\in[1..n]}\,]\rangle \mid [\![f_j]\!]\{^e/_{x_j}\}) \quad \text{(règle (red invk))} \\ & = & \mathbf{obj}\, e\, \mathbf{is}\, [\,(\boldsymbol{\lambda} x_i)[\![f_i]\!]^{i\in[1..n]}\,]\, \mathbf{in}\, ([\![f_j]\!]\{^e/_{x_j}\}) \end{array}$$

  Par conséquent, en utilisant le lemme 18.5, on obtient que:

$$[\![v{\cdot}l_j]\!] \stackrel{*}{\rightarrow} \mathbf{obj}\, e\, \mathbf{is}\, [\,(\boldsymbol{\lambda} x_i)[\![f_i]\!]^{i\in[1..n]}\,]\, \mathbf{in}\, ([\![f_j]\!]\{^e/_{x_j}\}) \approx_b [\![f_j\{^v/_{x_j}\}]\!]$$

- **cas (ac red updt):** soit v l'objet $[\, l_i = \varsigma(x_i) f_i^{\,i\in[1..n]}\,]$, et j un indice de l'intervalle $[1..n]$. Nous notons L le corps d'objet $(l_i = (\boldsymbol{\lambda} x_i)[\![f_i]\!]^{i\in[1..n]})$, et L' le corps $[\, L\,, l_j = (\boldsymbol{\lambda} x)[\![f]\!]\,]$. Nous montrons qu'il existe un processus Q, tel que $[\![v{\cdot}l_j \Leftarrow \varsigma(x) f]\!] \stackrel{*}{\rightarrow} Q$ et $Q \approx_b \mathbf{obj}\, e\, \mathbf{is}\, L'\, \mathbf{in}\, e$.

$$\begin{array}{rcll} [\![v{\cdot}l_j \Leftarrow \varsigma(x) f]\!] & = & (\boldsymbol{\nu} e)(\langle e \mapsto L\rangle \mid \mathbf{def}\, p = (\boldsymbol{\lambda} x)[\![f]\!]\, \mathbf{in}\, (e \cdot put_{l_j}\ p)) & \\ & \stackrel{*}{\rightarrow} \sim_d & (\boldsymbol{\nu} e)(\mathbf{def}\, p = (\boldsymbol{\lambda} x)[\![f]\!]\, \mathbf{in}\, \langle e \mapsto [\, L\,, l_j = p\,]\rangle \mid e) & (18.5) \\ & \sim_d & (\boldsymbol{\nu} e)(\langle e \mapsto [\, L\,, l_j = \mathbf{def}\, p = (\boldsymbol{\lambda} x)[\![f]\!]\, \mathbf{in}\, p\,]\rangle \mid e) & (18.6) \\ & \sim_d & (\boldsymbol{\nu} e)(\langle e \mapsto [\, L\,, l_j = (\boldsymbol{\lambda} x)[\![f]\!]\,]\rangle \mid e) & (18.7) \\ & = & \mathbf{obj}\, e\, \mathbf{is}\, L'\, \mathbf{in}\, e & \end{array}$$

  Dans la relation (18.5), on utilise la règle dérivée (red updt) et le théorème 17.1. Dans la relation (18.6), on utilise le lemme 8.2 pour distribuer la définition sur le nom p. Dans la relation (18.7), on utilise la proposition 4.4 et le fait que $\approx_b$ soit une congruence.

□

Une interprétation n'est pas seulement un codage d'un calcul vers un autre. On cherche aussi à préserver des notions tel que la convergence et le typage. Le théorème 18.6 permet de répondre à la première attente. Ainsi il est clair que si v est une valeur de $\mathbf{Ob}_{1<:}$, alors $[\![v]\!]$ est une valeur de $\boldsymbol{\pi}^\star$, dans le sens où $[\![v]\!] \Downarrow$. De plus le théorème 18.6 implique que si l'objet e converge vers une valeur, alors il existe un processus Q tel que: $[\![e]\!] \stackrel{*}{\rightarrow} Q \approx_b [\![v]\!] \Downarrow$. Par conséquent, si e converge, alors $[\![e]\!] \Downarrow$.

Nous prouvons maintenant que notre interprétation préserve les notions de typage et de sous-typage.

Théorème 18.7 (Simulation du typage) *Si $A <: B$, alors $[\![A]\!] \leqslant [\![B]\!]$, et si $E \vdash e : A$ $[\mathbf{Ob}_{1<:}]$, alors $[\![E]\!] \vdash [\![e]\!] : [\![A]\!]$ $[\mathtt{B}_{\leqslant}]$.*

Preuve La première propriété, concernant le sous-typage, est impliquée par le lemme 17.2 et la règle (sub top). La deuxième propriété est prouvée par induction sur l'inférence de $E \vdash e : A$ $[\mathbf{Ob}_{1<:}]$. On étudie la dernière règle de cette inférence. Soit A le type $[\, l_i : B_i^{\,i\in[1..n]}\,]$.

- **cas (ac type ax):** le résultat est obtenu en appliquant la règle (type ax), puis la règle dérivée (type clone), donnée dans la figure 17.2. La preuve est similaire dans le cas (ac type sub), pour lequel on utilise la règle (type sub);

– **cas (ac type obj):** les prémisses de cette règle sont que, pour tout indice i dans l'intervalle $[1..n]$, on a: E,$x_i : A \vdash f_i : B_i$, et la conclusion est: $E \vdash [\, l_i = \varsigma(x_i) f_i{}^{i\in[1..n]} \,] : A$. L'hypothèse d'induction permet de prouver que $[\![E]\!]$,$x_i : [\![A]\!] \vdash [\![f_i]\!] : [\![B_i]\!]$. Par conséquent, en utilisant les résultat de la section 17.2, et en particulier la règle dérivée (type obj), on obtient que $[\![E]\!] \vdash \mathbf{obj}\, e\, \mathbf{is}\, (l_i = [\![f_i]\!]^{i\in[1..n]})\, \mathbf{in}\, e : [\![A]\!]$, c'est-à-dire que $[\![E]\!] \vdash [\![\, [\, l_i = \varsigma(x_i) f_i{}^{i\in[1..n]} \,]\!]\!] : [\![A]\!]$. La preuve est similaire dans les cas (ac type sel) et (ac type updt), pour lesquels on utilise, respectivement, les règles (type invk) et (type updt).

□

Dans l'interprétation de $\mathbf{Ob}_{1<:}$ donnée dans la figure 18.1, nous utilisons l'opérateur dérivé de clonage pour coder l'appel au paramètre self. Intuitivement, au moment de l'invocation, une variable liée par $\boldsymbol{\varsigma}$ doit être remplacée par une copie du code de l'objet appelé: c'est ce que traduit le lemme 18.5. Il est possible de simplifier notre interprétation en ne clonant les variables que lorsque ceci est nécessaire. C'est le cas, par exemple, avec l'objet e défini ci-dessous, qui est un exemple de terme non normalisable de $\mathbf{Ob}_{1<:}$.

$$e =_{\text{def}} [\, l = \varsigma(x)(x{\cdot}l) \,]{\cdot}l \;\rightsquigarrow\; [\, l = \varsigma(x)(x{\cdot}l) \,]{\cdot}l \;\rightsquigarrow\; \cdots$$

Il est clair que le processus $(\mathbf{obj}\, e\, \mathbf{is}\, [\, l = (\boldsymbol{\lambda} x)(x \Leftarrow l) \,]\, \mathbf{in}\, (e \Leftarrow l))$, a le «même comportement» que $[\![e]\!]$, qui est le processus: $\mathbf{obj}\, e\, \mathbf{is}\, [\, l = (\boldsymbol{\lambda} x)(\mathbf{clone}(x) \Leftarrow l) \,]\, \mathbf{in}\, (e \Leftarrow l)$. On peut donc faire l'économie d'un clonage dans cet exemple. Néanmoins on peut aussi exhiber des cas dans lesquels l'utilisation du clonage est indispensable. Ainsi, si on considère l'exemple suivant [4], dans lequel un objet s'applique une modification à lui-même (on parle de «self-inflicted update» en anglais):

$$f =_{\text{def}} [\, l = \varsigma(x)(x{\cdot}l \Leftarrow \varsigma(y)x) \,]{\cdot}l \quad\rightsquigarrow\quad [\, l = \varsigma(y)[\, l = \varsigma(x)(x{\cdot}l \Leftarrow \varsigma(y)x) \,]\,]$$

On voit facilement que le processus obtenu à partir de $[\![f]\!]$ en éliminant le clonage, n'est pas équivalent à $[\![f]\!]$. En effet, si L est le corps $[\, l = (\boldsymbol{\lambda} x)(x \leftarrow l = (\boldsymbol{\lambda} y)x) \,]$, alors on a:

$$\begin{array}{rcl} \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\, (e \Leftarrow l) & \xrightarrow{*} \approx_b & \mathbf{obj}\, e\, \mathbf{is}\, L\, \mathbf{in}\, (e \leftarrow l = (\boldsymbol{\lambda} y) e) \\ & \xrightarrow{*} \approx_b & \mathbf{obj}\, e\, \mathbf{is}\, [\, l = (\boldsymbol{\lambda} y) e \,]\, \mathbf{in}\, e \end{array}$$

Ce dernier exemple est intéressant, puisque le comportement obtenu est exactement celui observé dans le $\boldsymbol{\varsigma}$-calcul impératif, dénoté $\mathbf{imp}\boldsymbol{\varsigma}$, décrit dans [3]. On peut donc se demander s'il est possible de modifier notre codage simplement, afin d'interpréter $\mathbf{imp}\boldsymbol{\varsigma}$ dans le calcul bleu. Cette question semble raisonnable. En effet nous avons vu que le calcul bleu permet de coder des opérateurs impératifs: les références (cf. section 16.2), les définitions avec appel par valeur (cf. section 3.5.3), ... De plus D. Sangiorgi et J. Kleist ont déjà proposé une interprétation de $\mathbf{imp}\boldsymbol{\varsigma}$ dans le π-calcul [76]. Aussi il n'est pas surprenant que nous répondions par l'affirmative à cette question dans le chapitre suivant. Nous faisons même mieux: nous donnons l'interprétation d'une extension concurrente de $\boldsymbol{\varsigma}$ qui contient $\mathbf{imp}\boldsymbol{\varsigma}$ comme sous-calcul.

CHAPITRE 19

Interprétation d'une extension concurrente de self

DANS CE CHAPITRE, nous étudions une version impérative et concurrente du calcul d'objets de M. ABADI et L. CARDELLI, dénoté **concς** et définie par A. GORDON et P. HANKIN dans [60]. Nous donnons une interprétation de **concς** dans le calcul bleu, qui est obtenue naturellement à partir de l'interprétation de $\mathbf{Ob}_{1<:}$ donnée dans la définition 17.4.

19.1 Le calcul

Le calcul **concς** est une extension du calcul d'objets impératif **impς**, par des opérateurs empruntés au π-calcul. De plus, comme le π-calcul, il est basé sur la notion de nommage.

Le calcul d'objets **concς** possède un opérateur de composition parallèle et un opérateur de restriction. Il possède aussi les opérateurs de **impς**, dont un opérateur pour cloner les objets: $\mathbf{clone}(p)$, et un opérateur de définition avec appel par valeur: $\mathbf{let}\, x = e\, \mathbf{in}\, f$. Dans cette section, nous donnons un aperçu de **concς**, et nous renvoyons le lecteur à [61] pour une définition plus complète de ce calcul.

La syntaxe de **concς** est donnée dans la figure 19.1. On remarque que chaque objet est nommé pour former une dénomination: $(p \mapsto [\, l_i = \varsigma(x_i) f_j{}^{i\in[1..n]}\,])$ qui, de manière informelle, a le même comportement que le processus dénomination défini dans $\boldsymbol{\pi}^\star$ (cf. définition 17.3). On remarque aussi que si on a ajouté à $\boldsymbol{\varsigma}$ de nouveaux opérateurs, on a également restreint le calcul. Ainsi on peut seulement appliquer la sélection et la modification de méthode à un résultat, et pas à un objet quelconque. Néanmoins on peut retrouver la syntaxe directe de $\boldsymbol{\varsigma}$ en introduisant les notations:

$$e{\cdot}l =_{\text{def}} \mathbf{let}\, x = e\, \mathbf{in}\, x{\cdot}l \qquad \text{et} \qquad e{\cdot}l \Leftarrow \varsigma(y)f =_{\text{def}} \mathbf{let}\, x = e\, \mathbf{in}\, (x{\cdot}l \Leftarrow \varsigma(y)f)$$

Et on peut représenter un objet de $\boldsymbol{\varsigma}$ par le terme $(\boldsymbol{\nu} p)((p \mapsto d) \Rsh p)$ – où $\Rsh$ est l'opérateur de composition parallèle de **concς**– comme dans notre interprétation des objets, cf. définition 17.1.

Comme pour $\boldsymbol{\pi}^\star$, la sémantique opérationnelle est donné dans le «style chimique». La relation d'équivalence structurelle, dénoté $\equiv$, est donnée dans la figure 19.2, et la relation de réduction dans la figure 19.3. Dans la définition de $\equiv$, on omet de donner les règles qui implique que cette relation est une congruence.

La majeure partie des règles de la figure 19.2, sont des équivalents des règles d'équivalence structurelle du calcul bleu (cf. figure 2.2). Ainsi deux règles seulement ne sont pas usuelles: les règles (struct par let) et (struct res let) permettent au terme e, dans le terme $\mathbf{let}\, x = e\, \mathbf{in}\, f$, d'interagir avec d'autres processus en parallèle. Nous avons cependant omis d'inclure une règle

résultats:	u,v	$::=$	x	variable
		$\mid$	p	nom
dénotations :	d	$::=$	$[\, l_i = \varsigma(x_i) f_j^{\,i\in[1..n]} \,]$	
termes:	e,f,g	$::=$	u	résultat
		$\mid$	$(p \mapsto d)$	dénomination
		$\mid$	$u{\cdot}l$	invocation de méthode
		$\mid$	$u{\cdot}l \Leftarrow \varsigma(x)b$	modification de méthode
		$\mid$	$\mathbf{clone}(u)$	clonage
		$\mid$	$\mathbf{let}\, x = e \,\mathbf{in}\, f$	let
		$\mid$	$(e \Rsh f)$	composition parallèle
		$\mid$	$(\boldsymbol{\nu} p)e$	restriction

Fig. 19.1: Syntaxe du calcul **concς**

$$\frac{}{(e \Rsh f) \Rsh g \equiv e \Rsh (f \Rsh g)}\ \text{(struct par assoc)} \qquad \frac{}{(e \Rsh f) \Rsh g \equiv (f \Rsh e) \Rsh g}\ \text{(struct par comm)}$$

$$\frac{}{(\boldsymbol{\nu} p)(\boldsymbol{\nu} q)e \equiv (\boldsymbol{\nu} q)(\boldsymbol{\nu} p)e}\ \text{(struct res res)} \qquad \frac{p \notin \mathbf{fn}(e)}{(\boldsymbol{\nu} p)(e \Rsh f) \equiv (e \Rsh (\boldsymbol{\nu} p)f)}\ \text{(struct par)}$$

$$\frac{p \notin \mathbf{fn}(f)}{(\boldsymbol{\nu} p)(e \Rsh f) \equiv ((\boldsymbol{\nu} p)e \Rsh f)}\ \text{(struct par)} \qquad \frac{p \notin \mathbf{fn}(f)}{(\boldsymbol{\nu} p)(\mathbf{let}\, x = e \,\mathbf{in}\, f) \equiv \mathbf{let}\, x = (\boldsymbol{\nu} p)e \,\mathbf{in}\, f}\ \text{(struct res let)}$$

$$\frac{}{e \Rsh \mathbf{let}\, x = f \,\mathbf{in}\, g \equiv \mathbf{let}\, x = (e \Rsh f) \,\mathbf{in}\, g}\ \text{(struct par let)}$$

Fig. 19.2: Équivalence structurelle dans **concς**

d'équivalence structurelle de **concς** dans la figure 19.2. Il s'agit de la règle (struct let assoc) ci-dessous.

$$\frac{y \notin \mathbf{fn}(g)}{\mathbf{let}\, x = (\mathbf{let}\, y = e\, \mathbf{in}\, f)\, \mathbf{in}\, g \equiv \mathbf{let}\, y = e\, \mathbf{in}\, (\mathbf{let}\, x = f\, \mathbf{in}\, g)} \text{ (struct let assoc)}$$

Cette règle fait partie de la définition de **concς**, mais elle n'est utilisée que pour simplifier la définition d'une forme normale pour les termes. De plus son absence n'affecte pas la réduction, dans le sens où si $\rightsquigarrow_{\mathrm{LA}}$ est la réduction avec la règle (struct let assoc), alors:

- si $e \rightsquigarrow_{\mathrm{LA}} f$, alors il existe un terme f' tel que $e \rightsquigarrow f' \equiv f$;
- si $e \rightsquigarrow f$, alors $e \rightsquigarrow_{\mathrm{LA}} f$.

Cette dernière propriété est due à P. Hankin [63], qui ne donne cependant pas de preuve formelle. Néanmoins, il est clair que l'ensemble des couples $(\mathbf{let}\, x = (\mathbf{let}\, y = e\, \mathbf{in}\, f)\, \mathbf{in}\, g, \mathbf{let}\, y = e\, \mathbf{in}\, (\mathbf{let}\, x = f\, \mathbf{in}\, g))$, tel que $y \notin \mathbf{fn}(g)$, forme une relation de bisimulation forte modulo $(\boldsymbol{\nu})$ et $\Rsh$. Aussi, dans la suite, nous choisissons de ne pas considérer cette règle dans notre interprétation.

Comme dans le calcul bleu, la composition parallèle est un opérateur dissymétrique: la règle (struct par comm) ne permet de commuter des processus qu'à gauche d'une composition parallèle. Néanmoins, et contrairement au cas du calcul bleu, ce choix est plus dicté par la sémantique opérationnelle du calcul, que par des considérations de typage. En effet, dans la règle de réduction du let (gh red let result), il faut pouvoir séparer dans un terme la partie résultat: la partie à droite du parallèle le plus extérieur, de l'environnement. L'intuition est que l'opérateur $\Rsh$ se comporte comme l'instruction *fork* des langages possédant des threads: l'exécution de $(e \Rsh f)$ revient à créer un nouveau thread e, dont seul les effets de bords, et pas le résultat, est important; le résultat de l'évaluation de $(e \Rsh f)$ étant le résultat de f.

Les règles définissant la réduction dans **concς** sont données dans la figure 19.3. On a simplement omit de définir les règles de réduction qui permettent de réduire un terme sous un parallèle ou une restriction: l'équivalent de la règle (red context) de la figure 2.3. On peut remarquer que les règles de la figure 19.3 sont très proches des règles de réduction dérivées des objets: (red invk), (red updt) et (red clone).

19.2 Interprétation

Contrairement à notre interprétation du calcul fonctionnel d'objet $\mathbf{Ob}_{1<:}$, notre codage des termes de **concς** utilise des continuations. En particulier le codage utilise les opérateurs dérivés $\mathbf{return}(p)$ et $(\mathbf{set}\, x = P\ \mathbf{in}\, Q)$, définis dans la section 3.5. Ces deux opérateurs dérivés nous permettent de simuler la sémantique avec appel par valeur de **concς**.

Il nous faut aussi modifier la définition de la dénomination pour les mêmes raisons. Soit $\mathsf{F}_{(s,\tilde{x},c)}$ l'enregistrement suivant.

Définition 19.1 ($\mathsf{F}_{(s,\tilde{x},c)}$)

$$\mathsf{F}_{(s,\tilde{x},c)} =_{\mathrm{def}} \left[\begin{array}{ll} \ldots & {\scriptstyle i\in[1..n]} \\ get_{l_i} = (s\tilde{x} \mid x_i e), & \\ put_{l_i} = (\boldsymbol{\lambda} y_i)(s x_1 \ldots y_i \ldots x_n \mid \mathbf{return}(e)), & \\ \ldots & \\ clone = (s\tilde{x} \mid c\tilde{x}) & \end{array}\right]$$

---

soit d la dénotation $d =_{\text{def}} [\, l_i = \varsigma(x_i) f_i^{\,i\in[1..n]} \,]$

$$\frac{e \rightsquigarrow e' \qquad e \equiv f}{f \rightsquigarrow e'} \text{ (gh red struct)}$$

$$\frac{j \in [1..n]}{(p \mapsto d) \Rsh p{\cdot}l_j \rightsquigarrow (p \mapsto d) \Rsh f_j\{p/x_j\}} \text{ (gh red select)}$$

$$\frac{d' = [\, l_j = \varsigma(x) f, l_i = \varsigma(x_i) f_i^{\,i\in[1..n], i\neq j} \,] \qquad j \in [1..n]}{(p \mapsto d) \Rsh (p{\cdot}l_j \Leftarrow \varsigma(x) f) \rightsquigarrow (p \mapsto d') \Rsh p} \text{ (gh red updt)}$$

$$\frac{q \notin \mathbf{fn}(d)}{(p \mapsto d) \Rsh \mathbf{clone}(p) \rightsquigarrow (p \mapsto d) \Rsh (\boldsymbol{\nu} q)((q \mapsto d) \Rsh q)} \text{ (gh red clone)}$$

$$\frac{}{\mathbf{let}\, x = p \,\mathbf{in}\, f \rightsquigarrow f\{p/x\}} \text{ (gh red let result)}$$

$$\frac{e \rightsquigarrow e'}{\mathbf{let}\, x = e \,\mathbf{in}\, f \rightsquigarrow \mathbf{let}\, x = e' \,\mathbf{in}\, f} \text{ (gh red let)}$$

Fig. 19.3: Réduction dans **concς**

Le codage de la dénomination «avec continuations», dénoté $\langle e \mapsto\!\!\!\!\circ\; L \rangle$, est similaire à celui donné dans le chapitre 18, à la différence que nous utilisons l'enregistrement $\mathsf{F}_{(s,\tilde{x},c)}$ à la place de $\mathsf{E}_{(s,\tilde{x},c)}$. Sommairement, on obtient $\langle e \mapsto\!\!\!\!\circ\; L \rangle$ à partir de $\langle e \mapsto L \rangle$ en ajoutant l'opérateur **return(.)** à chaque fois qu'une valeur est attendue.

Définition 19.2 (Dénomination avec continuations) Soit L le corps $[\, l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in[1..n]} \,]$. La dénomination $\langle e \mapsto\!\!\!\!\circ\; L \rangle$ est le processus défini par:

$$\begin{array}{l}\mathbf{def}\, c = (\boldsymbol{\lambda}\tilde{x})(\boldsymbol{\nu} f)((\mathbf{rec}\, s.(\boldsymbol{\lambda}\tilde{x})\langle f \Leftarrow \mathsf{F}_{(s,\tilde{x},c)}\rangle)\ \tilde{x} \mid \mathbf{return}(f)) \\ \mathbf{in}\ \ \mathbf{def}\, u_1 = (\boldsymbol{\lambda} x_1) P_1, \ldots, u_n = (\boldsymbol{\lambda} x_n) P_n \\ \qquad \mathbf{in}\ \ \mathbf{def}\, s = (\boldsymbol{\lambda}\tilde{x})\langle e \Leftarrow \mathsf{F}_{(s,\tilde{x},c)}\rangle \\ \qquad\qquad \mathbf{in}\, \langle e \Leftarrow \mathsf{F}_{(s,\tilde{u},c)}\rangle\end{array}$$

Le codage de **concς** est donné dans la définition 19.3. On remarque que le codage des opérateurs déjà présents dans $\mathbf{Ob}_{1<:}$ n'a pas varié, mis à part pour l'interprétation des variables, qui n'utilise plus le clonage.

Définition 19.3 (Codage de concς)

$$\begin{array}{rcl}[\![u]\!] & = & \mathbf{return}(u) \\ [\![(p \mapsto [\, l_i = \varsigma(x_i) f_i^{\,i\in[1..n]} \,])]\!] & = & \langle p \mapsto\!\!\!\!\circ\; [\, l_i = (\boldsymbol{\lambda} x_i)[\![f_i]\!]^{\,i\in[1..n]} \,]\rangle \\ [\![u{\cdot}l]\!] & = & u \Leftarrow\!\!\!\!\circ\; l \\ [\![u{\cdot}l \Leftarrow \varsigma(x) f]\!] & = & (u \leftarrow\!\!\!\!\circ\; l = (\boldsymbol{\lambda} x)[\![f]\!]) \\ [\![\mathbf{clone}(u)]\!] & = & \mathbf{clone}(u) \\ [\![\mathbf{let}\, x = e \,\mathbf{in}\, f]\!] & = & \mathbf{set}\, x = [\![e]\!] \,\mathbf{in}\, [\![f]\!] \\ [\![e \Rsh f]\!] & = & ([\![e]\!] \mid [\![f]\!]) \\ [\![(\boldsymbol{\nu} p) e]\!] & = & (\boldsymbol{\nu} p)[\![e]\!]\end{array}$$

Comme annoncé à la fin du chapitre précédent, nous montrons que le calcul bleu peut simuler **concς**.

Théorème 19.1 *Si $e \equiv f$, alors $[\![e]\!] \equiv [\![f]\!]$. Si $e \rightsquigarrow e'$, alors il existe un processus Q tel que $[\![e]\!] \xrightarrow{*} Q$ et $Q \approx_b [\![e']\!]$.*

Preuve La preuve de la première propriété est faite par induction sur l'inférence de $e \equiv f$. La preuve est très simple, en effet, comme nous l'avons déjà remarqué, seules les règles (struct res let) et (struct par let) ne sont pas des équivalents de règles de l'équivalence structurelle du calcul bleu. Pour ces deux règles, il suffit d'utiliser le codage de la définition 3.8: $(\mathbf{set}\, x = Q\ \mathbf{in}\, P) = (\boldsymbol{\nu} u)(\langle u \Leftarrow (\boldsymbol{\lambda} x)P\rangle \mid Qu)$, et les règles de distributivité et d'extension de la portée données dans la figure 2.2, de la façon suivante.

(struct res let) soit p un nom qui n'est pas libre dans f et u un nom nouveau, alors:

$$\begin{array}{rcl} [\![(\boldsymbol{\nu} p)(\mathbf{let}\, x = e\, \mathbf{in}\, f)]\!] & = & (\boldsymbol{\nu} p)((\boldsymbol{\nu} u)(\langle u \Leftarrow (\boldsymbol{\lambda} x)[\![f]\!]\rangle \mid [\![e]\!]\, u)) \\ & \equiv & (\boldsymbol{\nu} u)(\langle u \Leftarrow (\boldsymbol{\lambda} x)[\![f]\!]\rangle \mid (\boldsymbol{\nu} p)[\![e]\!]\, u) \\ & = & [\![\mathbf{let}\, x = (\boldsymbol{\nu} p)e\, \mathbf{in}\, f]\!] \end{array}$$

(struct par let) soit u un nouveau nom, alors:

$$\begin{array}{rcl} [\![e \Rsh \mathbf{let}\, x = f\, \mathbf{in}\, g]\!] & = & [\![e]\!] \mid (\boldsymbol{\nu} u)(\langle u \Leftarrow (\boldsymbol{\lambda} x)[\![g]\!]\rangle \mid [\![f]\!]\, u) \\ & \equiv & (\boldsymbol{\nu} u)(\langle u \Leftarrow (\boldsymbol{\lambda} x)[\![g]\!]\rangle \mid ([\![e]\!] \mid [\![f]\!])\, u) \\ & = & [\![\mathbf{let}\, x = (e \Rsh f)\, \mathbf{in}\, g]\!] \end{array}$$

La preuve de la seconde propriété est faite par induction sur l'inférence de $e \rightsquigarrow e'$. La preuve est semblable à la preuve du théorème 18.6. Ainsi on commence par donner un ensemble de relations similaires aux règles de réduction des opérateurs objets de la figure 17.1, qui permettent de simuler les règles (gh red select), (gh red updt) et (gh red clone). Nous omettons la preuve du fait que ces relations sont correctes dans notre système, car elle est très semblable à la preuve du théorème 17.1. Soit L le corps $[\, l_i = (\boldsymbol{\lambda} x_i)P_i^{\,i\in[1..n]}\,]$, on montre que:

- Si j est un indice dans l'intervalle $[1..n]$, alors:

$$(\langle e \mapsto L\rangle \mid e \Leftarrow l_j) \xrightarrow{*} \approx_b (\langle e \mapsto L\rangle \mid P_j\{e/x_j\})$$

- Si j est un indice dans l'intervalle $[1..n]$, et si L' est le corps $[\, L\, , l_j = (\boldsymbol{\lambda} x)P\,]$, alors:

$$(\langle e \mapsto L\rangle \mid (e \leftarrow l_j = (\boldsymbol{\lambda} x)P)) \xrightarrow{*} \approx_b (\langle e \mapsto L'\rangle \mid \mathbf{return}(e))$$

- si $f \notin \mathbf{fn}(L)$, alors:

$$(\langle e \mapsto L\rangle \mid \mathbf{clone}(e)) \xrightarrow{*} \approx_b (\langle e \mapsto L\rangle \mid (\boldsymbol{\nu} f)(\langle f \mapsto L\rangle \mid \mathbf{return}(f)))$$

Il ne reste donc plus qu'à montrer comment on simule les règles (gh red let) et (gh red let result). On rappelle que $\mathbf{return}(p) = (\boldsymbol{\lambda} r)(r\, p)$.

(gh red let) on a l'hypothèse que $e \rightsquigarrow e'$, et donc que $[\![e]\!] \xrightarrow{*} \approx_b [\![e']\!]$. Soit u un nouveau nom, on utilise le fait que $(\boldsymbol{\nu} u)(P \mid [\_]\, u)$ est un contexte d'évaluation et que $\approx_b$ est une congruence, pour montrer que:

$$\begin{array}{rcl} [\![\mathbf{let}\, x = e\, \mathbf{in}\, f]\!] & = & (\boldsymbol{\nu} u)(\langle u \Leftarrow (\boldsymbol{\lambda} x)[\![f]\!]\rangle \mid [\![e]\!]\, u) \\ & \xrightarrow{*} \approx_b & (\boldsymbol{\nu} u)(\langle u \Leftarrow (\boldsymbol{\lambda} x)[\![f]\!]\rangle \mid [\![e']\!]\, u) \\ & = & [\![\mathbf{let}\, x = e'\, \mathbf{in}\, f]\!] \end{array}$$

(gh red let result) soit u un nom frais, alors:

$$\begin{array}{rcl} [\![\mathbf{let}\, x = p\, \mathbf{in}\, f]\!] & = & (\boldsymbol{\nu} u)(\langle u \Leftarrow (\boldsymbol{\lambda} x)[\![f]\!]\rangle \mid \mathbf{return}(p)\, u) \\ & \rightarrow & (\boldsymbol{\nu} u)(\langle u \Leftarrow (\boldsymbol{\lambda} x)[\![f]\!]\rangle \mid u\, p) \\ & \xrightarrow{*} & (\boldsymbol{\nu} u)[\![f]\!]\{p/x\} \\ & \equiv & [\![f\{p/x\}]\!] \end{array}$$

□

Nous montrons également la validité de la règle (struct let assoc), dans le cas particulier où l'objet e est une valeur, c'est-à-dire que nous montrons que si $y \notin \mathbf{fn}(g)$, alors:

$$[\![\mathbf{let}\, x = (\mathbf{let}\, y = u\, \mathbf{in}\, f)\, \mathbf{in}\, g]\!] \approx_b [\![\mathbf{let}\, y = u\, \mathbf{in}\, (\mathbf{let}\, x = f\, \mathbf{in}\, g)]\!]$$

Nous prouvons cette équivalence en utilisant la relation d'expansion (cf. chapitre 10), et plus particulièrement le lemme 10.4. Soit v et w deux nouveaux noms, alors:

$$\begin{array}{lcl}
[\![\mathbf{let}\, x = (\mathbf{let}\, y = u\, \mathbf{in}\, f)\, \mathbf{in}\, g]\!] & = & (\boldsymbol{\nu} v)(\langle v \Leftarrow (\boldsymbol{\lambda} x)[\![g]\!]\rangle \mid (\boldsymbol{\nu} w)(\langle w \Leftarrow (\boldsymbol{\lambda} y)[\![f]\!]\rangle \\
& & \qquad\qquad \mid \mathbf{return}(u)\, w)\, v) \\
& \gtrsim_d & (\boldsymbol{\nu} vw)(\langle v \Leftarrow (\boldsymbol{\lambda} x)[\![g]\!]\rangle \mid \langle w \Leftarrow (\boldsymbol{\lambda} y)[\![f]\!]\rangle \mid w\, u\, v) \\
& \gtrsim_d & (\boldsymbol{\nu} vw)(\langle v \Leftarrow (\boldsymbol{\lambda} x)[\![g]\!]\rangle \mid [\![f]\!]\{^u/_y\}\, v) \\
\\
[\![\mathbf{let}\, y = u\, \mathbf{in}\, (\mathbf{let}\, x = f\, \mathbf{in}\, g)]\!] & = & (\boldsymbol{\nu} w)(\langle w \Leftarrow (\boldsymbol{\lambda} y)(\boldsymbol{\nu} v)(\langle v \Leftarrow (\boldsymbol{\lambda} x)[\![g]\!]\rangle \mid [\![f]\!]\, v)\rangle \\
& & \qquad\qquad \mid \mathbf{return}(u)\, w) \\
& \gtrsim_d & (\boldsymbol{\nu} w)(\langle w \Leftarrow (\boldsymbol{\lambda} y)(\boldsymbol{\nu} v)(\langle v \Leftarrow (\boldsymbol{\lambda} x)[\![g]\!]\rangle \mid [\![f]\!]\, v)\rangle \mid w\, u) \\
& \gtrsim_d & (\boldsymbol{\nu} vw)(\langle v \Leftarrow (\boldsymbol{\lambda} x)[\![g]\!]\rangle \mid [\![f]\!]\{^u/_y\}\, v)
\end{array}$$

Ce qui implique que $[\![\mathbf{let}\, x = (\mathbf{let}\, y = u\, \mathbf{in}\, f)\, \mathbf{in}\, g]\!] \approx_b [\![\mathbf{let}\, y = u\, \mathbf{in}\, (\mathbf{let}\, x = f\, \mathbf{in}\, g)]\!]$. Nous conjecturons la validité de cette règle dans le cas le plus général, néanmoins, comme nous l'avons dit précédemment, cette règle n'est pas nécessaire pour coder $\mathbf{conc}\varsigma$.

19.3 Conclusion

Dans ce chapitre, nous avons montré comment il est possible d'interpréter une extension concurrente de ς dans le calcul bleu. Cette interprétation a été donnée en utilisant un codage très proche de celui des objets fonctionnels, et nous pouvons d'ailleurs fournir des résultats similaire à ceux du chapitre 18, pour le typage des objets concurrents. Notre interprétation a aussi permis de valider notre codage des dénominations, qui apparaissent comme une catégorie importante de processus pour modéliser les objets, à l'exemple des définitions dans l'interprétation des fonctions, c'est-à-dire du λ-calcul. Ainsi, on peut considérer trois type de ressources. Les ressources unique: $\langle u \Leftarrow P\rangle$, qui sont consommées par la communication, les ressources répliquées: $\langle u = P\rangle$, qui sont infiniment disponible – et qui sont non modifiables – et finalement les dénominations: $\langle u \mapsto L\rangle$, qui peuvent s'interpréter comme des «déclarations linéaires». Il semble donc souhaitable d'incorporer directement les dénominations dans la syntaxe, comme un opérateur primitif de notre calcul.

Dans le chapitre suivant, nous nous intéressons à une nouveau modèle d'exécution des objets, qui consiste à permettre d'ajouter dynamiquement des méthodes à un objet. On trouve une définition formelle de ce modèle des *objets extensibles* dans [49], où l'auteur définit un calcul d'objets, dénoté $\boldsymbol{\lambda}\mathcal{O}\mathbf{bj}$, qui est basé sur le λ-calcul, et qui contient en plus des constructeurs de ς, un opérateur d'extension.

CHAPITRE 20

Objets extensibles

Une des raisons qui explique l'engouement des développeurs d'applications pour les «technologies objets», est que ce modèle de programmation favorise la réutilisation du code. Ainsi, dans les langages orientés objets par exemple, le mécanisme de l'héritage permet de réutiliser la définition d'une classe pour créer de nouvelles classes offrant plus de fonctionnalités. Cependant il existe une approche différente, celle des langages avec délégation, dans laquelle les classes sont remplacées par des *prototypes*, c'est-à-dire des objets extensibles, l'instanciation de classe est remplacée par le clonage, et l'héritage est remplacé par l'ajout de méthode.

Dans ce chapitre, nous nous intéressons à ce modèle d'objets. Plus précisément, nous donnons une interprétation du calcul $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$, défini par K. Fisher *et al.* dans [46, 48, 49], qui est un calcul d'objets fonctionnels et fortement typés, comme $\mathbf{Ob}_{1<:}$. À la différence des chapitres précédents, nous choisissons $\boldsymbol{\lambda\mathcal{D}}$ef, le calcul fonctionnel introduit dans le chapitre 3.4, comme calcul cible de notre interprétation, à la place du calcul bleu. Ce choix est motivé par le fait que le calcul étudié est fonctionnel, et donc que les «opérateurs concurrents» de $\boldsymbol{\pi}^\star$ ne sont pas utiles. Le codage des termes de $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ est donné dans la section 20.2, et nous étudions le typage et le sous-typage des objets dans la section 20.3. Le système de types cible que nous considérons est $\mathtt{BF}_\leqslant$ (cf. figure 14.4). Plus précisément, il s'agit de la restriction du système $\mathtt{BF}_\leqslant$ au opérateurs de $\boldsymbol{\lambda\mathcal{D}}$ef. Bien que la réduction de $\boldsymbol{\lambda\mathcal{D}}$ef diffère de celle du calcul bleu, le résultat de conservation du typage vaut toujours.

20.1 Le calcul des prototypes

Dans la suite de notre étude, nous utilisons indifféremment les termes prototypes et objets pour désigner les expressions du calcul de K. Fisher *et al.*. Nous verrons que ces termes correspondent à deux notions distinctes dans le calcul typé. Plus précisément, un objet est un prototype qui ne peut pas être étendu.

La syntaxe de $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ est donnée dans la figure 20.1. On remarque que le λ-calcul est inclus dans ce calcul. Ainsi, pour lier le paramètre self, on utilise la λ-abstraction plutôt qu'un lieur spécial comme ς, l'intuition étant que, dans l'extension $\langle e \leftarrow\!\!+ l = f\rangle$, le terme f est (ou se réduit en) une abstraction.

Remarque (Conventions) Nous pourrons noter $\langle l_i = f_i{}^{i\in[1..n]}\rangle$ – où les noms de méthodes dans $(l_i)_{i\in[1..n]}$ sont distincts – le prototype $\langle\langle\rangle \leftarrow\!\!+ l_1 = f_1\rangle \ldots \leftarrow\!\!+ l_n = f_n\rangle$, et nous utilisons la notation $\langle e \leftarrow\!\!\circ\, l = f\rangle$ pour désigner soit une extension $\langle e \leftarrow\!\!+ l = f\rangle$, soit une modification $\langle e \leftarrow l = f\rangle$. ■

Nous avons étendu la syntaxe initiale du calcul d'objets de K. FISHER *et al.*, en ajoutant un nouvel opérateur: Sel(e,l,obj), qui simplifie la définition de la relation de réduction. Intuitivement, le terme Sel(e,l,obj) représente la sélection de la méthode l dans l'objet e, mais avec le paramètre self lié à obj. On trouve la même notation dans d'autres études de $\boldsymbol{\lambda\mathcal{O}}$**bj** [13, 43].

prototypes:	obj	::=	$\langle\rangle$	prototype vide
		|	$\langle e \leftarrow\!\!+ l = f\rangle$	extension de l'objet e par la méthode l
		|	$\langle e \leftarrow l = f\rangle$	modification de la méthode l
valeurs:	v	::=	obj | $\lambda x.e$	
termes:	e,f,g	::=	x	variable
		|	$\lambda x.e$	λ-abstraction
		|	(ef)	application
		|	obj	prototypes
		|	$e \Leftarrow l$	invocation de la méthode l
		|	Sel(e,l,obj)	opération auxiliaire

Fig. 20.1: Syntaxe du calcul $\boldsymbol{\lambda\mathcal{O}}$**bj**

La relation de réduction entre termes de $\boldsymbol{\lambda\mathcal{O}}$**bj** est donnée dans la figure 20.2. On remarque l'utilisation de l'opération Sel(e,l,obj), à la place de la relation de «bookkeeping» trouvée dans [49]. On remarque aussi que la relation $\rightsquigarrow$ est paresseuse.

$$
\begin{array}{lll}
\text{(fm red beta)} & (\lambda x.e)f \rightsquigarrow e\{\!\{f/x\}\!\} & \\
\text{(fm red select)} & \text{obj} \Leftarrow l \rightsquigarrow \text{Sel(obj,}l\text{,obj)} & \\
\text{(fm red success)} & \text{Sel}(\langle e \leftarrow\!\!\circ\, l = f\rangle,l,e') \rightsquigarrow (f\ e') & \\
\text{(fm red next)} & \text{Sel}(\langle e \leftarrow\!\!\circ\, k = f\rangle,l,e') \rightsquigarrow \text{Sel}(e,l,e') & (k \neq l) \\
\text{(fm red context)} & e \rightsquigarrow e' \Rightarrow \left\{\begin{array}{l} (e\ f) \rightsquigarrow (e'\ f) \\ e \Leftarrow l \rightsquigarrow e' \Leftarrow l \\ \text{Sel}(e,l,f) \rightsquigarrow \text{Sel}(e',l,f) \end{array}\right. &
\end{array}
$$

Fig. 20.2: Réduction dans $\boldsymbol{\lambda\mathcal{O}}$**bj**

Pour donner une idée de l'expressivité du calcul de K. FISHER *et al.*, nous étudions l'exemple de l'objet point, qui est un classique de la littérature sur les langages à objets.

Exemple 20.1 (Le point mobile) Un exemple classique d'objet est celui du point, dans un espace à une dimension, dont la position peut être modifiée. C'est-à-dire un objet avec une méthode *pos* pour mémoriser la position du point, et une méthode *move* pour changer cette position.

$$
\begin{array}{rl}
\textsf{point} =_{\text{def}} & \langle\langle\rangle \leftarrow\!\!+ pos = \lambda x.0\rangle \\
& \leftarrow\!\!+ move = \lambda xd.(\langle x \leftarrow pos = \lambda y.(x \Leftarrow pos + d)\rangle)\rangle
\end{array}
$$

on montre alors les relations suivantes, qui sont respectivement un exemple d'invocation et d'extension de méthode, où $\approx_{\mathcal{O}}$ est la relation de conversion entre objet, c'est-à-dire la plus petite congruence qui contient $\rightsquigarrow$.

$$\begin{array}{rcl}(\mathbf{point} \Leftarrow\!\!\!+ \mathit{move}\ 5) & \overset{*}{\rightsquigarrow} & \lambda x d.\langle x \leftarrow \mathit{pos} = \lambda y.(x \Leftarrow \mathit{pos} + d)\rangle\ \mathbf{point}\,5 \\ & \overset{*}{\rightsquigarrow} & \langle \mathbf{point} \leftarrow \mathit{pos} = \lambda y.(\mathbf{point} \Leftarrow \mathit{pos} + 5)\rangle \\ & \approx_{\mathcal{O}} & \langle \mathbf{point} \leftarrow \mathit{pos} = \lambda y.5\rangle \\ & \approx_{\mathcal{O}} & \langle \mathit{pos} = \lambda y.5, \mathit{move} = \cdots\rangle \end{array}$$

$$\langle \mathbf{point} \leftarrow\!\!\!+ \mathit{color} = \lambda x.\mathrm{red}\rangle \approx_{\mathcal{O}} \langle \mathit{pos} = \lambda x.0, \mathit{move} = \lambda x.(\cdots), \mathit{color} = \lambda x.\mathrm{red}\rangle$$

□

Le calcul $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ est fortement typé, néanmoins le système de types est assez complexe et n'est introduit que dans la section 20.3. Ainsi, nous préférons d'abord définir notre codage des objets dans $\boldsymbol{\lambda\mathcal{D}}$ef.

20.2 Interprétation

Dans notre interprétation, nous représentons un objet $\langle l_i = f_i{}^{i\in[1..n]}\rangle$ par une définition récursive qui encapsule la fonction:

$$G \;=\; (\boldsymbol{\lambda}\mathrm{self})[\, l_1 = (M_1\ \mathrm{self}), \ldots, l_n = (M_n\ \mathrm{self})\,]$$

C'est-à-dire la fonction qui, étant donné un objet, retourne un enregistrement de toutes ses méthodes. On retrouve la fonction G dans d'autres interprétations des objets: dans les travaux de W. Cook [34] par exemple, ou dans [4, 2], où G est nommé *générateur d'objets*. On peut remarquer que dans le codage des objets de $\boldsymbol{\varsigma}$, nous utilisions un autre type de générateur d'objets, c'est-à-dire l'enregistrement: $[\, l_i = (\boldsymbol{\lambda}\mathrm{self})M_i{}^{i\in[1..n]}\,]$.

On peut utiliser la fonction G pour définir une interprétation très simple des objets, que nous donnons dans la définition 20.1. Nous nommons cette interprétation *self-application semantics* [3], car la sélection $(\!|e \Leftarrow l|\!)$ est codée par l'application du générateur d'objet $(\!|e|\!)$ à lui-même. Il faut noter que dans [3], l'interprétation de la sélection est $((\!|e|\!)\cdot l)\,(\!|e|\!)$, et non pas $((\!|e|\!)\,(\!|e|\!))\cdot l$.

Définition 20.1 (Self-application) Soit G la fonction $(\boldsymbol{\lambda}\mathrm{self})[\, l_i = ((\!|e_i|\!)\ \mathrm{self})^{i\in[1..n]}\,]$, la *self-application semantics* est le nom donné au codage $(\!|.|\!)$, tel que $(\!|.|\!)$ est un homomorphisme sur les opérateurs du λ-calcul et tel que:

$$\begin{array}{rcl}(\!|\langle l_i = f_i{}^{i\in[1..n]}\rangle|\!) & = & G \\ (\!|e \Leftarrow l|\!) & = & ((\!|e|\!)\ (\!|e|\!))\cdot l \\ (\!|\mathrm{Sel}(e,l,f)|\!) & = & ((\!|e|\!)\ (\!|f|\!))\cdot l \\ (\!|\langle e \leftarrow\!\!\!\circ\, l = f\rangle|\!) & = & (\boldsymbol{\lambda}\mathrm{self})[\,(\!|e|\!)\ \mathrm{self}\,, l = (\!|f|\!)\ \mathrm{self}\,] \end{array}$$

La self-application semantics permet de simuler la réduction des objets, dans le sens où on montre que si $e \rightsquigarrow e'$, alors $(\!|e|\!) \overset{*}{\rightarrow} (\!|e'|\!)$. Cependant elle pose de difficiles problèmes de typage, aussi nous proposons une nouvelle interprétation, dans laquelle nous séparons les deux usages possible d'un prototype: l'invocation de méthodes et l'extension.

L'idée de séparer les différents usages d'un objet n'est pas nouvelle, on la retrouve dans la «split method» [2, 137] utilisée dans l'interprétation de $\mathbf{Ob}_{1<:}$. Néanmoins nous devons aménager la technique de la split method pour permettre d'augmenter l'ensemble des méthodes d'un objet, ensemble qui n'est pas fixe dans notre cas.

Avant de donner notre codage des objets, nous définissons le processus générateur de prototypes: **Proto**. Nous désignons par $\mathbf{P}_{(s,x)}$, l'enregistrement suivant.

$$\mathbf{P}_{(s,x)} =_{\text{def}} [\, inht = x, invk = x(sx) \,]$$

Le codage des objets utilise un processus dont la définition est similaire à celle de la cellule mutable (cf. section 16.2). Nous désignons par **Proto** la définition récursive suivante.

$$\mathbf{Proto} \;=_{\text{def}}\; \mathbf{rec}\, s.(\boldsymbol{\lambda} x)\mathbf{P}_{(s,x)} \tag{20.1}$$

Intuitivement, le champ *inht* sert à interpréter l'extension – ou encore l'héritage –, tandis que *invk* sert à interpréter la sélection d'une méthode. En particulier, si on se représente x comme étant l'état de l'objet – c'est-à-dire un générateur d'objets comme dans la self-application semantics –, le champ *inht* permet de recréer un nouveau générateur en étendant le générateur «courant», tandis que le champ *invk* applique le générateur aux termes (sx), c'est-à-dire à une copie de l'objet.

En utilisant la définition de l'équivalence $\approx_{\mathcal{D}}$, entre termes de $\boldsymbol{\lambda}\mathcal{D}$ef (cf. section 3.4), on montre que:

$$\begin{array}{rcl} (\mathbf{Proto} M) & \approx_{\mathcal{D}} & \mathbf{def}\, s = (\boldsymbol{\lambda} x)\mathbf{P}_{(s,x)}\, \mathbf{in}\, (s\ M) \\ & \approx_{\mathcal{D}} & \mathbf{def}\, s = (\boldsymbol{\lambda} x)\mathbf{P}_{(s,x)}, x = M\, \mathbf{in}\, [\, inht = x, invk = x(sx) \,] \\ & \approx_{\mathcal{D}} & [\, inht = M, invk = M(\mathbf{Proto} M) \,] \end{array} \tag{20.2}$$

Où la notation $\mathbf{def}\, x_1 = N_1, x_2 = N_2\, \mathbf{in}\, M$ désigne le terme $\mathbf{def}\, x_1 = N_1\, \mathbf{in}\, (\mathbf{def}\, x_2 = N_2\, \mathbf{in}\, M)$.

Le codage des objets dans $\boldsymbol{\lambda}\mathcal{D}$ef, dénoté $[\![.]\!]$, est donné dans la définition suivante. On a cependant omis le codage des opérateurs du λ-calcul, qui est trivial.

Définition 20.2 (Codage de $\boldsymbol{\lambda}\mathcal{O}$bj dans $\boldsymbol{\lambda}\mathcal{D}$ef) On suppose que x est un nouveau nom.

$$\begin{array}{rcl} [\![\langle\rangle]\!] & = & \mathbf{Proto}\ (\boldsymbol{\lambda} x)[\,] \\ [\![e \Leftarrow l]\!] & = & [\![e]\!] \cdot invk \cdot l \\ [\![\mathrm{Sel}(e,l,f)]\!] & = & (([\![e]\!] \cdot inht)\, [\![f]\!]) \cdot l \\ [\![\langle e \hookleftarrow l = f \rangle]\!] & = & \mathbf{Proto}\ (\boldsymbol{\lambda} x)[\, ([\![e]\!] \cdot inht\ x)\, , l = ([\![f]\!]\ x) \,] \end{array}$$

On peut montrer que, si G est le générateur d'objet $(\boldsymbol{\lambda} x)[\, l_i = (\!|e_i|\!)\ x)^{i \in [1..n]}\,]$, alors le codage du prototype $\langle l_i = {e_i}^{i \in [1..n]} \rangle$ est équivalent à $(\mathbf{Proto}\ G)$, et donc est équivalent à l'enregistrement $[\, inht = G, invk = G(\mathbf{Proto}\ G) \,]$ (cf. équation (20.2)). En particulier, on remarque que le générateur d'objet associé à un prototype e, est le terme $([\![e]\!] \cdot inht)$. On remarque aussi que dans notre codage, on peut définir une opération d'extraction de (pré)méthodes, notée $e \hookrightarrow l$, qui se code de la manière suivante.

$$[\![e \hookrightarrow l]\!] = (\boldsymbol{\lambda} x)([\![e]\!] \cdot inht\ x\ \cdot l)$$

Ce qui correspond au fait que la (pré)méthode de nom l, du générateur G, est l'abstraction $(\boldsymbol{\lambda} x)(G\ x \cdot l)$.

Notre interprétation est assez similaire au modèle des objets de W. Cook, dans lequel on sépare les objets des générateurs d'objets, et où l'héritage n'agit que sur les générateurs. On retrouve également le fait que extension et modification de méthodes sont deux opérations opérationnellement équivalentes. Cependant on trouve aussi des différences avec ce modèle:

- nous réunissons les deux notions de générateurs et d'objets au sein d'un même enregistrement $([\, inht = G, invk = G(\mathbf{Proto}\ G) \,])$. Ce choix permet de simplifier le typage;

– les objets peuvent être dynamiquement étendus.

Nous montrons maintenant que notre codage est correct.

Théorème 20.1 ($\lambda\mathcal{D}$ef simule $\lambda\mathcal{O}$bj) *Si $e \rightsquigarrow e'$, alors $[\![e]\!] \approx_{\mathcal{D}} [\![e']\!]$.*

Preuve (théorème 20.1) La preuve est faite par induction sur l'inférence de $e \rightsquigarrow e'$. Dans la preuve, on utilise le fait que si e est de la forme $\langle\rangle$ ou $\langle e \hookleftarrow l = f\rangle$, alors $[\![e]\!]$ est équivalent à $[\, inht = G, invk = G(\mathbf{Proto}\ G)\,]$, où G est un générateur d'objet. Ce résultat est impliqué par l'équation (20.2) et la définition de $[\![e]\!]$.

– **cas (fm red beta):** on suppose que $x \notin \mathbf{fn}(f)$. le codage d'un rédex se réduit de la façon suivante.

$$[\![(\lambda x.e)\ f]\!] = (((\boldsymbol{\lambda} x)[\![e]\!])\ [\![f]\!]) \approx_{\mathcal{D}} (\mathbf{def}\, x = [\![f]\!]\, \mathbf{in}\, [\![e]\!]) \approx_{\mathcal{D}} ([\![e]\!]\{\!\{[\![f]\!]/x\}\!\})$$

– **cas (fm red select):** on montre que $[\![\mathrm{obj} \Leftarrow l]\!] \approx_{\mathcal{D}} [\![\mathrm{Sel}(\mathrm{obj},l,\mathrm{obj})]\!]$. Par définition obj est de la forme $\langle\rangle$ ou $\langle e \hookleftarrow l = f\rangle$. Il existe donc un enregistrement R tel que $[\![\mathrm{obj}]\!] = (\mathbf{Proto}\ (\boldsymbol{\lambda} x)R)$, et on a:

$$\begin{array}{rcl} [\![\mathrm{obj} \Leftarrow l]\!] & \approx_{\mathcal{D}} & [\, inht = (\boldsymbol{\lambda} x)R, invk = (\boldsymbol{\lambda} x)R\ (\mathbf{Proto}\ (\boldsymbol{\lambda} x)R)\,]\ \cdot invk\ \cdot l \\ & \approx_{\mathcal{D}} & ((\boldsymbol{\lambda} x)R\ (\mathbf{Proto}\ (\boldsymbol{\lambda} x)R))\ \cdot l \\ & \approx_{\mathcal{D}} & ((\boldsymbol{\lambda} x)R\ [\![\mathrm{obj}]\!])\ \cdot l \end{array}$$

$$\begin{array}{rcl} [\![\mathrm{Sel}(\mathrm{obj},l,\mathrm{obj})]\!] & \approx_{\mathcal{D}} & ([\, inht = (\boldsymbol{\lambda} x)R, invk = (\boldsymbol{\lambda} x)R\ (\mathbf{Proto}\ (\boldsymbol{\lambda} x)R)\,]\ \cdot inht\ [\![\mathrm{obj}]\!]) \cdot l \\ & \approx_{\mathcal{D}} & ((\boldsymbol{\lambda} x)R\ [\![\mathrm{obj}]\!])\ \cdot l \end{array}$$

Par conséquent $[\![\mathrm{obj} \Leftarrow l]\!] \approx_{\mathcal{D}} [\![\mathrm{Sel}(\mathrm{obj},l,\mathrm{obj})]\!]$, ce qui est le résultat attendu;

– **cas (fm red success):** par définition $[\![\langle e \hookleftarrow l = f\rangle]\!]$ est le terme $(\mathbf{Proto}\ (\boldsymbol{\lambda} x)R)$, où R est l'enregistrement $[\, [\![e]\!]\ \cdot inht\ x\,, l = [\![f]\!]\ x\,]$ (on suppose que $x \notin \mathbf{fn}(f)$). Par conséquent $[\![\mathrm{Sel}(\langle e \hookleftarrow l = f\rangle, l, e')]\!]$ est le terme $((\mathbf{Proto}\ (\boldsymbol{\lambda} x)R)\ \cdot inht\ [\![e']\!]\ \cdot l)$, et on a:

$$\begin{array}{rcl} [\![\mathrm{Sel}(\langle e \hookleftarrow l = f\rangle, l, \mathrm{obj})]\!] & \approx_{\mathcal{D}} & (\boldsymbol{\lambda} x)R\ [\![e']\!]\ \cdot l \\ & \approx_{\mathcal{D}} & \mathbf{def}\, x = [\![e']\!]\, \mathbf{in}\, [\, [\![e]\!]\ \cdot inht\ x\,, l = [\![f]\!]\ x\,]\ \cdot l \\ & \approx_{\mathcal{D}} & \mathbf{def}\, x = [\![e']\!]\, \mathbf{in}\, ([\![f]\!]\ x) \\ & \approx_{\mathcal{D}} & ([\![f]\!]\ [\![e']\!]) \end{array}$$

Ce qui est le résultat attendu. La preuve est similaire dans le cas (fm red next);

– **cas (fm red context):** il suffit d'utiliser le fait que $\approx_{\mathcal{D}}$ est une congruence.

On peut démontrer un résultat plus général que le théorème 20.1 [17], qui relie la convergence d'un objet à celle de son interprétation. On dit que le prototype e converge, noté $e \Downarrow$, s'il existe une valeur telle que $e \stackrel{*}{\rightsquigarrow} v$. Comme l'évaluation est déterministe, cette valeur, si elle existe, est unique. Nous pouvons alors la noter eval(e). De même, on peut définir le terme eval(M), si le terme M de $\boldsymbol{\lambda}\mathcal{D}$ef converge.

Proposition 20.2 *Un prototype e converge si et seulement si son interprétation converge, c'est-à-dire que $e \Downarrow \iff [\![e]\!] \Downarrow$.*

Cependant nous ne donnons pas la preuve de ce résultat ici, car celle-ci est assez technique et n'apporte pas d'éclaircissement sur notre codage.

Après avoir étudié la sémantique opérationnelle des objets, nous nous intéressons dans la section suivante à leur typage. Comme pour l'étude de la réduction, nous commençons par définir le système de types de $\boldsymbol{\lambda}\mathcal{O}\mathbf{bj}$, puis nous donnons une interprétation de ces termes dans le système $\mathtt{BF}_{\leqslant}$, et nous prouvons un résultat de correspondance.

20.3 Système de types pour les prototypes

Depuis la première définition du calcul des prototypes, plusieurs systèmes de types ont été proposés. La définition de plusieurs systèmes de types se comprend très bien, puisque c'est le typage des objets qui est le problème difficile, plus que l'étude de leur comportement opérationnel, C'est également l'étude des systèmes de types qui a motivé l'étude des calculs d'objets. En effet on peut dire, de manière grossière, que ce qui a motivé la définition des calcul d'objets [3], est l'échec des tentatives de donner une interprétation des langages à objets dans le λ-calcul qui permette d'obtenir des systèmes de types suffisamment puissant. En particulier en ce qui concerne le sous-typage.

Comme dans le cas des extensions typées de $\boldsymbol{\varsigma}$, les systèmes de types pour $\boldsymbol{\lambda\mathcal{O}\mathbf{bj}}$ sont basés sur un opérateur primitif, qui est noté $(\mathbf{pro}\, t.[\, l_i : \tau_i{}^{i\in[1..n]}\,])$ dans ce cas, car il s'agit du type des prototypes. Le système défini dans [48] est fondamentalement celui de [46], enrichi avec une forme de sous-typage. Dans notre interprétation, nous nous intéressons à une version simplifiée de ce système, qui est décrit dans la thèse de K. Fisher [49]. Plus précisément, nous considérons le système défini dans le chapitre 3 de [49], qui ne contient ni quantification existentielle, ni annotations de variance.

Nous commençons par décrire le système de types, la relation de sous-typage étant définie dans la section 20.5. Dans le système que nous étudions, on considère en plus des types et des sortes, une catégorie particulière d'expressions nommées *rangées*, et qui servent à définir l'interface des objets. Intuitivement, une rangée est l'équivalent des types enregistrement de $\mathtt{BF}_{\leqslant}$.

Définition 20.3 (Sortes, types et rangées) Les sortes sont engendrées par la grammaire suivante.

$$\kappa ::= \mathbf{T} \mid \mathbf{M} \mid \mathbf{T} \to \mathbf{M} \qquad \text{avec } \mathbf{M} = \{l_1, \ldots, l_n\}$$

On considère deux ensembles distincts de noms de variables: un pour les types et l'autre pour les rangées. Les rangées et les expressions de types sont engendrées par la grammaire suivante.

$$\begin{array}{rrcl} \text{rangées} & \varrho & ::= & r \mid [\,] \mid [\,\varrho\,, l : \tau\,] \mid (\lambda t.\varrho) \mid (\varrho\tau) \\ \text{types} & \tau,\sigma & ::= & t \mid (\tau \to \sigma) \mid (\mathbf{pro}\, t.\varrho) \end{array}$$

La sorte $\mathbf{T}$ est la sorte des types bien formés, tandis que $\{l_1, \ldots, l_m\}$ est la sorte des rangées qui ne contiennent pas les méthodes $(l_i)_{i\in[1..n]}$. La grammaire des sortes contient aussi le terme $\mathbf{T} \to \mathbf{M}$, qui est la sorte des interfaces, c'est-à-dire des fonctions des types vers les rangées. Les interfaces correspondent à la seule forme possible d'abstraction sur les types. Comme dans $\mathtt{BF}_{\leqslant}$, les environnements de typage contiennent des associations entre variables et types/sortes, et des contraintes de sous-typage:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, t :: \kappa \mid \Gamma, r <:_w \varrho$$

La contrainte $(r <:_w \varrho)$ fait intervenir la relation de sous-typage en largeur $<:_w$, qui est définie dans la figure 20.4. Dans le système de [49], on trouve une information de sorte avec chaque association: $(r <:_w \varrho :: \kappa)$, qui contraint la variable de rangée r à être instanciée par des types de sorte κ. L'ajout de cette hypothèse sur la sorte des variables de types – dans les environnements –, est motivé par l'utilisation des types existentiels dans [49]. Nous ne nous préoccupons pas des types existentiels dans notre étude, aussi nous supposons que κ est toujours égal à $\mathbf{T}\to\emptyset$ dans notre version simplifiée du système de types, et nous omettons cette annotation dans la suite. Nous conjecturons que cette hypothèse n'est pas trop contraignante en l'absence de types existentiels, dans le sens où on ne modifie pas l'ensemble des termes typables. Cette conjecture est aussi partagée par K. Fisher [50].

Les différents jugements du système de types de $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ sont les suivants:

$\Gamma \vdash *$	environnements bien formés	$\Gamma \vdash \varrho <:_w \varrho'$	sous-typage des rangées
$\Gamma \vdash \tau :: \mathbf{T}$	type bien formé	$\Gamma \vdash P : \tau$	le terme a le type
$\Gamma \vdash \varrho :: \kappa$	la rangée a la sorte		

La définition de ces jugements est donnée dans les figures 20.3 à 20.5, cependant nous omettons de donner les règles d'affaiblissement pour ces jugements, qui sont triviales. On définit aussi le méta-jugement $\Gamma \vdash \tau \cong_w \sigma$, qui est un abrégé pour $\Gamma \vdash \tau <:_w \sigma$ et $\Gamma \vdash \sigma <:_w \tau$.

$$\emptyset \vdash * \qquad \frac{\Gamma \vdash \tau :: \mathbf{T} \quad x \notin \mathbf{dom}(\Gamma)}{\Gamma,x : \tau \vdash *} \qquad \frac{\Gamma \vdash * \quad t \notin \mathbf{fn}(\Gamma)}{\Gamma,t :: \mathbf{T} \vdash *}$$

$$\frac{\Gamma \vdash \varrho :: \mathbf{T} \to \mathbf{M} \quad r \notin \mathbf{dom}(\Gamma)}{\Gamma,r <:_w \varrho \vdash *}$$

$$\frac{\Gamma \vdash * \quad (t :: \mathbf{T}) \in \Gamma}{\Gamma \vdash t :: \mathbf{T}} \qquad \frac{\Gamma \vdash \tau :: \mathbf{T} \quad \Gamma \vdash \sigma :: \mathbf{T}}{\Gamma \vdash (\tau \to \sigma) :: \mathbf{T}} \qquad \frac{\Gamma,t :: \mathbf{T} \vdash \varrho :: \mathbf{M}}{\Gamma \vdash (\mathbf{pro}\, t.\varrho) :: \mathbf{T}}$$

$$\frac{\Gamma \vdash * \quad (r <:_w \varrho) \in \Gamma}{\Gamma \vdash r :: \mathbf{T} \to \emptyset} \qquad \frac{\Gamma \vdash \varrho :: \mathbf{T} \to \mathbf{M} \quad \mathbf{M}' \subseteq \mathbf{M}}{\Gamma \vdash \varrho :: \mathbf{T} \to \mathbf{M}'}$$

$$\frac{\Gamma,t :: \mathbf{T} \vdash \varrho :: \mathbf{M}}{\Gamma \vdash \lambda t.\varrho :: \mathbf{T} \to \mathbf{M}} \qquad \frac{\Gamma \vdash \varrho :: \mathbf{T} \to \mathbf{M} \quad \Gamma \vdash \tau :: \mathbf{T}}{\Gamma \vdash (\varrho\tau) :: \mathbf{M}}$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash [\,] :: \mathbf{M}} \qquad \frac{\Gamma \vdash \varrho :: \mathbf{M} \uplus \{l\} \quad \Gamma \vdash \tau :: \mathbf{T}}{\Gamma \vdash [\,\varrho\,, l : \tau\,] :: \mathbf{M}}$$

Fig. 20.3: Environnement bien formés et système de sortes

On remarque que la définition d'environnement bien formé implique que si $\Gamma,t :: \mathbf{T} \vdash *$, alors t n'apparaît pas libre dans Γ. Le système de sortes est assez usuel, on remarque néanmoins que la règle de construction des rangées $[\,\varrho\,, l : \tau\,]$, interdit d'ajouter un champ qui est déjà présent. Ainsi la rangée $[\,l : \sigma\,, l : \tau\,]$ n'est pas bien formée dans ce système. Ceci explique les différences de définitions entre les relations de sous-typage en largeur de $\mathtt{BF}_{\leqslant}$ et de $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$.

$$\frac{\Gamma \vdash \varrho :: \kappa}{\Gamma \vdash \varrho <:_w \varrho} \qquad \frac{\Gamma \vdash * \quad (r <:_w \varrho) \in \Gamma}{\Gamma \vdash r <:_w \varrho} \qquad \frac{\Gamma \vdash \varrho_1 <:_w \varrho_2 \quad \Gamma \vdash \varrho_2 <:_w \varrho_3}{\Gamma \vdash \varrho_1 <:_w \varrho_3}$$

$$\frac{\Gamma,t :: \mathbf{T} \vdash \varrho_1 <:_w \varrho_2 \quad \Gamma \vdash \varrho_2 :: \mathbf{M}}{\Gamma \vdash \lambda t.\varrho_1 <:_w \lambda t.\varrho_2} \qquad \frac{\Gamma \vdash \varrho_1 <:_w \varrho_2 \quad \Gamma \vdash \varrho_2 :: \mathbf{T} \to \mathbf{M} \quad \Gamma \vdash \tau :: \mathbf{T}}{\Gamma \vdash (\varrho_1\tau) <:_w (\varrho_2\tau)}$$

$$\frac{\Gamma \vdash \varrho_1 <:_w (\lambda t.\varrho_2)\ \tau}{\Gamma \vdash \varrho_1 <:_w \varrho_2\{\tau/t\}} \qquad \frac{\Gamma \vdash (\lambda t.\varrho_1)\ \tau <:_w \varrho_2}{\Gamma \vdash \varrho_1\{\tau/t\} <:_w \varrho_2}$$

$$\frac{\Gamma \vdash \varrho_1 <:_w \varrho_2 \quad \Gamma \vdash \tau_1 \cong_w \tau_2 \quad \Gamma \vdash [\,\varrho_i\,, l : \tau_i\,] :: \mathbf{M}_i \quad i \in \{1,2\}}{\Gamma \vdash [\,\varrho_1\,, l : \tau_1\,] <:_w [\,\varrho_2\,, l : \tau_2\,]}$$

$$\frac{\Gamma \vdash \varrho_1 <:_w \varrho_2 \quad \Gamma \vdash [\,\varrho_1\,, l : \tau\,] :: \mathbf{M}}{\Gamma \vdash [\,\varrho_1\,, l : \tau\,] <:_w \varrho_2}$$

Fig. 20.4: Sous-typage en largeur: $<:_w$

Les règles de typage des prototypes sont données dans la figure 20.5. Les règles pour la partie λ-calcul sont usuelles et correspondent au système de types simples de $\boldsymbol{\Lambda}$. En ce qui concerne la

$$\frac{\Gamma \vdash * \qquad (x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \text{ (pro ax)} \qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \to \sigma} \text{ (pro abs)}$$

$$\frac{\Gamma \vdash e : \tau \to \sigma \qquad \Gamma \vdash f : \tau}{\Gamma \vdash (ef) : \sigma} \text{ (pro app)} \qquad \frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : (\mathbf{pro}\, t.[\,])} \text{ (pro void)}$$

$$\frac{\Gamma \vdash e : (\mathbf{pro}\, t.\varrho) \qquad \Gamma, t :: \mathbf{T} \vdash \varrho <:_w [\, l : \tau \,]}{\Gamma \vdash e \Leftarrow l : \tau\{(\mathbf{pro}\, t.\varrho)/t\}} \text{ (pro invk)}$$

$$\frac{\Gamma \vdash e : (\mathbf{pro}\, t.\varrho) \qquad \Gamma, t :: \mathbf{T} \vdash \varrho :: \{l\} \qquad \Gamma, r <:_w \lambda t.[\, \varrho\,, l : \tau \,] \vdash f : (t \to \tau)\{(\mathbf{pro}\, t.(r\ t))/t\}}{\Gamma \vdash \langle e \leftarrow\!\!+ l = f \rangle : (\mathbf{pro}\, t.[\, \varrho\,, l : \tau \,])} \text{ (pro ext)}$$

$$\frac{\Gamma \vdash e : (\mathbf{pro}\, t.\varrho) \qquad \Gamma, t :: \mathbf{T} \vdash \varrho <:_w [\, l : \tau \,] \qquad \Gamma, r <:_w \lambda t.\varrho \vdash f : (t \to \tau)\{(\mathbf{pro}\, t.(r\ t))/t\}}{\Gamma \vdash \langle e \leftarrow l = f \rangle : (\mathbf{pro}\, t.\varrho)} \text{ (pro over)}$$

Fig. 20.5: Système de types de $\boldsymbol{\lambda}\mathcal{O}\mathbf{bj}$

$$\frac{\Gamma \vdash * \qquad (x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \text{ (type ax)} \qquad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash (\boldsymbol{\lambda} x) M : \tau \to \sigma} \text{ (type abs)}$$

$$\frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash (MN) : \sigma} \text{ (type app)} \qquad \frac{\Gamma \vdash *}{\Gamma \vdash [\,] : [\,]} \text{ (type void)}$$

$$\frac{\Gamma \vdash M : [\, l : \sigma \,]}{\Gamma \vdash (M \cdot l) : \sigma} \text{ (type sel)} \qquad \frac{\Gamma \vdash M : \varrho \qquad \Gamma \vdash N : \tau}{\Gamma \vdash [\, M\,, l = N \,] : [\, \varrho\,, l : \tau \,]} \text{ (type over)}$$

$$\frac{\Gamma, x : \tau \vdash N : \tau \qquad \Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \mathbf{def}\, x = N\, \mathbf{in}\, M : \tau} \text{ (type def)}$$

$$\frac{\Gamma \vdash \tau \leqslant \sigma \qquad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma} \text{ (type sub)} \qquad \frac{\Gamma \vdash M : \sigma \qquad \Gamma, \Gamma' \vdash *}{\Gamma, \Gamma' \vdash M : \sigma} \text{ (type weak)}$$

$$\frac{\Gamma \vdash \tau' \sqsubseteq \tau \qquad \Gamma \vdash M : (\forall t \sqsubseteq \tau\,.\,\sigma)}{\Gamma \vdash M : \sigma\{\tau'/t\}} \text{ (type inst)} \qquad \frac{\Gamma, t \sqsubseteq \tau \vdash M : \sigma}{\Gamma \vdash M : (\forall t \sqsubseteq \tau\,.\,\sigma)} \text{ (type gen)}$$

Fig. 20.6: Système $\mathtt{BF}_{\leqslant}$ restreint aux opérateurs de $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$

partie objet, la règle de typage de l'invocation, règle (pro invk), indique que le paramètre t dans le type $(\mathbf{pro}\, t.\varrho)$, représente le type de l'objet. C'est la notion de «*self type*» qu'on retrouve par exemple dans [3].

On remarque l'existence de deux règles distinctes pour typer l'extension et la modification:

- dans la règle (pro ext), le jugement $\Gamma, t :: \mathbf{T} \vdash \varrho :: \{l\}$ indique que le prototype e ne possède pas de méthode nommée l, on est donc dans le cas d'une «vraie extension ».
- dans la règle (pro over), l'hypothèse $\Gamma, t :: \mathbf{T} \vdash \varrho <:_w [\, l : \tau \,]$ indique que le prototype e possède une méthode l, de type τ. Cette règle autorise donc la modification d'une méthode, à la condition que la nouvelle méthode soit du même type. Ainsi, on retrouve dans ce cas le même mécanisme de typage que dans $\mathbf{Ob}_{1<:}$.

Nous donnons un exemple d'utilisation de ces règles de typage, en étudiant le typage de l'objet **point**.

Exemple 20.2 (Type de point) Nous montrons que le prototype **point**, défini dans l'exemple 20.1, a le type $(\mathbf{pro}\, t.[\, pos : \mathtt{int}, move : (\mathtt{int} \to t)\,])$. Il est facile de voir que le type de la méthode *pos* est $(\tau \to \mathtt{int})$ pour n'importe quel τ, en effet:

$$r <:_w \lambda t.[\, pos : int\,] \vdash \lambda x.0 : (t \to \mathtt{int})\{(\mathbf{pro}\, t.(r\ t))/t\}$$

et par conséquent, en utilisant les règles (pro void) et (pro ext), on obtient que:

$$\emptyset \vdash \langle pos = \lambda x.0\rangle : (\mathbf{pro}\, t.[\, pos : \mathtt{int}\,]) \tag{20.3}$$

Nous montrons maintenant que le type de *move* est $(\tau \to (\mathtt{int} \to \tau))$, où τ est le type $(\mathbf{pro}\, t.(r\ t))$ avec la contrainte que $r <:_w \lambda t.[\, pos : \mathtt{int}, move : (\mathtt{int} \to t)\,]$. Soit ϱ_p la rangée $[\, pos : \mathtt{int}, move : (\mathtt{int} \to t)\,]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{r <:_w \lambda t.\varrho_p, x : (\mathbf{pro}\, t.(r\ t)) \vdash x : (\mathbf{pro}\, t.(r\ t)) \qquad \varrho_p <:_w [\, pos : \mathtt{int}\,]}{r <:_w \lambda t.\varrho_p, x : (\mathbf{pro}\, t.(r\ t)) \vdash x \Leftarrow pos : \mathtt{int}}\ \text{(pro invk)}}{r <:_w \lambda t.\varrho_p, d : \mathtt{int}, x : (\mathbf{pro}\, t.(r\ t)) \vdash \lambda y.(x \Leftarrow pos + d) : (t \to \mathtt{int})\{(\mathbf{pro}\, t.(r\ t))/t\}} \qquad r <:_w \lambda t.\varrho_p, t :: \mathbf{T} \vdash (r\ t) <:_w \varrho_p <:_w [\, pos : \mathtt{int}\,]}{r <:_w \lambda t.\varrho_p, d : \mathtt{int}, x : (\mathbf{pro}\, t.(r\ t)) \vdash x : (\mathbf{pro}\, t.(r\ t))}}{r <:_w \lambda t.\varrho_p, d : \mathtt{int}, x : (\mathbf{pro}\, t.(r\ t)) \vdash \langle x \leftarrow pos = \lambda y.(x \Leftarrow pos + d)\rangle : (\mathbf{pro}\, t.(r\ t))}\ \text{(pro over)}}{r <:_w \lambda t.\varrho_p, \vdash \lambda x d.\langle x \leftarrow pos = \lambda y.(x \Leftarrow pos + d)\rangle : (t \to (\mathtt{int} \to t))\{(\mathbf{pro}\, t.(r\ t))/t\}}$$

et par conséquent, toujours en utilisant la règle (pro ext), on montre que:

$$\dfrac{\begin{array}{c}\emptyset \vdash \langle pos = \lambda x.0\rangle : (\mathbf{pro}\, t.[\, pos : \mathtt{int}\,]) \qquad t :: \mathbf{T} \vdash [\, pos : \mathtt{int}\,] :: \{move\} \\ r <:_w \lambda t.\varrho_p \vdash \lambda x d.\langle x \leftarrow pos = \lambda y.(x \Leftarrow pos + d)\rangle : (t \to (\mathtt{int} \to t))\{(\mathbf{pro}\, t.(r\ t))/t\}\end{array}}{\emptyset \vdash \langle pos = \lambda x.0, move = \lambda x d. \ldots\rangle : (\mathbf{pro}\, t.[\, pos : \mathtt{int}, move : (\mathtt{int} \to t)\,])} \tag{20.4}$$

ce qui est le type annoncé. □

Comme nous avons ajouté l'opérateur $\mathrm{Sel}(e,l,f)$ à la syntaxe des objets considéré dans [49], nous ajoutons aussi une règle de typage pour cet opérateur.

$$\dfrac{\Gamma \vdash e : (\mathbf{pro}\, t.\varrho_1) \qquad \Gamma \vdash f : (\mathbf{pro}\, t.\varrho_2) \qquad \Gamma, t :: \mathbf{T} \vdash \varrho_2 <:_w \varrho_1 <:_w [\, l : \tau\,]}{\Gamma \vdash \mathrm{Sel}(e,l,f) : \tau\{(\mathbf{pro}\, t.\varrho_2)/t\}}\ \text{(pro select)}$$

En utilisant les résultats de la section suivante, nous montrerons que cette règle est valide.

20.4 Interprétation des types

Dans cette section, nous montrons comment on peut dériver les règles de typage des prototypes en utilisant notre codage des objets dans $\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$, et le système $\mathtt{BF}_{\leqslant}$. Nous rappelons, dans la

figure 20.6, les règles de types associées aux opérateurs de $\boldsymbol{\lambda}\mathcal{D}$ef.

Afin de simplifier notre présentation, nous définissons deux abréviations pour le codage des prototypes. Nous notons $M \Leftarrow l$ le terme $(M \cdot \mathit{invk} \cdot l)$, et $\langle\!\langle M \leftarrow l = N \rangle\!\rangle$ le terme $(\textbf{Proto}\ (\boldsymbol{\lambda}x)[\, M \cdot \mathit{inht}\ x\,, l = N\ x\,])$. Ainsi l'interprétation des prototypes peut se définir directement par $[\![e]\!] \Leftarrow l$ pour $[\![e \Leftarrow l]\!]$, et par $\langle\!\langle [\![e]\!] \leftarrow l = [\![f]\!] \rangle\!\rangle$ pour $[\![\langle e \leftarrow\!\!\circ\, l = f \rangle]\!]$.

Nous commençons par définir notre interprétation des sortes, qui est très simple. Ainsi, nous codons la sorte des types par son équivalent dans $\mathtt{BF}_{\leqslant}$ qui est $\mathbb{T}$, et nous codons les sortes des rangées par la sorte des enregistrements $\mathbb{R}$. Sans surprise, la sorte des interfaces est $\mathbb{I} = (\mathbb{T} \to \mathbb{R})$.

Définition 20.4 (Codage des sortes de $\boldsymbol{\lambda\mathcal{O}}$bj)

$$[\![\mathbf{T}]\!] = \mathbb{T} \qquad [\![\mathbf{M}]\!] = \mathbb{R} \qquad [\![\mathbf{T} \to \mathbf{M}]\!] = \mathbb{T} \to \mathbb{R}$$

Nous définissons aussi une nouvelle notation pour les sortes, soit $\mathbb{J} = (\mathbb{T} \to \mathbb{R}) \to \mathbb{R}$, c'est-à-dire la sorte des opérateurs qui prennent une interface en argument, et qui renvoient une rangée.

Avant de montrer comment nous simulons les règles de typage des prototypes, nous définissons deux notations pour des opérateurs de types de $\mathtt{BF}_{\leqslant}$, et nous donnons quelques inférences de type valides dans ce système.

Définition 20.5 (Opérateur de types pour les prototypes) Nous définissons un opérateur de types utile pour typer **Proto** (cf. équation (20.1)), et donc le codage des prototypes.

$$\mathsf{pro} \ =_{\text{def}}\ \mu p^{\mathbb{J}}.\Lambda s^{\mathbb{I}}.\mu o^{\mathbb{R}}.[\, \mathit{inht} : (\forall s' \sqsubseteq s\ .\ (ps' \to s(ps'))),\ \mathit{invk} : so\,]$$

la sorte de pro est la sorte de la variable récursive p, c'est-à-dire $(\mathbb{T} \to \mathbb{R}) \to \mathbb{R}$. Si ϱ est une rangée (et donc à la sorte $\mathbb{R}$), on dit que $(\mathsf{pro}\,(\Lambda t^{\mathbb{T}}.\varrho))$ est un *type prototype*. Ce type sera l'interprétation du type prototype de $\boldsymbol{\lambda\mathcal{O}}$**bj**, aussi nous emploierons abusivement la notation $(\mathbf{pro}\, t.\varrho)$ pour désigner $(\mathsf{pro}\,(\Lambda t^{\mathbb{T}}.\varrho))$. Nous définissons également un opérateur utile pour donner le type du champ *inht* de pro.

$$\mathsf{pre} \ =_{\text{def}}\ \Lambda t^{\mathbb{I}}.(\forall s' \sqsubseteq t\ .\ (\mathsf{pro}\, s' \to t(\mathsf{pro}\, s')))$$

la sorte de pre est $(\mathbb{I} \to \mathbb{T}) = (\mathbb{T} \to \mathbb{R}) \to \mathbb{T}$. Si ϱ est une rangée, on note $(\mathbf{pre}\, t.\varrho)$ le type $(\mathsf{pre}\,(\Lambda t^{\mathbb{T}}.\varrho))$.

Il est facile de montrer une relation entre types similaire à l'équation (20.2). Soit σ un type quelconque, on a:

$$(\mathsf{pro}\,\sigma) \ \sim\ [\, \mathit{inht} : (\mathsf{pre}\,\sigma),\ \mathit{invk} : \sigma\,(\mathsf{pro}\,\sigma)\,] \tag{20.5}$$

Preuve (la relation (20.5) est valide) En utilisant la μ-conversion, on montre facilement que $\mathsf{pro} \sim \Lambda s^{\mathbb{I}}.\mu o^{\mathbb{R}}.[\, \mathit{inht} : (\forall s' \sqsubseteq s\ .\ (\mathsf{pro}\, s' \to s(\mathsf{pro}\, s'))),\ \mathit{invk} : so\,]$. De plus on montre que $(\mathsf{pre}\,\sigma) \sim (\forall s' \sqsubseteq \sigma\ .\ (\mathsf{pro}\, s' \to \sigma(\mathsf{pro}\, s')))$. Par conséquent:

$$\begin{array}{rcl}
(\mathsf{pro}\,\sigma) & \sim & \mu o^{\mathbb{R}}.[\, \mathit{inht} : (\forall s' \sqsubseteq \sigma\ .\ (\mathsf{pro}\, s' \to \sigma(\mathsf{pro}\, s'))),\ \mathit{invk} : \sigma o\,] \\
& \sim & \mu o^{\mathbb{R}}.[\, \mathit{inht} : (\mathsf{pre}\,\sigma),\ \mathit{invk} : \sigma o\,] \\
& \sim & [\, \mathit{inht} : (\mathsf{pre}\,\sigma),\ \mathit{invk} : \sigma(\mathsf{pro}\,\sigma)\,]
\end{array}$$

□

ce qui nous permet de montrer que la règle suivante est admissible dans notre système:

$$\frac{\Gamma \vdash G : (\mathsf{pre}\,\sigma)}{\Gamma \vdash (\mathbf{Proto}\ G) : (\mathsf{pro}\,\sigma)}\ \text{(type proto)} \tag{20.6}$$

Preuve (l'inférence (20.6) est valide) Soit Δ l'environnement tel que x soit de type $(\mathsf{pre}\,\sigma)$, et s soit de type $(\mathsf{pre}\,\sigma) \to (\mathsf{pro}\,\sigma)$. Alors $\Delta \vdash (sx) : (\mathsf{pro}\,\sigma)$, et:

$$\frac{\Delta \vdash x : (\mathsf{pre}\,\sigma) \sim (\forall s' \sqsubseteq \sigma\,.\,(\mathsf{pro}\,s' \to \sigma(\mathsf{pro}\,s'))) \qquad \Delta \vdash \sigma \sqsubseteq \sigma}{\Delta \vdash x : (\mathsf{pro}\,\sigma \to \sigma(\mathsf{pro}\,\sigma))}\ \text{(type inst)}$$

Par conséquent $\Delta \vdash x(sx) : \sigma(\mathsf{pro}\,\sigma)$, ce qui implique que l'enregistrement $\mathbf{P}_{(s,x)}$ est de type:

$$\Delta \vdash \mathbf{P}_{(s,x)} : [\,\mathit{inht} = (\mathsf{pre}\,\sigma), \mathit{invk} = \sigma(\mathsf{pro}\,\sigma)\,] \sim (\mathsf{pro}\,\sigma) \qquad \text{cf. équation (20.5)}$$

En utilisant les règles (type abs) et (type rec), on montre alors que $\emptyset \vdash \mathbf{Proto} : (\mathsf{pre}\,\sigma) \to (\mathsf{pro}\,\sigma)$. Par conséquent, si $\Gamma \vdash G : (\mathsf{pre}\,\sigma)$, alors $\Gamma \vdash (\mathbf{Proto}\ G) : (\mathsf{pro}\,\sigma)$. □

Un cas où l'utilisation de la règle dérivée (type pro) est utile, est lorsque G est un générateur d'objets:

$$G\ =_{\text{def}}\ (\boldsymbol{\lambda} x)[\,l_1 = (M_1\ x), \ldots, l_n = (M_n\ x)\,]$$

tel que G peut être typé par le type $(\mathbf{pre}\,t.\varrho)$ (c'est-à-dire par $(\mathsf{pre}\,(\Lambda t^{\mathbb{T}}.\varrho))$), où ϱ est la rangée $[\,l_i : \tau_i{}^{i\in[1..n]}\,]$, en utilisant l'hypothèse que le paramètre self x a le type $(\mathsf{pro}\,s')$, sous la contrainte $s' \sqsubseteq (\Lambda t^{\mathbb{T}}.\varrho)$, et tel que M_i a le type $(t \to \tau_i)\{\mathsf{pro}\,s'/t\}$. Ainsi, on obtient une règle de typage moins générale que (20.6) qui est:

$$\frac{\Gamma, s' \sqsubseteq (\Lambda t^{\mathbb{T}}.\varrho), x : (\mathsf{pro}\,s') \vdash [\,l_i = (M_i\ x)^{i\in[1..n]}\,] : [\,l_i : \tau_i\{\mathsf{pro}\,s'/t\}^{i\in[1..n]}\,]}{\Gamma \vdash (\mathbf{Proto}\ G) : (\mathbf{pro}\,t.\varrho)}$$

Soit encore:

$$\frac{\Gamma, s' \sqsubseteq (\Lambda t^{\mathbb{T}}.\varrho) \vdash M_i : (t \to \tau_i)\{\mathsf{pro}\,s'/t\} \qquad \forall i \in [1..n]}{\Gamma \vdash (\mathbf{Proto}\ (\boldsymbol{\lambda} x)[\,l_i = (M_i\ x)^{i\in[1..n]}\,]) : (\mathbf{pro}\,t.[\,l_i : \tau_i{}^{i\in[1..n]}\,])}\ \text{(type proto)} \tag{20.7}$$

Cette dernière règle montre que le codage des prototypes est de type $(\mathbf{pro}\,t.\varrho)$. En particulier, la règle (type proto) implique que le prototype vide a le type $(\mathbf{pro}\,t.[\,])$:

$$\overline{\Gamma \vdash [\![\langle\rangle]\!] : (\mathbf{pro}\,t.[\,])}$$

Nous montrons maintenant comment typer l'invocation, l'extension et la modification de méthodes.

Invocation. En utilisant la règle (type proto) et l'équation (20.5), on montre que $(\mathbf{Proto}\ G)$ a le type suivant. Soit σ l'interface $(\Lambda t^{\mathbb{T}}.\varrho)$, on a:

$$(\mathbf{pro}\,t.\varrho) = (\mathsf{pro}\,\sigma) \sim [\,\mathit{inht} : (\forall s' \sqsubseteq \sigma\,.\,(\mathsf{pro}\,s' \to \varrho\{\mathsf{pro}\,s'/t\})),\ \mathit{invk} : \varrho\{\mathsf{pro}\,\sigma/t\}\,]$$

On remarque que le champ *invk* du codage d'un prototype, a pour type la rangée $[\,l_i : \tau_i\{\mathsf{pro}\,\sigma/t\}^{i\in[1..n]}\,]$. Si on sélectionne ce champ, ce qui correspond à l'invocation de méthode, on a donc uniquement accès aux méthodes de G. Ainsi, en utilisant la loi (20.5) et la règle (type sel), on montre une règle de typage admissible pour l'invocation:

$$\frac{\Gamma \vdash M : (\mathsf{pro}\,\sigma) \qquad \Gamma \vdash \sigma \sqsubseteq \Lambda t^{\mathbb{T}}.[\,l : \tau\,]}{\Gamma \vdash (M \cdot \mathit{invk} \cdot l) : \tau\{\mathsf{pro}\,\sigma/t\}}\ \text{(type invk)} \tag{20.8}$$

Ce qui, dans le cas spécial où σ est l'interface $\Lambda t^{\mathbb{T}}.\varrho$, donne une règle similaire à la règle (pro invk) de la figure 20.5:

$$\frac{\Gamma \vdash M : (\mathbf{pro}\, t.\varrho) \qquad \Gamma, t :: \mathbb{T} \vdash \varrho \sqsubseteq [\, l : \tau \,]}{\Gamma \vdash M \Leftarrow l : \tau\{(\mathbf{pro}\, t.\varrho)/t\}} \text{ (type proto invk)}$$

Le type du champ *inht*, par contre, est le type d'une fonction qui accepte des arguments de type $(\mathsf{pro}\,\sigma')$, pour tout σ' tel que $\sigma' \sqsubseteq \sigma$, c'est-à-dire pour toute extension possible du prototype, et qui retourne une rangée «spécialisée» à cette extension. On utilise fonction dans (le typage de) l'extension et de la modification.

En utilisant la loi (20.5), on montre que l'inférence suivante est valide. Soit G le générateur $G = (\boldsymbol{\lambda} x)[\, M \cdot \mathit{inht}\ x \,, l = N\ x \,]$, alors:

$$\frac{\Gamma \vdash M : (\mathsf{pro}\,\sigma) \qquad \Gamma \vdash \sigma' \sqsubseteq \sigma \qquad \Gamma \vdash N : (\forall s \sqsubseteq \sigma' \,.\, (\mathsf{pro}\, s \to \tau\{\mathsf{pro}\, s/t\}))}{\Gamma \vdash G : (\forall s \sqsubseteq \sigma' \,.\, (\mathsf{pro}\, s \to [\, \sigma(\mathsf{pro}\, s) \,, l : \tau\{\mathsf{pro}\, s/t\} \,]))} \tag{20.9}$$

Par conséquent, si on cherche à appliquer **Proto** au générateur G, il faut typer G avec le type $(\mathsf{pre}\,\sigma)$. Une solution est d'utiliser la règle (type sub) pour sous-typer le type de G, ce qui impose de trouver deux types σ et σ' qui vérifient l'équation:

$$(\forall s \sqsubseteq \sigma' \,.\, (\mathsf{pro}\, s \to [\, \sigma(\mathsf{pro}\, s) \,, l : \tau\{\mathsf{pro}\, s/t\} \,])) \leqslant (\forall s \sqsubseteq \sigma' \,.\, (\mathsf{pro}\, s \to \sigma'(\mathsf{pro}\, s)))$$

Soit plus simplement: $[\, \sigma(\mathsf{pro}\, s) \,, l : \tau\{\mathsf{pro}\, s/t\} \,] \leqslant (\sigma'(\mathsf{pro}\, s))$. De plus il faut que les solutions vérifient les prémisses de la loi (20.9), une deuxième contrainte est donc que $\Gamma \vdash \sigma' \sqsubseteq \sigma$. Nous étudions deux solutions possible à ces équations.

Extension. Une première solution consiste à choisir $\sigma' = \Lambda t^{\mathbb{T}}.[\, (\sigma\ t) \,, l : \tau \,]$. Cette solution correspond à l'extension de méthode. Dans ce cas, on montre facilement que $\sigma' \sqsubseteq \sigma$ et que $(\sigma'(\mathsf{pro}\, s)) \sim [\, \sigma(\mathsf{pro}\, s) \,, l : \tau\{\mathsf{pro}\, s/t\} \,]$. Par conséquent on peut remplacer σ' par sa définition dans l'inférence de l'équation (20.9), et utiliser la loi (20.6) pour obtenir une règle admissible pour l'extension:

$$\frac{\Gamma \vdash M : (\mathsf{pro}\,\sigma) \qquad \Gamma, t :: \mathbb{T} \vdash [\, \sigma \,, l : \tau \,] \sqsubseteq \sigma \qquad \Gamma \vdash N : (\forall s \sqsubseteq \Lambda t^{\mathbb{T}}.[\, (\sigma\ t) \,, l : \tau \,] \,.\, (\mathsf{pro}\, s \to \tau\{\mathsf{pro}\, s/t\}))}{\Gamma \vdash (\mathbf{Proto}\ G) : \mathsf{pro}\,(\Lambda t^{\mathbb{T}}.[\, (\sigma\ t) \,, l : \tau \,])}$$

Si σ est l'interface $\Lambda t^{\mathbb{T}}.\varrho$, cette inférence se réécrit plus simplement sous la forme:

$$\frac{\begin{array}{c}\Gamma \vdash M : (\mathbf{pro}\, t.\varrho) \qquad \Gamma, t :: \mathbb{T} \vdash [\, \varrho \,, l : \tau \,] \sqsubseteq \varrho \\ \Gamma, r \sqsubseteq \Lambda t^{\mathbb{T}}.[\, \varrho \,, l : \tau \,] \vdash N : (t \to \tau)\{\mathsf{pro}\, r/t\}\end{array}}{\Gamma \vdash \langle\!\langle M \Leftarrow l = N \rangle\!\rangle : (\mathbf{pro}\, t.[\, \varrho \,, l : \tau \,])} \text{ (type proto ext)}$$

Modification. Une seconde solution, moins évidente, consiste à choisir $\sigma' = \sigma$ avec l'hypothèse que $\sigma \sqsubseteq \Lambda t^{\mathbb{T}}.[\, l : \tau \,]$. En effet, dans ce cas, on peut utiliser la règle (wsub updt) de la figure 14.2 pour montrer que:

$$\frac{\dfrac{\dfrac{\sigma \sqsubseteq \Lambda t^{\mathbb{T}}.[\, l : \tau \,]}{\sigma(\mathsf{pro}\, s) \sqsubseteq (\Lambda t^{\mathbb{T}}.[\, l : \tau \,])\ (\mathsf{pro}\, s)}}{\sigma(\mathsf{pro}\, s) \sqsubseteq [\, l : \tau\{(\mathsf{pro}\, s)/t\} \,]}}{[\, \sigma(\mathsf{pro}\, s) \,, l : \tau\{\mathsf{pro}\, s/t\} \,] \sqsubseteq \sigma(\mathsf{pro}\, s)}$$

En appliquant le même raisonnement que dans le cas précédent, on obtient alors une règle admissible pour la modification:

$$\frac{\Gamma \vdash M : (\mathsf{pro}\,\sigma) \qquad \Gamma \vdash \sigma \sqsubseteq \Lambda t^{\mathbb{T}}.[\, l : \tau \,] \qquad \Gamma \vdash N : (\forall s \sqsubseteq \sigma \,.\, (\mathsf{pro}\, s \to \tau\{\mathsf{pro}\, s/t\}))}{\Gamma \vdash (\mathbf{Proto}\ G) : \mathsf{pro}\,\sigma}$$

Encore une fois, si σ est l'interface $\Lambda t^{\mathbb{T}}.\varrho$, cette inférence se réécrit plus simplement sous la forme:

$$\frac{\Gamma \vdash M : (\mathbf{pro}\, t.\varrho) \qquad \Gamma, t :: \mathbb{T} \vdash \varrho \sqsubseteq [\, l : \tau \,] \qquad \Gamma, r \sqsubseteq \Lambda t^{\mathbb{T}}.\varrho \vdash N : (t \to \tau)\{\mathsf{pro}\, r/t\}}{\Gamma \vdash \langle\!\langle M \leftarrow l = N \rangle\!\rangle : (\mathbf{pro}\, t.\varrho)} \text{ (type proto over)}$$

On peut aussi prouver la validité de la règle (pro select). Nous omettons la preuve ici, mais celle-ci utilise encore une fois la loi (20.5). Ainsi, on montre que l'inférence suivante est valide dans notre système:

$$\frac{\Gamma \vdash M : (\mathbf{pro}\, t.\varrho_1) \qquad \Gamma \vdash N : (\mathbf{pro}\, t.\varrho_2) \qquad \Gamma, t :: \mathbb{T} \vdash \varrho_2 \sqsubseteq \varrho_1 \sqsubseteq [\, l : \tau \,]}{\Gamma \vdash (M \cdot inht\ N\ \cdot l) : \tau\{(\mathbf{pro}\, t.\varrho_2)/t\}} \text{ (type proto select)}$$

Interprétation des types. Au regard des règles de typage dérivées que nous donnons dans cette section – c'est-à-dire les règles apparaissant dans un encadré –, l'interprétation des types de $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ doit maintenant être claire. D'autant plus que nous avons décidé de noter $(\mathbf{pro}\, t.\varrho)$ le type $(\mathsf{pro}\, (\Lambda t^{\mathbb{T}}.\varrho))$.

Définition 20.6 (Codage des types de $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$) La fonction de codage des rangées dans $\mathtt{BF}_{\leqslant}$ est l'homomorphisme tel que $[\![\lambda t.\varrho]\!] = \Lambda t^{\mathbb{T}}.[\![\varrho]\!]$. Le codage des types fonctionnels est lui aussi trivial: $[\![t]\!] = t$ et $[\![(\tau \to \sigma)]\!] = ([\![\tau]\!] \to [\![\sigma]\!])$, tandis que les types prototypes sont codés en utilisant l'opérateur pro: $[\![(\mathbf{pro}\, t.\varrho)]\!] = (\mathbf{pro}\, t.[\![\varrho]\!])$.

L'interprétation des environnements de typage est lui aussi trivial, en particulier on code la contrainte $(r <:_w \varrho)$, par la contrainte $(r \sqsubseteq [\![\varrho]\!])$. Il est facile de montrer, avec notre codage, que le système $\mathtt{BF}_{\leqslant}$ permet de simuler le système des figures 20.3 et 20.4.

Lemme 20.3 *Si* $\Gamma \vdash *$*, alors* $[\![\Gamma]\!] \vdash *$*. Si* $\Gamma \vdash \tau :: \mathbf{T}$*, alors* $[\![\Gamma]\!] \vdash [\![\tau]\!] :: \mathbb{T}$*. Si* $\Gamma \vdash \varrho :: \mathbf{M}$*, alors* $[\![\Gamma]\!] \vdash [\![\varrho]\!] :: \mathbb{R}$*.*

Nous omettons la preuve de ce résultat ici: elle est assez simple, car chaque règle a un équivalent dans $\mathtt{BF}_{\leqslant}$, néanmoins elle est longue et sans grand intérêt. Nous montrons que notre interprétation conserve le typage des termes.

Théorème 20.4 (Simulation du typage) *Si le jugement* $\Gamma \vdash e : \tau$ *peut être inféré dans* $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$*, alors* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$ *peut être inféré dans* $\mathtt{BF}_{\leqslant}$*.*

Preuve La preuve est par induction sur l'inférence de $\Gamma \vdash e : \tau$. On montre tout d'abord que le jugement $(\sharp)$ $\Gamma \vdash \varrho :: \{l\}$, implique que pour tout type τ on a: $[\![\Gamma]\!] \vdash [\, [\![\varrho]\!] , l : \tau \,] \sqsubseteq [\![\varrho]\!]$. La preuve de ce résultat est simple puisque $(\sharp)$ implique que le champ l n'est pas dans $[\![\varrho]\!]$. On prouve donc que la règle (pro ext) est «simulée» par la règle (type proto ext). Finalement, comme nous avons un équivalent à chacune des règles de la figure 20.5, par exemple (type proto invk) pour (pro invk), on peut simuler chaque preuve de typage dans $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$, par une preuve dans $\mathtt{BF}_{\leqslant}$ qui utilise le même arbre d'inférence. Ce qui implique le résultat. □

Un des intérêt de notre codage est de faire apparaître les règles distinctes de typage de l'extension et de la modification dans le système de [49], comme deux instances différentes d'une même inférence (20.9). Un deuxième intérêt de notre codage, est que notre interprétation permet de reformuler le système de types d'une manière plus simple, à l'aide de la relation de matching.

Nous définissons la relation de matching <#, par la loi suivante.

$$\frac{\Gamma,t :: \mathbb{T} \vdash \varrho_1 \sqsubseteq \varrho_2}{\Gamma \vdash (\mathbf{pro}\, t.\varrho_1) <\# (\mathbf{pro}\, t.\varrho_2)} \tag{20.10}$$

Cette relation est exactement le préordre introduit par K. Bruce dans [25], où il avance des arguments convaincants pour montrer que le matching est la relation de sous-typage nécessitée pour le typage de l'extension et de la modification des prototypes (bien que [25] traite plutôt des classes). On peut utiliser le matching pour reformuler les règles (type proto invk), (type proto ext) et (type proto over). Par exemple la première de ces règles peut se réécrire:

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma <\# (\mathbf{pro}\, t.[\, l : \tau\,])}{\Gamma \vdash M \Leftarrow l : \tau\{\sigma/t\}}$$

On obtient alors un système plus simple, qui est similaire à celui proposé récemment par V. Bono et M. Bugliesi [13] pour typer $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$.

20.5 Sous-typage pour les types objets

Les auteurs de [47] remarquent que l'ajout du sous-typage à un langage de prototypes ne permet pas de typer plus de programmes. L'invariance du type des prototypes apparaît clairement dans notre interprétation. En effet dans le type $(\mathsf{pre}\,\sigma)$, le type σ apparaît en position contravariante: en tant que borne dans le type $(\forall t \sqsubseteq \sigma\,.\,(\ldots))$. Le type σ est donc en position contravariante dans le champ *inht* du type prototype $(\mathsf{pro}\,\sigma)$. Or il apparaît aussi en position covariante dans le champ *invk* de ce même type. L'opérateur pro est donc invariant.

Afin d'ajouter du «polymorphisme» – au sens des type objets – dans leur calcul, J. Mitchell et K. Fisher on enrichi $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ en distinguant, au niveau des types, une catégorie de termes appelés objets, et qui se distinguent des prototypes par le fait qu'ils ne peuvent être ni étendus, ni modifiés [48]. La présentation de ce système utilise une relation de sous-typage, dénotée <:, et un nouveau constructeur de types pour les objets, noté $(\mathbf{obj}\, t.\varrho)$.

La définition de la relation <: est similaire à celle de notre relation de sous-typage générale $\leqslant$, à la différence qu'il existe aussi, dans $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$, des règles primitives pour les opérateurs objets. Ces règles sont essentiellement celles données ci-dessous:

$$\frac{\Gamma,t <: t' \vdash \varrho <: \varrho'}{\Gamma \vdash (\mathbf{pro}\, t.\varrho) <: (\mathbf{obj}\, \tau'.\varrho')}\ \text{(pro sub)} \qquad \frac{\Gamma,t <: t' \vdash \varrho <: \varrho'}{\Gamma \vdash (\mathbf{obj}\, t.\varrho) <: (\mathbf{obj}\, \tau'.\varrho')}\ \text{(obj sub)}$$

On trouve également une règle qui autorise l'invocation de méthode sur les objets.

$$\frac{\Gamma \vdash e : (\mathbf{obj}\, t.\varrho) \qquad \Gamma,t :: \mathbf{T} \vdash \varrho <:_w [\, l : \tau\,]}{\Gamma \vdash e \Leftarrow l : \tau\{(\mathbf{obj}\, t.\varrho)/t\}}\ \text{(obj invk)}$$

Puisque la différence entre un prototype et un objet est que ce dernier ne peut pas être modifié, il est raisonnable de représenter le type objet par une rangée ayant le champ *invk*, mais pas le champ *inht*.

Définition 20.7 (Codage du type objet)

$$[\![(\mathbf{obj}\, t.\varrho)]\!] = \mathsf{obj}\,(\Lambda t^{\mathbb{T}}.[\![\varrho]\!]) \qquad \text{avec} \qquad \mathsf{obj} =_{\text{def}} \Lambda s^{\mathbb{I}}.\mu o^{\mathbb{R}}.[\, invk : so\,]$$

On remarque que ce que nous avons défini en suivant notre intuition, est essentiellement ce que K. Bruce *et al.* qualifie de «*classical recursive record encoding*» dans [24], en effet il est simple de montrer que $(\mathbf{obj}\, t.\varrho) \sim \mu t^{\mathbb{R}}.[\, invk : \varrho\,]$.

Comme dans la section précédente, nous montrons que nous «simulons» le sous-typage des objets, en démontrant la validité de certaines inférence dans notre système. Ainsi, si on suppose que $\Gamma \vdash \sigma \leqslant \sigma'$, on peut montrer comment la règle (pro sub) est simulée dans notre système.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma,t \leqslant t' \vdash \varrho \leqslant \varrho'}{\Gamma,t \leqslant t' \vdash (\Lambda t^{\mathbb{T}}.\varrho)t \leqslant (\Lambda {t'}^{\mathbb{T}}.\varrho')t'}}{\Gamma,t \leqslant t' \vdash [\, invk : (\Lambda t^{\mathbb{T}}.\varrho)t\,] \leqslant [\, invk : (\Lambda {t'}^{\mathbb{T}}.\varrho')t'\,]}}{\Gamma,t \leqslant t' \vdash [\, inht : (\mathbf{pre}\, t.\varrho), invk : (\Lambda t^{\mathbb{T}}.\varrho)t\,] \leqslant [\, invk : (\Lambda {t'}^{\mathbb{T}}.\varrho')t'\,]}}{\Gamma \vdash \mu t^{\mathbb{T}}.[\, inht : (\mathbf{pre}\, t.\varrho), invk : (\Lambda t^{\mathbb{T}}.\varrho)t\,] \leqslant \mu {t'}^{\mathbb{T}}.[\, invk : (\Lambda {t'}^{\mathbb{T}}.\varrho')t'\,]}$$

Or une conséquence de l'équation (20.5) est que $(\mathbf{pro}\, t.\varrho) \sim \mu o^{\mathbb{T}}.[\, inht : (\mathbf{pre}\, t.\varrho),\ invk : (\Lambda t^{\mathbb{T}}.\varrho)o\,]$. Par conséquent on a démontré que:

$$\dfrac{\Gamma,t \leqslant t' \vdash \varrho \leqslant \varrho'}{\Gamma \vdash (\mathbf{pro}\, t.\varrho) \leqslant (\mathbf{obj}\, t'.\varrho')}\ \text{(type proto sub)}$$

Ce qui est le résultat attendu. De même, on montre un résultat similaire pour la règle (obj sub), c'est-à-dire que l'inférence suivante est valide dans $\mathtt{BF}_{\leqslant}$:

$$\dfrac{\Gamma,t \leqslant t' \vdash \varrho \leqslant \varrho'}{\Gamma \vdash (\mathbf{obj}\, t.\varrho) \leqslant (\mathbf{obj}\, t'.\varrho')}\ \text{(type obj sub)}$$

Comme le champ *inht* est absent du type $(\mathbf{obj}\, t.\varrho)$, il est impossible de typer une extension ou une modification d'un terme qui a ce type. Par contre il est simple de vérifier qu'on peut typer la sélection d'une manière similaire à la règle (obj invk):

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash M : (\mathsf{obj}\, \sigma)}{\Gamma \vdash M \cdot invk : \sigma(\mathsf{obj}\, \sigma)} \quad \dfrac{\sigma \sqsubseteq \Lambda t^{\mathbb{T}}.[\, l : \tau\,]}{\sigma(\mathsf{obj}\, \sigma) \sqsubseteq [\, l : \tau\{\mathsf{obj}\, \sigma/t\}\,]}}{\Gamma \vdash M \cdot invk : [\, l : \tau\{\mathsf{obj}\, \sigma/t\}\,]}}{\Gamma \vdash M \cdot invk \cdot l : \tau\{\mathsf{obj}\, \sigma/t\}}$$

Ainsi, dans le cas particulier où σ est l'interface $\Lambda t^{\mathbb{T}}.\varrho$, on obtient la règle de typage dérivée pour l'invocation des objets.

$$\dfrac{\Gamma \vdash M : (\mathbf{obj}\, t.\varrho) \qquad \Gamma,t :: \mathbb{T} \vdash \varrho \sqsubseteq [\, l : \tau\,]}{\Gamma \vdash M \Leftarrow l : \tau\{(\mathbf{obj}\, t.\varrho)/t\}}\ \text{(type obj invk)}$$

Nous avons exhibé dans $\mathtt{BF}_{\leqslant}$, un équivalent pour chaque règles de typage et de sous-typage de $\boldsymbol{\lambda\mathcal{O}\mathbf{bj}}_{<:}$, l'extension de $\boldsymbol{\lambda\mathcal{O}\mathbf{bj}}$ au sous-typage. Ceci est suffisant pour prouver la troisième propriété de préservation de notre interprétation des objets extensibles, c'est-à-dire que le sous-typage est préservé par $[\![.]\!]$.

Théorème 20.5 (Simulation du sous-typage) *Si le jugement* $\Gamma \vdash e : \tau$ *peut être inféré dans* $\boldsymbol{\lambda\mathcal{O}\mathbf{bj}}_{<:}$, *alors* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$ *peut être inféré dans* $\mathtt{BF}_{\leqslant}$.

Discussion

Liens avec d'autres travaux

Plusieurs études théoriques traitent de la modélisation des langages de programmation orientés objets dans des langages procéduraux [62, 3]. Mais seuls les codages les plus récents ont réussi à préserver les notions de typage et de sous-typage, qui sont très importantes. Cet échec des précédentes tentatives est d'ailleurs à l'origine de la définition des calculs d'objets, c'est-à-dire des calcul dans lesquels les objets sont des constructions primitives.

Dans [2], les auteurs proposent une interprétation compositionelle du calcul $\mathbf{Ob}_{1<:}$ dans $\mathtt{F}_{\leq\mu}$, le λ-calcul polymorphe du second ordre avec types récursifs et sous-typage. Ce résultat est amélioré dans [137], où l'auteur donne une interprétation dans un λ-calcul simplement typé, étendu avec des références et des enregistrements non-extensibles.

Dans cette partie, nous avons proposé une interprétation en utilisant le calcul bleu et un système de types du premier ordre. De plus nous avons défini, en nous inspirant de ce codage, un calcul d'objets concurrents qui apparaît aussi expressif que le calcul de A. Gordon et P. Hankin [61], et donc aussi expressif que le calcul d'objets impératifs.

Nous avons également étudié le calcul des prototypes ($\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$) de K. Fisher *et al.* [48, 49]. À notre connaissance, nous donnons dans cette partie la première interprétation de ce calcul qui préserve la notion de typage et de sous-typage. En nous basant sur le codage présenté dans cette thèse, nous avons donné une interprétation de $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ dans le λ-calcul [17]. Très récemment, V. Bono et M. Bugliesi ont publié un article sur le même sujet [14], et ont proposé une autre interprétation de $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$. Ayant eu connaissance de leurs travaux trop tardivement, nous ne pouvons pas fournir une comparaison formelle de nos deux approches dans cette section.

Le codage de $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ défini dans ces pages a été donné dans $\boldsymbol{\lambda\mathcal{D}}$ef, qui peut s'interpréter comme le sous-calcul fonctionnel de $\boldsymbol{\pi}^\star$. Par conséquent, de la même façon que le calcul du chapitre 17 est obtenu à partir du codage de $\boldsymbol{\varsigma}$, nous pouvons définir un calcul de prototypes impératifs et concurrents. Plus précisément, nous pouvons coder un nouveau type de dénominations, $\langle e \mapsto\!\!\!\!\!\rightarrow L\rangle$, qui vérifient les lois de la figure 20.7.

Les interprétations que nous avons données dans cette partie ont aussi pour intérêt de définir des calcul d'objets concurrents basés sur les références et le nommage des objets, alors que les calculs de [4, 42, 48] utilisent la substitution de code. Nous nous rapprochons en cela des calculs impératifs comme $\mathbf{imp}\boldsymbol{\varsigma}$ [3] et $\mathbf{conc}\boldsymbol{\varsigma}$ [60]. Ainsi, nous proposons un modèle dans lequel le code d'un objet est partagé au lieu d'être dupliqué, ce qui est plus proche de l'implantation des langages à objets.

Soit L le corps $(l_i = (\boldsymbol{\lambda} x_i) P_i^{\,i\in[1..n]})$.

$$\frac{j \in [1..n]}{\langle e \mapsto L\rangle \mid e \Leftarrow l_j \to_\varsigma \langle e \mapsto L\rangle \mid P_j\{e/x_j\}} \text{ (red invk)}$$

$$\frac{f \notin \mathbf{fn}(L)}{\langle e \mapsto L\rangle \mid \mathbf{clone}(e) \to_\varsigma \langle e \mapsto L\rangle \mid (\boldsymbol{\nu} f)(\langle f \mapsto L\rangle \mid \mathbf{return}(f))} \text{ (red clone)}$$

$$\frac{j \in [1..n] \qquad L' =_{\text{def}} [\, L\,, l_j = (\boldsymbol{\lambda} x) P\,]}{\langle e \mapsto L\rangle \mid (e \leftarrow l_j = (\boldsymbol{\lambda} x) P) \to_\varsigma \langle e \mapsto L'\rangle \mid \mathbf{return}(e)} \text{ (red updt)}$$

$$\frac{j \in [1..n] \qquad f \notin \mathbf{fn}(L) \qquad L' =_{\text{def}} [\, L\,, l = (\boldsymbol{\lambda} x) P\,]}{\langle e \mapsto L\rangle \mid (e \leftarrow\!\!+ l = (\boldsymbol{\lambda} x) P) \to_\varsigma \langle e \mapsto L\rangle \mid (\boldsymbol{\nu} f)(\langle f \mapsto L'\rangle \mid \mathbf{return}(f))} \text{ (red ext)}$$

Fig. 20.7: Prototypes concurrents

En ce qui concerne les langages à objets concurrents, on peut noter les travaux de C. Jones [73] et D. Walker [139], qui ont utilisé le π-calcul pour coder POOL, un langage parallèle orienté objets, et pour prouver la validité de certaines transformations. Mais POOL est un langage non typé et très restreint. Dans [118], D. Sangiorgi donne la première interprétation de $\mathbf{Ob}_{1<:}$ dans le π-calcul, et dans [76], les auteurs s'intéressent au calcul impératif. Notre interprétation se distingue de ces codages dans le π-calcul. Ainsi le codage et le système de type utilisé ici sont très différents. En particulier, dans notre codage, une modification de méthode ne crée pas un nouvel objet, mais est plus naturellement modélisée comme la modification d'un enregistrement. De plus, la preuve de correction opérationnelle du codage ne repose pas sur l'utilisation du système de type: dans [118], les résultats sont prouvés à l'aide d'une bisimulation typée; et notre codage de $\mathbf{Ob}_{1<:}$ est «direct», c'est-à-dire ne fait pas appel à des transformations du type *continuation passing style*.

Futurs développements

Un des résultats de cette partie est la définition d'un calcul d'objets impératifs et concurrents dérivé du calcul bleu. En particulier, ce calcul possède toute l'expressivité de $\boldsymbol{\pi}^\star$, ce qui permet de coder des «stratégies de synchronisation» très complexes entre les objets.

Les calculs de processus ont déjà été utilisés pour raisonner sur les 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 $\boldsymbol{\pi}^\star$ était la base idéale pour la conception d'un langage de programmation concurrent de haut-niveau, fortement typé et possédant les caractéristiques des langages à objets. Ce but a été atteint grâce à une interprétation des objets comme un cas spécial de processus. Dans ce sens, nous suivons la trace de B. Pierce et D. Turner sur le π-calcul asynchrone, qui étaient motivés par la conception du langage Pict [130].

Le codage des calculs d'objets $\boldsymbol{\varsigma}$ et $\boldsymbol{\lambda\mathcal{O}}\mathbf{bj}$ a été un test réussi qui a démontré l'expressivité de $\boldsymbol{\pi}^\star$. En continuant sur cette voie, il serait intéressant d'essayer de modéliser un langage distribué orienté objets tel que Obliq [29], ce qui demanderait de rajouter une notion de localité à notre calcul. On pourrait aussi étudier la modélisation des *Object Request Brokers*, comme Corba. En effet les types enregistrements que nous utilisons pour typer les processus de $\boldsymbol{\pi}^\star$, sont une réminiscence des langages de définition d'interface utilisés dans les ORBs [104].

Un autre développement envisageable est d'autoriser la restriction sur les noms de méthodes. Il suffit pour cela de considérer les étiquettes comme des noms du calcul bleu à part entière.

Nous conjecturons que la possibilité de restreindre la portée des noms, en utilisant l'opérateur ($\boldsymbol{\nu}$), permettrait de coder les mécanismes de contrôle de la visibilité des méthodes des langages orientés objets, tel que les annotations *private*, *protected*, ... utilisées dans C++ par exemple.

CHAPITRE 21

Conclusion

Arrivé au terme de cette thèse, il apparaît que le calcul bleu constitue une base adéquate à la définition d'un langage concurrent d'ordre supérieur ayant des traits fonctionnels. En effet, nous avons montré les correspondances qui existent – aussi bien au niveau de l'évaluation que du typage – entre les modèles de programmation définis par le λ-calcul et le calcul bleu. Cette correspondance est d'autant plus flagrante que nous avons utilisé des résultats prouvés dans cette thèse pour résoudre un problème du λ-calcul qui était ouvert. Plus précisément, nous avons utilisé les résultats du chapitre 20, consacré à l'interprétation des objets extensibles dans $\boldsymbol{\lambda}\mathcal{D}$ef– le sous-calcul déterministe de $\boldsymbol{\pi}^\star$ défini à la section 3.4 –, pour donner un codage du calcul de K. Fisher *et al.* préservant les notions de typage et de sous-typage [17], dans le λ-calcul.

Intuitivement, le modèle de programmation défini par le calcul bleu peut être interprété comme une version concurrente du langage Haskell [127, 70], qui est un langage de programmation paresseux – de la famille de ML– possédant certains opérateurs d'ordre supérieur dans son système de types. D'ailleurs, il faut noter qu'il existe une proposition pour intégrer à Haskell, un système d'enregistrements extensibles [57] qui peuvent se comparer aux enregistrements définis dans $\boldsymbol{\pi}^\star$. Ainsi, on peut considérer que notre étude du calcul bleu constitue une base pour la définition d'un langage de programmation concurrent «à la Haskell», de la même manière que le calcul Join sert de base théorique à la définition du langage JoCaml [85, 84], qui est une extension concurrente (et distribuée) du langage Caml.

Le calcul bleu possède également toutes les caractéristiques du π-calcul. Il comporte donc aussi des traits associés aux langages impératifs et aux langages à acteurs. Comme nous l'avons montré dans la dernière partie de cette thèse, il est possible d'étendre la syntaxe de $\boldsymbol{\pi}^\star$ avec des opérateurs concurrents empruntés aux calculs d'objets, et de les typer correctement. De même, on peut ajouter au calcul bleu des opérateurs caractéristiques des langages impératifs – les (cellules) références et la composition séquentielle –, ainsi que les opérateurs des langages fonctionnels avec appel par valeurs – les opérateurs dérivés **set** et **return** utilisés dans le codage de **concς**. On peut donc imaginer qu'un langage de programmation basé sur le calcul bleu possède des objets impératifs et concurrents, de la même manière que les calculs CAP [32] et TyCO [132, 136] sont utilisés comme fondement pour des – propositions de – langages à agents et à objets concurrents.

Pour résumer, on peut dire que le modèle de programmation induit par le calcul bleu offre le choix entre plusieurs méthodes pour composer des programmes – ou des systèmes – entre eux. Méthodes qui sont simples et théoriquement fondées. Il fourni notamment l'application et les fonctions d'ordre supérieur, comme avec Haskell [71]. Il fourni aussi la composition parallèle, la

communication et les objets, comme dans les langages «à composants».

Pour conclure, nous présentons plusieurs prolongements possibles aux travaux menés au cours de cette thèse. Suivant la symétrie du plan de notre thèse, nous donnons la liste de ces travaux restant à faire dans l'ordre des parties.

Une suite naturelle de notre travail est d'ajouter, au calcul bleu, des primitives pour la distribution. C'est, par exemple, la démarche suivie dans [51, 55] pour le JOIN calcul, et dans [123, 66] pour le π-calcul. Ceci permettrais de définir un langage distribué avec, potentiellement, de la mobilité de code. Toutefois, l'ajout de primitives pour la distribution – et la mobilité de code – à de nombreuses répercussions. Ainsi l'ajout de la distribution ne peut être entrepris sans une réflexion préalable sur le modèle de sécurité qu'on désire fournir aux programmeurs. En effet, les expériences d'ajout de mobilité à des langages déjà existant ont montré qu'il est illusoire de vouloir construire un langage distribué (et sûr), sans avoir pris en compte dès le début les problèmes de sécurité. L'ajout de la distribution nécessite également de définir un modèle pour traiter du problème des pannes [8, 7]. Un autre travail intéressant, qu'il reste à faire, est d'étudier les méthodes de compilation adaptés à un langage construit à partir de $\boldsymbol{\pi}^\star$. On peut, par exemple, chercher à définir un bon modèle de machine abstraite. Deux possibilités s'offrent à nous, choisir une machine abstraite utilisant des piles – comme dans PICT par exemple [105] –, ou bien choisir une méthode de compilation basée sur les continuations.

En ce qui concerne notre travail sur les systèmes de types, une suite normale est de s'attaquer au problème de l'inférence de types. Il est clair que ce problème n'est pas décidable pour le système $\mathtt{BF}_\leqslant$. Par contre, on peut espérer emprunter des méthodes utilisées dans les langages fonctionnels, pour résoudre ce problème dans le système avec sous-typage, et dans le système avec polymorphisme paramétrique, récursion et enregistrements. En effet, le seul résultat – sur l'inférence – que nous possédons, est l'existence d'un algorithme d'inférence de types pour le système avec polymorphisme «restreint aux définitions» [36], et dans un calcul sans enregistrements.

Nous avons parlé des prolongements de notre travail sur les objets concurrents à la fin du chapitre précédent. Ainsi, un premier développement envisageable est d'autoriser la restriction sur les noms de méthodes. Il suffit pour cela d'étendre l'opérateur $(\boldsymbol{\nu})$ aux noms d'étiquettes. On peut alors se demander si il est possible de coder les mécanismes de contrôle de la visibilité des langages orientés objets, tel que les annotations *private*, *protected*, ... qu'on rencontre dans le langage C++ par exemple. Un autre problème intéressant concerne l'étude des interactions entre «stratégie» de synchronisation et héritage – ou l'extension de méthodes si on suit l'approche des langages à délégation –, un problème connu sous le terme de *inheritance anomaly* [88].

Bibliographie

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, et Jean-Jacques Lévy. « Explicit Substitutions ». *Journal of Functional Programming*, 1(4):375–416, octobre 1991. Also appeared as SRC Research Report 54.

[2] Martín Abadi, Luca Cardelli, et Ramesh Viswanathan. « An Interpretation of Objects and Object Types ». Dans *Proc. of POPL '96 – 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 396–409, 1996.

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

[4] Martín Abadi et Luca Cardelli. « A Theory of Primitive Objects: Untyped and First-Order Systems ». *Information and Computation*, 2(125):78–102, mars 1996.

[5] Martìn Abadi. « Secrecy by Typing in Security Protocols ». Dans *Proc. of TACS '7 – Theoretical Aspects of Computer Software*, pages 611–638. Springer-Verlag, 1997. To appear in Journal of the ACM.

[6] Roberto M. Amadio, Ilaria Castellani, et Davide Sangiorgi. « On Bisimulations for the Asynchronous pi-calculus ». *Theoretical Computer Science*, 195(2):291–324, 1998.

[7] Roberto Amadio et Sanjiva Prasad. « Localities and Failures ». Extended version from FST & TCS '94, 1995.

[8] Roberto Amadio. « An Asynchronous Model of Locality, Failure and Process Mobility ». Dans *COORDINATION '97*, volume 1282 de *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[9] S. Arun-Kumar et Matthew Hennessy. « An Efficiency Preorder for Processes ». *Acta Informatica*, 29(8):737–760, 1992.

[10] Hamid Azzoune. « *Les types en Prolog. Un système d'inférence de type et ses applications* ». Thèse d'état, Institut National Polytechnique de Grenoble, 1984.

[11] Gérard Berry et Gérard Boudol. « The Chemical Abstract Machine ». *Theoretical Computer Science*, 96:217–248, 1992.

[12] Gérard Berry, Pierre-Louis Curien, et Jean-Jacques Lévy. Full Abstraction for Sequential Languages: State of the Art. Dans J. Reynolds eds M. Nivat, éditeur, *Algebraic methods in semantics*, pages 89–132. Cambridge University Press, 1985.

[13] Viviana Bono et Michele Bugliesi. « Matching Constraints for the Lambda Calculus of Objects ». Dans *Proc. of TLCA '97*, volume 1210 de *Lecture Notes in Computer Science*, pages 46–62. Springer-Verlag, 1997.

[14] Viviana Bono et Michele Bugliesi. « Interpretations of Extensible Objects and Types ». Dans *Proc. of FCT '99 – 12th International Symposium on Fundamentals of Computation Theory*, août 1999. (to appear).

[15] Michele Boreale, Cédric Fournet, et Cosimo Laneve. « Bisimulations in the Join-Calculus ». Dans D. Gries et W.P. de Roever, éditeurs, *Proc. of PROCOMET '98 – Programming Concepts and Methods*, pages 68–86. Chapman & Hall, juin 1998.

[16] Michele Boreale. « On the Expressiveness of Internal Mobility in Name-Passing Calculi ». *Theoretical Computer Science*, 195(2):205–226, 1998.

[17] Gérard Boudol et Silvano Dal-Zilio. « An Interpretation of Extensible Objects ». Dans *Proc. of FCT '99 – 12th International Symposium on Fundamentals of Computation Theory*, août 1999.

[18] Gérard Boudol et Cosimo Laneve. « The Discriminating Power of Multiplicities in the λ-Calculus ». Rapport technique 2441, INRIA, décembre 1994.

[19] Gérard Boudol. « The Lambda-Calculus with Multiplicities ». Rapport technique 2025, INRIA, septembre 1993.

[20] Gérard Boudol. « Some Chemical Abstract Machines ». Dans J. W. de Bakker, W.-P. de Roever, et G. Rozenberg, éditeurs, *A Decade of Concurrency - Reflections and Perspectives*, volume 803 de *Lecture Notes in Computer Science*, pages 92–123. Springer-Verlag, 1994.

[21] Gérard Boudol. « Typing the Use of Resources in a Concurrent Calculus ». Dans *Proc. of ASIAN '97 – Asian Computing Science Conference*, Lecture Notes in Computer Science, Kathmandu, décembre 1997. Springer-Verlag.

[22] Gérard Boudol. « The π-Calculus in Direct Style ». *Higher-Order and Symbolic Computation*, 11:177–208, 1998. Also appeared in Proc. of POPL '97, p. 228–241, January 1997.

[23] Gérard Boudol. « The Blue Calculus Homepage ». Web page. Available as `http://www.inria.fr/meije/personnel/Gerard.Boudol/blue.html`.

[24] Kim Bruce, Luca Cardelli, et Benjamin Pierce. « Comparing Object Encodings ». Dans *Proc. of TACS '97*, volume 1281 de *Lecture Notes in Computer Science*, pages 415–438. Springer-Verlag, 1997.

[25] Kim Bruce. « The Equivalence of Two Semantic Definitions for Inheritance in Object-Oriented Languages ». Dans *Proc. of MFPS '92*, volume 598 de *Lecture Notes in Computer Science*, pages 102–124. Springer-Verlag, 1992.

[26] Peter Canning, William Cook, Walter Hill, John Mitchell, et Walter Olthoff. « F-Bounded Polymorphism for Object-Oriented Programming ». Dans *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[27] Luca Cardelli et John C. Mitchell. « Operations on Records ». *Math. Structures in Computer Science*, 1(1):3–48, 1991.

[28] Luca Cardelli et Peter Wegner. « On Understanding Types, Data Abstraction, and Polymorphism ». *Computing Surveys*, 17(4):471–522, décembre 1985.

[29] Luca Cardelli. « A Language with Distributed Scope ». *Computer Systems*, 8:27–59, 1995.

[30] Felice Cardone et Mario Coppo. « Two Extensions of Curry's Type Inference System ». Dans P. Odifreddi, éditeur, *Logic and Computer Science*. Academic Press, 1990.

[31] Dominique Clément, Joelle Despeyroux, Thierry Despeyroux, et Gilles Kahn. « A Simple Applicative Language: Mini-ML ». Dans *Proc. of the ACM Conference on Lisp and Functionnal Programming*. ACM, 1986.

[32] Jean-Louis Colaço. « *Analyse statique d'un calcul d'Acteurs par typage* ». Thèse d'état, Institut National Polytechnique, Toulouse, octobre 1997.

[33] Adriana B. Compagnoni. « *Higher-Order Subtyping with Intersection Types* ». PhD thesis, University of Nijmegen, The Netherlands, janvier 1995. ISBN 90-9007860-6.

[34] William Cook, Walter Hill, et Peter Canning. « Inheritance is Not Subtyping ». Dans *Proc. of POPL '90 – 17th Annual ACM Symposium on Principles of Programming Languages*, janvier 1990.

[35] Pierre-Louis Curien et Giorgio Ghelli. « Coherence of Subsumption, Minimum Typing and the Type Checking in $F_{\leq}$ ». *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

[36] Silvano Dal-Zilio. « Implicit Polymorphic Type System for the Blue Calculus ». Rapport technique 3244, INRIA, septembre 1997.

[37] Silvano Dal-Zilio. « Quiet and Bouncing Objects: Two Migration Abstractions in a Simple Distributed Blue Calculus ». Dans Hans Hüttel et Uwe Nestmann, éditeurs, *Proc. of SOAP '98 – 1st International Workshop on Semantics of Objects as Processes*, volume NS-98-5 de *BRICS Notes Series*, pages 35–42. BRICS, juillet 1998.

[38] Silvano Dal-Zilio. « Objets concurrents dans un π-calcul applicatif ». Dans *Proc. of JFLA '99 – 10ème Journées Francophones des Langages Applicatifs*, février 1999. An english version is available at `http://www.inria.fr/meije/personnel/Silvano.Dal_Zilio/`.

[39] Luís Damas et Robin Milner. « Principal Type Schemes for Functional Programming ». Dans *Proc. of POPL '82 – 9th Annual ACM Symposium on Principles of Programming Languages*, 1982.

[40] Luís Damas. « *Type Assignment in Programming Languages* ». PhD thesis, University of Edinburgh, 1984.

[41] Nicolaas Govert de Bruijn. « Lambda-Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem ». *Indag. Math.*, 34(5):381–392, 1972.

[42] Paolo Di Blasio et Kathleen Fisher. « A Calculus for Concurrent Objects ». Dans *Proc. of CONCUR '96 – 7th International Conference on Concurrency Theory*, volume 1119 de *Lecture Notes in Computer Science*. Springer-Verlag, août 1996.

[43] Pietro Di Gianantonio, Furio Honsell, et Luigi Liquori. « A Lambda Calculus of Objects with Self-Inflicted Extension ». Dans *Proc. of OOPSLA '98 – ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, volume 33(10) de *SIGPLAN Notices*, octobre 1998.

[44] Jonathan Eifrig, Scott Smith, et Valery Trifonov. « Type Inference for Recursively Constrained Types and its Application to OOP ». Dans *Proc. Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, 1995.

[45] William Ferreira, Matthew Hennessy, et Alan Jeffrey. Combining Typed λ-Calculus with CCS. Dans Gordon Plotkin, Colin Stirling, et Mads Tofte, éditeurs, *Essays in Honour of Robin Milner*. MIT Press, 1998.

[46] Kathleen Fisher, Furio Honsell, et John C. Mitchell. « A Lambda Calculus of Objects and Method Specialization ». *Nordic Journal of Computing*, 1(1):3–37, 1994.

[47] Kathleen Fisher et John C. Mitchell. « Notes on Typed Object-Oriented Programming ». Dans *proc. of TACS '94*, volume 789 de *Lecture Notes in Computer Science*, pages 844–885. Springer-Verlag, 1994.

[48] Kathleen Fisher et John C. Mitchell. « A Delegation-Based Object Calculus with Subtyping ». Dans *proc. of FCT '95*, volume 965 de *Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag, 1995.

[49] Kathleen Fisher. « *Type Systems for Object-Oriented Programming Languages* ». PhD thesis, Standford University, août 1996.

[50] Kathleen Fisher. « Personal Communication », novembre 1998.

[51] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, et Didier Rémy. « A Calculus of Mobile Agents ». Dans *Proc. of CONCUR '96 – 7th International Conference on Concurrency Theory*, volume 1119 de *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, août 1996. Springer-Verlag.

[52] Cédric Fournet et Georges Gonthier. « The Reflexive Chemical Abstract Machine and the Join-Calculus ». Dans *Proc. of POPL '96 – 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 372–385, janvier 1996.

[53] Cédric Fournet, Luc Maranget, Cosimo Laneve, et Didier Rémy. « Implicit Typing à la ML for the Join-Calculus ». Dans *Proc. of CONCUR '97 – 8th International Conference on Concurrency Theory*, Warsaw, Poland, 1997.

[54] Cedric Fournet et Luc Maranget. « The Join-Calculus Language Documentation and User's Guide ». Available at `http://pauillac.inria.fr/join/`, 1997.

[55] Cédric Fournet. « *Le join-calcul : un calcul pour la programmation répartie et mobile* ». PhD thesis, École Polytechnique, novembre 1998.

[56] Benedict R. Gaster et Mark P. Jones. « A Polymorphic Type System for Extensible Records and Variants ». Rapport technique NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, novembre 1996.

[57] Benedict R. Gaster. « Polymorphic Extensible Records for Haskell ». Dans *Haskell Workshop*. ACM Press, juin 1997.

[58] Simon J. Gay. « A Sort Inference Algorithm for the Polyadic Pi-Calculus ». Dans *Proc. of POPL '93 – 20th Annual ACM Symposium on Principles of Programming Languages*, 1993.

[59] Jean-Yves Girard. « *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur* ». Thèse d'état, Université Paris VII, 1972.

[60] Andrew D. Gordon et Paul D. Hankin. « A Concurrent Object Calculus ». Dans *Proc. of HLCL '98 – 3rd International Workshop on High-Level Concurrent Languages*, Elsevier ENTCS, 1998.

[61] Andrew D. Gordon et Paul D. Hankin. « A Concurrent Object Calculus: Reduction and Typing ». Rapport technique 4XX, University of Cambridge Computer Laboratory, février 1999. Extended version of [60].

[62] Carl A. Gunter et John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[63] Paul Hankin. « Personal Communication », février 1999.

[64] Robert Harper et Mark Lillibridge. « Polymorphic Type Assignment and CPS Conversion ». *LISP and Symbolic Computation*, 6:361–380, 1993.

[65] Fritz Henglein. « Type Inference with Polymorphic Recursion ». *ACM Transactions on Programming Languages and Systems*, 15(254–289), avril 1993.

[66] Matthew Hennessy et James Riely. « A typed language for distributed mobile processes ». Dans *Proc. of POPL '98 – 25th Annual ACM Symposium on Principles of Programming Languages*, pages 378–390, janvier 1998.

[67] Daniel Hirschkoff. « *Mise en oeuvre de preuves de bisimulation* ». Thèse d'état, École Nationale des Ponts et Chaussées, 1999.

[68] Kohei Honda et Nobuko Yoshida. « On Reduction-Based Process Semantics ». *Theoretical Computer Science*, 152(2):437–486, 1995.

[69] Kohei Honda. « A Theory of Types for π-Calculus ». Available at `http://www.dcs.qmw.ac.uk/~kohei/`, novembre 1998.

[70] Paul Hudak, John Peterson, et Joseph H. Fasel. « A Gentle Introduction to Haskell, Version 1.4 ». Available at `http://haskell.systemsz.cs.yale.edu/tutorial/`, mars 1997.

[71] John Hughes. « Why Functional Programming Matters ». *The Computer Journal*, 32(2):98–107, 1989. Also in: David A. Turner (ed.): Research Topics in Functional Programming, Addison-Wesley, 1990, pp. 17–42.

[72] Trevor Jim. « What Are Principal Typings and What Are They Good For? ». Dans *Proc. of POPL '96 – 23rd Annual ACM Symposium on Principles of Programming Languages*, 1996.

[73] Cliff B. Jones. « A π-calculus Semantics for an Object-Based Design Notation ». Dans E. Best, éditeur, *Proc. of CONCUR '93 – 4th International Conference on Concurrency Theory*, volume 715 de *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.

[74] Assaf J. Kfoury, Jerzy Tiuryn, et Pawel Urzyczyn. « The Undecidability of the Semi-Unification Problem ». Dans *Proc. of the 22nd annual ACM symposium on theory of Computation (STOC)*, pages 468–476, New-York, 1990. ACM.

[75] Assaf J. Kfoury, Jerzy Tiuryn, et Pawel Urzyczyn. « Type Reconstruction in the Presence of Polymorphic Recursion ». *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, avril 1993.

[76] Josva KLEIST et Davide SANGIORGI. « Imperative Objects and Mobile Processes ». Dans *Proc. of PROCOMET '98 – Working Conference on Programming Concepts and Methods*. North-Holland, 1998.

[77] Jean-Louis KRIVINE. *Lambda-calcul : types et modèles*. Masson, 1990.

[78] John LAUNCHBURY. « A Natural Semantics for Lazy Evaluation ». Dans *Proc. of POPL '93 – 20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, S.C., janvier 1993.

[79] Carolina LAVATELLI. « *Sémantique du lambda calcul avec ressources* ». Thèse d'état, Université Paris VII, janvier 1996.

[80] Xavier LEROY. « Polymorphic Typing of an Algorithmic Language ». Rapport technique, INRIA, INRIA-Rocquencourt, Le Chesnay, France, 1992. English version of [81].

[81] Xavier LEROY. « *Typage polymorphe d'un langage algorithmique* ». Thèse d'état, Université Paris 7, 1992.

[82] Xavier LEROY. « Polymorphism by Name for References and Continuations ». Dans *Proc. of POPL '93 – 20th Annual ACM Symposium on Principles of Programming Languages*, pages 220–231, 1993.

[83] Xavier LEROY. « The Objective Caml system ». `http://caml.inria.fr/ocaml/`, 1996. Software and documentation available on the web.

[84] Fabrice LE FESSANT. « The JoCaml system documentation ». `http://pauillac.inria.fr/jocaml/`, 1999. Software and documentation available on the web.

[85] Fabrice LE FESSANT. « JoCaml : programmation distribuée et agents mobiles ». Dans *Proc. of CFSE '1 – 1ère Conférence Française sur les Systèmes d'Exploitation*, juin 1999.

[86] Camille LE MONIÈS DE SAGAZAN. « *Un système de types étiquetés polymorphes pour typer les calculs de processus à liaison noms-canaux dynamiques* ». Thèse d'état, Université Paul Sabatier, Toulouse, novembre 1995. Also CNRS/LAAS report 95077.

[87] Harry G. MAIRSON. « Decidability of ML Typing is Complete for Deterministic Exponential Time ». Dans *Proc. of POPL '90 – 17th Annual ACM Symposium on Principles of Programming Languages*, janvier 1990.

[88] Satoshi MATSUOKA et Akinori YONEZAWA. Analysis of inheritance anomaly in object-oriented concurrent programming languages. Dans Gul AGHA, Peter WEGNER, et Akinori YONEZAWA, éditeurs, *Research Directions in Concurrent Object-Oriented Programming*, Chapitre 4, pages 107–150. The MIT Press, 1993.

[89] Massimo MERRO et Davide SANGIORGI. « On Asynchrony in Name-Passing Calculi ». Dans *Proc. of ICALP '98*, volume 1443 de *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[90] Bertrand MEYER. *Object-Oriented Software Construction*. Prentice Hall, 2nd édition, 1997.

[91] Robin MILNER, Joachim PARROW, et David WALKER. « A Calculus of Mobile Processes, Parts I and II ». *Journal of Information and Computation*, 100:1–77, 1992.

[92] Robin MILNER et Davide SANGIORGI. « Barbed Bisimulation ». Dans W. KUICH, éditeur, *19th ICALP*, volume 623 de *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[93] Robin MILNER, Mads TOFTE, et Robert HARPER. *The Definition of standard ML*. MIT Press, 1990.

[94] Robin MILNER. « A Theory of Type Polymorphism in Programming ». *Journal of Computer and System Sciences*, 17:348–375, août 1978.

[95] Robin MILNER. *Communication and Concurrency*. C.A.R. Hoare, 1989.

[96] Robin MILNER. Semantics of Concurrent Processes. Dans Jan VAN LEEUWEN, éditeur, *Handbook of theoretical computer science*, Chapitre 19, pages 1203–1241. Elsevier, 1990.

[97] Robin MILNER. « The Polyadic π-calculus: a Tutorial ». Rapport technique ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991. Reprinted in Logic and Algebra of Specification, F. Bauer, W.Brauer and H. Schwichtenberg, Eds, Springer Verlag, 1993, 204–246.

[98] Robin Milner. « Functions as Processes ». *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[99] John C. Mitchell. « Coercion and Type Inference ». Dans *Proc. of POPL '84 – 11th Annual ACM Symposium on Principles of Programming Languages*, janvier 1984.

[100] John Mitchell. « Toward a Typed Foundation for Method Specialization and Inheritance ». Dans *Proc. of POPL '90 – 17th Annual ACM Symposium on Principles of Programming Languages*, pages 109–124, janvier 1990.

[101] Alan Mycroft et Richard A. O'Keefe. « A Polymorphic Type System for Prolog ». *Artificial Intelligence*, 23:295–307, 1984.

[102] Alan Mycroft. « Polymorphic Type Scheme and Recursive Definitions ». Dans *6th International Symposium on programming*, volume 167 de *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.

[103] Alan Mycroft. « Incremental Polymorphic Type Checking with Update ». Dans *Proc. of LFCS '92*, volume 620 de *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[104] Elie Najm et Jean-Bernard Stefani. « A formal semantics for the ODP computational model ». Rapport technique PAA/3527, CNET, mai 1993.

[105] Benjamin C. Pierce et David N. Turner. « Pict: A Programming Language Based on the Pi-Calculus ». Rapport technique 476, Indiana University CSCI, mars 1997.

[106] Benjamin Pierce et Martin Hofmann. « A Unifying Type-Theoretic Framework for Objects ». *Journal of Functional Programming*, 5(4):593–635, 1995.

[107] Benjamin Pierce et Davide Sangiorgi. « Typing and Subtyping for Mobile Processes ». *Mathematical Structures in Computer Science*, 11, 1995.

[108] Gordon Plotkin. « Call-by-name, Call-by-value and the λ-calculus ». *Theoretical Computer Science*, 1:125–159, 1975.

[109] François Pottier. « *Synthèse de types en présence de sous-typage: de la théorie à la pratique* ». Thèse d'état, Université Paris VII, juillet 1998.

[110] Didier Rémy et Jérôme Vouillon. « Objective ML: An effective object-oriented extension to ML ». *Theory And Practice of Object Systems*, 4(1):27–50, 1998. Also appeared in Proc. of POPL '97, p. 40–53, January 1997.

[111] Didier Rémy. Type Inference for Records in a Natural Extension of ML. Dans Carl A. Gunter et John C. Mitchell, éditeurs, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993. Also appeared as INRIA Research Report 1431.

[112] Didier Rémy. « Refined Subtyping and Row Variables for Record Types ». Available at `http://cristal.inria.fr/~remy/publications.html`.

[113] Davide Sangiorgi et Robin Milner. « Techniques of "weak bisimulation up-to" ». Dans *Proc. of CONCUR '92 – 3rd International Conference on Concurrency Theory*, volume 630 de *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.

[114] Davide Sangiorgi. « *Expressing Mobility in Process Algebra: First-Order and Higher-Order Paradigms* ». PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1993.

[115] Davide Sangiorgi. « From π-Calculus to Higher-Order π-Calculus – and Back ». Dans *Proc. of TAPSOFT '93*, volume 668 de *Lecture Notes in Computer Science*, pages 151–166, 1993.

[116] Davide Sangiorgi. « The Lazy Lambda Calculus in a Concurrency Scenario ». *Information and Computation*, 111(1), 1994.

[117] Davide Sangiorgi. « Bisimulation for Higher-Order Process Calculi ». *Information and Computation*, 131(2):141–178, 1996.

[118] Davide Sangiorgi. « An Interpretation of Typed Objects into Typed π-Calculus ». Rapport technique 3000, INRIA, 1996.

[119] Davide Sangiorgi. « Locality and Non-interleaving Semantics in Calculi for Mobile Processes ». *Theoretical Computer Science*, 155:39–83, 1996.

[120] Davide Sangiorgi. « The Name Discipline of Receptiveness ». Dans *Proc. of ICALP '97*, volume 1256 de *Lecture Notes in Computer Science*. Springer-Verlag, 1997. To appear in TCS.

[121] Davide Sangiorgi. « Interpreting Functions as π-Calculus Processes: a Tutorial ». Rapport technique 3470, INRIA, août 1998.

[122] Davide Sangiorgi. « On the Bisimulation Proof Method ». *Mathematical Structures in Computer Science*, 8(5):447–479, octobre 1998.

[123] Peter Sewell. « Global/Local Subtyping and Capability Inference for a Distributed π-calculus ». Dans *Proc. of ICALP '98 – International Colloquium on Automata, Languages and Programming*, volume 1443 de *Lecture Notes in Computer Science*, pages 695–706. Springer-Verlag, juillet 1998.

[124] James W. Stamos et David K. Gifford. « Remote Evaluation ». *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, octobre 1990.

[125] Leon Sterling et Ehud Y. Shapiro. *The Art of PROLOG : advanced programming techniques*. MIT press series in logic programming. AAAI press, 1986. ISBN 0-262-69105-1.

[126] Eijiro Sumii et Naoki Kobayashi. « A Generalized Deadlock-Free Process Calculus ». Dans *Proc. of HLCL '98 – High-Level Concurrent Languages*, volume 16.3 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[127] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999. Second Edition. ISBN 0-201-34275-8.

[128] Mads Tofte. « *Operational Semantics and Polymorphic Type Inference* ». PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, mai 1988.

[129] Mads Tofte. « Type Inference for Polymorphic References ». *Information and Computation*, 89:1–34, 1990.

[130] David N. Turner. « *The Polymorphic Pi-Calculus: Theory and Implementation* ». PhD thesis, University of Edinburgh, 1995.

[131] Pawel Urzyczyn. « Polymorphic Type Checking and Related Problems », 1996. (lectures notes).

[132] Vasco T. Vasconcelos et Rui Bastos. « Core-TyCO,the language definition, version 0.1 ». Rapport technique DI/FCUL TR 98-3, University of Lisbon, mars 1998.

[133] Vasco T. Vasconcelos et Kohei Honda. « Principal Typing-Schemes in a Polyadic Pi-Calculus ». Dans *4th International conference on concurrency theory*, volume 715, pages 524–538. Springer-Verlag, 1993.

[134] Vasco T. Vasconcelos et António Ravara. « Communication Errors in the π-Calculus Are Undecidable ». *Information Processing Letters*, 1999.

[135] Vasco T. Vasconcelos et Mario Tokoro. « A Typing System for a Calculus of Objects ». Dans *1st International Symposium on Object Technologies for Advanced Software*, volume 742 de *Lecture Notes in Computer Science*, pages 460–474. Springer-Verlag, novembre 1993.

[136] Vasco T. Vasconcelos. « TYped Concurrent Objects ». Web page. Available as `http://www.di.fc.ul.pt/~vv/tyco.html`.

[137] Ramesh Viswanathan. « Full Abstraction for First-Order Objects with Recursive Types and Subtyping ». Dans *Proc. of LICS '98 – 13th Symposium on Logic in Computer Science*, pages 380–391, 1998.

[138] Dennis Volpano, Geoffrey Smith, et Cynthia Irvine. « A Sound Type System for Secure Flow Analysis ». *Journal of Computer Security*, 4(3):167–187, décembre 1996.

[139] David Walker. « π-Calculus Semantics of Object-Oriented Programming Languages ». Dans *Proc. of TACS '91*, volume 526 de *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991.

[140] Mitchell Wand. « Complete Type Inference for Simple Objects ». Dans *Proc. of LICS '87 – 2nd Symposium on Logic in Computer Science*, pages 37–44, 1987. A corrigendum to the article appeared in LICS '88.

Liste des figures

Glossaire

symbole	description	première utilisation
$\{^{u}/_{x}\}$	substitution nom / variable	section 2.1
$\{\!\{^{M}/_{x}\}\!\}$	substitution terme / variable	section 2.1
$=_{\alpha}$	α-équivalence	section 2.1
$\mathbf{fn}(M)$,$\mathbf{bn}(M)$	noms libres et noms liés de M	section 2.1
$\mathbf{decl}(D)$	noms déclarés par D	définition 2.3 (p. 13)
$\mathbf{dom}(\Gamma)$,$\mathbf{dv}(\Gamma)$	noms déclarés et noms définis par Γ	définition 11.3 (p. 83) et 13.3 (p. 96)

calculs:

$\boldsymbol{\pi}^{\star}$	calcul bleu	figure 2.1 (p. 13)
$\boldsymbol{\pi}$	mini π-calcul asynchrone	figure 3.1 (p. 22)
$\boldsymbol{\Lambda}$	λ-calcul	section 3.3
$\boldsymbol{\Lambda}^{\star}$	λ-calcul avec passage de noms	figure 3.3 (p. 25)
$\boldsymbol{\lambda}\mathcal{D}\mathrm{ef}$	λ-calcul avec définitions et enregistrements	figure 3.5 (p. 26)
$\boldsymbol{\varsigma}$	calcul d'objets de M. Abadi et L. Cardelli	chapitre 16
$\mathbf{Ob}_{1<:}$	ς-calcul avec types simples et sous-typage	figure 16.1 (p. 132) et 16.2 (p. 133)
$\mathbf{conc}\boldsymbol{\varsigma}$	ς-calcul concurrent et impératif	figure 19.1 (p. 156)
$\boldsymbol{\lambda}\mathcal{O}\mathbf{bj}$	calcul des prototypes de K. Fisher *et al.*	figure 20.1 (p. 162)
$\boldsymbol{\lambda}\mathcal{O}\mathbf{bj}_{<:}$	calcul des prototypes avec sous-typage	section 20.5

réductions:

$\equiv$	équivalence structurelle	figure 2.2 (p. 15)
$\equiv$	équivalence structurelle dans $\mathbf{conc}\boldsymbol{\varsigma}$	figure 19.2 (p. 156)
$\rightarrow$	réduction	figure 2.3 (p. 17)
$\rightarrow_{(\rho)}$	réduction limitée à la communication	section 2.2.2
$\rightarrow_{(\beta)}$	réduction limitée à la sélection et à la béta-réduction	section 2.2.2
$\overset{\mu}{\longmapsto}$	transition étiquetée	section 5.1
$\rightsquigarrow$	réduction dans $\boldsymbol{\varsigma}$	figure 16.1 (p. 132)
$\rightsquigarrow$	réduction dans $\mathbf{conc}\boldsymbol{\varsigma}$	figure 19.3 (p. 158)
$\rightsquigarrow$	réduction dans $\boldsymbol{\lambda}\mathcal{O}\mathbf{bj}$	figure 20.2 (p. 162)

interprétations:

$M^\star$	codage de $\mathbf{\Lambda}$ dans $\mathbf{\Lambda}^\star$	définition 3.3 (p. 25)
$[\![P]\!]$	codage de $\boldsymbol{\pi}$ dans $\boldsymbol{\pi}^\star$	définition 3.1 (p. 22)
$[\![M]\!]$	codage de $\mathbf{\Lambda}^\star$ dans $\boldsymbol{\pi}^\star$	définition 3.4 (p. 26)
$[\![M]\!]$	codage de $\mathbf{\Lambda}$ dans $\boldsymbol{\pi}^\star$	définition 8.1 (p. 66)
$[\![\delta]\!]$	codage des sortes de $\boldsymbol{\pi}$ dans $\mathtt{B}$	définition 11.5 (p. 86)
$[\![e]\!]$	codage de $\mathbf{Ob}_{1<:}$ dans $\boldsymbol{\pi}^\star$	définition 18.1 (p. 147) et 18.2 (p. 147)
$[\![e]\!]$	codage de $\mathbf{concs}$ dans $\boldsymbol{\pi}^\star$	définition 19.3 (p. 158)
$[\![e]\!]$	codage de $\boldsymbol{\lambda\mathcal{O}bj}$ dans $\boldsymbol{\lambda\mathcal{D}}$ef	définition 20.2 (p. 164)
$(\!\|M\|\!)$	codage de $\mathbf{\Lambda}$ dans $\boldsymbol{\pi}^\star$	définition 8.1 (p. 66)
$(\!\|M\|\!)$	codage de mini-ML dans $\boldsymbol{\pi}^\star$	définition 13.6 (p. 99)
$(\!\|e\|\!)$	*self-application semantics*	définition 20.1 (p. 163)

équivalences:

$\approx_{\mathcal{D}}$	conversion dans $\boldsymbol{\lambda\mathcal{D}}$ef	section 3.4
$P\downarrow$	barbe	définition 4.1 (p. 37)
$P\Downarrow$	barbe faible	définition 4.1 (p. 37)
$\approx_b$	congruence à barbes	définition 4.3 (p. 37)
$\sim_d$	def-bisimulation forte	définition 5.2 (p. 44)
$\approx_d$	def-bisimulation faible	définition 5.2 (p. 44)
$\mathcal{D}_{\mathrm{C}}$	relation $\mathcal{D}$ modulo contextes	définition 6.1 (p. 47)
$\mathcal{D}_{\equiv}$	relation $\mathcal{D}$ modulo équivalence structurelle	définition 6.2 (p. 47)
$\mathcal{D}_R$	relation $\mathcal{D}$ modulo réplication	définition 6.3 (p. 48)
$\mathcal{D}_\infty$	relation $\mathcal{D}$ modulo F	définition 6.4 (p. 49)
$\lesssim_d$	expansion	définition 10.1 (p. 75)
$\gtrsim_d$	contraction	définition 10.1 (p. 75)

systèmes de types:

$\mathtt{B}$	système de types simples de $\boldsymbol{\pi}^\star$ avec récursion	figure 11.2 (p. 84)
$\mathtt{M}$	système de sortes de $\boldsymbol{\pi}$	figure 11.3 (p. 86)
$\mathtt{B}_\leqslant$	système de types simples de $\boldsymbol{\pi}^\star$ avec sous-typage	chapitre 12
$\mathtt{B}_\forall$	système de types de $\boldsymbol{\pi}^\star$ avec polymorphisme paramétrique	figure 13.1 (p. 97)
$\mathtt{DM}$	système de types de mini-ML	figure 13.2 (p. 98)
$\mathtt{MM}$	$\mathtt{DM}$ avec récursion polymorphe	définition 13.4 (p. 97)
$\mathtt{B}_{\mathtt{MM}}$	système de types de $\boldsymbol{\pi}^\star$ avec polymorphisme paramétrique et récursion polymorphe	définition 13.5 (p. 98)
$\mathtt{B}_{\mathrm{ML}}$	système de types de $\boldsymbol{\pi}^\star$ avec polymorphisme restreint aux définitions	figure 13.4 (p. 102)
$\mathtt{BF}_\leqslant$	système de types de $\boldsymbol{\pi}^\star$ avec polymorphisme contraint	figure 14.4 (p. 112)
$\prec$	substitutivité (ou subsumption)	définition 13.3 (p. 96)
$\mathrm{Gen}_\Gamma(\tau)$	clôture de τ dans Γ	définition 13.3 (p. 96)
$\leqslant$	sous-typage	figure 12.1 (p. 90) et 14.3 (p. 111)
$\sqsubseteq$	sous-typage en largeur	figure 14.2 (p. 111)
$\sim$	égalité entre types	définition 14.4 (p. 110)
$<:$	sous-typage de $\mathbf{Ob}_{1<:}$	définition 16.2 (p. 133)
$<:$	sous-typage de $\boldsymbol{\lambda\mathcal{O}bj}$	section 20.5
$<:_w$	sous-typage en largeur $\boldsymbol{\lambda\mathcal{O}bj}$	figure 20.4 (p. 167)
$<\#$	matching	section 20.4

jugements:

$\Gamma \vdash *$	l'environnement Γ est bien formé	figure 14.1 (p. 109)
$\Gamma \vdash \tau :: k$	le type τ à la sorte k	figure 14.1 (p. 109)
$\Gamma \vdash \tau \sqsubseteq \sigma$	le type τ est un sous-type en largeur de σ	figure 14.2 (p. 111)
$\Gamma \vdash \tau \leqslant \sigma$	le type τ est un sous-type de σ	figure 14.3 (p. 111)

catégories syntaxiques:

$P,Q,R,\ldots$	processus	chapitre 2
$M,N,L,\ldots$	termes (de $\boldsymbol{\Lambda}$ et $\boldsymbol{\lambda}\mathcal{D}$ef)	section 3.3
$a,b,c,\ldots$	noms	chapitre 2
$x,y,z,\ldots$	variables	chapitre 2
$u,v,w,\ldots$	références	chapitre 2
$k,l\,m,\ldots$	étiquettes, champs, méthodes	chapitre 2
$e,f,g,\ldots$	objets dans $\mathbf{Ob}_{1<:}$ et $\boldsymbol{\lambda}\mathcal{O}\mathbf{bj}$, ou références	chapitre 16
$\Gamma,\Delta,\ldots$	environnements pour les jugements de types	chapitre 11
$\tau,\varrho,\vartheta,\ldots$	types	chapitre 11
$\sigma,\eta,\ldots$	schéma de types	définition 13.1 (p. 95)
$\alpha,\beta,\ldots$	variables de types	chapitre 11
$\kappa,\chi,\ldots$	sortes	définitions 11.2 (p. 83) et 14.2 (p. 108)
$\mathbb{T},\mathbb{R},\mathbb{S},\mathbb{I}$	constantes de sorte	définitions 11.2 (p. 83) et 14.2 (p. 108)
$r,s,t,\ldots$	variables de types dans le système $\mathtt{BF}_{\leqslant}$	chapitre 14
$\delta_1,\delta_2,\ldots$	sortes de $\boldsymbol{\pi}$	définition 11.4 (p. 86)

Le calcul bleu: types et objets

Silvano Dal-Zilio
INRIA - projet MEIJE, B.P. 93, F-06902 Sophia-Antipolis Cedex, France.

Le calcul bleu, défini par G. Boudol, est une variante du pi-calcul polyadique qui intègre directement la notion de fonction. Dans cette thèse, nous étudions une version de ce calcul, que l'on étend avec des enregistrements. Nous montrons que le calcul bleu fournit une base adéquate à la conception d'un langage de programmation concurrent et typé, combinant des traits impératifs, d'ordre supérieur, et à objets. Nous étudions notamment comment modéliser la programmation fonctionnelle, et la programmation à objets, et comment intégrer les notions de typage polymorphe et d'héritage.

Cette thèse se divise en quatre parties, dont la première est consacrée à la présentation du calcul bleu, et à l'étude de son expressivité. Dans la seconde partie, nous définissons une équivalence comportementale, basée sur la notion classique de congruence à barbes, et une bisimulation étiquetée plus fine que cette congruence. Nous utilisons ensuite les techniques de preuves de bisimulation modulo pour prouver la validité de plusieurs lois algébriques comme, par exemple, un analogue du théorème de réplication de R. Milner pour le pi-calcul.

La troisième partie est consacrée à l'étude de systèmes de types pour le calcul bleu. Partant d'un système de types simples implicites, qui contient à la fois le système de Curry pour le lambda-calcul et les sortes du pi-calcul, nous proposons successivement trois enrichissements de complexité croissante. Nous étudions l'ajout du sous-typage, puis du polymorphisme paramétrique. Dans ce dernier cas, nous nous intéressons à la décidabilité du problème de l'inférence de type. Finalement, nous étudions un système de types d'ordre supérieur avec récursion et une forme particulière de quantification bornée. Ce système, qui est approprié à l'étude du typage des langages à objets, peut intuitivement être vu comme une version à la Curry du système F-sub. Nous prouvons la correction du typage pour ce système.

Dans la dernière partie, nous nous intéressons à l'étude des objets dans le calcul bleu. Nous donnons une interprétation typée de deux calculs d'objets populaires, le calcul d'objets de M. Abadi et L. Cardelli – dans sa version fonctionnelle, et dans sa version concurrente –, et le calcul d'objets extensibles de K. Fisher et J. Mitchell. Notre contribution principale est une interprétation typée du calcul d'objets extensibles qui préserve la relation de sous-typage.

Mots clés: *calcul de processus, pi calcul, calcul bleu, parallélisme, mobilité, ordre supérieur, bisimulation, types, sous-typage, polymorphisme, objets, délégation, prototypes.*

The Blue Calculus: Types and Objects

The blue calculus, defined by Boudol, is a variant of the polyadic pi-calculus that directly embeds the notion of function. In this thesis, we define a version of the blue calculus, extended with records, and we study whether it provides a good basis for a typed concurrent programming language with imperative, higher-order, and object features. We notably study the modeling of the functional and object oriented programming idioms, and the addition of polymorphic typing and inheritance.

The thesis is divided into four parts. The first part consists of a detailed analysis of the blue calculus and its expressiveness. In the second part, we define a behavioral equivalence based on the classical notion of barbed congruence, and a labeled bisimulation that is finer than this congruence. We then use up-to proof techniques to prove the validity of several algebraic laws like, for example, an analogous of Milner's replication theorem for the pi-calculus.

In the third part, we study type systems for the blue calculus. Starting from a simple implicit type system that encompasses both Curry's type system for the lambda-calculus, and Milner's sorting for the pi-calculus, we successively propose three extensions of increasing complexity. We study the addition of subtyping, then parametric polymorphism. In this last case, we also study the decidability of the type inference problem. Finally, we study an higher-order type system with recursion and a particular form of bounded universal quantification. This system, that is suitable for the typing of objects, can be intuitively viewed as a Curry style presentation of F-sub. We prove the soundness of this system.

The last part of the thesis is devoted to the study of objects in the blue calculus. We give a typed interpretation of two popular object calculi, namely Abadi-Cardelli's object calculus – in its functional version, and in its concurrent version –, and the calculus of extensible objects defined by Fisher et Mitchell. Our main contribution is a typed interpretation of the calculus of extensible objects that preserves subtyping.

Keywords: *process calculi, pi calculus, blue calculus, concurrency, mobility, higher-order, bisimulation, types, subtyping, polymorphism, objects, delegation, prototypes.*