UNIVERSITÉ JOSEPH FOURIER DE GRENOBLE

/__/__/__/__/__/__/__/__/__/__/__/

THÈSE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

préparée à l'INRIA Rhône-Alpes
dans le cadre de l'École Doctorale EDMI

présentée par

Philippe Bidinger

le 14 décembre 2005

Sujet :

Le Kell calcul, typage et implantation

JURY

Gérard Boudol	INRIA Sophia-Antipolis	Président
Frédéric Boussinot	CMA, Ecole des Mines de Paris	Examinateur
Jean-François Monin	Université Joseph Fourier	Examinateur
Davide Sangiorgi	Université de Bologne	Rapporteur
Jean-Bernard Stefani	INRIA Rhône-Alpes	Directeur de thèse
Vasco T. Vasconcelos	Université de Lisbonne	Rapporteur

Résumé

Nous nous intéressons dans cette thèse au Kell calcul, une famille de calculs de processus qui a pour objectif de fournir des bases formelles pour la programmation répartie à composants.

Il est basé sur le le π-calcul d'ordre supérieur, et présente une notion de localité qui permet de modéliser à la fois des composants logiciels, et des sites d'exécution ou des domaines d'administration. Le Kell calcul possède également un opérateur de passivation permettant de capturer l'état d'exécution d'une localité. La combinaison de l'ordre supérieur et de l'opérateur de passivation confère au Kell calcul un grand pouvoir expressif, permettant notamment de programmer simplement différentes formes de mobilité forte, de reconfiguration dynamique et de comportement réflexif.

Nous nous intéressons ensuite à deux systèmes de type pour le Kell calcul. Le premier permet de s'assurer que certaines localités ne seront jamais passivées. Le second assure l'unicité des noms de localité au cours de l'exécution d'un programme, et autorise une implantation répartie du calcul. Les invariants garantis par ces systèmes de type sont importants pour des questions d'efficacité et de modélisation de contraintes courantes dans une programmation répartie à composants.

Nous présentons la spécification formelle d'une implantation répartie du calcul sous la forme d'une machine abstraite, définie indépendamment de tout réseau de communication, et d'une bibliothèque réseau modélisée comme un ensemble de termes du Kell calcul. Nous avons prouvé la correction de la machine par rapport au calcul par l'intermédiaire de techniques co-inductives en utilisant une équivalence barbée. Par ailleurs, nous définissons un langage de programmation basé sur le Kell calcul dont l'implantation suit fidèlement la spécification de la machine abstraite.

Dans une dernière partie, nous présentons un système de type ciblé pour un domaine d'application spécifique : la construction d'intergiciels à composants basés sur le traitement des messages. Notre système de type permet de vérifier la cohérence de certains assemblages de composants, au-delà des vérifications offertes par les systèmes de type des langages de programmation classique comme Java.

Table des matières

Table des figures

Chapitre 1

Introduction

Durant les vingt dernières années, l'évolution des technologies de communication a considérablement modifié la forme des systèmes informatiques. Ces derniers sont passés de réseaux locaux interconnectants quelques dizaines de machines, à des réseaux à échelle planétaire regroupant des millions d'appareils hétérogènes, appartenant à des domaines d'administration distincts, éventuellement mobiles etc. L'incarnation la plus manifeste d'une telle infrastructure est bien sûr Internet. Nous pensons d'une part que de nouveaux modèles, langages, ou paradigmes de programmation sont nécessaires pour appréhender ces nouvelles infrastuctures. Par ailleurs, il est désirable que ces langages se basent sur des fondations mathématiques. Il s'agit en effet d'un prérequis pour pouvoir analyser, comprendre ou vérifier ces systèmes à la complexité accrue.

Luca Cardelli a identifé dans [13] la notion de *séparation* comme fondamentale dans les systèmes à grande échelle. La *séparation physique* des sites, les possibilités de panne de machine ou de réseau font qu'un programmeur ne peut ignorer la distance physique entre les sites. Par ailleurs, l'existence de domaines d'administration ou de sécurité peut s'interpréter comme une forme de *séparation logique*. Pour modéliser ces notions, il introduit le concept de localité comme primitive de modélisation et de programmation des réseaux à grande échelle. Une localité est un endroit nommé dans lequel quelque chose s'exécute. Les localités s'organisent de manière hiérarchique et modélisent des domaines d'administrations séparés, des agents mobiles physique (ordinateurs, téléphone portables) ou virtuel (composant logiciels, processus). Ce concept est formalisé par le calcul des ambients.

Nous pensons que le concept de localité présenté dans le calcul des ambients souffre de plusieurs limitations et notamment, ne permet pas de modéliser, ou alors de manière détournée, des aspects liés aux pannes, à la reconfiguration et à la protection. Ainsi, nous voulons pouvoir exprimer le fait qu'une localité peut tomber en panne et éventuellement modéliser différents types de défaillance. Concernant la protection, on désire pouvoir exercer un contrôle fin des interactions entre deux localités, à la manière de [63, 12], afin par exemple d'exécuter du code potentiellement non fiable dans un environnement sécurisé. Finalement, on veut pouvoir reconfigurer la hiérarchie de localités du système et cela de manière *objective*. Par reconfigurer, on entend détruire, stopper et réactiver, déplacer, dupliquer des localités. Une reconfiguration peut représenter aussi bien une mise à jour de logiciel, qu'une migration de processus ou encore le déploiement d'application. La reconfiguration est par ailleurs fondamentale pour la programmation par composant [47, 42].

Le Kell calcul, est une famille de langages qui a pour objectif la modélisation, ou la

programmation des systèmes répartis à grande échelle, et des systèmes à base de composants. Il est basé sur le le π-calcul d'ordre supérieur, et présente une notion de localité qui possède les caractéristiques dont nous venons de discuter. Il possède également une primitive de *passivation* qui permet de stopper un processus en cours d'exécution, et de le transformer en une valeur. Les éléments de cette famille différent les uns des autres par un langage de motif qui permet de représenter des modes de synchronisation différents. Au chapitre 2, nous justifions les principes de conception de ce calcul, et le définissons formellement. Nous présentons alors deux instances du calcul qui correspondent à des langages de motifs courants. Le πK-calcul utilise les communications du π-calcul, et le jK-calcul les communications du Join calcul.

Afin d'offrir des garanties statiques sur les programmes du Kell calcul, nous proposons au chapitre 3 deux systèmes de type. Le premier premier est indépendant du langage de motifs utilisés. En plus de la garantie d'éviter des erreurs liées à une utilisation incohérente des canaux de communication, il permet de garantir que certaines localités ne seront jamais passivées. Nous définissons ensuite un système de type polymorphe, sur deux instances du calcul, permettant d'isoler une classe de processus pour lesquels on sait garantir l'unicité des noms de localité.

Le chapitre 4 traite de l'implantation répartie du Kell calcul. L'objectif de ce travail est multiple. Tout d'abord, une implantation permet d'évaluer concrêtement si les primitives proposées sont adaptées à la programmation des systèmes considérés. Deuxièmement, elle permet de vérifier que le calcul est implantable efficacement de manière répartie. Notre implantation se base sur une spécification formelle définie sous la forme d'un calcul raffiné. Nous proposons deux versions de cette spécification. La première spécification est relativement proche du calcul et vérifie une propriété de correction forte. La deuxième spécification est plus proche de l'implantation.

Dans le dernier chapitre de cette thèse, nous nous éloignons du Kell calcul et présentons un système de type ciblé pour un domaine d'application spécifique. Le modèle de composant Fractal [10] est un modèle concret utilisé notamment pour la construction, la supervision ou le déploiement d'intergiciels. Il a été utilisé notamment pour développer des intergiciels basés sur le traitement des messages. Dans ce cadre, nous proposons un système de type permettant de vérifier la cohérence d'un assemblage de composants spécialisé dans le traitement des messages. Nous discutons également de perspectives d'utilisation du Kell calcul pour la modélisation de Fractal, et comme langage de description d'architecture dynamique.

Nous concluons au chapitre 6 en résumant nos contributions et en discutant de perspectives futures. Les annexes A et B contiennent respectivement les preuves des chapitres 3 et 4. L'annexe C contient une description du langage `CHALK` basé sur une instance Kell calcul, et de son implémentation en OCaml, correspondant au chapitre 4.

Chapitre 2

Le Kell calcul

2.1 Introduction

Le Kell calcul [64, 60, 7] est une famille de langage qui a pour objectif la modélisation, ou la programmation des systèmes répartis globaux et à base de composants. Il se veut suffisamment simple pour être décrit mathématiquement, tout en exhibant les concepts fondamentaux sous-jacents à ces systèmes.

La section 2.2 constitue un bref rappel sur les calculs de processus destiné au lecteur non familier avec ces concepts. En section 2.3, nous décrivons les principes de conception du calcul, et montrons en quoi il correspondent à des concepts clés des systèmes répartis globaux et à base de composants. Les sections 2.4 et 2.5 décrivent formellement le calcul, ainsi que deux instances importantes. La section 2.6 donne des exemples de programmes du calcul, en insistant notamment sur la notion de *membrane*. La section 2.7 compare le Kell calcul avec d'autres calculs de processus similaires dans leur forme ou leur vocation. Nous concluons en section 2.8 sur des travaux en cours ou futurs.

2.2 Rappel sur les calculs

Un calcul est un langage destiné à décrire formellement et de manière simple des paradigmes de programmation particuliers (programmation concurrente, fonctionnelle, par objets etc.), ou certains types de systèmes (par exemple des systèmes mobiles communicants ou des systèmes biologiques). La définition d'un tel calcul contient généralement et au minimum, sa syntaxe abstraite et sa sémantique opérationnelle. La sémantique opérationnelle donne un sens aux termes du calcul en décrivant leur exécution, ou leur évolution. Elle est donnée par une relation de réduction de la forme $P \rightarrow P'$ où P et P' sont des termes du langage. Elle est définie par inférence à partir d'actions élémentaires. La réduction $P \rightarrow P'$ signifie que le terme P peut évoluer en une étape vers le terme P'.

Selon le contexte, on utilise les mots “terme”, “système”, “processus” ou “programme” pour dénoter les termes d'un calcul. Par exemple, dans le contexte des calculs de processus, calculs mettant en avant la notion d'exécution ou d'évolution parallèle (dont fait parti le Kell calcul), on parle souvent de processus ou de programme. Cette identification peut prêter à confusion. Le mot “programme” désigne généralement un objet syntaxique, alors que “processus” désigne une structure qui correspond à un état d'exécution du programme.

Toutefois, ces deux concepts apparemment distincts s'identifient dans le cadre des calculs de processus. La raison est liée au mode de définition de la sémantique opérationnelle des calculs. On utilise généralement le même langage pour désigner des programmes, et des états d'exécution, c'est à dire des processus. Notons que ce n'est pas toujours le cas : pour modéliser les réductions d'un langage avec effet de bord, où pour avoir une sémantique opérationnelle plus proche d'une implantation, on pourra utiliser des structures plus riches pour modéliser l'état d'exécution d'un programme.

Les calculs utilisent souvent la notion de *substitution* dans la définition de la sémantique opérationnelle, comme un moyen simple de modéliser le remplacement des paramètres formels d'une fonction par les paramètres effectifs, lors de son invocation. Considérons par exemple la réduction suivante, typique des langages fonctionnels :

$$(\lambda x.(x+1))0 \to (x+1)\{0/x\}$$

L'opération $P\{0/x\}$ consiste à remplacer les occurrences de x par 0 dans le terme P, donc ici $(x+1)\{0/x\} = 0+1$. La définition rigoureuse de la substitution est néanmoins plus subtile, nous y reviendrons en section 2.4.

2.3 Principes de conception

π-calcul Le π-calcul [50, 57] est un formalisme de référence pour la modélisation et la programmation des systèmes concurrents. Le Kell calcul se situe dans la lignée du π-calcul dont il hérite les constructions principales. Ce choix est motivé par deux raisons. D'une part, les constructions principales du π-calcul restent pertinentes dans le cadre des systèmes que nous souhaitons modéliser et d'autre part le π-calcul a donné lieu à de nombreux travaux, concernant par exemple sa sémantique, son typage ou encore son implémentation, dont le Kell calcul pourrait bénéficier. Nous décrivons ici les constructions du Kell calcul héritées directement du π-calcul.

Un processus du Kell calcul est construit à partir des éléments suivants.

- Le processus nul, **0** qui n'effectue aucune action.
- L'opérateur de restriction, $\nu a.P$, qui crée un nom a local à P.
- L'opérateur de composition parallèle, $P \mid P'$ correspond au processus qui exécute P et P' de manière parallèle.
- Des messages sur des canaux nommés de la forme $a\langle V\rangle$, où a est un nom et V une valeur. Les noms sont des valeurs particulières.
- Des récepteurs sur des noms prenant la forme $a\langle x\rangle \triangleright P$. Un récepteur est un programme en attente d'un message sur un nom. Lorsqu'un message de la forme $a\langle V\rangle$ est émis, la *continuation* P du récepteur peut s'exécuter, et le paramètre x est remplacé par la valeur V. On formalise cette communication par la réduction suivante :

  $$a\langle V\rangle \mid (a\langle x\rangle \triangleright P) \to P\{V/x\}$$

  Un récepteur disparaît après une communication. On note $\xi \diamond P$ un récepteur *répliqué* qui peut recevoir un nombre arbitraire de messages. La règle de réduction correspondante est donnée par :

  $$a\langle V\rangle \mid (a\langle x\rangle \diamond P) \to P\{V/x\} \mid (a\langle x\rangle \diamond P)$$

En réalité, Les récepteurs du π-calcul ne correspondent qu'à une instance du Kell calcul. Ce dernier utilise une notion abstraite de récepteur de le forme $\xi \triangleright P$ où ξ est un *motif*. Le motif ξ détermine une condition de *filtrage* correspondant à une conjonction particulière de messages, avec éventuellement des contraintes sur leur contenu. Si ces conditions sont réalisées, le récepteur peut libérer sa continuation. Le motif permet également d'extraire des valeurs de ces messages. La donnée d'un langage de motifs définit une *instance* du Kell calcul.

L'utilisation d'un langage de motif permet essentiellement de pouvoir étudier dans un cadre général des calculs d'expressivité différente. De plus, d'un point de vue programmation, il est intéressant de ne pas se restreindre à un type de motif a priori. En effet, des langages de motifs différents sont utilisés dans des langages de programmation comme OCaml, Polymorphic C#, où JoCaml. En gardant non spécifié le langage de motif, on peut dériver des résultats généraux comme par exemple, la caractérisation de certaines formes d'équivalence [60], ou encore des systèmes de type comme nous le verrons au chapitre 3.

Localités hiérarchiques La notion de localité hiérarchique a été proposée par Cardelli et Gordon dans le calcul des ambiants mobiles [14], comme un élément de modélisation et de programmation des système répartis à grande échelle. De manière très générale, une localité est un endroit nommé où quelque chose s'exécute. Eles permettent de modéliser de manière naturelle l'organisation hiérarchique des systèmes à grande échelle. Par exemple, une localité peut modéliser un domaine d'administration, un système d'exploitation, une machine virtuelle, un processus, un composant logiciel etc.

Comme construction d'un langage de programmation, une localité permet d'englober et de nommer une partie d'un programme, ou plus exactement une partie d'un programme éventuellement en cours d'exécution. Elle peut alors remplir plusieurs rôles importants :

- Elle permet de réifier au niveau du langage l'endroit où un programme s'exécute, ce qui permet une programmation *sensible à l'emplacement*.
- Elle permet de modéliser la notion de composant en servant d'unité d'encapsulation ou de reconfiguration.
- Elle permet de modéliser le concept d'agent mobile, une localité étant également une unité de migration.

Formellement, nous rajoutons un nouveau constructeur à notre langage. On dit que $a[P]$ est une localité de nom a dans laquelle P s'exécute. Les localités peuvent s'imbriquer sans restriction : Par exemple, $a[b[a[P]] \mid Q]$ constitue un terme valide du calcul.

Ordre supérieur La mobilité de code fait partie intégrante des systèmes informatiques actuels. On la trouve par exemple dans des pages web à contenu dynamique (Javascript), des applets en Java, du code migrant dans des réseaux actifs, des mises à jour d'un logiciel ou d'un système d'exploitation, ou encore dans le téléchargement de code postscript vers une imprimante. Ces exemples correspondent à une forme de *mobilité faible* où simplement du code passif est communiqué. A l'inverse, on parle de *mobilité forte* lorsque du code actif, en cours d'exécution, se déplace d'un site à un autre.

Du point de vue de la programmation globale, la mobilité, faible ou forte, permet (en conjonction avec la liaison dynamique, cf. plus bas) par exemple des interactions fines entre un client et un serveur en s'affranchissant de problèmes de fluctuations de bande

passante. Par ailleurs, elle permet également la mise à jour de logiciels ou la reconfiguration dynamique.

Pour permettre d'exprimer directement la mobilité faible de manière directe, les communications dans le Kell calcul sont d'ordre supérieur : les valeurs transmises peuvent être non seulement des noms ou des valeurs primitives, mais également des programmes.

Passivation On appelle *passivation* l'opération qui consiste à stopper l'exécution d'un processus et stocker son état sous la forme d'une valeur que l'on pourra manipuler ultérieurement (pour, par exemple, la transmettre, la dupliquer ou la réactiver). Cette opération, en conjonction avec les communications d'ordre supérieur du calcul, est à la base de la de la mobilité forte.

La passivation est réalisée en généralisant les primitives de communications. Nous avons vu qu'un récepteur $a\langle x\rangle \triangleright P$ peut consommer un message $a\langle V\rangle$. De la meme façon, une localité $a[P]$ peut être consommée par un récepteur particulier de le forme $a[x] \triangleright Q$. Avant la passivation, le programme P s'exécute dans la localité a. Lors de la passivation, cette localité est détruite, et l'état d'exécution de P est transformé en une valeur. Cette valeur sera accessible par la variable x dans le programme Q. Une telle valeur pourra être détruite, réactivée, dupliquée... Formellement, cela se traduit par la réduction :

$$a[P] \mid (a[x] \triangleright Q) \rightarrow Q\{P/x\}$$

La valeur P substituée à x dans Q après réduction correspond intuitivement à un état d'exécution. On peut comparer cette réduction à une communication simple :

$$a\langle P\rangle \mid (a\langle P\rangle \triangleright Q) \rightarrow Q\{P/x\}$$

Dans le cas de la passivation, P est un processus en cours d'exécution. Dans le deuxième cas, P est une valeur. De plus le formalisme ne nous permet pas de savoir a priori si la valeur P correspond à un programme, ou à un processus stoppé dont l'état a été sauvegardé (cf. section 2.2). Néanmoins, il est clair qu'une implantation du calcul doit distinguer ces deux types d'objet, nous y reviendrons au chapitre 4.

Principe d'action locale Nous avons vu que les récepteurs du Kell calcul utilisent une notion abstraite de motif, permettant de spécifier une conjonction particulière de messages. Cette conjonction n'est pas arbitraire mais doit obéir à un *principe d'action locale* que l'on peut énoncer de la manière suivante : une action ne doit impliquer qu'une seule localité à la fois, ou une location et son environnement direct. Ce principe est motivé par deux raisons. Tout d'abord, il est nécessaire pour que les actions du calcul puissent être implantable dans un contexte asynchrone et réparti. Par ailleurs, il permet la réalisation de membranes programmables (cf. paragraphe suivant).

Dans le Kell calcul, essentiellement quatre types de réduction sont possibles. Nous les décrivons ci-dessous.

1. La réception d'un message local illustré par la réduction suivante, où le message, $a\langle V\rangle$, sur le port a, et transportant la valeur Q, est reçu par le récepteur $a\langle x\rangle \triangleright P$, ce qui correspond à la réduction suivante :

$$a\langle V\rangle \mid (a\langle x\rangle \triangleright P) \rightarrow P\{V/x\}$$

2. La réception d'un message provenant de son environnement direct, illustré par la règle ci-dessous, où un message, $a\langle V\rangle$, sur le port a, et transportant la valeur V est reçu par le récepteur $a\langle x\rangle^{\uparrow} \triangleright P$, situé dans le kell b (le motif $a\langle x\rangle^{\uparrow}$ indique qu'un message est attendu de l'extérieur du kell local) :

$$a\langle V\rangle \mid b[a\langle x\rangle^{\uparrow} \triangleright P] \rightarrow b[P\{V/x\}]$$

3. La réception originaire d'un sous-kell, illustrée par la réduction suivante, où un message, $a\langle V\rangle$, sur le port a, transportant la valeur V, et provenant du sous-kell b est reçu par le récepteur $a\langle x\rangle^{\downarrow} \triangleright P$, situé dans le kell parent du kell b (le motif $a\langle x\rangle^{\downarrow}$ indique qu'un message est attendu d'un kell situé à l'intérieur du kell local) :

$$(a\langle x\rangle^{\downarrow} \triangleright P) \mid b[a\langle Q\rangle \mid R] \rightarrow P\{Q/x\} \mid b[R]$$

4. La suspension d'un kell, illustrée par la réduction suivante, où le sous-kell de nom a est détruit, et le processus Q qu'il contient est placé dans un message sur le port b :

$$a[Q] \mid (a[x] \triangleright b\langle x\rangle) \rightarrow b\langle Q\rangle$$

Les actions de la forme 1 ci-dessus sont les actions standards du π-calcul. Les actions de la forme 2 et 3 sont simplement des extensions des actions de réceptions du π-calcul dans le cas où le récepteur est situé dans un kell. On peut les comparer aux actions de communications dans le calcul des Boxed Ambients ou dans le Seal calcul [16].

Membranes programmables Nous avons vu que le concept de localité vise à modéliser diverses structures intervenant dans les systèmes répartis à grande échelle, mais également à modéliser des composants logiciels. Une membrane, associée à une localité, offre la possibilité de modéliser de manière uniforme ces différents types de comportements à partir des trois mécanismes suivants :

- En établissant un protocole de communication entre une localité et son environnement.
- En établissant un protocole de communication entre des localités soeurs.
- En contrôlant les sous-localités

Dans le Kell calcul, les membranes ne sont pas primitives, mais sont facilement réalisable en vertu du principe d'actions locales, et des possibilités de contrôle offerte par l'opération de passivation. Par exemple, une membrane autour de $a[K]$ peut prendre la forme : $c[M(a) \mid a[K]]$, et dans ce cas, son comportement est défini par le processus $M(a)$.

Nous verrons des exemples en section 2.6 des exemples d'utilisation de membranes pour modéliser différents types de pannes. Au chapitre 4, nous utiliserons également ce mécanisme pour spécifier une implantation répartie du calcul indépendante du réseau considéré. Au chapitre 5, nous montrerons également comment ce type de membrane reflète la sémantique d'un modèle de composants.

Liaison dynamique On parle de liaison dynamique lorsqu'un programme peut se lier à certaines ressources (services, appel de fonctions, librairies) au moment de son exécution, plutôt qu'à la compilation. La possibilité de liaison dynamique est cruciale pour la

programmation globale pour des raisons d'efficacité, de sécurité, d'adaptabilité ou de fiabilité. Par exemple, disposer d'une forme de liaison dynamique est nécessaire dans les cas suivants :

- On veut permettre à un programme téléchargé sur un site distant de se lier aux ressources locales, par exemple pour réduire les coûts des communications.
- Exécuter un programme potentiellement malicieux dans un environnement sécurisé.
- Mettre à jour dynamiquement des librairies.

Dans le Kell calcul, la liaison dynamique est une conséquence du caractère local des communications. Par exemple, dans le système :

$$a[\mathtt{print}\langle x\rangle \triangleright P] \mid b[\mathtt{print}\langle x\rangle \triangleright Q]$$

Un message de la forme $\mathtt{print}\langle V\rangle$ réagira avec des récepteurs différents selon qu'il se situe sur a ou b.

2.4 Le Kell calcul : syntaxe et sémantique opérationnelle

2.4.1 Syntaxe

La syntaxe du calcul est définie figure 5.7. Nous nous donnons un ensemble infini de *noms* (ou *canaux*), NAMES, dont les éléments sont décrits par a,b,..., ainsi qu'un ensemble infini de variables, VARS, dont les éléments sont décrits par x,y,.... On appelle identifiant et on note u,v,w,... les éléments de NAMES $\cup$ VARS (on note cet ensemble ID). On note $\widetilde{V}$ un vecteur de la forme $(V_1,\ldots,V_n)$. La syntaxe du calcul est paramétrée par un ensemble E, que l'on appelle *ensemble de motifs*, on utilise les variables ξ et ζ pour désigner des *motifs* de E. On note l'ensemble des *processus* du Kell calcul K_E, un simplement K lorsque l'on peut déduire E du contexte. On utilise P, Q, ... pour les nommer. On appelle *valeur* et l'on note V,W,... un processus ou un identifiant. On note V_E, ou simplement V leur ensemble. On appelle *message* un processus de la forme $u\langle\widetilde{V}\rangle$. On appelle *kell*[1] un processus de la forme $u[P]$, où u est le nom du kell. Dans un kell de la forme $u[\ldots \mid u_j[P_j] \mid \ldots \mid Q_k \mid \ldots]$ on appelle *sous-kell* les processus $u_j[P_j]$. On appelle *récepteur* un processus de la forme $\xi \triangleright P$.

Contextes d'évaluation La syntaxe des contextes d'évaluation est donnée figure 2.2. Substituer l'emplacement désigné par "$\cdot$" dans un contexte d'évaluation $\mathbf{E}$ avec un terme du Kell calcul Q résulte en un terme du calcul noté $\mathbf{E}\{Q\}$.

Conventions Dans les termes $\nu a.P$ and $\xi \triangleright P$, la portée des opérateurs s'étend aussi loin à droite que possible. Ainsi, $\xi \triangleright P \mid Q$ est égal à $\xi \triangleright (P \mid Q)$, et $\nu a.P \mid Q = \nu a.(P \mid Q)$. Nous utilisons les abréviations standards du π-calcul : $\nu a_1 \ldots a_q.P$ pour $\nu a_1.\ldots.\nu a_q.P$, ou $\nu\widetilde{a}.P$ si $\widetilde{a} = (a_1 \ldots a_q)$. Par convention, si le vecteur de noms $\widetilde{a}$ est vide, alors $\nu\widetilde{a}.P \triangleq P$. On note également $\prod_{i\in I} P_i$, $I = \{1,\ldots,n\}$ la composition parallèle $(P_1 \mid (\ldots (P_{n-1} \mid P_n)\ldots))$. Par convention, si $I = \emptyset$, alors $\prod_{i\in I} P_i \triangleq \mathbf{0}$. Par abus de notation, on identifie $\widetilde{V}$ avec le mot $V_1 \ldots V_n$ et l'ensemble $\{V_1,\ldots,V_n\}$.

1. Le mot "kell" est supposé rappeler le mot anglais "cell", par une analogie lointaine avec des cellules biologiques

Processus

$P,Q,\ldots ::= \mathbf{0}$		nil
$\mid x$		variable
$\mid u\langle\widetilde{V}\rangle$		message
$\mid \xi \triangleright P$		récepteur
$\mid \nu a.P$		restriction
$\mid P \mid Q$		composition parallèle
$\mid u[P]$		kell

Identifiants

$u ::= a,b,c,\ldots$		canal
$\mid x,y,z,\ldots$		variable

Valeurs

$V,W,\ldots ::= u,v,w,\ldots$		identifiant
$\mid P$		processus

FIG. 2.1 – *Syntaxe du Kell calcul*

$$
\begin{aligned}
\mathbf{E} ::= {} & \cdot \\
\mid {} & \nu a.\mathbf{E} \\
\mid {} & u[\mathbf{E}] \\
\mid {} & P \mid \mathbf{E}
\end{aligned}
$$

FIG. 2.2 – *Contextes*

2.4.2 Sémantique opérationnelle

Substitution Soient T,T' deux ensembles tel que $T \subseteq T'$. On appelle *substitution*, et l'on note θ, une fonction de T vers T', qui est égale à l'identité hormis sur un ensemble fini de valeurs. On note leur ensemble $\Theta_{T,T'}$. On définit de plus les fonctions $\mathtt{dom}(\theta) = \{x \in T \mid \theta(x) \neq x\}$ et $\mathtt{ran}(\theta) = \sigma(\mathtt{dom}(\theta))$, respectivement le domaine et l'image de la substitution θ.

On définit deux ensembles particuliers de substitutions. On note Θ_{V_E} l'ensemble des substitutions de VARS vers V_E. On définit Θ_{ID} comme l'ensemble des substitutions de ID vers lui-même qui vérifient de plus $\phi(\mathrm{NAMES}) \subseteq \mathrm{NAMES}$.

Messages annotés Etant donné un ensemble de motif E, nous définissons une nouvelle classe de termes que nous appelons *messages annotés sur* E et nous notons M_E leur ensemble. Un message annoté est soit un message local $u\langle V\rangle$, soit un message venant du haut $u^{\uparrow}\langle V\rangle$, soit un message venant du bas $u^{\downarrow^v}\langle V\rangle$, soit un message actif $u[P]$, soit une composition parallèle de messages, soit un message vide. P et V étant respectivement des processus et valeurs de K_E. La syntaxe des messages est récapitulée figure 5.4.

On définit une relation d'équivalence $\equiv$ sur M_E comme la plus petite relation d'équivalence qui fait de $(M, \mid, \mathbf{0})$ un monoide commutatif. On définit 2.4 une notion de contexte.

$$
\begin{array}{lll}
m ::= & u\langle \widetilde{V} \rangle & \text{message local} \\
| & u^{\uparrow}\langle \widetilde{V} \rangle & \text{message venant du haut} \\
| & u^{\downarrow^{v}}\langle \widetilde{V} \rangle & \text{message venant du bas} \\
| & u[P] & \text{message actif} \\
| & m \mid m & \text{composition parallèle} \\
| & \mathbf{0} & \text{message vide}
\end{array}
$$

FIG. 2.3 – *Messages annotés*

$$
\begin{array}{lll}
\mathbf{C}_m ::= & u\langle V_1, \ldots, \mathbf{C}_i, \ldots, V_n \rangle \\
| & u^{\uparrow}\langle V_1, \ldots, \mathbf{C}_i, \ldots, V_n \rangle \\
| & u^{\downarrow^{v}}\langle V_1, \ldots, \mathbf{C}_i, \ldots, V_n \rangle \\
| & u[\mathbf{C}_i] \\
| & \mathbf{C}_m \mid m
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{C}_i ::= & \cdot \\
| & u\langle V_1, \ldots, \mathbf{C}_i, \ldots, V_n \rangle \\
| & u[\mathbf{C}_i]
\end{array}
$$

FIG. 2.4 – *Contextes de messages annotés*

Langage de motif Afin de pouvoir définir la sémantique opérationnelle du Kell calcul, nous avons besoin de définir les notions de noms et variables libres, d'α-équivalence, et de substitution évitant les captures. On veut également décrire formellement l'intuition qu'un motif permet de lier certaines variables d'un processus (dites "liées dans le motif") à des sous-termes d'un ensemble de message annotés. Pour ce faire, on va munir un ensemble de motifs d'une structure permettant de caractériser formellement ces opérations.

Définition 2.4.1 (Langage de motif) *Un langage de motif est une structure de la forme*

$$\mathsf{L} = (\mathsf{L}_s, \mathit{fn}, \mathit{fv}, \mathit{bv}, \mathit{match}, \mathit{subs})$$

L_s *est un ensemble quelconque que l'on appelle le support de* L. fn *est une fonction de* L_s *dans* $\mathcal{P}^f_m(\textsc{Names})$ *qui associe à un motif un ensemble fini de* noms libres. fv *et* bv *sont des fonctions de* L_s *dans* $\mathcal{P}^f_m(\textsc{Vars})$ *qui renvoient respectivement les* variables libres *et* liées *d'un motif.* subs *est une fonction de* $\mathsf{L}_s \times \Theta_{\text{ID}}$ *dans* $\mathsf{L}_s \uplus \{\bot\}$ *et calcule l'image d'un motif par une substitution. Pour finir, La relation* match*, appelée relation de filtrage, est une relation ternaire sur* $\mathsf{L}_s \times \mathcal{P}^f_m(\mathsf{M}_{\mathsf{L}_s}) \times \Theta_{\mathsf{V}_{\mathsf{L}_s}}$.

Un langage de motif doit vérifier les axiomes suivants :

$$\begin{aligned}
\mathit{dom}(\theta) \cap \mathit{fv}(\xi) = \emptyset &\implies \xi\theta = \xi \\
\mathit{match}(\xi, m, \theta) &\implies \mathit{dom}(\theta) = \mathit{bn}(\xi) \\
m \equiv m' &\implies (\mathit{match}(\xi, m, \theta) \iff \mathit{match}(\xi, m', \theta)) \\
\mathit{match}(\xi, m, \theta) &\implies (\forall x \in \mathit{dom}(\theta), C_m\{\theta(x)\} = m)
\end{aligned}$$

Remarquons que les seules valeurs pouvant être substituées aux variables libres d'un motif sont des identifiants. Un langage de motif L détermine une instance du Kell calcul. La syntaxe des processus et des messages annotés est donnée relativement à l'ensemble L_s. Dans la suite, nous supposons donné un langage de motif.

α-équivalence et substitution Nous définissons maintenant les notions d'α-équivalence et de substitution évitant les captures d'identifiants. Ces notions sont bien connues, mais dans notre cas, l'utilisation d'un langage de motif arbitraire nécessite de les repréciser.

La manière habituelle de définir ces notions consiste tout d'abord à définir une opération de substitution *simple* où tous les identifiants sont remplacés par leur image. On définit ensuite les notions de noms et variables libres d'un processus, puis l'α-équivalence, et finalement la substitution évitant les captures. On procède ici de manière similaire en utilisant les différentes fonctions de la structure de langage de motif.

Intuitivement la fonction `subs` correspond à une opération de substitution simple. Si $\theta \in \Theta_{\text{ID}}$, on note $P\theta$ le processus P dans lequel tous les identifiants sont remplacés par leur image par ϕ et les motifs ξ par leur leur images par $\texttt{subs}(.,\phi)$. Par ailleurs, si $P\theta$ est un terme syntaxiquement incorrect, ou si $\theta = \bot$, on pose $P\theta = \bot$.

Nous définissons figure 2.5 les *noms libres* et *variables libres* d'un processus, en utilisant les fonctions `fn`, `bn` et `fv` du langage de motif.

L'α-équivalence est la plus petite relation d'équivalence vérifiant les deux axiomes :

$$\begin{aligned}
&\nu a.P =_\alpha \nu b.P\{b/a\},\ b \notin \texttt{fn}(P) \\
&\xi \triangleright P =_\alpha \texttt{subs}(\xi, \phi) \triangleright P\phi
\end{aligned}$$

Noms libres	**Variables libres**
$\mathtt{fn}(\mathbf{0}) = \emptyset$	$\mathtt{fv}(\mathbf{0}) = \emptyset$
$\mathtt{fn}(x) = \emptyset$	$\mathtt{fv}(x) = \{x\}$
$\mathtt{fn}(a) = \{a\}$	$\mathtt{fv}(a) = \emptyset$
$\mathtt{fn}(\nu a.P) = \mathtt{fn}(P) \setminus \{a\}$	$\mathtt{fv}(\nu a.P) = \mathtt{fv}(P)$
$\mathtt{fn}(P \mid Q) = \mathtt{fn}(P) \cup \mathtt{fn}(Q)$	$\mathtt{fv}(P \mid Q) = \mathtt{fv}(P) \cup \mathtt{fv}(Q)$
$\mathtt{fn}(u[P]) = \mathtt{fn}(u) \cup \mathtt{fn}(P)$	$\mathtt{fv}(u[P]) = \mathtt{fv}(u) \cup \mathtt{fv}(P)$
$\mathtt{fn}(u\langle\widetilde{V}\rangle) = \mathtt{fn}(u) \cup \mathtt{fn}(\widetilde{V})$	$\mathtt{fv}(u\langle\widetilde{V}\rangle) = \mathtt{fv}(u) \cup \mathtt{fv}(\widetilde{V})$
$\mathtt{fn}(\xi \triangleright P) = \mathtt{fn}(\xi) \cup \mathtt{fn}(P)$	$\mathtt{fv}(\xi \triangleright P) = \mathtt{fv}(\xi) \cup (\mathtt{fv}(P) \setminus \mathtt{bv}(\xi))$

FIG. 2.5 – *Noms et variables libres*

Où $\theta \in \Theta_{\text{ID}}$ est injectif et tel que $\mathtt{dom}(\phi) \subseteq \mathtt{bv}(\xi)$, $\mathtt{ran}(\phi) \cap \mathtt{fv}(P) = \emptyset$ et $\mathtt{ran}(\phi) \subseteq$ VARS.

On peut alors définir la *substitution évitant les captures* à l'aide de la fonction `subs` et en renommant par α-équivalence les variables et noms liés pour éviter le phénomène de capture. On note $\theta(\widetilde{V})$ pour le vecteur $\theta(V_1), \ldots, \theta(V_n)$.

$$
\begin{aligned}
\theta(\mathbf{0}) &= \mathbf{0} \\
\theta(a) &= a \\
\theta(P \mid Q) &= \theta(P) \mid \theta(Q) \\
\theta(u[P]) &= \theta(u)[\theta(P)] \\
\theta(u\langle\widetilde{V}\rangle) &= \theta(u)\langle\theta(\widetilde{V})\rangle \\
\theta(\nu a.P) &= \nu a.\theta(P),\ a \notin \mathtt{fn}(\theta(P)) \\
\theta(\xi \triangleright P) &= \mathtt{subs}(\xi,\theta) \triangleright \theta(P),\ \mathtt{bv}(\xi) \cap \mathtt{fv}(\theta(P)) = \emptyset
\end{aligned}
$$

Dans la suite, toutes les substitutions seront de ce type. On notera $P\theta = \theta(P)$.

Réduction Nous disons qu'un processus est *clos* lorsqu'il ne contient pas de variables libres. La sémantique opérationnelle du Kell calcul est donnée par une relation de réduction $P \to Q$ entre termes clos. Cette relation est définie à partir d'une relation de congruence structurelle et une relation de sous-réduction, définies également sur des processus clos.

La relation d'équivalence structurelle $\equiv$ est la plus petite relation d'équivalence qui vérifie les règles de la figure 2.6. Elle est essentiellement définie comme pour le π-calcul. La règle S.ERR permet de caractériser simplement les processus erronés comme structurellement équivalents au processus $\perp$. Informellement, l'équivalence structurelle permet de mettre en juxtaposition des termes pour leur permettre de réagir à l'aide des axiomes de la relation de réduction.

En plus de la relation d'équivalence structurelle, nous définissons une relation de *sous-réduction* dont le rôle est d'extruder les créations de noms au-delà de la frontière d'un kell avant qu'il soit passivé. Nous détaillons ce point en section 2.4.3. La relation de sous-réduction est définie comme la plus petite relation binaire sur K qui vérifie les règles de la figure 2.7.

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \text{ S.Par.Assoc} \qquad P \mid Q \equiv Q \mid P \text{ S.Par.Com}$$

$$P \mid \mathbf{0} \equiv P \text{ S.Par.Nil} \qquad \nu a.\mathbf{0} \equiv \mathbf{0} \text{ S.Nu.Nil} \qquad \nu a.\nu b.P \equiv \nu b.\nu a.P \text{ S.Nu.Com}$$

$$\frac{a \notin \mathtt{fn}(Q)}{(\nu a.P) \mid Q \equiv \nu a.P \mid Q} \text{ S.Nu.Par} \qquad \frac{P =_\alpha Q}{P \equiv Q} \text{ S.}\alpha \qquad \frac{P \equiv Q}{\mathbf{E}\{P\} \equiv \mathbf{E}\{Q\}} \text{ S.Context}$$

$$\frac{}{\mathbf{E}\{\bot\} \equiv \bot} \text{ S.Err}$$

Fig. 2.6 – *Equivalence structurelle*

$$\frac{u \neq a}{u[\nu a.P] \rightsquigarrow \nu a.u[P]} \text{ SR.Kell} \qquad \frac{P \rightsquigarrow Q}{\mathbf{E}\{P\} \rightsquigarrow \mathbf{E}\{Q\}} \text{ SR.Ctx}$$

$$\frac{P' \equiv P \qquad P \rightsquigarrow Q \qquad Q \equiv Q'}{P' \rightsquigarrow Q'} \text{ SR.Str}$$

Fig. 2.7 – *Relation de sous-réduction*

On dit qu'un processus P est sous *forme normale* si $P_* \not\rightsquigarrow$. On désignera par P_*,Q_*,$R_* \ldots$, des processus sous forme normale. De plus, une caractérisation syntaxique simple de ces processus est donnée figure 2.8.

Par ailleurs, on peut montrer facilement que pour tout processus P, il existe un processus sous forme normale P_* tel que $P \rightsquigarrow^* P_*$. De plus, un tel processus est unique modulo équivalence structurelle.

$$\begin{aligned} P_* ::= {} & \mathbf{0} \\ \mid {} & x \\ \mid {} & \xi \triangleright P \\ \mid {} & P_* \mid P_* \\ \mid {} & u[P_*] \\ \mid {} & u\langle \widetilde{V} \rangle \end{aligned}$$

Fig. 2.8 – *Processus en forme normale*

La relation de réduction $\rightarrow$ est la plus petite relation binaire sur K qui vérifie les règles données en figure 2.10. Les règles R.Ctx et R.Str sont standards. La règle R.Str.Ext peut-être vue comme une une règle de congruence structurelle orientée, et permet l'extrusion des noms à l'extérieur des kells. Les règles importantes sont R.Red.L et R.Red.G. Elle utilisent la fonction `match` du langage de motif ainsi que les trois prédicats de réduction

définis figure 2.9.

Nous détaillons la règle R.RED.G. Intuitivement, on veut qu'un récepteur du calcul puisse réagir avec quatre catégories de *réactants*. Les réactants sont constitués des messages provenant des trois directions possibles : les kells parent, local ainsi que les sous-kells directs. Ce dernier type de réaction correspond aux actions de passivation. Le rôle des prédicats de réduction est d'isoler ces différents réactants, et éventuellement de les "transformer" en messages annotés afin de les utiliser dans le prédicat `match`. Dans les prémisses de R.RED.G, $\Delta(U_1)$ impose à U_1 d'être une composition parallèle de messages locaux. $\Upsilon(U_2)$ impose à U_2 d'être une composition parallèle de kells, ainsi que la condition supplémentaire que les processus présents dans ces kells soient tous sous forme normale. Le prédicat $\Psi(U_3,M_b)$ impose à U_3 d'être une composition parallèle de kell contenant des messages. M_b est le multi-ensemble contenant ces messages, annotés par leur kell émetteur. Cette annotation permet plus de liberté dans la définition de `match` qui peut ainsi exhiber des comportements différents en fonction de la provenance des messages. Le prédicat $\Gamma(U_4,M_h)$ impose à U_4 d'être une composition parallèle de messages, et M_h est le multi-ensemble des messages annotés correspondants.

La règle R.RED.L est quasi-identique. La seule différence est que le récepteur se situe au plus haut niveau et ne peut pas réagir avec des messages situés dans le kell parent.

$$
\begin{aligned}
\Delta(U) &\iff U = \prod_{j\in J} u_j\langle \widetilde{V_j}\rangle \\
\Upsilon(U) &\iff U = \prod_{j\in J} u_j[\widetilde{V_j}] \wedge \forall i \in J.P_j \not\rightsquigarrow \\
\Gamma(U,M_h) &\iff U = \prod_{j\in J} u_j\langle \widetilde{V_j}\rangle \wedge M_h = \prod_{j\in J} u_j^{\uparrow}\langle \widetilde{V_j}\rangle \\
\Psi(U,M_b) &\iff U = \prod_{j\in J} u_j[R_j \mid \prod_{i\in I_j} u_i\langle \widetilde{V_i}\rangle] \wedge M_b = \prod_{j\in J}\prod_{i\in I_j} u_i^{\downarrow^{u_j}}\langle \widetilde{V_j}\rangle
\end{aligned}
$$

FIG. 2.9 – *Prédicats de réduction*

$$
\frac{\texttt{match}(\xi,U_1 \mid U_2 \mid M_k,\theta) \qquad \Delta(U_1) \qquad \Upsilon(U_2) \qquad \Psi(U_3,M_b)}{(\xi \triangleright P) \mid U_1 \mid U_2 \mid U_3 \to P\theta}\ \text{R.RED.L}
$$

$$
\frac{\texttt{match}(\xi,U_1 \mid U_2 \mid M_b \mid M_u,\theta) \qquad \Delta(U_1) \qquad \Upsilon(U_2) \qquad \Psi(U_3,M_b) \qquad \Gamma(U_4,M_u)}{u[(\xi \triangleright P) \mid U_1 \mid U_2 \mid U_3 \mid R] \mid U_4 \to u[P\theta \mid R]}\ \text{R.RED.G}
$$

$$
\frac{P \to Q}{\mathbf{E}\{P\} \to \mathbf{E}\{Q\}}\ \text{R.CTX} \qquad\qquad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}\ \text{R.STR}
$$

$$
\frac{P \rightsquigarrow^* P' \qquad P' \to Q}{P \to Q}\ \text{R.STR.EXTR}
$$

FIG. 2.10 – *Relation de réduction*

2.4.3 Discussion

Extrusion des noms Nous n'avons pas de règles d'équivalence structurelle qui gèrent l'extrusion de la portée des noms au delà de la frontière d'un kell. Autrement dit, nous n'avons pas la règle du calcul des ambiants $u[\nu a.P] \equiv \nu a.u[P]$ lorsque $a \neq u$. La raison est que nous voulons éviter le phénomène suivant :

$$(a[x] \triangleright x \mid x) \mid a[\nu b.P] \rightarrow (\nu b.P) \mid (\nu b.P)$$
$$(a[x] \triangleright x \mid x) \mid \nu b.a[P] \rightarrow \nu b.P \mid P$$

Cet exemple montre l'interaction qu'il existe entre la duplication et la création de nom. Selon l'ordre des opérations de passivation et de création de nom, le nom b peut être ou non dupliqué. Ce genre de programme devient difficilement analysable puisque le programmeur n'a a priori aucun contrôle sur le moment ou le nom sera effectivement extrudé. Toutefois, une telle extrusion de nom est nécessaire pour permettre les communications à travers les frontières de kell. La solution adoptée ici consiste à restreindre la passivation à des processus sans restriction de nom en contexte d'évaluation. Autrement dit, lorsqu'un kell est passivé, toutes les créations de nom auront été effectuées, ce qui correspond à la première réduction de l'exemple ci-dessus. L'opération de restriction de nom est interprétée comme une création de nom nouveau, avec une priorité sur la passivation.

Une deuxième solution, adoptée par exemple dans le Seal calcul [16] ou dans d'autres versions du Kell calcul [7] consiste à n'extruder les noms que lors des communications, à l'aide de règles de réduction similaire à la réduction suivante

$$a[\nu b.c\langle b\rangle \mid Q] \mid c\langle x\rangle \triangleright P \rightarrow \nu b.a[Q] \mid P\{b/x\}$$

Erreurs d'exécution Si θ est une substitution et P un processus, l'opération $P\theta$ n'est pas toujours définie. Les termes de la forme $x[P]$, ou $x\langle V\rangle$ ou encore $\xi \triangleright P$ avec $x \in \mathtt{fv}(\xi)$ nécessitent $\sigma(x) \in \textsc{Id}$. La solution adoptée par exemple dans [60] consiste à utiliser deux catégories syntaxique différentes pour des variables de nom et des variables de processus. Néanmoins, cette solution n'interdit pas des termes de la forme $a\langle V\rangle \mid a\langle x,y\rangle \triangleright P$, où le nom a n'est pas utilisé de manière homogène.

A l'inverse, nous ne faisons pas de différence entre variable de nom et variable de processus et supposons que l'application d'une substitution est une fonction partielle. On définit au chapitre 3 un système de type générique qui permet d'assurer, d'une part que les opérations de substitution sont toujours définies au cours de la réduction d'un programme bien typé, et d'autre part que les canaux de communications sont utilisés de manière homogène.

2.4.4 Équivalence

Nous définissons une relation d'équivalence sur les termes du Kell calcul, basée sur la notion de bisimulation et de barbes (où encore, prédicat d'observation) qui nous sera utile au chapitre 4. Les définitions de bisimulations barbées fortes et de bisimilarité barbées fortes sont classiques [57]. Nous les rappelons ci-dessous.

Définition 2.4.2 (Bisimulation barbée forte) *Soient TS_1 et TS_2 deux ensembles de systèmes de transitions munis d'un même prédicat d'observation $\downarrow_a$, $a \in$ Names. Une relation $R \subseteq TS_1 \times TS_2$ est une* simulation barbée forte *si dès que $(A,B) \in R$, nous avons*

- *Si $A \downarrow_a$ alors $B \downarrow_a$*
- *Si $A \rightarrow A'$ alors il existe B' tel que $B \rightarrow B'$ et $(A',B') \in R'$*

Une relation R est une bisimulation barbée forte si R et R^{-1} sont toutes deux des bisimulations barbées fortes.

Définition 2.4.3 (Bisimilarité barbée forte) *Deux systèmes de transition A et B sont dits* fortement bisimilaires barbées*, et l'on note $A \sim B$, s'il existe une bisimulation barbée forte R telle que $(A,B) \in R$.*

Pour définir la bisimilarité forte pour les processus du Kell calcul, nous nous donnons le prédicat d'observation suivant.

Définition 2.4.4 (Prédicat d'observation pour processus du Kell calcul) *Si P est un processus du Kell calcul, On a $P \downarrow_a$ si et seulement si l'un des cas suivants est vrai :*

1. $P \equiv \rightsquigarrow^* \nu\widetilde{b}.a\langle\widetilde{V}\rangle \mid P'$, *avec* $a \notin \widetilde{b}$
2. $P \equiv \rightsquigarrow^* \nu\widetilde{b}.v[a\langle\widetilde{P}\rangle \mid R] \mid P'$, *avec* $a \notin \widetilde{b}$
3. $P \equiv \rightsquigarrow^* \nu\widetilde{b}.a[P] \mid P'$, *avec* $a \notin \widetilde{b}$

Intuitivement, une barbe sur a signifie qu'après un nombre arbitraire d'étapes de sous-réductions, un processus P peut exhiber un message local (clause 1), un message vers le haut (clause 2), ou un message de type kell (clause 3). Ces observations sont similaires à celles trouvées par exemple dans les calculs à base d'ambiants [49].

2.5 Instances du kell calcul

2.5.1 Une instance simple : le πK-calcul

Nous avons vu que le Kell calcul est défini de manière abstraite relativement à un langage de motifs. La figure 2.11 définit un langage de motif, et donc une instance du Kell calcul, dont les motifs correspondent aux récepteurs du π-calcul polyadique. Le calcul résultant constitue un sur-ensemble du π-calcul asynchrone et polyadique, et en hérite donc l'expressivité.

Un motif peut être un *motif haut* $u^{\uparrow}\langle\widetilde{x}\rangle$, un *motif bas* $u^{\downarrow}\langle\widetilde{x}\rangle$, un *motif local* $u\langle\widetilde{x}\rangle$, ou un *motif de contrôle* $u[x]$. Les règles de réductions générales de la figure 2.10 peuvent se réécrire plus simplement sans prédicat de réduction. Nous les donnons en 2.12.

2.5.2 Motifs de synchronisation : le jK-calcul

Nous définissons maintenant un langage de motifs plus riche que le précédent, basé sur des *motifs de synchronisation*, inspirés du Join calcul [27]. Ces motifs permettent d'attendre la présence simultanée de plusieurs messages avant de libérer une continuation. Le cas particulier où un motif n'attend qu'un message correspond au langage défini en 2.5.2. Réciproquement, les motifs de synchronisation peuvent être implantés à l'aide de motifs

Motif

$$\xi ::= u^d\langle\widetilde{x}\rangle \quad | \quad u[x]$$
$$d ::= \uparrow \ | \ \downarrow \ |$$

Noms libres

$$\mathtt{fn}(a) = \{a\}$$
$$\mathtt{fn}(x) = \emptyset$$
$$\mathtt{fn}(u^d\langle\widetilde{x}\rangle) = \mathtt{fn}(u)$$
$$\mathtt{fn}(u[x]) = \mathtt{fn}(u)$$

Variables libres

$$\mathtt{fv}(a) = \emptyset$$
$$\mathtt{fv}(x) = \{x\}$$
$$\mathtt{fv}(u^d\langle\widetilde{x}\rangle) = \mathtt{fv}(u)$$
$$\mathtt{fv}(u[x]) = \mathtt{fv}(u)$$

Variables liées

$$\mathtt{bv}(u^d\langle\widetilde{x}\rangle) = \widetilde{x}$$
$$\mathtt{bv}(u[x]) = \{x\}$$

Filtrage

$$\mathtt{match}(u\langle\widetilde{x}\rangle, u\langle\widetilde{V}\rangle, \{\widetilde{V}/\widetilde{x}\})$$
$$\mathtt{match}(u^{\uparrow}\langle\widetilde{x}\rangle, u\langle\widetilde{V}\rangle, \{\widetilde{V}/\widetilde{x}\})$$
$$\mathtt{match}(u^{\downarrow}\langle\widetilde{x}\rangle, u\langle\widetilde{V}\rangle, \{\widetilde{V}/\widetilde{x}\})$$
$$\mathtt{match}(u[x], u[P], \{P/x\})$$

Substitution

$$\mathtt{subs}(u^d\langle\widetilde{x}\rangle, \sigma) = \sigma(u)^d\langle\widetilde{\sigma(x)}\rangle$$
$$\mathtt{subs}(u[x], \sigma) = \sigma(u)[\sigma(x)]$$

FIG. 2.11 – *Langage de motifs simple : πK*

$$\frac{}{u\langle\widetilde{V}\rangle \mid v[R \mid (u\langle\widetilde{x}\rangle^{\uparrow} \triangleright Q)] \to v[R \mid Q\{\widetilde{V}/\widetilde{x}\}]}\ \text{R.IN} \qquad \frac{}{u\langle\widetilde{V}\rangle \mid (u\langle\widetilde{x}\rangle \triangleright Q) \to Q\{\widetilde{V}/\widetilde{x}\}}\ \text{R.LOCAL}$$

$$\frac{}{v[u\langle\widetilde{V}\rangle \mid R] \mid u^{\downarrow}\langle\widetilde{x}\rangle \triangleright Q \to v[R] \mid Q\{\widetilde{V}/\widetilde{x}\}}\ \text{R.OUT} \qquad \frac{}{v[P_*] \mid (v[x] \triangleright Q) \to Q\{P_*/x\}}\ \text{R.PASS}$$

$$\frac{P \to Q}{\mathbf{E}\{P\} \to \mathbf{E}\{Q\}}\ \text{R.CONTEXT} \qquad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}\ \text{R.STRUCT}$$

$$\frac{P \rightsquigarrow^* P' \qquad P' \to Q}{P \to Q}\ \text{R.STRUCT.EXTR}$$

FIG. 2.12 – *Relation de réduction du πK-calcul*

$$\xi ::= J \quad | \quad u[x] \quad | \quad u[x] \mid J \qquad \text{motif}$$
$$J ::= u^d\langle\widetilde{\epsilon}\rangle \quad | \quad J \mid J \qquad \text{motif de synchronisation}$$
$$d ::= \uparrow \ | \ \downarrow \ | \ - \qquad \text{direction}$$
$$\epsilon ::= x \qquad \text{variable}$$
$$| \quad (u) \qquad \text{constante}$$

FIG. 2.13 – *Syntaxe des motifs du jK-calcul*

Noms libres

$$\begin{aligned}
\mathtt{fn}(a) &= \{a\} \\
\mathtt{fn}(x) &= \emptyset \\
\mathtt{fn}((u)) &= \mathtt{fn}(u) \\
\mathtt{fn}(u^d\langle\widetilde{\epsilon}\rangle) &= \mathtt{fn}(u) \cup \mathtt{fn}(\widetilde{\epsilon}) \\
\mathtt{fn}(u[x]) &= \mathtt{fn}(u) \\
\mathtt{fn}(J \mid J) &= \mathtt{fn}(J) \mid \mathtt{fn}(J)
\end{aligned}$$

Variables libres

$$\begin{aligned}
\mathtt{fn}(a) &= \emptyset \\
\mathtt{fv}(x) &= \{x\} \\
\mathtt{fv}((u)) &= \mathtt{fv}(u) \\
\mathtt{fv}(u^d\langle\widetilde{\epsilon}\rangle) &= \mathtt{fv}(u) \cup \mathtt{fv}(\widetilde{\epsilon}) \\
\mathtt{fv}(y[x]) &= \{y\} \\
\mathtt{fv}(J \mid J') &= \mathtt{fv}(J) \cup \mathtt{fv}(J')
\end{aligned}$$

Variables liées

$$\begin{aligned}
\mathtt{bv}(a) &= \emptyset \\
\mathtt{bv}(x) &= \{x\} \\
\mathtt{bv}((u)) &= \emptyset \\
\mathtt{bv}(u^d\langle\widetilde{\epsilon}\rangle) &= \mathtt{bv}(\widetilde{\epsilon}) \\
\mathtt{bv}(u[x]) &= \{x\} \\
\mathtt{bv}(J \mid J') &= \mathtt{bv}(J) \cup \mathtt{bv}(J')
\end{aligned}$$

FIG. 2.14 – *Noms libres, variables libres et liées de jK*

simples. Ils constituent une primitive élégante pour la programmation de synchronisation, et présenteront également un intérêt pour le système de type définie en 3.3.

On définit le prédicat $\mathtt{match}'$ de la manière suivante :

$$\begin{aligned}
&\mathtt{match}'(x,V,\{V/x\}) \\
&\mathtt{match}'((u),(u),\emptyset) \\
\mathtt{match}'(\epsilon_i,V_i,\theta_i) \implies &\mathtt{match}'(\widetilde{\epsilon},\widetilde{V},\theta_1 \uplus \ldots \uplus \theta_n)
\end{aligned}$$

Un motif peut être un *motif de synchronisation* J, ou un *motif de contrôle* de la forme $J \mid a[x]$ ou $a[x]$. On note $a\langle\widetilde{\epsilon}\rangle$ pour $a^-\langle\widetilde{\epsilon}\rangle$.

Les variables liées d'un motif sont les variables x n'apparaissant pas entre parenthèses. Nous supposons que ces variables sont linéaires, c'est à dire que chaque variable du motif n'a qu'une occurrence. Les identifiants apparaissant entre parenthèses agissent comme des constantes lors du filtrage des messages. Ils font partis des ensembles de variables et noms libres d'un motif.

2.6 Exemples

Nous donnons dans cette section des exemples de membranes programmables. Nous supposons que tous les messages *vers* le kell a ont la forme $\mathtt{rcv}\langle a,op,args\rangle$ et que tous les messages *provenant* du kell a ont la forme $\mathtt{snd}\langle dest,op,args\rangle$:

Membrane transparente Il s'agit d'une membrane qui ne fait rien, elle permet simplement aux messages destinés à a ou émis par a d'être transmis sans aucun contrôle :

$$M_{trans} \triangleq (\mathtt{rcv}\langle(a),x,y\rangle^{\uparrow} \diamond \mathtt{rcv}\langle a,x,y\rangle) \mid (\mathtt{snd}\langle x,y,z\rangle^{\downarrow} \diamond \mathtt{snd}\langle x,y,z\rangle)$$

$$\frac{(d,d') \in \{(\uparrow,\uparrow),(-,-),(\downarrow,\downarrow^b)\} \qquad \texttt{match}'(\widetilde{\epsilon},\widetilde{V},\theta)}{\texttt{match}(u^d\langle\widetilde{\epsilon}\rangle,u^{d'}\langle\widetilde{V}\rangle,\theta)} \qquad \frac{\texttt{match}(J,m,\theta) \qquad \texttt{match}(J',m',\theta')}{\texttt{match}(J \mid J',m \mid m',\theta \uplus \theta')}$$

$$\frac{\texttt{match}(J,m,\theta) \qquad m \equiv m'}{\texttt{match}(J,m',\theta)}$$

Substitution

$$\begin{aligned}
\texttt{subs}(a,\sigma) &= \sigma(a)\\
\texttt{subs}(x,\sigma) &= \sigma(x)\\
\texttt{subs}((x),\sigma) &= (\sigma(x))\\
\texttt{subs}(u^d\langle\widetilde{\epsilon}\rangle,\sigma) &= \sigma(u)^d\langle\widetilde{\sigma(\epsilon)}\rangle\\
\texttt{subs}(u[x],\sigma) &= \sigma(u)[\sigma(x)]\\
\texttt{subs}(J_1 \mid J_2) &= \texttt{subs}(J_1,\sigma) \mid \texttt{subs}(J_2,\sigma)
\end{aligned}$$

FIG. 2.15 – *Filtrage et substitutions dans jK*

Membrane d'interception Il s'agit d'une membrane qui déclenche le comportement $P(b,V)$ lorsqu'un message $\texttt{rcv}\langle a,b,V\rangle$ cherche a entrer dans le kell a, et qui déclenche le comportement $Q(b,c,V)$ lorsqu'un message $\texttt{snd}\langle b,c,V\rangle$ cherche à quitter le kell a. Notons que cela permet également la définition de wrapper permettant le pré-traitement et post-traitement des messages :

$$M_{int} \triangleq (\texttt{rcv}\langle(a),x,y\rangle^{\uparrow} \diamond P(x,y)) \mid (\texttt{snd}\langle x,y,z\rangle^{\downarrow} \diamond Q(x,y,z))$$

Membrane de migration Il s'agit d'une membrane qui permet à de nouveaux processus d'entrer dans le kell a via l'opération `enter`, et permet au kell a de se déplacer vers un kell différent b via l'opération `go`. On peut comparer ces opérations avec les primitives de migration asynchrone des ambiants et le primitive de migration du Join calcul réparti.

$$\begin{aligned}
M_{mig} \triangleq &(\texttt{rcv}\langle(a),(\texttt{enter}),x\rangle^{\uparrow} \diamond (a[y] \triangleright a[x \mid y]))\\
&\mid (\texttt{go}\langle b\rangle^{\downarrow} \diamond (a[y] \triangleright \texttt{snd}\langle b,\texttt{enter},a[y]\rangle))
\end{aligned}$$

Membrane simulant des pannes sans reprise Cette membrane permet d'arrêter via une commande `stop` l'exécution du kell a (simulant une panne dans un modèle de type panne sans reprise). De plus, elle implante un détecteur de panne simple via une commande `ping`. Ces opérations sont comparables aux modèles de pannes du π_{1l}-calcul [4], et du Join calcul réparti :

$$\begin{aligned}
M_{fails} \triangleq &\nu cf.(\texttt{rcv}\langle(a),(\texttt{stop}),x\rangle^{\uparrow} \mid c \diamond (a[y] \triangleright f))\\
&\mid (\texttt{rcv}\langle(a),(\texttt{ping}),x\rangle^{\uparrow} \mid c \diamond \texttt{snd}\langle x,\texttt{up},a\rangle \mid c)\\
&\mid (\texttt{rcv}\langle(a),(\texttt{ping}),x\rangle^{\uparrow} \mid f \diamond \texttt{snd}\langle x,\texttt{down},a\rangle \mid f)\\
&\mid c
\end{aligned}$$

Membrane simulant des pannes avec reprise Cette membrane enrichit la précédente en rajoutant la possibilité de reprise :

$$\begin{aligned} M_{failr} \triangleq & \nu cf.(\texttt{rcv}\langle(a),(\texttt{stop}),x\rangle^{\uparrow} \mid c \diamond (a[y] \triangleright f\langle y\rangle)) \\ & \mid (\texttt{rcv}\langle(a),(\texttt{ping}),r\rangle^{\uparrow} \mid c \diamond \texttt{snd}\langle r,\texttt{up},a\rangle \mid c) \\ & \mid (\texttt{rcv}\langle(a),(\texttt{ping}),r\rangle^{\uparrow} \mid f\langle y\rangle \diamond \texttt{snd}\langle r,\texttt{down},a\rangle \mid f\langle y\rangle) \\ & \mid (\texttt{rcv}\langle(a),(\texttt{recover}),r\rangle^{\uparrow} \mid f\langle y\rangle \diamond a[y] \mid c \mid \texttt{snd}\langle r,\texttt{rcvd},a\rangle) \\ & \mid c \end{aligned}$$

2.7 Travaux connexes

Le Kell calcul se place dans la catégorie des calculs de processus présentant une notion de localité. Une partie de ces calculs sont comparés dans [31] et également [17], [71]. Dans cette section, nous présentons brièvement certains calculs représentatifs, en les comparant au Kell calcul et en nous basant notamment sur les principes de conception mis en évidence en section 2.3. Nous les divisons en deux catégories. Les calculs basés sur les ambiants mobile, les calculs basés sur le π-calcul étendu par des localités.

2.7.1 Calculs à base d'ambiants

Le calcul des ambiants [14] a été l'un des premiers langage à mettre en avant les notions de localités hiérarchiques et de mobilité comme primitive pour la modélisation et programmation de systèmes à grande échelle. Il est basé sur les concepts de localité, mobilité, autorisation de déplacement et communications locales. Un ambiant peut modéliser par exemple un agent se déplaçant entre divers domaines d'administration, eux aussi modélisés par des ambiants. Il franchit au plus une frontière à la fois, à condition d'avoir obtenu une *capacité* lui permettant de le faire.

Un ambiant, comme un kell, est un processus de la forme $a[P]$ ou a est un nom, et P un processus. Les ambiants peuvent s'imbriquer arbitrairement pour décrire une structure d'arbre. Comme dans le Kell calcul, les processus peuvent être mis en parallèle, comme dans $P \mid Q$, créer des noms unique, $\nu a.P$, ou être inactifs, comme le processus $\mathbf{0}$. Contrairement au Kell calcul, les communications dans le calcul des ambiants sont anonymes et n'ont lieu que localement à un ambiant. Par exemple, le processus $\langle M\rangle \mid (x)P$ évolue vers $P\{M/x\}$. La hiérarchie d'ambiant peut être modifiée dynamiquement à l'aide de processus de la forme $M.P$ ou M est une capacité. Ces capacités sont de trois types : `in` permet à un ambiant d'entrer dans un ambiant adjacent, `out` permet à un ambiant de sortir de l'ambiant qui le contient, et `open` permet à un ambiant de dissoudre sa frontière. Ces différentes actions sont résumées par les trois règles suivantes.

$$a[\texttt{in}\ b.P \mid Q] \mid b[R] \rightarrow b[a[P \mid Q] \mid R] \qquad \text{In}$$

$$a[b[\texttt{out}\ a.P \mid Q] \mid R] \rightarrow b[P \mid Q] \mid a[R] \qquad \text{Out}$$

$$\texttt{open}a.P \mid a[Q] \rightarrow P \mid Q \qquad \text{Open}$$

Le calcul des ambiants présente plusieurs inconvénients. Tout d'abord, il ne respecte pas le principe d'action locale énoncé en section 2.3. Par conséquent, le calcul n'est implantable dans un contexte asynchrone qu'au prix de protocoles coûteux.

D'autres limitations du calcul ont conduit à la définition de différentes variantes, bien qu'aucune ne respecte le principe d'action locale. Nous en mentionnons deux. La première concerne les problèmes de sécurité posés par la règle `open` et la deuxième concerne une forme de non-déterminisme peu désirable.

La règle de dissolution d'un ambiant est nécessaire, par exemple, pour permettre les communications entre ambiants distants. Un ambiant est utilisé comme messager et migre de l'ambiant source vers l'ambiant destination. Arrivé à destination, il est dissout et son contenu peut alors interagir avec l'ambiant destination. Cette opération de dissolution pose de lourds problèmes de sécurité puisque il n'est pas possible de contrôler de manière fine les interactions entre l'ambiant destination et le messager. Soit ils sont isolés, soit l'ambiant messager est ouvert et peut interagir sans restriction avec son hôte. Le calcul des "Boxed Ambients"contrôle [11] propose une version du calcul où la primitive `open` est supprimée au profit de communications orientées permettant de franchir la frontière d'un ambiant, permettant un contrôle plus fin des interactions entre deux ambiants.

Le calcul des "Safe Ambients" [45] est un calcul des ambiants modifié afin d'éviter une forme de non déterminisme (interférence grave) que les auteurs jugent non souhaitable et assimilent à une erreur de programmation. Ils introduisent pour cela des co-capacités $\overline{\texttt{in}}$, $\overline{\texttt{out}}$ et $\overline{\texttt{open}}$, afin d'imposer un consentement mutuel des deux ambients impliqués par réduction. Le calcul résultant permet de limiter les interférences graves, possède une théorie algébrique plus riche et se révèle plus propice à l'analyse statique que le calcul des ambients original. De plus, l'implémentation se trouve simplifiée.

2.7.2 Calculs basés sur le π-calcul avec localités

Nous considérons maintenant des calculs basés sur le π-calcul auquel des localités, hiérarchiques ou non, ont été ajoutés. Nous les divisons en deux catégories, en fonction de leur façon de gérer la mobilité. Les premiers utilisent une primitive de migration, permettant à un programme passif ou actif de s'exécuter sur un autre site. Les autres gèrent la mobilité de manière différente, comme un type de communication particulier.

Primitive de migration Le Join calcul [26] est une extension du π-calcul destinée à permettre une implantation répartie efficace. Les récepteurs du Join calcul sont par construction uniques et répliquées, et utilisent des motifs de synchronisation similaires à ceux présentéss en section 2.5.2. Dans ce contexte, un nom est toujours associé de manière unique à un récepteur de telle sorte que les envois de messages peuvent être implantés par un envoi de message sur un réseau asynchrone.

Le Join calcul réparti ajoute des locations hiérarchiques au Join calcul simple, et présente une mobilité forte obtenue à l'aide d'une primitive de migration subjective qui permet de déplacer un sous-arbre de localités. Les locations dans le Join calcul réparti sont simplement des unités de migrations. En particulier, elles sont orthogonales aux communications et ne permettent pas d'exercer un contrôle sur les communications. Par ailleurs, l'unicité des récepteurs interdit la liaison dynamique. Un processus ne peut pas se lier à des ressources différentes en fonction de sa localisation. Le problème de la liaison dynamique dans le Join calcul est considéré dans [61].

On peut également citer plusieurs calculs basé sur le π-calcul avec un modèle de location plat (et qui ne vérifie donc pas les pré-requis identifié en 2.3) [62] [34] [4] [29], [33].

Autre traitement de la mobilité Le Seal calcul [16] propose un modèle de localité et de communication similaire à celui du Kell calcul. Contrairement au Kell calcul, le calcul n'est pas d'ordre supérieur et différencie les communications des actions de migration. Les migrations sont réalisées à l'aide d'une primitive, similaire à une communication, qui permet à la fois la migration, la duplication ou la destruction d'une localité. Tout comme le Kell calcul, le Seal calcul vérifie le principe d'action locale et le principe de médiation.

Cette primitive s'avère moins générale que la passivation dans le Kell calcul. Par exemple, elle ne permet pas de passiver, puis de réactiver ultérieurement un processus. Par ailleurs, un autre avantage de l'ordre supérieur est la possibilité d'étendre le calcul à la manière de MetaKlaim [23].

Le M-calcul [58] partage les principes de conception du Kell calcul. Les localités du M-calcul permettent de contrôler les communications a travers les frontières à l'aide de membranes. Des règles de routages implantent des communications mettant en jeu le franchissement de plusieurs frontières. Un opérateur de passivation en conjonction avec des communications d'ordre supérieur permet d'implanter la mobilité active. Le Kell calcul se présente comme un calcul plus simple et de plus bas niveau que le M-calcul. La notion de membrane et les règles de routage ne sont plus primitives mais peuvent être programmées.

Le calcul Homer [35] est très proche du Kell calcul, et en particulier utilise une forme de passivation identique. Les valeurs transmises sont limitées aux processus. En particulier, les transmissions de noms ne sont pas autorisées. Par ailleurs, les communications (et passivations) peuvent avoir lieu à une distance arbitraire dans une branche de l'arbre de localités. Cela limite les possibilités de contrôle. Par exemple, dès lors que le chemin entre deux localités et connu, il est possible d'accéder à toutes les ressources.

2.8 Perspectives

Le Kell calcul présente des aspects originaux qui permettent de rendre compte de manière simple du concept de composants logiciels répartis et hiérarchiques. Néanmoins, le modèle doit être confronté à la réalité des systèmes qu'il modélise, et est amené à évoluer. Nous y reviendrons dans les chapitres 4 et 5. Nous pouvons citer trois axes de recherche.

Un critère important pour évaluer un calcul de processus est qu'il dispose de relations d'équivalence satisfaisantes. Par exemple, on veut pouvoir définir formellement le fait pour deux processus d'agir de manière identique, et cela dans tous contextes. On veut alors disposer de techniques de preuves simple pour prouver de telles équivalence. On utilise généralement la congruence barbée faible comme définition d'équivalence, et l'on cherche à la caractériser de manière co-inductive. Ce résultat est réputée difficile pour des calculs d'ordre supérieur est n'a pas été obtenu pour le Kell calcul. Seule une caractérisation co-inductive de la congruence barbée forte est donnée dans [59].

Le Kell calcul a été récemment étendu pour permettre le partage de Kell [37]. Pour pouvoir modéliser le partage d'une librairie entre plusieurs composants, où d'une machine entre plusieurs réseaux, il est tentant de considérer un modèle où la relation d'appartenance est un graphe plutôt qu'un arbre. L'approche retenue dans [37] identifie un parent particulier d'un Kell partagé qui peut le contrôler, via des opérations de passivation. Ce modèle présente une faille de sécurité : la construction $\nu a.a[a[P]]$ représentant un pare-feu parfait dans le Kell calcul ne l'est plus dès lors que P peut partager des kells. Une

amélioration possible du modèle consiste à traiter ce problème.

Chapitre 3

Types

3.1 Introduction

Les systèmes que nous considérons cumulent des caractéristiques qui rendent leur comportement difficilement prédictible ou analysable. On peut citer par exemple, le parallèlisme, la mobilité ou encore la programmation par composants. Un autre nouveau défi concerne la sécurité. Dans un environnement à grande échelle, on doit pouvoir contrôler l'accès à certaines ressources à des programmes potentiellement malicieux. Les systèmes de types se sont avérés efficaces pour aborder ces problèmes et sont utilisés pour offrir statiquement diverses garanties sur le comportement des programmes.

Le Kell calcul est basé sur le π-calcul d'ordre supérieur avec localités et un opérateur de passivation. L'utilisation conjointe de ces différents aspects est nouvelle, mais nous pensons néanmoins qu'un certain nombre de techniques développées pour le π-calcul et le π-calcul d'ordre supérieur peuvent s'adapter dans notre contexte. Dans ce chapitre, nous proposons deux systèmes de types pour le Kell calcul.

Le premier est un système de type simple qui permet de limiter les opérations de passivation, et garantit que des kells spécifiés comme non-passivables ne seront jamais passivées. Cette propriété est utile en pratique. Du point de vue de la programmation, on peut ainsi utiliser des kells comme structures d'accueil pour des langages pour lesquels on n'est pas capable d'implanter l'opération de passivation. On obtient également un contrôle d'une forme basique de mobilité puisque la migration de processus se base sur la passivation. Finalement, cela permet également de modéliser des entités pour lesquelles la passivation n'a pas de sens (un réseau, une machine). Par ailleurs, une caractéristique originale de ce système de type est qu'il est indépendant du langage de motifs. Nous identifions une famille de langage de motifs pour lesquels les résultats de correction du système de type sont toujours vérifiés.

Le deuxième système de type, défini pour deux instances du Kell calcul, permet d'assurer l'unicité des noms de localité actives dans un système. Cette propriété est désirable pour modéliser par exemple des identifiants uniques, et est non-triviale à assurer dans le cadre du Kell calcul, où l'on peut créer dynamiquement des kells ou encore dupliquer des processus. Formellement, le système de type est polymorphe et permet de typer finement les processus à l'aide des noms de localités qu'ils peuvent contenir.

Le chapitre s'organise de la manière suivante. En section 3.2, nous présentons le sys-

tème de type générique pour le contrôle de la passivation. En section 3.3, nous présentons le système de type polymorphe, permet d'assurer l'unicité des noms de kells actifs. Finalement, en section 3.4 nous discutons différentes perspectives d'amélioration de ces travaux.

3.2 Types génériques

3.2.1 Cas général

Nous avons vu au chapitre 2 que certains programmes du Kell calcul étaient erronés, en raison d'une utilisation non uniforme de certains canaux de communication. Considérons par exemple les deux termes suivant du πK-calcul.

$$P_1 = a\langle \mathbf{0} \rangle \mid (a\langle x \rangle \triangleright x[\mathbf{0}])$$
$$P_2 = a\langle \mathbf{0},\mathbf{0} \rangle \mid (a\langle x \rangle \triangleright \mathbf{0})$$

Dans le terme P_1, le canal a est utilisé d'une part pour envoyer une valeur d'ordre supérieur, et d'autre part pour recevoir un nom. Ce terme est incorrect et en effet, on a $P_1 \rightarrow \perp$. Dans le terme P_2, le canal a est utilisé comme un canal d'arité deux d'une part, et un canal d'arité un d'autre part. Contrairement à P_1, le terme P_2 ne se réduit pas vers $\perp$. Il n'a simplement pas de réduction possible. Néanmoins, on souhaite également interdire ce cas de figure, qui s'apparente à une erreur de programmation.

De manière plus intéressante, nous souhaitons limiter les opérations de passivation. Considérons le programme suivant :

$$P = a[S \mid d[R] \mid b[Q]] \mid (a[x] \triangleright c\langle x \rangle)$$

Plusieurs raisons peuvent faire que nous voulons interdire la passivation de b (ou de a). Par exemple, b peut modéliser une entité physique pour laquelle l'idée d'interrompre et de sauvegarder l'état n'a pas de sens. Une autre possibilité est que P soit un programme écrit en Kell calcul, et que Q modélise une librairie, implantée par exemple en langage C. Si l'on n'est pas capable de sauvegarder l'état d'exécution de Q, essayer de passiver le kell b (est par conséquent, également a) est une erreur. Par contre, on doit pouvoir passiver d si l'on le désire.

Pour résoudre ces problèmes, nous suivons l'approche classique utilisée dans le π-calcul, qui consiste à assigner des types aux canaux de communication lors de leur création. Ces types caractérisent d'une part l'arité d'un canal, et d'autre part le type des valeurs transmises. Dans le cas du Kell calcul, on peut procéder de la même manière, en considérant un nom de kell comme un canal particulier qui communique des processus. Un processus peut-être de deux types différents. Le type `proc` correspond à un type de processus arbitraire. Le type `procP` correspond au cas particulier des processus que l'on sait passiver. La syntaxe des types est définie figure 3.1.

Un environnement de typage est une liste d'association entre des identifiants et des types. On suppose de plus tous les identifiants distincts et on identifie deux environnements qui ne diffèrent que par l'ordre de leurs éléments.

On définit de plus une relation de sous-typage $\leq$ sur les types généraux comme la plus petite relation transitive et réflexive telle que :

$$\texttt{procP} \leq \texttt{proc}$$

.

Types

$\gamma ::= \texttt{chan}\langle\widetilde{\tau}\rangle$	type de canal
$\lambda ::= \texttt{kell}\langle\pi\rangle$	type de kell
$\sigma ::= \gamma \mid \lambda$	type de canal ou de kell
$\pi ::= \texttt{proc} \mid \texttt{procP}$	type de processus
$\tau ::= \pi \mid \sigma \mid \texttt{base}$	type général

Environnements

$\Gamma ::= \emptyset \mid \Gamma,x : \tau \mid \Gamma,a : \sigma$

FIG. 3.1 – *Syntaxe des types simples*

Par ailleurs, on modifie légèrement la syntaxe du calcul en ajoutant une information de typage dans les créations de noms :

$$P ::= \ldots \mid \nu a : \sigma.P$$

On modifie de même la relation d'équivalence structurelle et les relations de réduction et sous-réduction.

La difficulté par rapport au π-calcul provient du fait que le Kell calcul est défini de manière générique, avec un langage de motif abstrait comme paramètre. Nous définissons une notion de motifs typés qui spécialise la définition 2.4.1, en y ajoutant notamment un prédicat `type` vérifiant cinq axiomes dont nous détaillerons la signification plus bas.

Nous aurons besoin des deux définitions supplémentaires. Un contexte $\mathbf{C}_m$ a l'une des formes suivantes :

- $\mathbf{C}_m = \mathbf{C}\{u^*\langle V_1, \ldots, \cdot, \ldots, V_n\rangle\}$
- $\mathbf{C}_m = \mathbf{C}\{u[\,\cdot\,]\}$

A un contexte $\mathbf{C}_m$, on associe un élément de la forme u^i_n où u correspond à l'identifiant du canal ou du kell juste au dessus de l'emplacement libre du contexte, n correspond à l'arité du canal (1 dans le cas d'un kell) et i est l'indice du canal correspondant à l'emplacement libre. On note `cn` la fonction qui a un contexte associe un tel élément.

Soit Γ un environnement de typage, on définit la fonction $\Gamma(u^i_n)$ de la manière suivante.

$$\begin{aligned}
&\Gamma(u^i_n) = \tau_i \text{ si } \Gamma(u) = \texttt{chan}\langle\tau_1, \ldots, \tau_n\rangle\\
&\Gamma(u^1_1) = \pi \text{ si } \Gamma(u) = \texttt{kell}\langle\pi\rangle\\
&\Gamma(u^i_n) = \bot \text{ dans les autres cas}
\end{aligned}$$

Intuitivement $\Gamma(u^i_n)$ correspond au type transporté en position i par le canal (ou kell) u, si ce canal est d'arité n. Si u n'appartient pas à Γ, où que l'arité du canal est différente, alors cette fonction est indéfinie.

Nous donnons maintenant la définition d'un langage de motif.

Définition 3.2.1 (Langage de motifs typé) *Un langage de motifs typé est une structure de la forme*

$$\mathsf{L} = (\mathsf{L}_s, \mathit{fn}, \mathit{fv}, \mathit{bv}, \mathit{match}, \mathit{subs}, \mathit{type})$$

tel que $(\mathsf{L}_s,\mathit{fn},\mathit{fv},\mathit{bv},\mathit{match},\mathit{subs})$ *est un langage de motif, et* $\mathit{type}(\Gamma,\xi,\Gamma')$ *une relation vérifiant les propriétés suivantes :*

$$\mathit{type}(\Gamma,\xi,\Gamma') \implies \mathit{type}((\Gamma,u:\tau),\xi,\Gamma') \tag{3.1}$$

$$\mathit{type}((\Gamma,u:\tau),\xi,\Gamma') \wedge u \notin \mathit{fn}(\xi) \cup \mathit{fv}(\xi) \implies \mathit{type}(\Gamma,\xi,\Gamma') \tag{3.2}$$

$$\mathit{type}((\Gamma,x:\tau),\xi,\Gamma') \wedge (u,\sigma) \in \Gamma \implies \mathit{type}(\Gamma,\xi\{u/x\},\Gamma') \tag{3.3}$$

$$\mathit{type}(\Gamma,\xi,\Gamma') \wedge u \in \mathit{fv}(\xi) \cup \mathit{fn}(\xi) \implies \exists\sigma.(u,\sigma) \in \Gamma \tag{3.4}$$

$$\begin{cases} \mathit{type}(\Gamma,\xi,\Gamma') \\ \mathit{match}(\xi,M,\theta) \end{cases} \implies \begin{cases} \mathit{dom}(\theta) = \mathit{dom}(\Gamma') \\ (M = \mathbf{C}_m\{\theta(x)\} \wedge \Gamma(\mathit{cn}(\mathbf{C}_m)) = \tau) \implies \Gamma'(x) = \tau \end{cases} \tag{3.5}$$

On définit alors une relation de typage de la forme $\Gamma \vdash P : \pi$, qui caractérise les processus P bien typés dans un environnement de typage Γ. Cette relation utilise la relation auxiliaire $\Gamma \vdash V : \tau$. Elles sont définies en figure 3.2.

$$\frac{}{\Gamma \vdash \mathbf{0} : \mathtt{proc}}\ \text{T.Nil} \qquad \frac{\Gamma \vdash V : \tau \qquad \tau \leq \tau'}{\Gamma \vdash V : \tau'}\ \text{T.Sub} \qquad \frac{}{\Gamma,u : \tau \vdash u : \tau}\ \text{T.Id}$$

$$\frac{\Gamma \vdash P_1 : \pi \qquad \Gamma \vdash P_2 : \pi}{\Gamma \vdash P_1 \mid P_2 : \pi}\ \text{T.Par} \qquad \frac{\Gamma,a : \sigma \vdash P : \pi}{\Gamma \vdash \nu a : \sigma.P : \pi}\ \text{T.Res}$$

$$\frac{\Gamma \vdash P : \pi \qquad \Gamma \vdash u : \mathtt{kell}\langle\pi\rangle}{\Gamma \vdash u[P] : \pi}\ \text{T.Kell} \qquad \frac{\Gamma \vdash u : \mathtt{chan}\langle\widetilde{\tau}\rangle \qquad \Gamma \vdash V_i : \tau_i}{\Gamma \vdash u\langle\widetilde{V}\rangle : \mathtt{procP}}\ \text{T.Msg}$$

$$\frac{\mathtt{type}(\Gamma,\xi,\Gamma') \qquad \Gamma,\Gamma' \vdash P : \pi \qquad (u,\mathtt{kell}\langle\mathtt{proc}\rangle) \in \Gamma \implies (u \notin \mathtt{fv}(\xi))}{\Gamma \vdash \xi \triangleright P : \pi}\ \text{T.Trig}$$

Fig. 3.2 – *Système de type simple*

La plupart des règles sont immédiates et correspondent directement au typage du π-calcul polyadique. On peut déduire facilement à partir de T.Par et T.Sub la règle suivante :

$$\frac{\Gamma \vdash P_1 : \pi_1 \qquad \Gamma \vdash P_2 : \pi_2}{\Gamma \vdash P_1 \mid P_2 : \mathtt{max}(\pi_1,\pi_2)}\ \text{T.Par'}$$

où $\mathtt{max}(\pi_1,\pi_2)$ est le plus grand élément de π_1 et π_2 (il est clair qu'un tel élément existe). En particulier, la mise en parallèle de deux processus est passivable si et seulement si les deux processus le sont. La règle T.Kell est similaire à la règle de typage d'un canal, et correspond à l'idée qu'un kell est un canal particulier. Par ailleurs, un kell a le même type que le processus qu'il contient. Il est passivable si et seulement si il contient un processus passivable. La seule règle originale est T.Trig. Elle permet de typer un processus de la forme $\xi \triangleright P$. L'idée est la suivante. Un motif ξ contient d'une part des identifiants libres et d'autre part des variables liées éventuellement utilisées par P. Les identifiants libres de ξ doivent être utilisés conformément à l'environnement Γ. De plus, on doit pouvoir déterminer le type des variables liées dans ξ, a partir de Γ et de ξ. Le prédicat $\mathtt{type}(\Gamma,\xi,\Gamma')$ remplit

ces deux rôles. Il permet d'assurer que le motif ξ est bien typé dans Γ, et d'extraire un environnement Γ' qui associe les variables liées de ξ à un type. Les différents axiomes de la définition assurent que le système de type résultant vérifie les propriétés de préservation de typage et de progrès. Les axiomes 3.1, 3.2, 3.3 sont indépendantes de l'environnement Γ' et correspondent à des propriétés structurelles de la relation de typage (précisément aux lemmes d'affaiblissement, de renforcement et de substitution). La propriété impose aux identifiants libres de ξ d'être présents dans ξ et d'avoir un type de canal ou de kell. Finalement, l'axiome 3.5 permet de prouver la préservation du typage. On peut le comprendre de cette manière : si ξ est un motif bien typé, si $\mathbf{C}_m$ est un contexte utilisé pour extraire une valeur $\theta(x)$ et que ce contexte est en certain un sens cohérent, avec Γ et l'emplacement libre de ce contexte a pour type τ, alors Γ' défini x avec comme type τ.

Pour finir, la contrainte :

$$(u,\mathtt{kell}\langle\mathtt{proc}\rangle) \in \Gamma \implies (u \notin \mathtt{fv}(\xi))$$

de la règle T.Trig empêche la passivation de kells contenant des processus généraux, non passivables a priori, en interdisant aux noms de ces kells d'être présent dans les motifs[1].

Théorème 1 (Préservation du typage) *Si $P \to P'$ et $\Gamma \vdash P : \mathtt{proc}$, alors $\Gamma \vdash P : \mathtt{proc}$.*

3.2.2 Application au πK-calcul

Pour le πK-calcul, on définit le prédicat `type` comme suit :

$$\begin{aligned}\mathtt{type}(\Gamma,u^d\langle x_1,\ldots,x_n\rangle,\Gamma') &\iff (u,\mathtt{chan}\langle\tau_1,\ldots,\tau_n\rangle) \in \Gamma \wedge \Gamma' = x_1:\tau_1,\ldots,x_n:\tau_n\\ \mathtt{type}(\Gamma,u[x],\Gamma') &\iff (u,\mathtt{kell}\langle\pi\rangle) \in \Gamma \wedge \Gamma' = x:\pi\end{aligned}$$

Proposition 3.2.2 *Le langage de motifs du πK-calcul muni du prédicat `type` est un langage de motifs typé.*

On peut remplacer la règle T.Trig par les règles T.Trig.Kell et T.Trig.Chan définies figure 3.3.

$$\frac{\Gamma \vdash u : \mathtt{chan}\langle\tau_1,\ldots,\tau_n\rangle \qquad \Gamma,x_1:\tau_1,\ldots,x_n:\tau_n \vdash P : \pi}{\Gamma \vdash u^d\langle x_1,\ldots,x_n\rangle \triangleright P : \pi}\ \text{T.Trig.Chan}$$

$$\frac{\Gamma \vdash u : \mathtt{kell}\langle\mathtt{procP}\rangle \qquad \Gamma,x : \mathtt{procP} \vdash P : \pi}{\Gamma \vdash u[x] \triangleright P : \pi}\ \text{T.Trig.Kell}$$

Fig. 3.3 – *Système de type générique instancié pour le πK-calcul*

1. cette condition est un peu trop restrictive, dans la mesure où la présence d'un tel nom libre dans un motif n'implique pas forcément une passivation. On pourrait préciser cette contrainte en ajoutant une information supplémentaire dans la définition d'un langage de motif

3.3 Unicité des cellules actives

3.3.1 Introduction

L'unicité des noms de cellule est une propriété à établir dans le but d'obtenir une implantation efficace du calcul. Par exemple, un kell a modélisant un site d'exécution interconnecté via un réseau à grande échelle comme Internet aura des récepteurs de la forme $\texttt{rcv}\langle(a),(b),\widetilde{x}\rangle \mid \ldots \triangleright P$ avec a correspondant par exemple à une adresse de réseau à grande échelle. Dans ce contexte, le nom a doit être unique, au moins dans le contexte englobant qui modélise le comportement du réseau. Toutefois, assurer l'unicité des noms de kells est difficile en présence de communication d'ordre supérieur et de passivation de kell. Par exemple, supposons que l'on a défini le récepteur $\texttt{twice}\langle x\rangle \triangleright x \mid x$. Un récepteur de la forme $a[x] \triangleright \texttt{twice}\langle a[x]\rangle$ conduirait à la duplication illicite du kell a.

Nous présentons dans cette sous-section un système de type qui impose l'unicité des noms de kell, et plus précisément, l'unicité des kells *actifs*. Nous définissons ce système de type pour les deux instances du Kell calcul définies dans le chapitre 2. Un kell $a[Q]$ est dit actif dans P (et on dit que P contient le kell actif a) si $P = \mathbf{E}\{a[Q]\}$ et a est libre dans P. L'idée générale, empruntée au système de type du M-calcul est de définir le type d'un processus P comme un multi-ensemble Δ qui représente une borne supérieure sur le multi-ensemble des noms de kells qui sont, ou pourront éventuellement devenir actifs dans P. Intuitivement, un tel processus sera donc bien typé si Δ est en fait un ensemble.

3.3.2 Types

Syntaxe des types Les types peuvent utiliser des noms, ainsi que deux classes de variables de types : des variables de type de nom, que l'on note δ, et des variables de multi-ensembles, que l'on note ρ. On désignera par Δ un multi-ensemble contenant des noms et des variables de type.

Un type τ peut être un *type de processus* Δ, un *type de nom de kell* (ou simplement un type de kell) $\texttt{kell}(w)_{\Delta\to\Delta'}$, un *type de canal* $\langle\widetilde{\tau}\rangle_\Delta$, où un *type de canal redéfinissable* $\langle\widetilde{\tau}\rangle^+_\Delta$. Nous notons $\langle\widetilde{\tau}\rangle^t_\Delta$ pour représenter un type pouvant être soit un type de canal soit un type de canal redéfinissable

Un processus a pour type Δ si au cours de son exécution, il n'utilisera qu'au plus les kells dont les noms sont spécifiés dans Δ. Notons que Δ est un multi-ensemble et comptabilise le nombre d'occurrence de kell actifs. Un canal de type $\langle\widetilde{\tau}\rangle_\Delta$ peut transporter des tuples de valeurs de type τ_i. De plus, la réception de valeurs sur ce canal peut entraîner la création de kells dont les noms sont dans Δ. On utilisera des types de canaux redéfinissables $\langle\widetilde{\tau}\rangle^+_\Delta$ pour typer des variables de récepteur pouvant être instanciées par un nom venant d'être reçu.

Dans le kell calcul, les kells peuvent être vus comme des canaux transportant des processus actifs. Ainsi, le type d'un nom de kell prend la forme $\texttt{kell}(w)_{\Delta\to\Delta'}$ où Δ est le type du processus s'exécutant dans le kell, et Δ' le multi-ensemble des noms pouvant être crées après une passivation.

Comme de plus les noms de kell sont également des variables, on doit également autoriser des variables de types de nom δ comme argument des types de nom de kell. Remarquons par ailleurs qu'un nom de kell ne peut avoir le type $\texttt{kell}(\emptyset)_{\Delta\to\Delta'}$ (ces types sont introduits pour des raisons techniques).

On utilise $\forall\widetilde{\rho}\widetilde{\delta}.\tau$ pour représenter un schéma de type dans lequel les variables de type de nom $\widetilde{\delta}$ et les variables de multi-ensemble $\widetilde{\rho}$ sont généralisées dans τ.

On utilise Γ et ses variantes décorées pour représenter des environnements de typages, *i.e.* des listes d'associations entre des identifiants et des schémas de types.

La syntaxe des types, schémas et environnement est donnée figure 3.4.

Types

$\tau ::= \sigma \mid \Delta$	valeur
$\sigma ::= \lambda \mid \gamma$	nom
$\lambda ::= \texttt{kell}(w)_{\Delta\to\Delta'}$	kell
$\gamma ::= \langle\widetilde{\tau}\rangle^t_\Delta$	canal
$\beta ::= \rho \mid \delta$	variable de type
$\Delta ::= \emptyset \mid w,\Delta$	processus
$s ::= \forall\widetilde{\beta}.\tau$	schéma
$v ::= a \mid \beta$	nom ou variable de type
$w ::= a \mid \delta \mid \emptyset$	
$t ::= + \mid -$	

Environnements

$\Gamma ::= \emptyset \mid \Gamma,u : s$

FIG. 3.4 – *Syntaxe des types pour l'unicité des kells*

Multi-ensemble Les multi-ensembles Δ peuvent inclure des noms a, des variables de type de nom δ, et des variables de multi-ensembles ρ. Nous utilisons plusieurs opérations sur les multi-ensembles. La relation $\subseteq$ est l'inclusion standard de multi-ensembles. Δ,Δ' est l'union des multi-ensembles Δ et Δ'. Le multi-ensemble $\Delta \setminus a$ est le multi-ensemble Δ moins une seule occurrence du nom a. $\Delta \sqcup \Delta'$ est le plus petit multi-ensemble (en termes d'inclusion) qui contient à la fois Δ et Δ'.

Instanciation Une *instanciation de variables de type* est une fonction θ qui a une variable de type de nom associe un nom ou une variable de type de nom, et à une variable de multi-ensemble, un multi-ensemble ou une variable de multi-ensemble. On suppose de plus qu'une instanciation est égale à l'identité sauf sur un ensemble fini de variable.

Noms et variables libres On définit figure 3.5 l'ensemble des noms libres et des variables de types libres d'un type.

Calcul typé Pour définir ce système de type, on considère une syntaxe étendue pour le calcul où les opérations de création de nom sont annotées avec leur schéma de type. Ainsi on écrit $\nu a : s.P$ à la place de $\nu a.P$, où s est un schéma de type.

La notion de nom libre est modifiée pour prendre en compte la nouvelle syntaxe :

$$\texttt{fn}(\nu a : s.P) = (\texttt{fn}(P) \cup \texttt{fn}(s)) \setminus \{a\}$$

$$\begin{aligned}
\mathtt{fn}(\emptyset) &= \emptyset & \mathtt{ftv}(\emptyset) &= \emptyset \\
\mathtt{fn}(\rho) &= \emptyset & \mathtt{ftv}(\rho) &= \{\rho\} \\
\mathtt{fn}(\delta) &= \emptyset & \mathtt{ftv}(\delta) &= \{\delta\} \\
\mathtt{fn}(a) &= a & \mathtt{ftv}(a) &= \emptyset \\
\mathtt{fn}(\Delta,\Delta') &= \mathtt{fn}(\Delta) \cup \mathtt{fn}(\Delta') & \mathtt{ftv}(\Delta,\Delta') &= \mathtt{ftv}(\Delta) \cup \mathtt{ftv}(\Delta') \\
\mathtt{fn}(\mathtt{kell}(w)_{\Delta\to\Delta'}) &= \mathtt{fn}(w,\Delta,\Delta') & \mathtt{ftv}(\mathtt{kell}(w)_{\Delta\to\Delta'}) &= \mathtt{ftv}(w,\Delta,\Delta') \\
\mathtt{fn}(\langle\widetilde{\tau}\rangle^t_\Delta) &= \mathtt{fn}(\tau_1) \cup \ldots \cup \mathtt{fn}(\tau_n) \cup \mathtt{fn}(\Delta) & \mathtt{ftv}(\langle\widetilde{\tau}\rangle^t_\Delta) &= \mathtt{ftv}(\tau_1) \cup \ldots \cup \mathtt{ftv}(\tau_n) \cup \mathtt{ftv}(\Delta) \\
\mathtt{fn}(\forall\widetilde{\beta}.\tau) &= \mathtt{fn}(\tau) & \mathtt{ftv}(\forall\widetilde{\beta}.\tau) &= \mathtt{ftv}(\sigma) \setminus \widetilde{\beta} \\
\mathtt{fn}(\Gamma) &= \bigcup_{x\in\mathtt{dom}(\Gamma)} \mathtt{fn}(\Gamma(x)) & \mathtt{ftv}(\Gamma) &= \bigcup_{x\in\mathtt{dom}(\Gamma)} \mathtt{ftv}(\Gamma(x))
\end{aligned}$$

FIG. 3.5 – *Noms et variables libres de type et d'environnement*

Les règles de congruence structurelle S.NU.NIL, S.NU.COMM et S.NU.PAR sont modifiées comme suit :

$$\begin{aligned}
\text{S.NU.NIL}\quad & \nu a : s.\mathbf{0} \equiv \mathbf{0} \\
\text{S.NU.COMM}\quad & \nu a : s.\nu b : s'.P \equiv \nu b : s'.\nu a : s.P \text{ si } \ a \notin \mathtt{fn}(s') \text{ et } \ b \notin \mathtt{fn}(s) \\
\text{S.NU.PAR}\quad & \nu a : s.P \mid Q \equiv (\nu a : s.P) \mid Q \ \text{ si } \ a \notin \mathtt{fn}(Q)
\end{aligned}$$

La règle SR.KELL est également modifiée :

$$\frac{u \neq a}{u[\nu a : \sigma.P] \rightsquigarrow \nu a : \sigma.u[P]}\ \text{SR.KELL}$$

Environnements, types et instanciations bien formés Les types contiennent des noms qui représentent des noms de kell. Par conséquent, les types se définissent relativement à un environnement de typage. Pour qu'un type ait un sens, les noms qu'il utilise doivent être déclarés dans un environnement de typage, et être associés à un type de kell. De la même façon, lorsqu'un type est utilisé dans un environnement, les noms qu'il utilise doivent être déclarés avant (rappelons que les environnements sont des listes). Nous suivons l'approche de [69], et définissons deux jugements de bonne formation pour les types et les environnements. On dit qu'un environnement Γ est bien formé s'il vérifie $\Gamma \vdash \mathtt{Env}$. Un type τ est bien formé s'il vérifie $\Gamma \vdash \tau : \mathtt{tp}$. Ces deux prédicats sont définis figure 3.6. La définition de $\Gamma \vdash \mathtt{Env}$ est immédiate. L'environnement vide est bien formé (règle ENV.VIDE), et si Γ est bien formé, que u est un identifiant n'appartenant pas au domaine de Γ, et que τ est bien formé dans Γ, alors l'environnement $\Gamma,u : \tau$ est bien formé.

Sous-typage On définit une relation de sous-typage relativement à un environnement de typage. On dit que τ est un sous type de τ' relativement à un environnement Γ, et l'on note $\Gamma \vdash \tau \leq \tau'$. L'intuition derrière cette relation de sous-typage est qu'il est sûr (en

Environnement bien formé

$$\frac{}{\emptyset \vdash \texttt{Env}}\ \textsc{Env.Vide} \qquad \frac{\Gamma \vdash s : \texttt{tp} \qquad u \notin \texttt{dom}(\Gamma)}{\Gamma,u : s \vdash \texttt{Env}}\ \textsc{Env.Id}$$

$$\frac{s = \forall\widetilde{\beta}.\lambda \qquad \Gamma \vdash s\{\emptyset/a\} : \texttt{tp} \qquad a \notin \texttt{dom}(\Gamma)}{\Gamma,a : s \vdash \texttt{Env}}\ \textsc{Env.Kell}$$

Type bien formé

$$\frac{\Gamma \vdash \texttt{Env} \qquad a : \forall\widetilde{\beta}.\lambda \in \Gamma}{\Gamma \vdash a : \texttt{tp}}\ \textsc{Tp.Proc.Nom} \qquad \frac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash \beta : \texttt{tp}}\ \textsc{Tp.Proc.Var}$$

$$\frac{\Gamma \vdash \texttt{Env} \qquad \forall v \in \Delta.\Gamma \vdash v : \texttt{tp}}{\Gamma \vdash \Delta : \texttt{tp}}\ \textsc{Tp.Set} \qquad \frac{\Gamma \vdash \Delta,\Delta',w : \texttt{tp}}{\Gamma \vdash \forall\widetilde{\beta}.\texttt{kell}(w)_{\Delta\to\Delta'} : \texttt{tp}}\ \textsc{Tp.Kell}$$

$$\frac{\Gamma \vdash \tau_i : \texttt{tp} \qquad \Gamma \vdash \Delta : \texttt{tp}}{\Gamma \vdash \forall\widetilde{\beta}.\langle\widetilde{\tau}\rangle_\Delta : \texttt{tp}}\ \textsc{Tp.Chan} \qquad \frac{\Gamma \vdash \tau_i : \texttt{tp} \qquad \Gamma \vdash \Delta : \texttt{tp}}{\Gamma \vdash \langle\widetilde{\tau}\rangle^+_\Delta : \texttt{tp}}\ \textsc{Tp.Chan.Tran}$$

Instanciation

$$\frac{\Gamma \vdash \texttt{Env} \qquad \forall\beta \in \texttt{dom}(\theta).\Gamma \vdash \theta(\beta) : \texttt{tp}}{\Gamma \vdash \theta : \texttt{inst}}\ \textsc{Inst}$$

FIG. 3.6 – *Schémas de types et environnements bien formés*

$$\frac{\Gamma \vdash \alpha : \texttt{tp}}{\Gamma \vdash \alpha \leq \alpha}\ \textsc{S.Refl} \qquad \frac{\Gamma \vdash \Delta,\Delta' : \texttt{tp} \qquad \Delta \subseteq \Delta'}{\Gamma \vdash \Delta \leq \Delta'}\ \textsc{S.Proc}$$

$$\frac{\Gamma \vdash w : \texttt{tp} \qquad \Gamma \vdash \Delta'_1 \leq \Delta_1 \qquad \Gamma \vdash \Delta_2 \leq \Delta'_2}{\Gamma \vdash \texttt{kell}(w)_{\Delta_1\to\Delta_2} \leq \texttt{kell}(w)_{\Delta'_1\to\Delta'_2}}\ \textsc{S.Kell}$$

$$\frac{\Gamma \vdash \Delta \leq \Delta' \qquad \Gamma \vdash \tau'_i \leq \tau_i}{\Gamma \vdash \langle\widetilde{\tau}\rangle_\Delta \leq \langle\widetilde{\tau'}\rangle_{\Delta'}}\ \textsc{S.Chan}$$

FIG. 3.7 – *Sous-typage*

terme d'unicité de noms de kell) de remplacer un processus avec un processus qui contient moins de kell actifs. La relation de sous-typage est définie figure 3.7.

Dans la suite, on utilise la notation $\Gamma \vdash J$ pour représenter indifféremment l'un des deux jugements de bonne formation ou de sous-typage.

Jugements de typage Les jugements de typage prennent la forme générale $\Gamma \vdash V : \tau$, où Γ est un environnement, V est une valeur et τ est un type. Cependant, selon que V est un processus ou un identifiant représentant un kell ou un canal, ce jugement prenant l'une des deux formes :

$$\Gamma \vdash u : \sigma$$
$$\Gamma \vdash P : \Delta$$

Nous détaillons maintenant les différentes règles d'inférence qui définissent ce jugement. Tout d'abord, une règle de sous-typage exprime le fait que si une valeur V a pour type τ et que τ est un sous-type de τ', alors V a également pour type τ' :

$$\frac{\Gamma \vdash V : \tau \qquad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash V : \tau'} \text{ T.Sub}$$

Le processus nul $\mathbf{0}$ ne définit pas de kell, et par conséquent il est typé par $\emptyset$. La prémisse oblige l'environnement de typage a être bien formé. Le jugement de typage est défini de telle sorte que les types et environnements utilisés sont toujours bien formés :

$$\frac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash \mathbf{0} : \emptyset} \text{ T.Nil}$$

Lorsque deux processus P_1 et P_2 sont mis en parallèle, les kells actifs qu'ils peuvent utiliser s'additionnent. Au niveau de leur type, cela correspond à une concaténation de multi-ensembles. Ce que traduit la règle T.Par :

$$\frac{\Gamma \vdash P_1 : \Delta_1 \qquad \Gamma \vdash P_2 : \Delta_2}{\Gamma \vdash P_1 \mid P_2 : \Delta_1,\Delta_2} \text{ T.Par}$$

Le système de type utilise un polymorphisme similaire à celui de ML. Les identifiants dans l'environnement de typage sont associés à des schémas de type $\forall\widetilde{\beta}.\tau'$. Cela signifie que l'on peut utiliser ces identifiants à un type ou les variables de types abstraites $\widetilde{\beta}$ ont été instanciés. L'instanciation θ des variables de types doit vérifier une contrainte de bonne formation $\Gamma \vdash \theta : \texttt{inst}$, définie de telle sorte que le type $\tau'\theta$ est bien formé dans Γ. La règle T.Id est donnée par :

$$\frac{(u,\forall\widetilde{\beta}.\tau') \in \Gamma \qquad \Gamma \vdash \theta : \texttt{inst} \qquad \tau = \tau'\theta}{\Gamma \vdash u : \tau} \text{ T.Id}$$

On définit deux jugements pour typer un processus de la forme $\nu a : s.P$ en fonction de la forme de s. Si $s = \sigma$, a est un canal. Les kells actifs utilisés par P et $\nu a : s.P$ sont identiques. Par conséquent, si P a pour type Δ dans Γ,$a : s$ alors le processus $\nu a : s.P$ a

le même type dans Γ. Par ailleurs, la condition $\texttt{ftv}(s) = \emptyset$ impose à s de ne pas avoir de variables de types libres :

$$\frac{s = \forall\widetilde{\beta}.\gamma \qquad \texttt{ftv}(s) = \emptyset \qquad \Gamma,\, a : s \vdash P : \Delta}{\Gamma \vdash \nu a : s.P : \Delta}\ \text{T.Res.C}$$

Si $s = \gamma$ est un type de kell, la règle de typage est légèrement différente. P a pour type $\Delta \uplus \{a\}$ dans Γ. En effet, puisque a est un nom de kell, P peut définir des kells de cette forme et cela doit apparaître dans son type. De plus, pour assurer l'unicité des kells, on impose par l'union disjointe que a n'a pas d'autre occurrence dans Δ. Par ailleurs, on utilise le prédicat suivant :

$$\texttt{Ktype}(\lambda,a) \iff \lambda = \texttt{kell}(a)_{\rho\to\Delta} \wedge \rho \notin \Delta \setminus \rho$$

$$\frac{s = \forall\widetilde{\beta}.\lambda \qquad \texttt{ftv}(s) = \emptyset \qquad \Gamma,\, a : s \vdash P : \Delta \uplus \{a\} \qquad \texttt{Ktype}(\lambda,a)}{\Gamma \vdash \nu a : s.P : \Delta}\ \text{T.Res.K}$$

La règle de typage correspondant à un envoi de message ne présente pas de difficulté. Les valeurs V_i transportées par le message $u\langle\widetilde{V}\rangle$, doivent avoir pour type τ_i, où $\widetilde{\tau}$ est spécifié comme type de u. De plus, $u\langle\widetilde{V}\rangle$ après réception peut conduire à la création des kells dont les noms sont dans Δ, donc $u\langle\widetilde{V}\rangle$ a pour type Δ :

$$\frac{\Gamma \vdash u : \langle\widetilde{\tau}\rangle^t_\Delta \qquad \Gamma \vdash V_i : \tau_i}{\Gamma \vdash u\langle\widetilde{V}\rangle : \Delta}\ \text{T.Msg}$$

La règle de typage d'un kell est similaire à celle d'un message. La différence concerne le type attribué à $u[P]$. Si P a pour type Δ, et u pour type $\texttt{kell}(w)_{\Delta\to\Delta'}$, alors le processus $u[P]$ définit au plus les kells dans w,Δ. Si ce processus est passivé, il définira au plus les kells dons les noms sont dans Δ'. Par conséquent, on attribue à $u[P]$ le type $(w,\, \Delta) \sqcup \Delta'$. La règle T.Kell est donnée par :

$$\frac{\Gamma \vdash u : \texttt{kell}(w)_{\Delta\to\Delta'} \qquad \Gamma \vdash P : \Delta}{\Gamma \vdash u[P] : (w,\, \Delta) \sqcup \Delta'}\ \text{T.Kell}$$

Ces différentes règles, indépendantes du langage de motif sont résumées figure 3.8. Nous donnons maintenant les règles de typage des récepteurs pour le πK-calcul.

Pour typer un récepteur de la forme $u\langle\widetilde{x}\rangle \triangleright P$, on doit prendre en compte le caractère polymorphe des canaux de communication. Le typage de P doit se faire indépendamment de toute instanciation possible de variables de types dans le schéma de type associé à u. Si $(u,\forall\widetilde{\beta}.\langle\widetilde{\tau}\rangle^t_\Delta) \in \Gamma$, on suppose que les variable de type β ne sont pas présentes dans l'environnement (condition habituelle). Si $\Gamma,\widetilde{x} : \widetilde{\tau} \vdash P : \Delta$, alors $\Gamma \vdash u\langle\widetilde{x}\rangle \triangleright P : \emptyset$. De plus, si $u \in \textsc{Vars}$, u doit être de type canal redéfinissable :

$$\frac{\begin{array}{c}(u,\forall\widetilde{\beta}.\langle\widetilde{\tau}\rangle^t_\Delta) \in \Gamma \\ u \in \textsc{Vars} \implies t = + \qquad \texttt{ftv}(\Gamma) \cap \widetilde{\beta} = \emptyset \qquad \Gamma,\widetilde{x} : \widetilde{\tau} \vdash P : \Delta\end{array}}{\Gamma \vdash u\langle\widetilde{x}\rangle \triangleright P : \emptyset}\ \text{T.Trig.Msg}$$

La règle de typage d'un récepteur de la forme $u[x] \triangleright P$ est similaire à la précédente. Par contre, u est nécessairement un nom.

$$\frac{(a,\forall\widetilde{\beta}.\texttt{kell}(w)_{\Delta\to\Delta'}) \in \Gamma \qquad \texttt{ftv}(\Gamma) \cap \widetilde{\beta} = \emptyset \qquad \Gamma,x : \Delta \vdash P : \Delta'}{\Gamma \vdash a[x] \triangleright P : \emptyset}\ \text{T.Trig.Pass}$$

Un point important à noter est que dans ces deux règles, le type du processus $\xi \triangleright P$ est l'ensemble vide. En effet, les kells crées sont comptabilisés dans les types des messages et des kells (règles T.Msg et T.Kell).

Mise à part la lourdeur des notations, les règles de typage du jK-calcul ne sont que légèrement plus complexes. Les récepteurs peuvent utiliser des motifs de synchronisation, ainsi que des constantes dans les motifs. Nous examinons quelques cas particuliers avant de donner le cas général. Tout d'abord, considérons un motif de synchronisation sans constante. Le point important est que maintenant le multi-ensemble Δ qui type P dans les prémisses de la règle est obtenu comme la concaténation des Δ_i provenant des types de chacun des récepteurs du motif. De plus, les variables de types doivent être choisies différentes les unes des autres, et différentes de celles présentes dans l'environnement de typage. Les règles spécifiques au jK-calcul sont données figure 3.10.

$$\frac{\Gamma \vdash V : \tau \qquad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash V : \tau'}\ \text{T.Sub} \qquad \frac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash \mathbf{0} : \emptyset}\ \text{T.Nil}$$

$$\frac{\Gamma \vdash P_1 : \Delta_1 \qquad \Gamma \vdash P_2 : \Delta_2}{\Gamma \vdash P_1 \mid P_2 : \Delta_1,\Delta_2}\ \text{T.Par} \qquad \frac{(u,\forall\widetilde{\beta}.\tau') \in \Gamma \qquad \Gamma \vdash \theta : \texttt{inst} \qquad \tau = \tau'\theta}{\Gamma \vdash u : \tau}\ \text{T.Id}$$

$$\frac{s = \forall\widetilde{\beta}.\gamma \qquad \texttt{ftv}(s) = \emptyset \qquad \Gamma,\, a : s \vdash P : \Delta}{\Gamma \vdash \nu a : s.P : \Delta}\ \text{T.Res.C}$$

$$\frac{s = \forall\widetilde{\beta}.\lambda \qquad \texttt{ftv}(s) = \emptyset \qquad \Gamma,\, a : s \vdash P : \Delta \uplus \{a\} \qquad \texttt{Ktype}(\lambda,a)}{\Gamma \vdash \nu a : s.P : \Delta}\ \text{T.Res.K}$$

$$\frac{\Gamma \vdash u : \texttt{kell}(w)_{\Delta\to\Delta'} \qquad \Gamma \vdash P : \Delta}{\Gamma \vdash u[P] : (w,\, \Delta) \sqcup \Delta'}\ \text{T.Kell} \qquad \frac{\Gamma \vdash u : \langle\widetilde{\tau}\rangle^t_\Delta \qquad \Gamma \vdash V_i : \tau_i}{\Gamma \vdash u\langle\widetilde{V}\rangle : \Delta}\ \text{T.Msg}$$

Fig. 3.8 – *Règles de typage communes*

3.3.3 Correction du système de type

La correction du système de type est caractérisée par les définitions et théorèmes suivants.

$$\frac{\begin{array}{c}(u,\forall\widetilde{\beta}.\langle\widetilde{\tau}\rangle^{t}_{\Delta}) \in \Gamma \\ u \in \textsc{Vars} \implies t = + \qquad \mathtt{ftv}(\Gamma) \cap \widetilde{\beta} = \emptyset \qquad \Gamma,\widetilde{x} : \widetilde{\tau} \vdash P : \Delta\end{array}}{\Gamma \vdash u\langle\widetilde{x}\rangle \triangleright P : \emptyset}\ \textsc{T.Trig.Msg}$$

$$\frac{(a,\forall\widetilde{\beta}.\mathtt{kell}(w)_{\Delta\to\Delta'}) \in \Gamma \qquad \mathtt{ftv}(\Gamma) \cap \widetilde{\beta} = \emptyset \qquad \Gamma,x : \Delta \vdash P : \Delta'}{\Gamma \vdash a[x] \triangleright P : \emptyset}\ \textsc{T.Trig.Pass}$$

FIG. 3.9 – *Règles spécifiques au πK-calcul*

$$\frac{\begin{array}{c}(u_i,\forall\widetilde{\beta_i}.\langle\tau_i^{1..m_i}\rangle^{t_i}_{\Delta_i}) \in \Gamma \qquad u_i \in \textsc{Vars} \implies t_i = + \qquad \mathtt{ftv}(\Gamma) \cap \widetilde{\beta_i} = \emptyset \\ i \neq j \implies \mathtt{fn}(\widetilde{\beta_i}) \cap \mathtt{fn}(\widetilde{\beta_j}) = \emptyset \qquad \Gamma,x_{ij} : \tau_{ij} \vdash P : \Delta_1,\ldots,\Delta_n\end{array}}{\Gamma \vdash u_1\langle x_1^{1..m_1}\rangle \mid \ldots \mid u_n\langle x_n^{1..m_n}\rangle \triangleright P : \emptyset}\ \textsc{T.Trig.Msg}$$

$$\frac{\begin{array}{c}(a,\forall\widetilde{\beta}.\mathtt{kell}(w)_{\Delta\to\Delta_0}) \in \Gamma \\ (u_i,\forall\widetilde{\beta_i}.\langle\tau_i^{1..m_i}\rangle^{t_i}_{\Delta_i}) \in \Gamma \qquad u_i \in \textsc{Vars} \implies t_i = + \qquad \mathtt{ftv}(\Gamma) \cap \widetilde{\beta_i} = \emptyset \\ i \neq j \implies \mathtt{fn}(\widetilde{\beta_i}) \cap \mathtt{fn}(\widetilde{\beta_j}) = \emptyset \qquad \Gamma,x : \Delta,x_{ij} : \tau_{ij} \vdash P : \Delta_0,\ldots,\Delta_n\end{array}}{\Gamma \vdash a[x] \mid u_1\langle x_1^{1..m_1}\rangle \mid \ldots \mid u_n\langle x_n^{1..m_n}\rangle \triangleright P : \emptyset}\ \textsc{T.Trig.Pass}$$

FIG. 3.10 – *Règles spécifiques au jK-calcul*

On définit par induction la fonction partielle $\texttt{kellname}$.

$$\begin{aligned}
\texttt{kellname}(\mathbf{0}) &= \emptyset \\
\texttt{kellname}(x) &= \emptyset) \\
\texttt{kellname}(u\langle \widetilde{V} \rangle) &= \emptyset \\
\texttt{kellname}(\xi \triangleright P) &= \emptyset \\
\texttt{kellname}(\nu a : \forall \widetilde{\beta}.\lambda. \triangleright P) &= \begin{cases} \text{si } \texttt{kellname}(P) = \bot, \bot \\ \texttt{kellname}(P) \setminus \{a\} \end{cases} \\
\texttt{kellname}(\nu a : \forall \widetilde{\beta}.\gamma. \triangleright P) &= \texttt{kellname}(P) \\
\texttt{kellname}(P \mid Q) &= \begin{cases} \text{si } \texttt{kellname}(P) \cap \texttt{kellname}(Q) = \emptyset, \bot \\ \texttt{kellname}(P) \cup \texttt{kellname}(Q) \end{cases} \\
\texttt{kellname}(u[P]) &= \begin{cases} \text{si } u \in \textsc{Vars}, \bot \\ \{u\} \cup \texttt{kellname}(P) \end{cases}
\end{aligned}$$

Définition 3.3.1 (Processus erroné) *Un processus P sans variable libre est* erroné *si $\texttt{kellname}(P) = \bot$.*

Théorème 2 (Progrès) *Si $\Gamma \vdash P : \Delta$, avec P un processus clos et Δ est un ensemble de noms, alors le processus P n'est pas erroné.*

Proposition 3.3.2 (Préservation de la congruence structurelle) *Si on a $\Gamma \vdash P : \tau$ et $P \equiv Q$, alors $\Gamma \vdash Q : \tau$.*

Définition 3.3.3 *On dit qu'un environnement de typage Γ est* bon *s'il est bien formé, et si*

$$(a, \forall \widetilde{\beta}.\lambda) \in \Gamma \implies \texttt{Ktype}(\forall \widetilde{\beta}.\lambda, w)$$

Proposition 3.3.4 (Préservation de la sous-réduction) *Si Γ est un bon environnement, $\Gamma \vdash P : \tau$ et $P \rightsquigarrow Q$, alors $\Gamma \vdash Q : \tau$.*

Théorème 3 (Préservation de la réduction) *Si Γ est un bon environnement, $\Gamma \vdash P : \tau$ et $P \to Q$, alors $\Gamma \vdash Q : \tau$.*

3.3.4 Exemples

Exemple simple Considérons tout d'abord un processus simple dans lequel les continuations des récepteurs ne définissent pas de kell.

$$P = a[b\langle a \rangle] \mid b^{\downarrow}\langle x \rangle \triangleright c\langle x \rangle$$

Le kell a communique son nom à un service b qui le sauvegarde. On peut typer ce processus dans un environnement $\Gamma = a : s_a, b : s_b, c : s_c$ avec $s_a = \forall \rho.\texttt{kell}(a)_{\rho \to \emptyset}$ et $s_b = s_c = \forall \rho \rho' \delta.\langle \texttt{kell}(\delta)_{\rho \to \rho'} \rangle_{\emptyset}$. Le type s_a est polymorphe est ne restreint pas les les kells pouvant être contenus dans le kell a. De la même façon, les canaux b et c peuvent transmettre n'importe quel nom de kell, comme les y autorisent les types polymorphe s_b et s_c.

Création dynamique de kell Considérons le processus $P = a\langle x,y,z\rangle \triangleright x[y \mid z]$. Soient les environnements Γ et Γ' définis par :

$$\Gamma = a : \forall\delta\rho\rho'.\langle \mathtt{kell}(\delta)_{\rho,\rho'\to\emptyset},\rho,\rho'\rangle_{\delta,\rho,\rho'}$$

$$\Gamma' = \Gamma,x : \mathtt{kell}(\delta)_{\rho,\rho'\to\emptyset},y : \rho,z : \rho'$$

$$\dfrac{\dfrac{\dfrac{(x,\mathtt{kell}(\delta)_{\rho,\rho'\to\emptyset}) \in \Gamma'}{\Gamma' \vdash x : \mathtt{kell}(\delta)_{\rho,\rho'\to\emptyset}}\ \text{T.Id} \qquad \dfrac{\Gamma' \vdash y : \rho \qquad \Gamma' \vdash z : \rho'}{\Gamma' \vdash y \mid z : \rho,\rho'}\ \text{T.Par}}{\Gamma' \vdash x[y \mid z] : \delta,\rho,\rho'}\ \text{T.Kell}}{\Gamma \vdash P : \emptyset}\ \text{T.Trig}$$

L'utilisation de variables de type permet d'une part de transmettre sur a des processus y,z et un nom de kell x arbitraires, et d'autre part d'exprimer le multi-ensemble δ,ρ',ρ'' correspondant au processus $x[y \mid z]$. Soit $\Gamma' = \Gamma,b : \forall\rho.\mathtt{kell}(b)_{\rho\to\emptyset}$. On a $\Gamma \vdash P \mid a\langle a,\mathbf{0},a[\mathbf{0}]\rangle : a,a$.

Contrôle Le système de type permet une forme limitée de contrôle. Considérons l'environnement :

$$\Gamma = b : \forall\rho.\mathtt{kell}(b)_{\rho\to\emptyset},c : \forall\rho.\mathtt{kell}(c)_{\rho\to\emptyset},a : \langle b\rangle_\emptyset$$

Dans cet environnement, le canal a est contraint à ne transporter que des processus contenant (au plus) le nom de kell b. Par exemple, le processus $a\langle b[] \mid c[]\rangle$ n'est pas typable dans cet environnement. De la même façon, dans l'environnement $\Gamma' = \Gamma,d = \forall\rho\rho'.\langle c\rangle^{\rho}_{\rho'}$, le processus $d\langle b\rangle$ n'est pas typable.

Cette notion de contrôle est ici plutôt anecdotique. Elle n'est pas utile pour vérifier la contrainte d'unicité des noms de kells.

3.3.5 Limitations

Ce système de type présente malheureusement des limitations importantes.

Considérons le processus $P = a\langle x\rangle \triangleright b\langle y\rangle \triangleright x \mid y$ et l'environnement $\Gamma = a : s_a,b : s_b$. On aimerait définir s_a et s_b de telle sorte que les canaux a et b puissent accepter n'importe quel processus. On aura alors $s_a = \forall\rho.\langle\rho\rangle_{\Delta_a}$ et $s_b = \forall\rho.\langle\rho\rangle_{\Delta_b}$. Pour que P soit typable dans Γ, on doit avoir $\Gamma,x : \rho \vdash b\langle y\rangle \triangleright x \mid y : \Delta_a$. On peut choisir par exemple $\Delta_a = \emptyset$. On doit alors avoir $\Gamma,x : \rho,y : \rho' \vdash x \mid y : \Delta_b$. On devrait alors avoir $\rho,\rho' \leq \Delta_b$, ce qui est impossible puisque ρ ne peut pas appartenir à Δ_b. Intuitivement, le type de b ne peut pas tenir compte des noms contenus dans x.

On peut néanmoins contourner partiellement le problème dans le jK-calcul. Le processus $P = a\langle x\rangle \mid b\langle y\rangle \triangleright x \mid y$ est typable dans l'environnement $\Gamma = a : s_a,b : s_b$ avec $s_a = s_b = \forall\rho.\langle\rho\rangle_\rho$.

Une deuxième limitation concerne le typage des motifs contenant des variables libres (règles T.Trig et T.Pass). La règle T.Pass interdit simplement les récepteurs de la forme $x\langle\widetilde{y}\rangle \triangleright P$. La règle T.Trig $x\langle\widetilde{y}\rangle \triangleright P$ contraint x a avoir un type de canal redéfinissable.

3.4 Perspectives

Le système de type de la section 3.2 peut etre amélioré de diverses manières. La première consiste à enrichir les types de processus pour refléter plus de nuance sur le contrôle que l'on peut exercer sur un processus, ou encore la façon dont ils peuvent etre composés. Par exemple, on peut vouloir exprimer le fait que bien que l'on ne sache pas manipuler l'état d'un processus, on soit capable de l'interrompre ou de le tuer. Par ailleurs, nous avons mentionné le fait que l'on pouvait utiliser un kell comme une structure d'accueil pour des programmes écrits dans des langages différents. Dans cette optique, on peut vouloir limiter certaines compositions parallèles. Par exemple, on ne sait pas composer un programme écrit en Java avec un programme écrit directement en kell calcul. Mais on peut composer le programme Java encapsulé dans un kell. A la manière de [39] ou [70], on pourrait considérér un système de type générique, plus seulement par rapport au langage de motifs, mais également par rapport à un langage de type de processus et une relation de sous-typage sur les processus.

Afin de typer encore plus finement les processus, l'étape suivante est d'autoriser des noms dans les types de processus. On peut citer trois applications intéressantes :

- Permettre de limiter l'accés à un programme aux seules ressources apparaissant dans son type, à la manière de [69], [70] et [33].
- De manière similaire au système de type de la section 3.3, on pourrait imposer une unicité locale (et plus globale) des noms de kells actifs. Cette contrainte est pertinente dans l'implantation présentée au chapitre 4.
- Si l'on considère des kells comme des composants hiérarchiques dont la configuration évolue dynamiquement, il est souhaitable d'assurer certains invariants, tels que la présence de sous-kells exhibant certaines interfaces.

Dans ces trois cas de figure, l'utilisation de noms dans les types est nécessaire. Une difficulté est liée à la présence de localités et à l'extrusion des noms. Par ailleurs, un traitement général de ces problèmes semble à la fois possible et nécessaire.

Chapitre 4

Implantation

4.1 Introduction

Dans ce chapitre, nous nous intéressons au Kell calcul comme langage de programmation réparti, et discutons son implantation. L'intérêt de ce travail est multiple. Tout d'abord, nous voulons évaluer les primitives du Kell calcul dans le cadre d'un langage de programmation. Pour cela, il est intéressant de disposer d'une implantation du langage. Ensuite, on veut déterminer si l'implantation répartie du langage peut se faire de manière efficace, et s'il est possible de prouver sa conformité avec la sémantique du calcul.

Nous présentons dans ce chapitre deux machines abstraites réparties ainsi qu'une implantation, pour le πK-calcul présenté au chapitre 2. La première version est assez proche du calcul et bénéficie d'une propriété de correction forte. La deuxième version consiste en un raffinement de la première, et correspond de manière directe à l'implantation. Une caractéristique originale de ces deux versions de machine abstraite est que, en comparaison avec d'autres machines abstraites pour des calculs de processus, elles ne dépendent pas d'un réseau sous-jacent particulier, mais peuvent être utilisées pour implanter le calcul dans des configurations physiques différentes.

Nous avons vu au chapitre 2 qu'un des principes de conception du Kell calcul consistait à garder toutes les actions locales afin d'en faciliter l'implantation répartie, et de permettre à différentes formes de localités de coexister. Une conséquence de ce principe est que le calcul permet de modéliser (par des processus) différents types de réseau. Ainsi, d'un coté une implantation du calcul ne devrait pas avoir besoin de considérer des actions atomiques ayant lieu dans des systèmes réparties à grande échelle. D'un autre coté, une implantation du calcul ne devrait pas forcément se limiter à utiliser des communications purement asynchrones entre les localités lorsque d'autres services sont offerts par le réseau sous-jacent : ainsi des implantations légitimes du calcul peuvent être basées sur des propriétés synchrones ou quasi-synchrones d'environnement spécifiques. Par exemple, une machine locale avec différents processeurs, ou un réseau local à grande performance et faible latence pour une grappe de PC homogènes.

Cet objectif est atteint de la manière suivante. Une implantation de notre machine abstraite consiste en deux parties distinctes :

– Une implantation de la spécification de machine abstraite à proprement parler, qui est conforme aux règles décrites en section 4.2 ou 4.3 .

– Des librairies, dans le langage d'implantation choisi, qui fournissent des accès aux services de réseaux et qui sont conformes à un modèle de ces services. Ce modèle est décrit par un processus du Kell calcul.

Supposons par exemple que l'on veuille réaliser une configuration physique comprenant un réseau N, qui interconnecte deux machines m_1 et m_2, et que chacune d'elles exécute une implantation d'une machine abstraite pour le Kell calcul, ainsi qu'un programme du Kell calcul (respectivement P_1 et P_2). Cette configuration serait modélisée dans le Kell calcul par un terme :

$$C \triangleq N[\texttt{Net} \mid m_1[\texttt{NetLib} \mid P_1] \mid m_2[\texttt{NetLib} \mid P_2]]$$

où le processus `Net` modélise le comportement du réseau N, et les processus `NetLib` modélisent la présence, sur chaque site, d'une librairie donnant accès aux services du réseau modélisé par `Net`. Du point de vue de la machine abstraite du Kell calcul, la librairie `NetLib` est simplement un terme standard du kell calcul, mais dont les communications ont des effets de bord (par exemple, accéder aux services concrets offerts par le réseau modélisé par `Net`) à l'extérieur de l'implantation de la machine abstraite.

L'aspect intéressant de notre approche est que nous pouvons ainsi fournir des implantations pour différents environnements qui reposent sur la même description de machine abstraite et sur la même implantation. Considérons par exemple la configuration physique consistant en un réseau N, qui interconnecte deux ordinateurs m_1 et m_2, et qui exécutent chacun deux processus séparés, p_i^1 and p_i^2 (i = 1,2). Chaque processus p_i^j exécute une implantation de la machine abstraite, avec un programme Q_i^j. Cette configuration peut être modélisée par

$$C' \triangleq N[\texttt{Net} \mid M_1 \mid M_2]$$
$$M_1 \triangleq m_1[\texttt{NetOS} \mid \texttt{Ipc} \mid p_1^1[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_1^1] \mid p_1^2[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_1^2]]$$
$$M_2 \triangleq m_2[\texttt{NetOS} \mid \texttt{Ipc} \mid p_2^1[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_2^1] \mid p_2^2[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_2^2]]$$

où le processus `NetOS` modélise la présence, sur chaque site m_i, de certains moyens (par exemple une librairie d'un système d'exploitation) d'accéder aux services de réseau modélisés par `Net`, où le processus `Ipc` modélise la présence, sur chaque site, d'une librairie de communication locale (par exemple, une librairie de communications inter-processus fournie par le système d'exploitation local) et où les processus `NetLib` and `IpcLib` modélisent la présence, pour chaque processus p_i^j, d'interfaces pour accéder aux différents services de communications fournis respectivement, par la combinaison de `Net` et `NetOS`, et par `Ipc`. À nouveau, `NetLib` et `IpcLib` se présentent tous deux comme des processus du Kell calcul du point de vue de la machine abstraite (c'est à dire qu'ils communiquent avec les autres processus par échange de message et qu'ils peuvent être passivés avec leur localité englobante). Toutefois, les services de communications auxquels ils donnent accès peuvent avoir des sémantiques très différentes, ne serait-ce qu'en terme de fiabilité, de latence ou de sécurité. Le point important à noter est que différents services de communication peuvent coexister dans la même implantation, et peuvent être utilisés sélectivement par les applications.

Le caractère local des primitives du Kell calcul permet de rendre la spécification de machine abstraite indépendante des services offerts par les réseaux sous-jacents. Une conséquence est la simplification de la preuve de correction de la machine abstraite. En effet, la

preuve de correction ne nécessite pas la preuve de protocoles de migration répartis non-triviaux, comme c'est le cas pour l'implantation JoCaml du Join calcul réparti [24], ou de machines abstraites variées pour des calculs d'ambiant [28, 36, 24, 56] [1]. De plus, la correction de la machine est assurée, indépendamment des services de réseau utilisés pour l'implantation effective.

Le chapitre est organisé de la manière suivante. Dans la section 4.2, nous spécifions une machine abstraite de haut-niveau, proche du calcul, pour laquelle nous énonçons une propriété forte de correction. Nous présentons ensuite dans la section 4.3 une version raffinée corresondant de manière directe à l'implantation. Dans la section 4.4, nous décrivons un prototype écrit en OCaml de notre machine abstraite. Dans la section 4.5, nous comparons notre approche à des travaux similaires. Finalement, nous concluons en 4.6 sur les futures travaux possibles. Les preuves des propriétés de correction sont données dans l'annexe B. Une introduction au langage `CHALK` et son implantation est donnée dans l'annexe C.

4.2 Un calcul de machines abstraites

Nous présentons une machine abstraite non-déterminisme et proche du calcul. Comparée au calcul, elle réalise trois fonctions importantes : (1) elle implante concrètement l'opération de création de nom (2) elle introduit une structure de *localité* qui matérialise un Kell comme structure d'exécution. La hiérarchie entre localités est matérialisée à l'aide de pointeurs. (3) elle introduit une nouvelle classe de valeurs correspondant à des processus passivés, et distingue ainsi les valeurs correspondant à des programmes (au sens de code passif), des valeurs correspondant à des états d'exécution d'un programme (cf. discussion 2.2).

La correction de la machine est énoncée, de manière similaire à [56], sous la forme d'une bisimilarité barbée entre un processus du Kell calcul et son interprétation sous forme de machine abstraite. Toutefois, les résultats obtenus sont en fait plus forts que la pure bisimilarité barbée parce qu'ils utilisent une forme d'équivalence contextuelle. Les résultats sont énoncés à l'aide d'une forme forte de bisimilarité, car nous utilisons une seconde relation de réduction pour nous abstraite de réductions purement administratives.

4.2.1 Syntaxe

Nous suivons l'approche de [56] et spécifions notre machine abstraite sous la forme d'un calcul de processus dont les termes correspondent à des états de la machine abstraite. Intuitivement, un terme de ce calcul consiste en un ensemble de localités, chacune d'entre elles exécutant un programme différent, organisées en un arbre par le biais de pointeurs entre les localités. La syntaxe de la machine abstraite est donnée figure 4.1. Les termes générés par les productions M dans la grammaire de machine abstraite sont appelés des *termes de machines* (ou simplement *machines* lorsqu'il n'y a pas d'ambiguïté). Nous utilisons les variables M, N et leurs variantes décorées pour désigner les termes de machines. Nous notons M leur ensemble. Les termes de machine utilisent deux nouveaux ensembles de noms. Un ensemble de *noms de localité*, Locs dont les éléments sont décrits par les

1. On peut noter que la machine abstraite "Channel Ambient" présentée dans [52] suppose que des ambiants peuvent se synchroniser, en utilisant par exemple une primitive `in`. Cette hypothèse peut rendre difficile l'implantation dans un contexte réparti asynchrone.

variables l,m,n,..., et un ensemble de *noms résolus*, ResNames dont les éléments sont décrits par les variables i,j,k,.... On note p,q,r,... les éléments de ResNames $\cup$ Names. On appelle *localité* un terme de machine de la forme $l : p[P]_{m,S}$. Dans une localité $l : p[P]_{m,S}$, p est le nom du kell que la localité représente, l est le nom de la localité, m est le nom de la localité parente, S est l'ensemble des noms de ses sous-localités, et P est le *processus de machine* exécuté par la localité h. Nous utilisons deux noms de localités particulier : **r** et **rp**, qui dénotent respectivement, le nom de la localité de plus haut-niveau et le nom de la localité (virtuelle) parente de de **r**. On utilise également le nom particulier **r** pour désigner le kell de plus haut niveau.

Nous appelons MK l'ensemble des processus de machines, et nous notons P,Q,... ces processus. Leur syntaxe est donnée figure 4.1. Les processus de machine sont similaires à ceux du πK-calcul. Deux différences sont à noter. Premièrement, un nouveau terme **reify**(l,M) est introduit pour représenter une machine passivée. Le terme M est un arbre de machine encodé comme une composition parallèle de localités et l est le nom de la localité racine de cette arbre. Deuxièmement, les identifiants qui peuvent être utilisés par un processus de machine peuvent également être des noms résolus. Ce point sera détaillé dans la sous-section 4.2.2. Notons finalement que l'on a $\mathsf{K} \subseteq \mathsf{MK}$. Pour finir, remarquons que nous utilisons les même méta-variables pour dénoter des processus et des processus de machine. Lorsque ce n'est pas clair à partir du contexte, nous préciserons si une variable dénote un processus ou un processus de machine. Dans la suite, on supposera que tous les processus considérés vérifient la condition $\mathtt{fv}(P) = \emptyset$.

Machines et localités

$M ::= \mathbf{0} \mid L \mid M \mid M$ machine

$L ::= l : p[P]_{m,S}$ localité

$S ::= \emptyset \mid l \mid S, S$ ensemble des sous-localités

Processus de machine

$P ::= \mathbf{0} \mid x \mid \xi \triangleright P \mid \nu a.P \mid P \mid P \mid u[P] \mid u\langle \widetilde{V} \rangle \mid \mathbf{reify}(l,M)$

Valeur de machine

$V ::= u,v,w,\ldots$ identifiant

$\mid P$ processus

Identifiant

$u ::= p,q,r\ldots$ Canal ou nom résolu

$\mid x,y,z,\ldots$ variable

Noms résolus

$i,j,k,\ldots ::=$ nom résolu

Noms de localités

$l,m,n ::= \ldots$ nom de localité

FIG. 4.1 – *Syntaxe du calcul de machine abstraite*

On définit en 4.2 un sous-ensemble des machines et des processus de machine. On dit qu'un processus de machine est sous forme normale s'il ne contient pas d'opérateur de restriction, ni de kell en contexte d'évaluation. On note P_*,Q_*,... de tels processus. Une machine est sous forme normale lorsque tous les processus exécutés par ses localités sont

sous forme normale. On note ces termes M_*,N_*, ...

Terme de machines et localités

$M_* ::= \mathbf{0} \mid L_* \mid M_* \mid M_*$ terme de machine

$L_* ::= l : p[P_*]_{m,S}$ localité

Processus de machine

$P_* ::= \mathbf{0} \mid x \mid \xi \triangleright P \mid P_* \mid P_* \mid u\langle \widetilde{V} \rangle \mid \mathbf{reify}(l,M)$

FIG. 4.2 – *Processus et machine sous forme normale*

On définit la fonction **locnames**, qui renvoie les noms de localité d'une machine.

$$\mathbf{locnames}(\mathbf{0}) = \emptyset$$
$$\mathbf{locnames}(M \mid N) = \mathbf{locnames}(M) \cup \mathbf{locnames}(N)$$
$$\mathbf{locnames}(l : p[P]_{m,S}) = \{l,m\} \cup S$$

Les définitions et conventions données en section 2.4 s'étendent aux processus de machine.

Nous avons défini les termes de machine pour matérialiser par des pointeurs (*i.e.* les noms de localités) la structure d'arbre selon laquelle les localités s'organisent. Toutefois, la définition syntaxique des machines n'impose aucune contrainte sur les noms de localités. Pour y remédier, nous définissons le prédicat **tree** qui permet de restreindre l'ensemble des machines à celles présentant une structure d'arbre. Le prédicat $\mathbf{tree}(M,l,p,m)$ est défini comme suit :

$$\mathbf{tree}(M,l,p,m) = (M \equiv l : p[P]_{m,S} \mid \prod_{\alpha \in S} M_\alpha) \wedge_{\alpha \in S} \mathbf{tree}(M_\alpha,l_\alpha,p_\alpha,m))$$

avec la condition supplémentaire que les noms l,m,l_α sont tous distincts. Nous appelons *arbre* une machine M vérifiant le prédicat $\mathbf{tree}M,l,p,m$ pour des noms l,p,m quelconques. Nous disons de plus qu'une machine M est *bien-formée* si l'on a $\mathbf{tree}(M,\mathbf{r},\mathbf{rn},\mathbf{rp})$. L'ensemble des machines bien formées est noté WFM.

Finalement, nous aurons besoin de la relation $\cong$ sur les machines bien formées définie comme suit : $M \cong N$ si et seulement si $M\sigma \equiv N\sigma'$ où σ et σ' sont des renommages injectifs des noms de localités et des noms résolus.

4.2.2 Relation de réduction

Une étape d'exécution d'une machine abstraite peut-être de deux types, et est modélisée par deux relation de réduction différentes. Une *relation de réduction administrative*, définie comme la plus petite relation qui satisfait les règles de réduction de la figure 4.4 et une *relation de réduction simple*, ou simplement relation de réduction, définie comme la plus petite relation qui satisfait les règles de la figure 4.5.

Ces deux relations utilisent une relation d'équivalence structurelle, notée $\equiv$, et définie comme la plus petite relation d'équivalence qui vérifie les règles de la figure 4.3 et telle que l'opérateur de composition parallèle soit associatif, commutatif et admette $\mathbf{0}$ comme élément neutre. Cette définition utilise deux autres relations d'équivalence, notées également $\equiv$, sur les processus de machines et sur les ensembles de localités. Elles sont définies

$$\frac{M =_\alpha N}{M \equiv N}\ \text{M.SE.}\alpha \qquad \frac{P \equiv P' \qquad S \equiv S'}{l : p[P]_{m,S} \equiv l : p[P']_{m,S'}}\ \text{M.SE.Ctx}$$

FIG. 4.3 – *Equivalence structurelle sur les termes de machines*

comme les plus petites relations telles que l'opérateur | (respectivement, l'opérateur ,) soit associatif, commutatif et admette **0** (respectivement $\emptyset$) comme élément neutre.

L'équivalence structurelle permet de considérer les machines M comme des multi-ensembles de localités, et les termes S comme des multi-ensembles de noms de localités. Par ailleurs, remarquons que la relation d'équivalence structurelle entre processus de machine ne contient pas de règles liées à l'opérateur de création de noms. La création de nom est vue maintenant comme une étape d'exécution.

Les étapes d'exécution d'une machine sont représentées par deux relations de réduction différentes. D'un point de vue purement opérationnel, il n'y a pas de raison de distinguer ces relations. Par contre, ce choix permet de prouver la correction de la machine abstraite vis a vis du Kell calcul. Nous y reviendrons en section 4.2.3.

Intuitivement, les réductions administratives correspondent à des étapes d'exécution de la machine qui ne correspondent pas directement à des réductions du calcul. Nous détaillons les principales règles.

La règle M.S.New, soit

$$\frac{i\ \mathbf{fresh}}{l : p[(\nu a.P) \mid Q]_{m,S} \stackrel{\equiv}{\to} l : p[P\{i/a\} \mid Q]_{m,S}}\ \text{M.S.New}$$

implante l'opérateur de restriction comme une création de nom. On note i **fresh** et l'on dit que le nom i est *frais* lorsque ce nom est unique à l'échelle du système.

La règle M.S.Cell crée une nouvelle localité et met à jour les pointeurs régissant la structure d'arbre de la machine.

$$\frac{n\ \mathbf{fresh}}{l : p[q[P] \mid Q]_{m,S} \stackrel{\equiv}{\to} l : p[Q]_{m,(S,n)} \mid n : q[P]_{l,\emptyset}}\ \text{M.S.Cell}$$

La règle M.S.Act réactive une machine passivée. L'activation consiste d'une part à replacer le processus contenu dans la localité de plus haut niveau de la machine passivée dans la localité courante, d'autre part à définir les sous-localités de la machine passivée comme des sous-localités de la localité courante. On veut s'assurer de plus que les noms de localités de M_* sont renommés par des noms frais, afin d'éviter de les dupliquer en cas de plusieurs réactivations de M_*. On utilise pour ça la fonction $\mathbf{fresh}(l,n,M)$ définie comme suit. Si M est une machine telle que $\mathbf{tree}(M,l,p,m)$. La fonction $\mathbf{fresh}(l,n,M)$ renvoie M dans laquelle tous les noms de localités ont été remplacés par des noms frais, et n a été remplacé par l. Formellement $\mathbf{fresh}(l,n,M) = M\{l/n\}\sigma$ ou σ est un renommage injectif de $\mathbf{locnames}(M)$ vers des noms frais.

$$\frac{\mathbf{fresh}(l,n,M_*) = l : q[R_*]_{l',S'} \mid M'_*}{l : p[\mathbf{reify}(n,M_*) \mid P]_{m,S} \stackrel{\equiv}{\to} l : p[R_* \mid P]_{m,(S,S')} \mid M'_*}\ \text{M.S.Act}$$

$$\frac{i \textbf{ fresh}}{l : p[(\nu a.P) \mid Q]_{m,S} \xrightarrow{\equiv} l : p[P\{i/a\} \mid Q]_{m,S}} \text{ M.S.New}$$

$$\frac{n \textbf{ fresh}}{l : p[q[P] \mid Q]_{m,S} \xrightarrow{\equiv} l : p[Q]_{m,(S,n)} \mid n : q[P]_{l,\emptyset}} \text{ M.S.Cell}$$

$$\frac{\textbf{fresh}(l,n,M_*) = l : q[R_*]_{l',S'} \mid M'_*}{l : p[\textbf{reify}(n,M_*) \mid P]_{m,S} \xrightarrow{\equiv} l : p[R_* \mid P]_{m,(S,S')} \mid M'_*} \text{ M.S.Act}$$

$$\frac{M \xrightarrow{\equiv} M'}{M \mid N \xrightarrow{\equiv} M' \mid N} \text{ M.S.Ctx} \qquad \frac{M \equiv M' \qquad M' \xrightarrow{\equiv} M'' \qquad M'' \equiv M'''}{M \xrightarrow{\equiv} M'''} \text{ M.S.Str}$$

Fig. 4.4 – *Relation de réduction administrative sur les machines*

Nous détaillons maintenant la relation de réduction simple. La règle de réduction M.Out,

$$\frac{\xi = r\langle x \rangle^{\uparrow}}{l : p[\xi\varphi \mid P]_{m,S} \mid l' : q[(\xi \triangleright Q) \mid R]_{l,S'} \mapsto l : p[P]_{m,S} \mid l' : q[Q\varphi \mid R]_{l,S'}} \text{ M.Out}$$

modélise une communication impliquant un kell de nom l et un de ses sous-kells de nom l'. Un message contenu dans l réagit avec un récepteur de l'. Elle correspond précisément à la règle R.Out du πK-calcul. De la même façon, les règles M.Out et M.Local correspondent aux règles R.Out et R.Local.

La règle M.Pass correspond à la règle R.Pass du jK-calcul.

$$\frac{M_* = l_r : p[R_*]_{l,S'} \mid M'_* \qquad \textbf{tree}(M_*,l_r,p,l)}{l : q[(p[x] \triangleright P) \mid Q]_{m,S} \mid M_* \mapsto l : q[P\{\textbf{reify}(l_r,M_*)/x\} \mid Q]_{m,S\setminus\{l_r\}}} \text{ M.Pass}$$

La localité de nom l contient un récepteur de passivation $p[x] \triangleright P$. La racine du sous-arbre M_* a pour pour parent la localité l et correspond à un kell de nom p. Le sous-arbre M_* est transformé en une valeur $\textbf{reify}(l_r,M_*)$ que l'on substitue à x dans P. On impose au sous-arbre M_* d'être sous-forme normale afin de s'assurer que toutes les créations de noms ont été effectuées avant une passivation, conformément à la règle R.Pass du calcul.

4.2.3 Correction

Nous établissons la correction de la machine abstraite à l'aide de deux résultats. Le premier est élémentaire : nous montrons que la structure d'arbre de la machine abstraite est bien préservée lors de la réduction. Le deuxième constitue notre résultat principal et dit essentiellement qu'un processus du Kell calcul et *équivalent* à son interprétation par la machine abstraite, pour une relation d'équivalence $\sim$, définie plus bas. Ces propriétés correspondent aux deux théorèmes suivants.

Théorème 4 (Préservation de la bonne formation) *Si M est bien formée et $M \mapsto M'$ (resp. $M \xrightarrow{\equiv} M'$), alors M' est bien formée.*

$$\frac{\xi = r\langle x\rangle}{l : p[\xi\varphi \mid (\xi \triangleright Q) \mid P]_{m,S} \mapsto l : p[Q\phi \mid P]_{m,S}}\ \text{M.Local}$$

$$\frac{\xi = r\langle x\rangle^{\uparrow}}{l : p[\xi\varphi \mid P]_{m,S} \mid l' : q[(\xi \triangleright Q) \mid R]_{l,S'} \mapsto l : p[P]_{m,S} \mid l' : q[Q\varphi \mid R]_{l,S'}}\ \text{M.Out}$$

$$\frac{\xi = r\langle x\rangle^{\downarrow}}{l : p[(\xi \triangleright Q) \mid P]_{m,S} \mid l' : q[\xi\varphi \mid R]_{l,S'} \mapsto l : p[Q\varphi \mid P]_{m,S} \mid l' : q[R]_{l,S'}}\ \text{M.In}$$

$$\frac{M_* = l_r : p[R_*]_{l,S'} \mid M'_* \qquad \mathbf{tree}(M_*,l_r,p,l)}{l : q[(p[x] \triangleright P) \mid Q]_{m,S} \mid M_* \mapsto l : q[P\{\mathbf{reify}(l_r,M_*)/x\} \mid Q]_{m,S\setminus\{l_r\}}}\ \text{M.Pass}$$

$$\frac{M \mapsto M'}{M \mid N \mapsto M' \mid N}\ \text{M.Ctx} \qquad \frac{M \equiv M' \qquad M' \mapsto M'' \qquad M'' \equiv M'''}{M \mapsto M'''}\ \text{M.Str}$$

FIG. 4.5 – *Relation de réduction simple sur les machines*

Théorème 5 (Correction) *Pour tout processus du Kell calcul P nous avons $[\![P]\!] \sim P$.*

Dans le reste de cette sous-section nous donnons les définitions principales ainsi que les résultats intermédiaires qui interviennent dans la preuve du théorème 5. Nous commençons par définir la traduction d'un terme du Kell calcul dans le calcul de machines abstraites.

Définition 4.2.1 $[\![P]\!] = \mathbf{r} : \mathbf{rn}[P]_{\mathbf{rp},\emptyset}$

On voit aisément que $[\![P]\!]$ est bien formé pour tout processus P. A partir de maintenant et en vertu du théorème 4 nous ne considérons que des termes de machine M bien formés.

Nous utilisons la définition 2.4.3 pour définir la relation de bisimilarité forte $\sim$ sur les termes de machine bien formés et sur les processus du Kell calcul. Cette équivalence nous permet de comparer des systèmes de transitions différents, dès lors que l'on sait les munir d'un même prédicat d'observation. Le prédicat suivant correspond au prédicat sur le Kell calcul.

Définition 4.2.2 (Prédicat d'observation pour les machines) *Si M est une machine bien formée et $a \in$ NAMES, on a $M \downarrow_a$ si et seulement si l'un des cas suivants est vrai:*

1. $M \equiv\!\stackrel{\equiv}{\rightarrow}^{*} \mathbf{r} : \mathbf{rn}[a\langle\widetilde{P}\rangle \mid R]_{\mathbf{rp},S} \mid M'$
2. $M \equiv\!\stackrel{\equiv}{\rightarrow}^{*} l : p[a\langle\widetilde{P}\rangle \mid R]_{\mathbf{r},S} \mid M'$
3. $M \equiv\!\stackrel{\equiv}{\rightarrow}^{*} l : a[P]_{\mathbf{r},S} \mid M'$

Intuitivement, une barbe sur a signifie qu'après un nombre arbitraire de réductions administratives, une machine M peut exhiber un message local (clause 1), un message vers le haut (clause 2), ou un message de type kell (clause 3). Ces observations correspondent directement à celles définies sur le calcul en 2.4.4.

La relation de réduction utilisée sur le calcul de machine abstraite est $\rightarrow = \overset{\equiv}{\rightarrow}^{*} \mapsto$. La distinction entre réduction administrative et réduction simple nous permet d'utiliser une forme forte de bisimilarité. Une réduction du calcul va correspondre à éventuellement plusieurs réductions administratives suivies d'exactement une réduction simple de machine.

Nous définissons maintenant deux relations d'équivalences sur les machines qui nous serviront pour énoncer des propriétés de correction. La première identifie deux machines qui ont la même forme normale. La seconde correspond à une forme de congruence barbée forte.

Lemme 4.2.3 *$M \overset{\equiv}{\not\rightarrow}$ si et seulement si, il existe N_* tel que $M = N_*$.*

Lemme 4.2.4 (Forme normale) *Si M est un terme de machine, alors il existe M'_* tel que $M \overset{\equiv}{\rightarrow}^{*} M'_*$. De plus, si $M \equiv \overset{\equiv}{\rightarrow}^{*} M''_*$ alors $M'_* \cong M''_*$.*

Les opérations de création de noms frais utilisées dans les règles définissant la relation de réduction administrative sont non-déterministes. La relation $\cong$ permet d'identifier des machines qui ont des noms de localité et des noms résolus différents.

Définition 4.2.5 (Equivalence) *On dit que deux machines M et N sont* équivalentes*, et l'on note $M \doteq N$, si elles ont la même normale forme (modulo $\cong$).*

A partir de maintenant, lorsqu'il n'y a pas d'ambiguïté, nous utilisons la notation M_* pour désigner soit une méta-variable représentant une machine sous forme normale, soit une forme normale de M (*i.e.* un terme M_* tel que $M \overset{\equiv}{\rightarrow}^{*} M_* \overset{\equiv}{\not\rightarrow}$).

Définition 4.2.6 *Soit $M = l : p[P]_{m,S} \mid M'$ une machine telle que* **tree**(M,l,p,m)*. On définit :*

$$\begin{aligned} M \mid Q &= l : p[P \mid Q]_{m,S} \mid M' \\ q[M] &= l : p[\mathbf{0}]_{m,h} \mid h : q[P]_{l,S} \mid M'\{h/l\} \\ \nu a.M &= M\{i/a\} \end{aligned}$$

On étend ces définitions à tous les contextes ayant la forme suivante:

$$\mathbf{E} ::= . \quad | \quad (R \mid \mathbf{E}) \quad | \quad q[\mathbf{E}] \quad | \quad \nu a.\mathbf{E}$$

Définition 4.2.7 (Equivalence contextuelle pour les machines) *Deux machines bien formées M et N sont contextuellement équivalentes ($M \sim_c N$) si et seulement si $\forall \mathbf{E}, \mathbf{E}[M] \sim \mathbf{E}[N]$.*

On vérifie aisément que $\sim_c$ est la plus grande relation sur les machines incluse dans la bisimilarité barbée forte, préservée par $p[.]$, $\nu a..$ et $. \mid R$.

Lemme 4.2.8 *$\sim_c$, $\doteq$, $\cong$ et $\equiv$ sont des relations d'équivalence.*

Lemme 4.2.9 *Si l'on considère la restriction de $\equiv$ aux machines bien formées, on a $\equiv \subseteq \cong \subseteq \doteq \subseteq \sim_c$.*

Nous énonçons maintenant deux propriétés qui mettent en relation les réductions des processus du Kell calcul avec celles des termes du calcul de machine (correction), et les réductions des processus du Kell calcul avec les réductions des machines (complétude).

Proposition 4.2.10 (Correction) $[\![P]\!] \to M \implies P \to P'$ *avec* $[\![P']\!] \doteq M$.

Preuve 1 *Nous donnons ici un résumé de la preuve, dont la version détaillée est donnée en annexe . Nous définissons tout d'abord par induction une fonction de traduction inverse* $[\![.]\!]^{mac}$ *des machines vers les processus. Cette fonction a trois rôles. Elle "déplie" les processus réifiés, reconstruit la structure syntaxique d'arbre du terme et recrée les noms restreints à partir des noms résolus.*

La proposition de correction résulte des lemmes suivants :

Lemme 4.2.11 *Si* M *est bien formé et* $M \stackrel{\equiv}{\to} N$ *alors* $[\![M]\!]^{mac} \equiv \stackrel{\equiv}{\to}^{*} [\![N]\!]^{mac}$.

Lemme 4.2.12 *Si* M *est bien formé et* $M \mapsto N$ *alors* $[\![M]\!]^{mac} \mapsto [\![N]\!]^{mac}$.

Lemme 4.2.13 *Si* M *est une machine bien formée, alors* $[\![[\![M]\!]^{mac}]\!] \sim_c M$. *Si* P *est un processus, alors* $[\![[\![P]\!]]\!]^{mac} \equiv P$.

Proposition 4.2.14 (Complétude) $P \to P' \implies [\![P]\!] \to \sim_c [\![P']\!]$

Preuve 2 (Plan) *La preuve de cette proposition se fait par induction sur la dérivation de* $P \to P'$ *et nécessite les deux lemmes suivants :*

Lemme 4.2.15 *Si* $P \equiv P'$ *alors* $[\![P]\!] \doteq [\![P']\!]$. *Si* $P \stackrel{\equiv}{\to} P'$ *alors* $[\![P]\!] \doteq [\![P']\!]$.

Lemme 4.2.16 *Soit* P_* *un processus et* M_* *une machine telle que* $\mathbf{tree}(M_*,l,p,m)$. *Si nous avons* $l : p[P_*]_{m,\emptyset} \stackrel{\equiv}{\to}^{*} \cong M_*$ *alors pour toute machine* N *on a* $N\{\mathbf{reify}(l,M_*)/x\} \sim_c N\{P_*/x\}$.

Le preuve du théorème 5 découle immédiatement des propositions 4.2.10 et 4.2.14 en montrant que la relation $\{\langle [\![P]\!], P\rangle \mid P \in \mathsf{K}\}$ est une bisimulation barbée forte "up to" $\sim_c$.

4.3 Une machine raffinée

Nous décrivons maintenant une version raffinée de machine abstraite. Les différences principales par rapport à la section précédente sont d'une part l'ordonnancement des processus dans les localités, et d'autre part une gestion du routage des messages. La spécification résultante est maintenant quasi-déterministe (seul le parcours de l'arbre des localités n'est pas spécifié) et correspond précisément à l'implantation.

Ordonnancer les processus dans une localité consiste à donner une sémantique déterministe à l'opérateur de composition parallèle. Par exemple, le Kell calcul ou la machine de la section 4.2 identifie les processus $P \mid Q$ et $Q \mid P$. Dans cette version raffinée, nous les différencions et spécifions un ordre d'exécution.

Un problème plus complexe concerne le routage des messages. Considérons un message $p\langle\widetilde{V}\rangle$. Nous devons déterminer une manière efficace de trouver s'il existe un récepteur pouvant consommer ce message. Réciproquement, lorsqu'un récepteur $p\langle\widetilde{x}\rangle \triangleright P$ est évalué, nous devons déterminer si un message a déjà été émis sur le canal de nom p. Ce problème n'a pas été traité dans la section précédente. En effet, une seule règle de réaction atomique

(*e.g.* M.Local dans le cas local) décrivait la réaction. Une solution à ce problème a été donnée par Turner dans [65] dans le cadre du π-calcul. Essentiellement, il introduit une structure de *tas* global, où des noms sont associés à des *files de réaction*, qui peuvent contenir des messages ou des récepteurs. Lors de l'évaluation d'un message sur un certain nom, le premier élèment de la file correspondant à ce nom est examiné. S'il s'agit d'un message, le nouveau message est simplement inséré dans la file. S'il s'agit d'un récepteur, le message peut réagir et le récepteur est extrait de la file. L'évaluation d'un récepteur se fait de manière symétrique. Nous introduisons ici une structure similaire adapté à notre calcul, qui prend en compte les notions de localité, de messages orientés et de passivation.

4.3.1 Syntaxe

La syntaxe de la machine raffinée est donnée figure 4.6. La syntaxe des processus de machine est quasi-inchangée. La principale différence concerne l'utilisation de messages orientés. Nous notons M un message quelconque et T un récepteur. Dans la définition d'une location $\mathcal{L}$, une file d'exécution $\mathcal{R}$ remplace le processus de machine de la section précédente. Par ailleurs, une structure additionnelle $\mathcal{T}$, que nous appelons *réacteur* (figure 4.7) a été ajouté. La file d'exécution $\mathcal{R}$ consiste simplement en une liste de processus de machine. Un réacteur est un ensemble d'association entre des *clés* et des *files de réaction*. Une clé consiste en une direction et un nom. Une file de réaction est une liste de *réactants*. Un réactant peut-être soit un processus annoté par un nom de localité, soit un récepteur partiel (de la forme $\langle x\rangle \triangleright P$), soit un nom de localité.

File d'exécution

$\mathcal{R} ::= []$ liste vide
$\quad | \quad P :: \mathcal{R}$ ajout d'un élément en tête

Machines et localités

$\mathcal{M} ::= \mathbf{0} \quad | \quad \mathcal{L} \quad | \quad \mathcal{M} \mid \mathcal{M}$ machine
$\mathcal{L} ::= l : p[\mathcal{R}]_{m,S} : \mathcal{T}$ localité

Processus de machine

$P ::= M \quad | \quad T \quad | \quad \mathbf{0} \quad | \quad x \quad | \quad \nu a.P \quad | \quad P \mid P \quad | \quad u[P] \quad | \quad \mathbf{reify}(l,M)$ processus
$M ::= u\langle \widetilde{V}\rangle \quad | \quad u^{\downarrow^v}\langle \widetilde{V}\rangle \quad | \quad u^{\uparrow}\langle \widetilde{V}\rangle$ message
$T ::= u\langle \widetilde{x}\rangle \triangleright P \quad | \quad u[x] \triangleright P$ récepteur

Fig. 4.6 – *Syntaxe du calcul de machine raffinée*

Les prédicats **tree** et la relation $\cong$ se définissent sans changement sur les termes machines raffinées. On définit également le prédicat de bonne formation.

4.3.2 Réacteur

Un réacteur peut être vu comme un tas local, par opposition au tas global de Pict où il n'y a qu'une localité. Dans le cas de Pict et du π-calcul, on peut se contenter d'une file de réaction par nom. Dans notre cas, ce n'est pas possible pour deux raisons. Tout d'abord les messages et les récepteurs doivent exhiber des actions complémentaires. Par exemple,

$$
\begin{array}{rll}
C ::= & \langle l, \widetilde{V} \rangle & \text{message et localité d'origine} \\
| & \langle \widetilde{x} \rangle \triangleright P & \text{récepteur partiel} \\
| & \langle l \rangle & \text{localité} \\
\\
\mathcal{C} ::= & [] \mid C :: \mathcal{C} & \text{file de réactant} \\
\\
dir ::= & \triangle & \text{message venant d'en haut/ récepteur écoutant en haut} \\
| & \nabla & \text{message venant du bas/ récepteur écoutant en bas} \\
| & \leftrightarrow & \text{message ou récepteur local} \\
| & \times & \text{localité ou récepteur de passivation} \\
key ::= & p^{dir} & \text{nom } p \text{ et direction } dir \\
\\
\mathcal{T} ::= & \{k \to \mathcal{C}\}_{k \in key} & \text{réacteur}
\end{array}
$$

FIG. 4.7 – *Réacteur*

le terme $a[(b\langle\rangle \triangleright \mathbf{0}) \mid b^{\uparrow}\langle\rangle)$ ne peut pas se réduire. Ensuite, les récepteurs et les messages peuvent se situer dans des localités différentes et doivent être mis en correspondance.

Pour résoudre ces problèmes, en plus d'utiliser un tas par localité, nous utilisons des noms annotés (*clés*) pour désigner les files de réactions des réacteurs. Une clé est un nom annoté par une direction correspondant à quatre cas différents. Une clé de la forme $p^{\triangle}$ est associée à une file de message provenant de la localité parente, ou de récepteurs dans la localité courante (celle à laquelle le récepteur appartient) et écoutant vers le haut. Une clé de la forme $p^{\leftrightarrow}$ est associée à une file de messages ou récepteurs originaires de la localité courante. Une clé de la forme p^{∇} est associée à une file de messages venant d'une sous-localité ou de récepteurs de la localité courante et écoutant vers le bas. Finalement, une clé $p^{\times}$ est associée à une file de noms de localités et de récepteur de passivation. La figure 4.3.2 donne un exemple de réacteur, contenant deux messages originaires de la localité parente sur le canal a, deux récepteurs sur b écoutant localement et vers le haut. Et finalement, un sous-kell de nom l, et de nom de kell c.

$$
\begin{array}{ll}
\triangle & a \to \{\langle m, V_1 \rangle, \langle m, V_2 \rangle\}, b \to \{\langle x \rangle \triangleright \mathbf{0}\} \\
\leftrightarrow & b \to \{\langle y \rangle \triangleright y\} \\
\nabla & \\
\times & c \to \{\langle l \rangle\}
\end{array}
$$

FIG. 4.8 – *Exemple de réacteur*

On peut remarquer que les récepteurs restent toujours dans le réacteur de la localité qui les a définis. A l'inverse, les messages sont routés vers la localité que l'on peut déterminer à partir de leur direction (cf. la discussion en fin de section). Deuxièmement, pour éviter

les redondances avec les informations contenues dans la clé, on peut supprimer le nom du canal des messages et récepteurs contenus dans les files. Finalement, pour une raison expliquée plus loin, on conserve l'information de la location d'origine dans les messages. Les réactants sont donc soit des noms de localité de la forme $\langle l \rangle$, soit des récepteurs partiels $\langle \widetilde{x} \rangle \triangleright P$ soit des messages avec leur location d'origine $\langle l, \widetilde{V} \rangle$.

Insertion de réactants dans un réacteur Une opération de base de la machine abstraite consiste à insérer des réactants dans les files de réactions. Cette opération retourne des produits de réaction, et un nouveau réacteur. Par exemple, insérer le message $b\langle P \rangle$ dans le réacteur de la figure 4.3.2 retournera le produit de réaction P ainsi qu'un nouveau réacteur dans lequel la liste associée à $b^{\leftrightarrow}$ est vide. Lorsqu'une réaction implique une passivation, nous avons besoin d'informations supplémentaires. Supposons par exemple que l'on insère le récepteur $c[x] \triangleright x$ dans le réacteur : le nom l doit faire référence à une localité définie dans une machine $\mathcal{M}$. L'action de passivation transformera $\mathcal{M}$ en $\mathcal{M}'$.

Dans la suite, nous allons définir l'opération plus générale d'insérer le contenu d'un réacteur dans un autre. L'insertion d'un seul réactant est un cas particulier d'un réacteur ne contenant qu'une seule association d'une clé à une file ne contenant qu'un réactant. Cette opération prend la forme

$$\mathcal{M} : \mathcal{T}_1 \otimes \mathcal{T}_2 = \mathcal{M}', \mathcal{R}, \mathcal{T}_2'$$

et correspond à l'insertion du contenu du réacteur $\mathcal{T}_1$ dans $\mathcal{T}_2$, où les noms de localités sont définies dans un environnement $\mathcal{M}$. Elle renvoie un nouvel environnement $\mathcal{M}'$, un nouveau réacteur $\mathcal{T}'$ et une file d'exécution $\mathcal{R}$. L'opérateur $\otimes$ est défini à l'aide des fonctions `genreact` et `genrq` données en figure 4.9 et 4.10.

$$\mathcal{M} : \mathcal{T}_1 \otimes \mathcal{T}_2 = \texttt{genrq}(\mathcal{M}, \mathcal{T}_1, \mathcal{T}_2), \texttt{genreact}(\mathcal{T}_1, \mathcal{T}_2)$$

La fonction $\texttt{genreact}(\mathcal{T}_1, \_\mathcal{T}_2)$ retourne retourne le réacteur $\mathcal{T}_2$ privé des réactants qui ont réagi avec ceux de $\mathcal{T}_1$. La fonction $\texttt{genrq}(\mathcal{M}, \mathcal{T}_1, \mathcal{T}_2)$ calcule la file d'exécution et le nouvel environnement correspondant à $\mathcal{M}$ privé des localités ayant été passivées.

Les fonction `genreact` et `genrq` La fonction `genreact` est définie tout d'abord sur les files de réaction puis sur les réacteurs. $\texttt{genreact}(\mathcal{T}_1, \mathcal{T}_2)$ (resp. $\texttt{genreact}(\mathcal{C}_1, \mathcal{C}_2)$) retourne le réacteur $\mathcal{T}_2'$ (resp. la file de réaction $\mathcal{C}_2'$) correspondant à $\mathcal{T}_2$ (resp. $\mathcal{C}_2$) où les réactants qui ont réagi avec ceux de $\mathcal{T}_1$ (resp. $\mathcal{C}_1$) ont été supprimés.

La fonction `genrq` prend la forme

$$\texttt{genrq}(\mathcal{M}, \mathcal{T}_1, \mathcal{T}_2) = \mathcal{M}', \mathcal{R}$$

et renvoie la file d'exécution $\mathcal{R}$ résultant de la réaction des éléments de $\mathcal{T}_1$ avec ceux de $\mathcal{T}_2$. $\mathcal{M}$ est une machine contenant les localités dont les noms sont présents dans $\mathcal{T}_1$ ou $\mathcal{T}_2$ et qui seront passivés. $\mathcal{M}'$ correspond à la nouvelle machine où les localités passivées ont été supprimées. La définition de `genrq` utilise la fonction auxiliaire `genrqaux` définie sur les files de réaction qui prend la forme

$$\texttt{genrqaux}(\mathcal{M}, \mathcal{T}, \mathcal{C}_1, \mathcal{C}_2) = \mathcal{M}', \mathcal{R}$$

$$
\begin{aligned}
&\texttt{genreact}([]\ ,\mathcal{C}) = \mathcal{C}\\
&\texttt{genreact}(\mathcal{C},\ []) = \mathcal{C}\\
&\texttt{genreact}(\langle l,\widetilde{V}\rangle :: \mathcal{C},\langle l',\widetilde{V}'\rangle :: \mathcal{C}') = \langle l,\widetilde{V}\rangle :: \langle l',\widetilde{V}'\rangle :: \texttt{genreact}(\mathcal{C},\mathcal{C}')\\
&\texttt{genreact}(\langle \widetilde{x}\rangle \triangleright P :: \mathcal{C},\langle \widetilde{x}\rangle \triangleright Q :: \mathcal{C}') = \langle \widetilde{x}\rangle \triangleright P :: \langle \widetilde{x}\rangle \triangleright Q :: \texttt{genreact}(\mathcal{C},\mathcal{C}')\\
&\texttt{genreact}(\langle l,\widetilde{V}\rangle :: \mathcal{C},\langle \widetilde{x}\rangle \triangleright Q :: \mathcal{C}') = \texttt{genreact}(\mathcal{C},\mathcal{C}')\\
&\texttt{genreact}(\langle \widetilde{x}\rangle \triangleright Q :: \mathcal{C},\langle l,\widetilde{V}\rangle :: \mathcal{C}') = \texttt{genreact}(\mathcal{C},\mathcal{C}')\\
&\texttt{genreact}(\langle l\rangle :: \mathcal{C},\langle l'\rangle :: \mathcal{C}') = \langle l\rangle :: \langle l'\rangle :: \texttt{genreact}(\mathcal{C},\mathcal{C}')\\
&\texttt{genreact}(\langle l\rangle :: \mathcal{C},\langle x\rangle \triangleright Q :: \mathcal{C}') = \texttt{genreact}(\mathcal{C},\mathcal{C}')\\
&\texttt{genreact}(\langle x\rangle \triangleright Q :: \mathcal{C},\langle l\rangle :: \mathcal{C}') = \texttt{genreact}(\mathcal{C},\mathcal{C}')
\end{aligned}
$$

$$\texttt{genreact}(\mathcal{T}_1,\mathcal{T}_2) = \{k \to \texttt{genreact}(\mathcal{C}_1,\mathcal{C}_2) \mid k \to \mathcal{C}_1 \in \mathcal{T}_1 \wedge k \to \mathcal{C}_2 \in \mathcal{T}_2\}$$

FIG. 4.9 – *Fonction* ***genreact*** *sur les files de réaction et les réacteurs*

Cette fonction est définie de manière récursive figure 4.10. Dans un premier temps, on peut ignorer le premier argument qui n'est utilisé que dans le cas RCT.P. On détaille maintenant quelques cas de la définition de `genrqaux`. La règle

$$\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\mathcal{C},\mathcal{C}') = \texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\mathcal{C}',\mathcal{C})$$

traduit le fait que l'opération de faire réagir deux files de réaction est symétrique. Cela nous permet de ne pas dupliquer certaines des règles. La règle

$$\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\langle l,\widetilde{V}\rangle :: \mathcal{C},\langle l',\widetilde{V}'\rangle :: \mathcal{C}') = \mathcal{M},\ []$$

calcule le produit des réactions de deux files contenant des messages (rappelons qu'une file ne contient que des réactants du même type). Les éléments ne peuvent pas réagir et par conséquent le produit des réactions est une liste vide []. De plus, il n'y a pas eu de passivation, dont la machine $\mathcal{M}$ n'est pas modifiée. La règle

$$\frac{\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\mathcal{C},\mathcal{C}') = \mathcal{M}',\mathcal{R}}{\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\langle l,\widetilde{V}\rangle :: \mathcal{C},\langle \widetilde{x}\rangle \triangleright Q :: \mathcal{C}') = \mathcal{M}',Q\{\widetilde{V}/\widetilde{x}\} :: \mathcal{R}}\ \text{RCT.C}$$

calcule récursivement le produit des réactions de deux files contenant des messages pour l'une et des récepteurs pour l'autre. On peut facilement montrer que $\mathcal{M}' = \mathcal{M}$. De plus, on peut supposer que les vecteurs $\widetilde{V}$ et $\widetilde{x}$ ont même taille, si les processus considérés sont typés par un système de type comme ceux définis au chapitre 3.

Nous finissons avec le cas RCT.P qui correspond à la passivation.

$$\frac{\begin{array}{c}\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M}',\mathcal{C},\mathcal{C}') = \mathcal{M}'',\mathcal{R} \qquad \mathbf{tree}(\mathcal{M}_r,l,p,m)\\ \mathcal{M}_r = l : p[\mathcal{R}_r]_{m,S} : \mathcal{T}_r \mid \mathcal{M}'_r \qquad M_r \stackrel{\equiv}{\to}{}^* M_* \qquad \mathcal{R}_1 = \mathcal{R}_l^{\nabla}(\mathcal{T}_p) \qquad \mathcal{R}_2 = \mathcal{R}_p^{\triangle}(\mathcal{T}_r)\end{array}}{\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M}_r \mid \mathcal{M}',\langle l\rangle :: \mathcal{C},\langle x\rangle \triangleright Q :: \mathcal{C}') = \mathcal{M}'',Q\{\mathbf{reify}(l,\mathcal{M}_* :: \mathcal{R}_1)/x\} :: \mathcal{R}_2 :: \mathcal{R}}\ \text{RCT.P}$$

Cette règle calcule le produit de la réaction d'un réactant $\langle l \rangle$ avec un réactant $\langle x \rangle \triangleright Q$, puis de manière récursive le produit des files $\mathcal{C}$ et $\mathcal{C}'$. Le terme $\mathcal{M}_r$ correspond à un arbre de localités de racine l. Le terme $\mathcal{M}'$ correspond à un ensemble de localités qui n'interviennent pas dans la passivation. Pour pouvoir être réifiée, la machine $\mathcal{M}_r$ doit être sous forme normale (afin d'être conforme aux réductions du kell calcul). On obtient ce résultat grâce à une relation de sous-réduction $\stackrel{\equiv}{\rightarrow}$ définie en section 4.3.3.

Extraction de messages d'un réacteur Nous expliquons maintenant le rôle des termes $\mathcal{R}_1 = \mathcal{R}_l^{\triangledown}(\mathcal{T}_p)$ et $\mathcal{R}_2 = \mathcal{R}_p^{\triangle}(\mathcal{T}_r)$. La passivation d'une localité nécessite de retraiter certains messages. Expliquons ce point. Considérons le processus du Kell calcul suivant

$$P = a[b[r^{\uparrow}\langle\rangle] \mid s^{\downarrow}\langle\rangle]$$

Il n'y a pas de récepteur pouvant recevoir les messages sur r et s. r sera routé vers le réacteur de a, alors que s sera routé vers le réacteur de b. Ces deux messages resteront dans ces réacteurs jusqu'à ce qu'un récepteur correspondant soit défini dans a ou b. Supposons maintenant que, après que les messages sur r et s aient été placés dans les réacteurs, la localité de nom b soit passivée. Cela peut arriver par exemple si l'on exécute le processus suivant

$$P' = a[s^{\downarrow^b}\langle\rangle \mid b[r^{\uparrow}\langle\rangle] \mid b[x] \triangleright Q]$$

Il serait incorrect de laisser ces deux messages dans leur réacteur (le message sur s est dans le réacteur de b et le message sur r dans le réacteur de a) puisque cela permettrait à ces messages de réagir ultérieurement, même si b est situé à un autre endroit de l'arbre des localités (ce qui violerait la propriété de correction de la machine abstraite). Pour interdire cette possibilité, nous replaçons ces messages dans la file d'exécution de leur localité d'origine[2]. Pour ce faire, nous utilisons deux *fonctions d'extractions* $\mathcal{R}_{\cdot}^{\triangle}(.)$ et $\mathcal{R}_{\cdot}^{\triangledown}(.)$ définies en figure 4.11. La première prend comme argument le réacteur d'une *localité passivante* et renvoie une file (reconstruite) de messages. Ces messages ont été initialement envoyés à la localité passivante mais n'ont pas réagi. Dans la règle RCT.P, ils sont donnés par le terme $\mathcal{R}_1$ et sont replacés dans la file d'exécution de la *localité passivée* pour leur permettre d'être à nouveau traités lorsque la localité sera réactivée. La fonction $\mathcal{R}_{\cdot}^{\triangledown}(.)$ effectue l'opération inverse. A l'aide du réacteur de la localité passivée, elle régénère la liste des messages qui ont été précédemment envoyés vers cette localité et n'ont pas réagi. Dans la règle RCT.P, ces messages correspondent au terme $\mathcal{R}_2$ et sont replacés dans la file d'exécution de la localité passivante.

Pour finir, la fonction `genrq` utilise `genrqaux` pour faire réagir toutes les files des réacteurs $\mathcal{T}_1$ et $\mathcal{T}_2$ entre elles.

4.3.3 Relation de réduction

La relation de réduction est définie via une relation de congruence structurelle. La relation d'équivalence structurelle, définie figure 4.12 permet essentiellement de voir les

2. Remarquons que l'information de localité d'origine dans les réactants de type message est seulement utilisée pour les messages provenant d'une sous-location. Pour des raisons d'uniformité, nous la conservons pour tous les messages.

$$\begin{array}{l}
\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\mathcal{C},\mathcal{C}') = \texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\mathcal{C}',\mathcal{C}) \\
\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},[]\,,\mathcal{C}) = \mathcal{M},[] \\
\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\langle l,\widetilde{V}\rangle :: \mathcal{C},\langle l',\widetilde{V}'\rangle :: \mathcal{C}') = \mathcal{M},[] \\
\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\langle l\rangle :: \mathcal{C},\langle l'\rangle :: \mathcal{C}') = \mathcal{M},[] \\
\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\langle \widetilde{x}\rangle \triangleright P :: \mathcal{C},\langle \widetilde{x}\rangle \triangleright Q :: \mathcal{C}') = \mathcal{M},[]
\end{array}$$

$$\frac{\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\mathcal{C},\mathcal{C}') = \mathcal{M}',\mathcal{R}}{\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M},\langle l,\widetilde{V}\rangle :: \mathcal{C},\langle \widetilde{x}\rangle \triangleright Q :: \mathcal{C}') = \mathcal{M}',Q\{\widetilde{V}/\widetilde{x}\} :: \mathcal{R}}\ \textsc{Rct.C}$$

$$\frac{\begin{array}{c}\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M}',\mathcal{C},\mathcal{C}') = \mathcal{M}'',\mathcal{R} \qquad \mathbf{tree}(\mathcal{M}_r,l,p,m) \\ \mathcal{M}_r = l : p[\mathcal{R}_r]_{m,S} : \mathcal{T}_r \mid \mathcal{M}'_r \qquad M_r \stackrel{\equiv}{\to}{}^* M_* \qquad \mathcal{R}_1 = \mathcal{R}_l^{\triangledown}(\mathcal{T}_p) \qquad \mathcal{R}_2 = \mathcal{R}_p^{\triangle}(\mathcal{T}_r)\end{array}}{\texttt{genrqaux}(\mathcal{T}_p,\mathcal{M}_r \mid \mathcal{M}',\langle l\rangle :: \mathcal{C},\langle x\rangle \triangleright Q :: \mathcal{C}') = \mathcal{M}'',Q\{\mathbf{reify}(l,\mathcal{M}_* :: \mathcal{R}_1)/x\} :: \mathcal{R}_2 :: \mathcal{R}}\ \textsc{Rct.P}$$

$$\frac{\texttt{genrqaux}(\{k \to \mathcal{C}_2\} \uplus \mathcal{T}'_2,\mathcal{M},\mathcal{C}_1,\mathcal{C}_2) = \mathcal{M}',\mathcal{R}' \qquad \texttt{genrq}(\mathcal{M}',\mathcal{T}'_1,\mathcal{T}'_2) = \mathcal{M}'',\mathcal{R}''}{\texttt{genrq}(\mathcal{M},\{k \to \mathcal{C}_1\} \uplus \mathcal{T}'_1,\{k \to \mathcal{C}_2\} \uplus \mathcal{T}'_2) = \mathcal{M}'',\mathcal{R}' :: \mathcal{R}''}$$

FIG. 4.10 – *Fonctions* ***genrq*** *et* ***genrqaux***

machines comme des multi-ensembles de localités et les termes S comme des ensembles de noms de localités.

Nous définissons figure 4.14 une relation de réduction $\to$ sur les machines abstraites. Les règles R.CTX et R.STR permettent de considérer une machine abstraite comme un ensemble de localités s'exécutant en parallèle. Les autres règles décrivent les réductions possibles d'une localité en fonction du premier élément de leur file d'exécution.

Les règles R.NEW, R.PAR et R.NIL sont standards et correspondent à une implantation de la relation de congruence structurelle du Kell calcul. La règle R.NEW correspond à une création de nom. R.PAR implante la composition parallèle de processus et R.NIL le caractère neutre du processus nul. Par exemple, la règle R.PAR peut se lire : "pour exécuter $(P \mid Q)$ dans la localité l, on exécute d'abord P et on place Q en fin de file d'exécution", soit

$$l : p[(P \mid Q) :: \mathcal{R}]_{m,S} : \mathcal{T} \to l : p[P :: \mathcal{R} :: Q]_{m,S} : \mathcal{T}\ \textsc{R.Par}$$

Les règles R.MSG.LOC, R.MSG.UP et R.MSG.DOWN routent un message vers le réacteur correspondant, et si une réaction peut avoir lieu, le produit de la réaction est placé en fin de file d'exécution. Ces trois règles utilisent la fonction $\mu(M,h)$ définie figure 4.13, qui étant donné un message M et son origine h renvoie le réacteur qui correspond à l'association correspondante. Nous expliquons la règle R.MSG.UP donnée par

$$\frac{M = r\langle \widetilde{V}\rangle^{\uparrow} \qquad \mu(M,l') \otimes \mathcal{T} = \mathcal{R}'',\mathcal{T}''}{l : p[\mathcal{R}]_{m,(l',S)} : \mathcal{T} \mid l' : q[M :: \mathcal{R}']_{l,S'} : \mathcal{T}' \to l : p[\mathcal{R} :: \mathcal{R}'']_{m,(l',S)} : \mathcal{T}'' \mid l' : q[\mathcal{R}']_{l,S'} : \mathcal{T}''}$$

Le message M est inséré dans le réacteur $\mathcal{T}$ de sa localité destination. L'expression $\mu(M,l') \otimes \mathcal{T}$ nous renvoie le nouveau réacteur $\mathcal{T}''$ et une file d'exécution $\mathcal{R}''$. Cette liste ne

$$
\begin{array}{rl}
[]^{p,q}_{msg} = [] & \text{extraction des messages d'une file} \\
(\langle l,\widetilde{V}\rangle :: \mathcal{C})^{p,q}_{msg} = p^{\downarrow^q}\langle\widetilde{V}\rangle :: \mathcal{C}^{p,q}_{msg} & \\
(\_ :: \mathcal{C})^{p,q}_{msg} = \mathcal{C}^{p,q}_{msg} &
\end{array}
$$

$$
\begin{array}{rl}
[]^{p}_{l} = [] & \text{extraction des messages d'origine } l \\
(\langle l,\widetilde{V}\rangle :: \mathcal{C})^{p}_{l} = p^{\uparrow}\langle\widetilde{V}\rangle :: \mathcal{C}^{p}_{l} & \\
(\_ :: \mathcal{C})^{p}_{l} = \mathcal{C}^{p}_{l} &
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{R}^{\triangle}_{q}(\mathcal{T}) = \mathop{::}_{p^{\triangle}\to\mathcal{C}\in\mathcal{T}} \mathcal{C}^{p,q}_{msg} & \text{régénère les messages d'un parent vers son enfant } q \\
\mathcal{R}^{\nabla}_{l}(\mathcal{T}) = \mathop{::}_{p^{\nabla}\to\mathcal{C}\in\mathcal{T}} \mathcal{C}^{p}_{l} & \text{régénère les messages de l'enfant } l
\end{array}
$$

FIG. 4.11 – *Régénération des messages*

$$
\frac{\mathcal{M} =_{\alpha} \mathcal{N}}{\mathcal{M} \equiv \mathcal{N}}\ \text{R.SE.}\alpha
\qquad\qquad
\frac{P =_{\alpha} P' \qquad S \equiv S'}{l : p[P]_{m,S} \equiv l : p[P']_{m,S'}}\ \text{R.SE.CTX}
$$

FIG. 4.12 – *Equivalence structurelle sur les machines*

contient au plus qu'un élément, si un récepteur correspondant à ce message était présent dans le réacteur.

La règle R.TRIG

$$
\frac{\mathcal{M} : \tau(T) \otimes \mathcal{T} = \mathcal{R}',\mathcal{T}' \qquad S' = S \setminus \mathbf{locnames}(\mathcal{M})}{l : p[T :: \mathcal{R}]_{m,S} : \mathcal{T} \mid \mathcal{M} \to l : p[\mathcal{R} :: \mathcal{R}']_{m,S'} : \mathcal{T}'}\ \text{R.TRIG}
$$

correspond au cas où un récepteur T se trouve en tête de la file d'exécution d'une localité. Elle fait appel à la fonction τ définie figure 4.13, qui transforme un récepteur en un réacteur à un élément. Elle traite uniformément le cas où T est un récepteur simple ou un récepteur de passivation. La complexité de la règle est reportée dans l'expression $\mathcal{M} : \tau(T) \otimes \mathcal{T}$. Dans le premier cas, $\mathcal{M} = \emptyset$ et tout se passe comme pour la règle R.LOC. Dans le deuxième cas, s'il y a passivation, $\mathcal{M}$ est l'arbre des localités à passiver, qui n'apparaît plus après réduction. De plus, les noms de localités de $\mathcal{M}$ sont supprimées de S.

La règle R.KELL

$$
\frac{l' \textbf{ fresh} \qquad \mathcal{M} = l' : q[P ::[]]_{l,\emptyset} : \emptyset \qquad \mathcal{M} : \mu(q,l') \otimes \mathcal{T} = \mathcal{M}' : \mathcal{R}',\mathcal{T}'}{l : p[q[P] :: \mathcal{R}]_{m,S} : \mathcal{T} \to l : p[\mathcal{R} :: \mathcal{R}']_{m,(S,l')} : \mathcal{T} \mid \mathcal{M}'}\ \text{R.KELL}
$$

correspond au cas où un kell $q[P]$ se trouve en tête de la file d'exécution d'une localité. On définit le terme

$$\mathcal{M} = l' : q[P ::[]]_{l,\emptyset} : \emptyset$$

qui correspond à la localité à créer. On calcule alors l'expression suivante :

$$\mathcal{M} : \mu(q,l') \otimes \mathcal{T} = \mathcal{M}' : \mathcal{R}',\mathcal{T}'$$

Si une association de la forme

$$q^{\times} \to \langle x \rangle \triangleright Q :: \mathcal{C}$$

est présente dans le réacteur $\mathcal{T}$, on aura nécessairement $\mathcal{M}' = \mathbf{0}$ et $\mathcal{R}'$ qui contiendra la continuation Q. Sinon, $\mathcal{R}' = []$ et $\mathcal{M}' = \mathcal{M}$.

La règle R.Act

$$\frac{\mathcal{M}'_r \mid \mathcal{M} : \mathcal{T}_r \otimes \mathcal{T} = \mathcal{M}'',\mathcal{R}',\mathcal{T}' \qquad \mathbf{fresh}(l,n,\mathcal{M}_r) = l : j[\mathcal{R}_r]_{l',S'} : \mathcal{T}_r \mid \mathcal{M}'_r}{l : p[\mathbf{reify}(n,\mathcal{M}_r) :: \mathcal{R}]_{m,S} : \mathcal{T} \mid \mathcal{M} \to l : p[\mathcal{R} :: \mathcal{R}_r :: \mathcal{R}']_{m,(S,S')} : \mathcal{T}' \mid \mathcal{M}''} \text{ R.Act}$$

traite la réactivation d'une machine passivée $\mathbf{reify}(l,\mathcal{M}_r)$. $\mathbf{fresh}(l,n,\mathcal{M}_r)$, définie comme dans la section précédente, renomme les noms de localité de $\mathcal{M}_r$ par des noms frais, et le nom n en l. La réactivation d'une machine réifiée nécessite d'insérer le contenu du réacteur de la localité de plus haut niveau de la machine passivée ($\mathcal{T}_r$) dans le réacteur de la localité destination ($\mathcal{T}$). Cette règle est la seule où l'on insère plus d'un élément dans un réacteur. L'opération d'insertion est effectuée par l'expression

$$\mathcal{M}'_r \mid \mathcal{M} : \mathcal{T}_r \otimes \mathcal{T} = \mathcal{M}'',\mathcal{R}',\mathcal{T}'$$

Notons que plusieurs sous-localités peuvent être passivées en une seule réduction, et qu'il peut s'agir de localités présentes dans $\mathcal{M}'_r$ ou dans $\mathcal{M}$.

$$\begin{aligned}
\mu(p\langle \widetilde{V} \rangle,l) &= \{p^{\leftrightarrow} \to \langle \widetilde{V},l \rangle\} \\
\mu(p^q\langle \widetilde{V} \rangle,l) &= \{p^{\triangle} \to \langle \widetilde{V},l \rangle\} \\
\mu(p^{\uparrow}\langle \widetilde{V} \rangle,l) &= \{p^{\nabla} \to \langle \widetilde{V},l \rangle\} \\
\mu(p,l) &= \{p^{\times} \to \langle l \rangle\}
\end{aligned}$$

$$\begin{aligned}
\tau(p\langle \widetilde{x} \rangle \triangleright P) &= \{p^{\leftrightarrow} \to \langle \widetilde{x} \rangle \triangleright P\} \\
\tau(p^{\uparrow}\langle \widetilde{x} \rangle \triangleright P) &= \{p^{\triangle} \to \langle \widetilde{x} \rangle \triangleright P\} \\
\tau(p^{\downarrow}\langle \widetilde{x} \rangle \triangleright P) &= \{p^{\nabla} \to \langle \widetilde{x} \rangle \triangleright P\} \\
\tau(p[x] \triangleright P) &= \{p^{\times} \to \langle x \rangle \triangleright P\}
\end{aligned}$$

Fig. 4.13 – *Conversion des messages et récepteurs vers des associations de réacteur*

On définit la relation $\overset{\equiv}{\to}$ comme la plus petite relation vérifiant les règles R.Nil, R.Par, R.New, R.Kell, R.Act, R.Ctx et R.Str. Cette relation était utilisée dans la définition de $\otimes$ et par conséquent, ces deux définitions sont mutuellement récursives.

$$\frac{}{l : p[\mathbf{0} :: \mathcal{R}]_{m,S} : \mathcal{T} \to l : p[\mathcal{R}]_{m,S} : \mathcal{T}} \text{ R.Nil}$$

$$\frac{}{l : p[(P \mid Q) :: \mathcal{R}]_{m,S} : \mathcal{T} \to l : p[P :: \mathcal{R} :: Q]_{m,S} : \mathcal{T}} \text{ R.Par}$$

$$\frac{j \textbf{ fresh}}{l : p[(\nu a.P) :: \mathcal{R}]_{m,S} : \mathcal{T} \to l : p[P\{i/a\} :: \mathcal{R}]_{m,S} : \mathcal{T}} \text{ R.New}$$

$$\frac{M = q\langle \widetilde{V} \rangle \qquad \mu(M,l) \otimes \mathcal{T} = \mathcal{R}',\mathcal{T}'}{l : p[M :: \mathcal{R}]_{m,S} : \mathcal{T} \to l : p[\mathcal{R} :: \mathcal{R}']_{m,S} : T'} \text{ R.Msg.Loc}$$

$$\frac{M = r\langle \widetilde{V} \rangle^{\uparrow} \qquad \mu(M,l') \otimes \mathcal{T} = \mathcal{R}'',\mathcal{T}''}{\begin{array}{c} l : p[\mathcal{R}]_{m,(l',S)} : \mathcal{T} \mid l' : q[M :: \mathcal{R}']_{l,S'} : \mathcal{T}' \to \\ l : p[\mathcal{R} :: \mathcal{R}'']_{m,(l',S)} : \mathcal{T}'' \mid l' : q[\mathcal{R}']_{l,S'} : \mathcal{T}'' \end{array}} \text{ R.Msg.Up}$$

$$\frac{M = r\langle \widetilde{V} \rangle^{\downarrow^{r'}} \qquad \mu(M,l) \otimes \mathcal{T}' = \mathcal{R}'',\mathcal{T}''}{\begin{array}{c} l : p[M :: \mathcal{R}]_{m,(l',S)} : \mathcal{T} \mid l' : q[\mathcal{R}']_{l,S'} : \mathcal{T}' \to \\ l : p[\mathcal{R}]_{m,S} : \mathcal{T} \mid l' : q[\mathcal{R}' :: \mathcal{R}'']_{l,S'} : \mathcal{T}'' \end{array}} \text{ R.Msg.Down}$$

$$\frac{\mathcal{M} : \tau(T) \otimes \mathcal{T} = \mathcal{R}',\mathcal{T}' \qquad S' = S \setminus \mathbf{locnames}(\mathcal{M})}{l : p[T :: \mathcal{R}]_{m,S} : \mathcal{T} \mid \mathcal{M} \to l : p[\mathcal{R} :: \mathcal{R}']_{m,S'} : \mathcal{T}'} \text{ R.Trig}$$

$$\frac{l' \textbf{ fresh} \qquad \mathcal{M} = l' : q[P ::[]]_{l,\emptyset} : \emptyset \qquad \mathcal{M} : \mu(q,l') \otimes \mathcal{T} = \mathcal{M}' : \mathcal{R}',\mathcal{T}'}{l : p[q[P] :: \mathcal{R}]_{m,S} : \mathcal{T} \to l : p[\mathcal{R} :: \mathcal{R}']_{m,(S,l')} : \mathcal{T} \mid \mathcal{M}'} \text{ R.Kell}$$

$$\frac{\mathcal{M}'_r \mid \mathcal{M} : \mathcal{T}_r \otimes \mathcal{T} = \mathcal{M}'',\mathcal{R}',\mathcal{T}' \qquad \textbf{fresh}(l,n,\mathcal{M}_r) = l : j[\mathcal{R}_r]_{l',S'} : \mathcal{T}_r \mid \mathcal{M}'_r}{l : p[\textbf{reify}(n,\mathcal{M}_r) :: \mathcal{R}]_{m,S} : \mathcal{T} \mid \mathcal{M} \to l : p[\mathcal{R} :: \mathcal{R}_r :: \mathcal{R}']_{m,(S,S')} : \mathcal{T}' \mid \mathcal{M}''} \text{ R.Act}$$

$$\frac{\mathcal{M} \to \mathcal{M}'}{\mathcal{M} \mid \mathcal{N} \to \mathcal{M}' \mid \mathcal{N}} \text{ R.Ctx} \qquad \frac{\mathcal{M} \equiv \mathcal{M}' \qquad \mathcal{M}' \to \mathcal{M}'' \qquad \mathcal{M}'' \equiv \mathcal{M}'''}{\mathcal{M} \to \mathcal{M}'''} \text{ R.Str}$$

FIG. 4.14 – *Règles de réduction*

4.3.4 Correction

Nous conjecturons, sans les prouver, deux propriétés relatives à la correction de la machine abstraite raffinée. La première traduit la préservation de la structure d'arbre au cours des réduction. La deuxième est une propriété de correction qui stipule que les réductions de la machine raffinée correspondent toujours à des réductions du Kell calcul.

Théorème 6 (Préservation de la bonne formation) *Si M est bien formée et $M \to M'$, alors M' est bien formée.*

Pour énoncer la propriété de correction, on utilise une fonction de traduction d'un processus vers une machine raffinée.

Définition 4.3.1 $[\![P]\!] = \mathbf{r} : \mathbf{rn}[P]_{\mathbf{rp},\emptyset} : \emptyset$

On définit comme avant la notion de forme normale d'une machine, et une relation d'équivalence $\doteq$.

Lemme 4.3.2 *$M \not\xrightarrow{\equiv}$ si et seulement si, il existe N_* tel que $M = N_*$.*

Lemme 4.3.3 (Forme normale) *Si M est un terme de machine, alors il existe M'_* tel que $M \xrightarrow{\equiv}{}^* M'_*$. De plus, si $M \equiv \xrightarrow{\equiv}{}^* M''_*$ alors $M'_* \cong M''_*$. Par ailleurs, $M \not\xrightarrow{\equiv}$ si et seulement si $M = M'_*$ pour un certain M'_*.*

Ce résultat va nous permettre de définir une notion d'équivalence simple que nous utiliserons pour établir une propriété de correction.

Définition 4.3.4 (Equivalence) *On dit que deux machines M et N sont* équivalentes*, et l'on note $M \doteq N$, si elles ont la même normale forme (modulo $\cong$).*

Lemme 4.3.5 *$\doteq$, $\cong$ et $\equiv$ sont des relations d'équivalence.*

Lemme 4.3.6 *Si l'on considère la restriction de $\equiv$ aux machines bien formées, on a $\equiv \subseteq \cong \subseteq \doteq$.*

On peut maintenant énoncer la propriété de correction suivante.

Proposition 4.3.7 (Correction) $[\![P]\!] \to M \implies P \to P'$ *avec* $[\![P']\!] \doteq M$.

Cette propriété de correction garantit que les réduction de la machine raffinée correspondent à des réductions du calcul (ou de manière équivalente à des réduction de la première version de machine abstraite). Ce résultat est relativement faible : par exemple, une machine qui ne ferait rien le vérifierait. Par ailleurs, La propriété réciproque n'est malheureusement plus vraie, tout simplement parce que l'on a perdu le non-déterminisme des réceptions de messages. Considérons le terme :

$$a\langle V\rangle \mid a\langle V'\rangle \mid (a\langle x\rangle \triangleright P)$$

Si ce terme est exécuté par la machine abstraite raffinée, seule une communication est possible. Ce phénomène n'est pas choquant. Il traduit simplement le déterminisme de l'ordonnancement des processus. On le retrouve dans Pict ou dans JoCaml. Néanmoins, il enfreint la propriété de complétude.

4.3.5 Exemples

Nous illustrons maintenant le fonctionnement de la machine abstraite à travers une série d'exemples correspondant aux actions de base du Kell calcul. Nous commençons avec des communications simples, locales et à travers des frontières de kell. Nous illustrons ensuite une action de passivation et finalement, nous détaillons sur un exemple la règle R.Act, qui est la règle la plus complexe du système.

Message local

On considère le processus

$$P = \nu r.r\langle V\rangle \mid (r\langle x\rangle \triangleright Q)$$

qui modélise un envoi de message local sur un canal de communication privé. Nous allons montrer comment la réduction

$$P \to \nu r.Q\{V/x\} = P'$$

est implantée par le calcul de machine abstraite. L'état initial d'une machine abstraite exécutant P est donné par

$$\mathcal{M}_0 = \mathbf{r} : \mathbf{rn}[P]_{\mathbf{rp},\emptyset} : \emptyset$$

D'après les règles R.New et R.Par, on a les deux étapes de réduction suivantes :

$$\begin{aligned}\mathcal{M}_0 &\to \mathbf{r} : \mathbf{rn}[i\langle V'\rangle \mid (i\langle x\rangle \triangleright Q')]_{\mathbf{rp},\emptyset} : \emptyset \text{ avec } V' = V\{i/r\} \text{ et } Q' = Q\{i/r\}\\ &\to \mathbf{r} : \mathbf{rn}[i\langle V'\rangle :: i\langle x\rangle \triangleright Q']_{\mathbf{rp},\emptyset} : \emptyset = \mathcal{M}_1\end{aligned}$$

En regardant le premier élément de la file d'exécution, on détermine que la seule règle pouvant s'appliquer à $\mathcal{M}_1$ est R.Msg.Loc. Elle conduit à insérer le message $i\langle V'\rangle$ dans le réacteur vide. On a alors $\mathcal{M}_1 \to \mathcal{M}_2$ avec

$$\mathcal{M}_2 = \mathbf{r} : \mathbf{rn}[i\langle x\rangle \triangleright Q']_{\mathbf{rp},\emptyset} : \mathcal{T}_2 \text{ et } \mathcal{T}_2 = \{i^{\leftrightarrow} \to \langle \mathbf{r}, V'\rangle\}$$

On a alors par la règle R.Trig

$$\mathcal{M}_2 \to \mathbf{r} : \mathbf{rn}[Q'\{V'/x\}]_{\mathbf{rp},\emptyset} : \emptyset = \mathcal{M}_3$$

La machine $\mathcal{M}_3$ correspond bien à une machine initialisée avec le processus P'. Plus précisément, on a $\mathbf{r} : \mathbf{rn}[P']_{\mathbf{rp},\emptyset} : \emptyset \doteq \mathcal{M}_3$.

Message vers le haut

On considère le processus

$$P = a[r^{\uparrow}\langle V\rangle] \mid r^{\downarrow}\langle x\rangle \triangleright Q$$

qui modélise l'émission d'un message émis depuis un sous-kell vers son parent. On a la réduction

$$P \to a[] \mid Q\{V/x\} = P'$$

L'état initial d'une machine abstraite exécutant P est donné par :

$$\mathcal{M}_0 = \mathbf{r} : \mathbf{rn}[P]_{\mathbf{rp},\emptyset} : \emptyset$$

On a par la règle R.Par :

$$\mathcal{M}_0 \to \mathbf{r} : \mathbf{rn}[a[r^{\uparrow}\langle V\rangle] :: r^{\downarrow}\langle x\rangle \triangleright Q]_{\mathbf{rp},\emptyset} : \emptyset = \mathcal{M}_1$$

On peut appliquer la règle R.Kell. Elle crée une nouvelle localité avec un nom frais et l'initialise avec le contenu du kell a. De plus, le réacteur de $\mathbf{r}$ est mis à jour. Cela correspond à la réduction $\mathcal{M}_1 \to \mathcal{M}_2$ avec :

$$\begin{aligned}\mathcal{M}_2 =& \mathbf{r} : \mathbf{rn}[r^{\downarrow}\langle x\rangle \triangleright Q]_{\mathbf{rp},h} : \mathcal{T}_{\mathbf{r}} \mid h : a[r^{\uparrow}\langle V\rangle]_{\mathbf{r},\emptyset} : \emptyset \\ \mathcal{T}_{\mathbf{r}} =& \{a^{\times} \to \langle h\rangle\}\end{aligned}$$

Deux réductions sont maintenant possibles. On peut appliquer la règle R.Msg.Up à h, ou la règle R.Trig à $\mathbf{r}$. Nous considérons seulement le deuxième cas (les deux réductions mènent au même résultat après une réduction supplémentaire). On a $\mathcal{M}_2 \to \mathcal{M}_3$ avec :

$$\begin{aligned}\mathcal{M}_3 =& \mathbf{r} : \mathbf{rn}[[]]_{\mathbf{rp},h} : \mathcal{T}_{\mathbf{r}} \mid h : a[r^{\uparrow}\langle V\rangle]_{\mathbf{r},\emptyset} : \emptyset \\ \mathcal{T}_{\mathbf{r}} =& \{a^{\times} \to \langle h\rangle, r^{\nabla} \to \langle x\rangle \triangleright Q\}\end{aligned}$$

On applique maintenant la règle R.Msg.Up à h. Comme la file de réaction associée à la clé a se trouvant dans la localité parente de h n'est pas vide et contient un récepteur, il y a une réaction dont le résultat est inséré dans la file d'exécution de $\mathbf{r}$. On a donc finalement $\mathcal{M}_3 \to \mathcal{M}_4$ avec :

$$\begin{aligned}\mathcal{M}_4 =& \mathbf{r} : \mathbf{rn}[Q\{V/x\}]_{\mathbf{rp},h} : \mathcal{T}_{\mathbf{r}} \\ & |h : a[]_{\mathbf{r},\emptyset} : \emptyset \\ \mathcal{T}_{\mathbf{r}} =& \{a^{\times} \to \langle h\rangle\}\end{aligned}$$

Comme avant, on a bien $\mathbf{r} : \mathbf{rn}[P']_{\mathbf{rp},\emptyset} : \emptyset \doteq \mathcal{M}_4$.

Passivation

Trois règles peuvent conduire à la passivation d'une sous-localité : R.Trig, R.Kell et R.Act. Les exemples suivants illustrent les deux premiers cas.

Passivation dans la règle R.Trig Nous commençons par un exemple où l'opération de passivation est obtenue par la règle R.Trig. Considérons le processus

$$P = a[\mathbf{0}] \mid a[x] \triangleright Q$$

et la réduction $P \to P'$ avec :

$$P' = Q\{\mathbf{0}/x\}$$

L'état initial de la machine abstraite est

$$\mathcal{M}_0 = \mathbf{r} : \mathbf{rn}[P]_{\mathbf{rp},\emptyset} : \emptyset$$

Par la règle R.Par, on a :

$$\mathcal{M}_0 \to \mathbf{r} : \mathbf{rn}[a[\mathbf{0}] :: a[x] \triangleright Q]_{\mathbf{rp},\emptyset} : \emptyset = \mathcal{M}_1$$

La règle qui s'applique alors est R.Kell. Elle crée une nouvelle localité avec un nom frais et l'initialise avec le contenu du kell a. De plus, le réacteur de $\mathbf{r}$ est mis à jour. Ainsi, on a $\mathcal{M}_1 \to \mathcal{M}_2$ avec :

$$\begin{aligned}\mathcal{M}_2 =& \mathbf{r} : \mathbf{rn}[a[] \triangleright Q]_{\mathbf{rp},h} : \mathcal{T}_{\mathbf{r}} \mid h : i[]_{\mathbf{r},\emptyset} : \emptyset \\ \mathcal{T}_{\mathbf{r}} =& \{a^{\times} \to \langle h \rangle\}\end{aligned}$$

Par application de la règle R.Trig, on a $\mathcal{M}_2 \to \mathcal{M}_3$ avec :

$$\begin{aligned}\mathcal{M}_3 =& \mathbf{r} : \mathbf{rn}[Q\{\mathbf{reify}(\mathcal{M}_r,h)/x\}]_{\mathbf{rp},\emptyset} : \emptyset \\ \mathcal{M}_r =& h : i[]_{\mathbf{r},\emptyset} : \emptyset\end{aligned}$$

Passivation dans la règle R.Kell Supposons maintenant que le processus à exécuter soit :

$$P = \nu a.(a[x] \triangleright Q) \mid a[\mathbf{0}]$$

Comme avant, la machine qui exécute ce processus se réduit vers :

$$\mathcal{M}_1 = \mathbf{r} : \mathbf{rn}[a[x] \triangleright Q :: i[\mathbf{0}]]_{\mathbf{rp},\emptyset} : \emptyset$$

Maintenant, contrairement au cas précédent, le récepteur doit être évalué alors qu'il n'y a pas de sous-localité de nom a. On a :

$$\mathcal{M}_2 = \mathbf{r} : \mathbf{rn}[a[\mathbf{0}]]_{\mathbf{rp},\emptyset} : \mathcal{T}_2$$

$$\mathcal{T}_2 = \{a^{\times} \to \langle x \rangle \triangleright Q\}$$

La sous-localité de nom i sera passivée lors de sa création comme spécifié par la règle R.Kell.

Passivation et régénération de messages Un dernier exemple décrit la régénération des messages. Considérons une machine $\mathcal{M}$ avec un réacteur vide et une file d'exécution initiale

$$\mathcal{R} = i[j_1^{\uparrow}\langle\rangle] :: j_2^{\downarrow^i}\langle\rangle :: i[] \triangleright Q$$

La première réduction crée une sous-localité de nom i. Puis, le message sur j_1 est inséré dans le réacteur de la localité de plus haut-niveau et le message sur j_2 est inséré dans le réacteur de la localité nouvellement crée, ayant pour nom i. Sans détailler les réductions, on a $\mathcal{M} \to^* \mathcal{M}_1$ avec :

$$\begin{aligned}\mathcal{M}_1 =& \mathbf{r} : \mathbf{rn}[a[] \triangleright Q]_{\mathbf{rp},h} : \mathcal{T}_{\mathbf{r}} \mid h : i[]_{\mathbf{r},\emptyset} : \mathcal{T}_h \\ \mathcal{T}_{\mathbf{r}} =& \{a^{\times} \to \langle h \rangle, j_1^{\nabla} \to \langle\langle\rangle, h\rangle\} \\ \mathcal{T}_h =& \{j_2^{\triangle} \to \langle\langle\rangle, \mathbf{r}\rangle\}\end{aligned}$$

La file d'exécution de h étant vide, la seule réduction possible correspond à R.TRIG appliquée à $\mathbf{r}$. Elle implique la passivation de h. Comme expliqué précédemment, les messages envoyés entre h and $\mathbf{r}$ qui n'ont pas trouvé de récepteur avec qui réagir doivent être régénérés à partir des réacteurs et remis dans les files d'exécution des localités qui les ont émis. Les messages régénérés sont donnés par les expressions suivantes :

$$\mathcal{R}_1 = \mathcal{R}_h^{\triangledown}(\mathcal{T}_\mathbf{r}) = j_1^{\uparrow}\langle\rangle$$
$$\mathcal{R}_2 = \mathcal{R}_i^{\triangle}(\mathcal{T}_h) = j_2^{\downarrow^i}\langle\rangle$$

De la règle R.TRIG, on a alors :

$$\mathcal{M} = \mathbf{r} : \mathbf{rn}[Q\{\mathbf{reify}(\mathcal{M}_r,h)/x\} :: \mathcal{R}_2]_{\mathbf{rp},\emptyset} : \emptyset$$
$$\mathcal{M}_r = h : i[\mathcal{R}_1]_{\mathbf{r},\emptyset} : \emptyset$$

Les messages non traités ont bien été replacés dans leur localité d'origine.

Activation La règle R.ACT spécifie la réactivation d'une valeur $\mathbf{reify}(l,\mathcal{M}_r)$ réifiant un sous-arbre de localités $\mathcal{M}_r$ ayant pour racine la localité l. Comme dans la section 4.2, elle décrit l'ajout de la file d'exécution contenue dans la localité l dans la localité ou elle est réactivée, ainsi que l'ajout des sous-localités de l. Remarquons qu'il est nécessaire de remplacer les noms de localités dans $\mathcal{M}_r$ par des noms frais puisqu'une même machine réifiée peut être activé plusieurs fois. Le point délicat est que le réacteur de l doit être réévalué dans ce nouveau contexte. Un nombre arbitraire[3] de réactions (et passivations) peut avoir lieu dans une seule réduction liée à cette règle. Alors que dans les autre règles, le premier réacteur en argument de $\otimes$ était un réacteur à un seul élément, il est ici un réacteur quelconque.

Nous illustrons cette règle par un exemple. Soit $V_r = \mathbf{reify}(h,\mathcal{M}_r)$ une valeur réifiant la machine $\mathcal{M}_r$ définie de la manière suivante :

$$\mathcal{M}_r = h : i[]_{l,\emptyset} : \mathcal{T}_r \text{ avec } \mathcal{T}_r = \{j^{\times} \to \langle x\rangle \triangleright Q, k^{\leftrightarrow} \to \langle h,V\rangle\}$$

La machine $\mathcal{M}_r$ a une file d'exécution vide, mais son réacteur contient un récepteur de passivation sur le nom j, et un message local sur le nom k. Nous considérons maintenant la machine suivante :

$$\mathcal{M}_\mathbf{r} = \mathbf{r} : \mathbf{rn}[V_r ::[]]_{\mathbf{rp},h} : \mathcal{T}_\mathbf{r} \mid h : j[]_{\mathbf{r},\emptyset} : \emptyset$$
$$\mathcal{T}_\mathbf{r} = \{j^{\times} \to \langle h\rangle, k^{\leftrightarrow} \to \langle x\rangle \triangleright R\}$$

Cette machine est formée de deux localités, mais seule une réduction est possible, correspondant à la règle R.ACT et la localité $\mathbf{r}$. Le réacteur de la localité $\mathbf{r}$ contient deux entrées. La première, de la forme $j^{\times} \to \langle h\rangle$ indique l'existence d'une sous-localité de nom j. La deuxième, $k^{\leftrightarrow} \to \langle x\rangle \triangleright R$, correspond à un récepteur local sur le nom k.

La réactivation de V_r entraîne la passivation d'un sous-kell j, et une communication de message sur k. Après réduction, la file d'exécution, le réacteur et les sous-localités de $\mathbf{r}$ sont donnés respectivement par $\mathcal{R}'$, $\mathcal{T}'$ et $\mathcal{M}'$ avec :

$$h : j[]_{\mathbf{r},\emptyset} : \emptyset : \mathcal{T}_r \otimes \mathcal{T}_\mathbf{r} = \mathcal{M}', \mathcal{R}', \mathcal{T}'$$

3. Majoré par le nombre de réactants dans les deux réacteurs

D'après la définition de $\otimes$, on a :

$$\begin{aligned}\mathcal{R}' &= R\{V_r/x\} :: Q\{\mathbf{reify}(h,\mathcal{M})/x\} ::[]\\ \mathcal{T}' &= \emptyset\\ \mathcal{M}' &= \mathbf{0}\end{aligned}$$

On en déduit :

$$\mathcal{M}_{\mathbf{r}} \rightarrow \mathbf{r} : \mathbf{rn}[\mathcal{R}']_{\mathbf{rp},\emptyset} : \emptyset$$

4.3.6 Discussion

Comparaison avec Pict L'opération consistant à router un message ou un récepteur nécessite une recherche dans une table afin de trouver la file de réaction correspondante. Dans le cas du π-calcul, cette opération de recherche peut-être évitée. En effet, lorsqu'un nom frais est crée dans la règle R.New, le nom frais peut être simplement un pointeur vers la file de réactants. Cela n'est pas possible dans notre cas puisque l'association entre un nom et une file de réaction dépend de la localité. Une optimisation possible consisterait à associer l'adresse d'une file de réactant à un nom dès que c'est possible et d'invalider cette association lorsqu'elle n'est plus valable, par exemple lors du déplacement d'un nom d'une localité vers une autre.

Réductions administrative Dans la spécification de machine abstraite de la section 4.2, certaines réductions étaient qualifiées d'administratives. Ces réductions administratives étaient des réductions qui n'avaient pas de correspondance avec des réductions du Kell calcul (elle correspondait soit à l'équivalence structurelle, soit à la relation de sous-réduction). Elles étaient utiles pour énoncer les propriétés de correction en nous permettant d'utiliser une forme forte de bisimilarité. Dans cette version raffinée, nous ne pouvons plus faire une distinction aussi simple entre les différents types de réduction. Considérons par exemple la règle R.Msg.Loc : elle conduira à une réduction (au sens du Kell calcul) si et seulement si il y a un récepteur dans le réacteur prêt à réagir avec ce message. Dans le cas contraire, il s'agira plutôt d'une réduction administrative. Si nous voulions garder la distinction entre ces deux types de réduction, il faudrait dupliquer la plupart des règles.

Récepteurs répliqués Nous n'avons considéré ici que des récepteurs non répliqués. On pourrait implanter des récepteurs répliqués de manière native de façon similaire. Plutôt que d'enlever de la file de réactants un récepteur qui vient de réagir, nous le laissons dans la file. Toutefois, comme dans Pict, pour assurer une certaine forme d'équité, nous le replaçons en fin de file pour laisser aux autres récepteurs une chance de réagir à leur tour.

Substitution L'opération de substitution est implantée de manière classique à l'aide d'environnement. Afin de garder la présentation formelle lisible, nous ne les avons pas introduit dans les règles de réduction. Néanmoins, nous donnons maintenant quelques éléments. On appelle *fermeture* un couple de la forme (σ,P) où P est un processus de machine et σ un environnement (*i.e.* une liste d'associations entre des noms et des valeurs). Les processus dans la file d'exécution ainsi que les continuations de récepteurs sont en fait des fermetures. L'opération de substitution $P\{V/x\}$ est implantée comme un ajout d'une association en tête d'un environnement $((x,V) :: \sigma,P)$.

Messages orientés Nous avons fait le choix d'utiliser des messages orientés afin de pouvoir router les messages vers un réacteur unique. Dans la section 4.2, un message $a\langle V\rangle$ pouvait réagir avec n'importe quel récepteur écoutant sur le nom a à condition qu'il soit défini localement, dans la localité parente, ou dans une sous-localité. Dans ces conditions, si aucun récepteur n'est disponible, il y a essentiellement deux possibilités pour traiter ce message. La première est d'essayer de router le message plus tard (ce qui constitue la stratégie employée dans [32]). La seconde est d'informer les émetteurs de messages potentiels qu'un nouveau récepteur est disponible et qu'ils peuvent essayer de renvoyer leurs messages. Aucune de ces stratégies ne semble satisfaisante. Si l'on contraint les messages à être orientés, on peut déterminer quelle est la localité qui doit recevoir le message, même si aucun récepteur n'est présent. De plus, en tant que primitive d'un langage de programmation, l'utilisation de messages orientés semble naturelle. Plusieurs problèmes subsistent néanmoins. Premièrement, comme expliqué plus haut, en cas de passivation on doit replacer les messages dans leur localité d'origine. Il s'agit d'une opération potentiellement coûteuse, mais ce coût est acceptable si l'on considère que la passivation est dans tous les cas une opération inhabituelle, à l'inverse des envois de messages. Le deuxième problème provient du fait que plusieurs sous-localités peuvent avoir le même nom. Considérons par exemple le processus :

$$r^{\downarrow a}\langle\rangle \mid a[P] \mid a[Q]$$

Nous avons deux façons de router le message sur r. Si aucun des kells de nom a ne possède de récepteur sur r, nous avons le même problème que si l'on utilise des messages non orientés. La solution adoptée est d'interdire à deux sous-kells frères d'avoir le même nom Un compromis serait de pouvoir statiquement vérifier qu'un tel cas de figure ne peut pas se produire, à l'aide par exemple d'un système de type. Nous laissons cette possibilité comme un travail futur.

4.4 Implantation

Nous avons implanté un prototype (disponible librement [40]) en OCaml qui réalise un interprète pour le langage `CHALK` défini à partir du πK-calcul. Ce prototype implante de manière fidèle la machine abstraite de la section 4.3. On trouvera dans l'annexe C la documentation de `CHALK` qui comprend une présentation complète du langage, des exemples d'utilisations et une description de l'implantation. Nous présentons dans cette section les aspects principaux de `CHALK` et en particulier les différences avec le calcul. Puis nous parlons brièvement de l'implantation.

4.4.1 Le langage `CHALK`

Le lange `CHALK` est essentiellement une extension typée du jK-calcul avec des messages orientés et un ensemble enrichi de valeurs. Les valeurs sont soit de base (entier, listes, chaînes), d'ordre supérieur (abstractions de processus, processus passivés) ou des expressions construites à partir d'opérateurs classiques, tels que des opérateurs arithmétiques ou des primitives de sérialisation/désérialisation. Le système de type correspond au système de type générique présenté au chapitre 3, auquel des types de bases ont été ajoutés. Toutefois, les annotations de typage sont facultatives. Par ailleurs, la sémantique répartie du

langage est définie de manière rigoureuse à l'aide d'une modélisation en Kell calcul des services de réseau.

Un interprète est modélisé par un contexte $\texttt{vmid}[\texttt{Lib} \mid u[P]]$ exécutant un programme utilisateur P en conformité avec les règles de réduction de la machine abstraite. Le programme P peut interagir avec une *librairie* spécifiée par le processus `Lib`. Une configuration répartie d'interprètes est alors spécifiée comme suit. Si l'on exécute les programmes $P_0, \ldots, P_n$ dans différents interprètes, le comportement résultant est donné par le terme suivant :

$$\texttt{Net} \mid \texttt{vmid}_0[\texttt{Lib} \mid u[P_0]] \mid \ldots \mid \texttt{vmid}_n[\texttt{Lib} \mid u[P_n]]$$

où l'on suppose que les noms `vmid` sont distincts. Les processus `Lib` modélisent les librairies locales alors que le processus `Net` modélise le réseau. Dans notre implantation, ils sont essentiellement définis comme suit (en omettant les annotations de typage) :

$$\begin{aligned}
\texttt{Lib} &= (\texttt{send}^{\downarrow}\langle x,y\rangle \diamond \texttt{send}^{\uparrow}\langle x,y\rangle) \\
&\quad \mid (\texttt{recv}^{\uparrow}\langle x\rangle \diamond \texttt{msg}^{\downarrow^u}\langle x\rangle) \\
&\quad \mid (\texttt{echo}^{\downarrow}\langle x\rangle \diamond \mathbf{0}) \\
&\quad \mid (\texttt{echo_int}^{\downarrow}\langle x\rangle \diamond \mathbf{0}) \\
\texttt{Net} &= \texttt{send}^{\downarrow}\langle x,y\rangle \diamond \texttt{rcv}^{\downarrow^x}\langle y\rangle
\end{aligned}$$

Ces processus spécifient un environnement permettant l'échange de messages asynchrones entre des interprètes. Le nom `vmid` permet d'envoyer des messages à des kells désignés de manière unique. Nous n'avons pas chercher à spécifier le comportement des récepteurs `echo` et `echo_int`, qui sort du cadre de ce travail.

Nous donnons maintenant un exemple de programme réparti. Un serveur est en attente de données provenant d'un client sur le canal `receive`. Lorsqu'il reçoit un message, il le désérialise et l'exécute. Le client exécute un programme dans une location `a`. Ce programme est stoppé durant son exécution, transformé en une chaîne de caractère et envoyé au serveur. La syntaxe de `CHALK` est légèrement différente de celle du Kell calcul. La construction `on` introduit un récepteur répliqué, et la construction `once` correspond à un récepteur simple. L'opérateur `vm` est un constructeur qui construit un identifiant de machine virtuelle (typiquement pour localiser un serveur de nom) à partir d'une adresse et d'un port. Nous utilisons des fonctions de sérialisation et de désérialisation qui convertissent des valeurs arbitraires vers des chaînes et réciproquement.

```
(* client.kcl *)
new a in new signal in new move in
let serverid = vm ("localhost", 8000) in
a [
 on signal < i > {
   if i = 5 then
      env.move < > | signal < i +1 >
   else if i = 10 then
       nil
   else
       ( env.echo_int < i > | signal < i + 1 > )
```

```
 } |
 signal < 0 >
]
|
once move dn < > {
  once a [ x ] {
     env.send < serverid, marshall x : proc >
  }
}
|
on echo_int dn < x > { env.echo_int < x > }
```

```
(* server.kcl *)
on receive up < s > {
   let agent = unmarshall s as proc in agent
}
```

L'exécution du serveur et du client dans deux machines différentes donnent le résultat suivant :

```
$ chalk client.kcl -p 7000
01234
```

```
$ chalk server.kcl -p 8000
6789
```

4.4.2 Implantation

Les programmes des utilisateurs sont tout d'abord analysés syntaxiquement, puis typés à l'aide d'un algorithme d'inférence de type simple basé sur un algorithme d'unification. Ils sont ensuite exécutés par un moteur d'exécution, qui suit précisément les réductions de la machine abstraite.

Par rapport à la machine abstraite de la section 4.3, le parcours de l'arbre est déterministe. Les conditions de fraîcheur de noms (prédicat **fresh**) dans les règles M.S.Cell, M.S.Act et M.S.New sont implantées soit à travers l'utilisation de pointeurs au niveau du langage d'implantation (création de localité), soit par un générateur de noms globalement uniques pour les noms crées par l'instruction *new*.

Le processus `Lib` est implanté par un module séparé. Le programme utilisateur accède aux services qui correspondent à des fonctions natives, mais auxquelles on accède de manière transparente à partir de P comme pour n'importe quel autre récepteur. De manière similaire, des fonctions natives, vues comme des processus dans `Lib`, peuvent générer des messages vers u pouvant être reçus P.

4.5 Travaux connexes

Il y a eu de nombreux papiers récents concernant la description et l'implantation de machines abstraites pour des calculs de processus répartis. On peut notamment citer l'implantation JoCaml du Join calcul [27, 24], l'implantation du calcul des ambiants en Join

calcul. [28], Nomadic Pict [68, 66], la machine abstraite pour le M-Calcul [32], la machine à Fusion [30], les machines abstraites PAN et GCPAN pour les Safe Ambients [56, 36], la machine abstraite CAM pour les Channel Ambients [52]. De plus, il y a également eu diverses implantations de calculs distribué tels que le Seal calcul [67], Klaim [6], ou DiTyCO [46].

Notre spécification de machine abstraite a été conçue afin d'être indépendante de l'environnement dans lequel elle est implantée, et en particulier des services réseau qu'il fournit. Ainsi, elle peut être utilisée dans des configurations très différentes. Par exemple, elle n'impose pas une correspondance entre les localités de plus haut niveau avec des sites physiques, comme c'est le cas dans [28, 24, 32]. De la même façon, elle n'impose pas d'introduire une nouvelle abstraction de localité pour décrire les sites physiques comme c'est le cas dans [36, 56]. Cette séparation entre le comportement de la machine abstraite et la sémantique du réseau n'est pas présente dans les autre machines abstraites pour des calculs de processus répartis.

Le calcul de Seal [16] et le M-calcul [59] sont les seuls calculs qui partagent avec le Kell calcul une combinaison d'actions locales et de localités hiérarchiques et pourrait obtenir un résultat similaire d'indépendance entre machine abstraite et services de réseau. Toutefois, aucune machine abstraite n'est décrite pour le calcul de Seal (seule une implantation est mentionnée dans [67]), et la machine abstraite pour le M-calcul décrite dans [32] repose sur un modèle de réseau fixe, et une correspondance entre localités de plus haut niveau et site physiques. Les calculs qui présupposent un modèle de réseau plat, tel que Nomadic Pict, DiTyCO, Klaim ont des machines abstraites et des implantations qui présupposent une configuration physique donnée et un modèle de réseau correspondant.

Le machine des Fusion [30] implante le calcul de Fusion général, dans lequel il n'y a pas de notion de localité. Néanmoins, la machine abstraite en elle même se base sur un modèle de réseau asynchrone fixé. De plus, de part la nature des communication dans le calcul des Fusion, la machine abstraite nécessite un protocole de migration non trivial pour réaliser la synchronisation en présence de sites multiples. A l'inverse de nos calculs et machines abstraites, cela empêche les programmes du calcul de Fusion d'utiliser directement, et sans coût, des services de réseaux de bas niveau tels que des services de datagrammes.

Les machines abstraites et implantations pour des calculs de processus distribué avec hiérarchie de localités, autre que le calcul de Seal et le M-calcul, essentiellement le Join calcul et les différents calculs d'ambiants, doivent implanter des primitives de migrations, ce qui nécessite une dépendance par rapport à un modèle de réseau fixé. Par exemple la machine abstraite de JoCaml pour le Join calcul réparti [24] dépend d'un modèle de réseau à passage de messages asynchrones et d'une interprétation particulière des hiérarchies de localités (les localités de plus haut-niveau sont interprétées comme des sites physiques). Les machines abstraites PAN [56] et GCPAN [36] pour les "Safe Mobile Ambients" dépendent également d'un modèle de réseau à passage de messages asynchrones pour spécifier la migration des ambiants entre les sites (correspondant à l'interprétation de la primitive `open` du calcul des ambiants), et de l'introduction d'une notion de site d'exécution, indépendante des ambiants. La machine abstraite pour "Channel Ambient" [52] laisse en fait l'implantation de ses primitives `in` et `out` non spécifiée.

4.6 Conclusion

Nous avons présenté deux versions d'une machine abstraite pour une instance du Kell calcul, le πK-calcul, et discuté brièvement son implantation en OCaml.

L'originalité de cette machine abstraite repose sur le fait qu'elle est indépendante des services de réseau pouvant être utilisés pour son implantation répartie. En effet, comme notre prototype d'implantation en OCaml l'atteste, nous pouvons isoler les services de réseau fourni par un environnement donné sous la forme d'une librairie dans le langage hôte qui peut être réifiée sous la forme d'un processus standard du Kell calcul. Cette librairie peut être utilisée par les programmes du Kell calcul. Cela signifie que notre machine abstraite, de la même façon que le calcul, ne contient pas intrinsèquement d'abstraction sophistiquée pour la programmation répartie. En particulier, cela démontre que le calcul et sa machine abstraite fournissent des bases flexibles pour développer ces abstractions. Finalement, cette indépendance a l'avantage de simplifier les preuves de corrections de la machine abstraite qui ne dépendent plus de la preuve de correction de protocoles répartis complexes.

La version raffinée correspond de manière précise à l'implantation en OCaml. L'utilisation de messages orientés en conjonction avec l'unicité des noms de sous-kell frères permet un routage déterministe des message, et une implantation efficace des communications. Cette version raffinée souffre de deux limitations.

- La condition d'unicité des noms de sous-kells frères introduit une classe d'erreurs d'exécution. Nous pensons qu'un système de type simple pourrait permettre de garantir cette unicité. De manière alternative, le système de type garantissant l'unicité de tous les noms de cellules, présenté au chapitre 3 permettrait également d'atteindre ce résultat.
- Nous n'avons pas pour l'instant prouvé de propriétés de correction générale. De telles propriétés s'avèrent d'autant plus souhaitables que les règles de réduction sont relativement complexes. Nous avons conjecturé une propriété de correction, sans la prouver bien que le résultat semble atteignable en suivant les mêmes techniques que pour la machine simple. Par contre, en raison du caractère déterministe de la machine, la propriété de complétude est fausse et il serait intéressant d'avoir une propriété alternative garantissant par exemple une condition d'équité.

Nous avons mentionné en section 2.8 une extension du Kell calcul [37] qui autorise le partage de kells. Le partage semble être indispensable pour pouvoir utiliser le calcul comme un langage de programmation un tant soit peu réaliste. Typiquement, on aimerait partager des librairies entre plusieurs kells, ou encore disposer de membranes génériques définies à un seul endroit. Un autre point intéressant serait la possibilité de modéliser une *répartition verticale*, c'est à dire la possibilité d'avoir un kell qui s'exécute sur un site différent que son parent.

Concernant l'implantation, plusieurs améliorations sont nécessaires si l'on veut atteindre l'objectif initial qui était d'évaluer le Kell calcul comme base d'un langage de programmation par composants répartis. En plus de permettre le partage, il est nécessaire de disposer de plus de primitives de programmation. Une solution serait d'utiliser un langage hôte. Par exemple, un kell pourrait servir naturellement de structure d'accueil pour un programme écrit par exemple en OCaml. C'est l'approche suivie dans [51], ou le π-calcul est utilisé pour composer des composants primitifs écrits en Java. Dans notre

cas, la difficulté résiderait notamment en la passivation d'un tel processus. On pourrait néanmoins utiliser le système de type donné en 3.2 pour interdire, ou limiter la passivation de ces processus.

Chapitre 5

Instrumentation d'un modèle de programmation par composants

5.1 Introduction

Depuis deux décennies, de nombreuses infrastructures à base de composants sont apparues. Ces infrastructures sont utilisées communément pour la construction de systèmes logiciels variés, notamment des applications Web (EJB [21], CCM [48]), des intergiciels (dynamicTAO [41], OpenORB [9]), ou encore des systèmes d'exploitation (OSKit [25], THINK [22]). Dans ce chapitre, nous présentons un système de type pour Fractal [10], un modèle de programmation par composants destiné à la mise en oeuvre de systèmes logiciels tels que des intergiciels ou des systèmes d'exploitation. Le modèle Fractal est directement orienté vers la pratique et a été utilisé dans diverses applications de taille réelle. Un exemple d'utilisation est donné par le système Dream [43] : une infrastucture basée sur le modèle Fractal et dédiée à la construction d'intergiciels spécialisés dans la communication par message. Nous présentons dans ce chapitre un système de type spécialisé permettant de déceler certaines erreurs dans l'analyse d'assemblages de composants Dream. Notre système de type est basé sur [54, 55]. En conclusion de ce chapitre, nous faisons le lien entre le système Fractal et le Kell calcul et discutons de futures directions de recherche pour appliquer le Kell calcul à Fractal.

Le chapitre s'organise de la manière suivante. En section 5.2 nous présentons le modèle Fractal. En section 5.3, nous introduisons le système Dream basé sur Fractal et un système de type pour vérifier la cohérences d'assemblages de composants. Finalement, nous discutons en 5.4 de futures directions de recherches liant Fractal et le Kell calcul.

5.2 Le modèle de programmation Fractal

Fractal est un modèle général de composants destiné à la mise en oeuvre de système logiciels complexes et reconfigurables. Il permet non seulement la programmation de ces systèmes, mais également leur supervision, leur maintenance ou encore leur déploiement. Ses principale caractéristiques sont les suivantes :

– Des composants (eventuellement construits à partir d'autres composants) rendent explicite l'*architecture à l'exécution du système*, et permettent de le considérer à

différents niveau d'abstraction.

- Des capacités d'introspection des composants permettent de superviser ou de contrôler une application.
- Des capacités de reconfiguration permettent le déploiement et la reconfiguration dynamique d'un système.

Fractal distingue deux types de composants : les composants *primitifs* (correspondant par exemple à des classes Java standards) et les composants *composites*. Ces derniers permettent de considérer des groupes de composants comme des composants simples.

Un composant possède un ou plusieurs *ports* (ou *interfaces*) qui correspondent à des points d'accès. Les ports peuvent être de deux types : des ports serveurs, qui correspondent à des points d'accès pour des appels de méthodes entrants, et des ports clients pour des appels de méthode sortants. Les signatures pour ces deux types de ports sont décrites par des interfaces Java standard, avec une indication supplémentaire de rôle (client ou serveur).

Un composant composite est constitué de deux parties : le *contenu* qui est un ensemble de sous-composants (dans le cas d'un composant composite); la *membrane* qui contient des intercepteurs et des contrôleurs accessibles via des *interfaces de contrôle*. Les principaux contrôleurs d'un composants sont :

- Des *contrôleurs de liaison* qui permettent de connecter ou de déconnecter les interfaces client d'un composant aux interfaces serveur d'un autre composant.
- Des *contrôleurs de cycle de vie* qui permettent de démarrer ou d'arrêter des sous-composants.
- Des *contrôleurs de contenu* qui permettent d'ajouter ou de supprimer des sous-composants.

La figure 5.1 illustre les différentes constructions que l'on peut trouver dans une architecture Fractal typique. Les boites grises foncées représentent la partie de contrôle (*i.e.* sa membrane) d'un composant, alors que l'intérieur de ces boites correspond à son contenu. Les flèches correspondent aux liaisons, et les structures en forme de T dépassant des boites grises sont les ports.

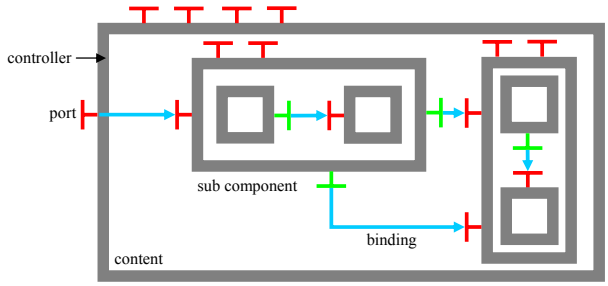


FIG. 5.1 – *Architecture d'un composant Fractal*

5.3 Typage de Dream

5.3.1 Introduction

Le système Dream [43] est un canevas logiciel pour la construction d'intergiciels permettant la communication par messages. Ce système est basé sur le modèle de composant Fractal [10] et son implémentation en Java. Il propose une librairie de composants qui peuvent être assemblés en utilisant le *langage de description d'architecture* (ADL) de Fractal et qui peuvent être utilisés pour implanter divers paradigmes de communication. De tels paradigmes peuvent être par exemple de type files de message, événement/réaction, publication/abonnement etc.

Un système construit à partir de composants Dream contient typiquement plusieurs composants pouvant échanger des *messages*, mais également les modifier (par exemple en y rajouter un time stamp), et qui peuvent réagir différemment en fonction du contenu des messages (par exemple, en routant les messages sur différents ports de sortie). Dans l'implémentation actuelle de l'infrastructure Dream, chaque message a un type `Message`, indépendamment de son contenu. En conséquence, certains assemblages de composants Dream, tout en étant bien typés et en compilant correctement en Java peuvent conduire à des erreurs d'exécution, typiquement lorsqu'un composant traite un message qui n'a pas la structure attendue.

Détecter de telles erreurs suffisamment tôt, au moment de l'écriture de la description d'architecture d'un assemblage de composant, permettrait un accroissement de la productivité des programmeurs utilisant l'infrastructure Dream. En d'autre terme, nous voudrions obtenir un langage de description d'architecture typé permettant le typage de composants et qui rejetterait des assemblages de composants manifestement incorrects.

Nous proposons ici un premier pas vers cet objectif en proposant un système de type pour les composants Dream, en nous concentrant sur des types de messages qui décrivent précisément la structure interne d'un message. Pour cela, nous adaptons des travaux existants sur les systèmes de type pour *enregistrements extensibles* [54, 55] et nous décrivons comment les composants et les assemblages de composants peuvent être typés. Le système de type résultant détecte un certain nombre d'erreurs qui peuvent être faire lorsque l'on écrit des descriptions de configurations Dream.

Le reste de cette partie est organisée de la manière suivante. En 5.3.2 nous décrivons l'infrastructure Dream. Nous identifions en 5.3.3 une classe d'erreurs courantes apparaissant dans des configurations Dream. Nous décrivons en 5.3.4 puis en 5.3.5 un système de type destiné à déceler de telles erreurs, Nous donnons des exemples d'utilisation de ce système de type en 5.3.6. Nous concluons en 5.3.7 en comparant ce travail à d'autres travaux connexes et en proposant plusieurs améliorations possibles.

5.3.2 L'infrastructure Dream

Les composants Dream sont des composants Fractal standards avec une une caractéristique supplémentaire : la présence d'interfaces d'entrée/sortie qui permettent aux composants Dream d'échanger des *messages*. Les messages sont des objets Java qui encapsulent des *chunks* nommés. Chaque chunk implante une interface qui définit son type. Par exemple, des messages qui nécessitent un ordonnancement causal ont un chunk qui implémente l'interface `Causal`. Cette interface définit des méthodes pour mettre à jour,

ou lire un horloge de matrice. Les messages sont toujours envoyé des sorties vers les entrées (figure 5.2 (a)). Il y a deux sortes d'entrée et sorties, correspondant aux deux types de connexions : *push* and *pull*. La connexion push correspond aux échanges de messages initiés par le port de sortie (figure 5.2 (b)). L'interaction pull correspond aux échanges de messages initiés par le port d'entrée (figure 5.2 (c)).

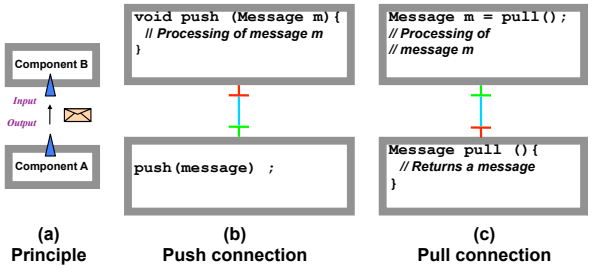


FIG. 5.2 – *Connexion d'interface d'entrée/sortie*

Dream fournit une libraire de composants qui encapsulent des fonctions et des comportements que l'on trouve couramment dans les intergiciels orientés messages. Ces composants peuvent être assemblés pour implanter des paradigmes de communication asynchrones variés : envoi de message, abonnement/publication, événement/réaction etc. Voici quelques exemples de composants Dream.

- Les *files de messages* sont utilisées pour stocker des messages. Elles diffèrent selon les manières dont les messages peuvent être triés : FIFO, LIFO, ordre causal, etc.
- Les *transformateurs* possèdent une entrée sur laquelle il reçoivent des messages, et une sortie sur laquelle ils délivrent des messages. Des transformateurs typiques sont par exemple des composants qui ajoutent des champs.
- Les *routeurs* possèdent une entrée et plus sorties (que l'on appelle également des "routes". Ils routent les messages qu'ils reçoivent sur leur entrée sur une pour plusieurs routes.
- Les *multiplexeurs* possèdent plusieurs entrée et une sortie; pour chaque message reçu sur une des entrées, le multiplexeur ajoute un chunk qui identifie l'entrée sur laquelle le message est arrivé; le multiplexeur transmet alors le message sur la sortie.
- Les *dupliqueurs* possèdent une entrée et plusieurs sorties. Ils copient les messages qu'ils reçoivent sur leur entrée sur toutes leurs sorties.
- Les *canaux* permettent les échanges de messages entre différents espaces d'adressage. Les canaux sont des composants répartis composite qui encapsulent au moins deux composants : un composant *ChannelOut* dont le rôle est d'envoyer les messages à l'autre espaces d'adressage, et un composant *ChannelIn* qui peut recevoir les messages envoyés par ChannelOut.

5.3.3 Problématique

Un intergiciel construit à l'aide de Dream est composé d'un ensemble de composants qui s'échangent des messages. Chaque composant effectue des traitements sur les messages ; ces traitements consistent principalement en l'ajout, la suppression et la modification de

chunks. Dans le système de types de base fourni par Julia, tous les messages ont le même type : l'interface Java `Message`. En conséquence, il n'est pas possible d'effectuer des vérifications de types portant sur les chunks qu'un message doit/peut posséder pour être traité par un composant. En effet, les seules vérifications de types possibles portent sur les types Java des interfaces utilisées par les composants pour communiquer (e.g. `Push` et `Pull`). De fait, un assemblage de composant perçu comme correct dans le système de types de Julia peut ne pas s'exécuter du fait que les composants reçoivent des messages ne possédant pas les chunks appropriés.

La figure 5.3 donne un exemple d'architecture erronée : le composant `readTS` attend des messages possédant un chunk dont le nom est *ts*, tandis que le composant `addTS` attend des messages n'ayant pas de chunk dont le nom est *ts*. Les deux composants recevant exactement les mêmes messages (dupliqués par le composant `Duplicator`), chaque message reçu provoquera une levée d'exception de la part d'un des deux composants.

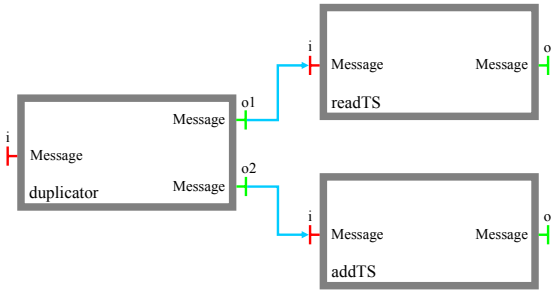


FIG. 5.3 – *Exemple d'architecture incorrecte.*

5.3.4 Un système de types pour le canevas Dream

Dans cette section, nous proposons un système de types polymorphe permettant de décrire la structure des messages (en terme de chunks). Ce système de types est une adaptation de travaux qui ont été faits dans le cadre des *enregistrements extensibles* [54, 55]. Nous décrivons ce système de type de façon intuitive en décrivant le typage des messages, le typage des composants et en montrant comment le système de type permet de détecter que l'architecture de l'exemple 5.3 est incorrecte. Une description formelle du système de type est donnée en 5.3.5.

Types de messages

Les *enregistrements* sont des structures de données utilisées dans plusieurs langages. Un enregistrement est un ensemble fini d'associations entre des *noms* et des *valeurs*. Dans [54, 55], Rémy décrit des extensions du langage ML permettant d'effectuer les opérations de base sur les enregistrements : ajout/retrait d'une association nom-valeur et concaténation d'enregistrements. Il définit également un système de types statique qui garantit que les programmes ne produiront pas d'erreurs tels que l'accès à des noms d'enregistrements inexistants. Les messages Dream peuvent être considérés comme des enregistrements associant des noms à des chunks. Les composants Dream peuvent, eux, être

considérés comme des fonctions polymorphes. Le polymorphisme est important pour les deux raisons suivantes : d'une part, un composant peut être utilisé dans différents contextes avec différents types. Par ailleurs, le polymorphisme permet de mettre en relation les types des interfaces clientes et serveurs, ce qui permet des descriptions plus précises du comportement des composants.

Le type d'un message est composé d'une liste de couples et d'une information supplémentaire. Le premier élément de chaque couple est un *nom*[1] ; le second élément peut prendre les formes suivantes :

- `abs` signifie que le message ne contient pas de chunk associé à *nom*.
- `pre`(τ) signifie que le message contient un chunk de type[2] τ associé à *nom*. Notons que τ peut être une *variable de type*, ce qui signifie qu'elle représente un type arbitraire de chunk.
- une *variable de champ* dont la valeur peut être `abs` où `pre`(τ).

L'information supplémentaire concerne tous les noms non présents dans la liste de couples. Elle peut prendre les formes suivantes :

- `abs` signifie que le type du message ne contient pas d'autres couples que ceux décrits dans la liste.
- une *variable de rangée* qui peut être `abs` où toute liste de couples.

$$\mu_1 = \{ipc : \texttt{pre}(\texttt{IPChunk}); mc : \texttt{pre}(\texttt{MonitoringChunk}); \texttt{abs}\}$$
$$\mu_2 = \{ipc : \texttt{pre}(\texttt{IPChunk}); mc : \texttt{pre}(\texttt{MonitoringChunk}); c : \texttt{abs}; \texttt{abs}\}$$
$$\mu_3 = \{a : \texttt{pre}(X); \texttt{abs}\}$$
$$\mu_4 = \{a : Y; \texttt{abs}\}$$
$$\mu_5 = \{ipc : \texttt{pre}(\texttt{IPChunk}); Z\}$$

FIG. 5.4 – *Exemples de types de messages.*

La figure 5.4 donne des exemples de types de messages. Un message de type μ_1 contient exactement deux chunks de types `IPChunk` et `MonitoringChunk`, respectivement associés aux noms *ipc* et *mc*. Le type μ_2 est identique au type μ_1. Le type μ_3 utilise une *variable de type* X, ce qui signifie qu'un message de type μ_3 doit contenir un chunk dont le nom est a, mais dont le type n'est pas spécifié. Le type μ_4 fait intervenir une *variable de champ* Y qui peut valoir `abs` ou `pre`(X), X étant une variable de type. Enfin, le type μ_5 fait intervenir une *variable de rangée* Z qui représente soit `abs`, soit tout liste de couples.

Types de composants

Le type d'un composant est représenté par un type de fonction polymorphe. Nous ne typons que les interfaces d'entrée et de sortie de messages (`Pull` et `Push`). La figure 5.5 donne des exemples de composants et leurs types associés. Le composant *duplicator* a une entrée i et deux sorties o_1 et o_2 : il reçoit des messages d'un type quelconque X sur son

1. Tous les noms de la liste doivent être distincts.
2. Le type d'un chunk est la classe Java d'implantation du chunk.

entrée et les duplique sur ses deux sorties. Le composant add_{ipc} ajoute un chunk `IPChunk` (associé au nom *ipc*) aux messages qu'il reçoit ; ces messages ne doivent pas posséder de chunk associé au nom *ipc*. Le composant $remove_{ipc}$ enlève le chunk associé au nom *ipc* si celui-ci est présent dans le message. Le composant *serializator* reçoit des messages d'un type quelconque X. Pour chaque message, il renvoie un nouveau message contenant un chunk de nom *bac*[3] qui contient la forme sérialisée du message reçu. Le type de ce chunk est créé à l'aide d'un constructeur de type, noté `ser`. Le composant *deserializator* implante le comportement dual du composant *serializator*.

$$
\begin{aligned}
\mathit{duplicator} &: \forall X.\{i : \{X\}\} \to \{o_1 : \{X\}; o_2 : \{X\}\} \\
\mathit{add}_{ipc} &: \forall X.\{i : \{\mathit{ipc} : \texttt{abs}; X\}\} \to \{o : \{\mathit{ipc} : \texttt{pre}(\texttt{IPChunk}); X\}\} \\
\mathit{remove}_{ipc} &: \forall X,Y.\{i : \{\mathit{ipc} : Y; X\}\} \to \{o : \{\mathit{ipc} : \texttt{abs}; X\}\} \\
\mathit{serializator} &: \forall X.\{i : \{X\}\} \to \{o : \{\mathit{bac} : \texttt{ser}(\{X\}); \texttt{abs}\}\} \\
\mathit{deserializator} &: \forall X.\{i : \{\mathit{bac} : \texttt{ser}(\{X\}); \texttt{abs}\}\} \to \{o : \{X\}\}
\end{aligned}
$$

FIG. 5.5 – *Exemples de types de composants*

Vérification de types

La figure 5.6 reprend l'exemple de la figure 5.3. Les ports des composants sont typés à l'aide du système de types décrit dans les paragraphes précédents. L'architecture représentée sur la figure est correctement typée si et seulement si le système d'équations suivant a une solution :

$$
\begin{aligned}
\{X\} &= \{ts : \texttt{pre}(\texttt{TSChunk}); Y\} \\
\{X\} &= \{ts : \texttt{abs}; Z\}
\end{aligned}
$$

Ce système n'a pas de solution. En conséquense, l'architecture décrite est incorrecte.

5.3.5 Description formelle du système de types

Syntaxe La syntaxe du système de types pour les messages est donnée sur la figure 5.7.

Nous supposons que les variables a,b,c,... représentent des éléments de l'ensemble dénombrable de noms N et que les variables L décrivent des ensembles finis de noms. Intuitivement, Dans une rangée, l'exposant L spécifie l'ensemble des noms qui ne doivent pas être utilisés. Ainsi, le type μ d'un message est défini par une rangée avec $\emptyset$ pour exposant, puisque le type d'un message doit décrire tous les champs possibles. Pour tous les champs qui ne sont pas dans l'exposant, une rangée spécifie si le champ est absent ou présent avec le type τ.

Afin de définir des ensembles de champs infinis, nous utilisons des *schémas de rangées* qui peuvent être `abs` ou des variables de rangées. Finalement, σ_B décrit des types de base, qui correspondent à des types Java dans Dream. Les exposants L interdisent la

3. "bac" est l'acronyme de *byte array chunk*.

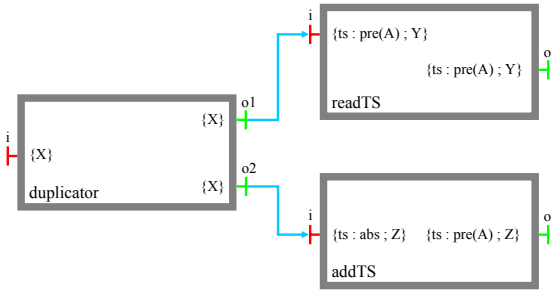


FIG. 5.6 – *Typage de l'architecture représentée sur la figure 5.3.*

$\rho^L ::= \xi^L \mid \texttt{abs}^L \mid a : \phi; \rho^{L \cup \{a\}}$	rangée
$\phi ::= \theta \mid \texttt{abs} \mid \texttt{pre}(\tau)$	champs
$\sigma_B ::= \texttt{A} \mid \texttt{B} \mid \ldots$	types de bases
$\mu ::= \{\rho^{\emptyset}\}$	types de messages
$\tau ::= \mu \mid \texttt{ser}(\tau) \mid \sigma_B \mid \alpha$	types généraux

FIG. 5.7 – *Syntaxe du système de types pour les messages Dream.*

redéfinition de champs. Ainsi le terme $\{a : \theta; (a : \theta'; \xi^L)\}$ n'est pas correct. De plus, nous écrivons $\{a : \phi; b : \phi'; \xi^L\}$ pour $\{a : \phi; (b : \phi'; \xi^L)\}$.

Nous présentons maintenant la syntaxe des types de composants.

$$\begin{array}{rll} I ::= i : \mu; I \quad | \quad \emptyset & & \text{Row} \\ C ::= \forall \widetilde{\alpha} \widetilde{\theta} \widetilde{\xi^L}.\{I^\emptyset\} \to \{I^\emptyset\} & & \text{Component} \end{array}$$

Nous définissons N' comme un ensemble dénombrable de *ports*, et nous utilisons les variables i,o éventuellement décorées pour décrire des ports. Un type de composant est composé d'un ensemble de *ports d'entrée* ainsi que de leur type, ainsi que d'un ensemble de ports de sorties avec leur type. Nous supposons que les ports d'un ensemble donné (ensemble d'entrée ou de sorties) sont distincts. Toutes les variables apparaissant dans des types de composants sont quantifiées. En 5.3.4, nous avions utilisé la même catégorie syntaxique pour les variables et avons omis les exposants. La raison est que la sorte des variables ainsi que les exposants peuvent être déduits automatiquement. Par exemple, le type $\forall X.\{i : \{X\}\} \to \{o : \{a : \mathtt{pre}(int); X\}\}$ n'est pas correct car il n'y a pas de variable L qui permette de le réécrire $\forall \xi^L.\{i : \{\xi^L\}\} \to \{o : \{a : \mathtt{pre}(int); \xi^L\}\}$. L devrait valoir $\emptyset$ dans la première occurrence de ξ et $\{a\}$ dans la deuxième.

Une *définition d'architecture* D est donnée par une liste de noms de composants et leur type, ainsi qu'une liste de connexion entre ports.

$$\begin{array}{rll} Cp ::= \epsilon \quad | \quad c : C,Cp & & \text{Composant} \\ Co ::= \epsilon \quad | \quad c.i = c.o,Co & & \text{Connexion} \\ D ::= (Cp,Co) & & \text{Définition d'architecture} \end{array}$$

Une définition d'architecture (Cp,Co) est bien formée si
- Les noms de composants dans Cp sont tous distincts.
- Pour chaque connexion $c.i = c'.o$ dans Co, $c : C$ appartient à Cp pour un certain C. De plus, i est un nom de port client de C et o un nom de port serveur.
- Un port peut être connecté au plus une fois.

Typage Nous notons T, l'ensemble des types de messages μ. Nous définissons une théorie équationnelle E sur T (en omettant les exposants) par les axiomes suivants :

$$\begin{aligned} a : \phi; (a' : \phi'; \rho) &= a' : \phi'; (a : \phi; \rho) \\ a : \phi; \mathtt{abs} &= a : \phi; (b : \mathtt{abs}; \mathtt{abs}) \\ a : \phi; \xi &= a : \phi; (b : \theta; \xi') \end{aligned}$$

Rémy a prouvé dans [54] que le problème de l'unification dans T modulo E est décidable et syntactique : tout problème d'unification qui possède une solution possède un unifiant plus général.

A partir d'une définition d'architecture $D = (Cp,Co)$, nous pouvons générer un ensemble d'équation $E(D)$. Tout d'abord, nous obtenons une liste Cp' en supprimant tous

les quantificateurs dans Cp. Nous supposons que les variables sont renomés de telle sorte que la même variable n'apparaisse jamais deux fois dans deux types distincts de composant. Nous pouvons voir Cp comme une fonction partielle de noms de composants vers des types de composants, et nous utilisons la notation $Cp(c)$. Pour un type de composant sans quantificateur C, nous notons $\text{CType}(C.i)$ pour le type associé à l'interface cliente i. De la même façons, nous définissons $\text{SType}(C.o)$. A l'aide de ces définitions, nous pouvons définir E comme suit:

$$E(Cp,Co) = \{\text{CType}(Cp'(c).i) = \text{SType}(Cp'(c).o), c.i = c.o \in Co\}$$

Un définition d'architecture D est typable si et seulement si E admet un unifiant.

5.3.6 Exemples d'utilisation

Exemple

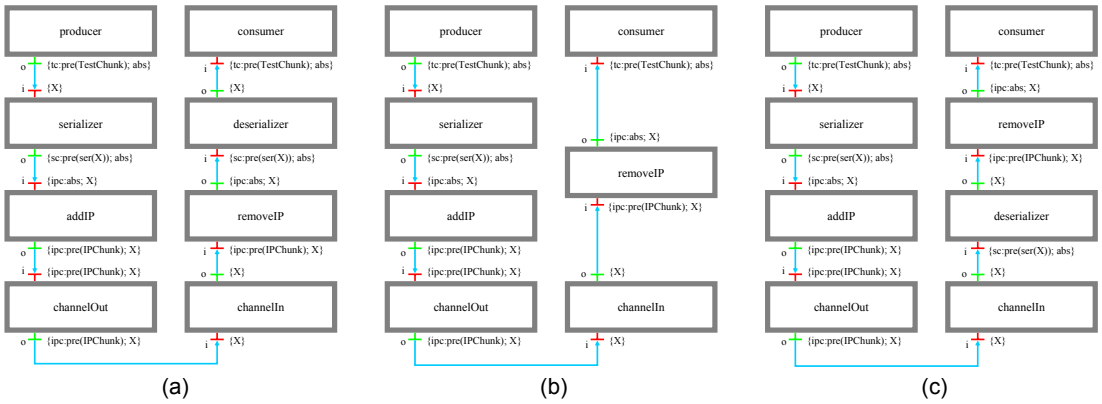


FIG. 5.8 – *Exemple: une pile de composants*

Le figure 5.8 (a) représente une pile de composants Dream. Le composant *producer* au sommet de la pile génère des messages consistant en un unique chunk de type `TestChunk` et nom *tc*. Son type est

$$producer : \{\} \rightarrow \{o : \{tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}\}\}$$

Le composant *serializer* retourne des messages possédant un chunk unique c qui est la forme sérialisée des messages reçus sur le port d'entrée i. Son type est

$$serializer : \forall X.\{i : \{X\}\} \rightarrow \{o : \{sc : \texttt{ser}(\{X\}); \texttt{abs}\}\}$$

Le composant *addIP* rajoute un chunk de type `IPChunk` de de nom *ipc* à message qui ne contient pas déjà un chunk *ipc*. Son type est

$$addIP : \forall X.\{i : \{ipc : \texttt{abs}; X\}\} \rightarrow \{o : \{ipc : \texttt{preIPChunk}; X\}\}$$

ChannelOut envoie les messages sur le réseau et attend qu'ils définissent au moins un champ nommé *ipc* de type `IPChunk`. La pile de droite effectue les actions symétriques.

Les figures 5.8 (b) et (c) montrent deux architectures incorrectes. Dans (b), le composant *deserializer* est manquant, dans (c), les composants *deserializer* et *addIP* sont inversés. L'architecture décrite en (a) est bien typée mais celles dans (b) et (c) ne le sont pas. Considerons par exemple l'architecture (b), nous déduisons des connexions entre composants les équations suivantes (notons que les variables liées ont été renommées).

$$\{tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}\} = \{U\} \tag{5.1}$$
$$\{sc : \texttt{pre}(\texttt{ser}U); \texttt{abs}\} = \{ipc : \texttt{abs}; Z\} \tag{5.2}$$
$$\{ipc : \texttt{pre}(\texttt{IPChunk}); T\} = \{ipc : \texttt{pre}(\texttt{IPChunk}); Z\} \tag{5.3}$$
$$\{ipc : \texttt{pre}(\texttt{IPChunk}); Z\} = \{Y\} \tag{5.4}$$
$$\{Y\} = \{ipc : \texttt{pre}(\texttt{IPChunk}); X\} \tag{5.5}$$
$$\{ipc : \texttt{abs}; X\} = \{tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}\} \tag{5.6}$$

De 5.6, nous déduisons

$$X = \texttt{pre}(\texttt{TestChunk}); \texttt{abs}$$

Puis, à partir de 5.5, nous avons

$$Y = ipc : \texttt{pre}(\texttt{IPChunk}); tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}$$

Il découle alors de 5.4 et 5.3 que

$$T = Z = tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}$$

Par ailleurs, nous déduisons de 5.2 que

$$Z = sc : \texttt{pre}(\texttt{ser}U); \texttt{abs}$$

Les deux dernières équations sont contradictoires et le système n'est pas typable.

Discussion et limitations

La limitation principale est que cette discipline de typage est trop restrictive pour typer certains composants Dream courants. Typiquement, ils peuvent exhiber différents comportements en fonction de la présence d'un certain label dans un message (par exemple des composants de type routeurs). Considérons par exemple un composant `route` qui route les messages qu'il reçoit sur son port client, sur deux ports serveur différents en fonction de la présence d'un label dans les messages en entrée. Nous voudrions pouvoir lui attribuer un type de cette forme:

$$\begin{aligned}\texttt{route} : \forall X.\{i : \{a : \texttt{pre}(\texttt{A}); X\}\} \rightarrow \{o_1 : \{X\}; o_2 : \{\texttt{abs}\}\} \\ \wedge \{i : \{a : \texttt{abs}; X\}\} \rightarrow \{o_1 : \{\texttt{abs}\}; o_2 : \{X\}\}\end{aligned}$$

De la même façon, certains composants peuvent émettre des messages de différents types :

$$\begin{aligned}\texttt{produce} : &\{\} \rightarrow \{o : \{a : \texttt{abs}; b : \texttt{B}; \texttt{abs}\}\} \\ &\wedge \{\} \rightarrow \{o : \{a : \texttt{A}; b : \texttt{abs}; \texttt{abs}\}\}\end{aligned}$$

Néanmoins, dans ces deux cas, nous pouvons trouver un type approximatif qui nous permet de typer une définition utilisant ces composants. Par exemple:

$$\texttt{route} : \forall XYZ.\{i : \{X\}\} \rightarrow \{o_1 : \{Y\}; o_2 : \{Z\}\}$$
$$\texttt{produce} : \forall XY.\{\} \rightarrow \{o : \{a : X; b : Y; \texttt{abs}\}\}$$

Ce faisant, nous perdons toutes garanties sur la correction de la définition d'architecture, puisque évidemment, le code des composants n'est pas conforme à ces types.

5.3.7 Travaux connexes et perspectives

Le système de type présenté dans cette section est adapté à la vérification de contraintes architecturales dans l'environnement à base de composants Dream. Une myriade de systèmes de type ont été proposés dans le cadre de langages généraux, comme ML ou Java, ou encore dans le cadre de calcul de processus pour vérifier toutes sortes de contraintes. Au contraire, la vérification de contraintes architecturales a reçu moins d'attention, même si l'on peut citer quelques travaux dans la dernière décennie. On peut mentionner par exemple le langage Wright [3], qui permet la vérification de contraintes comportementales dans une architecture logicielle, en faisant correspondre un composant à un rôle. Dans le langage ArchJava [2], un système de type à base d'ownership assure l'intégrité des communication dans un modèle de composant basé sur Java. On peut également citer [15] qui utilise des contrats comportementaux pour des assemblages de composants. Les systèmes de type (ou relations de compatibilité) utilisés dans ces travaux traitent des contraintes architecturales différentes de celles traités par notre système de type. Les contrats comportementaux de [15] ou le système Wright devraient être étendus pour gérer les types d'enregistrement qui caractérisent les messages Dream. Le système de type d'ArchJava est conçu pour assurer l'intégrité des communications, *i.e.* pour éviter l'aliasing qui peut entraîner la perte d'intégrité d'un composant. Le travail le plus proche du notre est sans doute le travail récent sur le système de type pour le système Ptolemy II [44], qui combine un ensemble riche de types de donnés tels des tableaux ou des enregistrements non mutables, et un système de type comportemental qui étend le travail sur les automates d'interface [18, 19] pour capturer des aspects temporels d'interfaces de composants. Toutefois, le système de type de Ptolemy II n'est pas directement applicable aux contraintes de Dream parce qu'il se limite à des types d'enregistrements non mutables.

Une limitation évidente de notre approche est que nous n'énonçons aucun résultat formel concernant les garanties fournies par le système de type. Le résultat à prouver serait l'absence d'erreur d'exécution liée à la lecture d'un champ absent, l'ajout d'un champ déjà présent ou encore l'utilisation d'un champ avec un type erroné. Nous avons choisi l'approche plus pragmatique d'implanter tout d'abord un prototype et de tester l'expressivité du système de types. Nous envisageons pour la suite de formaliser le comportement des composant Dream afin de pouvoir énoncer des garanties formelles.

Par ailleurs, le système de type n'est pas suffisamment précis pour spécifier des composants dont le comportement est fonction de la structure des messages, comme nous l'avons décrit en 5.3.6. Une possibilité éventuelle pour résoudre ce problème serait d'adapter des travaux existants sur les types intersections décrits par exemple dans [53].

5.4 Perspectives

Dans cette section, nous montrons tout d'abord comment le Kell calcul permet de capturer des éléments du modèle Fractal, puis nous discutons de futures directions de recherche.

5.4.1 Fractal en Kell

Nous montrons comment un composant Fractal peut se modéliser de manière assez directe dans le Kell calcul (par exemple dans le πK-calcul), en nous inspirant de [60, 38].

Un composant primitif, de nom a, se représente simplement par un kell

$$a[\texttt{Comp}]$$

où `Comp` est le code du composant. Une interface d'un composant est, conformément au modèle Fractal, un point d'accés vers l'extérieur. Une interface serveur correspond à un récepteur écoutant vers le haut. Une interface client à un message (orienté vers le haut si l'on utilise des messages orientés). Par exemple, le composant

$$a[\nu v.v\langle 0\rangle \mid (\texttt{get}^{\uparrow}\langle r\rangle \diamond (v\langle y\rangle \triangleright (r\langle y\rangle \mid v\langle y\rangle)) \mid (\texttt{set}^{\uparrow}\langle x\rangle \diamond (v\langle y\rangle \triangleright v\langle x\rangle))]$$

permet l'écriture ou la lecture d'une valeur, et présente deux interfaces `get` et `set`.

Un composant composite nommé a prend la forme

$$a[\texttt{Memb} \mid \texttt{Cont}]$$

Le processus `Memb` correspond à la membrane du composant et le processus `Cont` à son contenu. Le contenu du composant est un ensemble de sous-composants et est donc de la forme

$$\texttt{Cont} = c_1[Q_1] \mid \ldots \mid c_n[Q_n]$$

Nous avons vu au chapitre 2 et dans les exemples de membranes programmables donnés en 2.6 que le processus `Memb` décrit les interactions possibles entre les différents sous-composants, et entre les sous-composants et l'extérieur, il permet également de contrôler les sous-composant. Ce comportement correspond fidèlement au rôle de la membrane d'un composant de Fractal. Nous donnons dans la suite quelques exemples de contrôleurs.

Un contrôleur de contenu permet l'ajout ou la suppression de sous-composants et peut-être implanté à l'aide des deux récepteurs suivants :

$$\texttt{Add} = \texttt{add}^{\uparrow}\langle c,x\rangle \diamond c[x]$$
$$\texttt{Remove} = \texttt{remove}^{\uparrow}\langle c\rangle \diamond (c[x] \triangleright \mathbf{0})$$

L'interface `add` permet l'ajout d'un composant donné par son nom et son contenu. L'interface `remove` permet la suppression d'un composant donné par son nom.

Un contrôleur de cycle de vie permettant de stopper et de réactiver un composant peut s'implanter de la manière suivante :

$$\texttt{Resume} = \texttt{resume}^{\uparrow}\langle c\rangle \diamond (c\langle x\rangle \triangleright c[x])$$
$$\texttt{Suspend} = \texttt{suspend}^{\uparrow}\langle c\rangle \diamond (c[x] \triangleright c\langle x\rangle)$$

L'interface `suspend` suspend l'activité d'un composant et conserve son état d'un message. L'interface `resume` permet de réactiver un composant dont l'état a été sauvegardé précédemment.

On peut facilement créer des liaisons entre des composants. Par exemple un contrôleur de liaison peut avoir une interface de création de liaison définie par :

$$\texttt{Bind} = \texttt{bind}^{\uparrow}\langle c,d,i,j\rangle \diamond (i^{\downarrow}\langle x\rangle \diamond j\langle x\rangle)$$

Cette interface permet de lier une interface client i à une interface serveur j. Cette approche souffre de deux limitations. Tout d'abord, il n'est pas possible de supprimer une liaison et le nom d'une interface seul ne suffit pas a priori pour la désigner de manière non ambiguë. Un traitement plus satifaisant des contrôleurs de liaison est donné dans [38].

5.4.2 Vers un langage d'architecture dynamique

Nous venons de montrer que le Kell calcul permettait de représenter les caractéristiques principales du modèle de composants Fractal. En prolongeant ce travail, on pourrait envisager de spécifier complètement Fractal à l'aide du Kell calcul, ou d'une version autorisant le partage de Kell comme [38]. Une application intéressante du Kell calcul consiste à l'utiliser comme base d'un *langage de description d'architecture dynamique*. Les langages d'architecture utilisés habituellement proposent une description statique de l'architecture d'une application, correspondant par exemple à son état initial. Dans des systèmes où les reconfigurations dynamiques sont courantes, il est souhaitable de disposer d'un formalisme permettant de spécifier les reconfigurations possibles. Nous pensons que le Kell calcul est un candidat pour une telle application. Un avantage important serait de pouvoir hériter des différents résultats sur le calcul, comme les résultats sur le typage et l'implantation des chapitres précédents. Nous espérons notamment utiliser notre prototype d'implantation comme moteur de déploiement d'application Fractal.

Le système de type présenté en 5.3 permet de vérifier qu'un ensemble de composants Fractal, spécialisés dans le traitement de messages, sont assemblés de manière cohérente, par rapport à des contraintes sur la forme des messages traités par les composants. Il ne permet pas de prendre en compte d'éventuelles reconfigurations. Pour un nombre faible de reconfigurations, on peut simplement considérer à la main toutes les configurations possibles. Une manière plus satisfaite de traiter le problème serait d'intégrer justement ce système de type à un langage de description d'architecture dynamique.

Chapitre 6

Conclusion

A travers cette thèse, nous avons contribué à la définition du Kell calcul, un langage minimal destiné à la programmation des systèmes répartis à grande échelle et la programmation par composants. Nous avons développés des techniques de typages pour assurer certaines propriétés et avons proposé une implantation basée sur une spécification formelle de machine abstraite.

Au chapitre 2, nous avons rappelé les critères de conception du Kell calcul avant de le définir formellement. Le Kell calcul est une famille de langage d'ordre supérieur, avec localités, et un opérateur de passivation. De plus, les actions du Kell calcul vérifient un principe d'action locale qui permet notamment une implantation répartie efficace. Nous avons présenté deux instances particulières, le πK-calcul et le jK-calcul, utilisant deux types de récepteurs différents.

Dans le deuxième chapitre, nous avons étudié le typage du Kell calcul à travers deux systèmes de type. Le premier permet d'une part d'éviter certaines erreurs de programmation simples, et d'autre part de distinguer certaines localités non passivables. Une caractéristique originale de ce système de type est qu'il est défini de manière générique, indépendamment d'un langage de motifs particulier. Le deuxième système de type permet d'assurer l'unicité des noms de localité, et est basé sur un typage fin des processus.

Dans le troisième chapitre, nous avons proposé une implantation du πK-calcul, basée sur une machine abstraite que nous avons spécifié formellement. Nous avons défini deux niveaux de raffinements de spécification. La première a été prouvée équivalente au calcul, à l'aide d'une bisimilarité barbée forte. Pour la deuxième, nous avons conjecturé un résultat de correction. Une originalité de notre machine abstraite par rapport à des travaux similaires est que nous gardons une vue abstraite du réseau qui est modélisé lui-même comme un terme du Kell calcul. Notre implantation consiste en un interprète pour un langage simple basé sur le Kell calcul. Plusieurs interprètes peuvent communiquer à l'aide d'un modèle de communication réparti simple modélisé dans le calcul.

Dans le dernier chapitre, nous avons présenté le modèle de programmation par composants Fractal. Ce modèle de programmation a été notamment utilisé pour la programmation d'intergiciels à messages. Dans ce cadre, nous avons défini un système de type qui permet de vérifier un ensemble de contraintes sur une architecture logicielle. Ce système de type permet de garantir la cohérence d'un assemblage de composants spécialisés dans le traitement des messages.

Nous discutons maintenant différentes perspectives de recherches ouvertes par cette

thèse. Tout d'abord, le travail sur le Kell calcul peut être poursuivi de plusieurs manières. Le modèle peut-être enrichi de manière à permettre le partage de localités. Ce travail a été initié dans [37] mais ne prend pas en compte de manière satisfaite le contrôle des localités partagés. Par ailleurs, afin de modéliser des comportements de défaillance plus précis il est nécessaire de disposer d'aspects temporels dans le calcul pour représenter des détecteurs de panne par exemple. Un travail dans cette direction a été effectué en [5] pour le π-calcul. Finalement, il n'existe pas de caractérisation co-inductive d'une équivalence contextuelle faible pour le calcul. En particulier, la caractérisation donnée dans [59] se limite à une forme forte d'équivalence. A notre connaissance, ce résultat n'a jamais été prouvé pour pour les calculs proches du Kell calcul comme [35, 16].

Concernant le typage, notre système de type générique peut être amélioré en permettant un contrôle plus fin de la passivation. Par exemple, on aimerait pouvoir spécifier des types de passivation plus restrictifs de la forme $u[x] \triangleright \mathbf{0}$ qui consistent simplement à détruire une localité, ou de la forme $u[x] \triangleright v[u[x]]$ qui n'implique pas de capturer l'état du processus contenu dans u.

Le système de type garantissant l'unicité des noms de kells actifs est complexe et sans doute difficilement utilisable en pratique. Néanmoins, on peut espérer que les techniques utilisées (en particulier, l'utilisation de noms dans les types de processus) puissent avoir des applications pour garantir d'autres propriétés. Tout d'abord, un résultat d'unicité des noms de sous-kells locaux serait intéressant et semble atteignable. Nous aimerions également disposer d'un système de type permettant d'interprêter les kells comme des composants et garantir des propriétés d'intégrité d'assemblages de composants (par exemple, on voudrait garantir que des dépendences hiérarchiques entre certains composants sont toujours respectées et cela en présence de reconfiguration).

Concernant l'implantation du calcul, une preuve de correction de la machine raffinée est désirable, d'autant plus que les règles de réduction sont relativement complexes. Les propriétés conjecturées sont sans doute atteignables par des techniques semblables à celles de la machine simple, bien que fastidieuses. Toutefois, la propriété de correction est relativement faible et malheureusement, on ne peut espérer une preuve de complétude puisque l'implantation de la composition est déterministe et élimine des comportements valides du calcul.

Notre prototype est encore trop limité pour pouvoir programmer des exemples intéressants et nous aimerions avoir dans notre langage la possiblité de communiquer avec du code OCaml natif en utilisant des kells comme structure d'accueil. Nous souhaitons également implanter le partage proposé dans [37]. Le partage semble en effet nécessaire pour utiliser le langage de manière effective.

Le système de type pour Fractal présente deux améliorations possibles. Tout d'abord, nous n'avons prouvé aucun résultat de correction. Pour cela, nous envisageons de définir un langage pour spécifier le comportement des composants et rendre possible une propriété de correction du système de type. Ensuite, les contraintes que l'on peut exprimer sur les composants ne permettent pas de caractériser tous les composants couramment utilisés.

En plus des extensions que nous venons de considérer, nous pensons qu'un travail particulièrement prometteur consiste à appliquer le Kell calcul au modèle Fractal. Dans Fractal, les applications sont décrites comme des assemblages de composants à l'aide d'un langage de description d'architecture (ADL). Ce langage ne prend pas en compte les reconfigurations du système au cours du temps. Le Kell calcul permet d'exprimer naturelle-

ment les opérations de reconfiguration de Fractal et constitue donc un bon candidat pour un ADL dynamique. Les techniques développées dans cette thèse deviennent directement applicables. Nous aimerions par exemple utiliser notre prototype comme moteur de déploiement d'applications Fractal (le déploiement d'application, consistant à déplacer des parties de logiciels dans différents emplacements, est un cas particulier de reconfiguration).

Annexe A

Preuves du chapitre 3

A.1 Types génériques

Lemme A.1.1 (Affaiblissement) *Si* $\Gamma \vdash P : \texttt{proc}$*, alors* $\Gamma,u : \tau \vdash P : \texttt{proc}$*.*

Preuve 3 *Immédiat par induction sur la taille de l'arbre de dérivation. Le cas* T.Trig *utilise l'axiome 3.1 de la définition 3.2.1*

Lemme A.1.2 (Renforcement) *Si* $\Gamma,u : \tau \vdash P : \texttt{proc}$ *et* $u \notin \texttt{fn}(P) \cup \texttt{fv}(P)$*, alors* $\Gamma \vdash P : \texttt{proc}$*.*

Preuve 4 *Immédiat par induction sur la taille de l'arbre de dérivation. Le cas* T.Trig *utilise l'axiome 3.2 et le fait que si* $x \notin \texttt{fn}(\xi \triangleright P) \cup \texttt{fv}(\xi \triangleright P)$ *alors* $x \notin \texttt{fn}(\xi) \cup \texttt{fv}(\xi)$*.*

Lemme A.1.3 (Substitution) *Si* $\Gamma,x : \tau \vdash P : \texttt{proc}$ *et* $\Gamma \vdash V : \tau$*, alors* $\Gamma \vdash P\{V/x\} : \texttt{proc}$*.*

Preuve 5 *Par induction sur la dérivation de typage. Détaillons le cas* T.Trig*. On a*

$$(1)\,\texttt{type}((\Gamma,x : \tau),\xi,\Gamma')$$
$$(2) y \in \texttt{fv}(\xi) \implies (y,\sigma) \in \Gamma,x : \tau$$
$$(3)\Gamma,x : \tau,\Gamma' \vdash P : \texttt{proc}$$

L'hypothèse d'induction appliquée à (1) *nous donne* $\Gamma,\Gamma' \vdash P\{V/x\} : \texttt{proc}$*. On distingue deux cas.*

- $y \notin \texttt{fv}(\xi)$*. On a d'après 3.2,* $\texttt{type}(\Gamma,\xi,\Gamma')$*. On déduit de* (2)*,*

  $$y \in \texttt{fv}(\xi) \implies (y,\sigma) \in \Gamma$$

  On conclut alors par T.Trig

  $$\Gamma \vdash (\xi \triangleright P)\{V/x\} : \texttt{proc}$$

- $y \in \texttt{fv}(\xi)$*. On a d'après* (2)*,* $(y,\sigma) \in \Gamma$*, et par conséquent* $\tau = \sigma$*. D'après le lemme* **??***, on a* $V = u$ *et donc* $\Gamma \vdash u : \sigma$*. On a alors par l'axiome 3.3,* $\texttt{type}(\Gamma,\xi\{u/x\},\Gamma')$*. On applique alors la règle* T.Trig*.*

  $$\Gamma,\Gamma' \vdash \xi\{V/x\} \triangleright P\{V/x\} : \texttt{proc}$$

  On conclut en remarquant que $\xi\{V/x\} \triangleright P\{V/x\} = (\xi \triangleright P)\{V/x\}$*.*

Lemme A.1.4 *On suppose $m = \mathbf{C}_m\{V\}$. Si l'une des propositions suivantes est vraie :*

$$\Delta(m) \wedge \Gamma \vdash m : \pi \tag{A.1}$$
$$\Upsilon(m) \wedge \Gamma \vdash m : \pi \tag{A.2}$$
$$\Psi(U,m) \wedge \Gamma \vdash U : \pi \tag{A.3}$$
$$\Gamma(U,m) \wedge \Gamma \vdash U : \pi \tag{A.4}$$

Alors on a $\Gamma \vdash V : \tau$ avec $\tau = \Gamma(\mathtt{cn}\{\mathbf{C}_m\})$.

Preuve 6 *Tout d'abord, nous définissons une classe simplifiée de contexte dans lesquels les annotations de directions sont effacées.*

$$\begin{aligned}\mathbf{C} ::= {} & u\langle V_1,\ldots,\mathbf{C},\ldots,V_n\rangle \\ \mid {} & u[\mathbf{C}] \\ \mid {} & \mathbf{C} \mid \mathbf{C} \\ \mid {} & u[\,\cdot\,] \\ \mid {} & u\langle V_1,\ldots,\cdot\,,\ldots,V_n\rangle\end{aligned}$$

Soit $\mathbf{C}$ le contexte obtenu en effaçant les annotations de direction de $\mathbf{C}_m$. Puisque nos types ne prennent pas en compte d'information de direction, on a $\Gamma \vdash \mathbf{C}\{V\} : \pi$. On montre alors le résultat par induction sur la forme de $\mathbf{C}$.

Preuve 7 (Preuve du théorème 1) *Par induction sur la dérivation de la relation de réduction et par cas sur la dernière dérivation utilisée. Nous ne détaillons que le cas* R.RED.L*, le cas* R.RED.G *est similaire, et les autres cas ne posent pas de problème.*

$$(\xi \triangleright P) \mid U_1 \mid U_2 \mid U_3 \rightarrow P\theta$$

$$\Gamma \vdash (\xi \triangleright P) \mid U_1 \mid U_2 \mid U_3 : \pi$$

On a par les prémisses de la règle R.RED.L

$$\mathtt{match}(\xi,U_1 \mid U_2 \mid M_b,\theta)$$

On a aussi $\Delta(U_1)$, $\Upsilon(U_2)$, $\Psi(U_3,M_b)$. De plus, par les prémisses du jugement de typage, on sait que $\mathtt{type}(\Gamma,\xi,\Gamma')$ et $\Gamma,\Gamma' \vdash P : \pi$.

On veut prouver que $\Gamma \vdash P\theta : \pi$. Par le lemme A.1.3, il suffit de montrer que

$$\forall x \in \mathtt{dom}(\theta).\Gamma \vdash \theta(x) : \Gamma'(x)$$

On sait par l'axiome 3.5 que $\mathtt{dom}(\theta) = \mathtt{dom}(\Gamma')$. Soit $x \in \mathtt{dom}(\Gamma')$, d'après la définition **??***, il existe un contexte $\mathbf{C}_m$ tel que $U_1 \mid U_2 \mid M_k = \mathbf{C}_m\{\theta(x)\}$. On déduit alors par le lemme A.1.4 :*

$$\Gamma(\mathtt{cn}(\mathbf{C}_m)) = \tau \wedge \Gamma \vdash \theta(x) : \tau$$

D'après l'axiome 3.5, on a $\Gamma'(x) = \Gamma(\mathtt{cn}_m(\mathbf{C}_m))$ et l'on déduit le résultat.

Preuve 8 (Preuve de la proposition 3.2.2) *Il suffit de vérifier que le prédicat* `type` *vérifie les axiomes de la définition 3.2.1. Seul le cas 3.5 est non-immédiat. On suppose :*

$$\begin{aligned} \texttt{type}(\Gamma,\xi,\Gamma') \\ \texttt{match}(\xi,M,\theta) \\ \forall x \in \texttt{dom}(\theta).M = \mathbf{C}_m\{\theta(x)\} \wedge \Gamma(\texttt{cn}(\mathbf{C}_m)) = \tau \end{aligned}$$

On veut montrer que $\texttt{dom}(\theta) = \texttt{dom}(\Gamma')$ *et* $\Gamma'(x) = \tau$. *Supposons que* $\xi = u\langle x_1,\ldots,x_n\rangle$. *D'après les définitions des prédicats* `type` *et* `match` *dans le* πK*-calcul :*

$$\begin{aligned} (u,\texttt{chan}\langle\tau_1,\ldots,\tau_n\rangle) \in \Gamma \\ M = u^d\langle V_1,\ldots,V_n\rangle \\ \Gamma' = x_1 : \tau_1,\ldots,x_n : \tau_n \\ \theta = \{V_1/x_1,\ldots,V_n/x_n\} \end{aligned}$$

On a clairement $\texttt{dom}(\Gamma') = \texttt{dom}(\theta)$. *Soit* $x \in \texttt{dom}(\theta)$. *Pour fixer les idées, on suppose que* $x = x_1$. *On a alors* $\mathbf{C}_m = u\langle\cdot,V_2,\ldots,V_n\rangle$. *Par définition, on a* $\Gamma(\texttt{cn}(\mathbf{C}_m) = u_n^1$ *et donc* $\tau = \tau_1$. *On en conclut que* $\Gamma'(x) = \tau$.

Le cas ou $\xi = u[x]$ *se traite de la même manière.*

A.2 Unicité des noms de kells actifs

A.2.1 Lemmes sur les jugements de bonne formation

Lemme A.2.1

1. *(Instanciation) Si* $\Gamma \vdash \theta : \texttt{inst}$ *et* $\Gamma \vdash J$, *alors* $\Gamma\theta \vdash J$.
2. *(Affaiblissement) Si* $\Gamma,u : s \vdash \texttt{Env}$, $u \notin \texttt{dom}(\Gamma')$ *et* $\Gamma,\Gamma' \vdash J$, *alors* $\Gamma,u : s,\Gamma' \vdash J$.
3. *(Renforcement) Si* $\Gamma,x : s,\Gamma' \vdash J$, *alors* $\Gamma,\Gamma' \vdash J$. *Si* $\Gamma,a : s,\Gamma' \vdash J$, *alors* $\Gamma,\Gamma'\{\emptyset/a\} \vdash J\{\emptyset/a\}$.
4. *(Échange) Si* $\Gamma,u : s,v : s',\Gamma' \vdash J$ *et* $u \notin \texttt{fn}(s')$, *alors* $\Gamma,v : s',u : s,\Gamma' \vdash J$.

Preuve 9 *Par induction sur les dérivations de typage.*

A.2.2 Lemmes structuraux sur le jugement de typage

Lemme A.2.2 (Échange) *Si* $\Gamma,u : s,v : s',\Gamma' \vdash P : \tau$ *et* $u \notin \texttt{fn}(s')$, *alors* $\Gamma,v : s',u : s,\Gamma' \vdash P : \tau$.

Preuve 10 *On raisonne par induction sur la dérivation de* $\Gamma,u : s,v : s',\Gamma' \vdash P : \tau$. *Les cas* T.Sub, T.Nil *et* T.Id *utilisent le lemme A.2.1 (4). Les autres cas sont immédiats par induction.*

Lemme A.2.3 1. *Si* $\Gamma \vdash V : \sigma$, *alors* $V \in \text{Vars} \cup \text{Names}$.
2. *Si* $\Gamma \vdash V : \langle\widetilde{\tau}\rangle_\Delta^+$, *alors* $(V,\langle\widetilde{\tau}\rangle_\Delta^+) \in \Gamma$.

Lemme A.2.4 (Inversion de la relation de typage)

1. *Si* $\Gamma \vdash P_1 \mid P_2 : \tau$ *alors* $\tau = \Delta_1,\Delta_2$, $\Gamma \vdash P_1 : \Delta_1$ *et* $\Gamma \vdash P_2 : \Delta_2$.

2. *Si* $\Gamma \vdash \nu a : s.P : \tau$ *avec* $s = \forall\widetilde{\beta}.\gamma$, *alors* $\tau = \Delta$, $\Gamma,a : s \vdash P : \Delta$ *et* $\mathtt{fv}(s) = \emptyset$.
3. *Si* $\Gamma \vdash \nu a : s.P : \tau$ *avec* $s = \forall\widetilde{\beta}.\lambda$, *alors* $\tau = \Delta$, $\Gamma,a : s \vdash P : \Delta \uplus \{a\}$ *et* $\mathtt{fv}(s) = \emptyset$, $\mathtt{Ktype}(\lambda,a)$.
4. *Si* $\Gamma \vdash u\langle V\rangle : \tau$, *alors on a* $(a,\forall\widetilde{\beta}.\langle\tau_a\rangle_{\Delta_a}) \in \Gamma$ *et* $\Gamma \vdash V : \tau'$. *De plus il existe une instanciation* θ *telle que* $\Gamma \vdash \theta : \mathtt{inst}$, $\Delta_a\theta \leq \tau$ *et* $\tau' \leq \tau_a\theta$.
5. *Si* $\Gamma \vdash u[P] : \tau$, *alors on a* $(a,\forall\widetilde{\beta}.\mathtt{kell}(w)_{\Delta_a\to\Delta'_a}) \in \Gamma$ *et* $\Gamma \vdash P : \Delta$. *De plus il existe une instanciation* θ *telle que* $\Gamma \vdash \theta : \mathtt{inst}$, $(w,\Delta_a\theta) \sqcup \Delta'_a\theta \leq \tau$ *et* $\Delta \leq \Delta_a\theta$.

Lemme A.2.5 (Affaiblissement) *Si* $\Gamma,u : s \vdash \mathtt{Env}$, $u \notin \mathtt{dom}(\Gamma')$ *et* $\Gamma,\Gamma' \vdash P : \Delta$, *alors* $\Gamma,u : s,\Gamma' \vdash P : \Delta$.

Preuve 11 *On raisonne par induction sur la taille de la dérivation de* $\Gamma,\Gamma' \vdash P : \Delta$. *On ne détaille que les cas non immédiats.*

T.Nil *On applique le lemme A.2.1 (2) au jugement* $\Gamma,\Gamma' \vdash \mathtt{Env}$ *et on conclut par application de la règle* T.Nil.

T.Id *On applique le lemme A.2.1 (2) au jugement* $\Gamma,\Gamma' \vdash \theta : \mathtt{inst}$ *et on conclut par* T.Id.

T.Res.C *On peut supposer* $a \neq u$. *On applique l'hypothèse d'induction à* $\Gamma,\Gamma',a : s \vdash P : \Delta$ *en remarquant que* $u \notin \mathtt{dom}(\Gamma',a : s)$. *On conclut alors en applicant la règle* T.Res.C.

T.Res.K *Similaire au cas* T.Res.C.

T.Trig.Msg *On peut toujours renommer* $\widetilde{x}$ *pour avoir* $u \notin \widetilde{x}$, *et donc* $u \notin \mathtt{dom}(\Gamma',\widetilde{x} : \widetilde{\tau})$. *On peut donc appliquer l'hypothèse d'induction à* $\Gamma,\Gamma',\widetilde{x} : \widetilde{\tau} \vdash P : \Delta$. *On peut également renommer* $\widetilde{\beta}$ *pour avoir* $\mathtt{ftv}(s) \cap \mathtt{ftv}(\Gamma,u : s,\Gamma',\widetilde{x} : \widetilde{\tau}) = \emptyset$. *On conclut alors par* T.Trig.Msg.

T.Trig.Pass *Similaire au cas* T.Trig.Msg.

Lemme A.2.6 (Renforcement)

1. *(Variable) Si* $\Gamma,x : s,\Gamma' \vdash P : \tau$ *et* $x \notin \mathtt{fv}(P)$, *alors* $\Gamma,\Gamma' \vdash P : \tau$.
2. *(Nom) Si* $\Gamma,a : s,\Gamma' \vdash P : \tau$ *et* $a \notin \mathtt{fn}(P)$, *alors* $\Gamma,\Gamma'\{\emptyset/a\} \vdash P : \tau\{\emptyset/a\}$. *En particulier, si* s *est un type de canal (*i.e. $s = \forall\widetilde{\beta}.\gamma$*), on a* $\Gamma,\Gamma' \vdash P : \tau$.

Preuve 12 *Les deux parties se font par induction sur la dérivation de typage. Dans la première partie, les cas* T.Nil *et* T.Id *sont conséquence du lemme A.2.1(3). Les autres cas sont immédiats par induction.*

Dans la deuxième partie, les cas T.Nil *et* T.Id *se déduisent du lemme A.2.1(3). Dans le cas* T.Sub, *on suppose que le jugement hypothèse s'écrit* $\Gamma,a : s,\Gamma' \vdash P : \tau'$. *On applique l'hypothèse d'induction à la dérivation* $\Gamma,a : s,\Gamma' \vdash P : \tau$, *et le lemme A.2.1(3) au jugement* $\Gamma,a : s,\Gamma' \vdash \tau \leq \tau'$. *On conclut en appliquant la règle* T.Sub. *On traite le cas* T.Par *en appliquant l'hypothèse d'induction aux prémisses et en remarquant que* $\Delta_1\{\emptyset/a\},\Delta_2\{\emptyset/a\} = (\Delta_1,\Delta_2)\{\emptyset/a\}$. *Les autres cas sont immédiats par induction.*

Lemme A.2.7 (Instanciation) *Si* $\Gamma \vdash P : \tau$ *et* $\Gamma \vdash \theta : \mathtt{inst}$ *alors* $\Gamma\theta \vdash P : \tau\theta$.

Preuve 13 *Nous raisonnons par induction sur la dérivation de* $\Gamma \vdash P : \tau$. *Nous ne détaillons que les cas non immédiats.*

T.Nil *On applique le lemme A.2.1(1) au jugement* $\Gamma \vdash \mathtt{Env}$ *et on conclut par la règle* T.Nil.

T.Sub *On suppose que le jugement hypothèse s'écrit* $\Gamma \vdash P : \tau'$. *On applique l'hypothèse d'induction à la dérivation* $\Gamma \vdash P : \tau$, *et le lemme A.2.1(1) au jugement* $\Gamma \vdash \tau \leq \tau'$. *On conclut en appliquant la règle* T.Sub.

T.Res.K *On a* $a \notin \mathtt{dom}(\Gamma)$ *et* $a \notin \Delta$. *Par conséquent,* $a \notin \Delta\theta$. *Par induction, on obtient* $\Gamma\theta,a : s\theta \vdash P : \Delta\theta \uplus \{a\}$. *De plus* $\mathtt{ftv}(s) = \emptyset$, *par conséquent* $s\theta = s$. *On conclut alors par la règle* T.Res.K, *les autres prémisses étant trivialement vérifiés.*

T.Trig.Msg *On peut toujours renommer* $\widetilde{\beta}$ *pour avoir* $\mathtt{ftv}(\Gamma\theta) \cap \widetilde{\beta} = \emptyset$. *Le résultat est alors immédiat par induction.*

T.Trig.Pass *Similaire au cas* T.Trig.Msg.

Lemme A.2.8 (Substitution) *Si* $\Gamma, x : \tau, \Gamma' \vdash P : \tau'$ *et* $\Gamma \vdash V : \tau$ *alors* $\Gamma,\Gamma' \vdash P\{V/x\} : \tau'$.

Preuve 14 *On raisonne par induction sur la dérivation de* P. *Nous ne détaillons que les cas non triviaux.*

T.Sub *On applique l'hypothèse d'induction à la première prémisse, et le lemme A.2.6(1) à la deuxième. On conclut en appliquant la règle* T.Sub.

T.Nil *Conséquence du lemme A.2.6(1).*

T.Id *Si* $x \neq u$, *on conclut par le lemme A.2.6(1). Si* $x = u$, *le résultat est donné par l'hypothèse* $\Gamma \vdash V : \tau$ *et le lemme A.2.5.*

T.Res.C *On choisit* a *de telle sorte que* $a \notin \mathtt{fn}(V)$. *Par induction, on a alors* Γ, Γ', $a : s \vdash P\{V/x\} : \tau'$. *On conclut alors facilement par induction en remarquant que* $\nu a : s.(P\{V/x\}) = (\nu a : s.P)\{V/x\}$.

T.Res.K *Ce cas se traite comme le cas* T.Res.C.

T.Msg *La jugement hypothèse peut s'écrire*

$$\Gamma,x : \tau',\Gamma' \vdash u\langle\widetilde{V'}\rangle : \Delta$$

On a alors $\Gamma,x : \tau',\Gamma' \vdash u : \langle\widetilde{\tau}\rangle^t_\Delta$ *et* $\Gamma,x : \tau',\Gamma' \vdash V'_i : \tau_i$. *Par induction, on a* $\Gamma,\Gamma' \vdash V'_i\{V/x\} : \tau_i$ *et* $\Gamma,x : \tau',\Gamma' \vdash u\{V/x\} : \langle\widetilde{\tau}\rangle^t_\Delta$. *On conclut en appliquant la règle* T.Msg *à ces deux jugements. Il convient néanmoins de remarquer que si* $x = u$, *on a nécessairement* $\tau' = \langle\widetilde{\tau}\rangle^t_\Delta$, *et d'après lemme A.2.3,* $V \in \textsc{Names} \cup \textsc{Vars}$.

T.Kell *Similaire au cas* T.Msg.

T.Trig.Msg *Le jugement hypothèse peut s'écrire* Γ, $y : \tau'$, $\Gamma' \vdash u\langle\widetilde{x}\rangle \triangleright Q : \emptyset$. *On a alors*

$$\Gamma,\ y : \tau',\ \Gamma',\ \widetilde{x} : \widetilde{\tau} \vdash Q : \Delta$$

$$(u,\langle\widetilde{\tau}\rangle^t_\Delta) \in \Gamma,\ y : \tau',\ \Gamma'$$

De plus, on peut toujours supposer $\widetilde{x} \cap \mathtt{fv}(V) = \emptyset$. *Par induction, on a alors*

$$\Gamma,\ \Gamma',\ \widetilde{x} : \widetilde{\tau} \vdash Q\{V/y\} : \Delta$$

On distingue deux cas

- $u \neq y$. *On peut appliquer la règle* T.Trig.Msg. *On a alors* $\Gamma, \Gamma' \vdash a\langle\widetilde{x}\rangle \triangleright (Q\{V/y\}) : \emptyset$. *On conclut en remarquant que* $a\langle\widetilde{x}\rangle \triangleright (Q\{V/y\}) = (u\langle\widetilde{x}\rangle \triangleright Q)\{V/y\}$.
- $u = y$. *On a alors* $\tau' = \langle\widetilde{\tau}\rangle^+_\Delta$. *On déduit d'après le lemme A.2.3 que* $(V,\tau') \in \Gamma,y : \tau',\Gamma',\widetilde{x} : \widetilde{\tau'}$ *et on peut conclure comme avant par application de* T.Trig.Msg.

T.Trig.Pass *Similaire au cas* T.Trig.Msg.

A.2.3 Lemmes de préservation

Preuve 15 (Preuve de la proposition 3.3.2) *La preuve se fait par induction sur la taille de la dérivation de $P \equiv Q$, et par cas sur la forme de la dérivation $\Gamma \vdash P : \tau$ en utilisant le lemme A.2.4. On va prouver en fait simultanément les deux propositions suivantes.*

1. *Si $\Gamma \vdash P : \tau$ et $P \equiv Q$, alors $\Gamma \vdash Q : \tau$.*
2. *Si $\Gamma \vdash Q : \tau$ et $P \equiv Q$, alors $\Gamma \vdash P : \tau$.*

On introduit la deuxième proposition pour pouvoir utiliser l'hypothèse d'induction dans le cas correspondant à l'axiome de symétrie. Nous détaillons maintenant les seuls cas non immédiats.

S.Nu.Comm *On suppose que le jugement hypothèse s'écrit*

$$\nu a : s.\nu b : s'.P \equiv \nu b : s'.\nu a : s.P$$

avec $a \neq b$ et $\Gamma \vdash \nu a : s.\nu b : s'.P : \Delta$. On a également $b \notin \mathit{fn}(s)$ et $a \notin \mathit{fn}(s')$. On suppose de plus $s = \forall\widetilde{\beta}.\lambda$ et $s' = \forall\widetilde{\beta}.\lambda'$. On peut appliquer deux fois le lemme A.2.4(2). On a alors

$$\Gamma,\, a : s,\, b : s' \vdash P : \Delta \uplus \{a\} \uplus \{b\}$$

Avec $\mathit{ftv}(s) = \mathit{ftv}(s') = \emptyset$. Par ailleurs, l'hypothèse $a \notin \mathit{fn}(b)$ nous permet d'appliquer le lemme A.2.2. On obtient

$$\Gamma,\, b : s',\, a : s \vdash P : \Delta \uplus \{a\} \uplus \{b\}$$

On en déduit le résultat par deux applications successives de T.Res.C. *Les autres cas se traitent de manière identique.*

S.Nu.Par *Il y a quatre sous-cas à considérer selon le sens de lecture de la règle (extrusion ou intrusion) et le type de s (canal ou kell). Le raisonnement étant identique dans les deux cas, on ne traite que le cas où s est un type de kell, c'est à dire $s = \forall\widetilde{\beta}.\lambda$.*

extrusion, *a* **kell** *D'après le lemme A.2.4(1 et 3), on a $\Gamma \vdash (\nu a : s.P) \mid Q : \Delta,\Delta'$ avec $\Gamma,a : s \vdash P : \Delta \uplus \{a\}$ et $\Gamma \vdash Q : \Delta'$. On a également $\mathit{fv}(s) = \emptyset$ et $\mathit{Ktype}(\lambda,a)$. Par affaiblissement (lemme A.2.5), on a $\Gamma,a : s \vdash Q : \Delta'$. On déduit alors de* T.Par

$$\Gamma,a : s \vdash P \mid Q : (\Delta \uplus \{a\}),\Delta'$$

Par ailleurs, puisque $a \notin \mathit{dom}(\Gamma)$, on a $a \notin \Delta'$ et par conséquent $(\Delta \uplus \{a\}),\Delta' = (\Delta,\Delta') \uplus \{a\}$. On conclut alors par T.Res.K

$$\Gamma \vdash \nu a : s.P \mid Q : \Delta,\Delta'$$

intrusion, *a* **kell** *D'après le lemme A.2.4(3), on a $\Gamma,a : s \vdash P \mid Q : \Delta \uplus \{a\}$ avec $\mathit{fv}(s) = \emptyset$ et $\mathit{Ktype}(s,a)$. D'après le lemme A.2.4(1), on a $\Gamma,a : s \vdash P : \Delta_1$, $\Gamma,a : s \vdash Q : \Delta_2$ et $\Delta \uplus \{a\} = \Delta_1,\Delta_2$. On a alors deux cas possibles.*

- *$a \in \Delta_1$. On a alors $\Delta_1 = \Delta'_1 \uplus \{a\}$. Par* T.Res.K *on a $\Gamma \vdash \nu a : s.P : \Delta'_1$. Par ailleurs, comme par hypothèse $a \notin \mathit{fn}(Q)$ et $a \notin \Delta_2$, on a d'après A.2.6, $\Gamma \vdash Q : \Delta_2$. On obtient par* T.Par

$$\Gamma \vdash (\nu a : s.P) \mid Q : \Delta'_1,\Delta_2$$

D'où le résultat en remarquant que $\Delta'_1,\Delta_2 = \Delta$.

– $a \in \Delta_2$. *On peut écrire* $\Delta_2 = \Delta_2' \uplus \{a\}$. *On a alors par A.2.6,* $\Gamma \vdash Q : \Delta_2'$. *On a de plus* $\Gamma \vdash \Delta_1 \leq \Delta_1 \uplus \{a\}$. *Par* T.Sub *on déduit* $\Gamma,a : s \vdash P : \Delta_1 \uplus \{a\}$ *et par* T.Res.K, $\Gamma \vdash \nu a : s.P : \Delta_1$. *On obtient par* T.Par

$$\Gamma \vdash (\nu a : s.P) \mid Q : \Delta_1,\Delta_2'$$

D'où le résultat en remarquant que $\Delta_1,\Delta_2' = \Delta$.

Sc-Context *On procède par sous-induction sur la structure de* $\mathbf{E}$. *Le cas* $\mathbf{E} = \cdot$ *est immédiat. Les trois autres cas se traitent de manière similaire. On utilise le lemme d'inversion. On applique l'hypothèse d'induction et on conclut en appliquant les règles de typage.*

Preuve 16 (Preuve de la proposition 3.3.4) *Par induction sur la dérivation de* $P \rightsquigarrow P'$.

SR.Kell *On a par hypothèse*

$$a[\nu b : s.Q] \mapsto \nu b : s.a[Q]$$

$$\Gamma \vdash a[\nu b : s.Q] : \tau$$

Remarquons que le processus P *étant clos, on utilise un nom* a, *plutôt qu'un identifiant quelconque* u. *On se limite au cas ou* $s = \forall\widetilde{\beta}.\lambda$. *Le cas ou* s *est un type de canal ne présente pas de difficulté. On a par hypothèse sur l'environnement de typage* $(a,s') \in \Gamma$ *avec* $s' = \forall\widetilde{\beta}.\mathtt{kell}(a)_{\rho\to\Delta_a}$ *et* $\mathtt{fv}(s') = \emptyset$.
D'après le lemme A.2.4(5),

$$\Gamma \vdash \theta : \mathtt{inst} \tag{A.5}$$

$$(a,\rho\theta) \sqcup \Delta_a\theta \leq \tau \tag{A.6}$$

$$\Gamma,b : s \vdash Q : \Delta \uplus \{b\} \tag{A.7}$$

$$\Gamma \vdash \Delta \leq \rho\theta \tag{A.8}$$

$$\mathtt{fv}(s) = \emptyset \tag{A.9}$$

$$\mathtt{Ktype}(\lambda,b) \tag{A.10}$$

On définit alors l'instanciation θ' *de la manière suivante*

$$\theta' = \begin{cases} \rho \to \rho\theta \uplus \{b\} \\ \beta \to \beta\theta \end{cases}$$

Notons que l'on a bien $b \notin \rho\theta$ *car* θ *est bien formée dans l'environnement* Γ *qui ne contient pas* b.
Par T.Id, *on a*

$$\Gamma,b : s \vdash a : \mathtt{kell}(a)_{\rho\theta\uplus\{b\}\to\Delta_a\theta'}$$

Par ailleurs, on déduit de A.8

$$\Gamma,b : s \vdash \Delta \uplus \{b\} \leq \rho\theta \uplus \{b\}$$

$$\Gamma,b : s \vdash (a,\rho\theta \uplus \{b\}),\Delta_a\theta' \leq ((a,\rho\theta),\Delta_a\theta) \uplus \{b\}$$

On déduit alors de T.Sub *et* T.Kell

$$\Gamma \vdash \nu b : s.a[Q] : \tau$$

SR.Ctx *On raisonne par sous-induction sur la structure du contexte d'évaluation. Lorsque le contexte a pour forme* $\mathbf{E} = \nu a : \forall\widetilde{\beta}.\lambda.$, *on utilise la règle* T.Res.K *(plus précisément, le lemme d'inversion) et on peut appliquer l'hypothèse d'induction, l'environnement* Γ,$a : \forall\widetilde{\beta}.\lambda$ *reste bon en raison de l'hypothèse* $\texttt{Ktype}(a,\lambda)$ *dans la règle.*

SR.Str *Immédiat par induction en utilisant le lemme 3.3.2.*

Preuve 17 (Preuve du théorème 3) *On raisonne par induction sur la dérivation de* $P \to Q$. *Pour simplifier, on se place dans le cas monoadique et l'on considère seulement le* πK*-calcul.*

R.Loc *On suppose*

$$a\langle V\rangle \mid (a\langle x\rangle \triangleright P) \to P\{V/x\}$$

$$\Gamma \vdash a\langle V\rangle \mid (a\langle x\rangle \triangleright P) : \tau$$

D'après le lemme A.2.4(1), on a $\tau = \Delta_1,\Delta_2$ *avec* $\Gamma \vdash a\langle V\rangle : \Delta_1$ *et* $\Gamma \vdash a\langle x\rangle \triangleright P : \Delta_2$. *D'après le lemme A.2.4(4), on a* $(a,\forall\widetilde{\beta}.\langle\tau_a\rangle^t_{\Delta_a}) \in \Gamma$ *et* $\Gamma \vdash V : \tau'$, *ou* θ *est une instanciation bien formée dans* Γ *telle que* $\Delta_a\theta \leq \Delta_1$ *et* $\tau' \leq \tau_a\theta$. *Par ailleurs, on a* Γ,$x : \tau_a \vdash P : \Delta_a$. *Par le lemme A.2.7, on a* $\Gamma\theta$,$x : \tau_a\theta \vdash P : \Delta_a\theta$. *De plus* $\Gamma\theta = \Gamma$. *Par la règle* T.Sub, *on a* $\Gamma \vdash V : \tau_a\theta$. *On applique alors le lemme A.2.8 et on obtient* $\Gamma \vdash P\{V/x\} : \Delta_a\theta$. *Par* T.Sub, *puis* T.Par, *on conclut finalement* $\Gamma \vdash P\{V/x\} : \Delta_1,\Delta_2$.

R.Pass *Ce cas est similaire au précédent. On a*

$$a[P] \mid (a[x] \triangleright Q) \to Q\{P/x\}$$

$$\Gamma \vdash a[P] \mid (a[x] \triangleright Q) : \tau$$

D'après le lemme A.2.4(1), on a $\tau = \Delta_1,\Delta_2$. $\Gamma \vdash a[P] : \Delta_1$ *et* $\Gamma \vdash a[x] \triangleright Q : \Delta_2$. *D'après le lemme A.2.4(5), on a* $(a,\forall\widetilde{\beta}.\texttt{kell}(w)_{\Delta_a\to\Delta'_a}) \in \Gamma$ *et* $\Gamma \vdash Q : \Delta$, *avec* θ *une instanciation telle que* $\Gamma \vdash \theta : \texttt{inst}$, $(w,\Delta_a\theta) \sqcup \Delta'_a\theta \leq \Delta_1$ *et* $\Delta \leq \Delta_a\theta$. *Par ailleurs, on a* Γ,$x : \Delta_a \vdash Q : \Delta'_a$. *Par le lemme A.2.7, on a* $\Gamma\theta$,$x : \Delta_a\theta \vdash P : \Delta'_a\theta$. *De plus* $\Gamma\theta = \Gamma$. *Par la règle* T.Sub, *on a* $\Gamma \vdash P : \Delta_a\theta$. *On applique alors le lemme A.2.8 et on obtient* $\Gamma \vdash P\{V/x\} : \Delta'_a\theta$. *On a par ailleurs* $\Delta'_a\theta \leq (w,\Delta_a\theta)\sqcup\Delta'_a\theta \leq \Delta_1 \leq \Delta_1,\Delta_2$ *Par* T.Sub, *puis* T.Par, *on conclut finalement* $\Gamma \vdash P\{Q/x\} : \tau$.

Les autres cas se traitent de manière standard.

A.2.4 Progrès

Preuve 18 (Preuve du théorème 2) *On montre par induction sur la dérivation de typage de* P *que si* $\Gamma \vdash P : \Delta$, *alors* P *n'est pas erroné et de plus,* $\texttt{kellname}(P) \subseteq \Delta$. T.Sub *et* T.Res.C *sont immédiats par induction. Les cas* T.Nil, T.Trig.Msg *et* T.Trig.Pass *sont triviaux également puisque* $\texttt{kellname}(P) = \mathit{emptyset}$. *Le cas* T.Id *ne se produit jamais puisque* $P = a$ *n'est pas syntaxiquement correct et* $P = x$ *n'est pas clos. Dans* T.Par, Δ_1,Δ_2 *est un ensemble par hypothèse. Par conséquent,* Δ_1 *et* Δ_2 *sont des ensembles disjoints. Par induction, on a* $\texttt{kellname}(P_1) \subseteq \Delta_1$ *et* $\texttt{kellname}(P_2) \subseteq \Delta_2$. *On alors* $\texttt{kellname}(P_1 \mid P_2) = \texttt{kellname}(P_1) \cup \texttt{kellname}(P_2) \subseteq \Delta_1,\Delta_2$. *Finalement, dans* T.Res.K, *on a par induction* $\texttt{kellname}(P) \subseteq \Delta$ *et on a bien* $\texttt{kellname}(\nu a : s.P) \subseteq \Delta - a$.

Annexe B

Preuves du chapitre 4

B.1 Résultats généraux

Dans la suite, nous généralisons la relation $\cong$ aux arbres. Si M et N sont deux arbres vérifiant les prédicats **tree**(M,l,p,m) et **tree**(N,l,p,m), on a $M \cong N$ si et seulement si $M\sigma = N\sigma'$ où σ et σ' sont des renommages injectifs des noms de localités et des noms résolus (si aucune confusion n'est possible, on les appelera simplement renommages). On définit de plus la fonction **names** qui renvoit l'ensemble des noms résolus et noms de localité d'une machine, et **resnames** qui renvoit les noms résolus d'une machine.

Preuve 19 (Preuve du théorème 4) *La preuve se fait par induction sur la structure d'arbre de M.*

Preuve 20 (Preuve du lemme 4.2.8) *Immédiat.*

Lemme B.1.1 *Si $M \xrightarrow{\equiv} M'$ alors $M \equiv L \mid M''$ avec $L \xrightarrow{\equiv} M'''$ et $M' \equiv M''' \mid M''$. De plus $L \xrightarrow{\equiv} M'''$ est une des règles d'axiome définissant $\xrightarrow{\equiv}$.*

Preuve 21 *Immédiat.*

Lemme B.1.2 *Si $M \mapsto N$ alors $M' \equiv M_0 \mid M''$, $M_0 \mapsto M_0'$ et $N \equiv M_0' \mid M''$. De plus $M_0 \mapsto M_0'$ est une des règles d'axiome définissant $\mapsto$.*

Preuve 22 *Immédiat.*

Lemme B.1.3 *Si $M \cong N$ et $M \xrightarrow{\equiv} M'$, $M \mapsto M'$ ou $M \to M'$ alors $N \xrightarrow{\equiv} N'$, $N \mapsto N'$, $N \to N'$ respectivement, avec $M' \cong N'$.*

Preuve 23 *Par définition de $\cong$, on a* **tree**(M,l,p,m) *et* **tree**(N,l,p,m). *De plus, d'après le théorème 4, on a dans tous les cas* **tree**(M',l,p,m). *D'après le lemme B.1.1, on a $M \equiv L_0 \mid M_0''$ avec $L_0 \xrightarrow{\equiv} M_0'''$. Nous considérons le cas où $L_0 \xrightarrow{\equiv} M_0'''$ correspond à la règle* M.S.New. *On a :*

$$\frac{i_0 \textbf{ fresh}}{l_0 : p_0[(\nu a.P) \mid Q]_{m_0,S_0} \xrightarrow{\equiv} l_0 : p_0[P\{i_0/a\} \mid Q]_{m_0,S_0}}$$

Par ailleurs, on a $M\sigma \equiv N\sigma'$ *avec* σ *et* σ' *deux renommages. On peut écrire* $N \equiv L_1 \mid M_1''$ *avec* $L_0\sigma \equiv L_1\sigma'$ *et* $M_0''\sigma \equiv M_1''\sigma'$ *et*

$$L_1 \equiv l_1 : p_1[(\nu a.P') \mid Q']_{m_1,S'}$$

Ainsi, on a par M.S.New *:*

$$\frac{i_1 \textbf{ fresh}}{l_1 : p_1[(\nu a.P') \mid Q']_{m_1,S_1} \xrightarrow{\equiv} l_1 : p_1[P'\{i_1/a\} \mid Q']_{m_1,S_1}}$$

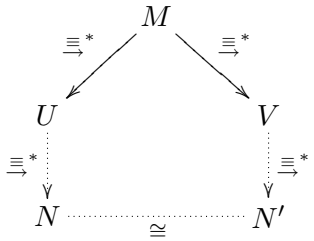
On a de plus $M_1''' = l_1 : p_1[P'\{i_1/a\} \mid Q']_{m_1,S_1}$ *et* $N \xrightarrow{\equiv} M_1''' \mid M_1'' = N'$*. On peut construire deux renommages* τ *et* τ' *tels que* $N'\tau \equiv M'\tau'$*. On en déduit* $M' \cong N'$*. Les cas correspondant à* M.S.Cell *et* M.S.Act *sont similaires.*

On procède de même pour $\mapsto$ *en utilisant le lemme B.1.2. Le résultat pour* $\rightarrow$ *découle de la définition de cette relation.*

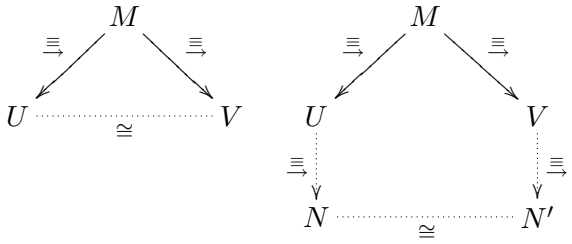
Lemme B.1.4

- *Si* $M \xrightarrow{\equiv}{}^* U$ *et* $M \xrightarrow{\equiv}{}^* V$ *alors* $U \xrightarrow{\equiv}{}^* N$ *et* $V \xrightarrow{\equiv}{}^* N'$ *avec* $N \cong N'$*.*
- *Si* $M \mapsto M'$ *et* $M \xrightarrow{\equiv}{}^* M''$ *alors, il existe un* N *tel que* $M'' \mapsto N$ *et* $M' \xrightarrow{\equiv}{}^* \cong N$*.*

Preuve 24 *Nous prouvons la première partie. Elle peut être illustrée par le diagramme suivant :*



Nous prouvons tout d'abord que si $M \xrightarrow{\equiv} U$ *et* $M \xrightarrow{\equiv} V$ *alors* $U \cong V$ *ou* $U \xrightarrow{\equiv} N$ *et* $V \xrightarrow{\equiv} N'$ *avec* $N \cong N'$ *:*



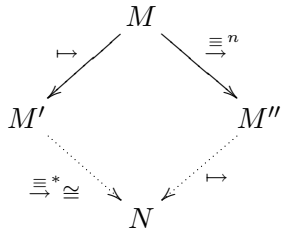
On raisonne par cas sur les règles de réduction qui conduisent à U *et* V*. Nous pouvons appliquer le lemme B.1.1 :* $M \equiv L_0 \mid M_0'$ *et* $L_0 \xrightarrow{\equiv} M_0''$ *où la dernière réduction correspond à l'un des axiomes définissant* $\xrightarrow{\equiv}$*. De la même façon,* $M \equiv L_1 \mid M_1'$ *et* $L_1 \xrightarrow{\equiv} M_1''$*. Si* $L_0 \not\equiv L_1$ *alors* $M \equiv L_0 \mid L_1 \mid M_2'$ *et le résultat est conséquence de la règle* M.S.Ctx *et d'un éventuel renommage des noms de localités ou des noms résolus. Si* $L_0 \equiv L_1$*, il y a neuf cas à considérer, en fonction des axiomes utilisés pour chaque réduction.*

Pour le cas général, considérons le prédicat suivant :

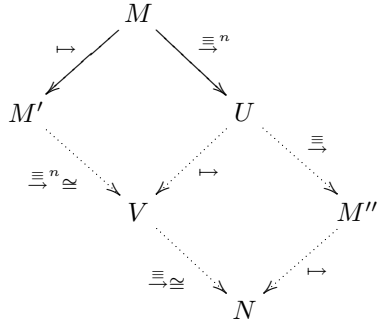
$$H(n,p) \triangleq (M \stackrel{\equiv}{\rightarrow}^{n} U \wedge M \stackrel{\equiv}{\rightarrow}^{p} V) \implies (U \stackrel{\equiv}{\rightarrow}^{n'} N \wedge V \stackrel{\equiv}{\rightarrow}^{p'} N' \text{ avec } N \cong N', n' \leq \mathbf{max}(n,p), p' \leq \mathbf{max}(n,p))$$

Le résultat découle de la proposition $\forall n,p \geq 0, H(n,p)$ *qui peut être prouvé par induction sur* $N = \mathbf{max}(n,p)$*, en utilisant le cas précédent est le fait que* $\cong$ *est clos par réduction pour la relation* $\stackrel{\equiv}{\rightarrow}$*.*

Prouvons maintenant la deuxième partie. Il suffit de prouver le diagramme suivant :



On le prouve par induction sur n*. Supposons le diagramme vrai pour* n *donné et supposons de plus* $M \mapsto M'$ *et* $M \stackrel{\equiv}{\rightarrow}^{n+1} M''$*. On a* $M \stackrel{\equiv}{\rightarrow}^{n} U \stackrel{\equiv}{\rightarrow} M''$ *pour une certaine machine* U*. Par induction, on a* $U \mapsto V$ *et* $M' \stackrel{\equiv}{\rightarrow}^{*} V' \cong V$ *pour deux machines* V *et* V'*. De plus, comme* $U \stackrel{\equiv}{\rightarrow} M''$ *on peut appliquer le cas* $n = 1$ *et on a* $V \stackrel{\equiv}{\rightarrow} \cong N$ *et* $M'' \mapsto N$ *pour une certaine machine* N*. A partir de* $M' \stackrel{\equiv}{\rightarrow}^{*} \cong V$*,* $V \stackrel{\equiv}{\rightarrow}^{*} \cong N$ *et du lemme B.1.3, nous déduisons* $M' \stackrel{\equiv}{\rightarrow}^{*} \cong N$*.*



Il reste maintenant à prouver le résultat pour $n = 1$*. D'après les lemmes B.1.1 et B.1.2, on a* $M \equiv M_0 \mid M_1$ *et* $M \equiv M'_0 \mid M'_1$ *avec* $M_0 \stackrel{\equiv}{\rightarrow} N_0$ *et* $M'_0 \mapsto N'_0$ *où seules les règles d'axiome sont utilisées dans ces deux réductions. Si* M_0 *et* M'_0 *correspondent à des localités différentes, alors le résultat est immédiat. Autrement, il y a plusieurs cas à considérer, en fonction des axiomes utilisés pour écrire les réductions. Tous les cas étant similaires, on ne considère que le cas correspondant aux axiomes* M.S.New *et* M.Out *où la localité réduite par* M.S.New *est celle contenant le message dans* M.Out*. On a*

$$M_0 \equiv l : p[(\nu c.P) \mid P']_{l',S} \stackrel{\equiv}{\rightarrow} l : p[P\{i/c\} \mid P']_{l',S}$$

$$M'_0 \equiv l : p[\xi\varphi \mid P'']_{l',S} \mid l' : q[(\xi \triangleright Q) \mid R]_{m,S'} \mapsto l : p[P'']_{l',S} \mid l' : q[Q\varphi \mid R]_{m,S'}$$

et

$$M_0 \equiv l : p[(\nu c.P) \mid P']_{l',S} \equiv l : p[\xi\varphi \mid P'']_{l',S}$$

On déduit que M_0 *et* M'_0 *peuvent s'écrire*

$$M_0 \equiv l : p[(\nu c.P) \mid \xi\varphi \mid R']_{l',S}$$

$$M'_0 \equiv l : p[\xi\varphi \mid (\nu c.P) \mid R']_{l',S} \mid l' : q[(\xi \triangleright Q) \mid R]_{m,S'}$$

De plus, on a

$$N_0 \equiv l : p[P\{i/c\} \mid \xi\varphi \mid R']_{l',S}$$

$$N'_0 \equiv l : p[(\nu c.P) \mid R']_{l',S} \mid l' : q[Q\varphi \mid R]_{m,S'}$$

D'après les règles M.Out, M.S.New *et* M.S.Cxt *on a*

$$l : p[P\{i/c\} \mid \xi\varphi \mid R']_{l',S} \mid l' : q[(\xi \triangleright Q) \mid R]_{m,S'}$$

$$\mapsto l : p[P\{i/c\} \mid R']_{l',S} \mid l' : q[Q\varphi \mid R]_{m,S'}$$

$$l : p[(\nu c.P) \mid R']_{l',S} \mid l' : q[Q\varphi \mid R]_{m,S'}$$

$$\stackrel{\equiv}{\rightarrow} l : p[P\{i'/c\} \mid R']_{l',S} \mid l' : q[Q\varphi \mid R]_{m,S'}$$

Finalement, si $N''_0 = l : p[P\{i/c\} \mid R']_{l',S} \mid l' : q[Q\varphi \mid R]_{m,S'}$ *et* $N = N''_0 \mid M'_1$*, on a* $N_0 \mid M_1 \mapsto N$ *et* $N'_0 \mid M'_1 \stackrel{\equiv}{\rightarrow} \cong N$*.*

Lemme B.1.5 *Une relation* $\mathcal{R}$ *est contextuelle si* $M\mathcal{R}N$ *implique* $\mathbf{E}\{M\}\mathcal{R}\mathbf{E}\{N\}$ *pour tous les contextes* $\mathbf{E}$*. Si* M *est une machine telle que* $\mathbf{tree}(M,l,p,m)$ *et* $\mathbf{E}$ *est un contexte, on a* $\mathbf{tree}(\mathbf{E}\{M\},l,p,m)$*. De plus, les relations* $\equiv$*,* $\cong$*,* $\mapsto$*,* $\stackrel{\equiv}{\rightarrow}$*,* $\rightarrow$ *et* $\doteq$ *sont contextuelles.*

Preuve 25 *Nous prouvons la conservation de la structure d'arbre et la contextualité de* $\equiv$*,* $\mapsto$*,* $\cong$ *et* $\stackrel{\equiv}{\rightarrow}$ *par induction sur la structure de* $\mathbf{E}$*. On détaille la preuve pour la relation* $\cong$*. Supposons que* $M \cong N$ *avec* $\mathbf{tree}(M,l,p,m)$*. Si* $\mathbf{E} = .$*, le résultat est immédiat. Supposons que pour un contexte* $\mathbf{E}$ *on ait* $\mathbf{E}\{M\}\sigma \equiv \mathbf{E}\{N\}\sigma'$ *avec* σ*,* σ' *deux renommages injectifs. On peut écrire* $\mathbf{E}\{M\} = l : p[P]_{m,S} \mid M'$ *et* $\mathbf{E}\{N\} = l : p[P']_{m,S'} \mid N'$ *.*

$\mathbf{E} = a[\mathbf{E}]$

$$p[\mathbf{E}\{M\}\sigma] = l : p[\mathbf{0}]_{m,n} \mid n : q[P\sigma]_{l,S\sigma} \mid M'\sigma\{n/l\}$$
$$p[\mathbf{E}\{N\}\sigma'] = l : p[\mathbf{0}]_{m,n'} \mid n' : q[P'\sigma']_{l,S'\sigma'} \mid N'\sigma'\{n'/l\}$$

On peut choisir τ*,tau'* *deux renommages injectifs tels que* $p[\mathbf{E}\{M\}]\tau \equiv p[\mathbf{E}\{N\}]\tau'$*.*

$\mathbf{E} = \mathbf{E} \mid Q$

$$\mathbf{E}\{M\}\sigma \mid Q = l : p[P\sigma \mid Q]_{m,S\sigma} \mid M'\sigma$$
$$\mathbf{E}\{N\}\sigma' \mid Q = l : p[P'\sigma' \mid Q]_{m,S'\sigma'} \mid N'\sigma'$$

On conclut que $(\mathbf{E}\{M\} \mid Q)\sigma \equiv (\mathbf{E}\{N\} \mid Q)\sigma'$ *en remarquant que* $\mathbf{names}(Q) \cap \mathbf{names}(\mathbf{E}\{M\}) = \emptyset$ *et* $\mathbf{names}(Q) \cap \mathbf{names}(M') = \emptyset$*.*

$\mathbf{E} = \nu c.\mathbf{E}$ *Similaire aux cas précédents.*

$\mapsto$ *et* $\stackrel{\equiv}{\rightarrow}$ *sont contextuelles, par conséquent* $\rightarrow$ *est contextuelle également.* $\doteq$ *est contextuelle d'après le corollaire B.1.7 et la contextualité de* $\stackrel{\equiv}{\rightarrow}$ *et* $\cong$*.*

Lemme B.1.6 *Pour toute machine bien formée M on a, $M \downarrow_a$ si et seulement si $M_* \cong \downarrow_a$.*

Preuve 26 *Prouvons d'abord le sens direct. On remarque que si M et N sont des machines bien formées, on a*

$$M \cong N \implies (\forall a \in \mathsf{N}, M \downarrow_a \iff N \downarrow_a)$$

La raison est que M et N diffèrent seulement par un renommage des noms de localités autre que **r** *et* **rp**, *et donc ces noms n'interviennent pas dans la définition du prédicat d'observation. Il est alors suffisant de montrer que si M est une machine telle que $M \stackrel{\equiv}{\to} M'$ et $M \downarrow_a$ alors $M' \downarrow_a$. On le montre facilement à l'aide du lemme B.1.1, en considérant les trois cas possibles. L'implication réciproque découle immédiatement de la définition 4.2.2.*

Preuve 27 (Preuve du lemme 4.2.4) *Nous commençons par prouver que la relation $\stackrel{\equiv}{\to}$ termine. Etant donné un terme de machine M, on définit* **size**(M) *comme le cardinal du multi-ensemble*

$$\{P \in \mathsf{MK}, (M \equiv l : p[P \mid R]_{m,S} \mid M') \wedge (P = \nu b.Q \vee P = q[Q] \vee P = \mathbf{reify}(n, M_*))\}$$

où l'on identifie des processus structurellement équivalent. Il est alors facile de voir que $M \stackrel{\equiv}{\to} M'$ pour un certain M' si et seulement si **size**$(M) > 0$, *et a alors de plus* **size**$(M') <$ **size**(M). *Par ailleurs,* **size**$(M) = 0$ *si et seulement si M est en forme normale. On déduit immédiatement que $\stackrel{\equiv}{\to}$ termine. De plus, l'unicité modulo $\cong$ de la forme normale est une conséquence de la première partie du lemme B.1.4.*

Corollaire B.1.7 *Si $M \stackrel{\equiv}{\to}^* \cong M'$, alors $M_* \cong M'_*$.*

Preuve 28 *Supposons que $M \stackrel{\equiv}{\to}^* N \cong M'$. d'après le lemme 4.2.4 on a, $N \stackrel{\equiv}{\to}^* N_*$ et $M_* \cong N_*$. D'après le lemme B.1.3, on a $M' \stackrel{\equiv}{\to}^* M''$ avec $M'' \cong N_*$. On voit alors facilement que M'' est une machine en forme normale. Donc, d'après le lemme 4.2.4, $M'' \cong M'_*$ et finalement, $M_* \cong M'_*$.*

Lemme B.1.8

- *Si $M \to N$ alors $M_* \to \stackrel{\equiv}{\to}^* \cong N_*$.*
- *Si $M_* \to N$ alors $M \to \stackrel{\equiv}{\to}^* N_*$.*

Preuve 29 *La deuxième partie est une conséquence du lemme 4.2.4 et de la définition de $\to$. Pour la première partie, on a $M \stackrel{\equiv}{\to}^* M' \stackrel{\equiv}{\to}^* M_*$ et $M' \mapsto N$. Il découle du lemme B.1.4 que $M_* \mapsto \stackrel{\equiv}{\to}^* \cong N'$ avec $N \stackrel{\equiv}{\to}^* N'$. En particulier, on a $N_* \cong N'_*$. Par ailleurs, par définition $N' \stackrel{\equiv}{\to}^* N'_*$ et $\cong$ est close par réduction pour la relation $\stackrel{\equiv}{\to}$ (lemme B.1.3). On en déduit $M_* \mapsto \stackrel{\equiv}{\to}^* \cong N_*$.*

Preuve 30 (Preuve du lemme 4.2.9) *L'inclusion $\equiv \subseteq \cong$ est une conséquence de la définition $\cong$. Du corollaire B.1.7, on a $\cong \subseteq \doteq$. On considère maintenant les restrictions de ces relations aux machines bien formées. Elles préservent toutes les barbes (cf. lemme B.1.6). La relation $\equiv$ est close par réduction d'après les règles* M.Str *et* M.S.Str. *$\cong$ est close par réduction d'après le lemme B.1.3. On montre la relation $\doteq$ est également close par réduction. Pour cela, supposons que l'on ait $M \doteq N$ et $M \to M'$. D'après la première partie de lemme B.1.8, on a $M_* \to U \stackrel{\equiv}{\to}^* \cong M'_*$ et l'on déduit $N_* \to V \cong U$ ($\cong$ est close*

par réduction). La deuxième partie du lemme B.1.8 nous dit alors que $N \to V' \stackrel{\equiv}{\to}^* V_*$. *Finalement, on a* $N \to V'$ *avec* $V' \doteq M'$.

On conclut que $\equiv$, $\cong$ *et* $\doteq$ *préservent les barbes et sont closes par réduction, et sont par conséquent des bisimulations barbées fortes. Finalement,* $\doteq$ *est préservé par tout contexte d'évaluation* $\mathbf{E}$ *d'après le lemme B.1.5.* $\sim_c$ *est la plus grande bisimulation forte préservée par tout contexte et donc,* $\doteq \subseteq \sim_c$.

B.2 Complétude

Lemme B.2.1

- *Si* $P \equiv P'$ *alors* $[\![P]\!] \doteq [\![P']\!]$
- *Si* $P \stackrel{\equiv}{\to} P'$ *alors* $[\![P]\!] \doteq [\![P']\!]$

Preuve 31 *Nous commençons par généraliser la définition 4.2.1 comme suit :*

$$[\![P]\!]_{l,p,m} = l : p[P]_{m,\emptyset}$$

On prouve alors le résultat plus général que pour tout nom p *et tous noms de localités distincts* l *et* m, $P \equiv P'$ *implique* $[\![P]\!]_{l,p,m} \doteq [\![P]\!]_{l,p,m}$. *Nous raisonnons par induction sur la dérivation de* $P \equiv P'$. *La relation* $\equiv$ *sur les processus de machine est toujours monoidale pour la composition parallèle avec* $\mathbf{0}$ *comme élément neutre. Ainsi, à l'aide de la règle* M.SE.Ctx, *on montre aisément le résultat lorsque* $P \equiv P'$ *correspond à l'une de ces règles monoidale (l'argument est le même pour la règle* M.SE.α*). Détaillons maintenant les autres règles.*

S.Nu.Nil $P = \nu a.\mathbf{0}$ *et* $P' = \mathbf{0}$. $[\![P]\!]_{l,p,m} \stackrel{\equiv}{\to} l : p[\mathbf{0}]_{m,\emptyset} = [\![P]\!]_{l,p,m}$. *On conclut par le corollaire B.1.7 que* $[\![P]\!]_{l,p,m} \doteq [\![P]\!]_{l,p,m}$.

S.Nu.Comm**,** S.Nu.par *Similaire au cas précédent.*

S.Context *Il est suffisant de le montrer pour les trois contextes de base* $q[\,\cdot\,]$, $\nu a.\cdot$ *et* $R \mid \cdot$. *Nous le prouvons pour le troisième cas, les deux autres se traitent de façon similaire. Par induction, on sait que* $[\![P]\!]_{l,p,m} \stackrel{\equiv}{\to}^* N_*$ *et* $[\![P]\!]_{l,p,m} \stackrel{\equiv}{\to}^* N'_*$ *avec* $N_* \cong N'_*$. *On a*

$$[\![P]\!]_{l,p,m} = [\![P]\!]_{l,p,m} \mid R \stackrel{\equiv}{\to}^* N_* \mid R$$

et de même $[\![P]\!]_{l,p,m} \stackrel{\equiv}{\to}^* N'_* \mid R$. *Ainsi* $[\![P]\!]_{l,p,m} \doteq [\![P]\!]_{l,p,m}$ *d'après le corollaire B.1.7.*

Corollaire B.2.2

- *Si* $P \equiv P'$ *alors* $[\![P]\!] \sim_c [\![P']\!]$
- *Si* $P \stackrel{\equiv}{\to} P'$ *alors* $[\![P]\!] \sim_c [\![P']\!]$

Preuve 32 (Preuve du lemme 4.2.16) *Nous commençons par prouver que la relation*

$$S = \{(N\{\mathbf{reify}(m,M_*)/x\}, N\{P_*/x\}), N \in \mathsf{WFM}\}$$

est une bisimulation "up to" $\doteq$. *Nous avons besoin des définitions et sous-lemmes suivants.*

Définition B.2.3 *Une occurrence d'une variable libre* x *est* gardée *dans* M *si elle n'apparaît pas dans la continuation d'un récepteur ou dans un message. Une variable libre est dite gardée si toutes ses occurrences sont gardées.*

Lemme B.2.4 *Si (U,V) appartient à S, alors il existe un N_* avec x gardé dans N_* tel que*

$$U_* \cong N_*\{\mathbf{reify}(m,M_*)/x\}$$
$$V_* \cong N_*\{P_*/x\}$$

Preuve 33 *Soit (U,V) un élément de S. On considère deux cas. Premièrement, supposons que x est gardé dans N. Il est facile de voir que $U_* \cong N_*\{\mathbf{reify}(m,M_*)/x\}$ et $V_* \cong N_*\{P_*/x\}$. La raison est qu'une occurrence gardée d'une variable ne peut pas ne plus l'être après une règle de réduction structurelle. Dans le deuxième cas, on définit un terme N' en renommant toutes les occurrences gardées de x dans N par une variable fraîche y. On a alors*

$$U = N'\{\mathbf{reify}(m,M_*)/y\}\{\mathbf{reify}(m,M_*)/x\}$$
$$V = N'\{P_*/y\}\{P_*/x\}$$

Nous montrons maintenant que $U \stackrel{\equiv}{\rightarrow}^* \cong N''\{\mathbf{reify}(m,M_*)/y\}$ et $V \stackrel{\equiv}{\rightarrow}^* \cong N''\{P_*/y\}$ avec toutes les occurrences de y gardées. On peut alors conclure avec le cas précédent. Pour simplifier, on suppose que N' a seulement une occurrence (non gardée) de x. Le raisonnement est le même que dans le cas général (par induction sur le nombre d'occurrences de x). On suppose que $M_* = l : q[R_*]_{l',S'} \mid M'_*$. Remarquons tout d'abord que, $N' \stackrel{\equiv}{\rightarrow}^* M' = n : m[x \mid Q]_{k,S} \mid M''$. Nous avons alors :

$$\begin{aligned} N'\{\mathbf{reify}(m,M_*)/x\} &\stackrel{\equiv}{\rightarrow} n : p[\mathbf{reify}(m,M_*) \mid Q]_{k,S} \mid M'' \\ &\stackrel{\equiv}{\rightarrow} n : p[R_* \mid Q]_{k,(S,S'\{k_i/l_i\}_{i\in I})} \mid M'_*\{n/l\}\{k_i/l_i\}_{i\in I} \mid M'' \\ &= U' \end{aligned}$$

$$\begin{aligned} N'\{P_*/x\} &\stackrel{\equiv}{\rightarrow} n : p[P_* \mid Q]_{k,S} \mid M'' \\ &\stackrel{\equiv}{\rightarrow}^* n : p[R_* \mid Q]_{k,(S,S'\{k'_i/l_i\}_{i\in I})} \mid M'_*\{n/l\}\{k'_i/l_i\}_{i\in I} \mid M'' \\ &= V' \end{aligned}$$

On déduit alors le résultat où $N' = U'\sigma = V'\sigma'$ avec σ et σ' des renommages injectifs.

Lemme B.2.5 *Si $(U,V) \in S$ et $U \rightarrow U'$, il existe alors V' tel que $V \rightarrow V'$ et $(U''_*,V''_*) \in S$ avec $U''_* \cong U'_*$ et $V''_* \cong V'_*$.*

Preuve 34 *Tout d'abord, en conséquence du lemme B.1.8 et par définition de $\rightarrow$, nous savons qu'il existe un A tel que $U \stackrel{\equiv}{\rightarrow}^* U_* \mapsto A \stackrel{\equiv}{\rightarrow} \cong U'_*$. Deuxièmement, par le lemme B.2.4, $U_* \cong B_*\{\mathbf{reify}(m,M_*)/x\}$ et $V_* \cong B_*\{P_*/x\}$ pour un terme de machine B_* avec x gardé dans B_*. En utilisant ce dernier point, on obtient*

$$U_* \rightarrow U' \iff \begin{cases} B_*(x) \rightarrow B'(x) \\ U' \cong B'\{\mathbf{reify}(m,M_*)/x\} \end{cases}$$

On déduit qu'il existe B' tel que $U' \cong B'\{\mathbf{reify}(m,M_*)/x\}$ et $V_* \rightarrow V' \cong B'\{P_*/x\}$. Par définition de $\rightarrow$, on conclut que $V \rightarrow V'$.

Si (U,V) appartient à S et $U \to U'$, on déduit immédiatement du lemme B.2.5 que $V \to V'$ pour un certain V' tel que $U' \doteq S \doteq V'$. De plus, on a besoin de montrer que $U \downarrow a \iff V \downarrow a$. D'après le lemme B.1.6, il suffit de prouver que $U_* \downarrow a \iff V_* \downarrow a$. D'après le lemme B.2.4 on sait que $U_* = N_*\{\mathbf{reify}(m,M_*)/x\}$ et $V_* = N_*\{P_*/x\}$ avec x gardé dans N_*. On en déduit que U et V possèdent les même barbes. Finalement, on a prouvé que S est une bisimulation forte barbée up to $\doteq$ et

$$\forall N \in \mathsf{WFM}, N\{\mathbf{reify}(m,M_*)/x\} \sim N\{P_*/x\}$$

Nous prouvons maintenant le résultat pour $\sim_c$. Soit N une machine bien formée, il est suffisant de montrer que pour tout contexte $\mathbf{E}$, on a

$$\mathbf{E}\{N\{\mathbf{reify}(m,M_*)/x\}\} \sim \mathbf{E}\{N\{P_*/x\}\}$$

Soit $\mathbf{E}$ un contexte, Q un processus de machine,et y une variable fraîche, on a

$$\begin{aligned}\mathbf{E}\{N\{Q/x\}\} &= \mathbf{E}\{N\{y/x\}\{Q/y\}\} \\ &= \mathbf{E}\{N\{y/x\}\}\{Q/y\} \\ &= N'\{Q/y\} \text{ avec } N' = \mathbf{E}\{N\{y/x\}\}\end{aligned}$$

Or on sait que $N'\{\mathbf{reify}(m,M_*)/y\} \sim N'\{P_*/y\}$ d'après le résultat précédent. On en déduit finalement

$$\mathbf{E}\{N\{\mathbf{reify}(m,M_*)/x\}\} \sim \mathbf{E}\{N\{P_*/x\}\}$$

Preuve 35 (Preuve de la proposition 4.2.14) *Nous raisonnons par induction sur la dérivation de $P \to P'$ et par cas sur la dernière règle d'inférence utilisée.*
R.Out *On a $P = \xi \triangleright Q \mid b[R \mid Q\varphi]$ et $P' = Q\varphi \mid b[R]$.*

$$\begin{aligned}[\![P]\!] &= \mathbf{r} : \mathbf{rn}[P]_{\mathbf{rp},\emptyset} \\ &\overset{\equiv}{\to} \mathbf{r} : \mathbf{rn}[\xi \triangleright Q]_{\mathbf{rp},l} \mid l : q[R \mid Q\varphi]_{\mathbf{r},\emptyset} \qquad \text{(M.S.Cell)}\end{aligned}$$

$$\begin{aligned}[\![P']\!] &= \mathbf{r} : \mathbf{rn}[Q\varphi \mid b[R]]_{\mathbf{r},\emptyset} \\ &\overset{\equiv}{\to} \mathbf{r} : \mathbf{rn}[Q\varphi]_{\mathbf{rp},m} \mid m : q[R]_{\mathbf{r},\emptyset} \qquad \text{(M.S.Cell)}\end{aligned}$$

Par ailleurs, on a par définition

$$l : q[R \mid Q\varphi]_{\mathbf{r},\emptyset} = l : q[R]_{\mathbf{r},\emptyset} \mid Q\varphi$$

D'après le lemme 4.2.4

$$l : q[R]_{\mathbf{r},\emptyset} \overset{\equiv}{\to}^* l : q[R'_*]_{\mathbf{r},S} \mid M_*$$

$$m : q[R]_{\mathbf{r},\emptyset} \overset{\equiv}{\to}^* m : q[R''_*]_{\mathbf{r},S'} \mid N_*$$

On déduit du lemme B.1.5

$$l : q[R \mid Q\varphi]_{\mathbf{r},\emptyset} \overset{\equiv}{\to}^* l : q[R'_* \mid Q\varphi]_{\mathbf{r},S} \mid M_*$$

Finalement, on a

$$\begin{aligned}[\![P]\!] &\stackrel{\equiv}{\to}^* \mathbf{r} : \mathbf{rn}[\xi \triangleright Q]_{\mathbf{rp},l} \mid l : q[R'_* \mid Q\varphi]_{\mathbf{r},S} \mid M_* \\ &\to \mathbf{r} : \mathbf{rn}[Q\varphi]_{\mathbf{rp},l} \mid l : q[R'_*]_{\mathbf{r},S} \mid M_* \qquad \text{(M.Out)} \\ &= U\end{aligned}$$

$$\begin{aligned}[\![P']\!] &\stackrel{\equiv}{\to}^* \mathbf{r} : \mathbf{rn}[Q\varphi]_{\mathbf{rp},m} \mid m : q[R''_*]_{\mathbf{r},S'} \mid N_* \\ &= V\end{aligned}$$

On conclut en remarquant que $U \cong V$ et $[\![P']\!] \doteq V$. D'après les lemmes 4.2.8 et 4.2.9, on déduit $U \sim_c [\![P']\!]$.

R.In, R.Local *Ces deux cas se traitent de la meme façon que le cas* R.Out.

R.Pass

$$\begin{aligned}[\![P]\!] &= \mathbf{r} : \mathbf{rn}[p[P_*] \mid (p[x] \triangleright Q)]_{\mathbf{rp},\emptyset} \\ &\stackrel{\equiv}{\to} \mathbf{r} : \mathbf{rn}[p[x] \triangleright Q]_{\mathbf{rp},l} \mid l : p[P_*]_{\mathbf{r},\emptyset} \qquad \text{(M.S.Cell)}\end{aligned}$$

D'après le lemme 4.2.4 et la proposition 4, on a

$$l : p[P_*]_{\mathbf{r},\emptyset} \stackrel{\equiv}{\to}^* M_* \text{ avec } \mathbf{tree}(M_*,l,p,\mathbf{r})$$

On en déduit

$$\begin{aligned}[\![P]\!] &\stackrel{\equiv}{\to}^* \mathbf{r} : \mathbf{rn}[p[x] \triangleright Q]_{\mathbf{rp},p} \mid M_* \\ &\to \mathbf{r} : \mathbf{rn}[Q\{\mathbf{reify}(m,M_*)/x\}]_{\mathbf{rp},\emptyset} \qquad \text{(M.Pass)} \\ &= U\end{aligned}$$

De plus, comme on a $[\![P']\!] = \mathbf{r} : \mathbf{rn}[Q\{P_*/x\}]_{\mathbf{rp},\emptyset}$ on peut appliquer le lemme 4.2.16 et conclure que $[\![P']\!] \sim_c U$.

R.Context *Par induction, on a $[\![P]\!] \to M$ et $M \sim_c M' = [\![P']\!]$. On veut prouver que pour tout contexte E tel que $\mathbf{E}\{P\} \to \mathbf{E}\{P'\}$ on a $[\![\mathbf{E}\{P\}]\!] \to N$ et $N \sim_c N' = [\![\mathbf{E}\{P'\}]\!]$. On procède par sous-induction sur le contexte E.*

$\mathbf{E} = .$ *Immédiat par induction.*

$\mathbf{E} = p[\mathbf{E}]$

$$\begin{aligned}[\![p[\mathbf{E}\{P\}]]\!] &= \mathbf{r} : \mathbf{rn}[p[\mathbf{E}\{P\}]]_{\mathbf{rp},\emptyset} \\ &\stackrel{\equiv}{\to}\cong p[[\![\mathbf{E}\{P\}]\!]] \\ &\to p[N] \\ &\sim_c p[N']\end{aligned}$$

De plus on a , $[\![p[\mathbf{E}\{P'\}]]\!] \stackrel{\equiv}{\to}\cong p[[\![\mathbf{E}\{P'\}]\!]]$ et par le corollaire B.1.7, $[\![p[\mathbf{E}\{P'\}]]\!] \doteq p[[\![\mathbf{E}\{P'\}]\!]]$, et donc $[\![p[\mathbf{E}\{P'\}]]\!] \sim_c p[[\![\mathbf{E}\{P'\}]\!]]$. Par ailleurs, $p[[\![\mathbf{E}\{P'\}]\!]] \sim_c p[N']$ parce que $\sim_c$ est une congruence. On conclut $[\![p[\mathbf{E}\{P'\}]]\!] \sim_c p[N']$.

$\mathbf{E} = \mathbf{E} \mid Q$ *Comme dans le cas précédent, on a*

$$\begin{aligned}[\![\mathbf{E}\{P\} \mid Q]\!] &= \mathbf{r} : \mathbf{rn}[\mathbf{E}\{P\} \mid Q]_{\mathbf{rp},\emptyset} \\ &= [\![\mathbf{E}\{P\}]\!] \mid Q \\ &\to N \mid Q \\ &\sim_c N' \mid Q\end{aligned}$$

Par ailleurs, on a $[\![\mathbf{E}\{P'\} \mid Q]\!] = [\![\mathbf{E}\{P'\}]\!] \mid Q \sim_c N' \mid Q$.

$\mathbf{E} = \nu c.\mathbf{E}$ *Comme avant*

$$\begin{aligned}[\![\nu c.\mathbf{E}\{P\}]\!] &= \mathbf{r} : \mathbf{rn}[\nu c.\mathbf{E}\{P\}]_{\mathbf{rp},\emptyset} \\ &\stackrel{\equiv}{\to} \mathbf{r} : \mathbf{rn}[\mathbf{E}\{P\}\{i/c\}]_{\mathbf{rp},\emptyset} \\ &\cong \nu c.[\![\mathbf{E}\{P\}]\!] \\ &\to \nu c.N \\ &\sim_c \nu c.N' \\ &\sim_c [\![\nu c.\mathbf{E}\{P'\}]\!]\end{aligned}$$

Ainsi, on a $[\![\nu c.\mathbf{E}\{P\}]\!] \to\sim_c [\![\nu c.\mathbf{E}\{P'\}]\!]$.

R.Struct *D'après le lemme B.2.1,* $[\![P]\!] \doteq [\![P']\!]$ *et* $[\![Q]\!] \doteq [\![Q']\!]$. *Par ailleurs, par induction, on a* $[\![P]\!] \to M \sim_c [\![Q]\!]$. *Puisque* $\doteq$ *est une bisimulation (d'après le lemme 4.2.9), on a* $[\![P']\!] \to M'$ *avec* $M' \doteq M$. *Finalement, comme* $\doteq \subseteq \sim_c$ *(lemme 4.2.9), et comme* $\sim_c$ *est une équivalence (lemme 4.2.8), on déduit que* $M' \sim_c [\![Q']\!]$.

R.Struct.Extr *On utilise le même raisonnement que pour la règle* R.Struct.

B.3 Correction

Nous prouvons maintenant la proposition 4.2.10. Pour ce faire, nous avons besoin d'une deuxième fonction de traduction $[\![.]\!]^{mac}$ des machines vers les processus. La définition utilise deux fonctions mutuellement récursives :

- $[\![.]\!]^{proc}$, des processus de machines vers les processus de machine.
- $[\![.]\!]^{loc}_{.}$, des machines ayant une structure d'arbre vers les processus de machine.

La première fonction a pour rôle de "déplier"les processus réfiés à l'aide de la deuxième fonction. La fonction $[\![.]\!]^{loc}_{.}$ a deux arguments : une machine, et le nom de la localité dans cette machine qui correspond à la racine de l'arbre. Les processus de machine renvoyés par ces deux fonctions peuvent utiliser des noms résolus, et par conséquent ne pas être des processus du Kell calcul. La fonction $[\![.]\!]^{mac}$ lie les noms résolus à l'aide de l'opérateur de restriction (plus précisément, ces noms devraient être remplacés par des noms de processus

frais).

Définition B.3.1

$$\begin{aligned}
[\![\mathbf{0}]\!]^{proc} &= \mathbf{0}\\
[\![u]\!]^{proc} &= u\\
[\![\xi \triangleright P]\!]^{proc} &= \xi \triangleright [\![P]\!]^{proc}\\
[\![\nu a.P]\!]^{proc} &= \nu a.[\![P]\!]^{proc}\\
[\![P \mid P']\!]^{proc} &= [\![P]\!]^{proc} \mid [\![P']\!]^{proc}\\
[\![p[P]]\!]^{proc} &= p[[\![P]\!]^{proc}]\\
[\![p\langle V_1,\ldots,V_n\rangle]\!]^{proc} &= p\langle [\![V_1]\!]^{proc},\ldots,[\![V_n]\!]^{proc}\rangle\\
[\![\mathbf{reify}(l,M_*)]\!]^{proc} &= [\![M_*]\!]^{loc}_l
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{0}]\!]^{loc}_p &= \mathbf{0}\\
[\![M]\!]^{loc}_l &= [\![P]\!]^{proc} \mid \Pi_{i\in 1..n} p_i[[\![M_i]\!]^{loc}_{l_i}]\\
&\quad \textit{avec } M \equiv l : p[P]_{p,S} \mid M_1 \mid \ldots \mid M_n \textit{ et } \mathbf{tree}(M_i,l_i,p_i,l)
\end{aligned}$$

$$[\![M]\!]^{mac} = \nu\widetilde{c}.[\![M]\!]^{loc}_{\mathbf{r}} \ (\widetilde{c} = \mathbf{resnames}(M))$$

Le lemme suivant sera central pour la preuve de la propriété de correction. Intuitivement, il nous permet d'isoler la partie de l'arbre qui peut réagir.

Lemme B.3.2 *Si M est un terme de machine tel que*

- $\mathbf{tree}(M,u,q,v)$
- $M \equiv l : p[P]_{k,S} \mid M' \mid M''$
- $\mathbf{tree}(l : p[P]_{k,S'} \mid M',l,p,k)$
- $S' \subseteq S$

alors il existe un contexte (de processus de machine) E et un processus de machine Q tel que

$$[\![M]\!]^{loc}_u \equiv E\{[\![l : p[P \mid Q]_{k,S'} \mid M']\!]^{loc}_l\}$$

De plus, E et Q ne dépendent pas de $l : p[P]_{k,S'} \mid M'$.

Preuve 36 *On prouve le résultat par induction sur la taille de l'arbre correspondant à M.*

Si $M = u : q[P]_{v,\emptyset}$ alors $q = p$, $u = l$, $v = k$ et $S = S' = \emptyset$. On peut choisir $E = \cdot$ et $Q = \mathbf{0}$.

Dans le cas général, M est de la forme :

$$M \equiv u : q[T]_{v,R} \mid M_1 \mid \ldots \mid M_n \textit{ avec } \mathbf{tree}(M_i,l_i,p_i,u)$$

On a alors :

$$[\![M]\!]^{loc}_u \equiv [\![T]\!]^{proc} \mid p_1[[\![M_1]\!]^{loc}_{l_1}] \mid \ldots p_n[[\![M_n]\!]^{loc}_{l_n}]$$

On peut alors distinguer deux cas : Soit $u = l$, soit le noeud l appartient à l'un des sous-arbres de racine l_i. Dans le premier cas, on a $q = p$, $u = l$, $v = k$, $S = R$ et l'on peut supposer que $S' = 1..m$ avec $m \leq n$.

$$\begin{aligned}
[\![M]\!]^{loc}_u &\equiv [\![T]\!]^{proc} \mid p_1[[\![M_1]\!]^{loc}_{l_1}] \mid \ldots \mid p_m[[\![M_m]\!]^{loc}_{l_m}] \mid \ldots \mid p_n[[\![M_n]\!]^{loc}_{l_n}] \\
&\equiv [\![T]\!]^{proc} \mid p_1[[\![M_1]\!]^{loc}_{l_1}] \mid \ldots p_m[[\![M_m]\!]^{loc}_{l_m}] \mid Q \\
&\equiv [\![T]\!]^{proc} \mid [\![Q]\!]^{proc} \mid p_1[[\![M_1]\!]^{loc}_{l_1}] \mid \ldots p_m[[\![M_m]\!]^{loc}_{l_m}] \\
&\equiv [\![T \mid Q]\!]^{proc} \mid p_1[[\![M_1]\!]^{loc}_{l_1}] \mid \ldots p_m[[\![M_m]\!]^{loc}_{l_m}] \\
&\equiv [\![l : p[T \mid Q]_{k,S'} \mid M_1 \mid \ldots \mid M_m]\!]^{loc}_u
\end{aligned}$$

On obtient le résultat avec $E = \cdot$ et $Q = p_1[[\![M_{m+1}]\!]^{loc}_{l_{m+1}}] \mid \ldots \mid p_m[[\![M_n]\!]^{loc}_{l_n}]$.
Dans le deuxième cas, on peut supposer que le noeud l appartient par exemple au sous-arbre de racine l_n. Par induction, on a

$$[\![M_n]\!]^{loc}_{l_n} \equiv E\{[\![l : p[P \mid Q]_{k,S'} \mid M']\!]^{loc}_l\}$$

On déduit que

$$[\![M]\!]^{loc}_u \equiv E'\{[\![l : p[P \mid Q]_{k,S'} \mid M']\!]^{loc}_l\}$$

avec

$$E' \equiv [\![T]\!]^{proc} \mid p_1[[\![M_1]\!]^{loc}_{l_1}] \mid \ldots \mid p_{n-1}[[\![M_{n-1}]\!]^{loc}_{l_{n-1}}] \mid E$$

Corollaire B.3.3 *Si M est un terme de machine tel que*

- $M \equiv l : p[P]_{l',S} \mid M' \mid M''$
- $\mathbf{tree}(l : p[P]_{l',S'} \mid M', l, p, l')$
- $S' \subseteq S$

Alors il existe un contexte E et un processus de machine Q tels que

$$[\![M]\!]^{mac} \equiv \nu\widetilde{c}.E\{[\![l : p[P \mid Q]_{l',S'} \mid M']\!]^{loc}_l\}$$

avec $\widetilde{c} = \mathbf{resnames}(l : p[P]_{l',S'} \mid M')$. De plus, E et Q ne dépendent pas de $l : p[P]_{l',S'} \mid M'$.

Preuve 37 (Preuve du lemme 4.2.11) *D'après le lemme B.1.1, on sait que $M \equiv L \mid M'$ avec $L = l : p[P]_{k,S'}$ et $L \stackrel{\equiv}{\rightarrow} M''$. On raisonne par cas sur la forme de $L \stackrel{\equiv}{\rightarrow} M''$.*

M.S.Cell *On a $L = l : p[q[P] \mid Q]_{n,S}$ et $M'' = l : p[Q]_{n,(S,l')} \mid l' : q[P]_{l,\emptyset}$. D'après le lemme B.3.2 on déduit*

$$[\![M]\!]^{mac} = E\{[\![l : p[q[P] \mid Q]_{n,\emptyset}]\!]^{loc}_l\}$$

$$[\![N]\!]^{mac} = E\{[\![l : p[Q]_{n,\{l'\}} \mid l' : q[P]_{l,\emptyset}]\!]^{loc}_l\}$$

En utilisant la définition de la fonction $[\![.]\!]^{loc}_.$ il est alors facile de vérifier que $[\![M]\!]^{mac} \equiv [\![N]\!]^{mac}$.

M.S.New *Dans ce cas, on a* $L = l : p[(\nu c.P) \mid Q]_{n,S}$ *et* $M'' = l : p[P\{l/c\} \mid Q]_{n,S}$. *On peut appliquer le lemme B.3.2 :*

$$[\![M]\!]^{mac} = E\{[\![l : p[(\nu c.P) \mid Q]_{n,\emptyset}]\!]^{loc}_l\}$$

$$[\![N]\!]^{mac} = \nu i.E\{[\![l : p[P\{i/c\} \mid Q]_{n,\emptyset}]\!]^{loc}_l\}$$

Il est alors facile de voir que $[\![M]\!]^{mac} \stackrel{\equiv}{\Rightarrow}^{*} [\![N]\!]^{mac}$.

M.S.Act *Comme avant, on applique le lemme B.3.2*

$$[\![M]\!]^{mac} = E\{[\![l : q[\mathbf{reify}(m,M_*) \mid P]_{n,\emptyset}]\!]^{loc}_l\}$$

$$[\![N]\!]^{mac} = E\{[\![l : q[R_* \mid P]_{n,S'\{n_i/l_i\}_{i\in I}} \mid M'_*\{l/m\}\{n_i/l_i\}_{i\in I}]\!]^{loc}_l\}$$

et l'on vérifie que $[\![M]\!]^{mac} \equiv [\![N]\!]^{mac}$.

Preuve 38 (Preuve du lemme 4.2.12) *On procède par cas sur la dérivation* $L \to L'$ *à l'aide du lemme B.1.2.*

M.Out

$$M \equiv l : p[\xi\varphi \mid P]_{l',S} \mid l' : q[(\xi \triangleright Q) \mid R]_{n',S'} \mid M'$$

$$N \equiv l : p[P]_{l',S} \mid h' : q[Q\varphi \mid R]_{n',S'} \mid M'$$

On peut appliquer le lemme B.3.2:

$$[\![M]\!]^{mac} = \mathbf{E}\{[\![l' : q[(\xi \triangleright Q) \mid R]_{n',\{l\}} \mid l : p[\xi\varphi \mid P]_{l',S} \mid M']\!]^{loc}_{n'}\}$$

$$[\![N]\!]^{mac} = \mathbf{E}\{[\![l' : q[Q\varphi \mid R]_{n',\{l\}} \mid l : p[P]_{l',S} \mid M']\!]^{loc}_{n'}\}$$

On vérifie alors que $[\![M]\!] \to [\![N]\!]$.

M.In, M.Local *Similaire au cas précédent.*

M.Pass *On procède comme précédemment*

$$M \equiv l : q[(p[x] \triangleright P) \mid Q]_{n,S} \mid M_* \mid M'$$

$$N \equiv l : q[P\{\mathbf{reify}(m,M_*)/x\} \mid Q]_{n,S\setminus m} \mid M'$$

On a alors, d'après le lemme B.3.2,

$$[\![M]\!]^{mac} = \mathbf{E}\{[\![l : q[(p[x] \triangleright P) \mid Q]_{n,\{l\}} \mid M_* \mid M']\!]^{loc}_n\}$$

$$[\![N]\!]^{mac} = \mathbf{E}\{[\![l : q[P\{\mathbf{reify}(m,M_*)/x\} \mid Q]_{n,\emptyset} \mid M']\!]^{loc}_n\}$$

On vérifie alors facilement que $[\![M]\!] \equiv [\![N]\!]$.

Lemme B.3.4 *Si* $M \to N$ *alors* $[\![M]\!]^{mac} \to [\![N]\!]^{mac}$.

Preuve 39 *Immédiat par les lemmes 4.2.12 et 4.2.11.*

Lemme B.3.5 *Pour toutes machines* M *et* N_* *telles que* $\mathbf{tree}(N_*,l,p,m)$ *avec de plus* M *bien formé, on a*

$$M\{\mathbf{reify}(l,N_*)/x\} \sim_c M\{[\![N_*]\!]^{loc}_l/x\}$$

Preuve 40 *D'après le lemme 4.2.16, il est suffisant de prouver que*

$$l : p[[\![N_*]\!]_l^{loc}]_{m,\emptyset} \stackrel{\equiv}{\rightarrow}^* \cong N_*$$

On le prouve par induction sur la taille de l'arbre N_*. Si N_* ne consiste qu'en une seule localité, on a alors $N_* = l : p[P_*]_{m,\emptyset}$. On en déduit $[\![N_*]\!]_l^{loc} = [\![P_*]\!]^{proc} = P_*$ et $l : p[[\![N_*]\!]_l^{loc}]_{m,\emptyset} = N_*$. Dans le cas contraire, on a

$$N_* = l : p[P_*]_{m,S} \mid N_*^1 \mid \ldots \mid N_*^n \text{ avec } \mathbf{tree}(N_*^i,l_i,n_i,l)$$

$$\begin{aligned} l : p[[\![N_*]\!]_l^{loc}]_{m,\emptyset} &= l : p[[\![P_*]\!]^{proc} \mid \Pi_{i\in 1..n} p_i[[\![N_*^i]\!]_{l_i}^{loc}]]_{m,\emptyset} \\ &\stackrel{\equiv}{\rightarrow}^* l : p[[\![P_*]\!]^{proc}]_{m,\{l'_1,\ldots,l'_n\}} \mid \Pi_{i\in 1..n} l'_i : p_i[[\![N_*^i]\!]_{l_i}^{loc}]]_{l,\emptyset} \\ &\stackrel{\equiv}{\rightarrow}^* l : p[P_*]_{m,\{l'_1,\ldots,l'_n\}} \mid \Pi_{i\in 1..n} N_*'^i \end{aligned}$$

De plus, $N_*'^i \cong N_*^i$. On déduit que $p : p[P_*]_{m,\{l'_1,\ldots,l'_n\}} \mid \Pi_{i\in 1..n} N_*'^i \cong N_*$.

Lemme B.3.6 *Si* $\mathbf{tree}(M,l,p,m)$ *et qu'il n'y a pas de processus réifié dans M, alors $[\![[\![M]\!]_l^{loc}]\!]_{l,n,m} \doteq M$. De plus, si M est bien formé, on a $[\![[\![M]\!]^{mac}]\!] \doteq M$.*

Preuve 41 *On raisonne par induction sur la taille de l'arbre. On peut écrire :*

$$M \equiv l : p[P]_{m,S} \mid M^1 \mid \ldots \mid M^n \text{ avec } \mathbf{tree}(M^i,l_i,p_i,l)$$

On a alors :

$$\begin{aligned} [\![[\![M]\!]_l^{loc}]\!]_{l,p,m} &= [\![[\![P]\!]^{proc} \mid \Pi_{i\in 1..n} p_i[[\![M^i]\!]_{l_i}^{loc}]]\!]_{l,p,m} \\ &= l : p[P \mid \Pi_{i\in 1..n} p_i[[\![M^i]\!]_{l_i}^{loc}]]_{m,\emptyset} \\ &\stackrel{\equiv}{\rightarrow}^* l : p[P]_{m,\{l_1,\ldots,l_n\}} \mid \Pi_{i\in 1..n} l_i : p_i[[\![M^i]\!]_{l_i}^{loc}]_{l,\emptyset} \\ &= p : p[P]_{m,S} \mid \Pi_{i\in 1..n} [\![[\![M^i]\!]_{l_i}^{loc}]\!]_{l_i,p_i,l} \end{aligned}$$

On peut appliquer l'hypothèse d'induction à $[\![[\![M^i]\!]_{l_i}^{loc}]\!]_{l_i,p_i,l}$. On a :

$$[\![[\![M^i]\!]_{l_i}^{loc}]\!]_{l_i,p_i,l} \stackrel{\equiv}{\rightarrow}^* M'^i$$

avec $M'^i \cong M_*^i$ et

$$[\![[\![M]\!]_l^{loc}]\!]_{l,p,q} \stackrel{\equiv}{\rightarrow}^* l : n[P]_{q,S} \mid \Pi_{i\in 1..n} M'^i = M'$$

Par ailleurs, on remarque que $M \stackrel{\equiv}{\rightarrow}^* \cong M'$. On en déduit $[\![[\![M]\!]_l^{loc}]\!]_{l,p,m} \doteq M$.

Si M est bien formé et $\widetilde{q} = \mathbf{resnames}(M)$ on a d'après la règle M.S.NEW *:*

$$[\![[\![M]\!]^{mac}]\!] = [\![\nu\widetilde{q}.[\![M]\!]_{\mathbf{r}}^{loc}]\!] \stackrel{\equiv}{\rightarrow}^* [\![[\![M\{\widetilde{r}/\widetilde{q}\}]\!]_{\mathbf{r}}^{loc}]\!]$$

où $\widetilde{r}$ est un vecteur de noms frais. Par le corollaire B.1.7 on a :

$$[\![[\![M]\!]^{mac}]\!] \doteq [\![[\![M\{\widetilde{r}/\widetilde{q}\}]\!]_{\mathbf{r}}^{loc}]\!]$$

Par ailleurs, par définition de $\cong$, on a $M\{\widetilde{r}/\widetilde{q}\} \cong M$. Or on a montré précédemment que

$$[\![[\![M\{\widetilde{r}/\widetilde{q}\}]\!]_{\mathbf{r}}^{loc}]\!] \doteq M\{\widetilde{r}/\widetilde{q}\}$$

Finalement, on en déduit que $[\![[\![M]\!]^{mac}]\!] \doteq M$.

Preuve 42 (Preuve du lemme 4.2.13) *La deuxième partie est immédiate. Pour la première partie, on définit une machine N, identique à M mais où l'on remplace chaque occurrence de processus réifié par une variable fraîche x_i, telle que $M = N\{\mathbf{reify}(l_i,M_*^i)/x_i\}$. Par définition de $[\![.]\!]^{mac}$, on voit facilement que $[\![M]\!]^{mac} = [\![N\{[\![M_*^i]\!]_{l_i}^{loc}/x_i\}]\!]^{mac}$. Par ailleurs, à partir du lemme B.3.6 on a $[\![[\![N]\!]^{mac}]\!] \stackrel{\equiv}{\rightarrow}^* U$ et $N \stackrel{\equiv}{\rightarrow}^* U'$ avec $U \cong U'$. En particulier,*

$$[\![[\![N\{[\![M_*^i]\!]_{l_i}^{loc}/x_i\}]\!]^{mac}]\!] \stackrel{\equiv}{\rightarrow}^* U\{[\![M_*^i]\!]_{l_i}^{loc}/x_i\}$$
$$N\{\mathbf{reify}(l_i,M_*^i)/x_i\} \stackrel{\equiv}{\rightarrow}^* U'\{\mathbf{reify}(l_i,M_*^i)/x_i\}$$

Et donc

$$[\![[\![N\{[\![M_*^i]\!]_{l_i}^{loc}/x_i\}]\!]^{mac}]\!] \doteq U\{[\![M_*^i]\!]_{l_i}^{loc}/x_i\}$$
$$N\{\mathbf{reify}(l_i,M_*^i)/x_i\} \doteq U'\{\mathbf{reify}(l_i,M_*^i)/x_i\}$$

De plus, on a $U'\{\mathbf{reify}(l_i,M_*^i)/x_i\} \cong U\{\mathbf{reify}(l_i,M_*^i)/x_i\}$. En utilisant le lemme B.3.5, on déduit finalement :

$$U\{[\![M_*^i]\!]_{l_i}^{loc}/x_i\} \sim_c U\{\mathbf{reify}(l_i,M_*^i)/x_i\}$$

Preuve 43 (Preuve de la proposition 4.2.10) *Supposons que $[\![P]\!] \rightarrow M$. Alors on a d'après le lemme B.3.4, $[\![[\![P]\!]]\!]^{mac} \rightarrow [\![M]\!]^{mac} = P'$. Si l'on applique la deuxième partie du lemme 4.2.13, on a $[\![[\![P]\!]]\!]^{mac} \equiv P$. De plus, on a $[\![P']\!] = [\![[\![M]\!]^{mac}]\!] \sim_c M$ par la première partie du lemme 4.2.13. On conclut finalement que $P \rightarrow P'$ avec $[\![P']\!] \sim_c M$.*

Annexe C

A Prototype Implementation of the Kell Calculus

C.1 Introduction

The Kell calculus [60, 8] is a family of higher-order process calculi with hierarchical locations and location passivation, which is indexed by the pattern language used in input constructs. It has been introduced to study programming models for wide-area distributed systems and component-based systems.

We present in this document a simple programming language, called `CHALK` based on an instance of the Kell calculus, and an interpreter for this language. The interpreter is briefly described in section C.2. We introduce the language in section C.3.series The complete syntax of the language is given in section C.5. Comments on the implementation are given in section C.6. Finally, section C.7 discusses the main differences between `CHALK` and the Kell calculus. We conclude with future work in section C.4.

C.2 Interpreter

The interpreter can be built with the `make` command in the root directory of the archive (an ocaml compiler is needed). The options are given by the following command:

```
$ chalk --help
usage: chalk file
  -v version
  -p port
  -o output name
  -s output size : default = 30 cycles
  -help  Display this list of options
  --help  Display this list of options
```

In order to allow several interpreters to communicate, a port number is needed and must be given after the `-p` flag. The `-o` flag is used to specify an output file that contains an html trace of execution. The size of the trace, specified as a number of execution steps, may be given after the `-s` flag. The `test` directory contains short programs that can be

run with the command `./runtests`. The `examples` directory contains all the examples programs presented in section C.3.

C.3 Chalk

C.3.1 Language

For a formal presentation of the calculus and the interpreter, we refer the reader to [8]. In this section, we present the language informally through a series of examples.

Names, messages and receivers The main constructs of the language are *asynchronous messages* and *receivers* over *named channels*. Names must be explicitly defined before they can be used. However, a few names are pre-defined and allow to access library services that can be seen as particular receivers. The following simple program consists in a message on the predefined name `echo` (the prefix `env.` is needed and will be explained below).

```
(* helloworld.kcl *)
env.echo<"Hello World\n">
```

```
$ chalk helloworld.kcl
Hello World
```

The next program consists in two messages and a receiver over a *newly created name* `a`. The term `|` is the *parallel composition* operator, with the semantics that the left and right arguments execute concurrently. Upon reception of a message, the receiver releases its continuation (*i.e.* the code between the braces) where the formal parameter `x` is substituted by the value carried by the message. The keyword `once` means that the receiver can receive at most one message. Alternatively, the keyword `on` is used for *replicated* receivers, *i.e.* receivers that can process an arbitrary number of messages.

```
(* simplemessage.kcl *)
new a in
a<"Hello"> |
a<"World"> |
once a<x> { env.echo<x> }
```

```
$ chalk simplemessage.kcl
Hello
```

The values can be of primitive types (string, integer, list), but can also be names, higher-order values and expressions built upon various operators. The next example shows how the transmission of name can be used to encode synchronous calls. The name `r` is called a return channel. It is created by the emitter of a message, and used by the receiver to send a return value (the concatenation of `x` and `y` in the example).

```
(* synchronous.kcl *)
new a in
```

```
( new r in
  a < "hello ", "world\n", r > |
  once r < x > { env.echo < x >  }
) |
once a < x, y, r > { r < x ^ y > }
```

```
$ chalk synchronous.kcl
Hello World
```

Messages can carry higher-order values: processes, and more generally parametrized processes. The following two examples illustrate these two cases. The term `@` is the application operator. Their execution will still send a `hello world` string on the standard output.

```
(* higherorder.kcl *)
new a in
a < { env.echo <"Hello World"> } > |
once a < x > { x }
```

```
(* abstraction.kcl *)
new a in
a < (x){ env.echo <x> } > |
once a < x > { x @ "Hello World"}
```

Locations We now present the important concept of *named location* or *kells*. A kell is a programming structure of the form `a[...]`, where `a` is a name. The dots can be replaced by an arbitrary program. A program in a kell will execute normally, much like if it were under parenthesis: the first role of a kell is to delimitate some part of a program and give it a name.

However, a location serves more useful purposes. First, it restricts the interaction between a program and its environment. A possible application is, for instance, to execute a potentially unsafe process in a controlled environment. Second, it provides a mean to dynamically control a part of a program during its execution. A program in a kell can be for instance dynamically stopped, duplicated, moved, or replaced by another program.

The first point is achieved through the use of *oriented messages* and *oriented receivers*. Messages can cross only one kell boundary at a time, and must explicitly mention it. For instance, the communication in the following example is not possible.

```
(* simplekell.kcl *)
new a in new b in
a<"Hello World"> |
b [
  once a<x> { env.echo(x) }
]
```

Messages can exhibit three different directions, they can be local (which was the case in the previous examples), directed upward (upward messages are prefixed by `env.`) or directed downward to a specific kell (prefixed by the name of the destination kell). Receivers

must exhibit a complementary direction and be located accordingly. The following example shows how a process in a kell can interact with its environment (the process outside the kell).

```
(* location.kcl *)
new a in new signal in new b in
b [
   once a up < > { env.signal < > }
]
|
b.a < >
|
once signal dn < > { env.echo < "Hello World" > }
```

A receiver listening downward can receive messages from any direct subkells. Optionally, it can get the name of the emitter subkell, as shown in the next example.

```
(* upmsg.kcl *)
new a in new signal in new b in
b [ env.signal < > ] |
a [ env.signal < > ] |
once signal dn x < > {
  if x = a then
     env.echo < "a" >
  else
     env.echo < "b" >
}
```

We can now justify the keyword `env` prefixing the library call `echo`: we made the choice to model library services as receivers defined upward the top-level program: messages sent to library services have to be directed upward. This point is detailed in subsection C.3.2. As a consequence, programs located in kells can only access library services if some kind of forwarder, or proxy, is available.

The ability to control subkells is obtained by generalizing the notion of receiver. We can see a kell as a message, whose content is a running process. We then define a corresponding receiver of the form `once a[x]{...}`. Upon reception (that we call *passivation*), the kell is consumed and `x` is bound to a value capturing the state of the process in location `a` at passivation time. This process can be reactivated simply by using `x` as a normal program. The next example depicts a passivation action. A program in a location `a` continuously sends increasing natural numbers to its environment. These numbers are received and printed. When the number 10 is received, a receiver whose role is to kill `a` is created. Note that the semantics of the Kell calculus (and of `CHALK`) gives no information on the order in which messages are processed. In particular, we cannot deduce from the following example that the output of messages will stop at 9 (even if it is the case in the current implementation).

```
(* passiv.kcl *)
new a in new signal in
```

```
a [
  on signal < i > {
     signal < i + 1 > | env.signal < i >
  } |
  signal < 0 >
] |
on signal dn < i > {
  if i = 10 then
     once a [ x ] { nil }
  else
     env.echo_int < i >
}
```

```
$ chalk passiv.kcl
0123456789
```

Distribution The language provides a very simple communication model based on asynchronous channels carrying strings and interpreter identifiers (IID). Every interpreter owns a unique network identifier that can be obtained by two different ways. First, programs may use the predefined name `thisloc` that is always bound to the IID of its interpreter. Second, the operator `vm` takes a hostname and a port number and returns the corresponding IID. The names `send` and `receive` are predefined and are used to communicate strings from one interpreter to an other. A message of the form `env.send < iid, value >` can be used to send the value `value` to the interpreter identified by `iid`. This value can be received by a receiver, on the destination interpreter, of the form `on receive up < value > {...}`. Other types of value can be transmitted using the `marshall` and `unmarshall` operators. These operators require as additional argument the type of the transmitted value (cf. subsection C.3.4), and can generate a runtime error if the value and type do not match.

The following example consists in a simple client executing a program in a location `a`. This program is stopped during its execution, its state is transformed as a string using the marshalling operator, and then transmitted to the server. Note that the receiver `on echo_int ...` forwards the messages on `echo_int` from the kell `a` to the library service. The server receives the string, transforms it back into a process and resumes its execution.

```
(* client.kcl *)
new a in new signal in new move in
let serverid = vm ("localhost", 8000) in
a [
 on signal < i > {
   if i = 5 then
      env.move < > | signal < i +1 >
   else if i = 10 then
       nil
   else
       ( env.echo_int < i > | signal < i + 1 > )
 } |
```

```
 signal < 0 >
]
|
once move dn < > {
  once a [ x ] {
     env.send < serverid, marshall x : proc >
  }
}
|
on echo_int dn < x > { env.echo_int < x > }

(* server.kcl *)
on receive up < s > {
   let agent = unmarshall s as proc in agent
}
```

The execution of the server and the client in two different terminal (possibly on different machines) gives the following result.

```
$ chalk client.kcl -p 7000
01234

$ chalk server.kcl -p 8000
6789
```

C.3.2 Libraries

We can model the libraries and the distributed communication model as a `CHALK` program. A set of program P_i executed by different interpreters can be modeled by the following term

$$\texttt{Net} \mid \texttt{vmid}_0[\texttt{Lib} \mid u[P_0]] \mid \ldots \mid \texttt{vmid}_n[\texttt{Lib} \mid u[P_n]]$$

Where we assume all $\texttt{vmid}_i$ to be distinct. The processes `Lib` model the local libraries and `Net` the network. In our implementation they are defined as follows:

```
Lib =
  on send dn <x, y> { env.send <x, y> } |
  on receive up <x> { receive dn <x> } |
  on echo <x> { print_string x } |
  on echo_int <x> { print_int x }

Net = on send dn <x, y> { x.receive <y> }
```

The terms `print_int` and `print_string` do not belong to the model but correspond to ocaml function in our implementation. We can see them as the `nil` process.

The names `echo`, `echo_int`, `send` and `receive` are predefined and associated to the following types.

```
   echo : <string>
echoint : <int>
   send : <vmid, string>
receive : <string>
```

C.3.3 Function encoding

To the core language described below, we add a few additional constructs in order to provide synchronous communications, similar to function calls. These constructs are not primitive in the language, but defined as a syntactic sugar, and are translated into the core language after the parsing.

In the current version of `CHALK` there is no syntactic difference in the definition of synchronous and asynchronous messages and receivers. An error is raised by the typechecker if the messages and receivers are not used in a coherent way.

Synchronous messages can now be used as values, and the corresponding receiver body must always execute a `reply` instruction. As with asynchronous communications, synchronous communications are oriented.

The following example corresponds to a local communication. The message on channel `plus` is used in a context where an integer is expected. The typechecker expects the corresponding receiver to return an integer, which is the role of the `reply` instruction in the receiver body.[1].

```
(* call.kcl *)
new plus in
on plus < x, y > {
   reply (x + y) to plus
}
|
env.echo_int < plus <1,2> >
```

Its execution gives the expected result.

```
$ chalk call.kcl
3
```

The next example is a synchronous call oriented upward.

```
(* callup.tex *)
new plus in new a in
a [
  env.echo_int < env.plus <2,3> >
]
|
on plus dn < x, y > {
```

1. the name `plus` corresponds to the channel on which the receiver is listening, and could be deduced from the context, but has to be specified anyway.

```
  reply (x + y) to plus
}
|
on echo_int dn < s > { env.echo_int < s > }

$ chalk callup.kcl
5
```

The last example is a synchronous call oriented downward.

```
(* calldown *)
new plus in new a in
a [
  on plus up < x, y > {
    reply (x + y) to plus
  }
]
|
env.echo_int < a.plus<1,2> >

$ chalk calldown.kcl
3
```

Furthermore, we define a sequencing operator `;` with the usual semantics. Process on the left hand-side must be a nil value `()`: usually a call to a trigger returning the nil value, For instance, the library service `print`.

```
(* sequence.kcl *)
env.print < "hello " > ; env.print < "world" > ; nil

$ chalk sequence.kcl
Hello World
```

In the following, we explain how this function calls are translated into the core language. Suppose we have a process P that uses a message (let say `a<v>` in a place where a value is expected. We replace the message by a *fresh* variable in P, (that is, a variable not already in P), for instance `x` and we call Q the resulting program. We replace `P` by

```
new r in (once r<x> { Q } | a<v,r>)
```

We modify as well the trigger(s) on channel `a`. For instance, a trigger of the form

```
on/once a<x1,...,xn> {
  ...
  reply v to a
}
```

will be transformed in

```
on/once a<x1,...,xn,reply> {
  ...
  reply<v>
}
```

If the message is of the form `env.a<v>` (*i.e.* listening upward), then we replace P with

```
new r in (once r<x> { Q } | a up <v,r>)
```

and the trigger(s) by

```
on/once a dn loc <x1,...,xn,reply> {
  ...
  loc.reply<v>
}
```

Remark that, in this case, we need to know which kell is the sender of the message. It is possible thanks to the `loc` variable.

The last case corresponding to a message oriented downward is treated in the same fashion.

C.3.4 Types

CHALK programs are typed using a simple monomorphic type system that we do not detail here. The syntax of types for the core language is given in C.5.2. Type information are deduced from the context but can be optionally specified by the programmer, in the `new` declarations, as in the following example.

```
(* type.kcl *)
new a : < proc > in
a < { env.echo <"Hello World"> } > |
once a < x > { x }
```

C.3.5 Runtime errors

A CHALK program can fail in two cases.

– two sibling kells have the same name.

– an unmarshalled value has not the expected type.

C.4 Future Work

The motivation behind CHALK is to evaluate the pertinence of the primitives of the Kell calculus in a programming language suited for component-based or distributed programming. In order to do so, we need firstly, more realistic programming constructs and secondly to be able to integrate library services in an uniform way. This could be achieved... how?

In the current version of CHALK, library services are modeled as receivers in the parent location of the (programmer) toplevel. As a consequence, subkells that want to use these services require the use of forwarders or proxies. An other solution would be to consider libraries as particular subkells. Following this approach, we could also use CHALK as a script language to glue components, in the line of the composition language Piccola ([51]). However, this raises the issue of passivating (and thus saving the execution state of) a subkell written in a possibly different language. A solution could be to forbid, or to constrain, the passivation of such specific subkells.

Having libraries as subkells has an other drawback: the same library could not be used directly by two different kells, as it would need to be a subkells of both kells, which is not allowed in our hierarchical locations model. This problem has been given a solution in [38]. We plan to extend the current implementation to allow the sharing of kells, following their approach.

The type system of chalk is basic and could be extended in many ways. A desirable feature would be to prevent the two kinds of runtime errors. The unmarshalling type error is not specific to `CHALK`. Solutions that mix dynamic and static types have been proposed ([1], [20]) and could be worth investigating in our setting. Besides, we are currently working on a type system to preventing sibling kells of sharing identical names.

C.5 Syntax

C.5.1 Core language

Process ::=	`nil`	Nil Process
	Name	Process Variable
	`let` *Name* `in` *Process*	Definition
	`new` *Name* [`:` *Type*] `in` *Process*	Name Creation
	Trig Pattern `{` *Process* `}`	Trigger
	`if` *Value* `then` *Process* `else` *Process*	If - Then - Else
	Process `|` *Process*	Parallel Composition
	`(` *Process* `)`	Bracketed Process
	DirMsg Name`<` *Value* `>`	Message
	Name`[` *Process* `]`	Kell

Value ::=	`()`	Void
	`[]`	Empty list
	`true`	True
	`false`	False
	String	String
	Integer	Integer
	Name	Variable
	Value `,` *Value*	Tuple Value
	Value Op Value	Binary operator
	UOp Value	Unary operator
	`(` *Value* `)`	Bracketed Value
	Args`{` *Process* `}`	Abstraction
	`marshall` *Value* `:` *Type*	Marshalling
	`unmarshall` *Value* `as` *Type*	Unmarshalling

Args ::=		Empty list
	`(`*Name* [`:` *Type*]`)` *Args*	Parameter

UOp ::=	`not`	Logical Negation
	`-`	Arithmetic Negation
	`vm`	Machine Name Constructor
	`head`	Head
	`tail`	Tail
	`isnil`	Isnil

Op ::=	`::`	Cons
	`+`	Addition
	`-`	Subtraction
	`/`	Division
	`and`	Logical And
	`or`	Logical Or
	`=`	Equality
	`@`	Application

Trig ::=	`once`	Simple trigger
	`on`	Replicated trigger

Pattern ::=	*Name Dir* `<` *ListeName* `>`	Simple Pattern
	Name `[` *Name* `]`	Passivation

Dir ::=	`up`	Up
	`dn` *Name*	Down
	`dn`	Down
		In

DirMsg ::=	`env.`	Up
	Name`.`	Down
		In

C.5.2 Types

Type ::=	*Name*	Type Variable
|	`unit`	Unit Type
|	`bool`	Boolean Type
|	`string`	String Type
|	`int`	Integer Type
|	`kell`	Locality Type
|	`vmid`	Machine Identifier Type
|	`proc`	Process Type
|	*Type* `list`	List Type
|	`<` *Type* `>`	Channel Type
|	*Type* `,` *Type*	Tuple Type
|	*Type* `->` *Type*	Abstraction Type

C.5.3 Functions encoding

Process ::=	...	
|	*Process*`;`*Process*	Sequencing operator
|	`reply` *Value* `to` *Name*	Reply

Value ::=	...	
|	*DirMsg Name* `<` *Value* `>`	Message

C.6 Implementation

We detail the general organization of the code source.

`Parser`, `Lexer` These two modules are generated by `ocamllex` and `ocamlyacc` and the files `parser.mly` and `lexer.mll`.

`Basesynt`, `Syntax`, `Ktype`, `Transform` Definition of the abstract syntax of the language and the types. The `ktype.generate` function generates fresh type variables and is used by the parser to fill the types not specified by the programmer. `Basesynt` define the abstract syntax of the language interpreted by the runtime and `Syntax` define the abstract syntax of the source language (including function calls). The module `Transform` defines a function that translates a program from the source language to the core language.

`Typecheck` Definition of the function `type_check` that returns `true` if a process is well-typed and `false` if not. This function first generates a set of constraints based on the typing rules of the language and then try to solve these constraints using a unification algorithm. As a side effect, it sets two string variables `html_constraint` and `html_solved` that contains the generated constraints and a solution if the program is typable.

`Html, Error` These modules allow the generation of html and error messages.

`Distr` This module acts as a layer between the runtime and the network. It defines functions that send and receive messages over the network. Uses the Unix socket interface and UDP protocol.

`Uin` Generates unique identifier. The function `generate` returns globally unique identifiers and `generate_well_known` returns identifiers that are used in order to designate library services, that are defined in every interpreter.

`Value` This module defines several datatypes. We describe only the most important ones. type `t` corresponds essentially to syntactic values in normal form (*i.e.* evaluated, without variable or operators). Moreover, a new kind of value is added for the passivated processes. Note that this type is polymorphic in the `'location` type variable, to overcome the mutual recursion between the type of values and the type of locations (corresponding to passivated processes). The type `environment` defines associations list between variables and values. Environment are classically used in order to avoid the substitution operation. A term of type `closure` is used to represent higher-order values: a program and its environment.

The main function of this module is `evaluate` which computes the normal form of a syntactic value in a given environment.

`Location` This module is the most complex of the system and implements all the reduction rules of the language. It defines a type `t`. A value of this type corresponds to the runtime structure of a kell. The module allows only the creation of the root location, with the `create_root` function, which corresponds to the initial state of the interpreter. This function take as argument a closure: the initial program and an environment that bind the well-know names (*i.e.* library services) to their respective value.

Sublocations might be created as a side effect of the function `reduction` which executes the content of a kell. These sublocations can be obtained with the `sublocations` function, and then executed with `reduction`.

The function `connect_to_lib` is used to connect the root location to the module `lib` that models the library. The `get_interface` returns an access point to the root location that will be used the module `lib`.

`Lib` This module acts as a layer between the root location and the outside world. From the root location, it is simply its parent location. The difference is that it is implemented differently. However, the interface is similar. The `create`, `reduction` and `get_interface` functions have the same semantics as those in module `location`. The function `connect_to_root`

allow to connect the lib location to the root location using the access point returned by `Location.get_interface`.

It also defines the initial typing environment and the predefined names. (`initenv` and `typingenv`).

`Runtime` This module is initialized with an abstract syntax tree and a port number by the function `init_runtime`. If the port number is valid, the runtime is executed in *server mode*, which allow it to receive remote messages. This function essentially create the lib and root locations and connect them to each other.

The function `schedule` executes one step of reduction of the program. It returns `false` until execution is over. In particular, when the runtime is in server mode, it always returns `false`. This function essentially iterates the locations tree (including the lib location) with the `reduction` function.

C.7 Discussion

We detail here the main differences between `CHALK` and the calculus defined in [8]. The main differences are oriented message, a richer set of values including basic datatypes and parametric processes, primitive operators over values, conditional and let expressions. Moreover, `CHALK` is a typed language.

Oriented messages The messages are oriented. A message can be sent

- to a receiver in its parent kell, it is then prefixed by `env.`
- to a local receiver, without prefix
- to a receiver in a subkell of name `a`, with a `a.` prefix.

As in the calculus, receivers are oriented and must exhibit a complementary direction so that a communication can occurs. Note that the pattern language used in `CHALK` allows a receiver listening downside to receive the name of the subkell sending the message it will react with. It is a convenient construct as it allows to encode a certain class of function calls (cf. paragraph on CPS encoding). For example, in

```
on a dn x <y> in { P } | b [ env.a <v> ]
```

Upon communication, the variable `x` will be bound to `b` in `P`.

The use of oriented messages makes the implementation more efficient as it allows direct routing of messages. However it imposes the additional condition that two sibling kells cannot have the same name, as discussed in section C.4).

Moreover, it seems to be quite natural from a programming point of view: a programmer should know where a service is defined.

Abstractions Higher-order values can be not only processes but also parametric processes.

Values In the calculus, values are names and processes. In `CHALK` the values are names, abstractions (parametric processes), integer, string, lists and values built upon these basic values and standard operators.

Bibliographie

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.

[2] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings ECOOP 2002, LNCS 2548*. Springer, 2002.

[3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. In *ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3*, pages 213–249, July 1997.

[4] R. Amadio. An asynchronous model of locality, failure, and process mobility. Technical report, INRIA Research Report RR-3109, INRIA Sophia-Antipolis, France, 1997.

[5] Martin Berger. *Towards Abstractions for Distributed Systems*. PhD thesis, Imperial College, Dept. of Computing, 2002.

[6] L. Bettini, M. Loreti, and R. Pugliese. Srtuctured nets in klaim. In *Proceedings of the 2000 ACM Symposium on Applied Computing, ACM Press*, 2000.

[7] P. Bidinger and J.B. Stefani. The Kell Calculus: Operational Semantics and Type System. In *Proc. 6th IFIP FMOODS International Conference*, volume 2884 of *LNCS*. Springer, 2003.

[8] Philippe Bidinger, Alan Schmitt, and Jean-Bernard Stefani. An abstract machine for the Kell calculus. In *7th IFIP International Conference on Formal Methods for Object-Based Distributed Systems (FMOODS)*, Athens, Greece, June 2005. Best Paper Award.

[9] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, , and K. Saikoski. The Design and Implementation of Open ORB v2. In *IEEE Distributed Systems Online Journal, vol. 2 no. 6*, November 2001.

[10] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An Open Component Model and its Support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004)*, Edinburgh, Scotland, 2004.

[11] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication Interference in Mobile Boxed Ambients. In *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FST-TCS '02)*, volume LNCS 2556. Springer, 2002.

[12] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed ambients. *Lecture Notes in Computer Science*, 2215:38–??, 2001.

[13] L. Cardelli. Wide area computation. In *Proc. Automata, Languages and Programming, 26th International Colloquium, (ICALP'99), J. Wiedermann, P. van Emde Boas, M. Nielsen (eds), Lecture Notes in Computer Science, Vol. 1644*. Springer, 1999.

[14] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures, M. Nivat (Ed.), Lecture Notes in Computer Science, Vol. 1378*. Springer Verlag, 1998.

[15] C. Carrez, A. Fantechi, and E. Najm. Behavioural contracts for a sound assembly of components. In *FORTE*, volume 2767 of *Lecture Notes in Computer Science*. Springer, 2003.

[16] G. Castagna and F. Zappa. The Seal Calculus Revisited. In *In Proceedings 22th FST-TCS*, number 2556 in LNCS. Springer, 2002.

[17] I. Castellani. Process algebras with localities. In *Handbook of Process Algebra, J. Bergstra, A. Ponse and S. Smolka (eds)*. Elsevier, 2001.

[18] L. de Alfaro and T. Henzinger. Interface Automata. In *Proceedings of the joint 8th European software engineering conference and 9th ACM SIGSOFT international symposium on the foundations of software engineering (ESEC/FSE 01)*, 2001.

[19] L. de Alfaro and T. Henzinger. Interface Thoeries for Component-Based Design. In *Proceedings of EMSOFT '01*, volume 2211 of *Lecture Notes in Computer Science*. Springer, 2001.

[20] Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Trans. Program. Lang. Syst.*, 21(1):11–45, 1999.

[21] Enterprise JavaBeansTM Specification, Version 2.1, August 2002. Sun Microsystems, `http://java.sun.com/products/ejb/`.

[22] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *USENIX Annual Technical Conference*, 2002.

[23] G. Ferrari, E. Moggi, and R. Pugliese. MetaKlaim: A Type-Safe Multi-Stage Language for Global Computing. *to appear in Mathematical Structures in Computer Science*, 2003.

[24] Fabrice Le Fessant. *JoCaml: Conception et Implantation d'un Langage à Agents Mobiles*. PhD thesis, Ecole Polytechnique, 2001.

[25] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 1997 ACM Symposium on Operating Systems Principles*, pages 38–51, St-Malo, France, October 1997.

[26] C. Fournet. *The Join-Calculus*. PhD thesis, Ecole Polytechnique, Palaiseau, France, 1998.

[27] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *In Proceedings 7th International Conference on Concurrency Theory (CONCUR '96), Lecture Notes in Computer Science 1119*. Springer Verlag, 1996.

[28] C. Fournet, J.J. Levy, and A. Schmitt. An asynchronous distributed implementation of mobile ambients. In *Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, Lecture Notes in Computer Science 1872*. Springer, 2000.

[29] Adrian Francalanza and Matthew Hennessy. A theory of system behaviour in the presence of node and link failures. In Martín Abadi and Luca de Alfaro, editors,

CONCUR, volume 3653 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2005.

[30] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. The fusion machine. In *CONCUR 2002*, volume 2421 of *LNCS*. Springer-Verlag, 2002.

[31] G.Boudol, I. Castellani, F. Germain, and M. Lacoste. Analysis of formal models of distribution and mobility: state of the art. Mikado Deliverable D1.1.1, 2002.

[32] F. Germain, M. Lacoste, and J.B. Stefani. An abstract machine for a higher-order distributed process calculus. In *Proceedings of the EACTS Workshop on Foundations of Wide Area Network Computing (F-WAN)*, July 2002.

[33] M. Hennessy, J. Rathke, and N. Yoshida. safedpi: a language for controlling mobile code. 2003.

[34] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. Technical report, Technical Report 2/98 – School of Cognitive and Computer Sciences, University of Sussex, UK, 1998.

[35] Thomas Hildebrandt, Jens Chr. Godskesen, and Mikkel Bundgaard. Bisimulation congruences for Homer — a calculus of higher order mobile embedded resources. Technical Report ITU-TR-2004-52, IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark, November 2004.

[36] D. Hirschkoff, D. Pous, and D. Sangiorgi. An Efficient Abstract Machine for Safe Ambients, 2004. Unpublished. Available at: http://www.cs.unibo.it/∼sangio/DOC_public/gcpan.ps.gz.

[37] Daniel Hirschkoff, Tom Hirschowitz, Damien Pous, Alan Schmitt, and Jean-Bernard Stefani. Component-Oriented Programming with Sharing: Containment is not Ownership. In *4th International Conference on Generative Programming and Component Engineering (GPCE)*, Tallinn, Estonia, September 2005.

[38] Daniel Hirschkoff, Tom Hirschowitz, Damien Pous, Alan Schmitt, and Jean-Bernard Stefani. Component-Oriented Programming with Sharing: Containment is not Ownership. In *4th International Conference on Generative Programming and Component Engineering (GPCE)*, Tallinn, Estonia, September 2005.

[39] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–141, New York, NY, USA, 2001. ACM Press.

[40] The Kell calculus page. http://sardes.inrialpes.fr/kells/.

[41] F. Kon, T. Yamane, K. Hess, R. H. Campbell, and M. D. Mickunas. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio, USA, January 2001.

[42] G. Leavens and M. Sitaraman (eds). *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[43] Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs. In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM'2004)*, Toronto, Canada, October 2004.

[44] E.A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3), 2004.

[45] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, 2000.

[46] L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: An Experiment in Code Mobility from the Realm of Process Calculi. In *Proceedings 5th Mobile Object Systems Workshop (MOS'99)*, 1999.

[47] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), 2000.

[48] Philippe Merle, editor. *CORBA 3.0 New Components Chapters*. OMG TC Document ptc/2001-11-03, November 2001.

[49] M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *29th ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon, 16-18 January*, 2002.

[50] R. Milner. *Communicating and mobile systems : the π-calculus*. Cambridge University Press, 1999.

[51] Oscar Nierstrasz. Piccola - a small composition language. In *ECOOP Workshops*, page 317, 1999.

[52] A. Phillips, N. Yoshida, and S. Eisenbach. A distributed abstract machine for boxed ambient calculi. In *Proceedings of ESOP 2004*, LNCS. Springer-Verlag, April 2004.

[53] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[54] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[55] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[56] D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proceedings of the 28th ICALP*, volume 2076 of *LNCS*. Springer-Verlag, 2001.

[57] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[58] A. Schmitt and J.B. Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. 2002.

[59] A. Schmitt and J.B. Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[60] A. Schmitt and J.B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In P. Quaglia, editor, *Global Computing*, volume 3267 of *LNCS*. Springer, 2004.

[61] Alan Schmitt. Safe Dynamic Binding in the Join Calculus. In *IFIP TCS'02*, Montreal, Canada, 2002.

[62] P. Sewell, P. Wojciechowski, and B. Pierce. Location-independent communication for mobile agents : a two-level architecture. Technical report, TR-462, Computer Lab, University of Cambridge, Cambridge, UK, 1998.

[63] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Box-π, wrappers and causality types. *Journal of Computer Security*, 11(2):135–188, 2003. Invited submission for a CSFW 00 special issue.

[64] J.B. Stefani. A Calculus of Kells. In *Proceedings 2nd International Workshop on Foundations of Global Computing*, 2003.

[65] D. Turner. The polymorphic π-calculus: Theory and implementation. Technical report, University of Edinburgh, GB, 1996.

[66] A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructures for Mobile Computation. In *Proceedings ACM Int. Conf. on Principles of Programming Languages (POPL)*, 2001.

[67] J. Vitek and G. Castagna. Towards a calculus of secure mobile computations. In *Proceedings Workshop on Internet Programming Languages, Chicago, Illinois, USA, Lecture Notes in Computer Science 1686, Springer*, 1998.

[68] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.

[69] N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.

[70] Nobuko Yoshida. Channel dependent types for higher-order mobile processes. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 2004. ACM Press.

[71] Silvano Dal Zilio. Mobile processes: a commented bibliography. pages 206–222, 2001.