Déterminisme et Confluence dans des systèmes concurrents et synchrones

Mehdi Dogguy

Université Paris Diderot - Paris 7

École doctorale
Sciences Mathématiques de Paris Centre

Thèse

pour l'obtention du diplôme de

Docteur de l'Université Paris Diderot

Spécialité Informatique

Déterminisme et Confluence dans des systèmes concurrents et synchrones

présentée et soutenue publiquement par

Mehdi Dogguy

le 27 janvier 2012

devant le jury composé de

M. Roberto	Amadio	*directeur*
M. Gérard	Boudol	
M. Daniel	Hirschkoff	*rapporteur*
M. Uwe	Nestmann	*rapporteur*
M. Marc	Pouzet	

Table des matières

Chapitre 1

Introduction

Les systèmes critiques sont utilisés dans plusieurs domaines industriels tels que le nucléaire et le transport. La défaillance de ces systèmes peut avoir des conséquences dramatiques au niveau financier, écologique ou humain, de part la nature des applications utilisées. Afin d'éviter les accidents, les industriels doivent respecter des contraintes strictes imposées par des autorités de certification indépendantes. Ces dernières ont la charge d'étudier la qualité des logiciels produits avant leur mise en service. Les précautions à prendre lors du développement de tels logiciels sont spécifiés dans des normes et peuvent varier suivant le domaine et la criticité de l'application. Parmis les normes mises en place, on peut citer la DO-178B (pour l'avionique), la IEC-61508 (pour les systèmes critiques) ou encore la ECSS-Q-80C (pour les systèmes spaciaux). Pour garantir la robustesse de ces systèmes, il est souvent interdit d'inclure des commandes ou fonctionnalités non prédictibles. En d'autres termes, le caractère déterministe du système est une propriété indispensable. Cela simplifie le développement de l'application et les tests de validation puisqu'un comportement fautif pourrait être reproduit à souhait.

Ces applications industrielles doivent interagir continuellement avec leur environnement et répondre le plus rapidement possible aux requêtes, tout en faisant tourner plusieurs composants en parallèle. Il est donc primordial de disposer d'outils où il est facile de gérer la concurrence, le temps et la causalité ensemble.

Les systèmes concurrents ou distribués sont souvent classés suivant deux critères principaux [LL90] : les vitesses des programmes et leur mécanisme de communication. Par rapport au premier critère, il existe plusieurs sortes de systèmes dont les systèmes synchrones, asynchrones, temps-réel, etc... En particulier, dans les systèmes synchrones, il existe une horloge globale qui orchestre le calcul. Chaque processus (ou programme) effectue une action et se synchronise avec les autres avant d'avancer davantage. Chaque étape dans le calcul est appelée *instant*, *phase* ou *tour*. Le terme désignant cette notion de temps diffère suivant les modèles. Du point de vue du calcul, tous

les programmes respectent la même notion de temps et avancent à la même vitesse.

Concernant le second critère, on peut se référer aux communications à mémoire partagée, aux appels de procédures à distance, canaux point à point, à la communication par diffusion, etc... Quelques mécanismes de communication forcent les différents acteurs à se synchroniser pour échanger une information. Cela est particulièrement vrai quand le canal de communication a une mémoire tampon de taille zéro. Cela force l'échange effectif du message avant de pouvoir commencer d'autres échanges. Une telle communication est appelée *communication synchrone*.

Dans la suite, on utilisera le terme *synchrone* pour désigner les systèmes où les programmes avancent à la même vitesse.

Les langages synchrones permettent de décrire le comportement des programmes parallèles qui sont exécutés de façon synchronisée et qui maintiennent une interaction permanente avec leur environnement [HP85]. Plusieurs approches ont été considérées pour formaliser la notion de synchronie. La principale différence entre les différentes approches est le codage de l'instant. Dans la première approche, la sémantique précise l'ensemble des actions qu'un processus doit effectuer pendant chaque instant. Cette approche a été introduite pour la première fois dans *SCCS* [Mil80] par Robin Milner en 1983. Un peu plus tard, en 1984, Didier Austry et Gérard Boudol présentent le modèle *Meije* [AB84]. Les deux modèles utilisent la même structure pour les actions mais définissent des opérateurs différents : le modèle *SCCS* a une composition parallèle *synchrone* et des opérateurs pour désynchroniser les processus alors que *Meije* dispose d'une composition parallèle asynchrone et d'opérateurs pour synchroniser les processus entre eux. Par exemple, deux processus composés de façon parallèle dans *SCCS* doivent chacun exécuter une action avant de pouvoir passer à l'instant suivant. Cela étant dit, les deux calculs sont en fait deux représentations différentes du même modèle puisque les opérateurs de l'un sont définissables en utilisant ceux de l'autre.

Malgré les propriétés mathématiques dont jouissent ces premiers modèles, ils n'ont pas été utilisés comme base d'un langage de programmation synchrone réaliste. La seconde approche est plus permissive que la première dans le sens où les interactions sont plus souples. En particulier, la condition de passage à l'instant suivant n'est plus activée par l'exécution d'une action mais par certaines conditions. L'environnement peut détecter que tous les processus sont bloqués et il peut passer à l'instant suivant. Dans ce cadre, plusieurs langages ont vu le jour et utilisent des techniques différentes où on peut distinguer principalement deux paradigmes de programmation : la programmation *flots de données* (Lustre [HCRP91], Signal [GLG87], ...) et la programmation orientée *contrôle* (Esterel [BG92], ...). Ces langages ont été conçus pour modéliser, spécifier, valider et implanter des systèmes temps-réel embarqués, et plus généralement, des systèmes critiques. Ils ont été bâti sur un modèle mathématique commun qui combine synchronie et concurrence

déterministe. Esterel a été introduit par Berry et Gontier en 1988. C'est un langage où la communication se fait à travers des signaux. Contrairement aux canaux, les signaux persistent durant un instant. Ainsi, une valeur émise sur un signal peut être reçue plusieurs fois dans l'instant. L'une des caractéristiques de ce langage est qu'il permet de tester la présence des signaux et réagir en conséquence dès l'instant courant. Ce choix de conception permet à l'utilisateur d'écrire des programmes dont la sémantique est problématique. Prenons par exemple le programme suivant :

$$\texttt{present } S \texttt{ then nothing else emit } S\,;$$

Ce programme teste la présence du signal S, et émet S s'il est absent. Et sinon, le cas où S est présent, il ne fait rien. Si on suppose que la définition du signal S est locale, alors on ne peut pas lui donner un état (présent ou absent). Si S est absent alors on émet S. Cela nous mène à une première contradiction. Si S est présent, alors il n'est pas émis, ce qui constitue la seconde contradiction. Dans Esterel, on rejette ces programmes incorrects à l'aide d'une analyse statique sur le code. Cependant, cela a motivé les chercheurs pour fournir un modèle synchrone où ce genre de programmes sont rejetés par construction. Le modèle SL [BDS96] a été introduit dans ce sens. C'est une relaxation du modèle Esterel où la présence d'un signal s'observe à la fin de l'instant uniquement.

Ces modèles de langages synchrones ont connu un succès majeur depuis leur création. Ils ont été utilisés pour la conception et l'implantation d'applications industrielles significatives dans divers domaines (services web, jeux multi-joueurs, aviation, etc...). Il existe aujourd'hui plusieurs langages de programmation ou suites de conception dont le coeur repose directement sur ces modèles (*Reactive ML* [MP05] en OCaml, *Reactive C* [Bou91] en C, *Scade* [ADS+06], *SugarCubes* [BS98] en Java, ...).

1.1 Contributions

Dans cette thèse, nous allons nous intéresser au problème du déterminisme dans des calculs concurrents synchrones avec échanges de valeurs. Principalement, nous allons étudier cette notion sous plusieurs angles de vue. La suite présente le contexte dans lequel nous avons mené notre travail et chacune des approches étudiées qui ont mené aux résultats obtenus.

Nous avons pris le Sπ-calcul [Ama07] comme base de travail. Ce calcul est un π calcul [Mil99] synchrone basé sur le modèle *SL*. La complexité du langage résultant est proche de celle du π-calcul. Il peut être considéré comme une extension du modèle SL où les signaux peuvent transporter des valeurs. De plus, les signaux sont considérés dans Sπ comme des valeurs de première classe, *i.e.* un signal peut transporter un signal. Du point de vue du temps, l'environnement fait passer le temps en utilisant une action spéciale N quand il détecte que tous les programmes ne peuvent plus avancer.

D'un point de vue de la communication, le Sπ-calcul est un calcul avec communication asynchrone. En d'autres termes, les émissions sur les signaux dans Sπ ne sont pas bloquantes. Cette idée trouve ses origines dans les travaux qui portent sur le π-calcul asynchrone [HT91, Bou92] et ceux qui ont suivis sur l'exploration de notions de bisimulations dans ce cadre là [ACS98, FG98].

Bisimulations

Dans ce contexte, nous avons cherché à développer une théorie compositionnelle de l'équivalence des programmes basée sur la notion de bisimulation. Cette notion a été largement étudiée dans le cadre de CCS [Mil89], et ensuite étendue au π-calcul [DW01, Mil99].

Dans [Ama07], on définit les premières bisimulations pour Sπ et on montre qu'elles ont de bonnes propriétés mathématiques. En particulier, on introduit un système de transitions étiquetées pour le Sπ-calcul avec la notion de bisimulation étiquetée associée. Ensuite, on présente une notion de bisimulation contextuelle basée sur celle qui a été introduite auparavant pour le π-calcul [HY95]. Dans le cadre de Sπ, l'auteur montre que la bisimulation étiquetée coïncide avec la bisimulation contextuelle et qu'elle a de bonnes propriétés de congruence.

Cependant, la bisimulation étiquetée présentée pour Sπ est différente de la bisimulation standard du π-calcul, où le traitement est uniforme pour toutes les actions. Nous avons donc proposé une nouvelle sémantique pour Sπ basée sur une notion standard de bisimulation sur un nouveau système de transition (non standard). Nous avons montrer que cette bisimulation était préservée par les contextes statiques (voir théorème 16). Ensuite, il s'agissait de montrer qu'elle coïncidait avec la bisimulation contextuelle de Sπ (théorème 25) qui a été introduite dans [Ama07].

La migration vers la nouvelle sémantique ne s'est pas faite en une étape. Nous sommes passés par des systèmes de transitions intermédiaires. À chaque étape, nous avons changé judicieusement quelques règles de transition et proposé une bisimulation associée. En montrant l'équivalence entres les différents systèmes, on arrive à mettre un lien entre la sémantique originelle et la nouvelle sémantique que nous proposons. Ce travail représente la première contribution apportée dans [AD07].

Confluence et Déterminisme

Dans [Mil89], la théorie de la réécriture a été généralisée en ajoutant des étiquettes aux réécritures. Cela a permis de revoir les concepts de déterminisme et de confluence, en particulier, dans le cadre de CCS. Par la suite, ces notions ont été étendues au π-calcul [PW97].

Dans cette partie, nous avons cherché à étudier les notions de déterminisme et de confluence dans le cadre du Sπ-calcul. Dans le contexte du π-calcul, on sait que la confluence implique déterminisme. Cependant, ces deux notions sont équivalentes dans le Sπ-calcul (théorème 35(1)).

Sous hypothèse de réactivité, on arrive à montrer que la confluence locale suffit pour garantir la confluence (théorème 35(2)).

Il est à noter que supposer la réactivité des programmes dans un contexte synchrone est assez habituel puisque cela évite, par exemple, les boucles instantanées. Plus généralement, l'hypothèse de réactivité assure que tous les instants terminent.

Confluence par typage pour Sπ

L'analyse du déterminisme faite dans la partie précédente, où la communication se fait par signaux, suggère l'existence, principalement, de deux situations problématiques :

- Au moins deux valeurs distinctes v_1 et v_2 sont envoyées sur le même signal dans l'instant.
- Au moins deux valeurs sont disponibles à la fin de l'instant.

Du point de vue du déterminisme, il paraît raisonnable d'interdire la première situation et d'accepter la seconde sous conditions. Dans ce dernier cas, il faudrait que le processus léger qui utilise la liste des valeurs reçues à la fin de l'instant ait un comportement qui ne dépende pas de l'ordre des valeurs dans la liste.

Dans cette partie, nous avons cherché à identifier le fragment déterministe de Sπ. Pour ce faire, nous avons mis au point une analyse statique capable d'identifier les programmes ayant la propriété recherchée. Notre solution sera basée sur un nouveau système de types conçu pour Sπ. S'agissant d'une propriété critique, il paraît pertinent de fournir une garantie statique de correction des programmes. Le caractère concurrent des programmes étudiés rend l'analyse encore plus pertinente car leur comportement est plus complexe que celui des programmes séquentiels.

Il existe bon nombre de systèmes de types conçus pour analyser diverses propriétés de programmes concurrents. Parmi ces systèmes de types, on peut citer [PS96] où les types servent à imposer un mode d'utilisation (en entrée ou en sortie) aux canaux ; [KPT96] où on utilise des types linéaires pour limiter le nombre d'utilisations d'un canal ; [Kob98] pour éviter les situations d'inter-bloquage ; [BC01] pour vérifier la propriété de non interférence entre processus ou encore [KSW06, HR02] pour la gestion des ressources.

Dans notre cas présent, l'analyse faite dans notre système de types pour le déterminisme est inspirée, entre autres, par la logique linéaire [Gir87], et le typage linéaire de programmes fonctionnels [Wad93] et des ressources linéaires sur les canaux [KPT96]. Dans un premier temps, nous allons définir, dans la section 2.5.1, une notion d'*usage affine* pour les signaux qui indique

leur mode d'utilisation en émission, réception et réception à la fin de l'instant. Parmi tous les modes de communication exprimables avec les usages, nous allons en identifier 5 qui garantissent le déterminisme.

En se basant sur la notion d'*usage*, nous allons définir un système de types pour le Sπ-calcul. Dans ce contexte, on montrera que tout processus P typable sera déterministe (théorème 52). Ce résultat sera prouvé en utilisant la notion d'équivalence sur les processus présentée précédemment. En particulier, nous allons définir une notion de bisimulation typée basée sur la bisimulation étiquetée qui va nous permettre d'explorer les comportements des programmes qui satisfont les critères de typage. Cela permet de simplifier l'analyse en éliminant bon nombre de comportements que l'on sait pas intéressant dans notre contexte typé. Ce travail a été publié dans [AD08].

Confluence dans un calcul synchrone avec canaux

Dans l'émission d'un signal, il y a une récursion cachée qui permet au signal de persister durant l'instant. Ce comportement est spécifique aux signaux. Il permet une communication par diffusion où un message envoyé une fois peut être reçu par plusieurs destinataires. Les canaux ont un comportement plus basique où la communication se fait point à point. Un message envoyé sur un canal ne sera effectivement reçu qu'au plus une fois. Le problème de déterminisme existe aussi dans des calculs utilisant les canaux comme mécanisme de communication.

La propriété de confluence a déjà été largement étudiée dans le cadre du π-calcul où des systèmes de types, garantissant cette propriété, ont vu le jour. La confluence étant une propriété technique impliquant le déterminisme. Cependant, ces travaux se sont intéressés à des communications point-à-point [NS97] ou des calculs qui n'ont aucune notion du temps [KPT96]. Il existe bien des calculs dotés d'une notion de temps et à base de canaux (comme *SCCS* [Mil80], *Meije* [AB84], TCCS [Ama09]). Cependant, ces modèles utilisent des canaux purs (ne transportant pas de valeurs) comme moyen de communication.

Dans cette partie, nous allons présenter un nouveau calcul de processus synchrone, nommé Tapis. Ce calcul est une extension synchrone du π-calcul, et il est inspiré par *TCCS*. Naturellement, les canaux sont considérés comme des valeurs de première classe dans Tapis et peuvent être envoyés comme message sur d'autres canaux. Comme dans Sπ, la nature des communications dans Tapis est asynchrone (*i.e.*, non bloquante). À l'aide des moyens de communication de Tapis, nous allons montrer comment simuler, dans une certaine mesure, la communication par signaux avec des canaux. Et nous comparerons ce nouveau calcul au modèle *SL* et à Sπ.

Dans le cadre de Tapis, nous allons chercher à concevoir un système de types capable d'identifier les processus confluents en nous inspirant de ce que nous avons fait précédemment pour Sπ. Nous allons proposer une notion

d'*usage* adaptée aux canaux. Puis, nous allons présenter un système de types garantissant la confluence (théorème 64), et par conséquent, le déterminisme aussi.

Formalisation en Coq

Pour parvenir à montrer la confluence des programmes typés dans Tapis, le lemme principal dont nous avons besoin est la préservation du typage par réduction (lemme 60). La preuve de ce lemme, en présence de canaux riches et d'usages, est étonnement technique. Cela a motivé la formalisation complète de la preuve dans l'assistant de preuve Coq [CDT10].

Divers efforts de formalisation dans différents assistants à la preuve ont été déployés. Parmi ces travaux, on peut citer le travail de formalisation du π-calcul dans l'assistant à la preuve HOL [Mel94] ou la formalisation du π-calcul en Coq [Hir97] où les variables sont représentées à l'aide d'indices de De Bruijn. Il existe aussi des travaux de formalisation d'un système de types simple pour le π-calcul [HG99, Des00].

Dans notre travail de formalisation, nous avons choisi d'adopter l'approche "*locally nameless*" [ACP+08] pour la représentation des variables où les variables liées sont représentées avec des indices de De Bruijn et les variables libres avec des noms. Cela nous permet d'éviter plusieurs problèmes liés à l'alpha-conversion puisque, avec cette approche, des termes alpha-équivalents auront des représentations identiques. Techniquement, ce travail de formalisation a soulevé des problèmes de codage intéressants pour la relation d'équivalence pour les processus et pour la représentation efficace des additions partielles sur les usages. Une partie importante de cette contribution utilise des "types classes" dans le sens de [SO08] pour factoriser des interfaces et alléger les notations. Cela nous permet, par exemple, d'utiliser qu'une seule notation définie pour plusieurs types différents. Nous avons aussi utilisé la bibliothèque Coq AAC [BP11] pour simplifier les preuves de réécritures modulo associativité et commutativité.

La réalisation de ce travail a nécessité 1 année/homme et a aboutit sur un projet de 12k lignes de code. La durée s'explique par le fait que nous avons testé plusieurs représentations pour les termes, les additions et les règles de types avant de trouver celle qui nous convenait le plus (au niveau de l'efficacité et l'aisance au niveau de la manipulation des structures). Cette contribution est disponible en ligne [DG11] et est distribué sous la licence libre CeCILL 2.0 [cec06].

1.2 Plan

Le manuscrit est divisé en deux chapitres "Signaux" et "Canaux".

Dans le chapitre 2, nous présentons le Sπ-calcul et nous le comparons à son ancêtre le π-calcul en mettant l'accent sur les différences entre les deux

calculs (section 2.1). Nous présentons la sémantique du Sπ-calcul qui est basée sur un système de transition étiquetée dans la section 2.2. Ensuite, nous introduisons une notion de bisimulation pour Sπ (section 2.3) et montrons qu'elle coïncide avec la bisimulation contextuelle pour Sπ. Dans la section 2.4, nous étudions les notions de déterminisme et confluence et montrons que les deux notions coïncident dans Sπ. Enfin, dans la section 2.5, nous présentons un système de types basé sur une notion d'*usages* pour les signaux et montrons que tout programme typable est déterministe.

Dans le chapitre 3, nous changeons de contexte et introduisons un nouveau calcul synchrone, TAPIS, utilisant des canaux *asynchrones* comme mécanisme de communication. Nous présentons sa sémantique dans 3.2 et nous le comparons au modèle *SL* (dans 3.3.1) et Sπ (dans 3.3.2). Ensuite, nous présentons un système de types qui garantit la confluence (section 3.4). Enfin, nous présentons la formalisation de ce système de types en Coq (section 3.5) où nous avons formalisé la preuve de la préservation du typage par réduction.

Chapitre 2

Signaux

2.1 Introduction du Sπ-calcul

Le Sπ-calcul est une variante synchrone du π-calcul qui est basée sur le modèle SL[BDS96] où la réaction à l'absence d'un signal se fait à la fin de l'instant. Contrairement à SL, les signaux du Sπ-calcul peuvent transporter des valeurs. En particulier, les signaux peuvent transporter des signaux. Ce sont des valeurs de première classe. Dans la section 2.2, nous allons d'abord présenter la syntaxe des programmes du Sπ-calcul et étudier leur sémantique. Ensuite, nous nous intéresserons à la notion de déterminisme et étudierons la relation de ce dernier avec la confluence dans 2.4. Enfin, nous fournirons un système de typage qui permet d'identifier les programmes déterministes dans la section 2.5.

Syntaxe Syntaxiquement, les termes du Sπ-calcul se divisent en plusieurs catégories qui sont décrites dans le tableau 2.1. Les noms de signaux peuvent désigner à la fois les constantes de signaux telles que celles générées par l'opérateur ν ainsi que les variables de signaux comme celles utilisées par l'opérateur de test de présence. Les variables issues de la classe syntaxique *Var* incluent la classe *Sig* et les variables d'autres types. Les valeurs (*Val*) sont des termes construits à partir des constructeurs (*Cnst*) et les noms de signaux. Les filtres (*Pat*) sont construits à partir des constructeurs et les variables. Les expressions (*Exp*) sont des termes construits à partir de filtres, fonctions et variables. Finalement, les (*Rexp*) représentent des expressions pouvant contenir des valeurs associées à des signaux à la fin de l'instant. Ces valeurs particulières sont notées $!s$ pour un nom de signal s donné. Intuitivement, c'est une liste de valeurs représentant l'ensemble des valeurs émises sur le signal considéré pendant l'instant précédent. La notation utilisée ici rappelle celle de la fonction qui lit la valeur d'une référence en ML.

Signaux
$Sig \quad ::= s \mid t \mid \cdots$
Variables
$Var \quad ::= Sig \mid x \mid y \mid z \mid \cdots$
Constructeurs
$Cnst \quad ::= * \mid \mathsf{nil} \mid \mathsf{cons} \mid \mathsf{c} \mid \mathsf{d} \mid \cdots$
Valeurs $v, v', \ldots$
$Val \quad ::= Sig \mid Cnst(Val, \ldots, Val)$
Motifs $u, u', \ldots$
$Pat \quad ::= Cnst(Var, \ldots, Var)$
Symboles de fonctions de premier ordre
$Fun \quad ::= f \mid g \mid \cdots$
Expressions $e, e', \ldots$
$Exp \quad ::= Var \mid Cnst(Exp, \ldots, Exp) \mid Fun(Exp, \ldots, Exp)$
Expressions avec déréférenciation $r, r', \ldots$
$Rexp \quad ::= !Sig \mid Var \mid Cnst(Rexp, \ldots, Rexp)$
$\qquad \mid Fun(Rexp, \ldots, Rexp)$

TABLE 2.1 – Catégories syntaxiques des expressions

La grammaire des processus dans Sπ est comme suit :

$$\begin{array}{rcl} P & ::= & 0 \\ & \mid & A(\mathbf{e}) \\ & \mid & \overline{s}e \\ & \mid & s(x).P, K \\ & \mid & [s_1 = s_2]P_1, P_2 \\ & \mid & [v \trianglerighteq u]P_1, P_2 \\ & \mid & P_1 \mid P_2 \\ & \mid & \nu s\ P \\ K & ::= & A(\mathbf{r}) \end{array}$$

Sémantique informelle Nous utiliserons la notation $\mathbf{m}$ pour désigner un vecteur $m_1, \ldots, m_n$ de taille $n \geq 0$. Le processus 0 est le programme terminé, qui ne fait rien. L'appel d'un processus léger dont l'identifiant est A et les paramètres sont $\mathbf{e}$ se note $A(\mathbf{e})$. Le processus léger A est défini avec une équation unique du type $A(\mathbf{x}) = P$ où chaque variable libre de P apparaît dans $\mathbf{x}$. Le processus $\overline{s}e$ évalue l'expression e et émet sa valeur sur le signal s. Le processus $s(x).P, K$ est le mécanisme fondamental de réception tiré du modèle SL. Il teste la présence du signal s dans l'instant courant. Si les valeurs $v_i, \ldots, v_n$ ont été émises sur le signal s alors $s(x).P, K$ évoluera de façon non déterministe en $[v_i/x]P$ (c'est à dire, le processus P où toutes les occurrences de la variable libre x ont été remplacées par v_i). Le cas échéant, si aucune valeur n'a été émise sur le signal en question durant l'instant, la

continuation K est évaluée à la fin de l'instant. Le processus $[s_1 = s_2]P_1, P_2$ est la fonction de test usuelle du π-calcul qui exécute P_1 quand s_1 et s_2 sont identiques, et P_2 dans le cas échéant. Notez que dans cette construction, s_1 et s_2 sont des noms libres. $[v \trianglerighteq u]P_1, P_2$ est l'opérateur de filtrage. Si la valeur v est compatible avec le motif u, alors θP_1 est exécuté où θ est une substitution calculée par $match(v, u) = \theta$. Si v ne correspond pas au motif u, alors il n'existe aucune substitution θ. Dans ce cas, $match(u, p) = \uparrow$ et le calcul continue en exécutant le processus P_2. Nous supposerons que v est une variable ou une valeur, et que p est de la forme $\mathsf{c}(\mathbf{x})$, où c est un constructeur et $\mathbf{x}$ un vecteur de variables distinctes. Si u est une variable x, nous ferons l'hypothèse supplémentaire que x n'apparaît pas comme une variable libre dans P_1. Le processus $\nu s\ P$ crée un nouveau signal s et exécute P. La construction $(P_1 \mid P_2)$ exécute P_1 et P_2 de façon parallèle.

Une continuation K est l'appel d'un processus léger (potentiellement) récursif où ses arguments sont des expressions ou des valeurs associées à des signaux à la fin de l'instant. Cette partie sera spécifiquement mentionnée en détails dans la partie traitant des expressions.

Nous détaillerons formellement la sémantique des processus de Sπ plus tard, dans la section 2.2.

Dans la suite, nous désignerons l'ensemble des noms de signaux libres dans l'entité p par $fn(p)$ où p peut être un processus ou un filtre et l'ensemble des noms de signaux liés par $bn(p)$. Par ailleurs, nous supposerons l'existence d'une fonction d'évaluation $\Downarrow$ qui est définie de telle façon que, pour toute fonction f et une liste de valeurs $v_1, \ldots, v_n$, il n'existe qu'une valeur unique v telle que $f(v_1, \ldots, v_n) \Downarrow v$ et $fn(v) \subseteq \bigcup_{i=1,\ldots,n} fn(v_i)$. La fonction d'évaluation $\Downarrow$ est étendue aux expressions où toutes les variables libres sont des noms de signaux.

2.1.1 Comparaison avec le π-calcul

La syntaxe du Sπ-calcul est proche de celle du π-calcul. Toutefois, il y a des différences importantes au niveau de la sémantique qui doivent être soulignées. Nous allons illustrer ces différences dans l'exemple suivant.

Supposons que l'on dispose de deux valeurs distinctes v_1 et v_2. Étudions le processus Sπ suivant :

$$\begin{array}{lll} P = \nu\ s_1, s_2 & (\overline{s_1}v_1 \mid \overline{s_1}v_2 & \\ & \mid s_1(x).\ (s_1(y).\ (s_2(z).\ A(x,y)\ \underline{,B(!s_1)}) & \\ & \qquad\qquad \underline{,0}) & \\ & \quad \underline{,0}) & \\ &) & \end{array}$$

Si on oublie la présence des continuations (éléments soulignés) et que l'on considère s_1 et s_2 comme des canaux, alors P peut être considéré comme un

processus du π-calcul. Dans ce cas, P se réduira vers P_1 :

$$P_1 = \nu s_1, s_2\ (s_2(z).A(\theta(x), \theta(y)))$$

où θ est une substitution telle que $\theta(x), \theta(y) \in \{v_1, v_2\}$ et $\theta(x) \neq \theta(y)$. En π-calcul, le processus P_1 est bloqué et ne peut plus évoluer car il a besoin d'une synchronisation sur s_2 qui n'arrivera pas.

En Sπ-calcul, les émissions sur les signaux persistent durant l'instant et P se réduit à P_2 :

$$P_2 = \nu s_1, s_2\ (\overline{s_1}v_1 \mid \overline{s_1}v_2 \mid (s_2(z).A(\theta(x), \theta(y)), \underline{B(!s_1)}))$$

où $\theta(x), \theta(y) \in \{v_1, v_2\}$.

L'environnement détecte que P_1 ne peut plus évoluer et cela marque la fin de l'instant. Ensuite, une action supplémentaire représentée par la relation $\xrightarrow{N}$ permet de faire évoluer P_2 vers l'instant suivant :

$$P_2 \xrightarrow{N} P_2' = \nu s_1, s_2\ B(v)$$

où $v \in \{[v_1; v_2], [v_2; v_1]\}$. Ainsi, à la fin de l'instant, un signal qui a été déréférencé en utilisant $!s_1$ se transforme en une liste de valeurs distinctes émises sur s_1 pendant l'instant précédent, et toutes les autres émissions sont remises à zéro.

2.1.2 Un exemple programmatique

Nous utiliserons la notation $\mathsf{pause}.K$ comme raccourci syntaxique pour le processus $\nu s\ s(x).0, K$ où s n'est pas une variable libre dans K. Ce processus attend la fin de l'instant pour exécuter K.

Nous introduisons un exemple programmatique pour illustrer les interactions synchrones possibles dans le Sπ-calcul. Dans cet exemple, il s'agit d'un *serveur* traitant une liste de requêtes qui ont été émises pendant l'instant précédent sur le signal s. Pour chaque requête de la forme $\mathsf{req}(s', x)$, le serveur émet sur le signal s' une réponse qui est une fonction de x.

$$\begin{array}{lcl} \mathit{Server}(s) & = & \mathsf{pause}.\mathit{Handle}(s, !s) \\ \mathit{Handle}(s, \ell) & = & [\ell \trianglerighteq \mathsf{req}(s', x) :: \ell'](\overline{s'}f(x) \mid \mathit{Handle}(s, \ell')), \mathit{Server}(s)\ . \end{array}$$

Pour communiquer avec le serveur $\mathit{Server}(s)$, il est indispensable d'implanter un client compatible qui émettrait des requêtes x sur le signal s. Un tel client pourrait ressembler au processus suivant :

$$\mathit{Client}(x, s, t) \quad = \quad \nu s'\ (\overline{s}\mathsf{req}(s', x) \mid \mathsf{pause}.s'(x).\overline{t}x, 0)\ .$$

En utilisant cette implantation du client, le résultat est final sera envoyé sur le canal t. Notez que l'implantation présentée du client est particulière puisqu'elle s'attend à une réponse du serveur dans l'instant suivant, puisque la continuation est nulle. Autrement, aucun traitement ne sera effectué dans les instants qui suivent.

2.2 Sémantique opérationnelle

Nous allons définir la sémantique opérationnelle du Sπ-calcul. Cette section se divise en deux parties. La première 2.2.1 présente les actions que nous allons utiliser comme étiquettes dans la relation de transition. Quant à la seconde 2.2.2, elle décrit effectivement le système de transition conçu pour le Sπ-calcul. Certains choix ont été faits lors de la conception du système de transition qui peuvent paraître pas très naturels. Ils seront justifiés dans la section 2.3 quand on abordera les bisimulations.

2.2.1 Actions

Comme le montre le tableau ci-dessous, les actions sont catégorisées. L'utilité de chaque catégorie est expliquée dans la suite.

act	$::= \alpha \mid aux$	(actions)
α	$::= \tau \mid \nu\mathbf{t}\ \overline{s}v \mid sv \mid N$	(actions principales)
aux	$::= s?v \mid (E, V)$	(actions secondaires)
μ	$::= \tau \mid \nu\mathbf{t}\ \overline{s}v \mid s?v$	(actions imbriquées)

Les *actions principales* seront utilisées directement par la relation de bisimulation. L'ensemble de ces actions comporte : l'action interne τ, l'action d'émission $\nu\mathbf{t}\ \overline{s}v$ (où l'on suppose que les noms de signaux dans $\mathbf{t}$ sont distincts, apparaissent dans v et sont différents de s), l'action de réception sv, et l'action N (dont le nom fait référence au mot $Next$) qui marque la fin de l'instant courant et fait évoluer le processus vers l'instant suivant. Nous utiliserons la notation $\nu\mathbf{t}, \mathbf{t}'\ \overline{s}v$ pour l'action où l'ensemble des noms liés est égal à $\mathbf{t} \cup \mathbf{t}'$.

Les *actions secondaires* sont l'action de réception $s?v$ qui donne lieu à une action τ lorsqu'elle est couplée à une émission et l'action (E, V) qui sert à produire les actions N de passage de l'instant qui intervient seulement lorsque le processus ne peut plus produire aucune action interne τ. À la fin de l'instant, elle a un rôle important qui consiste à collecter l'ensemble des valeurs émises sur chaque signal dans des listes, ré-initialiser tous les signaux et déclencher la continuation K pour chaque processus de la forme $s(x).P, K$.

Enfin, les *actions imbriquées* $\mu, \mu', \ldots$ sont des actions *principales* ou *secondaires*, peuvent être produites par un sous-processus, et doivent être propagées vers le niveau supérieur.

Pour mieux formaliser les opérations de calcul de fin de l'instant que l'on va voir dans 2.2.2, nous allons introduire quelques notations. Considérons E une fonction qui associe un ensemble fini de valeurs à un nom de signal. Nous noterons $\emptyset$ la fonction qui associe un ensemble vide à chaque nom de signal, et $[M/s]$ celle qui associe l'ensemble M au signal s et l'ensemble vide pour tous les autres signaux. Nous disposerons également d'une opération d'union qui est définie point à point et qui étend celle définie sur les ensembles.

Nous représenterons un ensemble de valeurs à l'aide de la liste des valeurs contenues dans l'ensemble en question. Nous noterons $v \Vdash\!- M$ quand v *représente* M. Cela est vrai lorsque $M = \{v_1, \ldots, v_n\}$ et $v = [v_{\pi(1)}; \ldots; v_{\pi(n)}]$ pour une permutation π sur l'ensemble $\{1, \ldots, n\}$. Cela impose certaines conditions sur la liste des valeurs contenues dans v. En particulier, aucune valeur ne peut y apparaître plus d'une fois même si elle a effectivement été émise dans l'instant à plusieurs reprises. Supposons que V est une fonction qui associe une liste de valeurs à un nom de signal. Nous écrirons $V \Vdash\!- E$ quand $V(s) \Vdash\!- E(s)$ pour chaque nom de signal s. Nous noterons $dom(V)$ l'ensemble $\{s | V(s) \neq []\}$. Si K est une continuation, alors $V(K)$ est K après avoir remplacé chaque $!s$ par sa valeur $V(s)$. Nous dénotons par $V[l/s]$ la fonction qui associe l à s et $V(s')$ pour tout autre nom de signal s'.

En utilisant les conventions précédemment définies, nous pouvons établir la relation $P \xrightarrow{(E,V)} P'$. Intuitivement, cela représente plusieurs faits :

- P est suspendu et ne peut plus évoluer,
- P a émis exactement les valeurs spécifiées par E,
- et le comportement P dans l'instant suivant sera identique à celui du processus P' et dépendra des valeurs stockées dans V.

Il faut noter que le calcul, telle que sa sémantique sera définie, ne fait pas la différence entre les deux processus suivants :

$$\overline{s}v \mid \overline{s}v \quad \text{et} \quad \overline{s}v$$

L'émission d'un signal persiste dans l'instant. Une valeur émise peut donc être reçue plus d'une fois. Il y a une récursion cachée dans l'émission. Considèrons le processus P suivant $P = s(x).Q, 0 \mid s(x).R, 0$. Quelque soit le nombre de fois où la valeur v est émise, le processus $P \mid \overline{s}v \mid \cdots \mid \overline{s}v$ se réduira vers $Q[v/x] \mid R[v/x] \mid \overline{s}v \mid \cdots \mid \overline{s}v$. Comme nous pourrons le remarquer dans la partie suivante, cela est vrai aussi quand il n'y a qu'une seule émission sur le signal s de la valeur v. Si toutefois l'utilisateur désire envoyer une valeur v plusieurs fois, il existe un moyen d'y arriver. Il suffit de s'assurer de l'unicité de la valeur transmise à l'aide du constructeur ν. Ainsi, on peut imaginer le processus suivant :

$$\nu y\ \overline{s} < v, y > | \ \nu y\ \overline{s} < v, y >$$

où <x,y> est un constructeur du type de données "couple". Dans cet exemple, le signal y est différent d'une branche à l'autre.

2.2.2 Système de transitions étiquetées

Nous supposerons que les règles de transition s'appliquent uniquement aux processus dont les variables libres sont des noms de signaux et utiliserons les conventions standard pour le renommage des variables liées. Le système de transition que l'on propose est comme suit :

Communication La règle (*out*) modélise le fait que les émissions persistent au cours de l'instant. La règle (*in*) est utile pour la continuation et enregistre le fait que l'environnement a émis $\overline{s}v$. Enfin, la règle (in_{aux}) est la réception standard telle que connue dans le π-calcul.

$$(out)\ \frac{e \Downarrow v}{\overline{s}e \xrightarrow{\overline{s}v} \overline{s}e} \qquad (in_{aux})\ \frac{}{s(x).P, K \xrightarrow{s?v} [v/x]P}$$

$$(in)\ \frac{}{P \xrightarrow{\overline{s}v} (P \mid \overline{s}v)}$$

Appel récursif Chaque processus léger est défini au moyen d'une équation unique. La règle (*rec*) vérifie que cette équation existe bien. L'effet de cette règle est de remplacer l'appel au processus léger A par le processus P où les variables libres $\mathbf{x}$ ont été remplacées par $\mathbf{v}$, le résultat de l'évaluation de l'expression $\mathbf{e}$.

$$(rec)\ \frac{A(\mathbf{x}) = P \quad \mathbf{e} \Downarrow \mathbf{v}}{A(\mathbf{e}) \xrightarrow{\tau} [\mathbf{v}/\mathbf{x}]P}$$

Règles de filtrage Le premier groupe (première ligne) décrit les règles de tests d'égalité sur les signaux. Le deuxième groupe conçerne le *pattern-matching* sur des valeurs inductives. Leur comportement a été expliqué informellement au début de ce chapitre.

$$(=_1^{sig})\ \frac{}{[s = s]P_1, P_2 \xrightarrow{\tau} P_1} \qquad (=_2^{sig})\ \frac{s_1 \neq s_2}{[s_1 = s_2]P_1, P_2 \xrightarrow{\tau} P_2}$$

$$(=_1^{ind})\ \frac{match(v, u) = \theta}{[v \trianglerighteq u]P_1, P_2 \xrightarrow{\tau} \theta P_1} \qquad (=_1^{ind})\ \frac{match(v, u) = \uparrow}{[v \trianglerighteq u]P_1, P_2 \xrightarrow{\tau} P_2}$$

Composition parallèle et synchronisation Deux processus composés de façon parallèle peuvent interagir de façon directe ou indirecte. La règle (*synch*) modélise une interaction directe où les deux processus s'échangent des valeurs sur le canal s. La règle (*comp*) quant à elle sert à composer deux processus où l'un effectue un calcul tandis que l'autre reste immobile.

$$(comp)\ \frac{P_1 \xrightarrow{\mu} P_1' \quad bn(\mu) \cap fn(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2}$$

$$(synch)\ \frac{P_1 \xrightarrow{\nu \mathbf{t}\ \overline{s}v} P_1' \quad P_2 \xrightarrow{s?v} P_2' \quad \{\mathbf{t}\} \cap fn(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\tau} \nu \mathbf{t}\ (P_1' \mid P_2')}$$

Gestion des noms frais Les noms frais générés grâce à l'opérateur ν peuvent être introduits sous certaines conditions qui sont décrites par les règles ci-dessous. Ici, la fonction $n(\mu)$ renvoie l'ensemble des noms liés dans l'action μ.

$$(\nu)\ \frac{P \xrightarrow{\mu} P' \quad t \notin n(\mu)}{\nu t\ P \xrightarrow{\mu} \nu t\ P'} \qquad (\nu_{ex})\ \frac{P \xrightarrow{\nu \mathbf{t}\ \overline{s}v} P' \quad \mathbf{t}' \neq s \quad \mathbf{t}' \in n(v)\backslash\{\mathbf{t}\}}{\nu t'\ P \xrightarrow{(\nu \mathbf{t}', \mathbf{t})\overline{s}v} P'}$$

Règles de fin de l'instant Ces règles sont spécifiques au Sπ-calcul et décrivent sous quelles conditions le passage de l'instant est effectué. Les émissions sont remises à zéro et les valeurs émises dans l'instant sont collectées.

$$(0)\ \frac{}{0 \xrightarrow{\emptyset, V} 0} \qquad (reset)\ \frac{e \Downarrow v \quad v \text{ apparaît dans } V(s)}{\overline{s}e \xrightarrow{[\{v\}/s], V} 0}$$

$$(cont)\ \frac{s \notin dom(V)}{s(x).P, K \xrightarrow{\emptyset, V} V(K)} \qquad (par)\ \frac{P_i \xrightarrow{E_i, V} P_i' \quad i = 1, 2}{(P_1 \mid P_2) \xrightarrow{E_1 \cup E_2, V} (P_1' \mid P_2')}$$

$$(next)\ \frac{P \succeq \nu \mathbf{s}\ P' \quad P' \xrightarrow{E, V} P'' \quad V \Vdash E}{P \xrightarrow{N} \nu \mathbf{s}\ P''}$$

Afin de simplifier le système de transition, nous utilisons la relation $\succeq$ pour déplacer les utilisations de ν en tête dans le processus. En d'autres termes, on écrit $P \succeq Q$ quand on arrive à obtenir Q de P en transformant récursivement tous les sous-processus $R \mid \nu s S$ en $\nu s(R \mid S)$ où $s \notin fn(R)$. Sans cette transformation, nous serions obligés de rajouter une sixième règle pour le traitement de la fin de l'instant pour gérer le constructeur ν de façon isolée. Par ailleurs, nous utiliserons la notation $P \xrightarrow{\alpha} \cdot$ comme raccourci pour $\exists P', P \xrightarrow{\alpha} P'$. Si on considère le processus $P = \overline{s}v_1 \mid \overline{s}v_2 \mid s'(x).0, A(!s)$ alors le traitement de fin de l'instant correspondra à cette dérivation :

$$\frac{\dfrac{\overline{s}v_1 \xrightarrow{E_1 = [\{v_1\}/s], V} 0 \quad \overline{s}v_2 \xrightarrow{E_2 = [\{v_2\}/s], V} 0 \quad s'(x).0, A(!s) \xrightarrow{\emptyset, V} A(V(s))}{P \xrightarrow{E = E_1 \cup E_2, V = [[v_1; v_2]/s]} 0 \mid 0 \mid A([v_1; v_2]) \quad V \Vdash E}}{\nu s' P \xrightarrow{N} \nu s'(0 \mid 0 \mid A([v_1; v_2]))}$$

À présent que le système de transition est énoncé, nous allons pouvoir définir la notion de *dérivé* et de *réactivité*.

Définition 1 (Dérivé). *Un processus Q est dit* dérivé *du processus P si :*

$$P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q \text{ où } n \geq 0$$

Remarque 2. *Un processus peut être son propre dérivé.*

Définition 3 (Réactivité)**.** *Un processus P est dit* réactif *si pour chaque dérivé Q, toute séquence de réductions avec l'action τ termine.*

Un processus est donc dit *réactif* si on sait qu'il va exécuter, pendant chaque cycle, l'action spéciale N. Son calcul ne reste donc pas bloqué dans un instant particulier.

Définition 4 (Suspension)**.**

$$\begin{array}{lll} P\downarrow & si \quad \neg(\ P \xrightarrow{\tau} \cdot\ ) & \textit{(suspension)} \\ P\Downarrow & si \quad \exists P'\ (\ P \stackrel{\tau}{\Rightarrow} P' \textit{ et } P'\downarrow\ ) & \textit{(suspension faible)} \\ P\Downarrow_L & si \quad \exists P'\ (\ P \mid P'\ )\Downarrow & \textit{(L-suspension)} \end{array}$$

Clairement, un processus qui est *suspendu* est aussi *suspendu faiblement* et *L-suspendu*. Cependant, le contraire n'est pas vrai. D'autre part, tous les programmes dérivés d'un processus *réactif* ont la propriété de *suspension faible*.

Exemple 5. *La différence entre la suspension et la suspension faible est claire. Le second permet au processus d'effectuer, par exemple, quelques synchronisations avant de se suspendre. Si on considère le processus P suivant : $P = \overline{s}v \mid s(x).0, 0$ alors on peut constater que $P \not\Downarrow$ mais que $P \Downarrow$ en prenant $P' = \overline{s}v \mid 0$.*

Pour la L-suspension*, il faut penser à d'autres comportements. Par exemple, prenons un processus P qui n'est pas réactif :*

$$P = A(s) \mid \overline{s}s \textit{ où } A(x) = x(y).A(y), 0$$

Le processus P n'est pas réactif car il va toujours avoir la possibilité de synchroniser sur le signal s pendant l'instant. Ainsi, la séquence de τ réductions ne terminera pas. Donc, $P \not\Downarrow$. Cependant, il est possible que son calcul termine un jour si on place un processus P' bien choisi en parallèle. En particulier, exécuter $P' = \overline{s}s'$ (pour tout s' différent de s) en parallèle à P permettra à ce dernier de trouver une porte de sortie. Après quelques réductions, $P \mid P'$ pourra se réduire en $Q = s'(y).A(y), 0|\overline{s}s \mid \overline{s}s'$. Q est bien suspendu puisqu'il n'y a plus aucun moyen d'effectuer des synchronisations.

Définition 6 (Engagement)**.** *Nous noterons $P \searrow \overline{s}$ quand $P \xrightarrow{\nu \mathbf{t}\ \overline{s}v} \cdot$ et dirons que P s'engage à émettre sur s, durant l'instant courant.*

2.3 Bisimulations

Afin d'étudier le comportement des processus, nous avons besoin de définir une relation d'équivalence sur les processus où, intuitivement, des processus équivalents ont le même comportement. À première vue, il pourrait

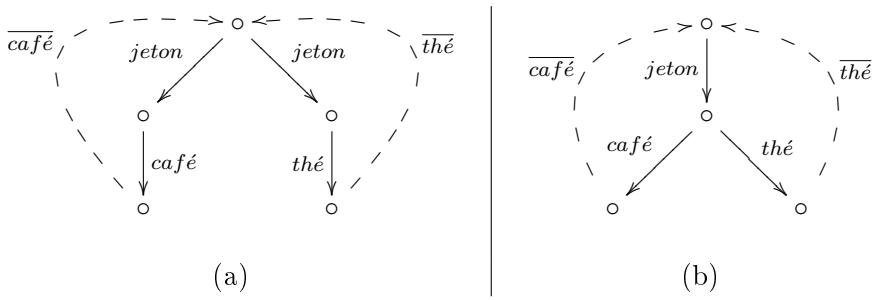


FIGURE 2.1 – Exemple de la machine à café

être raisonnable de considérer une relation d'équivalence basée sur l'ensemble des traces d'exécution des processus [Hoa85] où une trace serait définie comme une séquence d'actions effectuées lors de l'exécution. Cependant, il est assez facile de produire des exemples de processus ayant des comportements différents, mais produisant des traces identiques. Il semblerait que l'exemple canonique soit celui de la "machine à café". Les deux machines dont le comportement a été schématisé dans la figure 2.1 sont équivalentes du point de vue des traces. Cependant, dès la première étape, la machine *(a)* est contrainte de faire un choix qui va déterminer la boisson à servir alors que la machine *(b)* laisse l'utilisateur le choix de la boisson après avoir inséré un jeton.

Dans le cadre présent, nous avons besoin d'une notion d'équivalence sémantique pour les programmes qui soit plus fine que celle définie sur la base des traces. Nous allons nous baser sur la notion de bisimulation qui a été développée dans le cadre de CCS [Mil89], puis dans le cadre du π-calcul[Mil99, DW01] et qui est plus intéressante que l'équivalence sur les traces [BIM95, JS90]. Dans cette section, nous allons chercher à développer cette notion dans le cadre du Sπ-calcul et nous montrerons que la bisimulation étiquetée qui sera définie dans la suite est préservée par les contextes statiques. Enfin, nous allons montrer que notre bisimulation étiquetée coïncide avec la bisimulation contextuelle qui a été introduite dans [Ama07] pour Sπ. Cette dernière a été inspirée par la même notion de bisimulation contextuelle introduite pour le π-calcul dans [HY95].

Afin de pouvoir introduire la relation de bisimulation, nous avons besoin de définir une notion de transition étiquetée faible. La transition $\overset{\alpha}{\Rightarrow}$ est définie comme suit :

$$\overset{\alpha}{\Rightarrow} = \begin{cases} (\overset{\tau}{\rightarrow})^* & \text{si } \alpha = \tau \\ (\overset{\tau}{\Rightarrow}) \circ (\overset{N}{\rightarrow}) & \text{si } \alpha = N \\ (\overset{\tau}{\Rightarrow}) \circ (\overset{\alpha}{\rightarrow}) \circ (\overset{\tau}{\Rightarrow}) & \text{sinon} \end{cases}$$

Cette définition de transition faible est proche de la définition standard.

Toutefois, les deux définitions diffèrent sur la transition $\xrightarrow{N}$. Celle que l'on propose ici ne permet pas l'ajout de réductions internes (en utilisant l'action τ) après une réduction de fin de l'instant (*i.e.* $\xrightarrow{N}$).

Ce changement peut paraître anodin mais il est assez important comme nous le montre l'exemple 7.

Exemple 7 (Transition faible à la fin de l'instant)**.** *Prenons les processus suivants :*

$$P = \mathsf{pause}.(\overline{s'}1)$$
$$Q = \nu s(\mathsf{pause}.A(!s) \mid \overline{s}1)$$

où $A(x) = [x \trianglerighteq [1]]\overline{s'}1, 0$ *où* 1 *est une constante numérique.*

Nous voudrions distinguer P *et* Q*. Dans* P*, l'émission sur le signal* s' *est exécutée durant l'instant suivant, alors que dans le processus* P*, la valeur est envoyée d'abord sur le signal* s *et ensuite sur le signal* s' *pendant l'instant suivant.*

Si $\overset{N}{\Rightarrow}$ *était défini en tant que* $\overset{\tau}{\Rightarrow} \circ \xrightarrow{N} \circ \overset{\tau}{\Rightarrow}$*, alors* P *et* Q *auraient un comportement équivalent.*

Définition 8 (Bisimulation étiquetée faible)**.** *Une relation symétrique* $\mathcal{R}$ *sur les processus est une bisimulation étiquetée si et seulement si*

$$\frac{P \mathcal{R} Q \quad P \xrightarrow{\alpha} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{\exists Q' \ (\ Q \overset{\alpha}{\Rightarrow} Q' \qquad P' \mathcal{R} Q'\ )}$$

Nous dénotons par $\approx$ *la plus grande bisimulation étiquetée.*

Il est aussi possible de considérer une relation de bisimulation utilisant une transition faible pour la première réduction. Une telle bisimulation serait définie de la manière suivante :

Définition 9 (Bisimulation étiquetée faible (variante))**.** *Une relation symétrique* $\mathcal{R}$ *sur les processus est une bisimulation étiquetée faible (notée* w-bisimulation*) si et seulement si*

$$\frac{P \mathcal{R} Q \quad P \overset{\alpha}{\Rightarrow} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{\exists Q' \ (\ Q \overset{\alpha}{\Rightarrow} Q' \qquad P' \mathcal{R} Q'\ )}$$

Nous dénotons par $\approx_w$ *la plus grande bisimulation étiquetée faible.*

Il est intéressant de remarquer que les deux bisimulations $\approx$ et $\approx_w$ coïncident. Cela est prouvé par le lemme 10.

Lemme 10.

1. $\approx$ *est une relation d'équivalence.*
2. *Les relations* $\approx$ *et* $\approx_w$ *coïncident.*

PREUVE.

1. L'identité est une bisimulation et l'union de deux relations symétriques est une relation symétrique. La transitivité est vérifiée en montrant que $\approx \circ \approx$ est une bisimulation. Pour cela, il suffit de prendre 3 processus P, Q, et R tels que $P \approx Q \approx R$ en supposant que $\approx \circ \approx$ est une bisimulation. En se référant à la définition de $\approx$ et en vérifiant que leurs dérivés sont effectivement en relation. On en déduit donc la transitivité de $\approx$.
2. Par définition, une *w-bisimulation* est une bisimulation étiquetée. Pour montrer l'autre sens de l'énoncé, il suffit de prouver que $\approx$ est une *w-bisimulation*. Cela se fait de façon directe en analysant des diagrammes de réductions.

□

Pour la suite, il est utile de définir une *bisimulation up to bisimulation*.

Définition 11 (w-Bisimulation up to w-bisimulation)**.** *Une relation symétrique $\mathcal{R}$ sur les processus est une w-bisimulation up to w-bisimulation si et seulement si*

$$\frac{P \,\mathcal{R}\, Q \quad P \stackrel{\alpha}{\Rightarrow} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{\exists Q' \;\; (\; Q \stackrel{\alpha}{\Rightarrow} Q' \qquad P' \approx_w \circ\, \mathcal{R} \,\circ \approx_w Q' \;)}$$

Nous dénotons par $\approx'_w$ la plus grande de ces relations.

En fait, cette *w-bisimulation up to w-bisimulation* est incluse dans une bisimulation.

Lemme 12. *Si $\mathcal{R}$ est une* w-bisimulation up to w-bisimulation*, alors $\mathcal{R} \subseteq \approx_w$.*

PREUVE. En utilisant le lemme 10, on a que $\approx_w$ est une relation d'équivalence. En particulier, elle est transitive. Ensuite, en supposant que $\mathcal{R}$ est une *w-bisimulation up to w-bisimulation*, il suffit de vérifier que la relation $\approx_w \circ \mathcal{R} \circ \approx_w$ est une w-bisimulation. □

Une propriété importante de la bisimulation étiquetée est la préservation de la relation de bisimulation par les contextes.

Définition 13 (Contexte)**.** *Un* contexte statique *C est défini comme suit :*

$$C ::= [\,] \mid C \mid P \mid \nu s \; C$$

Il est assez fréquent de faire une analyse des transitions dans les preuves de divers lemmes utilisés dans cette section. Pour simplifier ces preuves là, il nous semble justifié d'introduire une relation d'équivalence structurelle $\equiv$ pour les processus.

Définition 14 (Équivalence structurelle)**.** *La relation d'équivalence* $\equiv$ *est définie sur les processus telle que :*

1. $\equiv$ *contient l'alpha conversion,*
2. $\equiv$ *est préservée par les contextes,*
3. *la composition parallèle est associative et commutative,*
4. *si* $s \notin fn(Q)$ *alors* $Q \mid \nu s\ P \equiv \nu s(Q \mid P)$,
5. $\overline{s}v \mid \overline{s}v \equiv \overline{s}v$,
6. $\overline{s}e \equiv \overline{s}v$ *si* $e \Downarrow v$.

Pour le système de transition considéré, on peut vérifier que des processus équivalents génèrent exactement les mêmes transitions, et se réduisent vers des processus équivalents à leur tour. Nous utiliserons cette hypothèse dans la suite.

À présent, on voudrait prouver que la bisimulation est préservée par les contextes. Il nous faut d'abord montrer qu'elle l'est pour certains processus engendrés par quelques règles de transitions.

Lemme 15.

1. *Si* $P_1 \approx P_2$ *alors* $(P_1 \mid \overline{s}v) \approx (P_2 \mid \overline{s}v)$.
2. *Si* $P_1 \approx P_2$ *alors* $\nu s\ P_1 \approx \nu s\ P_2$ *et* $(P_1 \mid Q) \approx (P_2 \mid Q)$.

Preuve.

1. Prenons la relation $\mathcal{R}'$ telle que $\mathcal{R}' = \{((P \mid \overline{s}v), (Q \mid \overline{s}v)) \mid P \approx Q\}$ et la relation $\mathcal{R}$ telle que $\mathcal{R} = \mathcal{R}' \cup \approx$. Nous allons montrer que $\mathcal{R}$ est une bisimulation. Supposons que $(P \mid \overline{s}v) \xrightarrow{\alpha} \cdot$ et $P \approx Q$. Il y a essentiellement deux cas intéressants à considérer :

   $(\alpha = \tau)$ Dans ce cas, $(P \mid \overline{s}v) \xrightarrow{\tau} (P' \mid \overline{s}v)$ et $P \xrightarrow{s?v} P'$. Par définition du système de transition, nous savons que $P \xrightarrow{sv} (P \mid \overline{s}v) \xrightarrow{\tau} (P' \mid \overline{s}v)$. Par définition de la bisimulation, $Q \overset{sv}{\Rightarrow} (Q'' \mid \overline{s}v) \overset{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$ et $(P' \mid \overline{s}v) \approx (Q' \mid \overline{s}v)$. Ainsi, nous pouvons conclure car $(Q \mid \overline{s}v) \overset{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$.

   $(\alpha = N)$ Dans ce cas, $(P \mid \overline{s}v) \xrightarrow{N} P'$. D'après le système de transition, nous avons $P \xrightarrow{sv} (P \mid \overline{s}v)$. L'hypothèse initiale nous aide à déduire :
   - $Q \overset{sv}{\Rightarrow} (Q'' \mid \overline{s}v) \overset{\tau}{\Rightarrow} (Q''' \mid \overline{s}v) \xrightarrow{N} Q'$,
   - $(P \mid \overline{s}v) \approx (Q'' \mid \overline{s}v) \approx (Q''' \mid \overline{s}v)$,
   - et $P' \approx Q'$.

   Alors $(Q \mid \overline{s}v) \overset{N}{\Rightarrow} Q'$.

2. Nous souhaitons montrer que

$$\mathcal{R} = \{(\nu \mathbf{t}\ (P_1 \mid Q), \nu \mathbf{t}\ (P_2 \mid Q)) \mid P_1 \approx P_2\} \cup \approx$$

   est une bisimulation étiquetée *up to* la relation d'équivalence structurelle $\equiv$.

(τ) Supposons que $\nu\mathbf{t}\ (P_1 \mid Q) \xrightarrow{\tau} \cdot$. Cela peut arriver si P_1 ou Q exécute l'action τ, ou bien quand P_1 et Q synchronisent. Nous devons analyser les différents sous cas :

(a) Supposons que $Q \xrightarrow{\tau} Q'$, alors $\nu\mathbf{t}\ (P_2 \mid Q) \xrightarrow{\tau} \nu\mathbf{t}\ (P_2 \mid Q')$.

(b) Si $P_1 \xrightarrow{\tau} P_1'$ alors $P_2 \stackrel{\tau}{\Rightarrow} P_2'$ et $P_1' \approx P_2'$. Donc $\nu\mathbf{t}\ (P_2 \mid Q) \stackrel{\tau}{\Rightarrow} \nu\mathbf{t}\ (P_2' \mid Q)$.

(c) Si $P_1 \xrightarrow{s?v} P_1'$ et $Q \xrightarrow{\nu\mathbf{t}'\ \overline{s}v} Q'$. Cela voudrait dire que $Q \equiv \nu\mathbf{t}'\ (\overline{s}v \mid Q'')$ et $Q' \equiv (\overline{s}v \mid Q'')$. En utilisant le lemme 15(1), on a $(P_1 \mid \overline{s}v) \approx (P_2 \mid \overline{s}v)$. D'autre part, $(P_1 \mid \overline{s}v) \xrightarrow{\tau} (P_1' \mid \overline{s}v)$. Donc, $(P_2 \mid \overline{s}v) \stackrel{\tau}{\Rightarrow} (P_2' \mid \overline{s}v)$ et $(P_1' \mid \overline{s}v) \approx (P_2' \mid \overline{s}v)$. Ensuite, nous remarquons que la transition $\nu\mathbf{t}\ (P_1 \mid Q) \xrightarrow{\tau} \cdot \equiv \nu\mathbf{t}, \mathbf{t}'\ ((P_1' \mid \overline{s}v) \mid Q'')$ correspond à $\nu\mathbf{t}\ (P_2 \mid Q) \xrightarrow{\tau} \cdot \equiv \nu\mathbf{t}, \mathbf{t}'\ ((P_2' \mid \overline{s}v) \mid Q'')$.

(d) Supposons que $P_1 \xrightarrow{\nu\mathbf{t}'\ \overline{s}v} P_1'$ et $Q \xrightarrow{s?v} Q'$. Alors $P_2 \stackrel{\nu\mathbf{t}'\ \overline{s}v}{\Rightarrow} P_2'$ et $P_1' \approx P_2'$. Nous pouvons conclure en remarquant que $\nu\mathbf{t}\ (P_2 \mid Q) \stackrel{\tau}{\Rightarrow} \nu\mathbf{t}, \mathbf{t}'\ (P_2' \mid Q')$.

(out) Supposons $\nu\mathbf{t}\ (P_1 \mid Q) \xrightarrow{\nu\mathbf{t}'\ \overline{s}v} \cdot$ et $\mathbf{t} = \mathbf{t_1}, \mathbf{t_2}$ et $\mathbf{t}' = \mathbf{t_1}, \mathbf{t_3}$ tels que l'émission cache exactement les noms $\mathbf{t}_1$ parmi les noms dans $\mathbf{t}$. Nous devons considérer deux cas suivant le processus qui effectue l'action.

(a) Si $Q \xrightarrow{\nu\mathbf{t_3}\ \overline{s}v} Q'$ alors $\nu\mathbf{t}\ (P_2 \mid Q) \xrightarrow{\nu\mathbf{t}'\ \overline{s}v} \nu\mathbf{t_2}\ (P_2 \mid Q')$, ce qui nous permet de conclure.

(b) Si $P_1 \xrightarrow{\nu\mathbf{t_3}\ \overline{s}v} P_1'$ alors $P_2 \stackrel{\nu\mathbf{t_3}\ \overline{s}v}{\Rightarrow} P_2'$ et $P_1' \approx P_2'$. D'où $\nu\mathbf{t}\ (P_2 \mid Q) \stackrel{\nu\mathbf{t}'\ \overline{s}v}{\Rightarrow} \nu\mathbf{t_2}\ (P_2' \mid Q)$, ce qui nous permet de conclure.

(in) Pour ce cas, il est suffisant de remarquer que, modulo renommage, $\nu\mathbf{t}\ (P_i \mid Q) \mid \overline{s}v \equiv \nu\mathbf{t}\ ((P_i \mid \overline{s}v) \mid Q)$ et rappeler qu'en utilisant le lemme 15(1), on a $(P_1 \mid \overline{s}v) \approx (P_2 \mid \overline{s}v)$.

(N) Supposons que $\nu\mathbf{t}\ (P_1 \mid Q) \downarrow$. Modulo équivalence structurelle, nous pouvons exprimer Q en termes d'émissions et de réceptions. En d'autres termes, Q pourrait ressembler à $\nu\mathbf{t}_Q\ (S_Q \mid I_Q)$ où S_Q est la composition parallèle des émissions et I_Q celle des réceptions. Ainsi, nous avons : $\nu\mathbf{t}\ (P_1 \mid Q) \equiv \nu\mathbf{t}, \mathbf{t}_Q\ (P_1 \mid S_Q \mid I_Q)$, et $\nu\mathbf{t}\ (P_2 \mid Q) \equiv \nu\mathbf{t}, \mathbf{t}_Q\ (P_2 \mid S_Q \mid I_Q)$ en supposant que $\{\mathbf{t}_Q\} \cap fn(P_i) = \emptyset$ pour $i = 1, 2$.

Si $\nu\mathbf{t}\ (P_1 \mid Q) \xrightarrow{N} P$ alors $P \equiv \nu\mathbf{t}, \mathbf{t}_Q\ (P_1'' \mid Q')$ où, en particulier, on a $(P_1 \mid S_Q) \downarrow$ et $(P_1 \mid S_Q) \xrightarrow{N} (P_1' \mid 0)$.

En supposant que $P_1 \approx P_2$, et en utilisant la définition de la bisimulation, nous en déduisons que :

(a) $(P_2 \mid S_Q) \stackrel{\tau}{\Rightarrow} (P_2'' \mid S_Q)$;

(b) $(P_2'' \mid S_Q)\downarrow$;

(c) $(P_2'' \mid S_Q) \xrightarrow{N} (P_2' \mid 0)$;

(d) $(P_1 \mid S_Q) \approx (P_2'' \mid S_Q)$;

(e) $(P_1' \mid 0) \approx (P_2' \mid 0)$.

Puisque $(P_1 \mid S_Q)$ et $(P_2'' \mid S_Q)$ sont suspendus et bisimilaires, les deux processus doivent s'engager (cf. définition 6) sur le même signal et, pour chaque signal, ils doivent émettre le même ensemble de valeurs (modulo renommage des noms liés). Il s'en suit que le processus $\nu\mathbf{t},\mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q)$ est suspendu. La seule possibilité restante pour exécuter une action interne est que l'émission dans P_2'' active une réception dans I_Q. Mais, cela contredit l'hypothèse que $\nu\mathbf{t},\mathbf{t}_Q\ (P_1 \mid S_Q \mid I_Q)$ est suspendu. De plus, $(P_2'' \mid S_Q \mid I_Q) \xrightarrow{N} (P_2' \mid 0 \mid Q')$.
Nous avons donc

$$\nu\mathbf{t}\ (P_2 \mid Q) \equiv \nu\mathbf{t},\mathbf{t}_Q\ (P_2 \mid S_Q \mid I_Q) \stackrel{\tau}{\Rightarrow} \nu\mathbf{t},\mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q),$$

$\nu\mathbf{t},\mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q)\downarrow$, et $\nu\mathbf{t},\mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q) \xrightarrow{N} \nu\mathbf{t},\mathbf{t}_Q\ (P_2' \mid 0 \mid Q')$. Maintenant $\nu\mathbf{t},\mathbf{t}_Q\ (P_1 \mid S_Q \mid I_Q)\ \mathcal{R}\ \nu\mathbf{t},\mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q)$ parce que $(P_1 \mid S_Q) \approx (P_2'' \mid S_Q)$ et $\nu\mathbf{t},\mathbf{t}_Q\ (P_1' \mid Q')\ \mathcal{R}\ \nu\mathbf{t},\mathbf{t}_Q\ (P_2' \mid Q')$ parce que $P_1' \approx P_2'$.

□

Théorème 16 (Compositionnalité)**.** *Si $P \approx Q$ et C est un contexte statique alors $C[P] \approx C[Q]$.*

Preuve. Conséquence directe du lemme 15. □

2.3.1 Caractérisation de la bisimulation

Lors de l'élaboration du système de transition, des choix techniques ont été faits qui peuvent sembler pas très naturels à première vue. Étudier des caractérisations alternatives pour la bisimulation introduite peut être un moyen de justifier ces choix là. Pour parvenir à notre but, nous allons utiliser la notion de bisimulation *contextuelle* présentée dans [Ama07] pour Sπ dont nous rappelons la définition.

Définition 17 (Bisimulation barbelée)**.** *Une relation symétrique $\mathcal{R}$ définie sur les processus est une bisimulation* barbelée *si pour tout P et Q tels que $P\ \mathcal{R}\ Q$, on a :*

B1 *Si $P \xrightarrow{\tau} P'$ alors $\exists Q'$ $(Q \stackrel{\tau}{\Rightarrow} Q'$ et $P'\ \mathcal{R}\ Q')$.*

B2 *Si $P \searrow \overline{s}$ et $P \Downarrow_L$ alors $\exists Q'$ $(Q \stackrel{\tau}{\Rightarrow} Q', Q' \searrow \overline{s},$ et $P\ \mathcal{R}\ Q')$.*

B3 *Si $P \xrightarrow{N} P''$ alors $\exists Q', Q''$ $(Q \stackrel{\tau}{\Rightarrow} Q', Q'\downarrow, P\ \mathcal{R}\ Q', Q' \xrightarrow{N} Q'',$ et $P''\ \mathcal{R}\ Q'')$.*

Systèmes de transition étiquetée		Bisimulations	
$(\xrightarrow{\alpha}_1)$	Règle (in_{aux}) remplacée par $(in^1_{aux})\dfrac{}{s(x).P, K \xrightarrow{s?v} [v/x]P \mid \overline{s}v}$	$(\approx_1)$	Comme dans la définition 9
$(\xrightarrow{\alpha}_2)$	Règle (in) supprimée et action $s?v$ remplacée par sv	$(\approx_2)$	Comme dans la définition 9 quand $\alpha \neq sv$. Sinon : $(Inp)\dfrac{P \mathcal{R} Q}{(P \mid \overline{s}v) \mathcal{R} (Q \mid \overline{s}v)}$
$(\xrightarrow{\alpha}_2)$		$(\approx_3)$	Comme dans la définition 9 quand $\alpha \neq sv$ et en remplaçant la règle (Inp) par : $\dfrac{P \mathcal{R} Q \qquad P \xrightarrow{sv}_2 P'}{\begin{array}{l}\exists Q',\ (\ Q \stackrel{\overline{s}v}{\Rightarrow}_2 Q' \wedge P' \mathcal{R} Q') \\ \vee \quad (Q \stackrel{\tau}{\Rightarrow}_2 Q' \wedge P' \mathcal{R} (Q' \mid \overline{s}v))\end{array}}$ Pour l'action $\alpha = N$, il faut requérir : $\dfrac{\begin{array}{c}P \mathcal{R} Q,\ (P \mid S) \xrightarrow{N} P' \\ S = \overline{s}_1 v_1 \mid \cdots \mid \overline{s}_n v_n\end{array}}{\begin{array}{c}\exists Q', Q''\ (\ (Q \mid S) \stackrel{\tau}{\Rightarrow}_2 Q'' \\ \wedge (P \mid S) \mathcal{R} Q'' \\ \wedge Q'' \xrightarrow{N}_2 Q' \quad P' \mathcal{R} Q'\ )\end{array}}$

TABLE 2.2 – Formulations équivalentes de la bisimulation étiquetée

Nous dénotons par $\approx_B$ la bisimulation barbelée *la plus large.*

Définition 18 (Bisimulation contextuelle)**.** *Une relation symétrique $\mathcal{R}$ définie sur les processus est une bisimulation* contextuelle *si elle est une bisimulation barbelée (conditions $(\mathbf{B1-3})$) et vérifie également la condition suivante : pour tous P et Q tels que $P \mathcal{R} Q$ on a*

C1 *$C[P] \mathcal{R} C[Q]$, pour tout contexte statique C.*

Nous dénotons par $\approx_C$ la bisimulation contextuelle la plus large.

Afin de caractériser la notion de bisimulation, nous modifions incrémentalement le système de transition et la relation de bisimulation associée. On aboutit ainsi à 3 caractérisations équivalentes de la notion de bisimulation. Les différents systèmes et relations que l'on va utiliser dans la suite sont définis dans le tableau 2.2.

Lemme 19. *La bisimulation $\approx$ coïncide avec la bisimulation $\approx_1$.*

PREUVE. La seule différence entre les deux systèmes de transition est le contenu de la règle (in_{aux}). Les autres règles sont identiques.

La règle (in_{aux}) produit $s?v$ qui est une action auxiliaire utilisée pour produire une action τ grâce à la règle $(sync)$.

Prenons l'exemple suivant pour montrer la différence entre les deux systèmes considérés. Supposons que $P = \overline{s}e \mid s(x).Q, K$ et $e \Downarrow v$. Alors :

$$P \xrightarrow{\tau} \overline{s}e \mid [v/x]Q = P' \text{ et } P \xrightarrow{\tau}_1 \overline{s}e \mid ([v/x]Q \mid \overline{s}v) = P'' \ .$$

En Sπ-calcul, nous ne distinguons pas entre les situations où une valeur a été émise plusieurs fois ou précisément une seule fois dans l'instant. En particulier, P' et P'' sont structurellement équivalents (cf. définition 14). □

À présent, nous allons nous concentrer sur les relations entre les systèmes de transition $\xrightarrow{act}_1$ et $\xrightarrow{act}_2$. Dans $\xrightarrow{act}_2$, la règle (in) a été supprimée et la règle (in_{aux}) a été modifiée de telle sorte qu'elle utilise l'action sv au lieu de l'action auxiliaire $s?v$. Remarquez donc que l'action auxiliaire $s?v$ n'est plus utilisée dans le système de transition $\xrightarrow{act}_2$.

Lemme 20.

1. *Si* $P \xrightarrow{act}_1 P'$ *et* $act \neq sv$, *alors* $P \xrightarrow{act'}_2 P'$ *et :*
   - *Si* $act = s?v$ *alors* $act' = sv$
   - *Sinon,* $act' = act$.
2. *Si* $P \xrightarrow{act}_2 P'$ *alors* $P \xrightarrow{act'}_1 P'$ *et :*
   - *Si* $act = sv$ *alors* $act' = s?v$
   - *Sinon,* $act' = act$.

PREUVE. La preuve de ce lemme est directe en inspectant la définition des deux systèmes de transition et en se rappelant de la définition de l'équivalence structurelle. □

Lemme 21. *Si* $P \approx_1 Q$ *alors* $(P \mid \overline{s}v) \approx_1 (Q \mid \overline{s}v)$.

PREUVE. La preuve de ce lemme est sensiblement similaire à celle du lemme 15(1). □

Lemme 22. *La bisimulation* $\approx_1$ *coïncide avec la bisimulation* $\approx_2$.

PREUVE.

($\approx_1 \subseteq \approx_2$) Si $\alpha = sv$ alors nous utilisons le lemme 21. Sinon, $P \approx_1 Q$ et $P \xrightarrow{\alpha}_2 P'$. En utilisant le lemme 20(2), $P \xrightarrow{\alpha}_1 P'$. Par définition de $\approx_1$, il existe un processus Q' tel que $Q \stackrel{\alpha}{\Rightarrow}_1 Q'$ et $P' \approx_1 Q'$. En utilisant le lemme 20(1), on arrive à montrer que $Q \stackrel{\alpha}{\Rightarrow}_1 Q'$.

($\approx_2 \subseteq \approx_1$) Si $\alpha = sv$ et $P \xrightarrow{sv}_1 (P \mid \overline{s}v)$ alors, par définition du système de transition, $Q \xrightarrow{\alpha}_1 (Q \mid \overline{s}v)$. Par définition de $\approx_2$, $(P \mid \overline{s}v) \approx_2 (Q \mid \overline{s}v)$.

Sinon, *i.e* cas où $\alpha \neq sv$, $P \approx_2 Q$ et $P \xrightarrow{\alpha}_1 P'$. En utilisant le lemme 20(1), on obtient que $P \xrightarrow{\alpha}_2 P'$. En se référant à la définition de $\approx_2$, on sait qu'il existe un Q' tel que $Q \overset{\alpha}{\Rightarrow}_2 Q'$ et $P' \approx_2 Q$. En utilisant le lemme 20(2), on arrive à déduire que $Q \overset{\alpha}{\Rightarrow}_1 Q'$.

□

Avant de pouvoir conclure, il reste à comparer $\approx_2$ et $\approx_3$. Il est utile de rappeler que ces deux bisimulations utilisent le même système de transition $\xrightarrow{\cdot}_2$. Avant de montrer la relation entre ces deux bisimulations, il convient de prouver ces quelques faits qui vont nous aider dans la suite.

Lemme 23.

1. *Si* $P \approx_2 Q$ *et* $P \xrightarrow{N} P'$ *alors* $\exists Q', Q''$ ($Q \overset{\tau}{\Rightarrow}_2 Q'', Q'' \xrightarrow{N} Q', P \approx_2 Q'', P' \approx_2 Q'$).
2. *Si* $P \approx_3 Q$ *alors* $(P \mid \overline{s}v) \approx_3 (Q \mid \overline{s}v)$.

PREUVE.

1. Si $P \xrightarrow{N} P'$ alors P ne peut pas exécuter des actions τ. Ainsi, si $P \approx_2 Q$ et $Q \overset{\tau}{\Rightarrow}_2 Q''$ alors nécessairement $P \approx_2 Q''$.
2. Nous réutilisons la structure de la preuve du lemme 15(1). Prenons $\mathcal{R}' = \{((P \mid \overline{s}v), (Q \mid \overline{s}v)) \mid P \approx_3 Q\}$ et $\mathcal{R} = \mathcal{R}' \cup \approx_3$. Nous voulons montrer que $\mathcal{R}$ est une 3-bisimulation ($\approx_3$). Supposons que $(P \mid \overline{s}v) \xrightarrow{\alpha}_1 \cdot$ et $P \approx_3 Q$. Il y a principalement deux cas à considérer :

   ($\alpha = \tau$) Supposons que $(P \mid \overline{s}v) \xrightarrow{\tau}_2 (P' \mid \overline{s}v)$ car $P \xrightarrow{sv}_2 P'$. Par définition de $\approx_3$, il y a deux possibilités :

   (a) $Q \overset{sv}{\Rightarrow}_2 Q'$ et $P' \approx_3 Q'$

   (b) ou bien $P' \approx_3 (Q' \mid \overline{s}v)$.

   Dans le cas 2a, $(Q \mid \overline{s}v) \overset{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$ et nous pouvons remarquer que $((P' \mid \overline{s}v), (Q' \mid \overline{s}v)) \in \mathcal{R}$. Dans le cas 2b, $(Q \mid \overline{s}v) \overset{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$ et nous remarquons que $(P' \mid \overline{s}v, (Q' \mid \overline{s}v) \mid \overline{s}v) \in \mathcal{R}$ et $(Q' \mid \overline{s}v) \mid \overline{s}v \equiv (Q' \mid \overline{s}v)$.

   ($\alpha = N$) Supposons que $((P \mid \overline{s}v) \mid S) \xrightarrow{N} P'$. Nous pouvons conclure en utilisant la définition de $\approx_3$, et en choisissant $S' = (\overline{s}v \mid S)$ $(Q \mid S') \overset{\tau}{\Rightarrow} Q'' \xrightarrow{N} Q'$, $(P \mid S') \approx_3 Q''$ et $P' \approx_3 Q'$.

□

Lemme 24. *La bisimulation* $\approx_2$ *coïncide avec la bisimulation* $\approx_3$.

PREUVE.

($\approx_2 \subseteq \approx_3$) Étudions d'abord les conditions présentes sur les transitions qui utilisent des actions de réception. Supposons $P \approx_2 Q$ et $P \xrightarrow{sv}_2 P'$. Par définition de $\approx_2$, $(P \mid \overline{s}v) \approx_2 (Q \mid \overline{s}v)$. De plus, $(P \mid \overline{s}v) \xrightarrow{\tau}_2 (P' \mid \overline{s}v) \equiv P'$. Par définition de $\approx_2$, $(Q \mid \overline{s}v) \stackrel{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$ et $P' \equiv (P' \mid \overline{s}v) \approx_2 (Q' \mid \overline{s}v)$. Deux cas peuvent exister :

1. Si $Q \stackrel{sv}{\Rightarrow} Q'$ alors $Q' \mid \overline{s}v \equiv Q'$. Cela satisfait le premier cas pour la 3-bisimulation.
2. Si $Q \stackrel{\tau}{\Rightarrow} Q'$ alors, modulo équivalence structurelle, nous pouvons satisfaire le second cas.

Considérons maintenant le cas des conditions sur les transitions à la fin de l'instant. Supposons $P \approx_2 Q$, $S = \overline{s_1}v_1 \mid \cdots \mid \overline{s_n}v_n$ et $(P \mid S) \xrightarrow{N}_2 P'$. Par la condition (*Inp*), $(P \mid S) \approx_2 (Q \mid S)$. Alors, par le lemme 23(1), la condition requise par la 3-bisimulation est satisfaite par la condition correspondante dans la 2-bisimulation en l'appliquant à $(P \mid S)$ et $(Q \mid S)$.

($\approx_3 \subseteq \approx_2$) La condition (*Inp*) est satisfaite grâce au lemme 23(2). La condition de la 2-bisimulation à la fin de l'instant est un cas spécial de celle de la 3-bisimulation où S est vide.

□

Théorème 25 (caractérisation de la bisimulation étiquetée)**.** *Si P et Q sont des processus réactifs, alors $P \approx Q$ si et seulement si $P \approx_C Q$.*

PREUVE. La preuve de ce théorème se fait en suivant quelques étapes. Ces étapes ont été explicitées dans les lemmes précédemment énoncés et prouvés dans cette section. Dans [Ama07], la bisimulation contextuelle est caractérisée en tant que variante de la bisimulation $\approx_3$ où les conditions sur les émissions sont formulées comme suit :

$$\frac{P \mathrel{\mathcal{R}} Q \qquad P \Downarrow_L \qquad P \xrightarrow{\nu \mathbf{t}\ \overline{s}v}_2 P' \qquad \{\mathbf{t}\} \cap fn(Q) = \emptyset}{Q \stackrel{\nu \mathbf{t}\ \overline{s}v}{\Rightarrow}_2 Q' \quad P' \mathrel{\mathcal{R}} Q'}$$

Clairement, si P est un processus réactif alors $P \Downarrow_L$. Notez que la définition de la réactivité utilise le système de transition où il est possible d'effectuer $P \xrightarrow{sv} (P \mid \overline{s}v)$. Donc, si P est réactif, alors $(P \mid \overline{s}v)$ l'est tout autant. De plus, si l'on commence à comparer deux processus réactifs alors tous ceux considérés par la bisimulation le seront aussi. Cela veut dire que tous les processus réactifs P satisfont la propriété $P \Downarrow_L$. Ainsi, la bisimulation $\approx_3$ coïncide avec la bisimulation considérée dans [Ama07].

Nous pouvons donc conclure, puisque nous avons trouvé une chaîne qui relie ces bisimulations. □

Il est important de remarquer que la bisimulation étiquetée ne coïncide pas totalement avec la bisimulation contextuelle quand il s'agit de comparer des processus qui ne sont pas réactifs. Par exemple, la bisimulation contextuelle identifie tous les processus qui ne sont pas L-suspendus. Par exemple, si on considère le processus $P = s(x).A(x), 0$ où $A(x) = x(y).A(y), 0$ alors on a bien $P \approx 0$ mais $P \not\approx_C 0$.

2.4 Déterminisme et Confluence

Les notions de déterminisme et confluence ont été étudiées pour CCS dans [Mil89] où la théorie de la réécriture a été généralisée dans deux directions en utilisant des étiquettes sur les opérations de réecriture et en vérifiant la commutation des diagrammes modulo une équivalence sémantique bien choisie. Puis, ces notions ont été étendues au π-calcul [PW97]. À présent, nous cherchons à développer une théorie de la bisimulation dans le cadre du Sπ-calcul en suivant ce qui a été fait pour CCS et le π-calcul. Dans Sπ, il s'avère que les deux notions de déterminisme et de confluence coïncident. De plus, pour les processus réactifs, la propriété de confluence locale est suffisante pour garantir le déterminisme.

Nous dénoterons par ϵ la séquence vide et par $s = \alpha_1 \dots \alpha_n$ la séquence finie (possiblement vide) contenant des actions différentes de τ. Nous définissons la relation suivante :

$$\stackrel{s}{\Rightarrow} = \begin{cases} \stackrel{\tau}{\Rightarrow} & \text{si } s = \epsilon \\ \stackrel{\alpha_1}{\Rightarrow} \cdots \stackrel{\alpha_n}{\Rightarrow} & \text{si } s = \alpha_1 \cdots \alpha_n \end{cases}$$

D'après [Mil89], un processus est considéré déterministe si l'exécution de la même suite d'actions résulte en le même processus, modulo équivalence sémantique.

Définition 26 (Déterminisme)**.** *Un processus est déterministe si pour toute séquence s, si $P \stackrel{s}{\Rightarrow} P_i$ pour $i = 1, 2$ alors $P_1 \approx P_2$.*

De plus, nous savons que tout processus déterministe est τ-inerte.

Définition 27 (τ-inerte)**.** *Un processus est τ-inerte si pour tous ses dérivés Q, $Q \stackrel{\tau}{\rightarrow} Q'$ implique $Q \approx Q'$.*

Afin de simplifier l'étude de la confluence dans la suite, nous allons introduire quelques notions telles que la compatibilité et le résidu d'une action.

Définition 28 (Compatibilité)**.** *Le prédicat de compatibilité $\downarrow$ est défini comme la plus petite relation binaire réflexive et symétrique sur les actions telle que $\alpha \downarrow \beta$ implique que $\alpha, \beta \neq N$ ou $\alpha = \beta = N$.*

En d'autres termes, l'action de fin de l'instant N est compatible avec elle-même seulement tandis que toutes les autres actions sont compatibles entre elles. Intuitivement, la relation de confluence devrait autoriser la commutation des actions exécutées pendant le *même* instant. Afin de simplifier la formulation de la relation de confluence, il convient d'introduire la notion de *résidu d'une action* $\alpha\backslash\beta$ qui représente ce qu'il reste à exécuter de l'action α après avoir exécuté β.

Définition 29 (Résidu d'une action)**.** *Le résidu d'une action $\alpha\backslash\beta$ est une opération définie sur les actions et uniquement quand $\alpha \downarrow \beta$ et satisfait les égalités suivantes :*

$$\alpha\backslash\beta = \begin{cases} \tau & si\ \alpha = \beta \\ \nu\mathbf{t}\backslash\mathbf{t}'\overline{s}v & si\ \alpha = \nu\mathbf{t}\ \overline{s}v\ et\ \beta = \nu\mathbf{t}'\overline{s'}v' \\ \alpha & sinon \end{cases}$$

À l'aide de cette opération, on peut finalement définir la relation de confluence où il s'agit de clore des diagrammes en utilisant des actions compatibles.

Définition 30 (Confluence)**.** *Un processus P est confluent si pour tous ses dérivés Q :*

$$\frac{Q \stackrel{\alpha}{\Rightarrow} Q_1 \quad Q \stackrel{\beta}{\Rightarrow} Q_2 \quad \alpha \downarrow \beta}{\exists\, Q_3, Q_4 \ (\ Q_1 \stackrel{\beta\backslash\alpha}{\Rightarrow} Q_3 \quad Q_2 \stackrel{\alpha\backslash\beta}{\Rightarrow} Q_4 \quad Q_3 \approx Q_4\ )}$$

Une version plus faible de la relation de confluence est la confluence locale, qui est généralement plus facile à vérifier. Elle s'énonce comme suit :

Définition 31 (Confluence locale)**.** *Un processus P est localement confluent si pour tous ses dérivés Q :*

$$\frac{Q \stackrel{\alpha}{\rightarrow} Q_1 \quad Q \stackrel{\beta}{\rightarrow} Q_2 \quad \alpha \downarrow \beta}{\exists\, Q_3, Q_4 \ (\ Q_1 \stackrel{\beta\backslash\alpha}{\Rightarrow} Q_3, \quad Q_2 \stackrel{\alpha\backslash\beta}{\Rightarrow} Q_4, \quad Q_3 \approx Q_4\ )}$$

Il est assez facile de trouver des exemples de processus qui sont localement confluents, mais pas confluents. Un tel processus pourrait être $A = \overline{s}_1 \oplus B$ où $B = \overline{s}_2 \oplus A$. Cependant, le lecteur attentif remarquera que ce processus n'est pas réactif. Pour ceux là, la notion de confluence et de confluence locale sont équivalentes.

À ce stade, il est possible d'établir un lien entre la notion de confluence et celle du déterminisme. En effet, Il apparaît que les deux notions coïncident. Nous allons procéder en montrant chaque sens de l'implication séparément.

Proposition 32. *Si un processus est confluent alors il est τ-inerte et déterministe.*

PREUVE.

Prenons $\mathcal{S} = \{(P, P') \mid P \text{ confluent et } P \stackrel{\tau}{\Rightarrow} P'\}$ et définissions la relation $\mathcal{R}$ telle que $\mathcal{R} = \mathcal{S} \cup \mathcal{S}^{-1}$. Nous allons démontrer que $\mathcal{R}$ est une w-bisimulation up to w-bisimulation (cf. lemme 12). D'abord, il est assez facile de vérifier que $\mathcal{R}$ est symétrique et réflexive. Pour le reste, supposons que P est confluent et que $P \stackrel{\tau}{\Rightarrow} Q$ (le cas où Q se réduit à P est un cas symétrique). Si $Q \stackrel{\alpha}{\Rightarrow} Q_1$ alors $P \stackrel{\alpha}{\Rightarrow} Q_1$ et $Q_1 \mathcal{R} Q_1$. D'un autre côté, si $P \stackrel{\alpha}{\Rightarrow} P_1$ alors, par confluence, il existe P_2 et Q_1 tels que $P_1 \stackrel{\tau}{\Rightarrow} P_2$, $Q \stackrel{\alpha}{\Rightarrow} Q_1$, et $P_2 \approx Q_1$. On a finalement que $P_1 \mathcal{R} \circ \approx Q_1$.

Donc si P est confluent et $P \stackrel{\tau}{\Rightarrow} P'$ alors $P \approx P'$. Et puisque tout dérivé Q d'un processus confluent P est confluent, alors si P est confluent, il est aussi τ-inerte.

Ensuite, nous démontrons que :

$$\frac{P_1 \approx P_2 \quad P_1 \stackrel{\alpha}{\Rightarrow} P_3 \quad P_2 \stackrel{\alpha}{\Rightarrow} P_4}{P_3 \approx P_4}\ .$$

Par définition de la bisimulation, $\exists\, P_5$ ($P_2 \stackrel{\alpha}{\Rightarrow} P_5, P_3 \approx P_5$). Par confluence, $\exists\, P_6, P_7$ ($P_5 \stackrel{\tau}{\Rightarrow} P_6, P_4 \stackrel{\tau}{\Rightarrow} P_7, P_6 \approx P_7$). En sachant que les processus en question sont τ-inertes et par transitivité, $P_3 \approx P_4$.

Enfin, nous pouvons itérer cette remarque et conclure que si $P \stackrel{\alpha_1}{\Rightarrow} \cdots \stackrel{\alpha_n}{\Rightarrow} P_1$ et $P \stackrel{\alpha_1}{\Rightarrow} \cdots \stackrel{\alpha_n}{\Rightarrow} P_2$ alors $P_1 \approx P_2$. □

Les actions d'émission et de réception partagent quelques propriétés dans le système de transition présenté dans la sous-section 2.2.2. En particulier, il est vérifiable que si $P \xrightarrow{\nu \mathbf{t} \overline{s} v} P'$ alors $P \equiv \nu \mathbf{t}(\overline{s}v \mid P'')$ et $P' \equiv (\overline{s}v \mid P'')$. Les commutations possibles entre différentes actions compatibles sont résumées dans le lemme suivant.

Lemme 33 (Commutations des entrées et sorties)**.**

$$(in-\tau)\qquad \frac{P \xrightarrow{sv} (P \mid \overline{s}v) \quad P \xrightarrow{\tau} P'}{(P \mid \overline{s}v) \xrightarrow{\tau} (P' \mid \overline{s}v) \quad P' \xrightarrow{sv} (P' \mid \overline{s}v)}$$

$$(in-in)\qquad \frac{P \xrightarrow{sv} (P \mid \overline{s}v) \quad P \xrightarrow{s'v'} (P \mid \overline{s'}v')}{\begin{array}{c}(P \mid \overline{s}v) \xrightarrow{s'v'} (P \mid \overline{s}v) \mid \overline{s'}v' \quad (P \mid \overline{s'}v') \xrightarrow{sv} (P \mid \overline{s'}v') \mid \overline{s}v \\ (P \mid \overline{s}v) \mid \overline{s'}v' \equiv (P \mid \overline{s'}v') \mid \overline{s}v\end{array}}$$

$$(out-\tau)\qquad \frac{\nu\mathbf{t}(\overline{s}v \mid P) \xrightarrow{\nu\mathbf{t}\ \overline{s}v} (\overline{s}v \mid P) \quad \nu\mathbf{t}(\overline{s}v \mid P) \xrightarrow{\tau} \nu\mathbf{t}(\overline{s}v \mid P')}{(\overline{s}v \mid P) \xrightarrow{\tau} (\overline{s}v \mid P') \quad \nu\mathbf{t}(\overline{s}v \mid P') \xrightarrow{\nu\mathbf{t}\ \overline{s}v} (\overline{s}v \mid P')}$$

$$(out-in)\qquad \frac{\nu\mathbf{t}(\overline{s}v \mid P) \xrightarrow{\nu\mathbf{t}\ \overline{s}v} (\overline{s}v \mid P) \quad \nu\mathbf{t}(\overline{s}v \mid P) \xrightarrow{s'v'} \nu\mathbf{t}(\overline{s}v \mid P) \mid \overline{s'}v'}{(\overline{s}v \mid P) \xrightarrow{s'v'} (\overline{s}v \mid P) \mid \overline{s'}v' \quad \nu\mathbf{t}(\overline{s}v \mid P) \mid \overline{s'}v' \xrightarrow{\nu\mathbf{t}\ \overline{s}v} (\overline{s}v \mid P) \mid \overline{s'}v'}$$

$$(out-out)\qquad \frac{\begin{array}{c}\nu\mathbf{t}(\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \xrightarrow{\nu\mathbf{t_1}\ \overline{s_1}v_1} \nu\mathbf{t}\backslash\mathbf{t_1}\ (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \\ \nu\mathbf{t}(\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \xrightarrow{\nu\mathbf{t_2}\ \overline{s_2}v_2} \nu\mathbf{t}\backslash\mathbf{t_2}\ (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P)\end{array}}{\begin{array}{c}\nu\mathbf{t}\backslash\mathbf{t_1}\ (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \xrightarrow{\nu\mathbf{t_2}\backslash\mathbf{t_1}\ \overline{s_2}v_2} (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \\ \nu\mathbf{t}\backslash\mathbf{t_2}\ (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \xrightarrow{\nu\mathbf{t_1}\backslash\mathbf{t_2}\ \overline{s_2}v_2} (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P)\end{array}}$$

Notez que le lemme 33 couvre toutes les commutations possibles d'actions compatibles, si l'on considère les cas modulo symétrie et équivalence structurelle. Les seuls cas non couverts sont ceux où les deux actions α et β considérées sont identiques et dans l'ensemble $\{\tau, N\}$.

Le fait qu'un processus confluent soit déterministe est un fait connu et une propriété standard. Cela est vrai particulièrement parce que les processus confluents sont τ-inertes. Par contre, le fait qu'un processus déterministe soit confluent est spécifique au Sπ-calcul. Cela est prouvable puisque les actions d'émission et de réception commutent automatiquement avec les autres actions compatibles.

Notons que les actions de réception commutent aussi dans le cadre du π-calcul avec communication asynchrone. Cependant, dans le même cadre, les émissions sur canaux ne commutent pas car elles ne persistent pas durant l'instant, contrairement aux émissions sur signaux. Pour illustrer ce fait, prenons l'exemple suivant en CCS :

$$P = a + b$$

Dans cet exemple, le processus P peut évoluer de deux manières différentes suivant l'action exécutée. Après l'exécution de ses deux actions, il n'est pas possible de clore le diagramme. Malgré cela, le processus P est déterministe, d'après la définition 26.

Proposition 34. *Tout processus déterministe est confluent.*

PREUVE. Nous rappelons que si un processus P est déterministe alors il est aussi τ-inerte. Supposons que Q est un dérivé de P, $\alpha \downarrow \beta$, $Q \stackrel{\alpha}{\Rightarrow} Q_1$ et $Q \stackrel{\beta}{\Rightarrow} Q_2$.

Si $\alpha = \beta$ alors, d'après la définition du déterminisme, $Q_1 \approx Q_2$. De plus, notez que $\alpha\backslash\beta = \beta\backslash\alpha = \tau$ et que $Q_i \stackrel{\tau}{\Rightarrow} Q_i$ pour $i = 1, 2$. Ainsi, toutes les conditions requises pour la confluence sont vérifiées.

Dans le cas où $\alpha \neq \beta$ et, modulo cas symétriques, nous avons 5 cas à considérer qui correspondent aux 5 situations déjà considérées dans le lemme 33.

Dans les deux cas où $\beta = \tau$, nous avons que $Q \approx Q_2$, puisque Q est τ-inerte. Par bisimulation, $Q_2 \stackrel{\alpha}{\Rightarrow} Q_3$ et $Q_1 \approx Q_3$. Enfin, $\alpha\backslash\tau = \alpha$, $\tau\backslash\alpha = \tau$, et $Q_1 \stackrel{\tau}{\Rightarrow} Q_1$. De nouveau, toutes les conditions pour la confluence sont remplies.

Il nous reste à considérer les 3 autres cas qui restent où α et β sont différents, actions d'émission ou réception. En utilisant le fait que Q est τ-inerte, nous pouvons nous concentrer sur le cas où $Q \stackrel{\alpha}{\Rightarrow} Q_1$ et $Q \stackrel{\beta}{\rightarrow} Q'_2 \stackrel{\tau}{\Rightarrow} Q_2$. À présent, en itérant le lemme 33, nous pouvons démontrer que :

$$\frac{Q\ (\stackrel{\tau}{\rightarrow})^n\ Q'_1 \quad n \geq 1 \quad Q \stackrel{\beta}{\rightarrow} Q'_2}{\exists\, Q''_2\ (\ Q'_1 \stackrel{\beta}{\rightarrow} Q''_2 \quad Q'_2\ (\stackrel{\tau}{\rightarrow})^n\ Q''_2\ )}$$

La situation finale à considérer est donc celle où $Q \stackrel{\alpha}{\rightarrow} Q'_1 \stackrel{\tau}{\Rightarrow} Q_1$ et $Q \stackrel{\beta}{\rightarrow} Q'_2 \stackrel{\tau}{\Rightarrow} Q_2$. En utilisant le lemme 33, nous obtenons : $Q'_1 \stackrel{\beta\backslash\alpha}{\longrightarrow} Q_3, Q'_2 \stackrel{\alpha\backslash\beta}{\longrightarrow} Q_4$, et $Q_3 \equiv Q_4$. Enfin, en utilisant la bisimulation et le fait que les processus manipulés sont τ-inertes, nous sommes capables de clore le diagramme de transitions. □

Théorème 35.

1. *Un processus est déterministe si et seulement si il est confluent.*
2. *Un processus réactif est déterministe si et seulement si pour tous ses dérivés Q :*

$$\frac{Q \stackrel{\alpha}{\rightarrow} Q_1 \quad Q \stackrel{\alpha}{\rightarrow} Q_2 \quad \alpha \in \{\tau, N\}}{\exists\, Q_3, Q_4\ \ (Q_1 \stackrel{\tau}{\Rightarrow} Q_3 \quad Q_2 \stackrel{\tau}{\Rightarrow} Q_4 \quad Q_3 \approx Q_4)}$$

La première partie du théorème est prouvée en combinant les résultats obtenus dans les propositions 32 et 34.

À propos de la deuxième partie du théorème, il faut d'abord remarquer que les conditions requises sont équivalents à celles de la confluence locale. Ensuite, en se basant sur les résultats de [GS96], nous sommes capables de vérifier que la confluence locale, en plus de la réactivité, suffisent pour assurer la confluence. Ce résultat généralise le lemme de Newman au cas des systèmes de transitions étiquetées. Ce dernier [New42] stipule que toute relation fortement normalisante et localement confluente est confluente.

Fait 36 ([GS96]). *Si un processus est réactif et localement confluent alors il est confluent.*

Nous concluons cette section en remarquant que les actions muettes τ commutent de façon forte. Cela est suffisant pour assurer le déterminisme. Pour ce faire, nous allons utiliser la nouvelle relation de transition $\overset{\alpha}{\rightsquigarrow}$ qui est définie comme $\xrightarrow{\alpha} \cup Id$ où Id est la relation d'identité.

Proposition 37. *Un processus est déterministe si pour tous ses dérivés Q :*

$$\frac{Q \xrightarrow{\tau} Q_1 \quad Q \xrightarrow{\tau} Q_2}{\exists Q' \ (Q_1 \overset{\tau}{\rightsquigarrow} Q' \quad Q_2 \overset{\tau}{\rightsquigarrow} Q')} \qquad \frac{Q \xrightarrow{N} Q_1 \quad Q \xrightarrow{N} Q_2}{Q_1 \approx Q_2}$$

Preuve. Appelons P *fortement confluent* s'il satisfait les hypothèses de la proposition 37. Prenons $\mathcal{S} = \{(P, Q) \mid P$ fortement confluent et $(P \equiv Q$ ou $P \xrightarrow{\tau} Q)\}$ et $\mathcal{R} = \mathcal{S} \cup \mathcal{S}^{-1}$. En utilisant les mêmes techniques utilisées dans d'autres lemmes, nous démontrons que $\mathcal{R}$ est une bisimulation. Cela nous aide à montrer que si un processus est fortement confluent alors il est aussi τ-inerte.

Notez que si $P \xrightarrow{\alpha} P_i$, pour $i = 1, 2$, et α est soit une émission soit une réception alors $P_1 \equiv P_2$. En utilisant le lemme 33 et en étudiant le diagramme de réduction, nous pouvons montrer que P est fortement confluent, car on arrive à clore le diagramme décrit dans l'énoncé de ce lemme. Ainsi, si $P \overset{\alpha}{\Rightarrow} P_i$ pour $i = 1, 2$, alors $P_1 \approx P_2$. Cela correspond à la condition requise pour le déterminisme. □

2.5 Déterminisme par typage

Pour garantir le déterminisme, il existe une autre approche : restreindre les actions possibles par typage. Grâce aux règles de typage, les processus seront déterministes par construction. Leurs interactions avec l'environnement respecteront les contraintes imposées par les types. L'un des intérêts de cette approche par rapport à celle qui est basée sur la notion de bisimulation est qu'elle donne un critère décidable. Elle pourra, de cette manière, être implémentée dans des langages synchrones ou des outils de vérification.

L'étude de la notion de déterminisme faite dans la section précédente suggère l'existence de deux situations qui nécessitent une attention particulière afin de garantir le déterminisme du processus considéré. Les deux situations sont :

1. Au moins deux valeurs peuvent être reçues pendant l'instant. Le processus suivant illustre la situation : $\overline{s}v_1 \mid \overline{s}v_2 \mid s(x).P, K$. Suivant l'action considérée par l'environnement, la première réception exécutée, le processus résultant sera différent.

2. À la fin de l'instant, au moins deux valeurs distinctes sont disponible sur un signal donné. Cela est illustré par : $\overline{s}v_1 \mid \overline{s}v_2 \mid \mathsf{pause}.A(!s)$. Dans ce cas là, le traitement effectué par le processus léger A pourrait dépendre de l'ordre de stockage des éléments dans la liste $!s$.

Pour garantir le déterminisme, une approche raisonnable semble d'interdire la première situation et d'accepter la seconde sous condition. En particulier, cette dernière pourrait être valide si on a la garantie que le traitement effectué par la continuation A (dans l'exemple précédent) ne dépend pas de l'ordre de stockage des éléments. Techniquement, nous utiliserons des *usages de signaux affines* pour interdire la première situation et une notion de type *ensemble* pour vérifier la validité de la seconde. Il s'agit de découvrir une collection d'*usages de signaux* que l'on peut composer facilement.

2.5.1 Usages

En première approximation, nous considérerons qu'un *usage* est un élément de l'ensemble $L = \{0, 1, \infty\}$. Les éléments de L déterminent comment une ressource devrait être utilisée :

- 0 indique que la ressource existe mais ne peut être utilisée,
- 1 assure que la ressource en question peut être utilisée au plus une fois,
- et ∞ n'applique aucune restriction quant à l'utilisation de la ressource. Elle peut être détenue et utilisée par un nombre arbitraire de processus.

Les usages peuvent être additionnés à l'aide de l'opération partielle $\oplus$ d'addition et qui est définie comme suit :

- 0 est l'élément neutre pour $\oplus$: Pour tout usage a, $a \oplus 0 = 0 \oplus a = a$.
- $\infty \oplus \infty = \infty$
- Les cas restants sont indéfinis. En particulier, $1 \oplus 1$ n'est pas défini car toute ressource affine doit le rester. Toute tentative pour définir $1 \oplus 1$ cassera cette condition.

À partir de l'opération d'addition $\oplus$, on déduit une relation d'ordre pour les usages : $a \leq b$ si $\exists c$ tel quel $a \oplus c = b$. En considérant cette relation d'ordre, 0 devient le plus petit élément tandis que 1 et ∞ deviennent incomparables. Et, si $a \leq b$, alors on est capables de définir une relation de soustraction $\ominus$ où $a \ominus b$ est défini comme étant l'usage c le plus grand tel que $a = b \oplus c$. Ainsi, on obtient les égalités suivantes :

- $a \ominus 0 = a$,
- $1 \ominus 1 = 0$,
- et $\infty \ominus \infty = \infty$

La classification des usages telle que présentée ici est adaptée pour gérer des données purement fonctionnelles où, intuitivement, les données ayant l'usage 1 auraient au plus un pointeur pointant sur eux [Wad93]. Cependant, il est nécessaire d'avoir une classification plus fine pour gérer des ressources telle que des références, canaux ou encore signaux. En particulier, un usage

peut être enrichi pour inclure de nouvelles informations quant à son utilisation pour :

1. émettre (ou écrire)
2. recevoir (ou lire)
3. et recevoir à la fin de l'instant.

À partir de ce point, un usage devient un triplet d'éléments de L, donc un élément de L^3. Parmi les 27 possibilités d'usages de la forme (a, b, c) pour $a, b, c \in L$, il n'y aurait que 5 sous-ensembles *principaux* comme décrits dans la table 2.3. De notre point de vue, un signal que l'on ne pourrait utiliser pour émettre ou recevoir n'est pas intéressant. Nous excluons donc tous les usages où $a = 0$ ou bien b et c sont égaux à 0. Nous retrouverons ces usages là plus tard, mais ne seront pas considérés comme principaux. La suite est résumée par le diagramme de décision dans la figure 2.2. Les 5 groupes principaux trouvés peuvent être divisés en deux sous groupes : (1) ceux qui permettent l'émission de plusieurs valeurs dans l'instant, et (2) ceux qui permettent l'émission d'au plus une valeur dans l'instant :

1. Puisque plusieurs valeurs sont disponibles pour la réception, alors b doit être égal à 0. Autrement, on risque de tomber dans une situation non déterministe. Il reste donc à considérer les valeurs possibles pour la composante c. Prendre $c = 0$ ne serait pas possible à cause de la condition imposée un peu plus tôt. Nous finissons donc avec une catégorie de signaux où $c = 1$ et que l'on nommera e_3 et une autre où $c = \infty$ et qui sera nommée e_1.
2. Si on fixe $a = 1$, alors nous avons de multiples choix quant aux valeurs possibles de b et c.
   (a) Il est possible de vouloir imposer une communication entre deux processus seulement, pour garantir une certaine confidentialité. Pour cela, il suffit de mettre $b = 0 \wedge c = 1$ ou bien $b = 1 \wedge c = 0$.
   (b) ou bien de laisser la communication libre et assigner donc $b = \infty \wedge c = \infty$.

   Le lecteur attentif remarquera l'existence de deux autres catégories qui sont $(1, \infty, 1)$ et $(1, 1, \infty)$. Bien que totalement valides d'un point de vue de déterminisme (pour des messages "non affine"), nous avons décidé de ne pas les considérer. Elles représentent des comportements assez spécifiques pour lesquels nous n'avons pas trouvé des exemples intéressants.

À partir des 5 usages principaux trouvés grâce à l'analyse précédente, nous obtenons les usages secondaires dérivés des usages principaux en remplaçant des 1 par des 0. Le résultat est décrit dans la table 2.3. Notez que certains usages (tel que $(1, 0, 0)$) font partie de catégories différentes. Cette

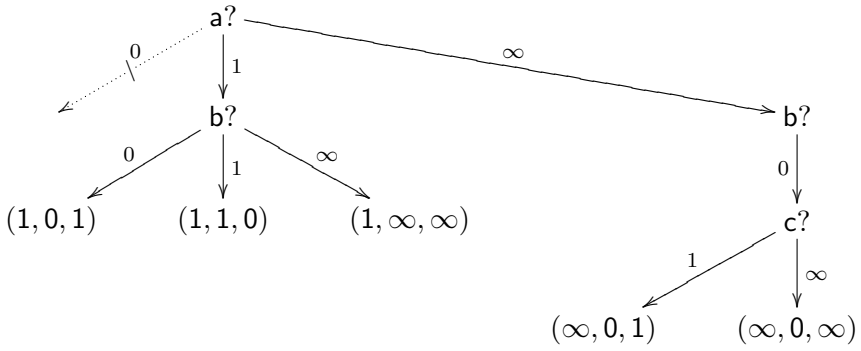


FIGURE 2.2 – Choix des usages

usages principaux	usages dérivés
$e_1 = (\infty, 0, \infty)$	$\_$
$e_2 = (1, \infty, \infty)$	$(0, \infty, \infty)$
$e_3 = (\infty, 0, 1)$	$(\infty, 0, 0)$
$e_4 = (1, 0, 1)$	$(1, 0, 0), (0, 0, 1), (0, 0, 0)$
$e_5 = (1, 1, 0)$	$(1, 0, 0), (0, 1, 0), (0, 0, 0)$

TABLE 2.3 – Catégories des usages

ambiguïté est résolue en demandant que la catégorie de l'usage soit mentionnée explicitement. Enfin, nous étendons les opérations d'addition et de soustraction et la relation de comparaison aux usages dans L^3 à partir de celles définies précédemment pour les usages dans L.

Dans un contexte synchrone, il est utile d'analyser l'évolution des usages de signaux durant le temps. Pour ce faire, la meilleure solution semble d'utiliser des *usages de signaux* définis comme des mots infinis sur l'alphabet L^3, *i.e.* tout x^ω où $x \in L^3$. Pour pouvoir effectuer des analyses plus fines, nous serons amenés à utiliser des usages de signaux de la forme xy^ω où $x, y \in L^3$. Ceux là pourraient être dérivés des usages principaux de la même catégorie.

Dans la suite, nous utiliserons U pour dénoter l'ensemble de tous les usages et $U(i)$ pour l'ensemble des usages de la catégorie i, pour $i = 1, \ldots, 5$. De plus, nous supposerons que l'opération d'addition $\oplus$ n'est définie que pour $u, u' \in U(i)$ et que $u \oplus u' \in U(i)$ pour un certain $i \in \{1, \ldots, 5\}$.

Si $u \in U$ alors $\uparrow u$, l'opération *shift* de u, est le mot infini dans U obtenu à partir de u en supprimant le premier caractère. Cette opération est toujours définie. Si u est un usage de signal, alors $u(i)$ pour $i \geq 0$ dénote le i-ème caractère de u et $u(i)_j$ pour $j \in \{1, 2, 3\}$ la j-ème composante de $u(i)$.

Nous classifions les usages suivant 3 propriétés : affinité, uniformité et préservation de l'affinité. Nous dirons qu'un usage est *affine* s'il contient un

$xy^\omega \in U(i)$ est	affine	uniforme	préserve l'affinité
$i = 1$	*non*	*oui*	*non*
$i = 2$	*oui/non*	*oui/non*	*non*
$i = 3$	*oui/non*	*oui/non*	*oui*
$i = 4$	*oui/non*	*oui/non*	*oui*
$i = 5$	*oui/non*	*oui/non*	*oui*

TABLE 2.4 – Classification des usages

1, et *neutre* sinon. Et qu'il est *uniforme* s'il est de la forme x^ω. Il s'avère que les éléments neutres le sont aussi par rapport à l'opération d'addition $\oplus$ et à la catégorie d'usages $U(i)$ à laquelle ils appartiennent. De plus, tous les usages neutres sont uniformes. Enfin, par définition, les usages appartenant à l'ensemble $\bigcup_{i\in\{3,4,5\}} U(i)$ préservent l'affinité. Cette classification est résumée dans le tableau 2.4.

2.5.2 Types

Les types considérés dans ce système sont les types de signaux et les types inductifs. Ces derniers sont définis à l'aide d'une équation. Prenons l'exemple du type $List(\sigma)$ de liste dont les éléments sont de type σ qui peut être défini comme $List(\sigma) = \mathsf{nil} \mid \mathsf{cons}\ \mathrm{of}\ \sigma, List(\sigma)$. Une valeur est de type $List(\sigma)$ si les constructeurs qui y sont utilisés sont imbriqués suivant le schéma spécifié par la définition du type en question.

Dans notre contexte, les types inductifs viennent avec une information supplémentaire, à savoir un usage qui indique comment les données de ce type doivent être utilisés. Ainsi, cet usage sera dans l'ensemble $\{1, \infty\}$ où le 1 indique que la donnée ne peut être utilisée qu'au plus une fois, et ∞ n'applique aucune restriction quant à l'utilisation de la donnée.

Pour résumer, si $\sigma_1, \ldots, \sigma_k$ sont des types déjà définis alors le type inductif $C_x(\sigma_1, \ldots, \sigma_k)$ est défini par cas sur les constructeurs de la forme $\mathsf{c}\ \mathrm{of}\ \sigma'_1, \ldots, \sigma'_m$ où chaque type σ'_j pour $j = 1, \ldots, m$ est soit l'un des σ_i pour $i = 1, \ldots, m$ ou bien le type inductif $C_x(\ldots)$ que l'on est en train de définir. Cependant, il y a une contrainte supplémentaire à prendre en compte qui impose que l'usage du type soit 1 si l'un des σ_i utilisés est affine car on doit préserver l'affinité. La grammaire détaillée des types est disponible dans la table 2.5.

Quand on collecte des valeurs à la fin de l'instant, nous avons recours à l'utilisation d'ensembles. Pour cette raison, nous introduisons le type Set qui est défini par l'équation $Set_x(\sigma) = \mathsf{nil} \mid \mathsf{cons}\ \mathrm{of}\ \sigma, Set_x(\sigma)$ qui est assez similaire à celle de $List(\sigma)$. Le lecteur attentif aura remarqué que nous utilisions les mêmes constructeurs dans la définition de $List(\sigma)$ et $Set(\sigma)$. Cependant, il est possible d'éviter toute ambiguïté lors du typage en ajoutant

κ	$::= C_\infty(\kappa) \mid Set_\infty(\kappa) \mid List_\infty(\kappa) \mid Sig_u(\kappa)$	(u neutre)
λ	$::= C_1(\sigma) \mid Set_1(\sigma) \mid List_1(\sigma) \mid Sig_u(\kappa) \mid Sig_v(\lambda)$ (u affine et uniforme, v pres. l'aff. et uniforme)	
σ	$::= \kappa \mid \lambda$	(types uniformes)
ρ	$::= \sigma \mid Sig_u(\kappa) \mid Sig_v(\lambda)$	(v pres. l'aff.)

TABLE 2.5 – Grammaire des types

une étiquette dans le nom des constructeurs.

Enfin, on dénoté par $Sig_u(\sigma)$ le type des signaux dont l'usage est u et le type des données transportées est σ. Comme c'est le cas pour les types inductifs, si le type σ est affine alors l'usage doit préserver l'affinité. Pour formaliser ces distinctions, nous utilisons des noms distincts dans la table 2.5 où :

- on dénote par κ l'ensemble des types non affines (neutres, ou encore *classiques*). Ces types sont uniformes et doivent donc avoir un usage uniforme ;
- on dénote par λ l'ensemble des types affines et uniformes ;
- on dénote par σ l'ensemble des types uniformes ;
- et par ρ l'ensemble de tous les types (incluant ceux cités précédemment en plus de ceux qui ont un usage non uniforme).

Remarquez que les types neutres peuvent être imbriqués de façon arbitraire alors que les types affines ne le sont que sous des types qui préservent l'affinité. Enfin, les types avec des usages non uniformes ne peuvent pas être imbriqués. L'utilisation de types dans l'ensemble ρ pose en effet quelques questions intéressantes telles que :

- Quelle est la signification d'envoyer des données contenant des informations qui dépendent du temps ?
- Cet usage devra-t-il être pris en compte lors de l'instant où l'émission a été faite ou lors de la réception ?

Nous ne répondrons pas à ces questions car elles ne nous semblent pas pertinentes pour l'étude du déterminisme.

Enfin, les opérations définies sur les usages de signaux sont étendus aux types. Ainsi, l'opération d'addition $\oplus$ sera étendue aux types telle que $Op_{u_1}(\sigma) \oplus Op_{u_2}(\sigma) = Op(_{u_1 \oplus u_2}(\sigma)$ où $Op \in \{C, Set, List, Sig\}$ [1] et en supposant que $u_1 \oplus u_2$ soit défini. Par exemple, $List_1(\sigma) \oplus List_1(\sigma)$ n'est pas défini car $1 \oplus 1$ ne l'est pas.

Un contexte de types (ou simplement, contexte) Γ est une fonction partielle avec un domaine fini $dom(\Gamma)$ des variables vers les types. Les contextes

1. C est utilisé comme le symbole de n'importe quel type inductif défini

peuvent être additionnés et nous définissons $\Gamma_1 \oplus \Gamma_2$ de la manière suivante :

$$(\Gamma_1 \oplus \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \oplus \Gamma_2(x) & \text{si } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & \text{si } x \in dom(\Gamma_1) \\ \Gamma_2(x) & \text{si } x \in dom(\Gamma_2) \\ \text{non défini} & \text{sinon} \end{cases}$$

Nous écrirons $(\Gamma_1 \oplus \Gamma_2) \downarrow$ quand la somme des deux contextes est définie. L'opération *shift* est aussi étendue aux contextes telle que :

$$(\uparrow \Gamma)(x) = \begin{cases} Sig_{\uparrow u}(x) & \text{si } \Gamma(x) = Sig_u(x) \\ \Gamma(x) & \text{sinon} \end{cases}$$

Enfin, nous dénotons par $\Gamma, x : \sigma$ le contexte Γ qui a été étendu avec la paire $x : \sigma$ où $x \notin dom(\Gamma)$. Un contexte est dit *neutre* s'il assigne des types neutres aux variables contenues dans $dom(\Gamma)$.

2.5.3 Instrumentalisation de la sémantique

Comme expliqué dans la partie précédente, chaque signal appartient à exactement une des 5 catégories des usages de signaux. Analysons en particulier la 5$^{\text{e}}$ catégorie dont l'usage principal est e_5. Le système de types qui sera présenté dans la suite est supposé garantir qu'une valeur émise sur un signal de la 5$^{\text{e}}$ catégorie ne sera reçu qu'au plus une fois durant l'instant. À présent, considérons le processus $P = \overline{s}t \mid s(x).\overline{x}, 0$ tout en supposant que l'usage attribué à s et t sera e_5^ω. À priori, le processus sera considéré comme bien typé. Cependant, P peut effectuer une synchronisation et se réduire sur le processus $Q = \overline{s}t \mid \overline{t}$ qui ne type pas car t est utilisé deux fois alors que son usage est affine. Par définition de l'usage e_5, s ne peut être lu qu'au plus une fois durant un instant. Pour exprimer ce fait, nous allons appliquer une petite modification au système de transition pour marquer (en les soulignant) les émissions de type 5 sur lesquels une émission a eu lieu durant l'instant. L'émission n'a aucun effet sur le système de transitions étiquetées car $\underline{\overline{s}e}$ se comporte exactement comme $\overline{s}e$.

$$(out)\ \frac{e \Downarrow v}{\overline{s}e \xrightarrow{\overline{s}v} \underline{\overline{s}e}} \qquad (\underline{out})\ \frac{e \Downarrow v}{\underline{\overline{s}e} \xrightarrow{\overline{s}v} \underline{\overline{s}e}} \qquad (\underline{reset})\ \frac{e \Downarrow v \quad v \text{ dans } V(s)}{\underline{\overline{s}e} \xrightarrow{[\{v\}/s],V} 0}$$

De plus, nous avons introduit une règle spéciale ($\underline{out}$) pour typer $\underline{\overline{s}e}$ qui requiert au moins l'usage $(1,1,0) \cdot (0,0,0)^\omega$ pour le signal s et oublie les contraintes sur e. De cette manière, on s'assure que le processus sera rejeté au typage s'il existe une deuxième tentative de réception sur le signal s. En d'autres termes, si le typage est conservé avec des transitions "compatibles", alors nous pouvons être sûrs qu'une valeur émise sur un signal de catégorie 5 sera reçue au plus une fois dans l'instant.

2.5.4 Système de types

Le système de types a été construit autour de quelques idées assez simples :

1. Les usages qui incluent la possibilité d'effectuer des opérations d'émissions et réceptions peuvent être décomposés en usages plus basiques. En particulier, l'usage $(1,1,0)^\omega$ peut être décomposé en $(1,0,0)\cdot(0,1,0)^\omega \oplus (0,1,0)\cdot(1,0,0)^\omega$.
2. Émettre des valeurs est une *garantie* de la disponibilité de la ressource alors que la réception *s'appuie* sur les ressources offertes.
3. Chaque ressource affine peut être consommée au plus une fois dans un jugement de typage (ou dans un calcul).

La formalisation du système de types a fait surgir le besoin de distinguer entre les expressions e et les expressions avec déréférenciation r qui interviennent à la fin de l'instant. Sans cette contrainte supplémentaire, on risquerait de comptabiliser des ressources non encore utilisées, ce qui ferait échouer le typage. La même remarque s'applique au processus légers où l'appel d'un processus léger récursif $A(\mathbf{e})$ ne doit pas être traité de la même façon que l'appel d'un processus léger récursif $A(\mathbf{r})$ à la fin de l'instant. Pour ce faire, nous allons utiliser une notation spéciale pour distinguer les deux sortes d'expressions et d'appels à des processus légers. Pour celles qui font intervenir des ressources à la fin de l'instant, nous les noterons $[r]$ au lieu de r et $[A(\mathbf{r})]$ au lieu de $A(\mathbf{r})$.

Nous allons considérer 4 types de jugements de typage : $\Gamma \vdash e : \rho$, $\Gamma \vdash [r] : \rho$, $\Gamma \vdash P$ et $\Gamma \vdash [A(\mathbf{r})]$. Afin d'homogénéiser ces notations, nous allons utiliser $\Gamma \vdash U : T$ en introduisant un type factice Pr pour les processus. Ainsi, nous pourrons écrire $\Gamma \vdash P : Pr$ et $\Gamma \vdash [A(\mathbf{r})] : Pr$ comme version longue de $\Gamma \vdash P$ et $\Gamma \vdash [A(\mathbf{r})]$. Dans la notation $\Gamma \vdash U : T$, U pourra donc être l'un des e, $[r]$, P ou $[A(\mathbf{r})]$ et T l'un des ρ ou Pr.

Nous supposerons que les symboles de fonctions sont connus à l'avance et ont des types non affines de la forme $(\kappa_1, \ldots, \kappa_n) \to \kappa$. Nous ferons l'hypothèse supplémentaire que k dénote un constructeur ou un symbole de fonction dont le type est explicitement donné.

Les règles du système de types sont données dans la table 2.6. Les règles de typage peuvent être divisées en 3 groupes distincts.

Expressions et Processus à la fin de l'instant Les règles $[!_{Set}]$ et $[!_{List}]$ décrivent le type d'un signal déréférencé en fonction de la catégorie à laquelle appartient son usage.

Catégorie 1 La collection des valeurs associée au signal à la fin de l'instant doit être traitée comme un ensemble (type *Set*).

Catégorie 2 Dans ce cas, nous savons que les valeurs envoyées sur le signal sont au plus au nombre de 1. Nous pouvons donc considérer

cette collection comme étant une liste puisqu'il n'existe qu'une seule façon d'ordonner une liste contenant qu'un seul élément.

Catégorie 3 La collection peut contenir plusieurs valeurs et doit donc être considérée comme étant de type *Set affine*.

Catégorie 4 Ce cas est traité de façon similaire au cas 2. La seule différence étant le type résultat qui devra être *affine*.

Catégorie 5 Cette catégorie d'usages ne permet pas la réception à la fin de l'instant.

La règle de typage $[var_{Sig}]$ quant à elle se distingue par sa forme. La raison étant que pour l'appel à un processus léger $K = A(s, !s)$ à la fin de l'instant, nous avons besoin de distinguer entre les ressources nécessaires pour typer $!s$, qui sont consommées pendant l'instant courant, et les ressources utiles pour typer s, qui entrent en jeu à l'instant suivant. Par exemple, nous voudrions typer le processus K dans le contexte $s : Sig_u(\sigma)$ où $u = (0,0,1)^\omega$. Grâce à la règle $[var_{Sig}]$, le typage est possible en divisant l'usage u en $u_1 \oplus u_2$ où $u_1 = (0,0,1) \cdot (0,0,0)^\omega$ et $u_2 = (0,0,0) \cdot (0,0,1)^\omega$. Ainsi, nous pouvons utiliser u_1 pour typer $[!s]$ et u_2 pour $[s]$.

Le type *Set* est une sorte de type quotient. Sa définition nécessite donc l'utilisation d'une relation d'équivalence $\sim_\rho$ sur les valeurs. Cette dernière est définie comme étant la plus petite relation d'équivalence telle que $s \sim_{Sig_u} s$, $c \sim_{C(\sigma)} c$ si c est un constructeur constant du type $C_u(\sigma)$ et :
– $\mathsf{c}(v_1, \ldots, v_n) \sim_{C_u(\sigma_1,\ldots,\sigma_n)} \mathsf{c}(u_1, \ldots, u_n)$ si $v_i \sim_{\sigma_i} u_i$ pour $i = 1, \ldots, n$.
– $[v_1; \ldots; v_n] \sim_{Set_u(\sigma)} [u_1; \ldots; u_m]$ si $\{v_1, \ldots, v_n\} \sim_{Set_u(\sigma)} \{u_1, \ldots, u_m\}$.
où $\{v_1, \ldots, v_n\} \sim_{Set_u(\sigma)} \{u_1, \ldots, u_m\}$ si pour une permutation π, nous avons $v_i \sim_\sigma u_{\pi(i)}$.

De plus, nous supposerons que chaque symbole de fonction f, défini avec type *neutre* de la forme $(\kappa_1, \ldots, \kappa_n) \to \kappa$, respecte le typage suivant les règles suivantes :

1. Si $v_i \sim_{\kappa_i} u_i$, $i = 1, \ldots, n$, $f(v_1, \ldots, v_n) \Downarrow v$ et $f(u_1, \ldots, u_n) \Downarrow u$ alors $v \sim_\kappa u$.
2. Si $\Gamma \vdash f(v_1, \ldots, v_n) : \kappa$ et $f(v_1, \ldots, v_n) \Downarrow v$ alors $\Gamma \vdash v : \kappa$.

Expressions Les arguments et résultats d'une fonction ont toujours un type uniforme.

Processus Nous supposerons que chaque identificateur de processus léger A est défini par une équation $A(\mathbf{x}) = P$ qui précise le type de chaque argument. Ces derniers doivent avoir un type uniforme car un appel à un processus léger peut intervenir à n'importe quel instant. Enfin, A doit avoir quelques propriétés supplémentaires pour s'assurer de son *bon* comportement :

1. Si $v_i \sim_{\sigma_i} u_i$ pour $i = 1, \ldots, n$ alors $A(v_1, \ldots, v_n) \approx A(u_1, \ldots, u_n)$. Cela permet de s'assurer que le comportement de A est invariable quand les arguments sont équivalents.
2. $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P$ est un jugement de typage dérivable.

Nous ferons l'hypothèse supplémentaire que chaque génération de nom frais indique son type *explicitement* comme dans $\nu s : \rho\ P$. Le système de transitions étiquetées présenté dans la table 3.3 est adapté de telle façon que l'action d'émission transporte les informations sur les types des noms dont la portée est restreinte. Ces informations de types seront transformées à l'aide de la fonction *shift* dans la règle (*next*) de la manière suivante : $\nu s : \rho\ s.0, A(s) \xrightarrow{N} \nu s :\uparrow \rho\ A(s)$.

2.5.5 Résultats

Nous commençons par prouver quelques résultats de base qui vont nous permettre de prouver que le typage est préservé après réduction.

Lemme 38 (Affaiblissement)**.** *Si* $\Gamma \vdash U : T$ *et* $(\Gamma \oplus \Gamma') \downarrow$ *alors* $(\Gamma \oplus \Gamma') \vdash U : T$.

Preuve. Par induction sur les règles de typage. À plusieurs reprises, nous utilisons le fait que l'opération $\oplus$ est associative et commutative, à la fois pour les types et les contextes. De plus, les règles de typage sont formulées de telle sorte qu'elles restent valides même quand les usages dans le contexte sont augmentés. La règle (*var*) est un exemple où l'on demande un usage minimum pour valider le typage.

□

Lemme 39.

1. *Si* $\Gamma \vdash U : T$, $\Gamma' \vdash v : \rho$, $(\Gamma \oplus \Gamma') \downarrow$, *et* $x \notin dom(\Gamma)$ *alors* $(\Gamma \oplus \Gamma') \vdash [v/x]U : T$.
2. *Si* $\Gamma \vdash v : \kappa$ *alors il existe un contexte neutre* Γ' *tel que* $\Gamma' \vdash v : \kappa$ *et* $\Gamma = \Gamma' \oplus \Gamma''$.
3. *Si* $\Gamma \vdash v : \rho$ *et* $\rho = \rho_1 \oplus \cdots \oplus \rho_n$ *alors il existe* $\Gamma_1, \ldots, \Gamma_n$ *tels que* $\Gamma_1 \oplus \cdots \oplus \Gamma_n = \Gamma$ *et* $\Gamma_i \vdash v : \rho_i$ *pour* $i = 1, \ldots, n$.

Preuve.

1. Si $x \in FV(U)$ alors obligatoirement, suivant les règles de typage énoncées, $x \in FV(e)$ où $\underline{\overline{s}e}$ est un sous terme de U. De plus, il est possible de typer $\underline{\overline{s}[v/x]e}$ exactement de la même manière que pour $\underline{\overline{s}e}$. Donc, $\Gamma \vdash \overline{[v/x]U : T}$ et nous utilisons le lemme d'affaiblissment pour conclure.

$$(var)\quad \frac{u \geq u' \quad Op \in \{Sig, Set, List, C\}}{\Gamma, x : Op_u(\sigma) \vdash x : Op_{u'}(\sigma)} \qquad (k)\quad \frac{\Gamma_i \vdash e_i : \sigma_i \quad i = 1, \ldots, n \quad k : (\sigma_1, \ldots, \sigma_n) \to \sigma \quad k = f \text{ ou } k = \mathsf{c}}{\Gamma_0 \oplus \Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash k(e_1, \ldots, e_n) : \sigma}$$

$$[var_C]\quad \frac{Op \in \{C, Set, List\}}{\Gamma, x : Op_u(\sigma) \vdash [x] : Op_u(\sigma)} \qquad [var_{sig}]\quad \frac{y^\omega \geq u}{\Gamma, s : Sig_{xy^\omega}(\sigma) \vdash [s] : Sig_u(\sigma)}$$

$$[k]\quad \frac{\Gamma_i \vdash [r_i] : \sigma_i \quad i = 1, \ldots, n \quad k : (\sigma_1, \ldots, \sigma_n) \to \sigma \quad k = f \text{ ou } k = \mathsf{c}}{\Gamma_0 \oplus \Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash [k(r_1, \ldots, r_n)] : \sigma}$$

$$[!_{Set}]\quad \frac{(u(0) \geq (\infty, 0, \infty) \ \wedge \ x = \infty) \vee (u(0) \geq (\infty, 0, 1) \ \wedge \ x = 1)}{\Gamma, s : Sig_u(\sigma) \vdash [!s] : Set_x(\sigma)} \qquad [!_{List}]\quad \frac{(u(0) \geq (0, \infty, \infty) \ \wedge \ x = \infty) \vee (u(0) \geq (0, 0, 1) \ \wedge \ x = 1)}{\Gamma, s : Sig_u(\sigma) \vdash [!s] : List_x(\sigma)}$$

$$(0)\quad \frac{}{\Gamma \vdash 0} \qquad (out)\quad \frac{\Gamma_1 \vdash s : Sig_u(\sigma) \quad u(0)_1 \neq 0 \quad \Gamma_2 \vdash e : \sigma}{\Gamma_1 \oplus \Gamma_2 \vdash \overline{s}e}$$

$$(\nu)\quad \frac{\Gamma, s : Sig_u(\sigma) \vdash P}{\Gamma \vdash \nu s : Sig_u(\sigma)\ P} \qquad (in)\quad \frac{\Gamma_1 \vdash s : Sig_u(\sigma) \quad u(0)_2 \neq 0 \quad \Gamma_2, x : \sigma \vdash P \quad (\Gamma_1 \oplus \Gamma_2) \vdash [A(\mathbf{r})]}{(\Gamma_1 \oplus \Gamma_2) \vdash s(x).P, A(\mathbf{r})}$$

$$(m_s)\quad \frac{s_1, s_2 \in dom(\Gamma) \quad \Gamma \vdash P_i \quad i = 1, 2}{\Gamma \vdash [s_1 = s_2]P_1, P_2} \qquad (m_{\mathsf{c}})\quad \frac{\mathsf{c} : (\sigma_1, \ldots, \sigma_n) \to \sigma \quad \Gamma_1 \vdash v : \sigma \quad \Gamma_2, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P_1 \quad (\Gamma_1 \oplus \Gamma_2) \vdash P_2}{\Gamma_1 \oplus \Gamma_2 \vdash [v \trianglerighteq \mathsf{c}(x_1, \ldots, x_n)]P_1, P_2}$$

$$(par)\quad \frac{\Gamma_i \vdash P_i \quad i = 1, 2}{\Gamma_1 \oplus \Gamma_2 \vdash P_1 \mid P_2} \qquad (rec)\quad \frac{A : (\sigma_1, \ldots, \sigma_n), \ \Gamma_i \vdash e_i : \sigma_i \quad i = 1, \ldots, n}{\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash A(e_1, \ldots, e_n)}$$

$$(\underline{out'})\quad \frac{\Gamma \vdash s : Sig_u(\sigma) \quad u(0) = (1, 1, 0)}{\Gamma \vdash \underline{\overline{s}e}} \qquad [rec]\quad \frac{A : (\sigma_1, \ldots, \sigma_n), \ \Gamma_i \vdash [r_i] : \sigma_i \quad i = 1, \ldots, n}{\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash [A(r_1, \ldots, r_n)]}$$

TABLE 2.6 – Système de types affine

2. Nous procédons par induction sur v. Pour l'étape d'induction, nous utilisons le fait que si $\mathsf{c}(v_1, \ldots, v_n)$ a un type neutre alors v_i doit avoir un type neutre aussi.
3. Si le type ρ est neutre alors $\rho = \rho_1 = \cdots = \rho_n$. En utilisant 2, nous parvenons à trouver un contexte neutre Γ' qui est suffisant pour $\Gamma' \vdash v : \rho$ et $\Gamma' \oplus \Gamma'' = \Gamma$. Ensuite, il suffit de prendre $\Gamma_1 = \Gamma' \oplus \Gamma''$ et $\Gamma_i = \Gamma'$ pour $i = 2, \ldots, n$.

   Si le type ρ est affine et est un type inductif ou un type *Set*, alors n doit être égal à 1. Cela nous permet de conclure pour ce cas.

   Enfin, si le type ρ est affine et est un type de signal alors l'usage du signal dans les types $\rho_1, \ldots, \rho_n$ permet de construire directement les contextes $\Gamma_1, \ldots, \Gamma_n$ recherchés.

□

Lemme 40 (Substitution)**.** *Si* $\Gamma, x : \sigma \vdash U : T$, $\Gamma' \vdash v : \rho$ *et* $(\Gamma \oplus \Gamma') \downarrow$ *alors on a* $(\Gamma \oplus \Gamma') \vdash [v/x]U : T$.

PREUVE. Nous procédons par induction sur le typage de U. Dans un premier temps, nous nous intéressons au cas des expressions dans l'instant :

(var) Supposons que $\Gamma, y : Op_u(\sigma) \vdash y : Op_{u'}(\sigma)$ avec $u \geq u'$.
- Si $\Gamma = \Gamma'', x : \rho$ et $x \neq y$ alors $((\Gamma'', y : Op_u(\sigma)) \oplus \Gamma')(y) = Op_{u''}(\sigma)$ avec $u'' \geq u$. Donc, en utilisant la règle (var), $(\Gamma'', y : Op_u) \oplus \Gamma' \vdash y : Op_{u'}$.
- Si $x = y$ alors $[v/x]y = v$. Si Op n'est pas Sig alors $u = u'$. Par hypothèse, $\Gamma' \vdash v : Op_u(\sigma)$ et par affaiblissement $\Gamma'' \oplus \Gamma' \vdash v : Op_u(\sigma)$. D'un autre côté, si Op est Sig alors, par (var), $(\Gamma'' \oplus \Gamma') \vdash v : Op_u(\sigma)$.

(k) Si k est une constante alors nous appliquons le lemme d'affaiblissement et nous obtenons directement le résultat souhaité. Sinon, supposons que $\Gamma, x : \rho = \Gamma_0 \oplus \Gamma_1 \oplus \cdots \oplus \Gamma_n$ avec $\Gamma_i \vdash e_i : \sigma_i$, $i = 1, \ldots, n$. Prenons $I = \{i \in \{1, \ldots, n\} \mid x \in dom(\Gamma_i)\}$. Si $i \in I$ alors nous supposons que $\Gamma_i = \Gamma_i'', x : \rho_i$. Nous avons $\rho = \oplus_{i \in I} \rho_i$. En utilisant le lemme 39(3), nous parvenons à trouver un contexte Γ_i' tel que $\Gamma_i' \vdash v : \rho_i$ pour $i \in I$ et $\Gamma' = \oplus_{i \in I} \Gamma_i'$. Si $i \notin I$ alors $\Gamma_i \vdash [v/x]e_i : \sigma_i$, (voir lemme 39(1)), et si $i \in I$ alors $(\Gamma_i \oplus \Gamma_i') \vdash [v/x]e_i : \sigma_i$, par hypothèse d'induction.

Les arguments utilisés dans les cas prouvés ci-dessus seront réutilisés à plusieurs reprises dans les cas restants. Comme nous l'avons déjà souligné dans la preuve du lemme 38 d'affaiblissement, un autre point important utile durant la preuve est le fait que les règles de typage permettent l'ajout de ressources supplémentaires au contexte sans affecter la conclusion. Toutefois, nous allons expliquer un point critique dans le cas de la règle $[var_{sig}]$ quand $\Gamma, s : Sig_{xy^\omega}(\sigma) \vdash [s] : Sig_u(\sigma)$, $y^\omega \geq u$, $\Gamma' \vdash s' : Sig_{xy^\omega}(\sigma)$ et $(\Gamma \oplus \Gamma') \downarrow$.

Ici, nous avons que $\Gamma'(s) = s' : Sig_{u'}(\sigma)$ avec $u' \geq xy^\omega$. Cela nous permet de déduire que $\uparrow(u') \geq y^\omega \geq u$.

□

Nous allons définir les conditions dans lesquelles un contexte Γ est dit *compatible* avec une action *act*, noté $(\Gamma, act)\downarrow$. Si $V(s) = [v_1; \ldots; v_n]$ alors $(V \backslash E)(s) = \{v_1, \ldots, v_n\} \backslash E(s)$. Ensuite, nous définissons le processus $P_{(V\backslash E)}$ comme la composition parallèle des émissions $\overline{s}v$ où $v \in (V\backslash E)(s)$. Cela représente les émissions de toutes les valeurs qui sont présentes dans la liste V mais pas dans l'ensemble E.

Définition 41. *À toute action act, nous associons un processus P_{act} minimal qui permet l'exécution de l'action en question :*

$$P_{act} = \begin{cases} 0 & si\ act = \tau\ ou\ act = N \\ \overline{s}v & si\ act = sv\ ou\ act = s?v \\ s(x).0, 0 & si\ act = \overline{s}v \\ P_{V\backslash E} & si\ act = (E, V) \end{cases}$$

Définition 42 (Compatibilité d'un contexte et d'une action)**.** *Un contexte Γ est* compatible *avec une action act, noté $(\Gamma, act)\downarrow$, si $\exists\Gamma'$ tel que $(\Gamma \oplus \Gamma')\downarrow$ et $\Gamma' \vdash P_{act}$.*

À présent, nous pouvons introduire le concept de *transition typée* qui est une transition étiquetée avec une action *act* et qui fait intervenir un processus P typé dans un contexte Γ, compatible avec l'action *act*.

Définition 43 (Transition typée)**.** *Nous noterons $P \xrightarrow[\Gamma]{act} Q$ (resp. $P \overset{act}{\underset{\Gamma}{\Rightarrow}} Q$) si les conditions suivantes sont vérifiées :*

1. $\Gamma \vdash P$
2. $(\Gamma, act)\downarrow$
3. $P \xrightarrow{act} Q$ *(resp.* $P \overset{act}{\Rightarrow} Q$*).*

Dans la suite, nous utiliserons une notion de *résidu de contexte*. Intuitivement, cela représente le contexte qui reste après une transition typée. D'abord, nous remarquons qu'étant donné un type uniforme σ et une valeur v, nous pouvons définir un contexte minimum $\Delta(v, \sigma)$ tel que $s : \sigma$ et $\Delta(\mathsf{c}, (v_1, \ldots, v_n)) = \Delta(v_1, \sigma_1) \oplus \cdots \oplus \Delta(v_n, \sigma_n)$ si $c : (\sigma_1, \ldots, \sigma_n) \to \sigma$. Notez que $\Delta(v, \sigma)$ peut-être un contexte vide quand $fn(v) = \emptyset$ et qu'il est un contexte neutre si σ l'est à la base.

Définition 44 (Résidu de contexte)**.** *Étant donné un contexte Γ et une action act compatible, le* résidu du contexte $\Gamma(act)$ *est défini de la manière suivante.*

$$\Gamma(act) = \begin{cases} \Gamma & si\ act = \tau \\ \uparrow \Gamma & si\ act = N \\ (\Gamma, \mathbf{t} : \sigma') \ominus \Delta(v : \sigma') \oplus \{s : Sig_{u_5}(\sigma')\} & si\ \Gamma(s) = Sig_u(\sigma'), act = \nu \mathbf{t} : \sigma' \overline{s} v, (1) \\ \Gamma \oplus \Delta(v, \sigma') \oplus \{s : Sig_{u_{out}}(\sigma')\} & si\ \Gamma(s) = Sig_u(\sigma'), act = sv, (2) \end{cases}$$

1. $u_5 = (0,1,0) \cdot (0,0,0)^\omega$ *si* $u \in U(5)$ *et est neutre sinon (*i.e., $u \in U(2)$*).*
2. u_{out} *est le plus petit usage faisant partie de la même catégorie que* u *qui permettra d'exécuter une émission dans l'instant. Un tel usage est toujours défini.*

La notion de résidu de contexte montre précisément de quelle manière les transitions affectent le typage. Pour caractériser cet effet, nous allons procéder en étapes en commençant par prouver que le type des expressions est préservé par l'évaluation.

Lemme 45 (Évaluation des expressions)**.** *Si* $\Gamma \vdash e : \rho$ *et* $e \Downarrow v$ *alors* $\Gamma \vdash v : \rho$.

Preuve. Par induction sur l'évaluation $e \Downarrow v$. Si e est un signal s ou une constante c alors $e = v$ et la conclusion est immédiate. Sinon, supposons que $e = k(e_1, \ldots, e_n)$, $k : (\sigma_1, \ldots, \sigma_n) \to \sigma$, $\Gamma = \Gamma_0 \oplus \Gamma_1 \oplus \cdots \oplus \Gamma_n$, $\Gamma_i \vdash e_i : \sigma_i$, et $e_i \Downarrow v_i$, pour $i = 1, \ldots, n$. En utilisant l'hypothèse d'induction, on a $\Gamma_i \vdash v_i : \sigma_i$, pour $i = 1, \ldots, n$. Si k est un constructeur c alors $v = \mathsf{c}(v_1, \ldots, v_n)$ et $\Gamma \vdash v : \sigma$, suivant la règle (k). Si k est une fonction f alors, encore suivant la règle (k), $\Gamma \vdash f(v_1, \ldots, v_n) : \sigma$ et, par hypothèse sur f, on a que $f(v_1, \ldots, v_n) \Downarrow v$ et $\Gamma \vdash v : \sigma$.

□

Le lemme suivant décrit l'effet de la substitution à la fin de l'instant.

Lemme 46 (Substitution à la fin de l'instant)**.**

1. *Si* $\Gamma \vdash [A(\mathbf{r})]$, $\Gamma' \vdash P_V$, *et* $(\Gamma \oplus \Gamma') \downarrow$ *alors* $\uparrow (\Gamma \oplus \Gamma') \vdash V(A(\mathbf{r}))$.
2. *Si, de plus, on a* V', E *tels que* $V, V' \Vdash\!\!- E$ *alors* $V(A(\mathbf{r})) \approx V'(A(\mathbf{r}))$.

Preuve.

1. L'effet de $V(A(\mathbf{r}))$ est de remplacer chaque occurrence de $!s$ dans $\mathbf{r}$ avec $V(s)$. Tout d'abord, remarquons que l'usage de s ne peut pas être de la catégorie 5 si $!s$ apparaît dans $\mathbf{r}$. De plus, si l'usage est de catégorie 1 ou 2 alors il peut y avoir plusieurs occurrences de $!s$ dans $\mathbf{r}$ mais le type des valeurs émises sur le signal devra être non affine. Notez que pour typer une valeur non affine, on n'a pas besoin de plus que d'un contexte non affine. Puisque les types non affines sont exactement les types neutres, nous pouvons réutiliser le contexte neutre autant de fois que nécessaire. D'autre part, si le signal est de catégorie 3 ou 4 alors les valeurs émises sur le signal peuvent être affines mais il n'y aura qu'au plus une occurrence de ! dans $\mathbf{r}$.

En tenant compte de ces quelques remarques préliminaires, on procède avec une analyse de cas sur les règles $[!_{Set}]$ et $[!_{List}]$. Dans chaque cas, il y a un jugement de typage dont la forme ressemble à :

$$\Gamma, s : Sig_u(\sigma) \vdash [!s] : Op_x(\sigma)$$

en sachant que $\Gamma' \vdash V(s) = [v_1; \ldots; v_n] : Op_x(\sigma)$.

2. Par définition, $V(A(r_1, \ldots, r_n)) = A(V(r_1), \ldots, V(r_n))$. Supposons que $A : (\sigma_1, \ldots, \sigma_n)$. On sait que $v_i \sim_{\sigma_i} u_i$ entraîne que $A(v_1, \ldots, v_n) \approx A(u_1, \ldots, u_n)$. Donc, cela est suffisant pour montrer que $V(r_i) \sim_{\sigma_i} V'(r_i)$ pour $i = 1, \ldots, n$. Nous procédons par induction sur la structure de r.

   Si r est un signal ou une constante alors par définition $r \sim_{\sigma_i} r$.

   Si r est de la forme $!s$ alors nous analysons la catégorie de l'usage de s. Si il est catégorie 2 ou 4 alors $V(s) = V'(s)$ (il y a au plus une valeur dans la liste). Par contre, si il est de catégorie 1 ou 3 alors $V(s)$ et $V'(s)$ sont égaux modulo permutation, suivant la définition de la relation $\sim$ des types *Set*. Enfin, si $r = k(\mathbf{r})$ alors nous appliquons l'hypothèse d'induction qui nous permet de conclure.

□

Enfin, nous démontrons le lemme de préservation du typage par réductions. Il vérifie que le processus retrouvé après une transition typée est typable dans le contexte résidu.

Théorème 47 (Préservation du typage par réduction). *Si* $P \xrightarrow[\Gamma]{act} Q$ *alors* $\Gamma(act) \vdash Q$.

PREUVE.

Nous procédons par induction sur la transition et par analyse de cas sur l'action *act* qui est exécutée. Nous détaillerons seulement les cas les plus intéressants.

(sv) Il y a une seule règle à considérer qui est (in). Supposons que $\Gamma(s) = Sig_u(\sigma')$. La définition de résidu de contexte fournit un contexte supplémentaire $\Delta(v, \sigma') \oplus \{s : Sig_{u_{out}}(\sigma')\}$ qui représente exactement ce qui est nécessaire pour typer $\overline{s}v$.

$(s?v)$ Il y a 3 règles à considérer : (in_{aux}), $(comp)$, et (ν). Nous détaillerons la première. Le traitement nécessaire pour les autres est sensiblement similaire à ce qui l'a été pour la première. Supposons que $(\Gamma_1 \oplus \Gamma_2) \vdash s(x).P, K, \Gamma_1 \vdash s : Sig_u(\sigma')$, $u(0)_2 \neq 0$, $\Gamma_2, x : \sigma' \vdash P$, et $\Gamma_1 \oplus \Gamma_2 \vdash [K]$. Obligatoirement, on aura $u \geq u_{in}$. Par construction, $\Delta(v, \sigma') \vdash v : \sigma'$. En utilisant le lemme 40 de substitution, $\Gamma_2 \oplus \Delta(v, \sigma') \vdash [v/x]P$. Il est alors suffisant d'appliquer le lemme d'affaiblissement pour calculer le résidu de contexte.

($\nu\mathbf{t} : \sigma\ \bar{s}v$) Il y a 5 règles à considérer : (*out*) (qui requiert un traitement spécial pour la catégorie 5 des usages), ($\underline{out}$), (ν_{ex}), (*comp*), et (ν).

(τ) Il y a 8 règles à considérer : (*synch*), (*rec*), ($=_i^{sig}$), ($=_i^{ind}$), (*comp*), et (ν) pour $i = 1, 2$. Nous nous intéresserons aux deux premières règles seulement.

(*synch*) Supposons que $P_1 \xrightarrow{\nu\mathbf{t}:\rho\bar{s}v} P_1'$, $P_2 \xrightarrow{s?v} P_2'$, $\Gamma_i \vdash P_i$, pour $i = 1, 2$, et $(\Gamma_1 \oplus \Gamma_2)(s) = Sig_u(\sigma')$. En utilisant l'hypothèse d'induction, nous avons :

$$(\Gamma_1, \mathbf{t} : \rho) \ominus \Delta(v, \sigma') \oplus \{s : Sig_{u_5}(\sigma')\} \vdash P_1'$$
$$\text{et } (\Gamma_2 \oplus \Delta(v, \sigma') \ominus \{s : Sig_{u_5}(\sigma')\} \vdash P_2'$$

Rappelons que u pourrait être de catégorie 2 ou 5 et que dans le premier cas u_5 est neutre. Dans les deux cas, nous obtenons $(\Gamma_1 \oplus \Gamma_2), \mathbf{t} : \rho \vdash (P_1' \mid P_2')$. Cela nous aide à conclure en appliquant la règle de typage (ν).

(*rec*) Supposons que $A : (\sigma_1, \ldots, \sigma_n)$, $\Gamma_i \vdash e_i : \sigma_i$, $e_i \Downarrow v_i$, pour $i = 1, \ldots, n$. En appliquant le lemme 45, nous obtenons que $\Gamma_i \vdash v_i : \sigma_i$, pour $i = 1, \ldots, n$. Par hypothèse, nous savons que $A(x_1, \ldots, x_n) = P$ et donc que $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P$. Donc, en itérant le lemme 40 de substitution, nous obtenons, comme demandé, $\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash [v_1/x_1, \ldots, v_n/x_n]P$.

(E, V) Il y a 5 règles à considérer : (0), (*reset*), ($\underline{reset}$), (*cont*), et (*par*). Nous allons nous concentrer sur les deux dernières.

(*cont*) Supposons que $s(x).P, K \xrightarrow{(\emptyset,V)} V(K)$ et $\Gamma \vdash s(x).P, K$. Cela nous aide à déduire que $\Gamma \vdash [K]$. En se basant sur le lemme 46(1), nous construisons le contexte Γ' en prenant $\Gamma' = \Delta(V, \Gamma)$, qui est uniforme, en l'ajoutant au contexte Γ'' qui fournit ce qui est nécessaire pour émettre pendant le premier instant les valeurs dans V sur les signaux dans $dom(V)$.

(*par*) Supposons que $\Gamma = (\Gamma_1 \oplus \Gamma_2)$, $\Gamma \vdash (P_1 \mid P_2)$, $(P_1 \mid P_2) \xrightarrow{(E_1 \cup E_2), V} (P_1' \mid P_2')$, $\Gamma_i \vdash P_i$, $P_i \xrightarrow{(E_i,V)} P_i'$, pour $i = 1, 2$. Suivant la définition de résidu de contexte, nous définissons :

$$\begin{array}{ll} Exp_i = \Delta(E_i, \Gamma_i) & Exp_{1,2} = \Delta(E_1 \cup E_2, \Gamma_1 \oplus \Gamma_2) \\ Imp_i = \Delta(V \backslash E_i, \Gamma_i) & Imp_{1,2} = \Delta(V \backslash (E_1 \cup E_2), \Gamma_1 \oplus \Gamma_2) \\ \Gamma_i' = \uparrow \Gamma_i \ominus Exp_i \oplus Imp_i & \Gamma' = \uparrow (\Gamma_1 \oplus \Gamma_2) \ominus Exp_{1,2} \oplus Imp_{1,2} \end{array}$$

pour $i = 1, 2$.

Nous voulons montrer que $\Gamma' = \Gamma_1' \oplus \Gamma_2'$. Nous procédons par analyse sur la contribution de chaque valeur $v \in V(s)$ au calcul de Imp_i, $Imp_{1,2}$, Exp_i, et $Exp_{1,2}$ où $\Gamma(s) = Sig_u(\sigma)$. Nous utilisons la notation $Imp_1(v)$ pour dénoter la contribution de la valeur v au calcul du contexte Imp_1.

- Si s est neutre alors, pour $i = 1, 2$, Imp_i, et $Imp_{1,2}$ sont des contextes neutres. Mais, Exp_i et $Exp_{1,2}$ sont des contextes vides. Modulo symétrie, v peut recevoir soit (i) Γ_i, $i = 1, 2$ soit (ii) Γ_1 et Γ_2. De plus, v sera émise par (i) $E_1 \cap E_2$, ou (ii) $E_1 \backslash E_2$, ou bien (iii) $E_2 \backslash E_1$, ou encore (iv) l'environnement. Chaque situation résultante doit être analysée. Elles sont au nombre de 8.
- Si σ est affine alors l'usage de u doit être de catégorie 3 ou 4 et, à la fin de l'instant, le signal s doit pouvoir lire exclusivement par (i) Γ_i, $i = 1, 2$ ou bien par (ii) l'environnement. D'un autre côté, v peut être émis par (i) $(E_1 \cap E_2)$, (ii) $(E_1 \backslash E_2)$, (iii) $(E_2 \backslash E_1)$ ou bien par (iv) $(V \backslash (E_1 \cup E_2))$. Si $v \in (E_1 \cap E_2)(s)$ alors $\Delta(v, \sigma)$ doit être neutre, car l'addition ne serait pas définie sinon. Nous procédons par une analyse de cas sur les 8 situations. Notez que si l'environnement reçoit v alors le contexte d'*import* $Imp_i, Imp_{1,2}$ sera vide alors que si Γ_i reçoit v alors Exp_i sera vide.

(N) Pour ce cas ci, il y a une seule règle à considérer qui est (*next*). Supposons que $\Gamma \vdash P$ et $P \succeq \nu \mathbf{s} : \rho\ P''$. Typer $(\nu s : \rho\ Q_1) \mid Q_2$ requiert les mêmes conditions que pour typer $\nu s : \rho\ (Q_1 \mid Q_2)$. Donc $\Gamma \vdash \nu \mathbf{s} : \rho\ P''$ et $\Gamma, \mathbf{s} : \rho \vdash P''$. Par définition de la règle (*next*), $P'' \xrightarrow{(E,V)} P'$ avec $V \Vdash\!\!- E$. En utilisant l'hypothèse d'induction et le lemme d'affaiblissement, on obtient $\uparrow(\Gamma, \mathbf{s} : \rho) \vdash P'$. D'où $\uparrow(\Gamma) \vdash \nu \mathbf{s} :\uparrow \rho\ \vdash P'$.

□

À présent, nous voudrions montrer que les processus typables dans le système de types que l'on vient de décrire sont effectivement déterministes. Pour ce faire, nous commençons par introduire une notion de *bisimulation typée* qui raffine celle donnée dans la définition 9 en se focalisant sur les processus typés et les transitions typées. Prenons Cxt l'ensemble des contextes. Si $\Gamma \in Cxt$, nous considérerons que $Pr(\Gamma)$ est l'ensemble des processus typables dans le contexte Γ.

Définition 48 (Bisimulation typée)**.** *Une bisimulation typée est une fonction $\mathcal{R}$ indéxée sur Cxt telle que pour tout contexte Γ, $\mathcal{R}_\Gamma$ est une relation symétrique sur $Pr(\Gamma)$ satisfaisant :*

$$\frac{P\ \mathcal{R}_\Gamma\ Q \qquad P \xrightarrow[\Gamma]{\alpha} P' \qquad bn(\alpha) \cap fn(Q) = \emptyset}{\exists Q'\ (\ Q \overset{\alpha}{\underset{\Gamma}{\Rightarrow}} Q' \qquad P'\ \mathcal{R}_{\Gamma(\alpha)}\ Q'\ )}$$

Nous dénotons par $\approx^t$ la plus grande bisimulation typée.

Comme attendu, la bisimulation typée est plus faible que la bisimulation non typée : Si on ne peut pas distinguer entre deux processus en exécutant

des actions arbitraires, alors nous ne pouvons pas les distinguer en exécutant des actions qui sont compatibles avec le typage.

Proposition 49. *Si $P, Q \in Pr(\Gamma)$ et $P \approx Q$ alors $P \approx^t_\Gamma Q$.*

PREUVE. Il suffit de prouver que la relation suivante est une bisimulation typée :

$$P \,\mathcal{R}_\Gamma\, Q \quad \text{si} \quad P, Q \in Pr(\Gamma) \text{ et } P \approx Q \ .$$

Supposons que $P \,\mathcal{R}_\Gamma\, Q$, $P \xrightarrow[\Gamma]{\alpha} Q$, et $bn(\alpha) \cap fn(Q) = \emptyset$. Alors :

$P \xrightarrow{\alpha} P'$	(par définition de la transition typée)
$\Gamma(\alpha) \vdash P'$	(par lemme de préservation de typage par réduction)
$Q \stackrel{\alpha}{\Rightarrow} Q', P' \approx Q'$	(par définition de la bisimulation standard)
$\Gamma(\alpha) \vdash Q'$	(par lemme de préservation de typage par réduction)

Cela nous permet de déduire que $P' \,\mathcal{R}_{\Gamma(\alpha)}\, Q'$.

□

Nous écrirons $P \overset{\tau}{\underset{\Gamma}{\rightsquigarrow}} Q$ si $P \xrightarrow[\Gamma]{\tau} Q$ ou $P = Q$. Le lemme suivant déclare une propriété de commutation forte des actions τ typées et nous en déduisons que la bisimulation typée est invariante sous les actions τ.

Lemme 50.

1. *Si $P \xrightarrow[\Gamma]{\tau} P_i$ pour $i = 1, 2$ alors il existe un Q tel que $P_i \overset{\tau}{\underset{\Gamma}{\rightsquigarrow}} Q$ pour $i = 1, 2$.*
2. *Si $P \overset{\tau}{\underset{\Gamma}{\Rightarrow}} Q$ alors $P \approx^t_\Gamma Q$.*

PREUVE.

1. Une inspection de la table 2.2.2 du système de transitions étiquetées révèle que deux réductions τ peuvent se superposer si elles sont produites par deux synchronisations sur le même signal, disons s. Dans ce cas là, s doit avoir un usage de catégorie 2 ou 5. Si il fait partie de la catégorie 2, le typage garantit l'existence d'au plus une valeur émise sur s de telle sorte qu'on puisse supposer que l'on soit dans la situation suivante :

   $$P = C[s(x).P_1, Q_1 \mid s(x).P_2, Q_2 \mid \overline{s}e]$$

   Puisqu'un signal persiste dans l'instant, il est possible de clore le diagramme en une étape. D'un autre côté, Si l'usage de s est de catégorie 5 alors il n'existe qu'au plus un destinataire et donc aucune interférence ne peut avoir lieu.

2. Nous démontrons que $\overset{\tau}{\underset{\Gamma}{\rightsquigarrow}}$ est une bisimulation typée. Si $P = Q$ alors le résultat est direct. Sinon, nous supposerons que $P \xrightarrow[\Gamma]{\tau} Q$. Clairement,

P peut simuler de façon faible toutes les actions que Q pourrait exécuter seulement en exécutant au début une action supplémentaire τ. Supposons donc que $P \xrightarrow[\Gamma]{\alpha} P'$. Notez que $\alpha \neq N$ puisque P pourrait exécuter une action τ.

$\alpha = \tau$ Dans ce cas ci, nous appliquons (1) en remarquant que $\overset{\tau}{\underset{\Gamma}{\rightsquigarrow}} \subseteq \overset{\tau}{\underset{\Gamma}{\Rightarrow}}$.

$\alpha = sv$ Ici, $P' = (P \mid \overline{s}v)$ et nous pouvons clore le diagramme en faisant $Q \xrightarrow{sv} (Q \mid \overline{s}v)$.

$\alpha = \nu \mathbf{t} \overline{s}v$ Encore une fois, puisque la valeur a été émise sur un signal persistant, il est équivalent d'utiliser une synchronisation interne. À partir de là, il suffirait d'exporter la valeur vers l'environnement ou de l'importer.

□

La deuxième propriété clé est que le calcul à la fin de l'instant est déterministe.

Lemme 51. *Si* $P \xrightarrow[\Gamma]{N} P_i$ *pour* $i = 1, 2$ *alors* $P_1 \approx^t_{\uparrow(\Gamma)} P_2$.

PREUVE. Grâce au lemme qui assure la préservation du typage par réduction, nous savons que $\uparrow(\Gamma) \vdash P_i$. Si nous arrivons à montrer que $P_1 \approx P_2$ alors nous pourrons conclure en utilisant la proposition 49.

D'après la règle (*next*) du système de transitions étiquetées, nous devons avoir :

- $P \succeq \nu \mathbf{s}_i\ P'$, pour $i = 1, 2$
- $\mathbf{s}_1$ permutation de $\mathbf{s}_2$
- $P' \xrightarrow{E, V_i} P''_i$, pour $i = 1, 2$
- $V_i \Vdash E$, pour $i = 1, 2$
- $P_i = \nu \mathbf{s}_i P''_i$, pour $i = 1, 2$

En utilisant le lemme 46(2) et le fait 36, nous arrivons à garantir que $P''_1 \approx P''_2$ et $P_1 \approx P_2$.

□

En combinant les deux lemmes précédents, nous en déduisons que les processus typables sont déterministes.

Théorème 52 (Déterminisme)**.** *Si* $P \overset{N}{\underset{\Gamma}{\Rightarrow}} \cdot \overset{N}{\underset{\Gamma'}{\Rightarrow}} \cdots \overset{N}{\underset{\Gamma'}{\Rightarrow}} P_i, i = 1, 2, \Gamma' = \uparrow \Gamma$ *alors* $P_1 \approx^t_{\Gamma'} P_2$.

PREUVE. La preuve est directe et utilise les résultats des lemmes 50(2) et 51 en plus de la définition de la bisimulation typée.

□

Chapitre 3

Canaux

L'étude du déterminisme dans le cadre d'un calcul synchrone avec signaux (2) a permis de développer une théorie des types qui s'appuie sur la notion d'usage de signaux. Ces usages déterminent l'ensemble des utilisations possibles d'un signal pendant un instant.

Les signaux sont utilisés comme moyen de communication de base dans Sπ. Lors de chaque émission d'un signal, il y a une récursion cachée puisqu'un signal émis une fois peut être reçu un nombre arbitraire de fois. Cette méthode de diffusion est spécifique aux signaux. En effet, la communication en utilisant des canaux se fait de pair à pair. Pour compléter ce mécanisme, on peut imaginer ajouter un opérateur de réplication. Ce dernier permettra, par exemple, de programmer un serveur pouvant recevoir plusieurs requêtes sur le même canal pendant le même instant.

L'algèbre des usages présentée dans 2.5.1 est totalement agnostique aux mécanismes de communication. Elle peut donc être réutilisée, en partie, dans d'autres modèles pour résoudre la question de la gestion des ressources. En particulier, nous allons présenter dans ce chapitre un calcul de processus synchrone avec canaux et utiliser la notion d'usages dans un nouveau système de types pour identifier un fragment de processus déterministes dans Tapis (3.4).

Bien que les deux calculs utilisent des moyens de communication différents, nous montrerons dans 3.3 comment Tapis se compare à Sπ et à SL [BDS96], son ancêtre. Au delà de ces différences, la sémantique de Tapis qui sera détaillée dans 3.2 se veut plus simple que celle de Sπ. En particulier, à la fin de l'instant, l'environnement ne collecte pas les valeurs émises durant l'instant et les processus légers ne font pas partie des programmes de Tapis. De plus, l'analyse de confluence qui sera présentée dans ce chapitre ne dépend plus de la théorie de bisimulation. Cela nous a permis de formaliser le calcul en Coq en se focalisant sur l'aspect de gestion des ressources. Ce dernier travail est présenté dans 3.5. Il comprend la formalisation de la sémantique opérationnelle de Tapis ainsi que le système de types qui l'accompagne où

nous avons prouvé en Coq la préservation des types par réduction.

3.1 Définitions

Le TAPIS-calcul est une version synchrone du π-calcul avec communication asynchrone. On s'est inspiré du calcul *TCCS* pour imaginer TAPIS en rajoutant des canaux riches, *i.e.* pouvant transporter des valeurs, et des canaux en particulier. Pour la gestion du temps, on a gardé le modèle *SL* comme référence tout en faisant attention à la cohérence, au niveau sémantique, du langage final. Les processus de TAPIS sont définis dans le tableau 3.1 et leur comportement est défini comme suit :

- Le processus 0 est le processus nul, qui ne fait rien ;
- $P \mid Q$ exécute P et Q en parallèle ;
- $a(x).P$ représente le même processus dans le π-calcul ;
- $!a(x).P$ est la version répliquée de $a(x).P$;
- L'opérateur $\rhd$, appelé *else-next*, a été introduit dans [Ama09]. Il est utilisé pour séparer le calcul qui doit être fait pendant l'instant présent de ceux qui doivent être préparés pour les instants suivants. Ainsi, $a(x).P \rhd Q$ représente le processus qui essaye de recevoir une information sur le canal a et continue avec le processus P dans l'instant courant, ou avec le processus Q dans l'instant suivant si la réception échoue.
- $a(x).P \mathbin{!\rhd} Q$ se comporte comme $!a(x).P$ dans l'instant courant et comme Q dans l'instant suivant. Cette construction n'existe pas dans le Sπ-calcul et est nouvelle. Nous montrerons un exemple d'utilisation plus tard.
- $[t \trianglerighteq u]P, Q$ est une construction de filtrage qui essaye de reconnaître t suivant le filtre u. Le processus évolue en θP, s'il existe une substitution θ telle que $match(t, u) = \theta$, et Q dans le cas contraire où $match(t, u) = \uparrow$. Cette construction fonctionne de la même façon que la construction équivalente présentée dans Sπ dans 2.1.

Dans la suite, on utilisera $fn(P)$ pour désigner l'ensemble des noms libres dans P.

TAPIS offre à l'utilisateur un large choix d'interactions possibles grâce aux opérateurs $\cdot \rhd \cdot$ et $\cdot \mathbin{!\rhd} \cdot$. Les différentes possibilités sont représentées dans le tableau 3.2. Le processus $a(x).P \mathbin{!\rhd} Q$ exécute $a(x).P$ de façon répliquée pendant l'instant courant, puis Q, une seule fois, l'instant suivant. Q peut être vu comme la continuation du processus dans le futur. La seule partie répliquée est dans le processus P, car le comportement de ce dernier dépend de la valeur réçue sur le canal a. Mais, Q est invariable car n'a aucune relation avec les messages reçus sur a.

Dans un contexte non typé, le dernier opérateur de réception introduit $a(x).P \mathbin{!\rhd} Q$ peut être dérivé à partir de la réception répliquée et de l'opéra-

Canaux
$Chan \quad ::= \quad a \mid b \mid \ldots$
Constructeurs
$Cnst \quad ::= \quad * \mid \mathsf{c} \mid \mathsf{d} \mid \mathsf{k} \ldots$
Valeurs (v,v',...)
$Val \quad ::= \quad Chan \mid k(Val, \ldots, Val)$
Variables (x,y,...)
$Var \quad ::= \quad Chan \mid x \mid y \mid \ldots$
Expressions (u,t,...)
$Exp \quad ::= \quad Chan \mid Var \mid Cnst(Exp, \ldots, Exp)$
Filtres (u, u', ...)
$Pat \quad ::= \quad Cnst(Var, \ldots, Var)$
Processus
$P, Q \quad ::= \quad 0 \mid P \mid Q \mid R \mid \nu a \; P \mid [t \trianglerighteq u]P, Q$
Actions
$R \quad ::= \quad \overline{a}\langle t \rangle \mid a(x).P \mid a(x).P \rhd Q \mid !a(x).P \mid a(x).P \; ! \rhd Q$

TABLE 3.1 – Syntaxe des processus

Temporisé \ Répliqué	Non	Oui
Non	$a(x).P$	$!a(x).P$
Oui	$a(x).P \rhd Q$	$a(x).P \; ! \rhd Q$

TABLE 3.2 – Modes de réception dans TAPIS

teur *else-next*. En effet, on pourrait utiliser le codage suivant pour le définir :

$$a(x).P \; ! \rhd Q \stackrel{def}{=} \overline{A}\langle\rangle \mid !A.a(x).(\overline{A}\langle\rangle \mid P), Q$$

Malheureusement, typer cet encodage afin de garantir la confluence mène à quelques complications et nous a amené à prendre cet opérateur $\_ \; ! \rhd \_$ comme un opérateur primitif dans le langage.

Exemple 53 (Une alarme simple)**.** *Voici une première version d'une alarme qui émet un* tictac *d'horloge à chaque s'instant et qui peut s'écrire de la manière suivante :*

$$\overline{tictac}\langle\star\rangle \mid !tictac(x).(0 \rhd \overline{tictac}\langle\star\rangle)$$

Cependant, le tictac *d'horloge émis ne pourra être lu que par l'alarme, elle-même. Cela est dû au fait que* $\overline{tictac}\langle\star\rangle$ *est un message émis sur un canal, et non pas sur un signal. Il ne sera donc reçu qu'une seule et unique fois. Pour résoudre ce défaut, voici une deuxième version qui fournit un service qui émet un* tictac *d'horloge à tous les processus ayant demandé l'inscription :*

$$P = !tictac(x).pause.(\overline{tictac}\langle x\rangle \mid \overline{x}\langle\star\rangle)$$

Ainsi, tous les processus intéressés par ce service devront envoyer sur le canal tictac *un canal sur lequel ils recevront les alertes, à partir de l'instant suivant. Une réduction de l'horloge composée de façon parallèle avec un client ressemblera à :*

$$\begin{array}{rcl} \nu x\ (\overline{tictac}\langle x\rangle \mid P) & \rightarrow & \nu x\ pause.(\overline{tictac}\langle x\rangle \mid \overline{x}\langle\star\rangle) \mid P \\ & \xrightarrow{N} & \nu x\ \overline{tictac}\langle x\rangle \mid \overline{x}\langle\star\rangle \mid P \\ & \rightarrow & \ldots \end{array}$$

Notez le fait que le processus P *ne change pas en fonction du temps car les réplications sans un* else-next *ne sont pas sensibles au tics de l'horloge.*

3.2 Sémantique

Nous allons définir la sémantique du TAPIS-calcul à l'aide d'un système de transition. Dans ce système, nous allons utiliser deux transitions différentes :

- une transition silencieuse, notée $\longrightarrow$, qui représente les interactions internes qui ont lieu pendant un même instant. Ici, on cherche à spécifier seulement les transitions internes des processus. En d'autres termes, on s'intéresse aux transitions que le processus peut effectuer sans la participation de l'environnement.
- et une transition de passage à l'instant suivant, notée $\xrightarrow{N}$, qui synchronise l'ensemble des processus à la fin de l'instant, et commence une nouvelle phase de calcul.

Afin de simplifier les régles de réduction que l'on va énoncer par la suite, nous allons introduire les contextes d'évaluation suivants :

$$\begin{array}{rcl} E & ::= & [\,] \mid [\,] \rhd P \\ F & ::= & ![\,] \mid [\,]\,! \rhd P \end{array}$$

Intuitivement, le contexte d'évaluation E, où on a remplacé le trou par une réception, représente un processus qui a, potentiellement, un *else-next* ($\rhd$) et qui ne s'est pas répliqué. Tandis que F, avec la même transformation, représente ceux qui peuvent se répliquer. On notera le fait que ces contextes doivent être manipulés avec soin car il est tout à fait possible d'obtenir des processus non valides. Par exemple, remplacer $[\,]$ dans F par une émission $\overline{a}\langle t\rangle$ donne, dans tous les cas, un processus qui ne peut être obtenu à l'aide

de la grammaire des processus donnée en 3.1. Ce défaut n'est pas contourné ou corrigé car l'utilisation des contextes dans le système de transition reste bien localisé.

Le système de transition est décrit dans la table 3.3, où la relation d'équivalence sur les processus est définie comme suit :

$$P \mid 0 \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\nu a P) \mid Q \equiv \nu a (P \mid Q) \quad \text{si } a \notin fn(Q)$$

$$\nu a\ \nu b\ P \equiv \nu b\ \nu a\ P$$

Le premier groupe de régles de transition décrit ce qui se passe durant un instant. Les règles (par), (ν) et ($\equiv$) sont plutôt standard. Les règles ($=_1^{\mathsf{ind}}$) et ($=_2^{\mathsf{ind}}$) sont utilisées pour le filtrage. Les deux règles (sync) et (!) décrivent les synchronisations entre les processus. Les contextes d'évaluation sont utilisés pour exprimer tous les cas possibles avec un minimum de règles. Cela nous amène à considérer quatre cas pour chaque régle de synchronisation.

Le second groupe de règles est spécifique au TAPIS-calcul et décrit le calcul à la fin de l'instant. Les émissions de valeurs disparaissent à la fin de l'instant, tel que le décrit la règle ($\rhd_{\mathsf{out}}$). Les règles ($\rhd_{\mathsf{in}}$) et ($\rhd_!$) décrivent les réceptions qui ne dépendent pas du temps et qui restent actives aussi longtemps qu'il le faut, en attente de valeurs. Tandis que les règles ($\rhd_{\mathsf{rec}}$) et ($\rhd_{\mathsf{else}}$) décrivent celles qui dépendent du temps, où l'on se débarasse de la partie qui précède l'opérateur *else-next* ($\rhd$) au passage à l'instant suivant.

Il y a quelques différences sémantiques par rapport à *TCCS*. En particulier, dans TAPIS, les processus où l'opérateur *else-next* apparaît sont les réceptions uniquement. Le processus $P \rhd Q$ n'est pas un processus valide pour tout processus P dans TAPIS. Cela reste cohérent avec l'esprit du modèle *SL*.

Exemple 54 (Une alarme un peu plus complexe)**.** *Maintenant que les régles de transition sont établies, il est possible d'imaginer des exemples un peu plus complexes et intéressants. L'alarme proposée dans la suite est capable d'envoyer un entier représentant le nombre de cycles observés depuis sa mise en marche, et qui peut être remis à zéro sur demande :*

$$\begin{aligned} !tick(x,r,n).r(y).\ & \\ & pause.(\overline{tick}\langle x,r,Z\rangle \mid \overline{x}\langle Z\rangle) \\ \rhd\ & (\overline{tick}\langle x,r,S(n)\rangle \mid \overline{x}\langle S(n)\rangle)) \end{aligned}$$

Cette horloge prend trois arguments qui sont : x (le canal privé sur lequel l'horloge préviendra les clients inscrits), r (un canal de remise à zéro) et n

$$(\mathsf{par})\ \frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad (\nu)\ \frac{P \to P'}{\nu a\ P \to \nu a\ P'}$$

$$(=_1^{\mathsf{ind}})\ \frac{match(t,u) = \theta}{[t \trianglerighteq u]P_1, P_2 \to \theta P_1} \qquad (=_2^{\mathsf{ind}})\ \frac{match(t,u) = \uparrow}{[t \trianglerighteq u]P_1, P_2 \to P_2}$$

$$(\equiv)\ \frac{P \equiv P' \to Q' \equiv Q}{P \to Q} \qquad (\mathsf{sync})\ \frac{}{E[a(x).P] \mid \overline{a}\langle t\rangle \longrightarrow [t/x]P}$$

$$(!)\ \frac{}{F[a(x).P] \mid \overline{a}\langle t\rangle \longrightarrow F[a(x).P] \mid [t/x]P}$$

$$(\rhd_{\mathsf{in}})\ \frac{}{a(x).P \xrightarrow{N} a(x).P} \qquad (\rhd_{\mathsf{else}})\ \frac{}{a(x).P \rhd Q \xrightarrow{N} Q}$$

$$(\rhd_!)\ \frac{}{!a(x).P \xrightarrow{N} !a(x).P} \qquad (\rhd_{\mathsf{rec}})\ \frac{}{a(x).P\ ! \rhd Q \xrightarrow{N} Q}$$

$$(\rhd_\nu)\ \frac{P \xrightarrow{N} P'}{\nu a\ P \xrightarrow{N} \nu a\ P'} \qquad (\rhd_{\mathsf{out}})\ \frac{}{\overline{a}\langle t\rangle \xrightarrow{N} 0}$$

$$(\rhd_{\mathsf{par}})\ \frac{P \xrightarrow{N} P' \quad Q \xrightarrow{N} Q' \quad P \mid Q \not\rightarrow \cdot}{P \mid Q \xrightarrow{N} P' \mid Q'}$$

TABLE 3.3 – Système de transition de TAPIS

(un entier). Le client peut à tout instant demander à l'horloge de remettre le compteur n à zéro.

Cet exemple suppose l'existence d'un type d'entier à la Peano dont l'implantation pourrait être : $Nat() = Z \mid S\ of\ Nat()$ *où le constructeur* Z *représente le zéro, et* S *le successeur qui prend un entier comme argument. La section 3.4 parlera, plus en détails, des types inductifs et de leur usage dans le système de types.*

3.3 Comparaison

Dans cette section, on cherche à comparer TAPIS à *SL* et Sπ. Dans une première partie, on montrera comment coder les directives de communication de *SL* dans TAPIS. On montrera un moyen de simuler, dans une certaine mesure, la programmation par signaux dans TAPIS à l'aide de canaux. Ensuite, on discutera de la différence en termes d'expressivité entre TAPIS et Sπ. On commentera des exemples pour souligner les limites entres les deux calculs.

Avant d'entamer la discussion sur la comparaison de TAPIS avec ces ancêtres, il convient de remarquer qu'il est possible de définir des processus légers récursifs dans TAPIS. Leur codage est possible grâce à la réception répliquée :

$$\mathsf{let\ rec}\ A(x) = P\ \mathsf{in}\ Q \stackrel{def}{=} !A(x).[\overline{A}\langle x\rangle/A(x)]P \mid [\overline{A}\langle x\rangle/A(x)]Q$$

Dans ce cas, il est important d'utiliser $!\cdot$ et non pas $\cdot\ !\rhd\ \cdot$. Utiliser cette dernière aurait pour conséquence la définition du *thread* dans l'instant courant seulement. Comme toute définition de *thread*, sa définition est unique. Il est possible de garantir cela par typage. Dans la suite, on utilisera ce codage comme raccourci syntaxique pour définir des services.

3.3.1 SL

TAPIS est inspiré, entre autres, de *SL*[BDS96] où les signaux sont utilisés comme méchanisme de communication de base. En tant que tel, les signaux peuvent aussi être définis dans TAPIS.

Les opérations de base sur un signal sont l'émission et le test de présence. Rappelons d'abord qu'un signal est censé persister durant un instant, *i.e.* une valeur peut-être reçue plusieurs fois pendant un instant. Dans notre contexte, les émissions et tests de présence seront implantés (respectivement) par des réceptions et des émissions. Une émission sur un signal peut être codée ainsi :

$$\mathsf{emit}(a, v) = a(x).\overline{x}\langle v\rangle\ !\rhd 0$$

Ce processus se compose de deux parties. La première, qui est $a(x).\overline{x}\langle v\rangle$, est exécutée de façon répliquée, seulement durant l'instant courant. Quant

à la seconde, elle n'est exécutée qu'une seule fois pendant l'instant suivant. Chaque processus voulant recevoir ce qui a été émis sur le canal a devra envoyer une requête avec un canal de retour, sur lequel il recevra la valeur recherchée. Ce codage a la bonne propriété d'expliciter le comportement récursif et persistant caché dans cette opération. Quant au test de présence, il pourrait s'écrire de la manière suivante :

$$\mathsf{present}\ a(x)\ \mathsf{do}\ P\ \mathsf{else}\ Q = \nu r(\overline{a}\langle r\rangle \mid r(x).P, Q)$$

Lors du test de présence, on envoie une requête à l'émetteur avec un canal de retour sur lequel il répondra avec la valeur recherchée. Si a est disponible dans l'instant courant, alors P recevra, sur un canal privé, la valeur recherchée. Sinon, dans l'instant suivant, le processus ayant fait la demande se réduira à Q. Cela correspond bien au comportement attendu.

En utilisant le codage présenté ci-dessus, il est possible d'extraire le fragment suivant de Tapis :

$$\begin{array}{rcl} P, Q & ::= & 0 \\ & \mid & P_1 \mid P_2 \\ & \mid & \nu a\ P \\ & \mid & \mathsf{emit}(a, v) \\ & \mid & \mathsf{present}\ a(x)\ \mathsf{do}\ P\ \mathsf{else}\ Q \\ & \mid & \mathsf{let\ rec}\ A(x) = P\ \mathsf{in}\ Q \end{array}$$

On arrive donc à extraire un sous-ensemble de Tapis qui correspond à ce que l'on trouve dans SL.

3.3.2 Sπ

Sπ et Tapis sont des calculs de processus qui semblent assez proches. Par rapport à Sπ, Tapis utilise des canaux pour la communication inter-processus, à la place des signaux, et la réplication, à la place de threads récursifs. Cela dit, Sπ offre un méchanisme de réception à la fin de l'instant qui est absent dans Tapis. Cependant, il est intéressant de remarquer que certains processus de Sπ peuvent être traduits en processus de Tapis, même s'ils utilisent la réception à la fin de l'instant. Si l'on considère l'exemple du *Client-Serveur* en Sπ, vu dans le chapitre précédent, il est possible de le réécrire en Tapis de manière assez simple, voir l'exemple 55.

Exemple 55 (Client-Serveur en Tapis)**.**

$$\begin{array}{lcl} Server(s) & = & !s\,(req(t, v))\,.(0 \rhd \overline{t}\langle f(v)\rangle) \\ Client(s, x, t) & = & \nu s'\ (\overline{s}\langle req(t, x)\rangle \mid pause.(s'(x).\overline{t}\langle x\rangle \rhd 0)) \end{array}$$

Chaque requête reçue par le serveur est traitée immédiatement. Le serveur lit la requête et envoie le résultat calculé pendant l'instant suivant. Quant au

client, il envoie la requête et essaye de recevoir le résultat pendant l'instant suivant. Dans le cas du serveur, on note qu'il n'est plus nécessaire de fournir une obligation de preuve pour garantir que l'utilisation des valeurs reçues ne dépend pas de leur ordre d'arrivée. En utilisant la réception répliquée, on sait que cela sera garanti grâce aux propriétés de l'opérateur de composition parallèle.

Néanmoins, Sπ reste plus expressif que TAPIS. Prenons l'exemple simple d'un processus de Sπ qui calcule la moyenne des valeurs reçues sur un signal, sans connaître leur nombre à l'avance, tel que montré dans l'exemple 56. Il ne semble pas évident que l'on puisse coder un processus déterministe qui ferait le même travail en TAPIS. La raison étant assez simple : Il faudrait que l'on puisse collecter des éléments et les stocker dans une liste, pendant le même instant. Dans Sπ, l'opération décrite est possible car l'environnement collecte les valeurs et les ré-injecte dans le processus en effectuant une substitution (remplacement de $!s$ par la liste, ou l'ensemble, des valeurs émises). Cependant, il n'existe aucun équivalent en TAPIS à cette opération.

Exemple 56 (Calcul de moyenne en Sπ)**.** *Le processus suivant reçoit une référence à un signal, et calcule, à l'instant suivant, la moyenne de toutes les valeurs reçues sur ce dernier. Le résultat est envoyé à celui qui l'a demandé à la fin du calcul.*

$$\begin{array}{lcl} Avg(s,r) & = & pause.Avg\_aux(s,r,!s,0,0) \\ Avg\_aux(s,r,\ell,s,n) & = & [\ell \trianglerighteq cons(x,\ell')] \quad Avg\_aux(s,r,\ell',s+x,n+1) \\ & & \quad ,[n \trianglerighteq 0]\ \overline{r}\langle 0\rangle,\ \overline{r}\langle \frac{s}{n}\rangle \end{array}$$

En utilisant le code des appels aux processus légers donné précédemment, on peut écrire le processus suivant en TAPIS pour calculer la moyenne :

$$Avg(s,x,n,r) = s(y).Avg(s,x+y,n+1,r) \rhd (\overline{r}\langle x/n\rangle \mid Avg(s,0,0,r))$$

Cependant, le comportement d'un processus utilisant Avg se sera pas garanti d'être déterministe puisqu'il est possible d'envoyer de façon concurrente plusieurs fois sur le canal Avg et on ne saura différencier les différentes émissions faites durant un instant pour calculer le résultat final.

3.4 Déterminisme par typage

La propriété de confluence a été étudiée dans [NS97] où on utilise la technique "*port-uniqueness*" pour garantir la confluence des processus typés. Cependant, cette technique ne permet pas de typer le processus $\overline{a}\langle b_1\rangle \mid \overline{a}\langle b_2\rangle$ bien qu'il soit confluent. Dans le système de types que l'on présente dans cette section, on adapte ce qui a été fait pour Sπ où on a identifié les situations qui doivent être analysées de façon à garantir le déterminisme. Cela permet d'identifier plus de processus confluents par rapport à ce qui a été

proposé dans [NS97]. On note cependant qu'il n'est pas possible, dans le formalisme étudié, que plusieurs processus puissent reçevoir la même valeur dans un instant tout en étant déterministes. Les situations déterministes sont résumées dans ce qui suit :

- le premier cas est très simple : deux processus s'échangent une valeur. Dans ce cas, l'émetteur et le récepteur sont uniques.
- le second cas fait intervenir plus que deux processus : un processus reçoit des valeurs émises par d'autres processus. Ici, seulement le récepteur est unique.

Les autres cas restant ne sont pas déterministes. Une analyse assez simple permet de le vérifier. En particulier, le cas où un seul processus envoie la même valeur à plusieurs processus n'est pas déterministe car la valeur ne pourra être reçue que par un seul autre processus. Néanmoins, cette situation peut-être encodée en TAPIS en utilisant la même technique utilisée pour simuler les signaux en avec des canaux (c'est-à-dire, en inversant émissions et réceptions et en utilisant un canal de retour).

La suite décrira le système de types proposé pour garantir la confluence. Dans un premier temps, j'introduirai les usages et types utilisés. Puis, on parlera des règles de typage et des propriétés et théorèmes démontrés.

3.4.1 Usages

Dans cette partie, nous utiliserons les *usages* tels que détaillés dans la section 2.5.1. Contrairement à Sπ où il y a deux formes de réception possibles (réception dans l'instant et réception à la fin de l'instant), on n'en a qu'une seule dans TAPIS. On se contentera donc de considérer qu'un couple, au lieu d'un triplet, pour les *usages de canaux dans l'instant* où :

- La première composante représentera le nombre de ressources disponibles (ou autorisées) en émissions,
- et la seconde sera réservée au nombre de ressources autorisées pour la réception.

Chaque *usage de canal* (a, b) sera donc un couple composé d'un *usage* qui appartient à l'ensemble $L = \{0, 1, \infty\}$. Parmi les 9 possibilités, nous allons utiliser seulement 6 usages. La sélection des usages principaux peut être justifiée de la manière suivante : Tout d'abord, nous devons exclure l'usage où a et b sont égaux à zéro. Ensuite, nous devons faire une analyse par rapport aux autres possibilités. Si a est égal à ∞, alors b doit être différent de ∞, sinon l'usage que l'on pourrait faire d'un canal avec de telles ressources ne serait pas déterministe (*i.e.* tous les processus peuvent envoyer et écouter sur le même canal. Il n'y a aucun moyen de garantir, par exemple, qu'un tel processus sera toujours capable de recevoir). On obtient donc la première famille dont l'usage principal est $(\infty, 1)$ (le deuxième usage de cette famille est $(\infty, 0)$, où la ressource utilisée pour la réception a été consommée). Ensuite, si a est égal à 1, alors b ne peut pas être égal à ∞ car cela voudrait dire que plusieurs

processus pourraient recevoir ce qu'un seul processus aurait envoyé, ce qui résulterait en un comportement non déterministe. Ce comportement interdit peut être retrouvé en utilisant la première famille d'usages, en utilisant la technique décrite dans la section 3.3.1. Ainsi, on trouve la deuxième famille dont l'usage principal est $(1,1)$. Les familles d'usages que l'on va considérer dans la suite sont décrites dans la figure 3.1. Dans la figure mentionnée, les familles d'usages sont ordonnées de gauche à droite, et les usages principaux apparaissent en haut.

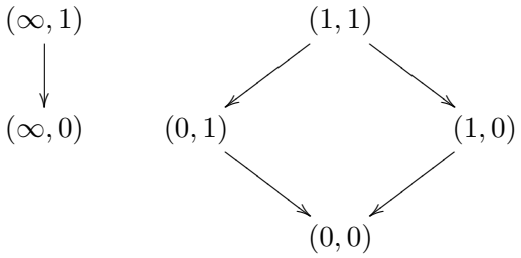


FIGURE 3.1 – Usages

Dans la suite, nous utiliserons le symbole U pour désigner l'ensemble des usages retenus, et par $U(i)$ l'ensemble des usages de la famille d'usages i, pour $i \in \{1,2\}$. L'opération d'addition $\oplus$ sur les usages est définie seulement pour des usages du même ensemble $U(i)$ pour un i donné. Le résultat est considéré comme défini si et seulement si ce dernier appartient au même ensemble de départ, *i.e.* pour $u, u' \in U(i)$, $u \oplus u'$ est défini si $u \oplus u' \in U(i)$ pour $i \in \{1,2\}$. On utilisera la même convention pour les opérations de soustraction et de comparaison.

Similairement à ce qui a été considéré pour Sπ (*cf.* section 2.5.4), le système de types qui sera présenté dans la section 3.4.3 utilisera des mots infinis de couples sur l'alphabet L, *i.e.* $u \in (L^2)^\omega$. Ces mots seront appelés *usages de canaux*. Dans la suite, on parlera d'usages uniformes s'ils sont de la forme u^ω ou non-uniformes quand ils sont de la forme $u \cdot v^\omega$ et $u \neq v$. De plus, on classifiera les usages principalement suivant deux critères :

affine qui contient au moins un 1.

neutre qui n'est pas *affine*.

Contrairement à ce qui a été étudié pour Sπ, toutes les familles d'usage considérées pour TAPIS conservent les contraintes sur les valeurs *affines*. Il n'est donc pas nécessaire d'avoir le troisième critère de classification pour les usages qui préservent l'affinité.

3.4.2 Types

TAPIS offrant la possibilité d'avoir du filtrage sur les valeurs, il est naturel de considérer dans le système de typage les types inductifs, les canaux et le

Types neutres (u est neutre)
$\kappa \quad ::= T_\infty(\kappa_1, \ldots, \kappa_n) \mid Ch_{u^\omega}(\kappa) \mid 1$
Types affines (u est affine)
$\lambda \quad ::= T_1(\sigma_1, \ldots, \sigma_n) \mid Ch_{u^\omega}(\sigma)$
Types uniformes
$\sigma \quad ::= \kappa \mid \lambda$
Types non-uniformes (v et w sont affines)
$\rho \quad ::= \sigma \mid Ch_{u \cdot u'^\omega}(\kappa) \mid Ch_{v \cdot w^\omega}(\sigma)$

TABLE 3.4 – Types

type *unit*.

Un type inductif est défini avec la donnée d'un usage de type et les différents cas de l'inductif. Par exemple, le type des listes d'éléments de type σ :

$$\begin{aligned} List_\alpha(\sigma) = \quad & nil \\ \mid \quad & cons \ \ of \ \ \sigma, \ List_\alpha(\sigma) \end{aligned}$$

La définition d'un type inductif précise un *usage* qui est dans l'ensemble $\{1, \infty\}$. Cet usage sert à signler l'usage que l'on va faire de la donnée stockée dans une valeur de ce type. Si l'usage est 1 alors le type sera affine et il ne pourra y avoir qu'un seul pointeur sur cette valeur. Sinon, le type sera neutre et la valeur pourra être utilisée sans aucune contrainte. Cela dit, une autre contrainte vient se rajouter à celles déjà citées pour garder une certaine cohérence dans la définition des types inductifs : un type inductif neutre ne peut pas utiliser de types affines.

Les types pour les canaux sont notés $Ch_u(\sigma)$. Cela représente un canal transportant des valeurs de type σ suivant l'usage de canal u. Similairement aux types inductifs, si σ est affine alors l'usage u doit être affine également.

Enfin, le type particulier singleton, noté 1, n'est habité que par une unique valeur, notée (). Ce type est aussi appelé *unit* dans d'autres languages.

Pour résumer les différentes combinaisons possibles, nous organiserons les types sur différents niveaux. Dans cette classification, nous aurons le groupe des types affines qui sera représenté par λ. Ensuite, un autre groupe de types représentera les types neutres, noté κ. Enfin, deux autres groupes σ et ρ représenteront (respectivement) les types uniformes et ceux qui ne le sont pas. Cette classification est résumée dans le tableau 3.4.

L'opération partielle d'addition sur les usages est étendue pour les types de la façon suivante :

$$C_u(\sigma) \oplus C_v(\sigma) = C_{u \oplus v}(\sigma)$$

où $C \in \{Ch, T\}$ et en supposant que $u \oplus v$ est bien défini.

Pour les besoins du système de types, nous définissons un contexte de typage, noté Γ, comme une fonction partielle des variables aux types où son domaine est noté $dom(\Gamma)$. L'opération d'addition $\Gamma_1 \oplus \Gamma_2$ sur les contextes est définie comme suit :

$$(\Gamma_1 \oplus \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \oplus \Gamma_2(x) & \text{si } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & \text{sinon si } x \in dom(\Gamma_1) \\ \Gamma_2(x) & \text{sinon si } x \in dom(\Gamma_2) \\ \uparrow & \text{sinon} \end{cases}$$

Nous utilisons la notation $(\Gamma_1 \oplus \Gamma_2)\downarrow$ pour dire que la somme est définie et que tous les types sont sommés correctement, suivant l'opération d'addition définie sur ces types, *i.e.* $\forall x \in dom(\Gamma_1) \cap dom(\Gamma_2)$, $(\Gamma_1(x) \oplus \Gamma_2(x))\downarrow$.

L'opération *shift* est également étendue aux contextes de telle sorte que $(\uparrow \Gamma)(x) = Ch_{\uparrow u}(\sigma)$ si $\Gamma(x) = Ch_u(\sigma)$ et $(\uparrow \Gamma)(x) = \Gamma(x)$ sinon. D'autre part, $\Gamma, x : \sigma$ dénote le contexte typage Γ auquel on aurait ajouté la paire $x : \sigma$. Enfin, un contexte Γ est dit *uniforme* (respectivement *neutre*) s'il associe que des types *uniformes* (respectivement *neutres*) aux variables. De plus, nous utilisons les notations $\varkappa(\Gamma)$ quand Γ est *neutre* et $\varsigma(\Gamma)$ quand Γ est uniforme.

3.4.3 Régles de typage

Le système de typage est décrit formellement dans le tableau 3.5. Nous supposons que chaque canal nouvellement créé vient avec son type. Nous utiliserons des jugements de typage de la forme $\Gamma \vdash P$, où P est un processus et Γ un contexte de typage, ou $\Gamma \vdash t : \sigma$, où t est un terme dont le type est σ et typé dans le contexte Γ. Le contexte de typage Γ donne une idée sur l'ensemble des actions qui peuvent être utilisées dans P ou t. Pour garantir la confluence, il suffit de limiter les communications aux deux cas suivants :

- Durant chaque instant, il est possible d'envoyer au plus une valeur et recevoir au plus une fois.
- Quand un processus offre un service, ce service doit être unique et un nombre arbitraire de clients doivent être capables d'envoyer leurs requêtes au service.

Par rapport au système de typage pour Sπ présenté dans le chapitre 2, il y a quelques différences qu'on doit souligner. Tout d'abord, l'ensemble des usages utilisés est différent pour les raisons évoquées dans la section 3.4.1. Ensuite, le contexte de typage qui type la continuation dans l'instant des processus répliqués doit être neutre. Cela est nécessaire pour pouvoir prouver le lemme de préservation du typage par réduction. Enfin, un processus qui ne dépend pas du temps doit être typé dans un contexte uniforme, *i.e.* qui associe des types neutres à chaque valeur.

Processus

$$(0)\ \frac{}{\Gamma \vdash 0} \qquad (\nu)\ \frac{\Gamma, a : Ch_{u^\omega}(\sigma) \vdash P \qquad \varsigma(Ch_{u^\omega}(\sigma))}{\Gamma \vdash \nu a : Ch_{u^\omega}(\sigma)\ P}$$

$$(\mathsf{par})\ \frac{\Gamma_i \vdash P_i \quad i = 1,2}{\Gamma_1 \oplus \Gamma_2 \vdash P_1 \mid P_2} \qquad (\mathsf{out})\ \frac{\Gamma_a \vdash a : Ch_{u \cdot v^\omega}(\sigma) \quad \Gamma_e \vdash e : \sigma \quad u(1) > 0}{\Gamma_a \oplus \Gamma_e \vdash \overline{a}\langle e \rangle}$$

$$(\triangleright_{\mathsf{in}})\ \frac{\Gamma_a \vdash a : Ch_{u \cdot v^\omega}(\sigma) \quad \Gamma_P, x : \sigma \vdash P \quad u = (0,1) \quad \uparrow(\Gamma_a \oplus \Gamma_P) \vdash Q}{\Gamma_a \oplus \Gamma_P \vdash a(x).P \triangleright Q} \qquad (\mathsf{in})\ \frac{\Gamma_a \vdash a : Ch_{u^\omega}(\sigma) \quad \Gamma_P, x : \sigma \vdash P \quad u = (0,1) \quad \varsigma(\Gamma_P, x : \sigma)}{\Gamma_a \oplus \Gamma_P \vdash a(x).P}$$

$$(!\triangleright_{\mathsf{in}})\ \frac{\Gamma_a \vdash a : Ch_{u \cdot v^\omega}(\sigma) \quad \Gamma_P, x : \sigma \vdash P \quad u = (\infty,1) \quad \uparrow \Gamma_Q \vdash Q \quad \varkappa(\Gamma_P)}{\Gamma_a \oplus \Gamma_P \oplus \Gamma_Q \vdash a(x).P\ !\triangleright Q} \qquad (!_{\mathsf{in}})\ \frac{\Gamma_a \vdash a : Ch_{u^\omega}(\sigma) \quad \Gamma_P, x : \sigma \vdash P \quad u = (\infty,1) \quad \varsigma(\sigma) \quad \varkappa(\Gamma_P)}{\Gamma_a \oplus \Gamma_P \vdash\ !a(x).P}$$

$$(\mathsf{m_c})\ \frac{\mathsf{k} : (\sigma_1, \ldots, \sigma_n) \to T_u(\sigma_1, \ldots, \sigma_m) \qquad \Gamma_1 \vdash e : T_u(\sigma_1, \ldots, \sigma_m) \quad \Gamma_2, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P_1 \qquad \Gamma_1 \oplus \Gamma_2 \vdash P_2}{\Gamma_1 \oplus \Gamma_2 \vdash [e \trianglerighteq \mathsf{k}(x_1, \ldots, x_n)]P_1, P_2}$$

Termes

$$(\mathsf{var_c})\ \frac{}{\Gamma, x : T_u(\sigma) \vdash x : T_u(\sigma)} \qquad (\mathsf{c})\ \frac{\Gamma_i \vdash e_i : \sigma_i \quad i = 1, \ldots, n \quad \mathsf{k} : (\sigma_1, \ldots, \sigma_n) \to T_u(\sigma_1, \ldots, \sigma_m)}{\Gamma_0 \oplus \Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash \mathsf{k}(e_1, \ldots, e_n) : T_u(\sigma_1, \ldots, \sigma_m)}$$

$$(\mathsf{var_{ch}})\ \frac{\Gamma_1(c) = Ch_{u \cdot v^\omega}(\sigma)}{\Gamma_1 \oplus \Gamma_2 \vdash c : Ch_{u \cdot v^\omega}(\sigma)}$$

TABLE 3.5 – Système de types affine pour TAPIS

3.4.4 Résultats

Le but est de montrer que tout processus typable est confluent. Avant de pouvoir s'attaquer à la démonstration de ce théorème, il faut s'assurer que le système de types a de bonnes propriétés. Les preuves des différents lemmes qui vont suivre seront commentées de façon succinte puisqu'elles feront l'objet d'un développement plus important dans la section 3.5.

Tout d'abord, il faut montrer quelques propriétés de base comme le lemme d'*affaiblissement* et la *substitution* qui vont être utilisées plus tard. Ensuite, il convient de démontrer la préservation du typage par réduction. Enfin, on énoncera et démontrera le théorème de confluence.

Lemme 57 (Affaiblissment)**.**

$$\frac{\Gamma \vdash U : T \qquad (\Gamma \oplus \Gamma') \downarrow}{\Gamma \oplus \Gamma' \vdash U : T}$$

PREUVE.

L'idée de la preuve est assez simple et repose sur deux éléments essentiels :

- L'opération d'addition $\oplus$ sur les types et contextes est associative et commutative.
- Les régles de typage sont formulées de telle sorte que l'on puisse ajouter des ressources sans casser le typage. Cela est particulièrement vrai pour la régle ($\mathsf{var_{ch}}$) où l'on cherche seulement à savoir si la ressource présente dans le contexte de typage suffit pour ce dont on a besoin. Les autres régles sont construites avec ce principe à l'esprit.

Par induction sur les régles de typage et en utilisant les faits mentionnés, on arrive à obtenir le résultat voulu.

□

Similairement, le lemme de *substitution* s'énonce comme suit :

Lemme 58 (Substitution)**.**

$$\frac{\Gamma, x : \rho \vdash U : T \qquad \Gamma' \vdash v : \rho \qquad (\Gamma \oplus \Gamma') \downarrow}{\Gamma \oplus \Gamma' \vdash [v/x]U : T}$$

PREUVE. Par induction sur la taille du terme U, et ensuite par cas sur la forme de U. Pour chaque cas, il faut distinguer 3 sous-cas suivant l'emplacement de la variable à substituer x. Elle peut se trouver soit à gauche (seulement), soit à droite (seulement), ou bien dans les deux.

□

Les régles de réduction sont définies modulo congruence structurelle. Nous devons donc montrer que deux processus équivalents peuvent être typés dans le même environnement de typage. Le lemme 59 énonce et prouve cela.

Lemme 59.

$$\frac{P \equiv Q}{\Gamma \vdash P \iff \Gamma \vdash Q}$$

PREUVE. Il suffit de remarquer que l'opérateur de composition parallèle ainsi que l'opération d'addition $\oplus$ sur les types sont associatifs et commutatifs. Le résultat est obtenu par induction sur la définition de la relation de congruence structurelle.

□

Enfin, le lemme de *préservation des types après réduction* dit que :

- Si P est typable dans un contexte Γ et se réduit à P' dans le même instant, alors P' est typable dans le même contexte.
- Si P est typable dans un contexte Γ et se réduit à P' à la fin de l'instant, alors P' est typable dans le contexte $\uparrow \Gamma$.

Cela peut être décrit formellement de la manière suivante :

Lemme 60 (Préservation des types par réduction)**.**

$$1)\ \frac{\Gamma \vdash P \qquad P \to P'}{\Gamma \vdash P'} \qquad\qquad 2)\ \frac{\Gamma \vdash P \qquad P \xrightarrow{N} P'}{\uparrow \Gamma \vdash P'}$$

PREUVE.

Par induction sur la transition activée, et par cas sur le processus P.

□

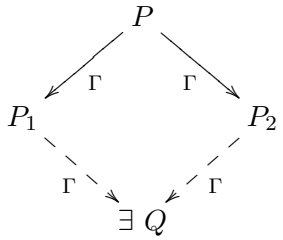
Pour simplifier l'énoncé de certains lemmes, nous allons introduire une notion de *transition typée*.

Définition 61 (Transitions typées)**.**

- *On écrit* $P \underset{\Gamma}{\longrightarrow} Q$ *si* $\Gamma \vdash P$ *et* $P \to Q$
- *On écrit* $P \underset{\Gamma}{\xrightarrow{N}} Q$ *si* $\Gamma \vdash P$ *et* $P \xrightarrow{N} Q$

Avant de pouvoir prouver la confluence des processus typés, nous devons montrer que les réductions commutent dans l'instant, et qu'elles sont déterministes à la fin de l'instant.

Lemme 62 (Commutation des réductions dans l'instant)**.** *Pour tout processus* P*, on a :*



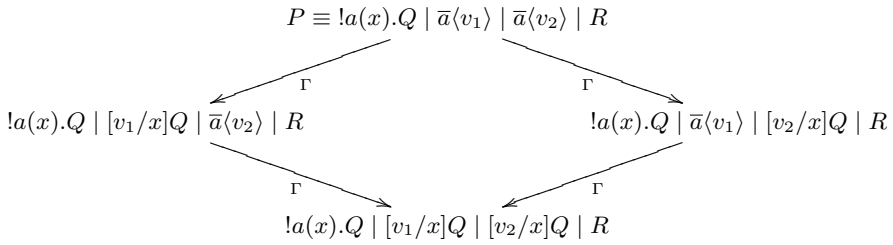
où la relation $\dashrightarrow_\Gamma = Id \cup \longrightarrow_\Gamma$ *et* Id *la relation identité.*

PREUVE. La preuve se fait par analyse de cas sur les transitions possibles. Une première passe permet d'éliminer des cas non problématiques *c-à-d* qui

ne sont pas en conflit. Par exemple, les régles (par) et $(=_1^{\mathsf{ind}})$ n'agissent pas sur les mêmes processus et ne sont donc pas en conflit. Parmi les cas restants et intéressants, on retouve celui où un processus P est capable de déclencher deux synchronisations différentes sur un même canal. Plusieurs sous-cas existent mais un seul est typable, celui où deux processus émettent vers le même canal et durant le même instant. Nous allons donc nous concentrer sur ce cas là en particulier où P ressemblerait (à équivalence près) à :

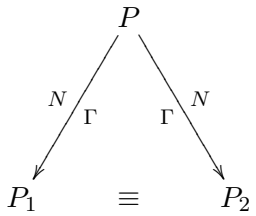
$$P \equiv !a(x).P \mid \overline{a}\langle v_1\rangle \mid \overline{a}\langle v_2\rangle \mid R$$

Grâce à la réception répliquée, il est toujours possible de recevoir un nombre arbitraire de valeurs, sans mettre en jeu la confluence. Il est donc possible de clore le diagramme en une étape de la façon suivante :



□

Lemme 63 (Confluence à la fin de l'instant)**.** *Pour tout processus P, on a :*



Preuve. Aucune communication n'est établie à la fin de l'instant. Il n'y a donc aucune source de conflit possible. En utilisant le fait que l'opérateur de composition parallèle est associatif et commutatif, il est possible d'en déduire que P_1 et P_2 sont équivalents.

□

En combinant les deux derniers lemmes, on arrive à démontrer que tout processus typable est confluent.

Théorème 64. *Tout processus typable est confluent.*

Preuve.

La preuve de ce théorème est directe en utilisant les lemmes 62 et 63.

□

3.5 Formalisation en Coq

Le lemme principal 60 qui permet de montrer que les processus typés sont confluents est le lemme de la préservation du typage par réduction. La preuve de ce dernier est surprenamment technique à cause des usages et des opérations d'addition qui y sont associées. Cela a motivé la formalisation complète de Tapis avec ses systèmes de réduction et de types, ainsi que la preuve de la préservation des types par réduction. Nous avons choisi l'assistant de preuve Coq[CDT10] pour ce travail. Puisque les détails et choix techniques du calcul et du système de types associés ont déjà été abordés dans la partie précédente, nous allons nous concentrer, ici, sur leur implantation en Coq.

3.5.1 Syntaxe

En Coq, la syntaxe de Tapis est définie de la façon suivante :

```
Inductive term :=
| Tindex : index-> term
| Tname : name-> term
| Tconstr : constr-> list term-> term.

Inductive process :=
| Pnull : process
| Ppar : process-> process-> process
| Psend : term-> term-> process
| Precv : term-> process-> process
| Precv_elsenext : term-> process-> process-> process
| Precv_dup : term-> process-> process
| Precv_dup_elsenext : term-> process-> process-> process
| Pnu : process-> process
| Pmatch : term-> constr-> nat-> process-> process-> process.
```

Un point clé dans la formalisation du calcul en Coq est la représentation des variables. On utilise l'approche *locally nameless* qui a été introduite par McKinna, Pollack, Gordon et McBride [MM04, MP99, Gor94]. Cette technique consiste à utiliser les indices de De Bruijn pour les variables liées (`Tindex`) et des noms pour les variables libres (`Tname`). Cela nous évite la majorité des problèmes liés à l'*alpha* conversion puisque des termes *alpha*-équivalents auront des représentations identiques. Comme décrit dans [ACP+08], cette technique utilise deux opérations de base sur les objets du langage (termes et processus) qui sont `open` et `close` où, intuitivement, l'une est le dual de l'autre :

`open : index -> B -> A -> B` Prend en argument un indice de De Bruijn k, deux termes u et t et remplace les occurrences de k dans t avec u. Cette opération n'a aucun effet si k n'est pas utilisé dans t. Nous utiliserons la notation `t@u` pour le terme `t` ouvert avec `u` à l'indice `0`.

`close : index -> name -> B -> B` Remplace le nom z dans t avec l'indice de De Bruijn k.

Bien que les opérations `open` et `close` soient, fondamentalement, des opérations de substitutions, on a décidé de garder la même notation utilisée dans [ACP+08] pour les différencier de l'implantation plus courante de la substitution qui gère autrement les indices de De Bruijn.

Par convention, nous supposerons que le premier argument de `open` ou `close`) est égal à zéro quand il est omis. Dans la suite, nous utilisons quelques autres fonctions implantées autour des fonctions `open` et `close`. Parmi ces fonctions, on peut citer `open_nb'` et `opens_p`. Elles itèrent sur un processus donné en argument en utilisant une liste de noms. La fonction `open_nb'` commence la première "ouverture" avec un indice 0 et elle finit avec n où $n+1$ est la taille de la liste de noms prise en argument. Quant à la fonction `opens_p`, elle effectue un traitement similaire à celui de `open_nb'` sauf qu'elle commence avec un indice k au lieu de 0.

On dispose de plus d'une troisième opération, plutôt standard, sur les termes et processus qui est la substitution. Son symbole de fonction dans la suite sera `subst : name -> A -> B -> B` où `name` est le type des noms, `A` le type de l'objet qui replacera le nom passé en premier argument et enfin `B` le type de l'objet où l'opération sera exécutée. Nous noterons `t[x:=u]` pour le terme `t` où le nom `x` est remplacé par `u`.

Les deux catégories syntaxiques dont on dispose ici sont les termes et les processus. Les termes sont des variables (`Tindex` et `Tname`) ou bien un constructeur (`Tconstr`) où le premier argument est un identificateur du constructeur choisi que l'on détaillera plus tard. Quant aux processus, on reprend les constructions définies dans 3.1. Il y a 4 constructeurs dont le nom commence avec `Precv`. Ils servent à représenter les 4 manières différentes de recevoir dans TAPIS. Ils sont décrits dans la table 3.2.

Exemple 65 (Horloge/Alarme). *On reprend l'exemple 53 de l'alarme simple. En Coq, le même processus s'écrirait de la façon suivante :*

```
let x := Tindex 0 in
let pause P := Pnu (Precv_elsenext x Pnull P) in
let star := Tconstr 0 nil in
let tac := Tname 0 in
Precv_dup tac (pause (Psend tac x // Psend x star))
```

Les opérations `open`, `close` et `subst` sont des opérations de base qui seront utilisées tout au long du développement. Il convient donc de vérifier que leur définition est correcte. Pour ce faire, nous avons prouvé ces quelques lemmes :

```
Class subst_props A B (* ... *) := {
  subst_close_fresh : forall (t:B) (x:name),
    x ∉ FV t -> close x t = t;
  subst_close_open : forall (x:name) (t:B),
```

```
    x ∉ FV t -> close x (t @ x) = t;
  subst_fresh : forall (x:name) u (t:B),
    x ∉ FV t -> t[x:=u] = t;
  subst_open : forall (x:name) (u:A) (t:B) (w:A),
    locally_closed u ->
    (t @ w)[x := u] = (t[x:=u]) @ (w[x:=u]);
  subst_open_var : forall (x:name) (u:A) (t:B) (y:name),
    locally_closed u -> x <> y ->
    (t @ (Tname y))[x := u] = (t[x := u]) @ (Tname y);
  subst_intro : forall (x:name) (u:A) (t:B),
    locally_closed u -> x ∉ FV t ->
    (t @ (Tname x))[x := u] = t @ u
}.
```

Par exemple, le lemme `subst_fresh` s'assure que l'opération `subst x u t` résulte en `t` si `x` n'apparaît pas comme variable libre dans `t`.

Les opérations `open`, `close` et `subst` partagent les mêmes propriétés sur les processus et sur les termes. Pour gérer efficacement ces catégories syntaxiques, nous avons fait appel à des classes de type telles que celles décrites dans [SO08]. Ainsi, toutes ces propriétés sont rassemblées dans une seule classe de types nommée `subst_props`. Le lecteur attentif remarquera la surcharge des noms et des notations utilisés.

3.5.2 Sémantique

Relation d'équivalence structurelle

La sémantique du calcul considéré utilise une relation d'équivalence structurelle ($\equiv$) définie sur les processus. Les détails sur cette relation peuvent être lus dans la section 3.2.

En Coq, cette relation est nommée `peq` et est implantée comme un inductif qui a cette forme :

```
Inductive peq : process -> process -> Prop :=
  | peq_neutral : forall P,
       P // Pnull ≡ P
  | peq_commut : forall P Q,
       P // Q ≡ Q // P
  | peq_assoc : forall P Q R,
       (P // Q) // R ≡ P // (Q // R)
  | peq_refl : forall P,
       P ≡ P
  | peq_trans : forall P Q R,
       P ≡ Q ->
       Q ≡ R ->
       P ≡ R
  | peq_sym : forall P Q,
       P ≡ Q ->
       Q ≡ P
  | peq_par : forall P P' Q Q',
       P ≡ P' ->
       Q ≡ Q' ->
```

```
        P // Q ≡ P' // Q'
  | peq_nu : forall P P' L,
        (forall a, a ∉ L -> P @ a ≡ P' @ a) ->
        Pnu P ≡ Pnu P'
  | peq_recv : forall P P' a L,
        (forall b, b ∉ L -> P @ b ≡ P' @ b) ->
        Precv a P ≡ Precv a P'
  | peq_prenex : forall P Q,
        (Pnu P) // Q ≡ Pnu (P // lift Q)
  | peq_exchange : forall P Q L,
        (forall a b, a ∉ L -> b ∉ L -> a <> b ->
          open_nb' (a::b::nil) P ≡ open_nb' (b::a::nil) Q
        ) ->
        Pnu (Pnu P) ≡ Pnu (Pnu Q)
where "P ≡ Q" := (peq P Q).
```

Certaines règles quantifient universellement sur un ensemble de noms, généralement appelé `L`, qui nous aide à choisir un nom suffisamment frais par rapport au terme (ou processus) manipulé. Cette technique est connue sous le nom de "quantification cofinie" [ACP+08]. À première vue, cette approche peut paraître différente de celle qui est plus standard et qui consiste à contraindre la nouvelle variable à ne pas apparaître dans l'ensemble des noms libres de l'objet de la règle, disons P. Cela dit, il n'en est rien puisqu'elles coïncident pour $L = fn(P)$. De plus, quand on construit une nouvelle dérivation à partir d'une autre, cette technique nous permet d'éviter de nous préoccuper des variables de renommage en élargissant assez l'ensemble de noms `L`.

On remarquera que la règle `peq_prenex` ne suit pas totalement les recommandations du style *locally nameless* puisqu'il est utilise explicitement la fonction `lift` pour manipuler les indices de De Bruijn présents dans le processus Q. En fait, cette règle est la seule qui utilise `lift` explicitement. À première vue, cela peut paraître pas très naturel, mais, heureusement, on peut l'expliquer assez facilement.

Nous supposons que les processus manipulés par `peq` sont bien formés (c'est à dire qu'ils ne contiennent pas d'indices de De Bruijn orphelins, sans lieur associé).

La règle d'équivalence comprend deux sens de conversion : celle qui fait passer le processus Q sous le lieur `Pnu`, et celle qui l'extrait.

Le premier sens est assez facile puisque Q est supposé être bien formé. Par contre, l'autre sens l'est moins. En particulier, on doit vérifier que les variables liées ne sont pas mélangées après extraction de Q. Cette vérification est garantie par l'utilisation de `lift`, qui nous assure que la variable d'indice $n+1$ aura l'indice n après extraction.

En fait, il existe un moyen de réparer cette règle en combinant `open` et `close`. Le problème se ramène donc à choisir l'opération qui est plus facile à composer avec `open`. Nous avons fait le choix de `lift`, car elle n'opère que sur l'indice des variables liées et ne change ni leur nombre, ni l'ensemble des

variables libres dans le terme considéré, contrairement à `close`.

Système de transitions

Le système de transitions utilise deux transitions définies de façon inductive :

– La relation `--->` précise les interactions possibles durant l'instant.

```
Inductive internal_transition
 : process -> process -> Prop :=
  | it_comp :
    forall P P' Q,
      P ---> P' -> P // Q ---> P' // Q
  | it_nu :
    forall P P' L,
      (forall x, x ∉ L -> P @ x ---> P' @ x) ->
      Pnu P ---> Pnu P'
  | it_ind_match :
    forall t constr arity tl P1 P2,
      do_match t constr arity = Some tl ->
      Pmatch t constr arity P1 P2 ---> opens_p 0 tl P1
  | it_ind_no_match :
    forall t constr arity P1 P2,
      do_match t constr arity = None ->
      Pmatch t constr arity P1 P2 ---> P2
  | it_eq :
    forall P P' Q Q',
      P ≡ P' ->
      P' ---> Q' ->
      Q' ≡ Q ->
      P ---> Q
  | it_sync :
    forall P a t,
      Precv a P // Psend a t ---> open P t
  | it_sync_elsenext :
    forall P Q a t,
      Precv_elsenext a P Q // Psend a t ---> open P t
  | it_bang :
    forall P a t,
      Precv_dup a P // Psend a t --->
        Precv_dup a P // open P t
  | it_bang_elsenext :
    forall P Q a t,
      Precv_dup_elsenext a P Q // Psend a t --->
        Precv_dup_elsenext a P Q // open P t
where "P ---> Q" := (internal_transition P Q).
```

Les règles qui ne font pas intervenir de lieurs sont assez simples et correspondent à la version "papier" de façon évidente.
Les autres règles font intervenir des fonctions pour manipuler les variables liées. À titre d'exemple, analysons le contenu des règles `it_nu` et `it_ind_match`.

`it_nu` La transition `Pnu P` vers `Pnu P'` est possible quand `P` peut se réduire vers un processus `P'`. Or, le processus `P` a un trou, une variable liée d'indice 0 et la transition suppose que les processus manipulés sont localement clos. Il faudra donc remplacer la variable liée d'indice 0 par un nom suffisamment frais.

En particulier, cette règle utilise une technique connue sous le nom de "quantification co-finie" [ACP+08]. La variable `L` est un ensemble fini de noms. Intuitivement, l'ensemble `L` représente tous les noms connus et les noms libres dans le terme ou processus considéré en particulier. Les conditions établies par la règles doivent être satisfaits quelque soit la valeur de `L`. Cela nous aide à garantir que les noms choisit sont suffisamment frais.

`it_ind_match` Il s'agit là de vérifier que le constructeur utilisé a bien l'arité indiquée, disons n. Puis, il suffit de continuer le calcul avec le processus $\mathtt{P}_1$ où toutes les variables liées dont l'indice est inférieur à n ont été remplacées par des noms tirés de la liste `tl`.

– Quant à la relation `-=->`, elle sert à migrer le calcul de l'instant courant vers l'instant suivant.

```
Inductive tick_transition : process -> process -> Prop :=
  | tt_elsenext :
    forall a P Q, Precv_elsenext a P Q -=-> Q
  | tt_dup_elsenext :
    forall a P Q, Precv_dup_elsenext a P Q -=-> Q
  | tt_next :
    forall a P, Precv a P -=-> Precv a P
  | tt_dup_next :
    forall a P, Precv_dup a P -=-> Precv_dup a P
  | tt_nu :
    forall P P' L,
      (forall x, x ∉ L -> P @ x -=-> P' @ x) ->
      Pnu P -=-> Pnu P'
  | tt_send :
    forall a t, Psend a t -=-> Pnull
  | tt_par :
    forall P Q, (forall R, ~ (P // Q ---> R)) ->
    forall P' Q',
      P -=-> P' -> Q -=-> Q' -> P // Q -=-> P' // Q'
where "P -=-> Q" := (tick_transition P Q).
```

Ces règles sont plus simples que celles introduites par l'autre relation de transition. Seulement la règle `tt_nu` fait intervenir des variables fraîches de la même manière que précédemment.

3.5.3 Système de types

La formalisation correcte et efficace du système de types conçu pour Tapis passe par la conception de plus petites composantes importantes pour

représenter les types, les usages et les opérations d'addition autour de ces structures. Nous allons détailler les choix faits pour chaque partie.

Usages

Les usages sélectionnés sont représentés dans la figure 3.1. Leur formalisation dans Coq est en fait assez simple :

- un type inductif pour chaque famille d'usages :

```
Inductive uk1 := U81 | U80.
Inductive uk2 := U11 | U01 | U10 | U00.
```

- un type inductif pour représenter les familles d'usages :

```
Inductive usage_family := UK1 | UK2.
```

- un type dépendant :

```
Definition usage x := match x with
  | UK1 => uk1
  | UK2 => uk2
end.
```

La fonction `usage` est utilisée à chaque fois qu'une fonction ou un lemme effectue un calcul ou exprime une propriété sur les usages. Cela nous permet d'écrire des énoncés assez complexes de façon simple et flexible. Ce qui suit est un exemple d'utilisation de `usage`.

```
Definition foo {uk} (x y : usage uk) := ...
```

Types

Les types sont organisés en niveaux, comme le montre 3.4. Chaque niveau est codé par un type inductif en Coq. Une première couche permet de formaliser les objets de base.

```
Inductive U := U1 | U8.
Inductive sigma :=
  | Channel   : forall {uk}, channel_usage uk -> sigma -> sigma
  | User_type : nat -> sigma.
Record user_type := {
  UT_usage : U;
  UT_args : list sigma;
  UT_constructors : list (list sigma)
}.
```

Il est possible de marquer la différence entre les divers genres de types : *neutre*, *affine*, *uniforme* ou encore *général*. Le genre *neutre* est codé de la manière suivante en Coq :

```
Inductive neutral {i : raw_envi} : sigma -> Prop :=
| isn_channel :
  forall uk (su : channel_usage uk) sigma,
    su_is_neutral su = true ->
```

```
    neutral sigma ->
    neutral (Channel su sigma)
| isn_user :
  forall t T,
    nth_error i t = Some T ->
    UT_usage T = U8 ->
    neutral (User_type t)
.
```

Les autres genres sont définis de façon analogue.

Nous utilisons une structure particulière pour référencer les types connus. Cette structure porte le nom de `envi` et contient, entre autres, la liste des types définis par l'utilisateur. Ainsi, pour représenter un type dans cette liste, on utilisera `User_type n` où n est la position du type dans la liste en question. La structure de cet environnement comporte une première composante `envi_raw` qui est une liste de types définis par l'utilisateur qui stocke, pour chacun d'entre eux, les informations suivantes :

`UT_usage` l'usage de la donnée dans $\in \{1, \infty\}$. Les types déclarés avec un usage 0 ne peuvent pas être utilisés. Nous avons donc décidé de ne pas les considérer. L'usage indique combien de fois une valeur de ce type peut être partagée ou utilisée, où 1 signifie *au plus une fois* et ∞ *un nombre arbitraire de fois.*

`UT_args` la liste des arguments du type.

`UT_constructors` la liste des constructeurs en précisant leurs arguments. Chaque constructeur est identifié par sa position dans la liste.

La hiérarchie des types a été conçue de telle sorte que les contraintes de déterminisme ou d'affinité ne puissent pas être violées. Or, la structure de `envi_raw` n'impose rien qui puisse nous prémunir de ces situations dégénérées. Pour ce faire, nous avons rajouté à la structure `envi` le nécessaire. En particulier, nous supposerons que les propriétés suivantes sont toujours vraies :

– *Les types affines n'utilisent que des types affines ou neutres.* Remarquez que l'inverse n'est pas vrai.

```
envi_wf0 : forall T,
  In T envi_raw ->
  UT_usage T = U1 ->
  (forall t, In t (UT_constructors T) ->
     Forall affine t)
  /\ Forall affine (UT_args T);
```

– *Les types neutres peuvent utiliser des types neutres seulement.*

```
envi_wf1 : forall T,
  In T envi_raw ->
  UT_usage T = U8 ->
  (forall t, In t (UT_constructors T) ->
     Forall neutral t)
  /\ Forall neutral (UT_args T);
```

– *Les types définis par l'utilisateur sont uniformes.*

```
envi_wf2 : forall T,
  In T envi_raw ->
  (forall t, In t (UT_constructors T) ->
     Forall uniform t)
  /\ Forall uniform (UT_args T)
```

Additions

Les usages et types précédemment définis sont des objets qui peuvent représenter une certaine quantité de ressources existantes. Quand on raisonne de manière compositionnellee sur les processus, il est souvent nécessaire de pouvoir additionner ces ressources (tant que cela est possible). Il a donc été nécessaire d'implanter des opérations d'addition dont le comportement a été décrit dans 3.4.1 et 3.4.2. Ces opérations sont associatives, commutatives et partielles. À notre connaissance, il n'existe pas de moyen canonique de gérer de telles opérations dans Coq. Nous avons considéré 3 approches (disons, pour les additions de valeurs de type `A`) :

1. des fonctions binaires retournant une valeur de type `option A`. Le type de ces fonctions est donc `A -> A -> option A`.
2. des relations ternaires de type `A -> A -> A -> Prop` où le troisième argument est le résultat supposé du calcul.
3. des fonctions binaires opérant sur des `option A` : `option A -> option A -> option A`.

La première solution semble la plus naturelle pour des fonctions partielles mais l'asymétrie entre les entrées et sorties la rend difficile à composer. En particulier, il est nécessaire de nommer explicitement les résultats intermédiaires pour exprimer l'associativité. Cela nous mène à des formulations de ce style :

```
forall x y z r1 r2,
  f x y = Some r1 /\ f r1 z = Some r2 ->
  exists r3, f y z = Some r3 /\ f x r3 = Some r2
```

Cette formulation de l'associativité est particulièrement pénible à manipuler dans les preuves. Cela nous a forcé à chercher une meilleure façon de la représenter.

La deuxième solution mène à des définitions proches de celles écrites sur papier. Son utilisation est aussi facilitée grâce aux tactiques Coq `induction` et `inversion`. Cependant, cette solution a les mêmes défauts que la première solution, par rapport à la difficulté de les composer.

Enfin, la troisième solution, celle adoptée dans notre développement, est pratique et résout le problème de composition. Cela nous permet de tirer bénéfice de techniques de raisonnement sur des opérations associatives et

commutatives développées en Coq [BP11]. En particulier, nous utilisons la tactique `aac` pour résoudre des équations de réécriture complexes.

Pour chaque type d'objet (usage, usage de canal, type, ...), nous avons implanté l'opération d'addition associée en suivant la signature de fonction donnée dans la troisième solution. Pour uniformiser ce travail, nous avons créé une classe qui requiert les propriétés d'associativité et de commutativité et transformé les implantations faites en instances de cette classe. Cela nous permet d'utiliser des noms et symboles de fonctions génériques ($\oplus$ et $+$) pour tous les types et de façon transparente. De plus, nous fournissons une preuve générique de l'équivalence de cette formulation avec celle qui utilise des fonctions ternaires.

Règles de typage

La traduction des règles de typage en Coq est directe. Leur écriture a été facilitée grâce au travail qui a été fait en amont sur les additions et les types. Des notations Coq nous permettent même de retrouver la notation utilisée sur papier pour les jugements de typage.

Les contextes de typage ont été implantés à l'aide de la structure d'ensemble `FMap` de Coq où les clés sont des `name` et les valeurs des `sigma`.

3.5.4 Résultats obtenus

Le résultat principal prouvé en Coq est la préservation des types par réduction. Dans la suite, nous mentionnerons les lemmes principaux utilisés pour aboutir au résultat final. Le cheminement des preuves ressemble à celui qu'on aurait suivi pour faire les preuves sur papier. Néanmoins, il faut souligner que le plus important dans ce travail a été la représentation et gestion des lieurs, la gestion des opérations d'addition et la représentation des usages et types.

Dans les règles de typage, on suppose manipuler des termes et processus localement clos. Plus précisément, nous voudrions que tout objet typable n'ait aucune variable de De Bruijn orpheline et que les noms libres dans le terme soient liés dans le contexte de typage. Les lemmes suivants s'assurent de la validité des propriétés mentionnées.

```
Lemma typ_term_locally_closed : forall Γ t s,
  Γ ⊢ t : s -> locally_closed t.
Lemma typ_process_locally_closed : forall Γ P,
  Γ ⊢ P -> locally_closed P.
```

Ces deux lemmes sont prouvés par induction sur les jugements de typage. Dans la suite, les lemmes de renommage seront nécessaires.

```
Lemma typ_term_renaming : forall (t:term) n Γ s ty,
  forall x, x ∉ dom Γ -> x ∉ FV t -> Γ,x:s ⊢ (t @ x) ty ->
  forall y, y ∉ dom Γ -> y ∉ FV t -> Γ,y:s ⊢ (t @ y) ty.
Lemma typ_renaming : forall Γ s (P:process),
```

```
forall x, x ∉ dom Γ -> x ∉ FV P -> Γ,x:s ⊢ P @ x ->
forall y, y ∉ dom Γ -> y ∉ FV P -> Γ,y:s ⊢ P @ y.
```

Une dernière étape avant d'atteindre le théorème final est la preuve du lemme de substitution. À son tour, ce dernier a besoin des lemmes suivants :

- Le lemme d'`affaiblissment` déclare qu'enrichir le contexte de typage avec de nouvelles ressources ou de nouveaux noms ne casse pas le typage :

  ```
  Lemma weakening : forall Γ1 Γ2 Γ3 P,
    !Γ1+!Γ2 === !Γ3 -> Γ1 ⊢ P -> Γ3 ⊢ P.
  ```

  Notez ici que + represente l'opération partielle d'addition pour les environnements. Elle tient en compte des ressources. Il ne suffit pas de fusionner les environnements pour retrouver leur somme, comme nous avons l'habitude de faire dans d'autres systèmes. Dans notre contexte, il est nécessaire de faire l'addition point à point.
- Le lemme de *renforcement* déclare que la suppression de ressources non utilisées du contexte de typage ne casse pas le typage :

  ```
  Lemma strengthening_p: forall n, forall P, p_size P<=n ->
    forall Γ x s,
    x ∉ FV P -> x ∉ dom Γ -> (Γ,x:s) ⊢ P -> Γ ⊢ P.
  ```

- Des processus équivalents doivent être typables dans le même contexte de typage :

  ```
  Instance typ_equiv :
    Proper (eq ==> peq ==> iff) typ_process.
  ```

  En d'autres termes, $\forall \Gamma\ P_1\ P_2,\ P_1 \equiv P_2 \Longrightarrow \Gamma \vdash P_1 \iff \Gamma \vdash P_2$.

En utilisant ces lemmes, on peut enfin prouver le résultat final décrit dans le lemme 66.

Lemme 66 (Préservation des types par réduction en Coq)**.**

```
Lemma subject_reduction_1 : forall P P' Γ,
  Γ ⊢ P -> P ---> P' -> Γ ⊢ P'.
Lemma subject_reduction_2 : forall P P' Γ,
  Γ ⊢ P -> P -=-> P' -> ↑Γ ⊢ P'.
```

Ce développement en Coq est une partie importante de cette thèse. Il a nécessité un an/homme de travail et représente aujourd'hui 12k lignes de code. Cette durée se place largement au-dessus des premières estimations de durées faites au commencement du travail [Ben06]. Il a fallu tester plusieurs représentations pour les termes et diverses structures pour bien modéliser les additions de façon efficace. Les tailles et contenus des divers modules Coq développés dans ce travail sont résumés dans le tableau 3.6. Il est disponible en ligne [DG11] sous la licence libre CeCILL [cec06] 2.0. La consultation de la documentation du développement en Coq est plus facile en ligne. Pour cette raison là, on n'a pas éprouvé le besoin de l'inclure en annexe dans ce document.

Fichier	Lignes	Contenu
`CoreLibrary.v`	343	Extensions à la librairie standard
`Syntax.v`	743	Définitions des termes, processus et environnements
`Additions.v`	2109	Définitions des additions partielles
`Names.v`	2719	Constructions relatives à la technique “Locally nameless”
`Typing.v`	3014	Lemmes auxiliaires à propos du typage et gestion des noms
`Tpi.v`	3071	Lemmes principaux

TABLE 3.6 – Coq modules

Chapitre 4

Conclusion

Dans cette thèse, nous avons étudié le problème du déterminisme dans le contexte de calculs concurrents synchrones. Nous avons mené cette étude dans deux calculs différents. Le premier est une extension du modèle *SL* qui intégre les fonctionnalités de passage de noms et de valeurs empruntés au π-calcul. Le deuxième est une extension du calcul TCCS avec des canaux riches.

Pour Sπ, nous avons introduit une nouvelle sémantique basée sur une notion de bisimulation étiquetée qui est associée à un nouveau système de transitions étiquetées et nous avons montré que notre bisimulation coïncidait avec celle qui a été introduite originellement pour Sπ. Dans les deux contextes, nous avons conçu des systèmes de types pour garantir le déterminisme. Puis, nous avons formalisé le système de types pour Tapis et nous avons montré qu'il avait de bonnes propriétés mathématiques.

Dans cette conclusion, nous allons présenter quelques perspectives de recherche en lien avec les travaux présentés dans ce manuscrit.

Implantation de l'analyse statique pour le déterminisme Nous avons décrit les conditions à satisfaire pour montrer que le comportement d'un programme est déterministe. La preuve du premier système repose sur une théorie d'équivalence basée sur la notion de bisimulation que nous avons introduits. Cependant, comme nous l'avons montré dans le chapitre 3, il est possible de présenter une version du système qui ne dépend plus de la bisimulation. Cela représente un avantage considérable et permet l'implantation de notre analyse statique dans des langages réactifs ou synchrones comme *ReactiveML* [MP05].

Futurs développements dans la formalisation en Coq Le développement en Coq présenté dans la section 3.5 formalise la preuve de préservation de typage par réduction. Nous aimerions arriver à formaliser la preuve de confluence dans Tapis en Coq pour compléter le travail. Un autre axe

de développement serait l'extraction, en code certifié, d'un vérificateur de types pour le systèmes de typage de Tapis. Cela pourrait simplifier l'intégration d'une telle analyse statique dans un langage existant en ayant plus de garanties.

Une nouvelle approche pour la représentation des noms L'approche "locally nameless" utilisée dans notre développement Coq pour représenter les variables est assez intéressante et pratique à l'utilisation. Cependant, elle comporte quelques problèmes de compositionnalité puisqu'elle ne permet pas, en l'état, d'*ouvrir* un terme de façon partielle. En d'autres termes, si un lieur liant n variables se trouve à la tête d'un terme, l'approche actuelle est de trouver n noms frais (par rapport à ce terme) et d'ouvrir ce dernier pour l'explorer. Cependant, on ne peut pas appliquer une ouverture de m noms, où $m < n$. Permettre cette application partielle pourrait donner plus de flexibilité lors de la manipulation des fonctions `open` et `close`. Cette extension ne changera pas la puissance de l'approche mais pourrait améliorer l'expérience de son utilisateur.

Bibliographie

[AB84] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theor. Comput. Sci.*, 30 :91–131, 1984.

[ACP+08] B. Aydemir, A. Charguéraud, B.C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. *ACM SIGPLAN Notices*, 43(1) :3–15, 2008.

[ACS98] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Theor. Comput. Sci.*, 195(2) :291–324, 1998.

[AD07] R. M. Amadio and M. Dogguy. Determinacy in a synchronous pi-calculus. In Y. et al. Bertot, editor, *From semantics to computer science : essays in honor of Gilles Kahn*. Cambridge University Press, 2007.

[AD08] R. M. Amadio and M. Dogguy. On affine usages in signal-based communication. In G. Ramalingam, editor, *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2008.

[ADS+06] P. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing safe, reliable systems using SCADE. *Leveraging Applications of Formal Methods*, pages 115–129, 2006.

[Ama07] R. M. Amadio. A synchronous pi-calculus. *Information and Computation*, 205(9) :1470–1490, 2007.

[Ama09] R. M. Amadio. On convergence sensitive bisimulation and the embedding of CCS in timed CCS. In *Electronic Notes in TCS 242*, 2009.

[BC01] G. Boudol and I. Castellani. Noninterference for concurrent programs. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 382–395. Springer, 2001.

[BDS96] F. Boussinot and R. De Simone. The SL synchronous language. *IEEE Trans. on Software Engineering*, 22(4) :256–266, 1996.

[Ben06] N. Benton. Machine obstructed proof. In *Workshop on Mechanizing Metatheory*, 2006.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language. *Science of computer programming*, 19(2) :87–152, 1992.

[BIM95] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1) :232–268, 1995.

[Bou91] F. Boussinot. Reactive C : An extension of C to program reactive systems. *Software : Practice and Experience*, 21(4) :401–428, 1991.

[Bou92] G. Boudol. Asynchrony and the pi-calculus. *Rapport de recherche 1702, INRIA, Sophia-Antipolis*, 1992.

[BP11] M. Braibant and D. Pous. Tactics for reasoning modulo AC in Coq. In *First International Conference on Certified Programs and Proofs*, 2011.

[BS98] F. Boussinot and J.F. Susini. The SugarCubes tool box : a reactive java framework. *Software : Practice and Experience*, 28(14) :1531–1550, 1998.

[CDT10] The Coq Development Team. The Coq proof assistant reference manual, version 8.3. `http://coq.inria.fr/refman/`, 2010.

[cec06] Contrat de licence de logiciel libre CeCILL. `http://www.cecill.info/licences/Licence_CeCILL_V2-en.html`, 2006.

[Des00] J. Despeyroux. A higher-order specification of the π-calculus. *Theoretical Computer Science : Exploring New Frontiers of Theoretical Informatics*, pages 425–439, 2000.

[DG11] M. Dogguy and S. Glondu. Confluence in TAPIS. `http://www.pps.jussieu.fr/~dogguy/?research/tapis`, 2011.

[DW01] Sangiorgi D. and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[FG98] C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 844–855. Springer, 1998.

[Gir87] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50 :1–102, 1987.

[GLG87] T. Gautier and P. Le Guernic. SIGNAL : A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987.

[Gor94] A. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. *Higher Order Logic Theorem Proving and Its Applications*, pages 413–425, 1994.

[GS96] J.F. Groote and MPA Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1-2) :47–81, 1996.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[HG99] L. Henry-Gréard. Proof of the subject reduction property for a π-calculus in Coq. 1999.

[Hir97] D. Hirschkoff. A full formalisation of pi-calculus theory in the calculus of constructions. In E. L. Gunter and A. P. Felty, editors, *TPHOLs*, volume 1275 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 1997.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HP85] D. Harel and A. Pnueli. *On the development of reactive systems.* Weizmann Institute of Science, Dept. of Computer Science, 1985.

[HR02] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Trans. Program. Lang. Syst.*, 24(5) :566–591, 2002.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *ECOOP*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991.

[HY95] K. Honda and N. Yoshida. On reduction-based process semantics. *Theor. Comput. Sci.*, 151(2) :437–486, 1995.

[JS90] C.-C. Jou and S. A. Smolka. Equivalences, congruences, and complete axiomatizations for probabilistic processes. In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 367–383. Springer, 1990.

[Kob98] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2) :436–482, 1998.

[KPT96] N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the pi-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 358–371. ACM, 1996.

[KSW06] N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the *pi*-calculus. In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2006.

[LL90] L. Lamport and N. A. Lynch. Distributed computing : Models and methods. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, pages 1157–1199. 1990.

[Mel94] T. F. Melham. A mechanized theory of the pi-calculus in HOL. *Nord. J. Comput.*, 1(1) :50–76, 1994.

[Mil80] R. Milner. *A calculus of communicating systems*, volume 92. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[Mil99] R. Milner. *Communicating and mobile systems : the pi-calculus*, volume 13. Cambridge University Press, 1999.

[MM04] C. McBride and J. McKinna. Functional pearl : I am not a number–I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9. ACM, 2004.

[MP99] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3) :373–409, 1999.

[MP05] L. Mandel and M. Pouzet. ReactiveML : a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 82–93. ACM, 2005.

[New42] M. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2) :223–243, 1942.

[NS97] U. Nestmann and M. Steffen. Typing confluence. In *Second International ERCIM Workshop on Formal Methods in Industrial Critical Systems*, pages 77–101, 1997.

[PS96] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5) :409–453, 1996.

[PW97] A. Philippou and D. Walker. On confluence in the pi-calculus. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 314–324. Springer, 1997.

[SO08] M. Sozeau and N. Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag.

[Wad93] P. Wadler. A taste of linear logic. *Mathematical Foundations of Computer Science 1993*, pages 185–210, 1993.