HAL
open science

Composants et Typage

Michael Lienhardt

To cite this version:

Michael Lienhardt. Composants et Typage. Informatique et langage [cs.CL]. Université Joseph-Fourier - Grenoble I, 2010. Français. NNT : . tel-00749351v3

HAL Id: tel-00749351

https://theses.hal.science/tel-00749351v3

Submitted on 16 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 Août 2006

Présentée par

Michael Lienhardt

Thèse dirigée par **Jean-Bernard Stefani**
et codirigée par **Alan Schmitt**

préparée au sein **LIG**
et de **Mathématiques Sciences et Technologies de l'Information**

Composants et Typage

Thèse soutenue publiquement le **07 Septembre 2010**,
devant le jury composé de :

Monsieur Gerard Boudol
Chercheur Emérite à l'INRIA Sofia-Antipolis, Président
Monsieur Mario Coppo
Professeur des Universités à Turin (Italie), Rapporteur
Monsieur Vasco Vasconcelos
Professeur des Universités à Lisbonne (Portugal), Rapporteur
Madame Florence Maraninchi
Professeur des Universités à Grenoble INP/ENSIMAG, Examinatrice
Monsieur Jean-Bernard Stefani
Directeur de recherche à L'INRIA Rhône-Alpes, Directeur de thèse
Monsieur Alan Schmitt
Chargé de recherche à L'INRIA Rhône-Alpes, Co-Directeur de thèse

Résumé

Depuis le milieu des années 1990, de nombreux modèles à composant ont vu le jour, comme Fractal développé par France Télécom et l'INRIA, OSGi développé par IBM, Click par le MIT, etc. Ces différents modèles ont tous pour but de pouvoir construire des programmes très larges et complexes, comme Eclipse basé sur OSGi ou la chaîne de compilation Fractal. Bien que ces modèles facilitent la conception de gros programmes, il est néanmoins toujours aisé de commettre des erreurs pouvant causer la levée d'une exception ou l'arrêt brutale du programme. Dans ce travail de thèse, nous nous intéressons à deux aspects de la validation des modèles à composant : nous définissons tout d'abord deux systèmes de types permettant de s'assurer qu'aucune erreur grave n'est présente dans un assemblage statique et typé ; nous étudions dans un simple langage de programmation Oz/K comment il est possible de décrire formellement l'interaction entre les composants et du code fonctionnel. Ce dernier langage nous a aussi permis d'étudier certaines commandes de manipulation d'assemblages qui seraient plus adaptées pour prouver des propriétés sur les programmes à base de composants.

Mots clefs : Composants, Types, Inférence, Langage de Programmation

Summary

Since the mid-1990s, many component models have emerged, such as Fractal developed by France Telecom and INRIA, OSGi developed by IBM, Click by MIT, etc. These models are all designed to build very large and complex programs, such as Eclipse based on OSGi or the Fractal toolchain. Although these models facilitate the design of large programs, it is nevertheless always easy to make mistakes that can rise exceptions or cause the sudden stop of the program. In this thesis, we focus on two aspects of component models validation : First we define two types systems to ensure that no serious error is present in a static and typed program ; we also study in a simple programming language Oz/K how it is possible to formally describe the interaction between components and functional code. This language also allow us to study some commands to manipulate component assemblies which would be more appropriate to prove properties on component based programs.

Keywords : Components, Types, Inference, Programming Language

Remerciements

Le document ici présent n'aurait jamais vu le jour sans cette foule de personnes qui m'ont guidées, qui m'ont soutenu dans les moments difficiles, qui m'ont enseigné tant d'élements utiles et d'annecdotes intéressantes durant ces presque quatre années de thèse. Mes remerciements vont en premier à jean-Bernard Stefani et Alan Schmitt, mon directeur et co-directeur de thèse, qui m'ont accueilli au sein de Sardes, avec pour sujet un article de sept pages, et qui m'ont guidé pour le faire fructifier en ce document de plus de deux cent pages. Les longues discussions que j'ai eu avec eux, leurs conseils, leur expérience et intuition ont été pour moi une source intarissable de nouveaux articles à étudier, de nouvelles pistes à explorer. Je remercie toutes les personnes qui fait l'honneur d'accepter de faire partie de mon jury de thèse. Je remercie en particulier Vasco Vasconcelos et Mario Coppo qui ont accepté l'immense tâche de relire intégralement ce document. Leurs commentaires et suggestions furent précieuses pour rendre ce document aussi clair que possible.

Mes remerciements vont aussi à tous ces gens qui ont fait de Sardes une équipe accueillante et chaleureuse, ainsi qu'à l'environnement exceptionnel de travail qu'est l'INRIA. Maude et Diane pour leur gentillesse et leur efficacité ; Marie pour ses sourires et ses chocolats chauds. Willy, Fabien et Serguei ont su élever le tetrinet au rang d'une institution ; Sylvain a su donné une définition précise à la notion de geek. Ludovic fut une inspiration pour la cuisine ; Baptiste fut un compagnon incroyable de balades en vélo dans les belles montagnes alpines. Merci à eux, et merci à tous mes amis et famille qui sont venus de loin pour assister à ma soutenance de thèse.

Table des matières

1 Introduction **6**
1.1 Le contexte de notre travail 6
1.2 Structure de ce document 7

2 Routage structurel **10**
2.1 Le calcul 10
2.1.1 Exemples et discussion 15
2.2 Le système de type 22
2.2.1 La syntaxe 24
2.2.2 Le typage des messages et des composants 25
2.2.3 Exemples 31
2.3 Discussion 36
2.4 Conclusion 41

3 Routage sémantique **42**
3.1 Le calcul 44
3.1.1 Exemples et discussion 47
3.2 Le système de type 59
3.2.1 La syntaxe 61
3.2.2 Les substitutions 65
3.2.3 Les règles de typage 70
3.2.4 Exemples et discussion 73
3.3 Discussion 84
3.4 Conclusion 88

4 Inférence et implémentation **90**
4.1 L'inférence de type 90
4.1.1 La syntaxe des contraintes 91
4.1.2 La génération des contraintes 92
4.1.3 La résolution de contrainte 94
4.1.4 Les propriétés de base de l'inférence 99
4.2 L'implémentation de l'inférence de type 101
4.2.1 Click et Dream 101
4.2.2 La structure de notre implantation 103

4.2.3 Le programme central d'inférence 110
4.2.4 Nos résultats 115
4.3 Conclusion 117

5 Le langage Oz/K **120**
5.1 La syntaxe 122
5.1.1 La partie Oz du langage 122
5.1.2 La partie K du langage 125
5.2 La sémantique opérationnelle du langage 130
5.2.1 La relation de réduction 132
5.3 Exemples 147
5.3.1 Portes et isolation 148
5.3.2 Distribution 149
5.3.3 Mobilité de code 152
5.3.4 Gestion des erreurs 155
5.4 Discussion 156
5.4.1 Isolation et implémentation. 157
5.4.2 Composants 158
5.4.3 Langage 159
5.5 Conclusion 161

6 Conclusion 163
6.1 Contributions 163
6.2 Perspectives 164

A Annexe du chapitre 2 175
A.1 Les types principaux 175
A.2 L'indécidabilité de l'inférence de type 178

B Annexe du chapitre 3 184
B.1 Les propriétés du sous-typage 184
B.1.1 La forme normale d'une dérivation de type 185
B.2 L'extraction de type 189
B.3 Correction 192
B.4 Stabilité 193

C Annexe du chapitre 4 196
C.1 Constraint Generation 196
C.1.1 Correction 196
C.1.2 Completeness 197
C.2 Constraint solving 201
C.2.1 Constraint Satisfiability 203

Chapitre 1

Introduction

1.1 Le contexte de notre travail

Les composants ont été introduit comme un nouveau paradigme de programmation vers le milieu des années 1990, principalement avec les workshops WCOP (pour Workshop of Component-Oriented Programming). Ce nouveau paradigme a été défini dans le but d'étendre le modèle à objet aux systèmes dynamiques et extensibles [107]. En effet, bien qu'étant très pratiques et populaires, les objets ne font pas mention explicite des ressources et autres objets qu'ils utilisent pour s'exécuter, et il est donc difficile d'adapter un programme à un nouvel environnement d'exécution, ou d'extraire une partie de programme pour l'inclure dans un autre [8]. Les composants sont construits à partir de trois notions :

1. Les *interfaces entrantes* d'un composant définissent de quelles ressources ce dernier a besoin pour s'exécuter convenablement ;
2. Les *interfaces sortantes* donnent quelles sont les ressources mises à disposition par le composant ;
3. Les *liaisons* relient les interfaces sortantes aux interfaces entrantes, satisfiant ainsi de manière explicite les besoins des composants.

Avec cela, les composants sont généralement considérés comme des boîtes que l'on peut assembler et désassembler aisément, simplement en modifiant une liaison.

Il existe aujourd'hui une grande variété de modèles utilisant les composants, chacun proposant sa propre définition de ce qu'est un assemblage de composants, et comment il peut être modifié. Nous avons par exemple le modèle Fractal [17] qui structure hiérarchiquement ses assemblages en un arbre, et qui offre de plus des primitives de modification des liaisons et des interfaces d'un composant durant son exécution. Le modèle OSGi est assez populaire, est basé sur une notion de service remplaçant la notion d'interface et gère de manière automatique l'élaboration des liaisons entre composants par la résolution de contraintes définissant quels services sont attendus par quels composants. Bien

que le modèle suggère que les liaisons peuvent être modifiées dynamiquement (c'est à dire durant l'exécution du programme), cette capacité n'est pas encore implémentée. Nous avons d'autres modèles plus spécifiques, comme COM [32] ou Java Beans [106] qui sont plus tournés vers la conception de services internet. De fait, il existe de nombreux outils à base de composants qui permettent à partir de composants élémentaires, de construire des protocoles de communication pouvant être très complexes, comme Appia [79], Click [80], Coyote [12] ou Dream [65].

La motivation originale de ce travail de thèse vient du concepteur de Dream qui remarquait que si la conception de composants simples est relativement aisé, leur assemblage en un programme complexe est source d'un grand nombre d'erreurs souvent assez difficiles à identifier. En effet, les programmes évolués en général, et en particulier ceux implémentant des protocoles de communication comportent de nombreux éléments de concurrence et des flux de données assez subtils : les interactions entre les différentes parties d'un programme sont alors suffisamment complexes pour qu'il soit difficile de trouver la cause d'un comportement particulier du programme, et en particulier, d'un de ses bugs. C'est pourquoi il est nécessaire d'avoir des outils de vérification permettant d'identifier et de corriger simplement la plupart des erreurs survenues lors de la conception des assemblages de composants. Plus précisément, l'approche proposée dans [13] suggère l'utilisation d'un système de type pour s'assurer que les composants manipulent correctement les messages échangés entre les différents acteurs d'un protocole.

Plusieurs études visant à l'élaboration d'outils de validation ont déjà été étudié avant cette thèse, pour des modèles à composant tels que *Ensemble* [70], Aster [54], ou Plastick [59]. Néanmoins, aucun de ces travaux ne s'intéresse aux systèmes de types et aucun n'est adapté à la vérification des manipulations effectuées sur les messages. De plus, les systèmes de type existant ne sont pas suffisant pour typer de manière satisfaisante les flux de donnés que l'on peut trouver dans les assemblages Dream, et dans les modèles à composant en général. Typiquement, Dream possède des composants (appelés *multiplexeurs*) combinant deux flux de donnés, ainsi que d'autres (appelés *routeurs*) permettant de les séparer : comme nous le verrons par la suite, ces deux composants n'ont pas de type satisfaisant dans les systèmes de type classique à la ML [76] ou à objet (Java, C++).

1.2 Structure de ce document

L'approche que nous avons suivie dans cette thèse étend de trois façons le travail entrepris dans [13].

1. L'objet de notre étude n'est plus seulement Dream, l'objectif de notre travail étant de pouvoir s'adapter facilement à la plupart des modèles à composant existant ;
2. Pour ce faire, nous nous abstrayons des spécificités de chaque modèle en utilisant un calcul de composant identifiant de manière précise les

différents éléments (composants, messages, procédures de routage) que nous souhaitons typer. Ceci est illustré Figure 1.1. De plus, ce modèle offre une base solide pour prouver formellement les propriétés de vérification du système de type construit.

3. Enfin, nous étendons les idées originales de [13] pour pouvoir prendre en compte dans nos types de nouveaux éléments, comme le routage.

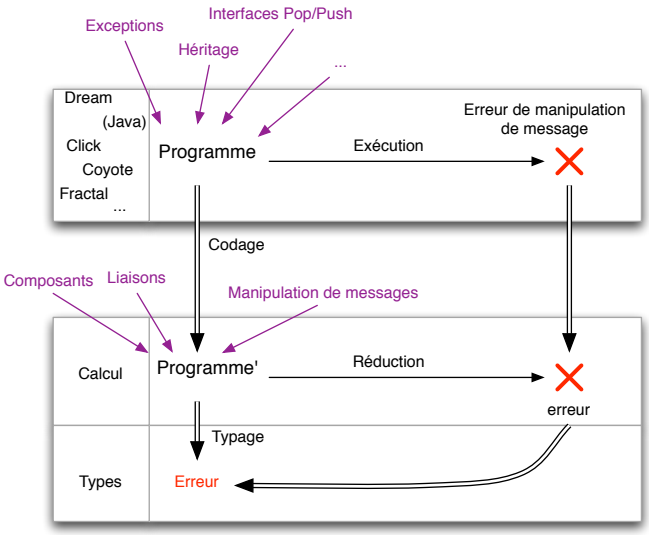


FIGURE 1.1 – Notre approche

La finalité de notre travail de thèse est donc double : nous cherchons à la fois à définir un calcul formel incluant des composants, des primitives de manipulation de la structure d'un programme à chaud (comme le fait FRACTAL) et des primitives de manipulation de flux de données (comme les routeurs et multiplexeurs de DREAM) ; et nous voulons définir un système de type pour un tel calcul. Bien sûr cet objectif est très ambitieux, et nous n'avons pas pu l'achever. Nous présentons alors dans ce document trois travaux visant à l'accomplissement de notre objectif.

Le typage du routage structurel. Le chapitre 2 présente un premier calcul visant à abstraire les modèles à composant, ainsi que le système de type que nous avons construit sur ce calcul. Ce dernier est relativement simple et intègre une procédure de routage basée sur la structure des messages transmis. C'est pourquoi ce travail se nomme *routage structurel*. Nous verrons dans ce chapitre que cette méthode de routage implique certaines propriétés au système de type associé.

Le typage du routage sémantique. Nous présentons dans le chapitre 3 une nouvelle instance de notre approche, incluant un nouveau calcul et un nouveau système de type. Le calcul est légèrement modifié par rapport au précédent, et la plus grande innovation concerne la méthode de routage, qui est maintenant basée sur des annotations placées sur les messages. Cette nouvelle méthode se nomme donc *routage sémantique*, et offre des propriétés différentes au système de type. De plus, nous avons implémenté ce système pour les modèles Dream et Click, ce qui est présenté Chapitre 4.

Le langage Oz/K**.** Finalement, nous avons défini Chapitre 5 un langage incorporant plusieurs fonctionnalités importantes des modèles à composant comme la possibilité de modifier la structure d'un assemblage en cours d'exécution. L'intérêt de ce travail est d'étudier comment il est possible de définir formellement ces opérations et de raffiner ces définitions jusqu'à obtenir une sémantique satisfaisante pour celles-ci. Nous n'avons pas construit de système de type pour ce langage.

Notons que nous n'avons pas de chapitre d'état de l'art dans cette thèse. En effet, comme ce travail touche beaucoup de domaine, nous avons préféré décrire en détail les constructions que nous faisons dans ce document afin d'en simplifier la lecture. Les références vers les travaux précédents ainsi que les discussions par rapport à notre travail sont présentées dans chaque chapitre, principalement dans les parties de discussion.

Chapitre 2

Routage structurel

Le premier travail auquel nous nous sommes attelés durant cette thèse a été de définir un calcul de processus simple, sans ordre supérieur, qui puisse servir de base pour une adaptation d'un système de type tel que l'on peut en trouver pour les langages de la famille ML [35, 78, 87, 92, 3, 82]. Ce calcul, inspiré du kell-calcul [98, 14, 52], comporte des processus échangeant des messages similaires à des valeurs ML, et aussi des messages structurés tels des *enregistrements* [93, 112]. Ce type de message est en effet beaucoup utilisé en programmation, et en particulier dans les protocoles de communication. De plus, ces structures ont un typage totalement inférable pour les langages à la ML [24, 93, 112], ce qui convenait parfaitement à notre approche, le but de ce premier travail étant de définir un système de type inférable pour un calcul simple, avant de complexifier la tâche avec de l'ordre supérieur et de la reconfiguration dynamique. Finalement, nous voulions aussi intégrer dans ce calcul une primitive de routage de message, car une fois encore, une telle opération, similaire au filtrage de motif [56, 57] et à l'analyse de type [33, 113], est extrêmement commune dans les langages de programmation, et souvent utilisée dans les protocoles de communication [80, 13, 12]. De plus, parmis les différents travaux qui ont servis de base à cette thèse, comme le langage Pict [85, 109], les travaux sur le calcul $\lambda\pi_v$ [116, 117, 115], le kell, ou différentes versions du π-calcul [15, 77, 66], aucun n'intègre une construction de routage. Cette première approche fut donc une bonne opportunité pour définir une primitive de routage dans un calcul de processus, et de définir un système de type adapté à cette construction.

2.1 Le calcul

Le calcul que nous proposons dans cette première approche est relativement simple, et consiste en un assemblage élémentaire de différentes notions, telles que les messages structurés ou les composants, dans un même langage. Nous sommes partis du kell calcul [98], auquel nous avons ôté les filtrages de motif et l'ordre supérieur qui ont pour but de permettre la reconfiguration dynamique.

D	$::=$		Programmes
		$b[D']$	Composant
	$\mid$	B	Processus
	$\mid$	$D_1 \mid D_2$	Mise en parallèle
B	$::=$		Processus
		0	Processus vide
	$\mid$	R	Envoi de message
	$\mid$	$e(x).(B_1 \mid \ldots \mid B_n)$	Recepteur
	$\mid$	$!B$	Réplication
R	$::=$		Envoi de message
		$\overline{e}\langle M \rangle$	Envoi direct
	$\mid$	$\mathrm{IfPre}(a, M, s_1, s_2)$	Routage
M	$::=$		Message
		x	Variable
	$\mid$	$\{a_1 = M_1; \ldots; a_n = M_n\}$	Record
	$\mid$	$\mathfrak{c}$	Constantes
	$\mid$	$(M_1\ M_2)$	Application

FIGURE 2.1 – Syntaxe de notre calcul

Afin d'avoir une sémantique vraiment minimale, nous avons aussi rendu le calcul asynchrone, imitant ainsi le π-calcul asynchrone [15], étendu par des localités. Mais contrairement au π-calcul, nos processus échangent des valeurs à la [75], celles-ci pouvant être soit des constantes de base telles que des opérateurs ou des entiers, ou des messages structurés. Finalement, nous avons dû décider d'une procédure de routage pour notre calcul. Parmis les différentes possibilités qui se sont offertes, nous avons choisi un routage assez simple, basé sur la structure des messages structurés. Ce choix était assez arbitraire, mais permet tout de même, malgré sa simplicité, d'encoder beaucoup d'opérations disponibles dans les programmes distribués et les protocoles de communication existants. Ceci sera illusté par quelques exemples à la fin de la présentation de ce calcul.

La syntaxe de notre calcul est présentée Figure 2.1. Nous utilisons 'a', 'c' et 'd' pour désigner les noms de champs des messages structurés, et 'e', 's' ou 'i' pour désigner des canaux de communication. Un composant $b[D]$ est une boîte avec un nom b et un contenu D. $D_1 \mid D_2$ est la mise en parallèle des deux processus D_1 et D_2. Les processus peuvent être soit le processus vide 0, un envoi de message R, une réplication infinie de processus $!B$ ou un receveur $e(x).(B_1 \mid \ldots \mid B_n)$ qui attend un message sur le canal e et, à sa

reception, execute les programmes B_i. L'envoi de message peut être soit un envoi direct $\overline{e}\langle M\rangle$ où le message M est directement envoyé sur le canal e, ou un routage $\mathrm{IfPre}(a, M, s_1, s_2)$. Cette dernière opération suppose que M est un message structuré et teste si le champ 'a' y est présent : si c'est le cas, le message est envoyé sur s_1, sinon, il est envoyé sur s_2. Les messages consistent en un simple calcul fonctionnel sans abstraction (les fonctions sont supposées être définies comme des constantes), et avec messages structurés. Nous avons donc des variables x, des structures $\{a_1 = M_1; \ldots; a_n = M_n\}$, des constantes $\mathfrak{c}$, et l'application de deux messages $(M_1\ M_2)$. L'ensemble des constantes n'est pas précisé, ce qui permet d'avoir un calcul relativement générique. Nous supposons néanmoins, pour définir nos exemples, que nous avons à disposition les opérations de base sur les messages structurés, ainsi que les entiers et les chaînes de caractères.

Comme pour la syntaxe, la sémantique opérationnelle de notre calcul est assez intuitive, basée sur un kell-calcul simplifié et une sémantique classique des constantes [102]. La façon dont s'exécutent les processus est représentée dans notre modèle par une relation de réduction, notée $\triangleright$, définie sur les termes clos. Cette réduction est construite modulo (i) une relation d'équivalence sur les processus qui identifie des programmes similaires, (ii) des contextes d'évaluation qui définissent où une exécution peut prendre place dans un programme et (iii) la sémantique opérationnelle des constantes du langage.

L'équivalence structurelle entre deux messages ou deux programmes, notée $\equiv$, est la plus petite relation d'équivalence qui est aussi une congruence et qui satisfait les règles données Figure 2.2. Cette équivalence est assez naturelle, et

$$D_1 \mid D_2 \equiv D_2 \mid D_1 \qquad D_1 \mid (D_2 \mid D_3) \equiv (D_1 \mid D_2) \mid D_3 \qquad D \mid 0 \equiv D$$

$$!B \equiv B \mid !B \qquad \{a_1 = M_1; \ldots; a_i = M_i; a_{i+1} = M_{i+1}; \ldots; a_n = M_n\} \equiv \{a_1 = M_1; \ldots; a_{i+1} = M_{i+1}; a_i = M_i; \ldots; a_n = M_n\}$$

FIGURE 2.2 – Équivalence structurelle

reprend les relations classiques sur les processus et les messages structurés : l'ordre des champs et des processus importe peu et le processus '0' est l'élément neutre de la composition parallèle. De plus, nous traitons la réplication de processus $!B$ de manière similaire à ce qui est fait dans le π-calcul, en simulant l'existence d'une infinité d'instances de B.

Les contextes d'évaluations, dont la syntaxe est décrite Figure 2.3, sont d'une part constituées d'un trou '[]' qui symbolise où l'exécution peut prendre place,

et d'un processus englobant qui correspond au contexte d'exécution. Il est à noter que ce contexte n'est pas modifié par l'exécution prenant place dans le trou.

E	$::=$	$[]$	Trou
	$\mid$	$(M\ E) \quad \mid \quad (E\ M)$	Application
	$\mid$	$\{a_1 = E; a_2 = M_2; \ldots; a_n = M_n\}$	Message structuré
	$\mid$	$\overline{s}\langle E \rangle$	Envoi de message
	$\mid$	$\mathrm{IfPre}(a, E, s_1, s_2)$	Routage
	$\mid$	$E \mid D$	Composition parallèle
	$\mid$	$b[E]$	Composants

FIGURE 2.3 – Contexte d'évaluation

Les constantes de notre calcul peuvent être de simples données, comme des entiers ou des chaînes de caractères, mais aussi des opérateurs, comme les manipulateurs de messages structurés '$.a$', '$+(a = M)$' ou '$-a$'. Afin de connaître l'action de ces opérateurs lors de l'exécution d'un programme, nous supposons données deux relations définissant pour chaque opérateur, quels sont ses paramètres valides (définit par la relation '`match`'), et le résultat correspondant (définit par la relation '`eval`'). Afin d'assurer la convergence du calcul des messages ainsi que la correction de notre futur système de type, nous imposons toutefois une contrainte sur la relation '`match`', qui ne doit donner que des *valeurs* comme paramètres valides aux opérateurs :

Définition 1. *Une* valeur *est un message sans application :*

v	$::=$		*Les valeurs*
		$\{a_1 = v_1; \ldots; a_n = v_n\}$	*Un message structuré*
	$\mid$	x	*Une variable*
	$\mid$	c	*Une constante*

La Figure 2.4 presente un exemple de définition des deux relations, décrivant la sémantique opérationnelle des opérateurs de base sur les messages structurés. Par la suite, nous notons $\texttt{match}(c, v)$ lorsque la paire (c, v) est dans la relation '`match`'.

Finalement, la relation de réduction est définie comme la plus petite relation vérifiant les règles données Figure 2.5. Les deux règles CONTEXT-D et

$$\texttt{match} \triangleq \left(\begin{array}{l} \{(.a, \{a = v, a_1 = v_1; \ldots; a_n = v_n\}) \mid 0 \leq n\} \\ \cup \{(-a, \{a = v, a_1 = v_1; \ldots; a_n = v_n\}) \mid 0 \leq n\} \\ \cup \{(+(a = v), \{a_1 = v_1; \ldots; a_n = v_n\}) \mid \forall 0 < i \leq n, a_i \neq a\} \end{array} \right)$$

$$\begin{aligned} \texttt{eval}(.a, \{a = v, a_1 = v_1; \ldots; a_n = v_n\}) &\triangleq v \\ \texttt{eval}(-a, \{a = v, a_1 = v_1; \ldots; a_n = v_n\}) &\triangleq \{a_1 = v_1; \ldots; a_n = v_n\} \\ \texttt{eval}(+(a = v), \{a_1 = v_1; \ldots; a_n = v_n\}) &\triangleq \{a = v; a_1 = v_1; \ldots; a_n = v_n\} \end{aligned}$$

FIGURE 2.4 – Un exemple de relations `match` et `eval`

$$\textsc{Context-D}\ \frac{D \ \rhd\ D'}{E[D] \ \rhd\ E[D']} \qquad \textsc{Context-M}\ \frac{M \ \rhd\ M'}{E[M] \ \rhd\ E[M']} \qquad \textsc{App}\ \frac{\texttt{match}(c, M)}{(c\ M) \ \rhd\ \texttt{eval}(c, M)}$$

$$\textsc{ifPre}\ \frac{M = \{a = M_1; a_2 = M_2; \ldots; a_n = M_n\}}{\text{IfPre}(a, M, s_1, s_2) \ \rhd\ \overline{s_1}\langle M \rangle}$$

$$\textsc{ifAbs}\ \frac{M = \{a_1 = M_1; \ldots; a_n = M_n\} \qquad \forall 0 < i \leq n,\ a_i \neq a}{\text{IfPre}(a, M, s_1, s_2) \ \rhd\ \overline{s_2}\langle M \rangle}$$

$$\textsc{Com1}\ \overline{e}\langle M \rangle \mid e(x).(B_1 \mid \ldots \mid B_n) \ \rhd\ (B_1 \mid \ldots \mid B_n)\{^M/_x\}$$

$$\textsc{Com2}\ \overline{e}\langle M \rangle \mid b[e(x).(B_1 \mid \ldots \mid B_n) \mid D] \ \rhd\ b[(B_1 \mid \ldots \mid B_n)\{^M/_x\} \mid D]$$

$$\textsc{Com3}\ b[\overline{e}\langle M \rangle \mid D] \mid e(x).(B_1 \mid \ldots \mid B_n) \ \rhd\ b[D] \mid (B_1 \mid \ldots \mid B_n)\{^M/_x\}$$

FIGURE 2.5 – Les règles de réduction

CONTEXT-M utilisent les contextes d'évaluation afin de permettre l'exécution à l'intérieur d'un programme. La règle APP définit le résultat d'une application, qui est $\texttt{eval}(c, M)$ lorsque M est un argument valide. Nous avons bien sûr deux réductions possibles pour la procédure de routage : la règle IFPRE définit ce qu'il se passe lorsque le champ requis est présent (le message est envoyé sur 's_1') et la règle IFABS indique que lorsque le champ requit n'est pas présent, le message est envoyé sur 's_2', comme indiqué en introduction de ce chapitre. Finalement, nous avons trois règles de communication entre processus, suivant ainsi le modèle du

kell-calcul. La première permet la communication entre deux processus qui font parti d'un même composant, et les deux autres règles étendent cette capacité à des processus qui ne sont distant que d'une frontière de composant (les communications distantes sont ainsi empéchées). Notons que [52] étend le kell-calcul en lui permettant de *partager* des composants, lui offrant ainsi une plus grande souplesse de communication que proposé ici. Nous n'avons pas voulu intégrer ce mode de communication distante pour garder notre calcul simple lors de la définition de notre premier système de type.

2.1.1 Exemples et discussion

La charte graphique. Lors de la création d'exemples afin d'illustrer notre calcul, nous nous sommes aperçu que le mode de communication pris du kell-calcul [98] imposait une discipline forte de nommage, car il n'est pas évident de s'assurer, dans un grand système, que deux processus ne peuvent communiquer. En effet, nous avons conçu ce calcul pour modéliser des composants simples, sans modification dynamique de leurs structures. Par construction, il est donc naturel de considérer les canaux de communication comme les interfaces de ces composants, et de les appeler e_i pour les interfaces d'entrée, s_i pour les interfaces de sortie, et par exemple i_i pour les interfaces internes (locales à chaque composant). Mais comme les communications peuvent traverser une frontière de composant, cette politique de nommage simple ne fonctionne pas, et entraine des communications non voulues entre composants pères et fils. Avec une politique de nommage un peu plus évoluée, il est facile de remédier à ce problème, mais les exemples deviennent alors assez long et difficiles à lire, dû simplement à la longueur des noms des canaux de communication. C'est pourquoi nous adoptons dans ce document une présentation graphique de nos assemblages, où les composants sont de simples rectangles, leurs interfaces des T (rouges pour les interfaces d'entrée, verts pour les sorties), et les liens entre les composants de simples flèches. Avec ceci, nous pouvons avoir une politique simple de nommage des canaux pour les processus s'exécutant dans les composants.

Les premiers composants, dit *primitifs* [1], que nous proposons Figure 2.6 sont extrèmement simples, et présentent des opérations élémentaires sur les messages, comme l'accès à un champ de message structuré, ou un routage. Les générateurs tels que le premier composant de cette figure sont simplement constitués d'un message envoyé sur le canal de sortie une infinité de fois. Le second composant, `acsTTL`, écoute sur le canal d'entré e, et à la réception d'un message, renvoie le contenu de son champ `TTL` sur le canal s. Le troisième ajoute aux messages rentrant un champ `TTL` de valeur 10, et le quatrième ôte ce champ des messages reçus. Un *routeur* dans notre calcul est un composant comportant une simple procédure de routage envoyant les messages reçus sur une des deux sorties en fonction de leurs structures. Finalement, un *multiplexeur* comporte deux entrées,

1. plus précisément, les composants primitifs sont ceux qui n'ont pas de composant fils, ces derniers étant appelés composants *composites* [17]

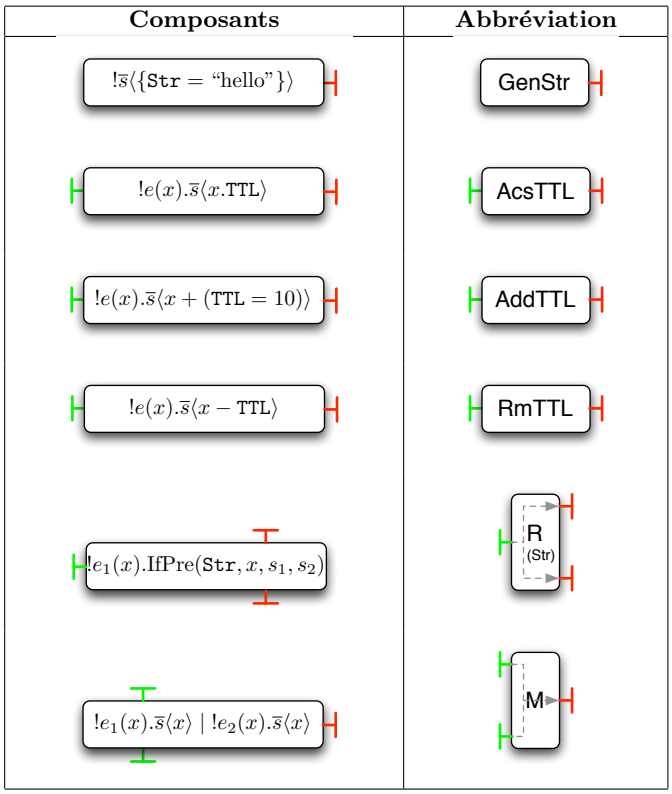


FIGURE 2.6 – Exemples de composants primitifs

et chaque message reçu est renvoyé sans modification sur l'unique canal de sortie, mixant ainsi deux flux de messages. Notons que les processus internes de tous ces composants sont précédés de l'opérateur de réplication ! afin que leurs fonctionnalités ne soient pas perdues après la réception du premier message. Par la suite, nous utiliserons les abbréviations lors des utilisations des composants primitifs, en modifiant leur nom pour signaler un changement trivial dans leur sémantique : par exemple le composant `addIP` est similaire au composant `addTTL`, sauf qu'il rajoute un champ `IP`, et non `TTL` aux messages en entrée. De même, le routeur `R(Str)` route par rapport à l'existence ou non du champ `Str` dans les messages, alors que `R(IP)` route par rapport au champ `IP`.

Nous pouvons assembler ces composants en de simples chaînes, comme proposées Figures 2.7 et 2.8. Dans ces deux assemblages, nous introduisons trois nouveaux composants primitifs, `Send`, `Receive` et `PrintStr`. Les deux premiers composants vont par paire, et représentent l'échange de messages au travers d'un réseau comme internet. Malgré la mecanique potentiellement complexe derrière la communication via un réseau, ces composants se modélisent dans notre cal-

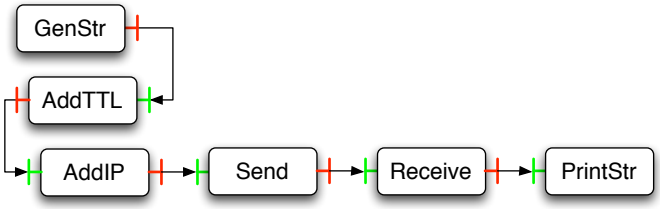


FIGURE 2.7

cul par la simple transmission des messages reçus, sans aucune modification (précisement, le code interne de ces composants a la forme $!e(x).\overline{s}\langle x\rangle$). Finalement, le composant `PrintStr` affiche à l'écran le contenu du champ `Str` des messages reçus. Il est encodé dans notre calcul par le processus interne $!e(x).0$ [2] qui jette les messages que l'on lui envoie. Le comportement de ces deux assem-

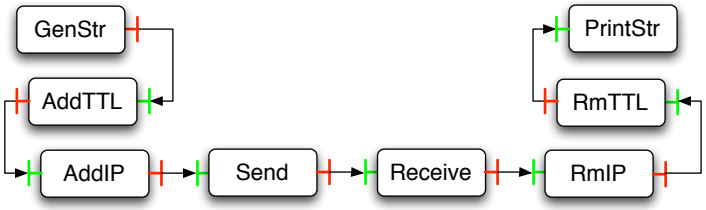


FIGURE 2.8

blages (Figures 2.7 et 2.8) est assez évident : ils génèrent tous deux un message contenant une chaîne de caractère, lui ajoutent un champ `TTL` et une adresse IP afin que la transmission par les composants `Send` et `Receive` se fasse sans problème, et finalement affichent la chaîne de caractère générée à l'écran. Le second assemblage est légèrement plus évolué, car il ôte les champs `TTL` et `IP` du message après la transmission, ce qui fait que le message à l'entrée de `PrintStr` est identique à celui qui était généré par `GenStr`. Cette construction est donc conceptuellement plus propre que la première, et permet d'éviter certains types d'erreur dans les programmes à base de composants [13].

Abstraction et isolation. Beaucoup de modèles à composants existants [80, 7, 46, 38, 70] ne permettent pas d'assembler un ensemble de composants simples en un unique de comportement complexe et intéressant pour la réalisation d'un

2. Remarquons que nous pouvons définir ces composants de manière un peu plus évoluée, en s'assurant par exemple que les messages reçus par `Send` possèdent bien un champ `TTL` et `IP`

programme informatique. Néanmoins, nous adoptons ici la même approche que le kell et le M-calcul [97] en offrant la possibilité de construire de tels composants *composites*. Ce genre de composant est en effet assez utile, et offre des avantages d'abstraction similaires aux modules [72, 21] en assemblant un ensemble de comportements simples en un unique comportement complexe, tout en cachant la manière dont celui-ci est obtenu. Un exemple d'un tel assemblage est présenté Figure 2.9. Ce composant est constitué d'une simple chaîne traitant toutes les étapes de transfert d'un message via un reseau. L'ajout des champs `IP` et `TTL`, l'envoi et la réception, ainsi que l'effacement de ces deux champs (utiles uniquement pour le transfert des messages) y sont totalement traité. Ce composant, que l'on peut nommer `transmit`, peut maintenant être utilisé par tous les assemblages nécessitant le tranfert de message via un réseau, et ce, sans avoir à reconstruire à chaque fois le processus interne d'envoi et de reception des messages. Notons que l'intérêt de ce genre de composant est un

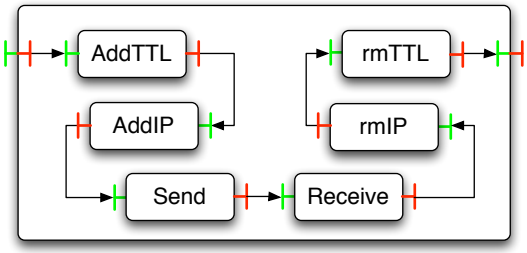


FIGURE 2.9

peu limité dans notre calcul, dû à la *porosité* des frontières des composants. En effet, comme nous l'avons signalé en début de la présentation des exemples, les communications peuvent traverser une frontière de composant sans aucun contrôle de la part du programmeur. Ainsi, il est nécessaire de connaître le processus interne d'un composant composite pour pouvoir l'intégrer sans danger à un autre processus, rendant l'abstraction des composants composites moins pratique que ce que nous espérions. Néanmoins, nous supposons dans le reste de ce document que le principe de nommage cité précédemment est appliqué à tous nos exemples, et que nous pouvons donc assembler les composants sans risquer des problèmes de communication.

Les assemblages que nous avons construits jusqu'à présent dans nos exemples sont de simples chaînes de composants. Nous pouvons un peu complexifier ces structures simples en les mettant en parallèle comme cela est montré Figure 2.10, mais aussi en utilisant les multiplexeurs et routeurs, permettant ainsi de fusionner différents flux de messages, et de les séparer par la suite. L'intérêt d'une telle manipulation de flux est présentée Figure 2.10 : dans cet assemblage,

FIGURE 2.10

deux messages sont généré, et envoyé via deux composants `transmit` vers un ordinateur distant pour y être tout les deux affichés. Nous avons donc ici un composant `transmit` redondant : il serait en effet logique de n'en avoir qu'un seul qui assure l'envoi et la réception des deux types de messages. Il faut néanmoins être prudent ici, car après que les deux types de messages soient réceptionnés sur la seconde machine, nous devons être capable de les distinguer, les séparer, et les envoyer vers le composant d'affichage correspondant. Cela est possible grâce à un routeur, et l'assemblage résultant est présenté Figure 2.11. Cet assemblage

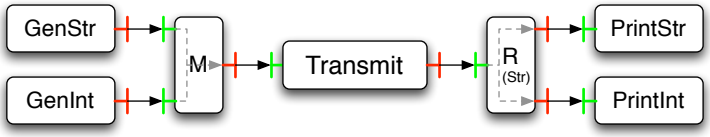


FIGURE 2.11

est assez intuitif : nous avons nos deux générateurs, qui sont connectés à un multiplexeur envoyant tous les messages vers un unique composant `transmit`. La sortie de ce composant est liée à un routeur, envoyant tout les messages ayant un champ `Str` (donc généré par `genStr`) vers le composant `printStr`, et les autres (donc généré par `genInt`) vers `printInt`. Les assemblages 2.10 et 2.11 opèrent donc exactement les mêmes opérations sur les messages, le second étant néanmoins plus efficace en terme de duplication de code. Nous pouvons aussi remarquer une caractéristique intéressante du routage dans cet assemblage : la procédure de routage `R(Str)` (et toutes les procédures de routage en général) possède deux sorties, une pour les messages ayant un champ `Str` et une pour les autres. Or, la seconde sortie du routeur est connectée au composant `PrintInt`, indiquant ainsi que les messages n'ayant pas de champ `Str` ont un champ `Int`.

Les dépendances du routage. Les procédures de routage peuvent donc introduire ce que nous appelons des *dépendances* entre les champs des messages : dans l'assemblage 2.12, un message a soit le champ `Str`, soit il ne l'a pas, mais alors il possède le champ `Int` (notons que le message peut avoir à la fois les

champs `Str` et `Int`). Ce genre de dépendances a quelques effets sur la manière

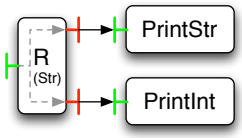


FIGURE 2.12

dont nous typons nos assemblages comme nous le verrons dans la partie suivante, mais ils peuvent être évités par l'introduction de nouveaux routeurs, comme cela est fait Figure 2.13. Ici, tout les cas de messages sont traités : soit le messages

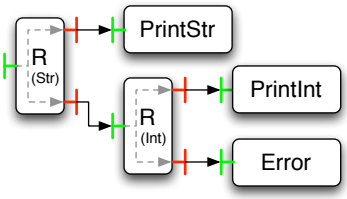


FIGURE 2.13

possède le champ `Str`, auquel cas il est traité par `printStr`, soit il ne possède pas ce champ, mais contient tout de même `Int` et est alors traité par `printInt`. Finalement, si le message ne possède aucun des deux champs, il est pris en charge par le composant `error`. Nous pouvons ainsi voir que la gestion des filtrages de motifs exhaustifs est possible dans notre calcul, mais assez lourde tout de même, car nous devons introduire une procédure de routage et un canal de communication pour chaque motif du filtrage.

Finalement, notre dernière série d'exemples présente différentes boucles simples que l'on peut construire avec notre calcul. La première, décrite Figure 2.14, montre comment on peu générer la totalité des entiers naturels, en supposant que le + fait partie des opérateurs de notre calcul. Cet assemblage comporte deux composants primitifs formant la boucle, ainsi qu'un composant composite cachant la mécanique interne et n'exposant que l'interface vers laquelle les entiers générés sont envoyés. Le premier composant envoie l'entier 0 au second, commençant le processus de génération des entiers. Le second composant transmet l'entier reçus vers la sortie, mais le renvoie aussi vers le premier composant,

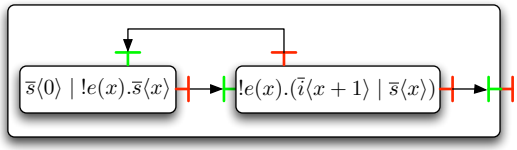


FIGURE 2.14

en lui ajoutant 1. Finalement, le premier composant renvoie sans modification au second tout message reçu : ce dernier obtient donc tour à tour les entiers 0, 1, 2, etc, qu'il transmet vers la sortie de l'assemblage.

Notre second exemple de boucle est plus évolué, et présente l'envoi de messages via un réseau, avec un méchanisme de callback pour lancer l'envoi d'un nouveau message. La structure générale de l'assemblage, présenté Figure 2.15,

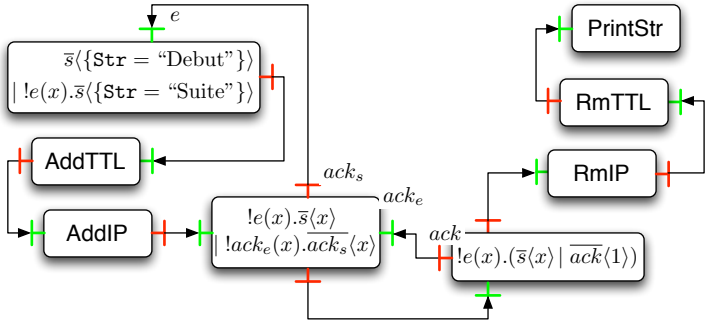


FIGURE 2.15

est identique à notre assemblage d'envoi précédent (Figure 2.8) avec un génerateur de messages, des composants `addIP` et `addTTL`, un composant d'envoi et un autre de réception, deux composants `rmIP` et `rmTTL` terminés par `printStr`. Les modifications que nous avons apportés à l'assemblage concernent les composant de génération, d'envoi et de reception, afin que ce dernier signale à l'envoyeur qu'il a bien reçus un message, lançant alors l'émission d'un nouveau message. Notons que certaines interfaces sont annotées par leur nom dans cette figure : de cette façon, les connexions entre composants ne sont pas ambiguës.

Finalement, notre dernier exemple présente une boucle exprimable dans notre calcul, et qui possède un comportement assez particulier. Ce composant, décrit Figure 2.16, est construit à partir d'un routeur, d'un multiplexeur, et d'un composant donnant la valeur du champ `Sub` des messages qu'il reçoit en entrée.

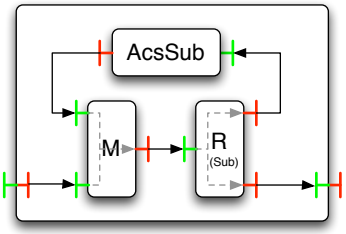


FIGURE 2.16

La façon dont ces composants sont agencés font que l'assemblage donne en sortie le message structuré placé sous un nombre arbitraire de champ `Sub`. Il est en effet facile de vérifier que si on lui donne en entrée le message $\{\texttt{Str} = \text{“}\texttt{toto}\text{”}\}$, ou $\{\texttt{Sub} = \{\texttt{Str} = \text{“}\texttt{toto}\text{”}\}\}$ ou encore $\{\texttt{Sub} = \{\texttt{Sub} = \{\texttt{Str} = \text{“}\texttt{toto}\text{”}\}\}; \texttt{Int} = 1\}$, le composant rendra à chaque fois le message $\{\texttt{Str} = \text{“}\texttt{toto}\text{”}\}$.

2.2 Le système de type

L'approche que nous avons suivi pour la conception de ce système de type est assez similaire à celle suivi pour le typage du join-calcul [41, 31]. Le but est en effet de créer une simple adaptation du typage de ML [35] pour notre calcul de processus, avec pour seule contrainte de prendre en compte les messages structurés et leur routage. Nous n'avons en effet pas envisagé de suivre l'exemple de Pict [85], qui utilise une variante du système F [44] pour typer ses processus, rendant ainsi l'inférence de type impossible [58], ou celle de [22], qui s'intéresse plutôt à la mobilité des processus. De plus, bien que proche des types de sessions [119, 111] du fait que nous nous intéressons au typage des messages, notre approche en est tout de même différente, simplement car nous n'intégrons pas les même constructions dans notre langage. Néanmoins, notre adaptation est assez différente de ce qui a été fait avec le join-calcul, car bien que nous soyons aussi en présence d'un calcul de processus, ceux-ci n'échangent pas de nom de canaux, et n'ont pas de définitions similaires à celles trouvées dans le Join. De plus, un des objectifs de ce système de type était de permettre un typage souple de la procédure de routage, afin qu'elle puisse prendre en entrée différents types de message, et il s'est avéré que la prise en compte de cet élément ne peut pas se faire en utilisant simplement les constructions introduites dans [35, 56, 93, 112] ou dans les système de types à la ML qui ont suivi. En effet, considérons un assemblage composé d'un ensemble de programmes communicant par paire, et utilisant un composant de cryptage pour sécuriser leurs messages. Un tel asssemblage est présenté Figure 2.17. Les composants de la partie gauche de cet assemblage génèrent un ensemble de messages de struc-

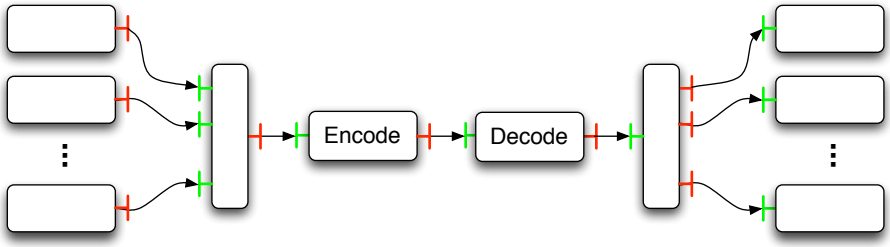


FIGURE 2.17 – Un assemblage classique utilisant multiplexeur et routeur

tures différentes, qui sont ensuite transmis à un *multiplexeur* les envoyant tous au composant `Encode` assurant le chiffrement des messages durant leur transmission sur le réseau. Ces messages sont ensuite traités par un décodeur `Decode`, et un *routeur* qui définit vers quels composants ces messages doivent être envoyés. Au niveau des types, les composants de gauches génèrent des types tous différents, qu'il faut pouvoir récupérer intacts sur le côté droit de l'assemblage. Une approche naturelle pour faire cela serait d'utiliser un constructeur de type, comme un n-uplet, pour typer la sortie d'un multiplexeur. Ce n-uplet pourrait alors être détruit par la procédure de routage, et les types seraient ainsi transmis intacts aux composants du côté droit. Cette approche n'est malheureusement pas possible, en raison des composants d'encodage et de décodage placés entre le multiplexeur et le routeur. En effet, ces composants manipulent un nombre inconnu de messages multiplexés, et donc, avec cette approche, ils doivent prendre en compte des n-uplets de taille inconnu et de profondeur arbitraire (en effet, plusieurs multiplexages peuvent se suivre dans un assemblage complexe). Il est donc impossible de donner un type unique aux procédures d'encryptage et de décryptage de message, et de fait, à toutes les fonctions opérant sur les messages. Même en appliquant des systèmes de types très généraux comme [82, 102, 104] à notre calcul, nous obtenons le même problème : il est impossible de typer correctement les opérations de base sur les messages.

Afin de résoudre ce problème de typage, nous avons étendu les types à la ML par des types ensembles [3, 1]. Ainsi, nos types de processus [117, 73] consistent en des associations finies entre des noms de canaux et des ensembles finis de types qui correspondent aux différents messages transitant sur ces canaux durant l'exécution du programme : les types des messages étant alors indépendant les uns des autres, il est possible des les traiter séparément et d'éviter ainsi le problème précédent pour la définition des types des composants de cryptage et de décryptage. Finalement, cette approche nous a semblé consistente avec notre recherche d'inférence de types [2].

T	$::=$	$E \quad \mid \quad \forall\alpha.T$	Schéma de type
E	$::=$		Type élémentaire
		η	Variable
	$\mid$	$\{W^{\emptyset}\}$	Type rangé
	$\mid$	$E \rightarrow E$	Fonction
	$\mid$	$t(E_1, \ldots, E_{a(t)})$	Constructeur de type
W^l	$::=$		Type rangé
		$a : Pre(E); W^{l \uplus \{a\}}$	Définition du champ a
	$\mid$	$a : Abs; W^{l \uplus \{a\}}$	Définition du champ a
	$\mid$	Abs^l	La rangée vide
	$\mid$	ρ^l	Variable de rangé
S	$::=$		Type de processus
		$\emptyset$	Type vide
	$\mid$	$e \quad \mid \quad e : (T)$	Définition d'un canal
	$\mid$	$S \cup S'$	Union

FIGURE 2.18 – La syntaxe des types

2.2.1 La syntaxe

L'algèbre de nos types est construite à partir d'un ensemble de variables $\mathcal{V}$ et de constructeurs de types $\mathcal{C}$. Nous supposons que $\mathcal{V}$ est l'union disjointe des ensembles $\mathcal{V}^m$ (pour les variables de type de message) et $\mathcal{V}^r(l)$ (pour les variables de rangées) où l est un ensemble d'étiquette (c'est à dire un nom de champ). Nous notons η les variables dans $\mathcal{V}^m$ et ρ^l les variables dans $\mathcal{V}^r(l)$. Finalement, chaque constructeur de type t est défini avec une arité $a(t)$. La syntaxe de nos types est présentée Figure 2.18. Les schémas de types, ainsi que les types élémentaires et les types rangés sont utilisés pour typer les messages. Les types rangés sont conçus pour avoir la même structure que les messages typés. Par exemple, le message $\{a = 1; c = \text{“}\texttt{name}\text{”}\}$ a pour type $\{a : \mathrm{Pre(int)}; c : \mathrm{Pre(string)}; \mathrm{Abs}\}$: les champs 'a' et 'c' sont définis dans le message et contiennent respectivement un entier et une chaîne de caractère. Comme il n'y a pas d'autre champ dans le message, le type est terminé par Abs. Les messages dont la structure est partiellement connue, comme dans les arguments de fonctions, peuvent être typés par des variables de rangé. Par exemple, le type $\{a : \mathrm{Pre}(\alpha); \rho^{\{a\}}\}$ s'applique pour un message qui contient le champ 'a' : le reste de sa structure n'est pas spécifié et pourra être précisé lors d'ultérieures instanciations du type. Finalement, les types rangés sont annotés par un ensemble d'étiquettes 'l', permettant de s'assurer qu'un champ n'est défini qu'une seule fois dans une même rangée.

Cet exposant a un but purement technique, et nous ne le spécifierons pas dans le reste de ce document, sauf lorsque cela sera nécessaire.

Les types de processus S sont construits comme des ensembles finis de déclarations de canaux, associant ou non ces derniers à des types de message. Ils déclarent ainsi les canaux de communications utilisés par les processus typés, et définissent quels types de message transitent sur chaque canal. Ainsi cette syntaxe est adaptée pour permettre à un canal de transporter plusieurs sortes de données différentes. Par exemple le programme $\overline{e}\langle 1\rangle \mid \overline{e}\langle \text{“}\texttt{toto}\text{”}\rangle$ est typé par $e : (\text{int}) \cup e : (\text{string})$. Ce genre de flexibilité n'est pas courante dans les systèmes de type existant, mais est nécessaire dans notre cas, comme nous l'avons vu en introduction de cette partie. Finalement, le type vide '$\emptyset$' est utilisé pour typer les processus qui n'utilisent pas de canal (typiquement, le processus '0') et le type 'e' sert à typer un processus utilisant e, mais n'envoyant aucune donnée dessus (typiquement, un recepteur sur e).

Nous pouvons noter que la structure des types pour les messages est assez conventionnelle : la seule différence par rapport à [93] concerne la présentation des types rangés, reprise de [13]. L'innovation de nos types se porte donc au niveau des types de processus. En effet, les systèmes de type existant pour des calculs de processus associent à chaque canal de communication, non pas un ensemble de type, mais un type unique, ce qui ne leur permet pas de prendre en compte les comportements tels que le routage. Dans la suite de ce document, nous notons fv la fonction très utilisée renvoyant l'ensemble des variables libres d'un type. Comme nous utilisons des structures relativement simple pour décrire nos types, nous pensons qu'il n'est pas nécessaire de donner une définition précise de cette fonction.

2.2.2 Le typage des messages et des composants

Le typage de nos messages et assemblages est défini de manière classique, par un ensemble de règles de typage. Ces règles sont construites à partir de quelques définitions, notamment une équivalence structurelle identifiant certains types, des substitutions permettant l'instanciations des schémas de types et la définition des types des constantes du calcul.

L'équivalence structurelle. Cette relation est la plus petite relation d'équivalence qui est une congruence satisfaisant les règles présentées Figure 2.19. Informellement, cette relation identifie les types de sémantique similaire : les types rangés mappant les mêmes champs aux mêmes types sont dans la relation, ainsi que les types de processus définissant les mêmes canaux avec les mêmes types de données. Notons par exemple que le type $e \cup e : (T)$ définit e, et spécifie que le type T transite par ce canal, ce qui est identique à simplement indiquer que T transite par e (le canal est alors implicitement déclaré). Finalement, il est facile de voir que tout type S est équivalent à un type de la forme suivante,

$$a : K; b : K'; W^l \equiv b : K'; a : K; W^l \qquad a : \mathrm{Abs}; \mathrm{Abs}^{\{a\} \uplus l} \equiv \mathrm{Abs}^l \qquad S \cup \emptyset \equiv S$$

$$\forall \alpha.\forall \beta.T \equiv \forall \beta.\forall \alpha.T \qquad \frac{T \equiv_\alpha T'}{T \equiv T'} \qquad S \cup S' \equiv S' \cup S$$

$$S \cup (S_1 \cup S_2) \equiv (S \cup S_1) \cup S_2 \qquad S \cup S \equiv S \qquad e \cup e : (T) \equiv e : (T)$$

FIGURE 2.19 – L'équivalence structurelle des types

où les canaux e_i peuvent être utilisés plusieurs fois :

$$(\bigcup_{i \in I} e_i : (T_i)) \cup (\bigcup_{j \in J} e_j)$$

Considérons un type S, et soit S' un type équivalent décrit sous la forme précédente : nous pouvons alors définir les deux notations qui suivent.

- $dc(S) \triangleq \{e_k \mid k \in I \cup J\}$: $dc(S)$ est l'ensemble des canaux déclarés dans S.
- $S(e) \triangleq \{T_i \mid i \in I \wedge e_i = e\}$: $S(e)$ est l'ensemble des types que transporte le canal 'e' dans le type S.

Nous pouvons aisément voir que $S \equiv S' \Rightarrow dc(S) = dc(S') \wedge \forall e,\ S(e) = S'(e)$. Notre définition est donc cohérente avec la relation d'équivalence.

Les substitutions. Une substitution σ est une fonction des variables de types vers les types et qui respecte les kinds : pour tout $\eta \in \mathcal{V}^m$ (resp. $\rho^l \in \mathcal{V}^r(l)$), nous avons $\sigma(\eta) \in E$ (resp. $\sigma(\rho^l) \in W^l$). Nous étendons ces fonctions sur tous types par induction sur la structure de ces derniers. Finalement, nous notons $\mathrm{dom}(\sigma)$ le domaine de la substitution et $\Im(\sigma)$ l'ensemble contenant toutes les variables de types dans l'image de σ. Formellement, nous avons :

$$\mathrm{dom}(\sigma) \triangleq \{\alpha \mid \sigma(\alpha) \neq \alpha\} \qquad \Im(\sigma) \triangleq \bigcup_{\alpha \in \mathrm{dom}(\sigma)} fv(\sigma(\alpha))$$

Avec ces substitutions, nous définissons un prédicat de *dérivation* qui précise quand un type peut être instancié en un autre. Cela simplifie grandement la présentation de certaines règles de typage.

Définition 2. *Un* environement de typage Γ *est un une table associant des variables du calcul à un type qui leur est assigné. Nous notons généralement l'environement vide* $\emptyset$*, et l'ajout d'une association* $\Gamma; x : T$*, où* T *est un type de message quelconque.*

Supposons donnés un environement de typage Γ *et deux schémas de type* T *et* T'*. Nous notons* $\Gamma : T' \Leftarrow T$ *si et seulement si les trois conditions suivantes sont validées :*

- *il existe un ensemble de variables de type* $\overline{\alpha}$ *et un schéma de type* T_1 *tels que* $\overline{\alpha} \cap fv(\Gamma) = \emptyset$ *et* $T = \forall\overline{\alpha}.T_1$ *;*
- *il existe un ensemble de variables de type* $\overline{\gamma}$ *et un schéma de type* T_2 *tels que* $\overline{\gamma} \cap fv(\Gamma) = \emptyset$ *et* $T' = \forall\overline{\gamma}.T_2$ *;*
- *il existe une substitution* σ *telle que* $\mathrm{dom}(\sigma) \subset \overline{\alpha}$ *et* $\sigma(T_1) = T_2$*.*

De manière générale, $\Gamma \vdash T_2 \Leftarrow T_1$ signifie que T_1 peut s'instancier en T_2 : tout message typé par T_1 peut l'être par T_2. Cela permet par une simple application de ce prédicat de montrer par exmple que le type $\forall\eta_1, \rho.\{a : \mathrm{Pre}(\eta_1); \rho\}$ est plus général que $\{a : \mathrm{Pre}(\mathrm{int} \to \mathrm{int}); \mathrm{Abs}\}$.

Le typage des constantes. Finalement, nous utilisons une approche classique pour assigner un type aux différentes constantes de notre calcul. Nous supposons donnée une fonction Ψ qui donne pour toute constante un type. Cette fonction est contrainte dans sa définition : on ne peut en effet donner n'importe quel type pour les constantes, par exemple le type 'string' ne convenant pas à l'opérateur '+'. Il faut donc s'assurer que les paramètres et le résultat d'un opérateur est cohérent avec sa sémantique opérationnelle. Formellement, nous supposons :

$$\forall c,\ fv(\Psi(c)) = \emptyset$$
$$\forall c, v,\ \exists\Gamma, T, \quad (\Gamma \vdash (c\ v) : T) \Rightarrow (\mathtt{match}(c, v) \wedge \Gamma \vdash \mathtt{eval}(c, v) : T)$$

Notons que cette dernière contrainte, bien qu'elle ne mentionne pas explicitement Ψ, s'applique bien à cette fonction qui est utilisée pour typer $(c\ v)$. Finalement, nous illustrons cette définition par un exemple de fonction Ψ qui donne Figure 2.20 les types des trois opérateurs de base sur les messages structurés.

$$\Psi(.a) \triangleq \forall\eta, \rho^{\{a\}}.\{a : \mathrm{Pre}(\eta); \rho^{\{a\}}\} \to \eta$$
$$\Psi(-a) \triangleq \forall\eta, \rho^{\{a\}}.\{a : \mathrm{Pre}(\eta); \rho^{\{a\}}\} \to \{a : \mathrm{Abs}; \rho^{\{a\}}\}$$
$$\Psi(+(a = \_)) \triangleq \forall\eta, \rho^{\{a\}}.\{a : \mathrm{Abs}; \rho^{\{a\}}\} \to \eta \to \{a : \mathrm{Pre}(\eta); \rho^{\{a\}}\}$$

FIGURE 2.20 – La fonction Ψ pour les manipulations de messages structurés

Le typage des messages. Les jugements de de typage pour les messages sont de la forme $\Gamma \vdash M : T$ avec Γ l'environement de typage utilisé pour typer le message M et lui donner T comme type. Les règles de typage définissant les types des messages sont présentées Figure 2.21. Les messages formant un lambda-calcul sans abstraction mais avec constantes et messages structurés, leur typage est grandement inspiré de [90]. Ainsi, la règle T :Var est utilisée pour typer les variables, la règle T :Const type les constantes, T :Message type les messages structurés, T :App type l'application, et les deux règles T :Inst et T :Gen gèrent la manipulation des schémas de type.

$$\text{T :Var}\quad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{T :Const}\quad \Gamma \vdash c : \Psi(c)$$

$$\text{T :Message}\quad \frac{\forall 0 < i \leq n, \Gamma \vdash M_i : E_i}{\Gamma \vdash \{a_1 = M_1; \ldots; a_n = M_n\} : \{a_1 : \mathrm{Pre}(E_1); \ldots; a_n : \mathrm{Pre}(E_n); \mathrm{Abs}\}}$$

$$\text{T :App}\quad \frac{\Gamma \vdash M_1 : E \to E' \qquad \Gamma \vdash M_2 : E}{\Gamma \vdash (M_1\ M_2) : E'} \qquad \text{T :Inst}\quad \frac{\Gamma \vdash M : \forall \alpha.T}{\Gamma \vdash M : T\{^{E}/_{\alpha}\}}$$

$$\text{T :Gen}\quad \frac{\Gamma \vdash M : T \qquad \alpha \in fv(T) \setminus fv(\Gamma)}{\Gamma \vdash M : \forall \alpha.T}$$

Figure 2.21 – Les règles de typage pour les messages

Le typage des composants. Suivant l'approche de [22, 85, 73], nos types de processus ne sont pas propres à une construction spécifique d'un assemblage, mais global à tout les éléments d'un processus. Dans notre cas, comme nous n'avons pas de restriction de nom [77], le type est global à tout le programme typé. Par exemple, si un assemblage D est composé de deux sous-parties ($D \triangleq D_1 \mid D_2$), alors tout type S valide pour D l'est aussi pour chacune des deux sous-parties. Cela se traduit par des jugements de la forme $S, \Gamma \vdash D$ (signifiant ainsi que S est bien global) et des règles de typage qui contraignent la forme de ce type global, plutôt que de lui donner une syntaxe précise, comme cela est fait pour la partie ML du calcul. Par exemple, la règle T :Canal impose qu'un type valide S pour un envoi de message $\overline{e}\langle M \rangle$ doit contenir $e : (T)$, où T est un type de M. Mais il n'existe aucune autre contrainte sur S, qui peut inclure d'autre canaux transportant d'autres types de données. Un autre exemple est

la règle T :Zero, qui n'impose aucune contrainte sur le type du processus vide. La procédure de routage est typée à l'aide de deux règles, dérivées de l'*analyse intensionnel de type* [34]. La première, T :IfPre1 est utilisée lorsque le message en paramètre possède le champ 'a' : dans ce cas, le routage correspond à l'envoi de message $\overline{s_1}\langle M\rangle$: le typage demande donc que $s_1 : (T)$ soit inclus dans son type (avec T étant le type de M). La seconde s'applique lorsque le champ 'a' n'est pas présent dans le message, et un type valide pour ce routage doit donc contenir $s_2 : (T)$. La règle T :Recepteur type les recepteurs, et s'assure que pour tout les types d'entrée de ces derniers, c'est à dire tous les messages que le recepteur va devoir traiter, les messages de sortie générés par ces entrées sont bien pris en compte dans le type. Finalement, T :Bang type la réplication, et T :Boîte type les composants.

$$\text{T :Canal}\quad \frac{\Gamma \vdash M : T \qquad T \in S(s)}{S, \Gamma \vdash \overline{s}\langle M\rangle}$$

$$\text{T :IfPre1}\quad \frac{\Gamma \vdash M : T \qquad \Gamma \vdash T \Leftarrow \forall \eta, \rho^{\{a\}}.\{a : \mathrm{Pre}(\eta); \rho^{\{a\}}\} \qquad T \in S(s_1)}{S, \Gamma \vdash \mathrm{IfPre}(a, M, s_1, s_2)}$$

$$\text{T :IfPre2}\quad \frac{\Gamma \vdash M : T \qquad \Gamma \vdash T \Leftarrow \forall \rho^{\{a\}}.\{a : \mathrm{Abs}; \rho^{\{a\}}\} \qquad T \in S(s_2)}{S, \Gamma \vdash \mathrm{IfPre}(a, M, s_1, s_2)}$$

$$\text{T :Recepteur}\quad \frac{e \in dc(S) \qquad \forall T \in S(e), \forall 0 < i \leq n, (S, \Gamma \uplus \{x : T\} \vdash B_i)}{S, \Gamma \vdash e(x).(B_1 \mid \ldots \mid B_n)}$$

$$\text{T :Parallèle}\quad \frac{S, \Gamma \vdash D_1 \qquad S, \Gamma \vdash D_2}{S, \Gamma \vdash D_1 \mid D_2} \qquad \text{T :Zero}\quad S, \Gamma \vdash 0 \qquad \text{T :Bang}\quad \frac{S, \Gamma \vdash B}{S, \Gamma \vdash !B} \qquad \text{T :Boîte}\quad \frac{S, \Gamma \vdash D}{S, \Gamma \vdash b[D]}$$

Figure 2.22 – Les règles de typage pour les processus

De manière similaire à la syntaxe des types, la partie concernant les messages des règles de typage est assez conventionnelle, et n'apporte aucune originalité par rapport aux travaux existant, de même que l'équivalence structurelle, ou les substitutions. L'intérêt de ce système de type se trouve encore une fois du côté des processus, et concerne les règles (T :IfPre1), (T :IfPre2), (T :Boîte) et principalement (T :Recepteur). Les règles gérant le routage définissent le comportement de la procédure de routage ayant un unique message en paramètre. Ces règles sont donc simples, et présentent cette procédure comme une fonction

sur les messages, similaire donc à une constante du calcul (la seule différence étant que le résultat de son application est un processus, et non pas un message). L'intérêt du routage, comme nous l'avons discuté en début de cette partie, et de pouvoir avoir différents types de message sur un même canal, ce qui est pris en compte par la règle (T :Recepteur) : cette règle, qui ressemble à une règle d'application généralisée, permet à un processus d'avoir plusieurs types de message en entrée, et de les gérer indépendamment. Considérons par exemple l'assemblage décrit Figure 2.23, qui représente une simple instanciation de l'exemple précédent (Figure 2.17), avec comme encodeur le composant $\{+\texttt{tmp} = 1\}$. Notons $T_1 \triangleq \{\texttt{int} : \text{Pre(int)}; \text{Abs}\}$ et $T_2 \triangleq \{\texttt{Str} : \text{Pre(string)}; \text{Abs}\}$: cet as-

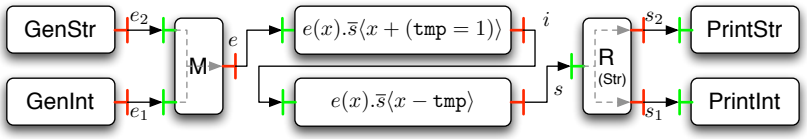


Figure 2.23

semblage est typable par

$$S \triangleq \begin{pmatrix} e_1 : (T_1) \cup e_2 : (T_2) \cup e : (T_1) \cup e : (T_2) \\ \cup\, i : (\{\texttt{int} : \text{Pre(int)}; \texttt{tmp} : \text{Pre(int)}; \text{Abs}\}) \\ \cup\, i : (\{\texttt{Str} : \text{Pre(string)}; \texttt{tmp} : \text{Pre(int)}; \text{Abs}\}) \\ \cup\, s : (T_1) \cup s : (T_2) \cup s_1 : (T_1) \cup s_2 : (T_2) \end{pmatrix}$$

Considérons l'entrée du composant `Encode` : $S(e)$ consiste en l'ensemble $\{T_1, T_2\}$. Utilisant la règle de typage (T :Recepteur), nous pouvons appliquer le composant à chacun de ces types de manière indépendante, ce qui aurait été impossible avec une règle d'application classique. De fait, cet assemblage n'est typable dans aucun des systèmes de type que nous avons cités.

En ce qui concerne la règle de typage (T :Boîte), nous pouvons voir qu'elle n'intègre pas de gestion de l'isolation inhérente à la structure de composant. Ceci est dû au mode de communication de ces mêmes composants, hérité du kell-calcul, qui autorise librement les communications entre les composants pères et leurs fils. Pour prendre en compte l'isolation, il faudrait différencier les canaux utilisés par les pères de ceux utilisés par les fils, faire de même avec les types de données transportés par les canaux commun, ce qui rajouterait une lourdeur dans le système de type, aussi bien dans la syntaxe que dans les règles. Nous avons choisi de ne pas intégrer une telle lourdeur dans notre système, et de reporter cette isolation à un travail ultérieur, sur un calcul permettant de la mettre en place plus simplement. Finalement, notons que nous ne définissons pas de schémas de type pour les processus. Nous n'en avons en effet pas besoin, car notre calcul ne comporte pas de contruction comme le `let` de ML qui nécessite la généralisation des types. Nous avons des schémas de types pour les messages

par contre, pour permettre l'utilisation de constantes fonctionnelles, comme les opérateurs sur les messages structurés.

2.2.3 Exemples

Notre système de type à été construit pour pouvoir manipuler correctement la transmission de messages entre composants, les opérations sur les messages structurés, ainsi que le routage. Ceci est illustré Figure 2.24 qui donne des types valides pour chaque composant primitif que nous avons présenté dans notre calcul. Comme nous pouvons le voir, et suivant la même approche que

Composants	Types
$!\overline{s}\langle\{\texttt{Str} = \text{“hello”}\}\rangle$	$s : (\{\texttt{Str} : \text{Pre(string)}; \text{Abs}\})$
$!e(x).\overline{s}\langle x.\texttt{TTL}\rangle$	$e : (\{\texttt{TTL} : \text{Pre}(\eta); \rho\}) \cup s : (\eta)$
$!e(x).\overline{s}\langle x + (\texttt{TTL} = 10)\rangle$	$e : (\{\texttt{TTL} : \text{Abs}; \rho\}) \cup s : (\{\texttt{TTL} : \text{Pre(int)}; \rho\})$
$!e(x).\overline{s}\langle x - \texttt{TTL}\rangle$	$e : (\{\texttt{TTL} : \text{Pre}(\eta); \rho\}) \cup s : (\{\texttt{TTL} : \text{Abs}; \rho\})$
$!e_1(x).\text{IfPre}(\texttt{Str}, x, s_1, s_2)$	$e : (\{\texttt{Str} : \text{Pre}(\eta); \rho\}) \cup e : (\{\texttt{Str} : \text{Abs}; \rho'\})$ $\cup s_1 : (\{\texttt{Str} : \text{Pre}(\eta); \rho\}) \cup s_2 : (\{\texttt{Str} : \text{Abs}; \rho'\})$
$!e_1(x).\overline{s}\langle x\rangle \mid !e_2(x).\overline{s}\langle x\rangle$	$e_1 : (\eta_1) \cup e_2 : (\eta_2) \cup s : (\eta_1) \cup s : (\eta_2)$

FIGURE 2.24 – Exemples de types pour composants primitifs

beaucoup d'autres systèmes de types pour calculs de processus [117, 73, 85], nous ne faisons pas la différence entre la partie entrée et sortie d'un type, comme cela est fait pour les fonctions en ML. Ainsi, l'access au champ `Str` d'un message est typé $\forall\eta,\rho.\{\texttt{Str} : \mathrm{Pre}(\eta);\rho\} \to \eta$ en ML, et $e : (\{\texttt{Str} : \mathrm{Pre}(\eta);\rho\}) \cup s : (\eta)$ dans notre système. Nous pouvons aussi remarquer que nous n'avons pas de schémas de type pour généraliser nos types de composant. Cela est dû, comme pour beaucoup d'autres calculs de processus, au fait que nous n'avons pas de lieurs dans notre calcul tels que le `let` de ML ou le `def` du join-calcul, et donc, la généralisation des types n'est pas utile dans notre cas. Le type que nous donnons au routeur montre que celui-ci accepte deux sortes de message en entrée : le type union $e : (\{\texttt{Str} : \mathrm{Pre}(\eta);\rho\}) \cup e : (\{\texttt{Str} : \mathrm{Abs};\rho\})$ donne deux types de message au canal e, correspondant aux deux paramètres possibles de la procédure de routage. De plus, le routeur différencie ces deux types de paramètre pour les envoyer sur des canaux de sortie différents, ce qui est exprimé dans les types par $s_1 : (\{\texttt{Str} : \mathrm{Pre}(\eta);\rho\}) \cup s_2 : (\{\texttt{Str} : \mathrm{Abs};\rho\})$. Finalement, le type du multiplexeur est similaire au type du routeur, avec la déclaration $e_1 : (\eta_1) \cup e_2 : (\eta_2)$ représentant les deux interfaces d'entrée du composant, et $s : (\eta_1) \cup s : (\eta_2)$ signifiant que le deux types de messages reçus sont transmis intégralement sur le canal s.

Nous illustrons ces différents types par deux exemples de dérivations, présentant respectivement le typage du composant `AcsTTL` (Figure 2.25) et du routeur `R(Str)` (Figure 2.26). Ces deux dérivations de type, bien qu'un peu longues,

$$
\dfrac{\dfrac{\dfrac{\dfrac{x : \{\texttt{TTL} : \mathrm{Pre}(\eta);\rho\} \vdash x : \{\texttt{TTL} : \mathrm{Pre}(\eta);\mathrm{Abs}\} \qquad \dfrac{\Psi(.\texttt{TTL}) = \forall\eta,\rho.\{\texttt{TTL} : \mathrm{Pre}(\eta);\rho\} \to \eta}{x : \{\texttt{TTL} : \mathrm{Pre}(\eta);\rho\} \vdash .\texttt{TTL} : .\texttt{TTL} : \mathrm{Pre}(\eta);\rho\} \to \eta}}{x : \{\texttt{TTL} : \mathrm{Pre}(\eta);\rho\} \vdash x.\texttt{TTL} : \eta} \qquad \eta \in S(s)}{S, x : \{\texttt{TTL} : \mathrm{Pre}(\eta);\rho\} \vdash \overline{s}\langle x.a\rangle}}{S, \emptyset \vdash e(x).\overline{s}\langle x.a\rangle}}{S, \emptyset \vdash b[e(x).\overline{s}\langle x.a\rangle]}
$$

avec $S \triangleq e : (\{\texttt{TTL} : \mathrm{Pre}(\eta);\rho\}) \cup s : (\eta)$

FIGURE 2.25 – Typage du composant `AcsTTL`

illustrent assez bien les deux caractéristiques principales de notre système de type : les opérations sur les messages sont typées en suivant la même approche

$$
\dfrac{\dfrac{\dfrac{x : \{\texttt{Str} : \mathrm{Abs}; \rho'\} \qquad \{\texttt{Str} : \mathrm{Abs}; \rho'\} \in S(s_2)}{S, x : \{\texttt{Str} : \mathrm{Abs}; \rho'\} \vdash \mathrm{IfPre}(\texttt{TTL}, x, s_1, s_2)} \quad \vdots \quad \dfrac{x : \{\texttt{Str} : \mathrm{Pre}(\eta); \rho\} \qquad \{\texttt{Str} : \mathrm{Pre}(\eta); \rho\} \in S(s_1)}{S, x : \{\texttt{Str} : \mathrm{Pre}(\eta); \rho\} \vdash \mathrm{IfPre}(\texttt{TTL}, x, s_1, s_2)}}{S, \emptyset \vdash e(x).\mathrm{IfPre}(\texttt{TTL}, x, s_1, s_2)}}{S, \emptyset \vdash b[e(x).\mathrm{IfPre}(\texttt{TTL}, x, s_1, s_2)]}
$$

$$
\begin{aligned}
\text{avec } S \triangleq\ & e : (\{\texttt{Str} : \mathrm{Pre}(\eta); \rho\}) \cup e : (\{\texttt{Str} : \mathrm{Abs}; \rho'\}) \\
& \cup\, s_1 : (\{\texttt{Str} : \mathrm{Pre}(\eta); \rho\}) \cup s_2 : (\{\texttt{Str} : \mathrm{Abs}; \rho'\})
\end{aligned}
$$

FIGURE 2.26 – Typage du routeur

que les langages fonctionels à la ML, et la transition avec la partie processus de notre calcul se fait par la règle (T :RECEPTEUR) qui permet à un composant de recevoir différents types de messages en entrée. Cette capacité de notre système de type s'apparente à la généralisation des types, avec toutefois une différence notable : la généralisation permet que chaque appel d'une fonction — ou en d'autres tèrmes chaque *instance syntaxique* de la fonction — soit typé indépendamment. Dans notre cas, nous permettons qu'une unique instance syntaxique d'un composant puisse avoir différents types, toujours dans le but qu'un unique composant de routage puisse effectivement router différents types de messages.

Un des defauts de notre système de type est qu'il ne prend pas en compte l'isolation inhérente à la structure hiérarchique des composants, tout en gardant trace de tout les messages transitant sur chacun des canaux. Ainsi, le type d'un assemblage devient très vite illisible, car même pour des assemblages de petites tailles, il peut déclarer un très grand nombre de canaux. Afin de pouvoir tout de même pouvoir continuer notre présentation d'exemples de typage, nous intégrons les types à nos figures, pointant directement chaque canal par le ou les types y transitant. De fait, nous avons renoncé à présenter la dérivation de type des composants qui vont suivre, car même si dans le principe, la construction d'une telle dérivation est assez simple, sa complexité syntaxique en limite grandement la compréhension et l'intérêt. Notre premier exemple, Figure 2.27, présente le typage du composant que nous avons introduit Figure 2.14 : tout les canaux sont typés par int, comme les seules données échangées par les composants sont des

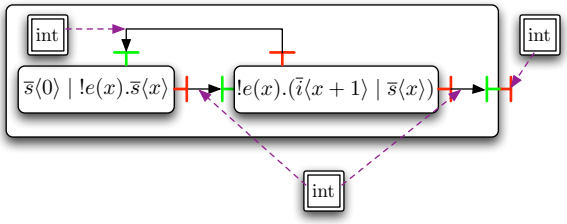


FIGURE 2.27

entiers. Notre second exemple, présenté Figure 2.28, possède une architecture plus simple, sans boucle, mais comporte beaucoup plus de types différents sur ses canaux. Néanmoins, le comportement, ainsi que le typage de ce composant reste assez élémentaire, et il est asse facile de vérifier que nos annotations de types sont valides. Suivant la même approche que pour les exemples du calcul,

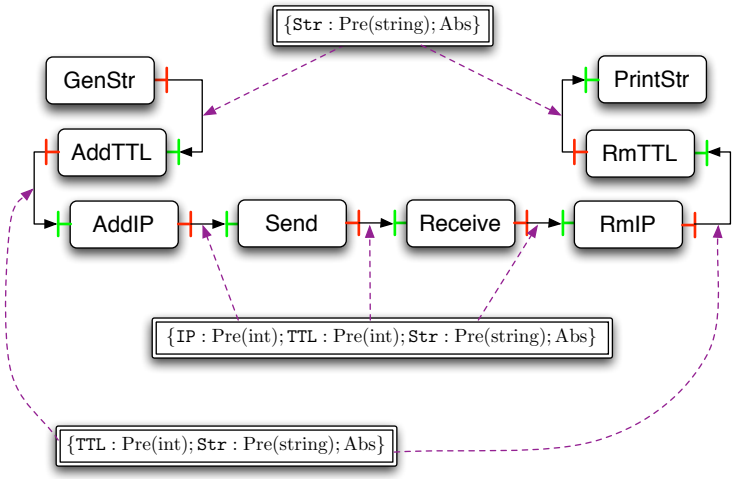


FIGURE 2.28

nous continuons notre série d'exemple par le composant 2.29 présentant le typage du composant composite permettant la communication entre deux entités au travers d'un reseau. Nous pouvons y voir que les types spécifient bien que le message en entrée doit avoir une structure bien particulière, sans champ `TTL` ni `IP` afin que l'ajout de ces champs par cet assemblage se fasse sans souci. De plus, nous pouvons remarquer que le type de sortie de l'assemblage est identique à celui spécifié en entrée : le message n'est en effet pas modifié par son passage dans

le composant. L'exemple suivant (Figure 2.30) illustre de manière graphique

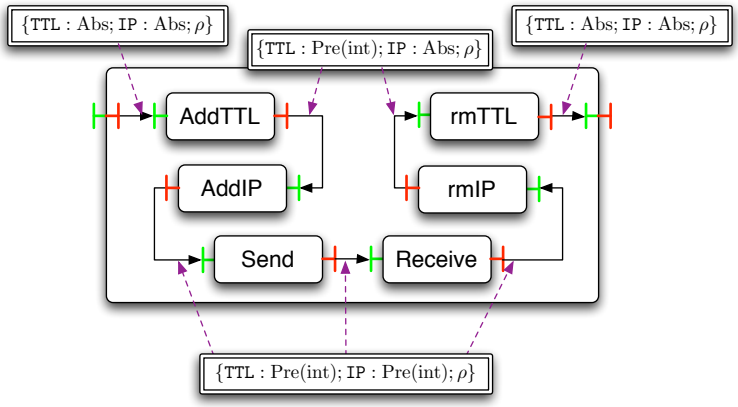


FIGURE 2.29

comment le multiplexeur et le routeur agissent sur les types. Le composant `GenStr` produit en sortie un message de structure {`Str` : Pre(string); Abs}, alors que le composant `GenInt` génère un message de structure totalement différente ({`Int` : Pre(int); Abs}). Bien que totalement différents, ces types sont transmis à l'identique par le multiplexeur, et placé sur un même canal. De manière similaire à la réduction du calcul, ces types sont envoyés au composant `Transmit`, puis au routeur qui les distingue et les envoie chacun au composant d'affichage correspondant à leur structure. Notons toutefois que cet assemblage est bien typé car

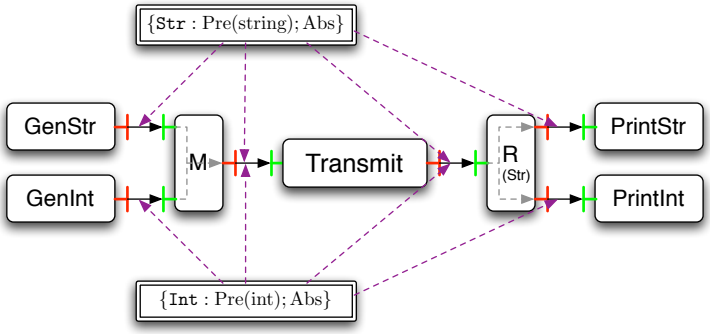


FIGURE 2.30

les messages sont des structures : le routeur n'accepte pas en entrée de messages

tels que des entiers ou des chaînes de caractères. Notre dernier exemple reprend l'assemblage présenté Figure 2.15, et type chacun de ses canaux. La Figure 2.31 résultant de ce typage commence à être parsemée de beaucoup d'annotations, en comparaison à nos premiers exemples. Néanmoins, le typage de chacun des canaux de cet assemblage reste assez simple.

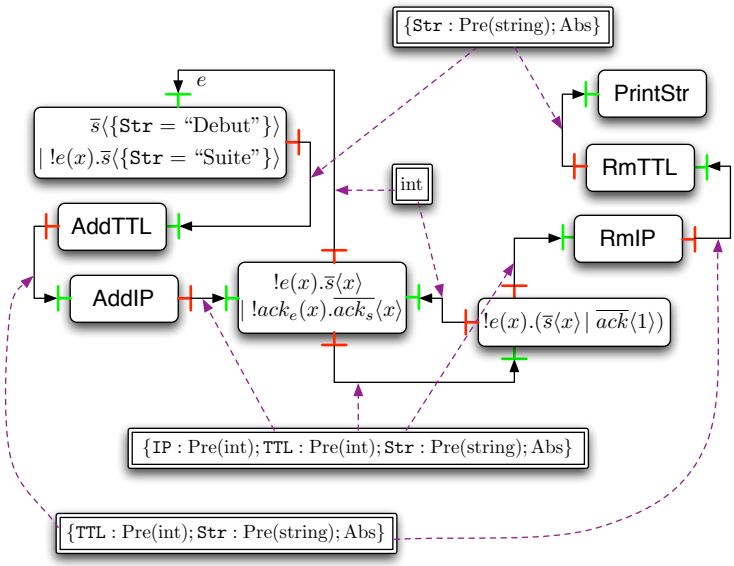


FIGURE 2.31

2.3 Discussion

Cette série d'exemples montre que tous les composants que nous avons introduits dans la présentation du calcul – à part celui décrit Figure 2.16 que nous nommons TheLoop – sont typables avec des types relativement intuitifs, même si ces derniers sont syntaxiquement complexes. Avant de reprendre l'assemblage TheLoop et d'expliquer pourquoi nous n'avons pas proposé un type pour ce composant, montrons tout d'abord que notre système de type est correct et vérifie bien que certains types d'erreurs ne sont pas présents dans un assemblage typé. Nous montrons cela de manière classique, par une définition du type d'erreur visé, et la présentation des deux théorèmes de correction et de stabilité.

Définition 3. *Un programme D a une* erreur *si et seulement si une des conditions suivantes est validée :*

– *Il existe un contexte E, deux valeurs v et v' tels que $D = E[(v\ v')]$ et v' n'est pas un argument valide de v,* i.e. $(v, v') \notin \texttt{match}$.
– *Il existe un contexte E, une valeur v qui n'est pas un message structuré, un nom de champ a et deux canaux s_1 et s_2 tel que $D = E[\mathrm{IfPre}(a, v, s_1, s_2)]$.*

Théorème 1 (Correction)**.** *Supposons donné un jugement valide de typage $\emptyset \vdash M : T$ (resp. $S, \emptyset \vdash D$). Alors M (resp. D) ne contient aucune ereur.*

Théorème 2 (Stabilité)**.** *Supposons donné un jugement valide de type $\Gamma \vdash M : T$ (resp. $S, \Gamma \vdash D$). Alors, pour tout message M' (resp. processus D) tel que $M \rhd M'$ (resp. $D \rhd D'$), le jugement de type $\Gamma \vdash M' : T$ (resp. $S, \Gamma \vdash D$) est valide.*

Les preuves de ces théorèmes, bien qu'existantes, ne sont pas présentées dans ce document. L'intérêt de celles-ci est en effet assez limité, car elles sont directement basées sur les preuves classiques des systèmes de type à la ML.

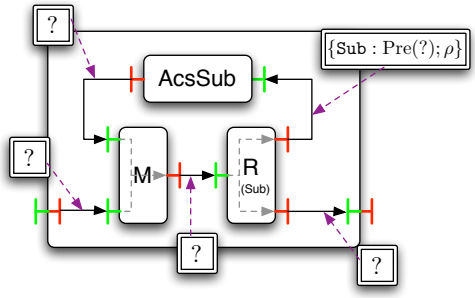


FIGURE 2.32 – Le composant TheLoop

Types principaux. La définition d'un type pour l'assemblage TheLoop, que nous rappelons Figure 2.32, est assez délicat. En effet, ce composant est trivialement typable par le type associant à chaque canal un ensemble de message vide, comme cela est illustré Figure 2.33. Néanmoins, ce type ne nous informe que de quels canaux sont utilisés par le programme, et non pas quelle forme doivent avoir les messages en entrée du composant. De fait, il n'existe pas de type donnant ce genre d'informations pour cet assemblage. Informellement, ce composant donne en sortie le message structuré placé sous un nombre arbitraire de champ `Sub`, et cette propriété n'est pas exprimée par nos types, bien que l'assemblage soit typable. De fait, il n'existe en général pas de types principaux pour un composant donné : considérons par exemple le type

$$S \triangleq e : (\{\texttt{TTL} : \mathrm{Pre}(\mathrm{int}); \mathrm{Abs}\}) \cup s : (\mathrm{int}) \cup e : (\{\texttt{TTL} : \mathrm{Pre}(\eta); \rho\}) \cup s : (\eta)$$

$$\dfrac{\dfrac{e \cup i \cup s, \emptyset \vdash i(x).\overline{e}\langle x.\texttt{Sub}\rangle \qquad e \cup i \cup s, \emptyset \vdash e(x).\mathrm{IfPre}(\texttt{Sub}, x, i, s)}{e \cup i \cup s, \emptyset \vdash e(x).\mathrm{IfPre}(\texttt{Sub}, x, i, s) \mid i(x).\overline{e}\langle x.\texttt{Sub}\rangle}}{e \cup i \cup s, \emptyset \vdash b[e(x).\mathrm{IfPre}(\texttt{Sub}, x, i, s) \mid i(x).\overline{e}\langle x.\texttt{Sub}\rangle]}$$

FIGURE 2.33

Il est assez facile, par une dérivation de type similaire à celle présentée 2.25 de voir que ce type est valide pour le composant `AcsTTL`, mais comme il comporte plusieurs déclarations du canal e, il n'est pas unifiable avec le type que nous avons donné à ce composant, et qui pourtant semble être principal (car directement issu du type principal de l'operateur `.TTL`). Nous avons considéré, pour obtenir des types principaux, de généraliser les substitutions pour leur permettre de *dupliquer* les définitions de canaux, (permettant ainsi d'unifier les deux types précédents) mais le problème subsiste tout de même (voir annexe A.1). Dans les langages comme ML, il est usuel d'utiliser les types principaux pour caractériser les éléments typables du langage. Ceci n'est pas possible dans notre cas.

Types minimaux. Nous avons toutefois trouvé une nouvelle façon de caractériser ces objets dans notre calcul. Nous avons remarqué que tout les processus typables admettent un unique (modulo équivalence structurelle) type *minimal* qui, informellement, décrit que les canaux utilisé par l'assemblage ainsi que les types des messages qui transiteront effectivement sur ces canaux.

Définition 4. *Supposons donné un environnement de typage* Γ *et un processus* D*.* D *a un* type minimal S *pour* Γ *si et seulement si :*
- *le jugement de type* $S, \Gamma \vdash D$ *est valide.*
- *Pour tout type* S' *tel que* $S', \Gamma \vdash D'$*, nous avons* $\Gamma : S' \Leftarrow S$*.*
- $fv(S) \setminus fv(\Gamma) = \emptyset$*.*

De fait, la Figure 2.33 donne le type minimal pour le composant TheLoop. Ce type n'est malheureusement pas très informatif sur quelles spécificités les messages d'entrée et de sortie doivent avoir : tout les composants `AcsTTL`, `RmTTL` et `addTTL` ont pour type minimal $e \cup s$. Seul échappe à la règle le composant `GenTTL`, car ce composant implique un message qui est effectivement envoyé sur sa sortie : le type que nous avons donné pour cet assemblage est bien son type minimal.

L'inférence de type. Finalement une considération importante lors de la définition de ce système de type était de pouvoir construire un algorithme d'inférence, permettant ainsi une utilisation simple de nos types, malgré leur grande complexité syntaxique. Malheureusement, nous avons prouvé que l'inférence de type est dans notre cas indécidable, en encodant le problème de correspondance de Post (PCP) [86] dans notre notion de typage. Nous avons été assez surpris de ce résultat, car nous avons construit notre système de type en utilisant des notions totalement décidables, et même, le typage des composants nous semble assez trivial. En fait, nous avons commencé à douter de l'existence d'un algorithme d'inférence après beaucoup de recherches et la découverte de composants tels que TheLoop, ou d'autres basés sur le même style d'utilisation du routeur dans une boucle. Nous avons pu ainsi construire des composants typables qui pouvait *additionner*, *multiplier*, *soustraire* des types [3] comme cela est fait Figure 2.34 et 2.35. Nous étions proches de pouvoir exprimer la logique de

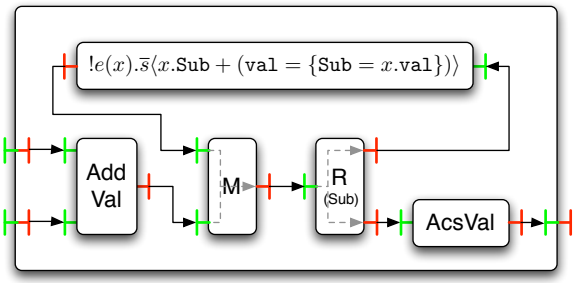
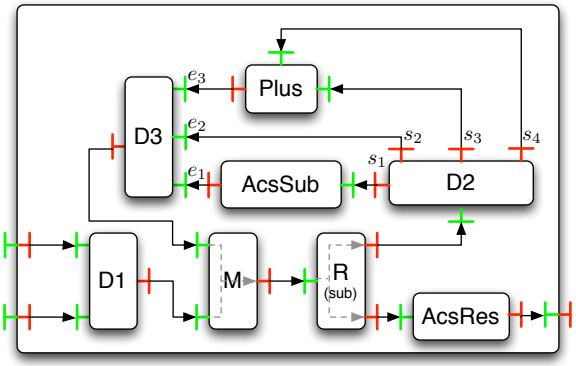


FIGURE 2.34 – Composant pour l'addition de deux types

Peano dans nos types, et, de fait, notre encodage de PCP, présenté en annexe, est construit en utilisant le même genre d'assemblages, basé sur un routeur au centre d'une boucle. Par conséquence, nous avons longtemps cru que la cause de l'indécidabilité de l'inférence de type était due à notre routeur basé sur les messages structurés. En effet, la nature hiérarchique de ces messages permet de construire des boucles telles que TheLoop où un même routeur est utilisé sur un message mais aussi ses sous-messages. Mais des récents développements dans nos recherches nous font penser qu'en réalité, le routeur n'est pas vraiement en cause dans l'indécidabilité de l'inférence. Une structure hiérarchique comme base du routage semble en effet possible tout en ayant une possibilité d'inférence de type : ce serait en fait la structure de la boucle qui serait en cause, et notre système de type est trop laxe en permettant de construire de telles boucles. Malgré l'indécidabilité de l'inférence de type, nous avons néanmoins construit

3. nous pouvons encoder les entiers dans les types en représentant n par une hauteur de n champ $\mathtt{Sub}$: $0 \triangleq \{\mathrm{Abs}\}$, $1 \triangleq \{\mathtt{Sub} = \mathrm{Pre}(\{\mathrm{Abs}\}); \mathrm{Abs}\}$, etc



avec `Plus` additionnant deux types, et :

$$\texttt{D1} \triangleq e_1(x).e_2(y).\overline{s}\langle x + (\texttt{val} = y) + (\texttt{res} = \{\})\rangle$$
$$\texttt{D2} \triangleq e(x).(\overline{s_1}\langle x - \texttt{val} - \texttt{res}\rangle \mid \overline{s_2}\langle x.\texttt{val}\rangle \mid \overline{s_3}\langle x.\texttt{val}\rangle \mid \overline{s_4}\langle x.\texttt{res}\rangle)$$
$$\texttt{D3} \triangleq e_1(x).e_2(y).e_3(z).\overline{s}\langle x + (\texttt{val} = y) + (\texttt{res} = z)\rangle$$

FIGURE 2.35 – Multiplication de deux types

deux programmes : un semi-algorithme, calculant le type minimal d'un assemblage dans la plupart des cas, mais ayant une exécution infinie pour des assemblages tels que celui présenté Figure 2.36 (où le composant `D` *duplique* le message reçus en le renvoyant sur ses deux interfaces de sortie). Cet assemblage

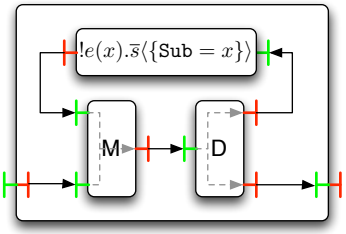


FIGURE 2.36

est en tout point correct et ne génère aucune erreur durant son exécution, mais en fait, produit un nombre infini de messages de types différents. Notre semi-algorithme, en essayant de calculer tout les types produits, ne finira donc jamais. Notons toutefois que ce genre d'erreur de conception dans un assemblage est as-

sez rare, et ce semi-algorithme est utilisable, au même titre que la récursion polymorphe dans le langage OCAML [81, 50]. Notre second programme est un algorithme d'inférence utilisant des annotations de types. En fait, l'exécution infinie de notre semi-algorithme est dûe à la méconnaissance du comportement des boucles : en annnotant celles-ci par leur type, l'algorithme n'a plus à essayer de le determiner, et est ainsi assuré de terminer. Ces deux programmes sont décrit dans [69].

2.4 Conclusion

Dans ce chapitre, nous avons présenté un calcul simple impliquant à la fois des processus, des localités, et des messages à la ML. Nous avons choisi d'intégrer dans ce calcul une procédure de routage – ce qui à notre connaissance n'a jamais été fait jusqu'à présent – afin d'étudier, du point de vu des types, le comportement d'une telle construction qui reste présente dans la majeur partie des protocoles de communication. Nous avons défini sur ce calcul un système de type assez intuitif et permettant d'avoir plusieurs types de message sur un même canal. Malgré sa relative simplicité, ce système de type ne possède ni types principaux ni algorithme d'inférence de type. Il serait nénamoins intéressant d'étudier des évolutions de ce système de type, en y incluant par exemple l'isolation des composants, ou même des procédures de routages plus évoluées en intégrant les primitives décrites dans [33]. Nous avons toutefois choisi une autre direction de recherche. En effet, nous nous sommes plus intéressé à l'obtention des propriétés classique d'un système de type, telles que les types principaux, ou à l'inférence de type (cette dernière, seulement présente dans ML, n'étant pas si classique).

Chapitre 3

Routage sémantique

Suite à notre première approche, deux possibilités s'offraient à nous : soit nous utilisions ce travail comme base pour la définition d'un nouveau système de type construit sur un calcul plus complexe et comportant par exemple de l'ordre supérieur ou des primitives de manipulation dynamique de code (le typage de ce genre de calcul étant la motivation principale de ce travail de thèse) ; soit nous restions sur un calcul assez basique, et travaillions sur un nouveau système de type permettant toujours le routage [1], mais ayant de meilleures propriétés que celui que nous venons de définir. Nous avons opté pour la seconde possibilité, et ce, pour plusieurs raisons. Tout d'abord, le système de type de notre première version comporte nombre de défauts, comme nous l'avons présenté dans le chapitre 2 et l'annexe A : (i) l'inférence de type est indécidable ; (ii) il n'existe pas de type principal pour tous les composants ; (iii) le type d'un composant n'est pas une caractérisation du comportement de ce dernier. De plus, continuer nos recherches sur un calcul simple permettait de modifier celui de notre première approche et de l'améliorer en simplifiant sa syntaxe ou en modifiant par exemple le mode de communication entre composants. En effet, nous avons déjà discuté du problème que suscitait la communication à la kell sur la définition d'exemples, ainsi que la prise en compte de l'isolation dans un système de type. Il serait donc intéressant de mettre à profit un nouveau calcul pour tester un nouveau mode de communication entre composants. Finalement, nous considérons la syntaxe de notre calcul assez lourde, car elle combine à la fois des éléments de ML et des éléments de systèmes distribués. Les opérations sur les messages structurés de ML sont en effet nécessaires dans une approche où le routage est basé sur ces structures, mais elles peuvent certainement être abstraites pour éviter de devoir inclure toute une partie de ML dans notre calcul.

C'est à partir de ces considérations que nous avons construit notre seconde approche. Nous avons tout d'abord défini un nouveau calcul se distinguant foncièrement de notre calcul précédent, du kell, et même du π-calcul. En ef-

1. la prise en compte du routage dans un système de type est une innovation majeure de notre premier travail, et nous la considérons comme extrêmement utile, le routage étant présent dans la plupart des systèmes de communication existant

fet, notre calcul ne contient pas de processus en tant que tel, il n'est construit que sur la base de composants et de messages circulant sur des canaux de communication. Toutes les opérations de manipulation de message sont prises en compte par les composants primitifs, qui sont maintenant de simples noms sans définition dans notre langage (de manière similaire aux constantes que l'on a dans ML). La communication inter-composants se fait maintenant au travers de portes qu'un message peut franchir : les composants composites définissent un ensemble de canaux entrants et un ensemble de canaux sortants. Ces définitions agissent comme l'ouverture de portes dans la frontière d'un composant composite : les messages portés par un canal dans l'ensemble entrant sont libres d'entrer dans le composant et d'y être manipulés, et les messages internes ne peuvent sortir du composant que s'ils sont portés par des canaux présents dans l'ensemble des canaux de sortie. Ce système de communication est assez élégant, et tranche par rapport aux communications implicites entre composants père et fils du kell-calcul. De plus, il permet d'intégrer facilement l'isolation au niveau du système de type, comme nous pourrons le voir ultérieurement. Finalement, l'innovation majeure de cette version concerne le routage et le multiplexage. Notre travail précédent, et principalement la preuve de l'indécidabilité de l'inférence de type, suggérait en effet que notre approche de ces fonctions permettait de construire des boucles assez délicates à typer. Permettre le typage de ces boucles donnait le système de type de notre première approche avec ses défauts, et empécher leur typage contraignait fortement l'expressivité de notre calcul. Nous avons donc choisi, dans ce nouveau calcul, d'utiliser une nouvelle définition des routeurs et multiplexeurs. Informellement, les routeurs et les multiplexeurs fonctionnent maintenant par paire, ou autrement dit, ces opérations sont *duales* l'une de l'autre. Pour illustrer cela, considérons un multiplexeur et un routeur de notre approche précédente. Typiquement, le multiplexeur crée un nouveau flux de donnée de sortie contenant tous les messages de tous les flux de donnée qu'il reçoit en entrée. Le routeur quant à lui, sépare son flux d'entrée en fonction de la structure des messages y transitant : cette séparation n'est en aucun cas l'opposé de l'opération de fusion opérée par les multiplexeurs. Ces deux composants ne sont donc pas *duaux* l'un de l'autre. Nous optons ici pour une approche où ces deux composants sont duaux, car cette propriété est en fait souvent présente dans les systèmes de communication (comme dans l'exemple Figure 2.17 présenté dans le chapitre précédent). Nos multiplexeurs et routeurs fonctionnent donc par paire, un multiplexeur assemblant deux flux en entrée, et le routeur associé séparant ces deux mêmes flux. Techniquement, une paire de multiplexeur et routeur associés est identifiée par un nom dit *de routage*, généralement noté r. Afin que le routeur de la paire puisse dissocier les deux flux qu'il a précédemment fusionnés, le multiplexeur *annote* les messages qu'il traite. Les messages passant par la première entrée (celle du haut) sont annotés par $\uparrow r$, et les ceux passant par la seconde (celle du bas) sont annotés par $\downarrow r$. Le routeur peut alors distinguer les messages traités par le multiplexeur, et les envoyer sur des sorties différentes après leur avoir enlevé leur annotation $\uparrow r$ ou $\downarrow r$. Notons que les annotations *de routage* que portent les messages ne se limitent pas à celles données par un unique multiplexeur : un message peut transiter par

un multiplexeur nommé r, puis par un autre r' avant d'être routé par le routeur r et le routeur r', sans contrainte sur l'ordre dans lequel ils sont placés. Notons de plus que nos multiplexeurs et routeurs sont considérés dans notre syntaxe comme des composants primitifs, c'est à dire, de simples noms [2]. Le calcul n'expose de notre approche du routage que les annotations que portent les messages ; il est par contre bien entendu que nos exemples d'assemblage feront intervenir des routeurs et multiplexeurs tels que nous les avons présentés.

3.1 Le calcul

Le calcul que nous proposons dans cette section est extrêmement épuré par rapport à notre précédente approche, et même par rapport à d'autres calculs simples, comme le kell ou les ambients. Comme nous l'avons dit, les seuls processus que contient ce calcul sont des composants mis en parallèle, et des messages annotés par des informations de routage et envoyés sur des canaux de communication. Finalement, nous n'avons pas introduit de primitive de manipulation dynamique de code dans notre nouvelle approche, pour nous concentrer principalement sur l'étude des communications inter-composants et du routage.

D	$::=$		Processus
		p	Composant primitif
	$\mid$	$c[I\,/\,O][D]$	Composant composite
	$\mid$	$\overline{e}\langle M\rangle$	Envoi de message
	$\mid$	$D_1 \mid D_2$	Composition parallèle
M	$::=$	$v^{\delta^{\emptyset}}$	Message annoté
δ^l	$::=$		Annotation de routage
		$\emptyset$	Annotation vide
	$\mid$	$\downarrow r; \delta^{l \uplus \{r\}} \quad \mid \quad \uparrow r; \delta^{l \uplus \{r\}}$	Ajout d'une annotation
v	$::=$		Message
		$\{a_1 = v_1; \ldots; a_n = v_n\}$	Message Structuré
	$\mid$	c	Constante

FIGURE 3.1 – La syntaxe du calcul

La syntaxe de notre calcul est présentée Figure 3.1. Dans celle-ci, p et ses

2. Les composants primitifs viennent avec une sémantique propre, définie par une relation mathématique. Il est clair que la sémantique des routeurs et multiplexeurs peut être exprimée de la même manière

variantes désignent un composant primitif alors que les composants composites sont nommés c. Ces derniers sont construits sur le modèle $c[I\,/\,O][D]$: (i) I est un ensemble de canaux de communication définissant les capacités d'entrée du composant ; (ii) O désigne la capacité de sortie du composant, et est lui aussi un ensemble de canaux ; et (iii) D est le code interne du composant. Les composants s'échangent des messages *routés* M via des canaux de communication (généralement notés e ou s). Ces messages sont constitués d'une valeur v (pouvant être soit une constante, soit un message structuré) qui constite le corps du message, et d'une annotation de routage. Cette dernière est formée d'une liste finie d'élements de la forme $\uparrow r$ ou $\downarrow r$, et contrainte par une règle de kinds imposant qu'un nom de routage r ne peut y apparaitre au plus qu'une seule fois (une discussion présentant la motivation de cette restriction est donnée avec nos exemples d'assemblages, décrits à la fin de cette partie). Informellement, cette annotation décrit quelles opérations de multiplexage ont été effectuées sur cette valeur v, quels routeurs pourront la traîter et quel en sera le résultat. Nous avons décidé de garder les messages structurés dans notre calcul même s'ils ne sont plus nécessaires pour la définition du routage, car ils restent des valeurs très utilisées dans les échanges entre programmes. De plus, la plupart des exemples d'assemblage de notre première approche utilisaient des messages structurés, et les avoir ici permet de comparer plus facilement nos deux approches. Finalement, nous noterons dans le reste de ce chapitre $\mathcal{P}$ (resp. $\mathcal{C}$) l'ensemble des composants primitifs (resp. l'ensemble des constantes) de notre calcul. Notons que ces ensembles ne sont nullement précisés pour le moment, rendant ainsi le calcul extrêmement souple d'un point de vu sémantique. Cette souplesse sera bien sûr largement utilisée dans nos exemples.

La sémantique opérationnelle de notre calcul est construite de manière identique à celle de notre premier calcul : les règles de réduction que nous définissons sont basées sur une équivalence structurelle entre les termes de notre syntaxe ainsi que des contextes d'évaluation et une relation donnant la sémantique des opérateurs de base du calcul. Dans notre cas, ces opérateurs sont les composants primitifs (contrairement à notre première approche, où ceux-ci étaient les constantes des messages).

L'équivalence structurelle est construite comme étant la plus petite relation d'équivalence étant une congruence et vérifiant les règles données Figure 3.2. Comme il est usuellement le cas pour les calculs de processus, cette relation rend la composition parallèle associative et commutative. De plus, les messages structurés définissant les mêmes champs et portant les mêmes valeurs sont eux aussi mis en relation. Notons que notre calcul n'inclue pas le processus vide 0 qui peut être considéré comme un composant primitif : par conséquent le parallèle n'a pas d'élément neutre ici.

Les contextes d'évaluation E sont beaucoup simplifiés par rapport à notre approche précédente, du fait que les messages sont de simples valeurs qui ne se réduisent pas. Les seuls sous-programmes se réduisant sont alors les processus à l'intérieur de composants composites, ou bien ceux en parallèle à d'autres

$$D_1 \mid D_2 \equiv D_2 \mid D_1 \qquad D_1 \mid (D_2 \mid D_3) \equiv (D_1 \mid D_2) \mid D_3$$

$$\{a_1 = v_1; \ldots; a_i = v_i; a_{i+1} = v_{i+1}; \ldots; a_n = M_n\} \equiv \{a_1 = v_1; \ldots; a_{i+1} = v_{i+1}; a_i = v_i; \ldots; a_n = v_n\} \qquad \frac{v \equiv v'}{v^\delta \equiv v'^\delta}$$

FIGURE 3.2 – La relation d'équivalence

processus. Nos contextes, qui mettent en place cette notion relativement simple de sous-programmes, sont décrits par la syntaxe suivante :

$$E \quad ::= \quad [] \quad | \quad c[I\,/\,O][E] \quad | \quad E \mid D \quad \text{Les contextes d'évaluation}$$

La forme [3] des relations `match` et `eval`, donnant la sémantique des opérateurs de base (c.à.d des composants primitifs), est toutefois plus complexes que celle donnée dans notre première approche (qui s'occupait des constantes, de sémantique naturellement plus simple). La difficulté principale de cette définition est de prendre en compte les capacités étendues d'entrée et de sortie des composants, qui peuvent émettre et recevoir plusieurs messages envoyés sur différents canaux de communication. Nous avons résolu ce problème en introduisant une syntaxe de *motifs* pouvant exprimer quels sont les paramètres valides pour un composant, ainsi que l'ensemble des messages qu'il peut envoyer : un motif est un ensemble de messages, comme le présente la déclaration suivante

$$J \quad ::= \quad | \quad \overline{e}\langle R\rangle \quad | \quad J \mid J \quad \text{Patterns}$$

La première partie de la syntaxe, vide, correspond au fait qu'un composant primitif peut ne prendre aucun message en paramètre pour émettre néanmoins un résultat, et réciproquement, un composant peut n'envoyer aucun résultat. Avec cette syntaxe de motifs J, la forme des deux relations `match` et `eval` est relativement simple à définir : (i) `match` est un ensemble de paires constituées d'un composant primitif p et d'un motif J (signifiant alors que J est un paramètre valide pour p) ; (ii) `eval` est une fonction de domaine `match` et donnant comme résultat un motif : `eval`(p, J) correspond au motif calculé par p avec le paramètre J.

Finalement, nous présentons Figure 3.3 les règles définissant la relation de réduction $\triangleright$ entre deux processus. La règle R :CONTEXTE utilise les contextes

3. Notre calcul étant en effet générique, nous ne pouvons donner une définition précise de ces deux relations. Nous ne pouvons qu'en contraindre la forme afin de pouvoir les utiliser dans la définition de notre sémantique opérationnelle. Un exemple de définition est toutefois donné en fin de partie, avec nos exemples

$$\text{R :Contexte}\quad \frac{D \rhd D'}{E[D] \rhd E[D']} \qquad \text{R :Entrée}\quad \frac{e \in I}{\overline{e}\langle M\rangle \mid c[I\,/\,O][D] \rhd c[I\,/\,O][\overline{e}\langle M\rangle \mid D]}$$

$$\text{R :Sortie}\quad \frac{s \in O}{c[I\,/\,O][\overline{s}\langle M\rangle \mid D] \rhd c[I\,/\,O][D] \mid \overline{s}\langle M\rangle} \qquad \text{R :Primitif}\quad \frac{\texttt{match}(p, J)}{J \mid p \rhd p \mid \gamma(p, J)}$$

Figure 3.3 – Operational semantic

d'évaluation afin de permettre l'exécution à l'intérieur d'un programme. Les deux règles R :Entrée et R :Sortie permettent le passage des messages à travers les composants tout en respectant l'isolation qu'apporte notre modèle de portes. La dernière règle R :Primitif utilise les deux relations `match` et `eval` pour calculer le resultat qu'envoie un composant primitif lorsqu'un paramètre valide est présent. Cet ensemble de quatre règles est bien plus simple que la réduction de notre approche précédente. Cette relative simplicité est en partie causée par notre calcul lui aussi simplifié, et basé sur les composants : toutes les constructions concernant le routage, l'accès aux messages sur les canaux et leur manipulation existent au travers des composants primitifs dans notre nouveau calcul, et leurs sémantiques doivent maintenant être définies dans `match` et `eval`. De plus, la communication inter-composants est grandement simplifiée, et nous pensons plus intuitive, grâce à ce nouveau système de portes.

3.1.1 Exemples et discussion

Une des premières séries d'exemples à laquelle nous nous sommes attelés s'est intéressée au test de notre nouveau mode de communication entre composants. La motivation initiale de la mise en place de *portes* sur les composants était d'avoir une notion claire de l'isolation et d'éviter les problèmes de nommage que l'on a eu dans notre première approche.

Portes et isolation. Il est assez évident de voir qu'en effet, nous avons maintenant une notion simple d'isolation : un message transitant sur un canal e peut entrer dans (ou sortir de) un composant $c[I\,/\,O][D]$ si la porte correspondante est ouverte, c'est à dire si $e \in I$ (resp. $e \in O$). Cela est bien plus simple que ce qui était proposé dans notre première approche, où l'isolation dépendait du contexte dans lequel chaque composant s'exécutait [4]. Par contre, ces portes

4. un message ne peut franchir une frontière de composant que s'il existe à sa destination un processus pouvant le traiter. Ainsi la destination définit un contexte qui autorise ou pas la

ont deux défauts majeurs : (i) le nommage des canaux reste difficile lors de la conception de programme ; et (ii) il est impossible d'assurer une communication sécurisée entre deux processus distants (ceci sera détaillé Chapitre 5). Le problème de nommage que nous avons ici est bien moins important que celui que nous avions dans la version précédente du calcul, où il fallait contrôler l'ensemble des canaux d'un composant et de ses fils afin de s'assurer qu'il n'y ait pas d'erreur. Néanmoins, il reste impossible de définir un composant sans connaître le contexte dans lequel il s'exécutera. Considérons par exemple un composant simple c_1, avec une entrée e et une sortie s. Ce composant attend donc un ou des messages sur e, que l'on peut supposer générés par un autre composant c_2, ayant lui aussi une entrée e et une sortie s. Avec notre système de porte, nous avons besoin de constructions non triviales et impliquant des composants primitifs transmettant les messages d'un canal vers un autre pour permettre à ces deux composants de communiquer. Si nous notons $\mathrm{Trans}[e \to s]$ le composant prenant les messages sur e et les envoyant sur s, un des moyens les plus simples pour faire communiquer c_1 et c_2 serait :

$$c[e_1 \,/\, s_1][\mathrm{Trans}[e_1 \to e] \mid c_2 \mid \mathrm{Trans}[s \to s_1]] \mid \mathrm{Trans}[s_1 \to e] \mid c_1$$

Cette structure est extrêmement complexe, vu que son but est simplement de permettre la communication entre deux composants. Alors qu'avec un nommage idéal des canaux utilisés par les deux composants c_1 (qui écouterait sur i et enverrait sur s) et c_2 (qui écouterait sur e et enverrait sur i)), l'assemblage serait simplement $c_2 \mid c_1$. Il existe une convention de nommage permettant de s'assurer que les communications s'effectuent comme attendues, mais comme pour notre approche précédente, nous préférons utiliser notre charte graphique pour décrire nos exemples, plutôt que d'utiliser une syntaxe avec des noms de canaux peu intuitifs.

Les composants primitifs. Nous nous sommes ensuite intéressés aux composants de manipulation de messages. Ces composants sont classiquement des composants primitifs (comme dans notre première approche) et leur définition implique donc la construction d'une relation `match` et d'une fonction `eval`. De fait, l'utilisation de ces deux relations, bien que contraignante, offre une très grande expressivité et l'on peut dans ce nouveau calcul décrire tout les composants primitifs de notre première approche : les composants de manipulation de messages aussi bien que les multiplexeurs et les routeurs basés sur la structure des messages. De plus, nous pouvons aussi définir les multiplexeurs et routeurs basés sur les annotations de routage tels que nous les avons présentés en début de chapitre. La première étape de la définition de ces composants est de définir leurs noms, pour pouvoir ensuite construire leur sémantique via les relations `match` et `eval`. Nous notons par la suite

- $\mathrm{Gen}(\mathit{Str})[\emptyset/\, s]$ le composant qui génère un message sur s et qui comporte un unique champ Str ;

mobilité des messages, c'est à dire, l'isolation

- Add$(a)[e_1 e_2 \,/\, s]$ le composant qui, à la réception de M_1 sur e_1 et M_2 sur e_2, renvoie sur s le message M_1 étendu par le champ a contenant M_2 ;
- Acs$(a)[e \,/\, s]$ le composant qui, à la réception de M sur e, renvoie sur s le contenu du champ a de M ;
- Sub$(a)[e \,/\, s]$ le composant qui, à la réception de M sur e, renvoie sur s le même message M sans son champ a ;
- IfPre$(Sub)[e \,/\, s_1 s_2]$ le routeur de notre approche précédente testant l'existence du champ 'Sub' sur le message reçu sur e, et l'envoyant soit sur s_1 ou s_2 ;
- Trans$[e \rightarrow s]$ le composant transmettant les messages du canal e vers le canal s (le multiplexeur de notre première approche peut s'encoder avec deux de ces composants) ;
- Route$[e \,/\, s_1 s_2][r]$ le routeur nommé r avec e comme interface entrante et s_1 et s_2 comme interfaces sortantes ;
- Mult$[e_1 e_2 \,/\, s][r]$ le multiplexeur nommé r écoutant sur e_1 et e_2, et renvoyant les messages reçus sur s.

À partir de ces différents noms de composants, nous pouvons construire Figure 3.4 la relation `match` définissant les paramètres valides de chaque composant. Cette relation est assez intuitive, et il est assez facile de voir qu'elle

$$
\begin{aligned}
&\{(\mathrm{Gen}(\mathtt{Str})[\emptyset/\, s], \quad)\} \\
\cup\, &\{(\mathrm{Add}(a)[e_1 e_2 \,/\, s], \overline{e_1}\langle\{a_1 = v_1 \ldots a_n = v_n\}^{\delta}\rangle \mid \overline{e_2}\langle v^{\delta'}\rangle) \mid \forall 1 \leq i \leq n,\, a_i \neq a\} \\
\cup\, &\{(\mathrm{Acs}(a)[e \,/\, s], \overline{e}\langle\{a = v; a_1 = v_1; \ldots; a_n = v_n\}^{\delta}\rangle) \mid 0 \leq n\} \\
\cup\, &\{(\mathrm{Sub}(a)[e \,/\, s], \overline{e}\langle\{a = v; a_1 = v_1; \ldots; a_n = v_n\}^{\delta}\rangle) \mid 0 \leq n\} \\
\cup\, &\{(\mathrm{IfPre}(\mathtt{Sub})[e \,/\, s_1 s_2], \overline{e}\langle\{a_1 = v_1 \ldots a_n = v_n\}^{\delta}\rangle)\} \\
\cup\, &\{(\mathrm{Trans}[e \to s], \overline{e}\langle M\rangle)\} \\
\cup\, &\{(\mathrm{Route}[e \,/\, s_1 s_2][r], \overline{e}\langle v^{\delta}\rangle) \mid r \in \delta\} \\
\cup\, &\{(\mathrm{Mult}[e_1 e_2 \,/\, s][r], \overline{e_1}\langle v^{\delta}\rangle) \mid r \notin \delta\} \cup \{(\mathrm{Mult}[e_1 e_2 \,/\, s][r], \overline{e_2}\langle v^{\delta}\rangle) \mid r \notin \delta\}
\end{aligned}
$$

FIGURE 3.4 – la relation `match`

correspond aux spécifications que nous nous sommes données concernant les composants. Notons que dans la déclaration des entrées valides du routeur et du multiplexeur, nous utilisons les notations $r \in \delta$ (et $r \notin \delta$). Informellement, cette notation signifie que le nom de routage r doit être présent dans l'annotation δ, c'est à dire que cette dernière doit contenir soit $\uparrow r$ soit $\downarrow r$. Nous définissons la fonction `eval` Figure 3.5. Les résultats des composants primitifs que nous avons déjà étudiés ($\mathrm{Gen}(Str, s)$, $\mathrm{Add}(a)[e_1 e_2 \,/\, s]$, $\mathrm{Acs}(a)[e \,/\, s]$. $\mathrm{Sub}(a)[e \,/\, s]$, et $\mathrm{IfPre}(Sub)[e \,/\, s_1 s_2]$) n'ont pas changé depuis notre première approche. La sémantique du composant $\mathrm{Trans}[e \to s]$ est elle-même très élémentaire. Concernant le routeur, les choses restent simple : les messages annotés par $\uparrow r$ sont envoyés sur la première sortie (s_1), et ceux annotés par $\downarrow r$ sont envoyés sur s_2. La définition de cette sémantique simple est toutefois allongée par une difficulté technique : il n'y a pas de règles d'equivalence permettant de permuter les annotations de routage entre elles. Il serait naturel et assez souhaitable d'ajouter une telle règle, mais elle est en fait incompatible avec le système de type que nous avons défini sur ce calcul : ceci sera discuté dans la partie suivante concernant le typage. Sans cette règle, nous devons parcourir l'annotation de routage jusqu'à trouver la partie concernant le nom r (ceci est fait en écrivant l'annotation sous

$$\texttt{eval}(\mathrm{Gen}(\texttt{Str})[\emptyset/\,s],\quad) \triangleq \overline{s}\langle\{\texttt{Str} = \text{“}\texttt{toto}\text{”}\}\rangle$$

$$\texttt{eval}(\mathrm{Add}(a)[e_1e_2\,/\,s], \overline{e_1}\langle\{a_1 = v_1\ldots\}^{\delta}\rangle \mid \overline{e_2}\langle v^{\delta'}\rangle) \triangleq \overline{s}\langle\{a = v; a_1 = v_1;\ldots\}^{\delta}\rangle$$

$$\texttt{eval}(\mathrm{Acs}(a)[e\,/\,s], \overline{e}\langle\{a = v; a_1 = v_1\ldots\}^{\delta}\rangle) \triangleq \overline{s}\langle v^{\delta}\rangle$$

$$\texttt{eval}(\mathrm{Sub}(a)[e\,/\,s], \overline{e}\langle\{a = v; a_1 = v_1\ldots\}^{\delta}\rangle) \triangleq \overline{s}\langle\{a_1 = v_1\ldots\}^{\delta}\rangle$$

$$\texttt{eval}(\mathrm{IfPre}(\texttt{Sub})[e\,/\,s_1s_2], \overline{e}\langle\texttt{Sub} = v; a_1 = v_1\ldots\}^{\delta}\rangle) \triangleq \overline{s_1}\langle\{\texttt{Sub} = v; a_1 = v_1\ldots\}^{\delta}\rangle$$

$$\texttt{eval}(\mathrm{IfPre}(\texttt{Sub})[e\,/\,s_1s_2], \overline{e}\langle\{a_1 = v_1\ldots a_n = v_n\}^{\delta}\rangle) \triangleq \overline{s_2}\langle\{a_1 = v_1\ldots a_n = v_n\}^{\delta}\rangle$$

$$\text{si } \forall 1 \leq i \leq n,\ a_i \neq \texttt{Sub}$$

$$\texttt{eval}(\mathrm{Trans}[e \to s], \overline{e}\langle M\rangle) \triangleq \overline{s}\langle M\rangle$$

$$\texttt{eval}(\mathrm{Route}[e\,/\,s_1s_2][r], \overline{e}\langle M^{\updownarrow r_1;\ldots;\updownarrow r_i;\uparrow r;\delta}\rangle) \triangleq \overline{s_1}\langle M^{\updownarrow r_1;\ldots;\updownarrow r_i;\delta}\rangle$$

$$\texttt{eval}(\mathrm{Route}[e\,/\,s_1s_2][r], \overline{e}\langle M^{\updownarrow r_1;\ldots;\updownarrow r_i;\downarrow r;\delta}\rangle) \triangleq \overline{s_2}\langle M^{\updownarrow r_1;\ldots;\updownarrow r_i;\delta}\rangle$$

$$\texttt{eval}(\mathrm{Mult}[e_1e_2\,/\,s][r], \overline{e_1}\langle M^{\delta}\rangle) \triangleq \overline{s}\langle M^{\uparrow r;\delta}\rangle$$

$$\texttt{eval}(\mathrm{Mult}[e_1e_2\,/\,s][r], \overline{e_2}\langle M^{\delta}\rangle) \triangleq \overline{s}\langle M^{\downarrow r;\delta}\rangle$$

FIGURE 3.5 – La fonction `eval`

la forme $\updownarrow r_1;\ldots;\updownarrow r_i;\updownarrow r;\delta$ avec $\updownarrow r_i$ signifiant soit $\uparrow r_i$ soit $\downarrow r_i$), et donner en résultat le message annoté par une liste comportant exactement les mêmes éléments sans $\updownarrow r$ et ordonnés de la même façon. La description du multiplexeur est bien plus simple, et correspond à ce que nous avons décrit en début de ce chapitre : les messages reçus sont envoyés sur le canal de sortie s avec une annotation renseignant sur leur origine (soit la première entrée soit la seconde du composant). Afin de pouvoir construire les exemples qui vont suivre dans le reste du chapitre, nous complétons informellement notre ensemble de composants primitifs par des générateurs de messages stockant un entier ($\mathrm{Gen}(\texttt{Int})[\emptyset/\,s]$), par des duplicateurs ($\mathrm{Dupl}[e\,/\,s_1, s_2][\cdot]$) et autres composants assez classiques. De

plus, nous donnons Figure 3.6 une représentation graphique des deux nouveaux composants (le nouveau routeur et le nouveau multiplexeur) afin d'avoir une présentation simple et graphique de nos différents exemples.

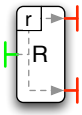
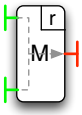
Composants	Abbréviations
Route$[e \,/\, s_1 s_2][r]$	
Mult$[e_1 e_2 \,/\, s][r]$	

FIGURE 3.6 – La charte graphique pour le routeur et le multiplexeur

Les annotations de routage. Nous avons mentionné le fait que nos annotations de routage ne peuvent comporter plusieurs occurrences d'un même nom de routage. De fait, les annotations de routage ont pour but de connaître l'origine d'un message quand elle existe, mais surtout, de pouvoir décider de sa destination sans se tromper (ce qui a une certaine influence sur le typage des composants et l'existence d'un algorithme d'inférence). Et cette propriété est incompatible avec la possibilité d'avoir plusieurs occurrence d'un même nom de routage dans une annotation. Considérons en effet l'exemple Figure 3.7 où nous autorisons exceptionnellement que l'annotation des messages générés ait deux mentions du nom r. Dans cet assemblage, les messages produits par le com-

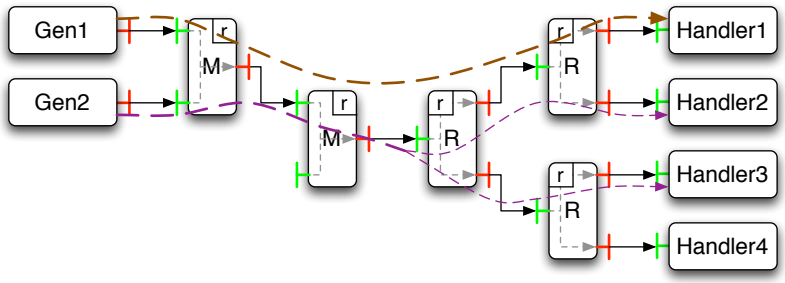


FIGURE 3.7

posant `Gen1` sont tous traités par le composant `Handler1`, mais un problème survient pour les messages générés par `Gen2`, qui sont annotés par $\uparrow r; \downarrow r$ sur

le canal central (entre le second multiplexeur et le premier routeur). Ces messages peuvent en effet être traités par le composant `Handler2`, aussi bien que par `Handler3` : il est impossible de connaître la destination de ces messages à partir de leurs annotations. C'est pourquoi nous avons décidé de restreindre les annotations de routage à ne contenir au plus qu'une unique occurrence par nom de routage, ce qui rend ce genre d'assemblage invalide [5].

Comparaison avec notre première approche

Les composants primitifs. Nous avons une dernière remarque concernant la sémantique de nos composants primitifs : elle n'est pas tout à fait identique à celle des composants de notre première approche. En effet, alors que nos anciens composants consommaient leurs messages les uns après les autres avant de générer un résultat (ce qui peut entraîner des inter-blocages), la forme de nos nouvelles relations `match` et `eval` imposent que la consommation des messages en paramètre se fasse de manière atomique, de la même façon que les jointures du Join-calcul [39]. Cette différence peut être essentielle lorsque l'on étudie les propriétés d'un calcul, mais elle n'est pas problématique dans notre cas : notre étude porte sur plusieurs éléments spécifiques du calcul, comme la façon de router les messages, mais ne s'intéresse pas encore à la différence entre les réceptions séquentielles et les jointures.

Les chaînes de composants. Comme nous pouvons re-créer ici tout les composants de notre première approche, il est possible de construire à nouveau tous les assemblages que nous avons déjà présentés dans le chapitre précédent. Néanmoins, l'introduction des annotations de routage dans notre calcul lui offre une plus grande expressivité, que nous pouvons comparer aux anciennes procédures de multiplexage et de routage. Le premier exemple de notre étude est présenté Figure 3.8 et consiste en un simple assemblage avec deux générateurs. un multiplexeur, un routeur et deux handlers. Typiquement, ce genre d'assemblage correspond à deux applications distribuées – la première (resp. la seconde) étant constituée de `GenStr` et de `Handler1` (resp. `GenInt` et `Handler2`) – mises en parallèle et partageant une connection internet. Dans cette figure, nous avons noté par des flèches marrons en pointillé le chemin que suivent les messages parcourant l'assemblage. Nous pouvons ainsi remarquer que les messages générés par `GenStr` sont traîtés par `Handler1`, et les seconds types de messages sont eux traîtés par le second handler : les deux applications sont donc bien identifiées par le routeur qui envoie les messages de chaque application au handler correspondant. Lorsque l'on remplace le multiplexeur et le routeur de cet assemblage par leur nouvelle version (ce qui est fait Figure 3.9), nous obtenons

5. Notons qu'il est aussi possible de remédier à ce problème en changeant la sémantique des routeurs : un routeur nommé r ne pourrait traiter que les messages ayant une annotation commençant par $\uparrow r$ ou $\downarrow r$. Cette solution a le désavantage que l'ordre dans lequel les routeurs sont placés dans un assemblage est alors totalement défini par les annotations des messages. Nous avons jugé cette restriction trop forte, et avons donc choisi de contraindre les annotations.

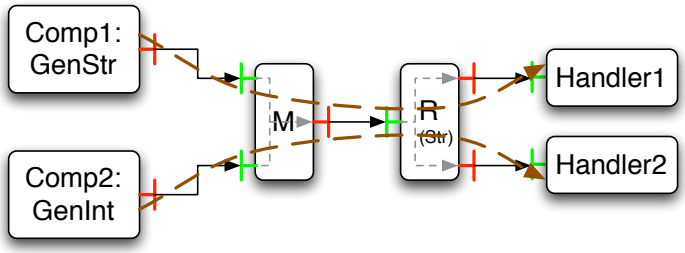


FIGURE 3.8

exactement le même comportement. Mais alors que dans la version précédente, la distinction entre les deux programmes distribués était basée sur la différence de structure des messages échangés, ici c'est le multiplexeur qui identifie les deux programmes avec l'ajout d'une annotation de routage sur les messages. Ce

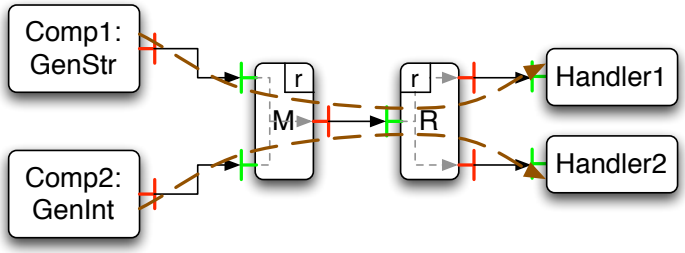
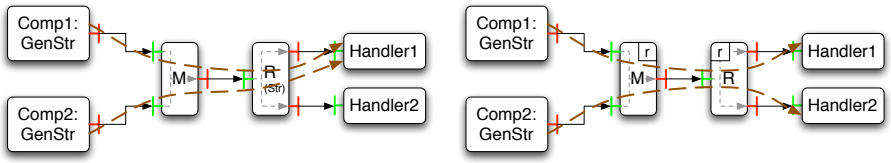


FIGURE 3.9

distinguo de sémantique est assez flagrant lorsque nous modifions légèrement les deux applications Figure 3.10 qui maintenant échangent des messages de structures similaires. Notre ancien routeur est alors incapable de distinguer les messages provenant d'une application par rapport aux autres, et les transmet tous au premier ou au second handler (dans notre exemple Figure 3.10a, les messages sont envoyés au premier handler). Par contre, notre nouvelle approche permet toujours de distinguer les messages en fonction de leur origine, grâce au multiplexeur. C'est pourquoi l'assemblage Figure 3.10b a un comportement correct en n'envoyant pas par erreur un message de la seconde application au handler de la première, par exemple. Notons que le routage structurel tel qu'il est fait dans notre première approche est assez intéressant : il permet de s'assurer qu'un message a la bonne structure pour les manipulations qu'il devra subir, tout en permettant le traitement des cas d'erreur. Ses capacités d'expression sont suffisamment grandes pour permettre la définition de tous les exemples de



(a) (b)

FIGURE 3.10

notre première approche, mais au vu de ce premier exemple comparatif, il nous est apparu assez flagrant qu'un routage sémantique (utilisant des annotations sur des messages de structures possiblement similaires) est bien plus adapté à la création de programmes distribués.

Les boucles. Notre second comparatif étudie le composant `TheLoop` dont nous avons déjà beaucoup parlé, et que nous rappelons Figure 3.11. Cet assemblage a la particularité d'effectuer à un message entrant un nombre de tour de boucle arbitraire afin d'en extraire le message structuré se trouvant sous une pile de champ `Sub`. Par contre, bien que ce composant est assez facilement

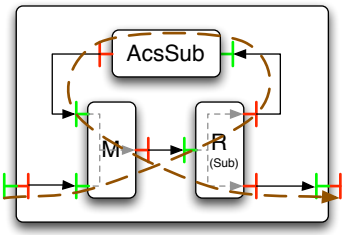


FIGURE 3.11

définissable avec nos anciens routeurs et multiplexeurs, il est impossible de créer un assemblage similaire dans notre nouvelle approche. Considérons par exemple l'assemblage Figure 3.12a où nous avons simplement repris l'assemblage `TheLoop` en remplaçant le multiplexeur et le routeur par leur nouvelle version. Nous pouvons bien voir sur la figure que le flux de donnée est alors totalement différent de celui que nous avions précédemment. En effet, l'entrée de l'assemblage étant celle du bas du multiplexeur, tous les messages entrants sont automatiquement annotés par $\downarrow r$. Par conséquent, le routeur les transmet sur sa sortie du bas, qui est le canal de sortie de l'assemblage. De plus, les messages qui pourraient se trouver sur la partie constituant la boucle de l'assemblage ne peuvent en

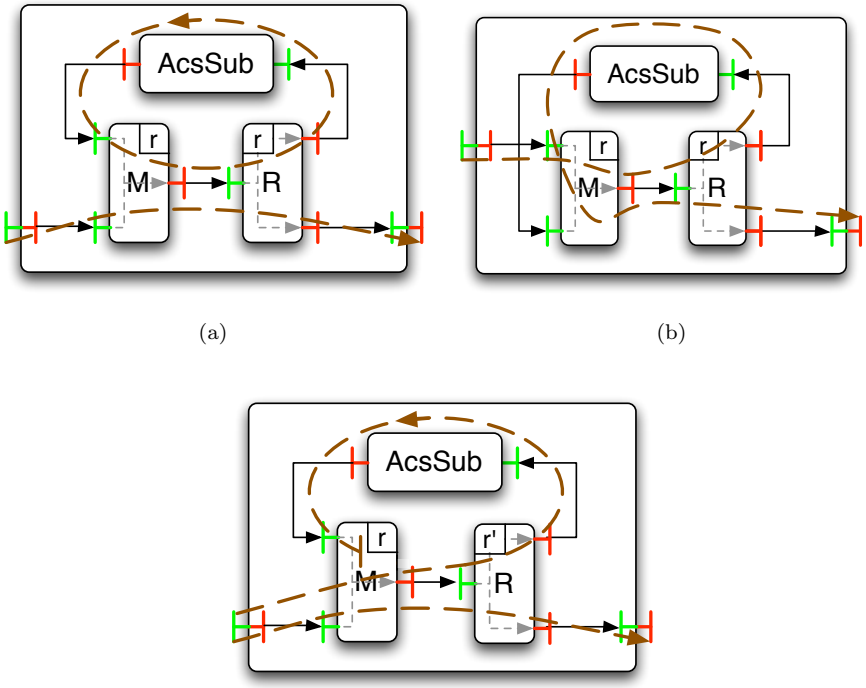
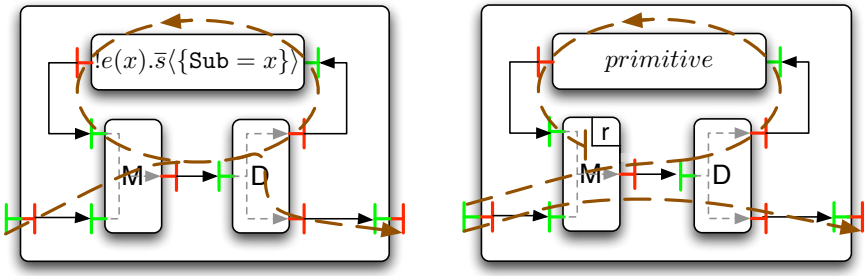


FIGURE 3.12

sortir, toujours à cause de la façon dont le routage fonctionne. Par conséquence cet assemblage est erroné, car il n'arrête jamais d'accéder au champ `Sub` des messages sur la boucles, même quand ceux-ci ne comportent plus un tel champ. Une seconde possibilité pour recréer l'assemblage `TheLoop` dans notre nouvelle approche est proposée Figure 3.12b. Cet assemblage est en tout point similaire à notre première adaptation, sauf en ce qui concerne son entrée : maintenant celle-ci est liée à l'entrée haute du multiplexeur. Par conséquent, le routeur envoie les messages reçus dans la boucle. Évidemment, il ne teste pas ce faisant si le champ `Sub` est bien présent dans le message. Les messages sur la boucle sont ensuite envoyés sur l'interface basse du multiplexeur, et donc envoyés sur la sortie de l'assemblage. Il est intéressant de voir que juste en permutant deux connections, nous avons maintenant un assemblage correct, mais dont le comportement est assez élémentaire (il correspond simplement au composant primitif Acs(`Sub`)$[e\,/\,s]$) Nous proposons finalement une troisième adaptation possible, décrite Figure 3.12c, où le multiplexeur et le routeur n'ont pas le même nom. Nous avons alors deux types de messages entrant : ceux qui sont annotés par $\downarrow r'$ et ceux annotés par $\uparrow r'$. Ces deux types de messages se voient rajouter l'annotation $\downarrow r$ par le multiplexeur, mais alors que les premiers sortent directement

de l'assemblage, les seconds entrent dans la boucle, sont manipulés par le composant Acs(Sub)$[e\,/\,s]$ avant d'être renvoyés sur l'entrée haute du multiplexeur. Et comme ces messages sont déjà annotés par $\downarrow r$, ils ne peuvent pas être pris en compte par le multiplexeur, ce que nous considérons comme étant une erreur. Une erreur similaire survient aussi dans notre adaptation du composant proposé Figure 3.13 : l'annotation $\downarrow r$ n'est jamais ôtée des messages lors de leur passage dans la boucle, causant ainsi une erreur lorsqu'ils sont envoyés de nouveau au multiplexeur.



(a) Un assemblage de notre première approche

(b) Son adaptation

FIGURE 3.13

La manipulation des annotations

Nous terminons notre présentation d'exemples par une petite étude d'expressivité que l'on peut avoir avec des composants primitifs et nos annotations de routage. En effet, pour le moment, nous avons considéré que seuls les multiplexeurs et les routeurs pouvaient modifier les annotations de routage des messages, mais comme l'ensemble des composants primitifs est totalement libre, nous pouvons définir d'autres composants manipulant ces annotations de manière différente, et les étudier. Nous pouvons par exemple définir des composants de la forme Add$[e\,/\,s][\uparrow r]$ ou Rem$[e\,/\,s][\downarrow r]$ qui étendent ou réduisent les annotations de routages des messages traités. Il est intéressant de remarquer qu'avec de tels composants, le multiplexeur Mult$[e_1 e_2\,/\,s][r]$ peut facilement s'encoder avec l'assemblage suivant :

$$c[e_1 e_2\,/\,s][\mathrm{Add}[e_1\,/\,s][\uparrow r] \mid \mathrm{Add}[e_2\,/\,s][\downarrow r]]$$

De manière similaire, nous pouvons décomposer, et même généraliser le routeur en un assemblage de composants plus élémentaires. Pour ce faire, nous introduisons un nouveau composant primitif dont la sémantique est assez semblable au filtrage de motif : nous notons Filtre$[e\,/\,(\uparrow r \mapsto s_1), (\uparrow r' \mapsto s_2), (\downarrow r \mapsto s_3)\ldots]$ le composant qui filtre les messages reçus sur le canal e en envoyant ceux annotés

par $\uparrow r$ sur le canal s_1, ceux annotés par $\uparrow r'$ sur le canal s_2, etc[6]. Avec un tel composant, le routeur Route$[e \,/\, s_1 s_2][r]$ est équivalent à l'assemblage suivant :

$$c[e \,/\, s_1 s_2][\text{Filtre}[e \,/\, (\uparrow r \mapsto i_1), (\downarrow r \mapsto i_2)] \mid \text{Rem}[i_1 \,/\, s_1][\uparrow r] \mid \text{Rem}[i_2 \,/\, s_2][\downarrow r]]$$

Remarquons que nous pouvons encore généraliser notre composant de filtrage en traitant aussi les messages dont l'annotation ne comporte pas un certain nom de routage (par exemple, avec une syntaxe du style Filtre$[e \,/\, (\neg r \mapsto s_1), \ldots])$. Remarquons de plus que ce composant de filtrage permet de construire des boucles un peu plus subtiles que celles que nous avons vues jusqu'à présent. En effet, nos exemples précédents montraient des boucles assez limitées, où les messages, soit ne rentraient jamais, soit seulement une fois, soit étaient bloqués à l'intérieur et ne pouvaient jamais en sortir. Bien sûr, ceci peut être résolu par l'ajout d'une conditionnelle dans notre ensemble de composants primitifs, mais il n'en est pas moins que le routage paraît limité. Avec notre procédure de filtrage, nous pouvons avoir des boucles traitant les messages entrant un nombre n donné de fois. Par exemple, l'assemblage proposé Figure 3.14 applique trois fois le composant D aux messages entrants en utilisant quatre noms de routage (r pour décider quand sortir de la boucle, et les r_i pour le i-ème tour). Nous n'avons donc pas la

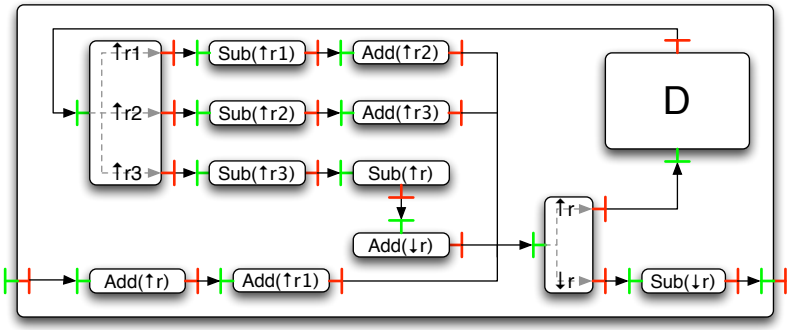


FIGURE 3.14

flexibilité de notre première approche, où ne nombre de tours de boucle effectué par un message pouvait dépendre du message lui-même, mais comme le but de ce travail est de produire une procédure de routage typable avec de l'inférence de type, ce genre d'expressivité est déjà intéressant. De fait, nous avons été assez surpris lors de cette étude des possibilités qu'offrent nos annotations de routage. En effet, nous les avions introduites pour pouvoir gérer un routage assez simple distinguant deux types d'entrée, et il apparait clairement ici que leur utilisation peut être grandement généralisée. Informellement, nous pouvons considérer que ces annotations rajoutent une dimension sémantique aux valeurs, et deux valeurs identiques de sémantiques différentes (c'est à dire, ne portant pas la même

6. Bien sûr, une présentation formelle de cette procédure de filtrage doit définir ce qui se passe lorsqu'un message peut être traité par des branches différentes

annotation) peuvent subir un traitement différent (en utilisant le composant de filtrage). Dans notre approche actuelle, cette sémantique consiste simplement en un encodage de l'origine des messages : deux messages provenant de branches différentes d'un multiplexeur peuvent être différenciés. Nous pensons qu'il serait très intéressant d'étudier un peu plus cette nouvelle notion d'annotation *sémantique*, dans un cadre de routage distribué mais aussi dans d'autres environnements, comme ML qui offre déjà avec les variants des fonctionnalités similaires.

3.2 Le système de type

Comme nous l'avons vu, les résultats de notre premier système de type étaient assez mitigés. Bien que les propriétés essentielles de correction et de stabilité étaient bien vérifiées par cette approche, cette dernière avait quelques défauts suffisamment importants pour nous pousser à essayer de les résoudre dans ce nouveau travail. C'est pourquoi nous avons testé dans notre calcul une nouvelle procédure de routage, et notre objectif principal maintenant est de trouver un moyen élégant de prendre en compte ces annotations de routage dans nos types. Bien sûr, il nous faut garder la possibilité d'avoir plusieurs types de message sur un même canal, sans quoi le routage serait inutile. Une première solution serait de suivre le même système que dans notre approche précédente, avec des types de la forme $e : (T) \cup e : (T') \cup \ldots$: nous pouvions déclarer plusieurs fois le même canal e, chaque déclaration portant un type de message différent. Cette syntaxe offre énormément de souplesse mais, à cause de celle-ci, il est impossible d'imposer des contraintes limitant le nombre de types de messages passant par chaque canal de communication. Et sans ce genre de contraintes, nous obtenons un système de type similaire à celui de notre approche précédente. Nous avons donc opté pour une syntaxe où la déclaration des canaux dans les types est contrôlée avec, en général, une seule déclaration par canal et plusieurs types de message par déclaration. Afin de mettre en place ce dernier point, nous nous sommes intéressé au typage des *variants* en ML [92]. En effet, les variants sont une structure assez similaire à nos messages routés : ils consistent en des messages uniques, annotés par des étiquettes – similaires à nos annotations – permettant ainsi à ces structures d'être des arguments valides pour une procédure de filtrage de motif – correspondant à notre routeur. De plus, ils sont typés par des types rangés qui associent à un étiquette un unique type de message – ce qui convient très bien à notre approche du routage – et qui offrent l'avantage de permettre l'inférence de type [93]. Néanmoins, il est impossible d'adapter ces travaux à nos messages routés, car ils sont trop limités en ce qui concerne la manipulation des étiquettes. En effet, même en utilisant la généralisation des types rangés [89], nous ne pouvons typer les composants tels que le multiplexeur et le routeur, car ils modifient l'annotation d'un message sans la connaître précisément : une telle opération correspond typiquement à

un filtrage de motif infini [7]. La généralisation de ces travaux (pour y permettre le typage des manipulations des annotations) s'étant révélée assez difficile, nous avons finalement opté pour la création d'une nouvelle forme de types : les types *routés* typant les messages annotés. Ces types prennent la forme d'arbres binaires, où les noeuds internes sont des noms de routage, et les feuilles sont des types de message annotés par des *variables de flux*. Le but de ces types routés est de construire une association finie entre une annotation de routage et un type de message (tout en identifiant chaque nom de routage, permettant ainsi leur rajout et suppression par les multiplexeurs et les routeurs). La façon dont ce mapping est construit est relativement simple : considérons en effet l'arbre donné Figure 3.15. Cet arbre comporte trois branches qui peuvent être notées

$$r_1(\xi_1[\text{int}], r_2(\xi_2[\text{string}], \xi_2[\eta])) \quad \triangleq$$

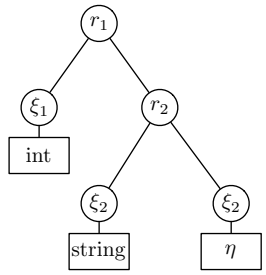


FIGURE 3.15 – Un type routé

(i) $\downarrow r_1 \rightarrow \xi_1[\text{int}]$, (ii) $\uparrow r_1; \downarrow r_2 \rightarrow \xi_2[\text{string}]$ et (iii) $\uparrow r_1; \uparrow r_2 \rightarrow \xi_2[\eta]$: $\uparrow$ correspond à aller sur le sous-arbre gauche, et $\downarrow$ sur le sous-arbre droit. Ainsi, chaque branche correspond à une annotation de routage sur un message dont le type est la feuille de la branche : la structure d'arbre binaire permet de bien distinguer chaque type d'entrée d'un multiplexeur, et donc permet un typage simple des procédures de routage et de multiplexage. Cette structure comporte néanmoins un défaut que nous discuterons lors de la présentation des règles de typage. Les variables de flux, quant à elles, servent pour l'unification des types routés. De fait, cette unification est assez unique en son genre, car elle fait appel à ce que nous appelons de la *duplication*. Considérons par exemple le composant $\text{Acs}(a)[e\,/\,s]$ (qui renvoie sur s le contenu du champ a des messages reçus sur e). L'entrée de ce composant est donc typiquement typée par $T \triangleq \xi[\{a : \text{Pre}(\alpha); \rho\}]$ avec ξ signifiant que les messages reçus peuvent avoir n'importe quelle annotation. Maintenant, si ce composant reçoit des messages portant des annotations différentes (et donc typés avec un arbre tel que nous l'avons présenté), nous devons être capable d'unifier cet arbre avec T, ce qui demande l'utilisation de variables (d'où nos variables de flux) et de substitutions assez uniques que nous

7. L'opération de routage en elle-même est clairement typable et correspond à un filtrage de motif à deux branches, mais la suppression de $\downarrow r$ et $\uparrow r$ de l'annotation n'est, elle, pas typable

présenterons au cours de cette partie.

Finalement, nous avons profité de cette nouvelle approche pour y inclure du sous-typage structurel [20, 103]. Plusieurs travaux se sont déjà intéressés au sous-typage dans les systèmes distribués [115], et tous associent deux types à chaque canal : un type d'entrée correspondant aux messages que le canal envoie vers un processus, et unß type de sortie correspondant aux messages reçus d'un processus. Ceci permet de gérer la co- et la contra- variance. Nous avons simplement repris ces travaux ici, en modifiant toutefois la syntaxe : la séparation entre les types d'entrée et les types de sortie est faite avec le symbole $\rightarrow$, ce qui permet d'être plus proche syntaxiquement des types usuels à la ML, et de donner une séparation claire entre les types des paramètres et les types des résultats d'un assemblage.

3.2.1 La syntaxe

L'algèbre de nos types est construite à partir d'un ensemble de variables $\mathcal{V}$ et de constructeurs de types $\mathcal{C}$. Nous supposons que $\mathcal{V}$ est l'union disjointe des ensembles $\mathcal{V}^m$, $\mathcal{V}^r(l)$ et $\mathcal{V}^t(R)$ où l est un ensemble d'étiquette et R un ensemble de nom de routage. Nous notons η les variables dans $\mathcal{V}^m$, ρ^l les variables dans $\mathcal{V}^r(l)$ et ξ^R les variables de l'ensemble $\mathcal{V}^r(R)$. Finalement, chaque constructeur de type t est défini avec une arité $a(t)$. La syntaxe de nos types est présentée Figure 3.16. Les types élémentaires servent, comme dans la version précédente,

$$
\begin{array}{rcll}
E & ::= & \eta \quad | \quad \{W^{\emptyset}\} \quad | \quad s(E_1, \ldots, E_n) & \text{Type élémentaire} \\
W^l & ::= & \rho^l \quad | \quad \mathrm{Abs}^l \quad | \quad a : K; W^{l \uplus \{a\}} & \text{Type rangé} \\
K & ::= & \mathrm{Pre}(E) \quad | \quad \mathrm{Abs} & \text{Presence ou non d'un champ} \\
T^R & ::= & & \text{Type routé} \\
 & & \xi^{R'}[E] \quad \text{with } R \subset R' & \text{Feuille} \\
 & | & r(T_1^{R \uplus \{r\}}, T_2^{R \uplus \{r\}}) & \text{Noeud interne} \\
S & ::= & \emptyset \quad | \quad e : (T^{\emptyset}) \quad | \quad S \cup S' & \text{Type ensemble} \\
F & ::= & S \rightarrow S \quad | \quad \forall \gamma . F & \text{Type de processus}
\end{array}
$$

FIGURE 3.16 – La syntaxe des types

à typer les messages, et sont construits à partir de variables η, de types rangés $\{W^{\emptyset}\}$ et de constructeurs de types qui permettent de typer des données de base, comme des entiers ou chaînes de caractères. Un type rangé consiste en une liste de déclaration de champs, précisant s'il contient une donnée ou non, et terminée soit par Abs, signifiant qu'il n'y a pas d'autre champ définit dans

le type, ou par ρ, signifiant que le reste du type est encore à spécifier. Ces types sont toujours annotés par un ensemble de champ 'l' en exposant qui empêche qu'un type ait deux fois le même champ. Les types routés servent à typer les messages annotés. Une *feuille* $\xi[E]$ correspond à un message sans annotation de routage, ou dont l'annotation n'est pas spécifiée comme dans les types de composants primitifs. Un *noeud interne* $r(T_1, T_2)$ consiste en un nom de routage 'r' et deux sous arbres : (i) celui de gauche correspondant à un ensemble de messages annotés par $\downarrow r$, et (ii) celui de droite correspondant à un ensemble de messages annotés par $\uparrow r$. Les types routés possèdent une *kind* R similaire à l'exposant utilisé par les types rangés. Cette kind est un ensemble de nom de routage, ce qui permet de s'assurer qu'un tel nom n'est utilisé qu'une seule fois dans un type (suivant ainsi la restriction imposée sur les annotations). Comme précédemment, nous ne noterons cette kind, pour les types routés ou pour les types rangés que lorsqu'elle sera importante pour nos explications, ou non communiquée par le contexte. Les types ensembles servent à typer les entrées et sorties des composants et processus. Ils sont très similaires aux types de processus de notre première version : ils déclarent les canaux de communication utilisés ainsi que les types des données transitant par ceux-ci, mais contrairement à notre première approche, il est impossible de déclarer un canal sans type de message y transitant. Cette limitation permet de contrôler la validité d'un assemblage du point de vue des types, et ce, même lorsqu'il n'est pas utilisé (c'est à dire qu'aucun message n'y est manipulé). Finalement, les types de processus déclarent les entrées et les sorties de ce dernier : nous disons par la suite qu'un type ensemble est *paramètre* (resp. *résultat*) lorsqu'il est utilisé à gauche (resp, à droite) du symbole $\rightarrow$. Ces types ont aussi une construction pour avoir des schémas de types, afin d'avoir une certaine généralité dans les types des composants primitifs [8]. Notons que ces schémas de types font intervenir les variables γ, qui correspondent à n'importe quel élément de $\mathcal{V}$. Notons que notre syntaxe permet de déclarer plusieurs fois un canal de communication dans un type S : nous définissons dans un paragraphe prochain une notion de types *valide* correspondant à l'approche que nous avons décrit dans l'introduction de cette partie. Cette notion contraint bien les entrées des types de processus à n'avoir qu'au plus une déclaration par canal, mais impose aussi aux types une autre restriction nécessitant la définition de la relation de sous-typage. Afin de terminer la présentation de la syntaxe, nous introduisons dans le reste de cette partie l'équivalence structurelle entre les types, puis la relation de sous-typage, et enfin la définition des types valides.

L'équivalence structurelle. Cette relation est la plus petite relation d'équivalence qui est une congruence satisfaisant les règles présentées Figure 3.17. Informellement, cette relation identifie les types de sémantiques similaires : les types rangés mappant les mêmes champs aux mêmes types, les types routés

8. Notons en effet que nous n'avons pas de construction telle que le `Let` de ML dans notre calcul, et donc les schémas de types perdent beaucoup de leur intérêt. Leur unique utilité est donc dans le typage des composants primitifs, que l'on peut utiliser plusieurs fois dans un même assemblage

déclarant les mêmes annotations portées par les mêmes types de messages et les types ensemble définissant les mêmes canaux avec les mêmes types de données. De plus, nous proposons dans la définition suivante quelques opérateurs sur les

$$\frac{K_1 \equiv K_1' \qquad K_2 \equiv K_2' \qquad W^l \equiv W^{l'}}{a : K_1; b : K_2; W^l \equiv b : K_2'; a : K_1'; W^{l'}} \qquad \mathrm{Abs}^l \equiv a : \mathrm{Abs}; \mathrm{Abs}^{l \uplus \{a\}}$$

$$r(r_1(T_1, T_2), r_1(T_3, T_4)) \equiv r_1(r(T_1, T_3), r(T_2, T_4)) \qquad S_1 \cup S_2 \equiv S_2 \cup S_1$$

$$(S_1 \cup S_2) \cup S_3 \equiv S_1 \cup (S_2 \cup S_3)$$

FIGURE 3.17 – L'équivalence structurelle des types

types ensembles, qui seront largement utilisés dans le reste de ce chapitre.

Définition 5. *Supposons donné un type ensemble* $S \triangleq \bigcup_{i \in I} e_i : (T_i)$ *: nous notons* $dc(S)$ *l'ensemble* $\{e_i \mid i \in I\}$ *et* $S(e)$ *l'ensemble* $\{T_i \mid i \in I \wedge e_i = e\}$*. Supposons donné un autre type ensemble* $S' \triangleq \bigcup_{i \in I'} e_i' : (T_i')$ *: nous définissons la relation* $\subset$ *sur ces types par* $S \subset S'$ *si et seulement si pour tout canal* $e \in dc(S)$ *et tout type* $T \in S(e)$*, nous avons* $T \in S'(e)$*. Finalement, supposons donné un ensemble* I *de noms de canaux : nous notons* $S \cap I$ *le plus petit type ensemble* S'' *(w.r.t* $\subset$*) tel que* $S''(e) = S(e)$ *pour tout* $e \in I$*.*

La relation de sous-typage. Une des nouveautés de ce système de type par rapport au premier est l'ajout d'une relation de sous-typage. La motivation de ce rajout est relativement simple : les données qu'échangent les composants dans notre calcul peuvent être des messages structurés comme des constantes correspondant par exemple à des objets (Java, C++, etc) qui demandent l'utilisation d'une relation de sous-typage pour être traité correctement par un système de type. Nous avons donc intégré une relation de sous-typage dans notre approche pour voir si cela n'entrait pas en conflit avec notre procédure de routage, et aussi pour permettre l'application de nos travaux à des problématiques réelles, comme nous le verrons dans le chapitre suivant. Nous avons de plus choisi d'avoir du sous-typage structurel pour pouvoir manipuler, au moins de manière approximative, les types tels que les *generics* de Java.

Notre définition du sous-typage est directement reprise de [89], que nous rappelons ici. Nous supposons donnée une relation d'ordre $\leq_s$ sur les constructeurs de types, telle que $(\mathcal{C}, \leq)$ forme un treillis[9]. À partir de cette relation

9. Bien sûr, la relation de sous-typage entre les classes de Java ne forme pas un treillis : il manque l'élément $\top$, et une remarque similaire s'applique aussi au C++. Cette lacune est

initiale, nous construisons Figure 3.18 une relation de sous-typage pour notre système de type : soit $\leq$ la plus petite congruence transitive et réflexive vérifiant la règle décrite Figure 3.18. Cette relation d'ordre est très simple, et permet de

$$\frac{s \leq s' \qquad \forall 1 \leq i \leq \min(m,n),\ E_i \leq_s E'_i}{s(E_1, \ldots, E_n) \leq s'(E'_1, \ldots, E'_m)}$$

FIGURE 3.18 – La relation de sous typage

construire une structure de treillis sur les types élémentaires E, ainsi que les types rangés W^l, les types routés T^R, les types ensemble S et les types de processus F.

Les types valides. Les types, de part leur utilisation, doivent correspondre à une certaine notion de validité sémantique afin de pouvoir s'assurer que les programmes typés sont bien corrects. Or, tels que nous les avons définis, ceux-ci comportent encore quelques erreurs. Nous avons déjà contraint nos types avec l'utilisation de kind s'assurant qu'un nom de champ (resp. un nom de routage) n'était utilisé au plus qu'une seule fois dans un type rangé (resp. dans un type routé). Nous avons aussi signalé la nécessité de contraindre le nombre de déclarations par canal de communication dans les paramètres des types de processus, afin de ne pas retomber dans les travers de notre approche précédente. Enfin, nous avons besoin d'une dernière contrainte, dite de *cohérence*, pour s'assurer de la validité de nos types. Supposons qu'un assemblage D boucle sur un canal e qui est à la fois son canal d'entrée et de sortie. Ce processus a donc un type de la forme $F \triangleq e : (T) \rightarrow e : (T')$ où T est le type des messages qu'il acccepte en entrée, et T' est le type de ceux renvoyés en résultat. Comme ces messages de sortie sont les paramètres de l'assemblage, il faut s'assurer au niveau des types qu'ils sont bien des paramètres valides pour D, et donc, qu'ils sont aussi typables par T : un type F typant l'assemblage D, pour être valide, doit donc vérifier que $T \leq T'$. Par conséquent, nous proposons deux définitions : la première décrit un prédicat définissant formellement la cohérence entre un type ensemble paramètre et un type ensemble résultat ; et la seconde décrit les deux dernières contraintes que nous imposons à nos types.

simplement due au fait que ces langages n'ont pas de constructeurs de types contra-variants, et $\top$ n'est donc pas nécessaire. Il est néanmoins facile d'adapter notre relation de sous-typage à ces langage en y rajoutant le constructeur manquant, ce qui n'a aucune incidence sur le typage des langages d'origine

Définition 6. *Supposons donnés deux types ensemble* $S \triangleq \bigcup_{i \in I} e_i : (T_i)$ *et* $S' \triangleq \bigcup_{i \in I'} e'_i : (T'_i)$. *Nous définissons tout d'abord la relation* $\precsim$ *entre deux types ensemble qui est telle que* $S \precsim S'$ *si et seulement si pour tous canaux* $e \in dc(S) \cap dc(S')$ *et tous types* $T \in S(e)$, *il existe* $T' \in S'(e)$ *avec* $T \leq T'$.

Définition 7. *Un type de processus* $F \triangleq \forall \overline{\alpha}.S_1 \to S_2$ *est* valide *si et seulement si* $S_2 \precsim S_1$ *et pour tous les canaux* $e \in dc(S_1)$ *nous avons* $\#S_1(e) = 1$.

3.2.2 Les substitutions

Le but des substitutions dans un système de type est de permettre l'unification de termes, généralement dans le but de vérifier que l'application (dans le cas de ML) ou l'envoi de message (dans le cas des calculs distribués) est valide : la fonction ou le processus est bien capable de traiter le message reçu en paramètre. Classiquement, cette unification se base sur des variables de types, que l'on peut remplacer par une substitution en un type quelconque (tout en respectant les contraintes de kind quand il y en a). L'unification dans notre système de type reprend ces idées pour les types élémentaires et les types rangés, mais elles ne suffisent pas pour traiter le cas des types routés. En effet, considérons le composant $\mathrm{Acs}(a)[e\,/\,s]$: son type paramètre est typiquement de la forme $T_1 \triangleq \xi[\{a : \mathrm{Pre}(\alpha); \rho\}]$, spécifiant ainsi que l'annotation de routage des messages en paramètre n'est pas contrainte, mais que ceux-ci doivent être des messages structurés ayant le champ a présent. Maintenant, supposons que tout un ensemble de messages est envoyé à ce composant : cet ensemble est typé par un type routé T_3 relativement complexe, et afin de vérifier si ces différents messages sont bien des paramètres valides pour le composant, nous devons être capable d'effectuer l'unification présentée Figure 3.19 (l'arbre représentant bien des paramètres valides pour $\mathrm{Acs}(a)[e\,/\,s]$). La méthode d'unification que nous

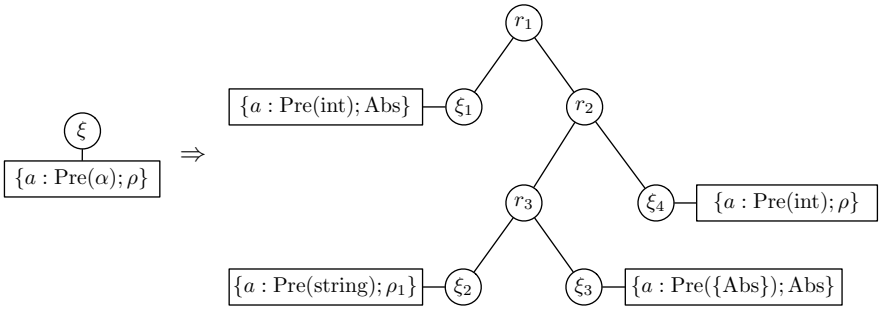


FIGURE 3.19

proposons ici se fait en plusieurs étapes. Nous essayons dans un premier temps de créer, à partir de T_1, un arbre de structure similaire à T_3. Pour ce faire, nous utilisons ξ : cet élément est une variable (de flux) que nous pouvons donc

instancier en un arbre T_2 quelconque (en respectant toutefois les contraintes de kind). Néanmoins, les feuilles de T_2 ne doivent pas être choisies au hasard : elles doivent spécifier, comme T_1, que les messages valides pour le composant $\mathrm{Acs}(a)[e\,/\,s]$ doivent être des messages structurés avec le champ a présent. Cela est résolu dans notre approche en assignant à toute les feuilles de T_2 le type porté par la feuille de T_1 : de fait, cette étape se nomme *duplication*, car elle prend la feuille unique du type original, et la duplique sur toutes les feuilles de l'arbre résultat. Cette duplication, effectuée par une substitution notée σ, est illustrée dans la Figure 3.20. Avec cette première substitution σ, nous arrivons donc à

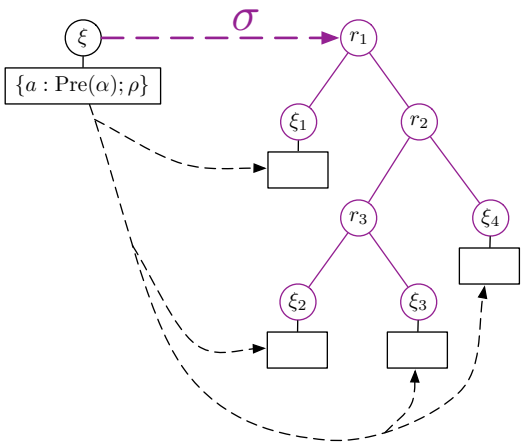


FIGURE 3.20 – La première étape d'unification des types routés

un arbre de structure similaire à T_3 et dont les feuilles indiquent bien quels sont les paramètres valides pour $\mathrm{Acs}(a)[e\,/\,s]$. Nous devons maintenant instancier ces feuilles pour les unifier avec celles de T_3 : nous obtiendrons ainsi le type T_3, et pourrons en conclure que les messages sont bien valides pour notre composant. Une difficulté de cette opération est que les feuilles de T_2 sont toutes identiques, comportent toutes les mêmes variables, alors que celles de T_3 sont deux à deux différentes : ils est donc impossible d'unifier T_2 avec T_3 avec les substitutions classiques qui n'associent qu'une image par variable. Pour résoudre ce problème, nous utilisons ce que nous appelons des substitutions *locales* qui ne s'appliquent que sous une variable de flux donnée. Par exemple si la substitution σ' est locale à ξ, elle modifiera le type $\xi[\alpha]$, mais laissera inchangé le type $\xi'[\alpha]$. Ainsi, en utilisant quatre substitutions locales, nous pouvons terminer notre unification, comme cela est montré Figure 3.21.

En conclusion, l'ajout de la duplication et des substitutions locales nous permet d'unifier des types routés. Cette capacité nous servira grandement lors de la définition des règles de typage, où l'envoi de message à un processus pourra être typé de manière classique, par l'utilisation de substitutions (contrairement

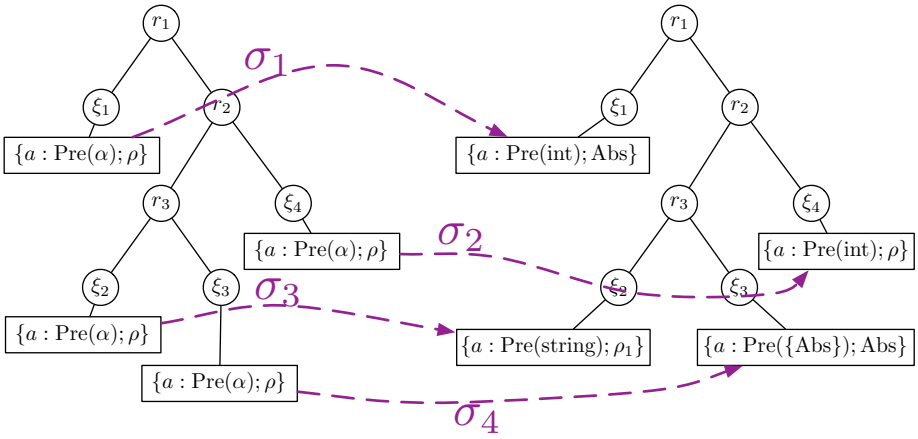


FIGURE 3.21 – Les étapes finales d'unification des types routés

à notre première approche, où notre règle d'application utilisait une clause de la forme $\forall T \in S(e)$). Néanmoins, cette capacité d'unification n'est pas souhaitable dans certains cas, et doit être contrôlée. C'est pourquoi nous définissons une nouvelle forme de kind : les kind de substitutions.

Les kinds de substitution. Nous avons déjà introduit deux kinds, une pour les types rangés et une pour les types routés, dont l'objectif est de contraindre la syntaxe des types valides. Nous rajoutons deux nouvelles kinds aux variables : les variables de types élémentaires et de types rangés se voient attribuer une kind de *localité* pouvant être soit $\mathfrak{L}$ ou $\mathfrak{G}$, alors que les variables de routage possèdent une kind de *duplication* pouvant être soit $\mathfrak{D}$ ou bien $\mathfrak{F}$. Ces kinds servent à limiter l'utilisation des substitutions locales et de la duplication. Considérons en effet un message M avec une annotation de routage vide : son type est de la forme $\xi[E]$. Ce type correspondant à un message unique, il n'est pas souhaitable de pouvoir dupliquer son type (ce qui correspondrait à considérer un message unique comme un ensemble de message). C'est pourquoi dans ce cas, nous associons la kind $\mathfrak{F}$ à la variable ξ : elle n'est maintenant plus duplicable et nous assure que le message M sera bien considéré comme unique par notre système de type. Les variables de flux des types des composants primitifs ont par contre généralement une kind $\mathfrak{D}$. La motivation pour les kinds de localité est un peu plus subtile : considérons le composant primitif $\mathrm{Add}(a)[e_1 e_2 \,/\, s]$ qui attend un message structuré $v_1^{\delta_1}$ sur e_1, un autre message $v_2^{\delta_2}$ sur e_2 et renvoie sur s le message $(v_1 + (a = v_2))^{\delta_1}$. Le type de ce composant est donc de la forme

$$F \triangleq e_1 : (\xi_1[\{a : \mathrm{Abs}; \rho\}]) \cup e_2 : (\xi_2[\eta]) \to s : (\xi_1[\{a : \mathrm{Pre}(\eta); \rho\}])$$

Maintenant, considérons que ce composant soit utilisé avec un entier sur son entrée e_2. Si nous utilisons une substitution locale pour instancier F, nous ob-

tenons

$$e_1 : (\xi_1[\{a : \mathrm{Abs}; \rho\}]) \cup e_2 : (\xi_2[\mathrm{int}]) \to s : (\xi_1[\{a : \mathrm{Pre}(\eta); \rho\}])$$

Nous pouvons remarquer que η n'a pas été remplacé dans le type résultat : une substitution locale n'est pas suffisante pour traiter le composant $\mathrm{Add}(a)[e_1 e_2 \,/\, s]$, et en général, les composants faisant des jointures. C'est pourquoi dans ce cas, nous associons la kind $\mathfrak{G}$ à la variable η : elle est maintenant globale et ne peut être manipulée par des substitutions locales, seules les substitutions classiques peuvent la modifier. Néanmoins, les variables de types élémentaires et de types rangés sont généralement locales. Finalement, nous notons `kind` l'opérateur qui donne la kind des variables, et nous l'étendons aux types élémentaires et types rangés Figure 3.22 : cela sera utile pour définir la validité d'une substitution dans le paragraphe suivant. Afin de factoriser les éléments de la figure, nous notons $\mathfrak{G} \cap \mathfrak{G} \triangleq \mathfrak{G}$ et $\mathfrak{L} \cap \kappa \triangleq \mathfrak{L}$, κ étant soit $\mathfrak{L}$ ou $\mathfrak{G}$.

$$\mathtt{kind}(\{W^{\emptyset}\}) = \mathtt{kind}(W^{\emptyset}) \qquad \mathtt{kind}(s(E_1, \ldots, E_n)) = \bigcap_i \mathtt{kind}(E_i)$$

$$\mathtt{kind}(\mathrm{Abs}^l) = \mathtt{kind}(\mathrm{Abs}) = \mathfrak{G} \qquad \mathtt{kind}(\mathrm{Pre}(E)) = \mathtt{kind}(E)$$

$$\mathtt{kind}(a : K; W^l) = \mathtt{kind}(K) \cap \mathtt{kind}(W^l)$$

FIGURE 3.22 – La kind des messages

La présentation formelle des substitutions. Comme nous l'avons vu jusqu'à présent, nos substitutions sont des constructions complexes assez difficile à présenter simplement. C'est pourquoi nous avons choisi de les définir par une syntaxe relativement simple, donnée Figure 3.23, que nous restreignons par la suite pour respecter les contraintes de kinds des variables. Finalement nous donnons aussi la sémantique de ces substitutions, afin qu'elles puissent effectivement manipuler les types. Notons que nous utilisons α pour désigner soit une variable de type élémentaire η soit une variable de type rangé ρ, et τ pour designer un type élémentaire E ou un type rangé W^l.

Les substitutions sont construites à partir de la substitution identité id, qui ne modifie aucune variable, la composition de substitutions $\sigma_2 \circ \sigma_1$, le remplacement *global* d'une variable par un type $\alpha \to \tau$, ou son remplacement local $\xi^R[\alpha \to \tau]$, dans le cas où la variable α est locale. Nous avons aussi l'instanciation des variables de routage $\xi^R \to s^{R'}$ qui identifie ces variables à des *formes*.

$$
\begin{array}{rcll}
\sigma & ::= & \mathrm{id} \quad | \quad \sigma_2 \circ \sigma_1 \quad | \quad \alpha \to \tau & \text{Substitution de base} \\
 & | & \xi^R[\alpha \to \tau] & \text{Substitution locale} \\
 & | & \xi^R \to s^{R'} \quad \text{with } R \subset R' & \text{Instantiation de routage} \\
s^R & ::= & \xi^R[\bullet] \quad | \quad r(s_1^{R \uplus \{r\}}, s_2^{R \uplus \{r\}}) & \text{Forme}
\end{array}
$$

FIGURE 3.23 – Ses substitutions

Ces formes sont typiquement des types routés, avec un trou '•' à la place que devrait prendre un type élémntaire. C'est ce trou qui est remplacé par le type contenu par ξ^R lors de l'application de la substitution, comme cela a été présenté Figure 3.20.

$$
\vdash \mathrm{id} \qquad
\frac{\vdash \sigma_1 \qquad \vdash \sigma_2}{\vdash \sigma_2 \circ \sigma_1} \qquad
\frac{\mathtt{kind}(\xi^R) = \mathfrak{D}}{\vdash \xi^R \to s^{R'}} \qquad
\frac{\mathtt{kind}(\eta) = \mathtt{kind}(E) = \mathfrak{G}}{\vdash \eta \to E}
$$

$$
\frac{\mathtt{kind}(\rho^l) = \mathtt{kind}(W^l) = \mathfrak{G}}{\vdash \rho^l \to W^l} \qquad
\frac{\mathtt{kind}(\eta) = \mathtt{kind}(E) = \mathfrak{L}}{\vdash \xi^R[\eta \to E]}
$$

$$
\frac{\mathtt{kind}(\rho^l) = \mathtt{kind}(W^l) = \mathfrak{L}}{\vdash \xi^R[\rho^l \to W^l]} \qquad
\frac{\mathtt{kind}(\xi_1^{R_1}) = \mathfrak{F}}{\xi_1^{R_1} \to \xi_2^{R_2}[\bullet]}
$$

FIGURE 3.24 – Les règles de validation des substitutions

Comme nous l'avons introduit, les substitutions sont de plus contraintes par un ensemble de règles à respecter, afin que les kinds des variables soient correctement manipulées. Par exemple, une variable de routage non duplicable ne peut être dupliquée, et les variables globales ne peuvent être remplacées par un type contenant une variable locale. Toutes ces règles sont formellement présentées Figure 3.24.

Finalement, Figure 3.25 montre comment l'application des substitutions aux types fonctionne. Les substitutions ne s'appliquent qu'aux types routés dans notre présentation, et sont naturellement étendues aux types ensembles et types

de processus par induction sur la structure du type. Il est à noter que l'on peut aisément étendre leur définitions aux formes aussi, ce qui sera utile dans le chapitre suivant.

$$\mathrm{id}(\xi^R[E]) = E \qquad \sigma_1 \circ \sigma_2(\xi^R[E]) = \sigma_1(\sigma_2(\xi^R[E]))$$

$$(\eta \to E)(\xi^R[E']) = \xi^R[E'\{^E/_\eta\}] \qquad (\rho^l \to W^l)(\xi^R[E']) = \xi^R[E'\{^{W^l}/_{\rho^l}\}]$$

$$\xi_1^{R_1}[\eta \to E](\xi_2^{R_2}[E']) = \begin{cases} \xi_1^{R_1}[E'\{^E/_\eta\}] & \text{si } \xi_1^{R_1} = \xi_2^{R_2} \\ \xi_2^{R_2}[E'] & \text{sinon} \end{cases}$$

$$\xi_1^{R_1}[\rho^l \to W^l](\xi_2^{R_2}[E']) = \begin{cases} \xi_1^{R_1}[E'\{^{W^l}/_{\rho^l}\}] & \text{si } \xi_1^{R_1} = \xi_2^{R_2} \\ \xi_2^{R_2}[E'] & \text{sinon} \end{cases}$$

$$(\xi_1^{R_1} \to s^{R'})(\xi_2^{R_2}[E]) = \begin{cases} s^{R'}\{^E/_\bullet\} & \text{si } \xi_1^{R_1} = \xi_2^{R_2} \\ \xi_2^{R_2}[E] & \text{sinon} \end{cases}$$

FIGURE 3.25 – La sémantique des substitutions

Finalement, nous prouvons une propriété importante de la validité des types, qui est gardée même après l'application d'une substitution. C'est en effet une condition nécessaire pour s'assurer que les types que notre système donne aux processus sont bien valides eux aussi.

Lemme 1. *Supposons donné un type de processus F valide, et une substitution σ. Alors le type $\sigma(F)$ est valide lui aussi.*

3.2.3 Les règles de typage

L'introduction à nos règles de typage est presque terminée. Nous avons décrit la syntaxe de nos types valides, ainsi que l'équivalence structurelle, le sous-typage et les substitutions permettant de manipuler ces types valides. Il ne nous reste donc plus qu'à présenter dans le paragraphe suivant les fonctions permettant de typer les constantes ainsi que les composants primitifs pour finaliser cette introduction.

Le typage des constantes et des composants primitifs. Nous suivons ici la même approche que dans notre premier système de type : les types des constantes (resp. des composants primitifs) sont donnés par la fonction Ψ (resp. la fonction Ω) dont les images sont des types élémentaires (resp. des types de processus). Comme il n'y a pas d'application ni de réduction dans les messages, nous n'avons pas besoin de contraindre Ψ comme cela a été fait dans notre

première approche. C'est maintenant Ω qui est contraint, car ce sont les composants primitifs qui servent de base à la sémantique opérationnelle de notre calcul. Informellement, nous imposons deux types de restrictions sur Ω : (i) cette fonction ne doit pas permettre qu'un paramètre non-valide d'un composant primitif p puisse être accepté par le type de p ; et (ii) cette fonction doit prendre en compte les différents résultats qu'un composant p peut envoyer à la suite d'une réception de paramètres valides J. Formellement, Ω est tel que :

1. Pour tout composant primitif p, tout motif J tel que $dc(J) \subset \texttt{Inc}(p)$ et tout type de processus $F \triangleq \forall\overline{\gamma}.S_1 \to S_2$ tel que le jugement de type $p \mid J : F$ est valide, il existe J' tel que $\texttt{match}(p, J \mid J')$.
2. De plus, il existe un type de processus $S_1' \to S_2'$ tel que $S_1' \subset S_1$ et $p \mid \gamma(p, J \mid J') : F'$ est valide.

Finalement, nous demandons aussi que $\Omega(p)$ soit valide pour tout composant primitif p.

Le typage des messages et messages annotés. Les règles de typage pour ces éléments sont présentées Figure 3.26. Les jugements de typage pour les messages sont de la forme $v : E$ où E est le type de v. Nous n'avons en effet pas besoin d'environnement de typage pour typer nos messages, ni même d'ailleurs les messages routés ni les processus, car notre langage ne comporte pas de lieur de variables, tel que l'abstraction du λ-calcul ou la réception de message des calculs de processus classiques. Les messages sont typés sans difficulté avec les deux règles (T :Constante) et (T :Message). Le typage des messages annotés est un peu plus subtil, et nécessite des jugements de la forme $R \vdash M : T$ où T est le type de M et R l'ensemble des noms de routage ne pouvant apparaitre dans T. Typiquement, R permet de s'assurer qu'un nom de routage n'apparait qu'une seule fois dans les branches de l'arbre T (et donc dans les annotations de routage correspondantes). Le typage des messages annotés v^δ se fait par induction sur la taille de l'annotation δ. Lorsqu'elle est vide, nous appliquons la règle (T :Vide) qui crée une feuille à partir du type du message v et d'une variable de flux quelconque dont la kind de substitution est $\mathfrak{F}$: cette feuille ne peut pas être dupliquée. Maintenant, si l'annotation comporte des informations de routage telles que $\uparrow r$ ou $\downarrow r$, nous devons étendre cette première feuille en un arbre de routage plus complexe, afin de prendre en compte ces informations dans notre type. Cela est fait avec les règles (T :Droit) et (T :Gauche) qui rajoute une nouvelle racine r à l'arbre précédemment construit, tout en s'assurant (en manipulant l'ensemble R) que r n'a pas déjà été utilisé. De fait, ces trois dernières règles forment le point négatif principal de nos types routés : elles ne sont pas dirigées par la syntaxe (la règle (T :Vide) introduit arbitrairement une variable de flux, alors que les règles (T :Droit) et (T :Gauche) créent à partir de rien tout un sous-arbre de routage). Ainsi, un message unique doit être typé par un type routé arbitrairement gros simulant l'existence de nombreux messages ayant des annotations différentes.

$$\text{T :Constante}\quad c : \Psi(c)$$

$$\text{T :Message}\quad \frac{\forall 1 \leq i \leq n,\ v_i : E_i \qquad \forall 1 \leq i \neq j \leq n,\ a_i \neq a_j}{\{a_1 = v_1; \ldots; a_n = v_n\} : \{a_1 : \mathrm{Pre}(E_1); \ldots; a_n : \mathrm{Pre}(E_n); \mathrm{Abs}\}}$$

$$\text{T :Vide}\quad \frac{M : E \qquad R \subset R' \qquad \mathtt{kind}(\xi^{R'}) = \mathfrak{F}}{R \vdash M^{\emptyset} : \xi^{R'}[E]}$$

$$\text{T :Droit}\quad \frac{\mathcal{R} \cup \{r\} \vdash M^{\delta} : T \qquad \mathcal{R} \cup \{r\} \vdash T_k}{\mathcal{R} \vdash M^{\uparrow r;\delta} : r(T, T_k)}$$

$$\text{T :Gauche}\quad \frac{\mathcal{R} \cup \{r\} \vdash M^{\delta} : T \qquad \mathcal{R} \cup \{r\} \vdash T_k}{\mathcal{R} \vdash M^{\downarrow r;\delta} : r(T_k, T)}$$

Figure 3.26 – Les règles de typage pour les messages et messages routés

Le typage des processus. Les règles de typage pour nos composants et processus sont présentées Figure 3.27. Nous avons choisi, en formant ces règles, de suivre une approche différente de ce qui se fait généralement pour le typage des calculs distribués : nous adoptons ici un typage compositionnel où le type assigné à un processus ne dépend pas du contexte dans lequel celui-ci s'exécute. Par exemple, dans notre première approche, les types que nous assignons aux processus étaient globaux, et le même type était donné à chaque sous-partie d'un programme complexe. Ici, le type de chaque partie lui est propre, ce qui permet d'avoir un typage assez précis où les types donnent une bonne approximation de la sémantique des programmes. Par conséquent, nos jugements de typage pour les processus sont de la forme $D : F$ afin de signaler que F n'est pas un paramètre du typage, mais bien un résultat. De fait, la majorité de nos règles de typage sont assez classiques : le typage des composants primitifs est similaire à celui des constantes, nous avons une règle d'instanciation permettant d'ôter un lieur d'un schéma de type, une règle de substitution appliquant une substitution à un type de processus, une règle de subsomption permettant d'inclure le sous-typage à notre système ainsi qu'une règle de généralisation rajoutant un lieur à un schéma de type. Cette dernière règle n'est pas essentielle, car comme nous l'avons déjà présenté, notre calcul ne comporte par de construction telle que le `let` de ML. Nous l'avons toutefois intégrée pour faciliter la manipulation des noms de variables lors du typage des composants, et éviter une capture de nom non voulue. Les trois dernières règles sont toutefois propres à notre système de type. (T :Parallèle) présente comment la composition parallèle est typée, et comporte trois contraintes sur les deux processus en parallèle. La première

$$\text{T :Primitif}\quad p : \Omega(p)$$

$$\text{T :Inst}\quad \frac{D : \forall\alpha.F}{D : F}$$

$$\text{T :Subst}\quad \frac{D : F}{D : \sigma(F)}$$

$$\text{T :Sub}\quad \frac{D : F \qquad F \leq F'}{D : F'}$$

$$\text{T :Gen}\quad \frac{D : F}{D : \forall\alpha.F}$$

$$\text{T :Parallèle}\quad \frac{D : S_1 \to S_2 \qquad D' : S'_1 \to S'_2 \qquad S_2 \precsim S'_1 \qquad S'_2 \precsim S_1 \qquad dc(S_1) \cap dc(S'_1) = \emptyset}{D \mid D' : (S_1 \cup S'_1) \to (S_2 \cup S'_2)}$$

$$\text{T :Canal}\quad \frac{\emptyset \vdash M : T}{\overline{e}\langle M\rangle : \emptyset \to e : (T)}$$

$$\text{T :Boîte}\quad \frac{D : S_1 \to S_2}{c[I\,/\,O][D] : (S_1 \cap I) \to (S_2 \cap O)}$$

Figure 3.27 – Les règles de typage des processus

contrainte $S_2 \precsim S'_1$ impose que tous les messages envoyés par D' sur des canaux entrants de D sont des entrées valides pour D. La seconde contrainte est la réciproque : les messages envoyés par D sur les canaux entrant de D' doivent être des paramètres valides pour D'. Finalement la dernière contrainte s'assure que le processus resultant de la composition parallèle n'a qu'un seul composant écoutant sur chaque canal entrant : son type est donc valide [10]. Afin de pouvoir typer les envois de messages par des types de processus et ainsi avoir un typage uniforme, nous utilisons la règle (T :Canal) qui identifie (pour les types) ces processus à des composants sans entrée et envoyant le message sur la bonne sortie. Finalement, la règle (T :Boîte) type les composants, et prend en compte l'isolation en limitant le type affiché par le composant à ses canaux d'entrée et de sortie.

3.2.4 Exemples et discussion

Le système de type que nous venons de définir est novateur sur beaucoup de point. Nous avons en effet les types routés qui forment une nouvelle catégorie de type axée sur la prise en compte du routage ; notre système de type est *compositionnel*, ce qui est assez rare dans les calculs de processus ; et ce dernier prend finalement en compte l'isolation impliquée par la structure des composants de notre calcul. Nous présentons ici une série d'exemples illustrant et discutant ces

10. Une autre possibilité pour s'assurer que le type donné à une composition parallèle $D_1 \mid D_2$ est bien valide serait simplement d'imposer que tout les canaux d'entré partagés par D_1 et D_2 portent le même type.Nous n'avons pas opté pour cette solution ici car son intérêt nous paraissait limité : elle demande en effet plus de constructions purement techniques sans apporter quoique ce soit à la motivation de notre travail, le routage

différentes caractéristiques, afin de bien comprendre comment elles s'intègrent dans notre système de type. Notons que dans tous nos exemples, les variables de types élémentaires et de types rangés seront, sauf mention explicite du contraire, supposées locales : $\texttt{kind}(\eta) = \texttt{kind}(\rho) = \mathfrak{L}$. Il en est de même pour les variables de flux que nous supposerons par défaut duplicables : $\texttt{kind}(\xi) = \mathfrak{D}$.

Les composants primitifs. Notre premier exemple, décrit Figure 3.28, introduit les types des composants primitifs de base, comme le routeur, le multiplexeur ou les composants manipulant les messages. Il est intéressant de com-

Un composant p	**Le types** $\Omega(p)$
GenStr	$\emptyset \to s : (\{\texttt{Str} : \mathrm{Pre}(\mathrm{string}); \mathrm{Abs}\})$
AcsTTL	$\forall \xi, \eta.e : (\xi[\{\texttt{TTL} : \mathrm{Pre}(\eta); \rho\}]) \to s : (\xi[\eta])$
AddTTL	$\forall \xi, \xi', \eta, \rho.e_1 : (\xi[\{\texttt{TTL} : \mathrm{Abs}; \rho\}]) \cup e_2 : (\xi'[\eta])$ $\to s : (\xi[\{\texttt{TTL} : \mathrm{Pre}(\eta); \rho\}])$ avec $\texttt{kind}(\eta) = \mathfrak{G}$
RmTTL	$\forall \xi, \eta, \rho.e : (\xi[\{\texttt{TTL} : \mathrm{Pre}(\eta); \rho\}])$ $\to s : (\xi[\{\texttt{TTL} : \mathrm{Abs}; \rho\}])$
r R	$\forall \xi_1, \xi_2, \eta_1, \eta_2.e : (r(\xi_1[\eta_1], \xi_2[\eta_2]))$ $\to s_1 : (\xi_1[\eta_1]) \cup s_2 : (\xi_2[\eta_2])$
r M	$\forall \xi_1, \xi_2, \eta_1, \eta_2.e_1 : (\xi_1[\eta_1]) \cup e_2 : (\xi_2[\eta_2])$ $\to s : (r(\xi_1[\eta_1], \xi_2[\eta_2]))$

FIGURE 3.28 – Exemples de types pour composants primitifs

parer ces types à ceux que nous avons donnés aux composants primitifs (Figure 2.24) de notre première version, et de voir quelles sont leurs similitudes. Tout d'abord, nous pouvons voir que pour les composants manipulant la structure des messages (sauf pour le composant $\mathrm{Add}(\texttt{TTL})[e_1 e_2 \,/\, s]$ car il n'a pas la même sémantique que celui de notre première version), la modification permettant de passer de notre première version des types vers celle de notre approche actuelle suit toujours le même motif :

1. Séparation entre la partie paramètre et la partie résultat du type
2. Encapsulation des types de message par une variable de flux afin de prendre en compte nos annotations de routage
3. Liage des variables libres du type résultant

Considérons par exemple le composant $\mathrm{Acs}(\mathtt{TTL})[e\,/\,s]$: la séparation entre ses paramètres et ses résultats donne le terme $e : (\{\mathtt{TTL} : \mathrm{Pre}(\eta); \rho\}) \to s : (\eta)$; l'encapsulation donne le type $e : (\xi[\{\mathtt{TTL} : \mathrm{Pre}(\eta); \rho\}]) \to s : (\xi[\eta])$; et finalement, lier les variables libres de ce type donne le type que nous avons assigné à ce composant. De fait, ce motif s'applique très bien à la plupart des types des composants qui ne sont pas dédiés au routage. Une petite difficulté survient toutefois pour les composants mixant dans un même résultat des données venant de plusieurs sources, comme ce qui est fait par le composant $D \triangleq \mathrm{Add}(\mathtt{TTL})[e_1 e_2\,/\,s]$. Dans ce cas, il faut bien contrôler la kind de substitution des variables : la variable η dans le type de D est globale. Concernant les routeurs et multiplexeurs, leurs types sont incomparables avec ceux de notre approche précédente. En effet, ces composants utilisent maintenant les annotations de routage pour guider les messages au travers des assemblages. Informellement, les types que nous donnons au multiplexeur et au routeur nommés r suivent le schéma donné Figure 3.29. Le multiplexeur possède deux entrées, e_1 et e_2 typées par deux types

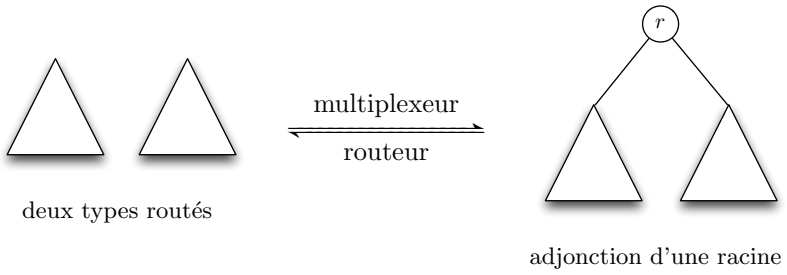


FIGURE 3.29 – L'effet des opérations de routage sur les types routés

routés, correspondant à deux ensembles de messages. Ces deux ensembles étant inconnus, leurs types sont symbolisés dans notre figure par deux triangles et dans notre syntaxe par des termes de la forme $\xi[\eta]$ (notons toutefois que ces messages ne peuvent être annotés par r, et ainsi, notre type interdit par la règle des kinds que leur type comporte un noeud r). Finalement le multiplexeur annote les messages reçus sur e_1 par $\downarrow r$, plaçant le premier type routé comme sous-arbre gauche d'une racine r, alors que les messages reçus sur e_2 sont annotés par $\uparrow r$, plaçant le second type routé à droite de cette racine. Le type résultat de ce composant est donc $r(\xi_1[\eta_1], \xi_2[\eta_2])$, ce qui correspond sans faille à sa sémantique opérationnelle. Le routeur ayant une sémantique duale, l'opération qu'il effectue sur les types est elle-même opposée, ce qui correspond au type que nous lui avons assigné. Finalement, nous pouvons remarquer que, contrairement aux types de notre première approche, ceux que nous donnons à nos

composants ici sont principaux. Cette propriété est due en grande partie au fait que nous avons un typage compositionnel et que nous n'autorisons qu'une seule déclaration de canal dans la partie paramètre : nous n'avons donc pas besoin d'une notion de type minimal pour limiter un type à la sémantique d'un composant en particulier.

Le sous-typage. Notre exemple suivant illustre l'utilisation du sous-typage dans la définition du type d'un assemblage. Pour ce faire, nous supposons donnés un ensemble de constructeurs de type (*Object*, *Integer*, *String*, etc) et une relation de sous-typage, les deux similaires à ce que l'on peut trouver dans une distribution classique de Java ; de plus, nous considérons que nous avons à disposition un certain nombre de composants primitifs copiant certaines fonctions de base de Java : (i) *Integer* génère des objets du type *Integer* sur son interface de sortie ; (ii) *getClass* retourne en sortie la classe de l'objet reçu en paramètre ; etc. Notre exemple, décrit Figure 3.30, consiste en une simple chaîne de composants, créant à son origine des objets de type *Integer* et affichant en sortie le nom de la classe de ces objets. Cet assemblage (que nous notons D) corres-

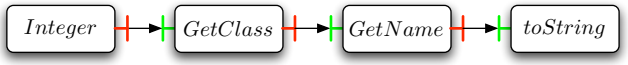


FIGURE 3.30

pond à un appel de procédure totalement valide en Java, mais nécessite, pour être typable dans ce langage, l'utilisation du sous-typage impliqué par l'héritage des classes. La problématique est similaire dans notre exemple : le composant *GetClass* prend en paramètre n'importe quelle valeur Java (typé par *Object*), et nous lui envoyons un entier Java (typé par *Integer*) ; le composant *toString* prend lui aussi en paramètre une valeur Java quelconque, et nous lui envoyons une chaîne de caractères (typée par *String*). Comme notre approche actuelle prend en compte le sous-typage, cet assemblage est typable comme le montre la dérivation de type J Figure 3.31. Dans ce type, le canal c_1 (resp. c_2, c_3) correspond au canal de communication entre les composants *integer* et *GetClass* (resp. *GetClass* et *GetName*, *GetName* et *toString*). Nous pouvons remarquer que ce type consiste en une simple juxtaposition des comportements de chaque composant de l'assemblage : dans la partie paramètre, $c_1 : (\xi[Object])$ spécifie que le composant *GetClass* attend n'importe quel objet Java en entrée, alors que dans la partie résultat, $c_1 : (\xi[Integer])$ montre que le composant *Integer* lui envoie des entiers. Ainsi, le sous-typage est utilisé dans ce type pour valider le fait qu'un entier est un paramètre valide d'un composant demandant des simples objets Java. De fait, nous pouvons aussi donner un typage moins précis à cet

$$J_1 \triangleq \dfrac{\begin{array}{c} \dfrac{Integer\ :\ \forall\xi.\emptyset \to c_1 : (\xi[Integer])}{Integer\ :\ \emptyset \to c_1 : (\xi[Integer])} \\ \vdots \qquad \dfrac{GetClass\ :\ \forall\xi.c_1 : (\xi[Object]) \to c_2 : (\xi[Class])}{GetClass\ :\ c_1 : (\xi[Object]) \to c_2 : (\xi[Class])} \end{array}}{\begin{array}{l} Integer \mid GetClass\ : \\ c_1 : (\xi[Object]) \to c_1 : (\xi[Integer]) \cup c_2 : (\xi[Class]) \end{array}}$$

$$J_2 \triangleq \dfrac{\begin{array}{c} \dfrac{GetName\ :\ \forall\xi.c_2 : (\xi[Class]) \to c_3 : (\xi[String])}{GetName\ :\ c_2 : (\xi[Class]) \to c_3 : (\xi[String])} \\ \vdots \qquad \dfrac{toString\ :\ \forall\xi.c_3 : (\xi[Object]) \to \emptyset}{toString\ :\ c_3 : (\xi[Object]) \to \emptyset} \end{array}}{\begin{array}{l} GetName \mid toString\ : \\ c_2 : (\xi[Class]) \cup c_3 : (\xi[Object]) \to c_3 : (\xi[String]) \end{array}}$$

$$J \triangleq \dfrac{J_1 \qquad J_2}{D\ :\ \left(\begin{array}{c} c_1 : (\xi[Object]) \cup c_2 : (\xi[Class]) \cup c_3 : (\xi[Object]) \\ \to c_1 : (\xi[Integer]) \cup c_2 : (\xi[Class]) \cup c_3 : (\xi[String]) \end{array}\right)}$$

FIGURE 3.31

assemblage comme le montre la dérivation J Figure 3.32. Ici, le sous-typage est utilisé au travers de la règle de subsomption qui unifie les types des paramètres avec ceux des sorties. Le type est alors moins précis que le premier que nous avons donné, car il est impossible d'en déduire que le composant $Integer$ génère des entiers : nous savons simplement qu'il génère des valeurs Java. Finalement, afin de simplifier la correspondance entre nos assemblages et nos types, nous adoptons dans la suite une charte graphique pour nos types, similaire à celle que nous avons utilisée dans notre première approche. La Figure 3.33 donne à notre exemple le type F_1 : chaque canal porte deux annotations de types, correspondant au type de sa partie paramètre ainsi que celui de sa partie résultat. Notons que le canal c_2 (entre le composant $GetClass$ et $GetName$) échappe à la règle, car sa partie paramètre a le même type que sa partie résultat : dans ce cas, nous donnons seulement une annotation pour le canal.

$$
J_1 \triangleq \dfrac{\dfrac{\dfrac{\dfrac{Integer : \forall\xi.\emptyset \to c_1 : (\xi[Integer])}{Integer : \emptyset \to c_1 : (\xi[Integer])}}{Integer : \emptyset \to c_1 : (\xi[Object])} \quad \vdots \quad \dfrac{\dfrac{GetClass : \forall\xi.c_1 : (\xi[Object]) \to c_2 : (\xi[Class])}{GetClass : c_1 : (\xi[Object]) \to c_2 : (\xi[Class])}}{}}{\begin{array}{l} Integer \mid GetClass : \\ c_1 : (\xi[Object]) \to c_1 : (\xi[Object]) \cup c_2 : (\xi[Class]) \end{array}}
$$

$$
J_2 \triangleq \dfrac{\dfrac{\dfrac{\dfrac{GetName : \forall\xi.c_2 : (\xi[Class]) \to c_3 : (\xi[String])}{GetName : c_2 : (\xi[Class]) \to c_3 : (\xi[String])}}{GetName : c_2 : (\xi[Class]) \to c_3 : (\xi[Object])} \quad \vdots \quad \dfrac{\dfrac{toString : \forall\xi.c_3 : (\xi[Object]) \to \emptyset}{toString : c_3 : (\xi[Object]) \to \emptyset}}{}}{\begin{array}{l} GetName \mid toString : \\ c_2 : (\xi[Class]) \cup c_3 : (\xi[Object]) \to c_3 : (\xi[Object]) \end{array}}
$$

$$
J \triangleq \dfrac{J_1 \qquad J_2}{D : \begin{pmatrix} c_1 : (\xi[Object]) \cup c_2 : (\xi[Class]) \cup c_3 : (\xi[Object]) \\ \to c_1 : (\xi[Object]) \cup c_2 : (\xi[Class]) \cup c_3 : (\xi[Object]) \end{pmatrix}}
$$

FIGURE 3.32

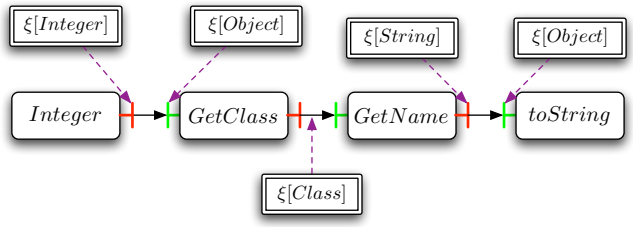


FIGURE 3.33

Les chaînes de composants. Notre troisième exemple illustre simplement comment nous typons le routage et le multiplexage dans cette nouvelle approche. La Figure 3.34 décrit un assemblage classique où deux composants envoient des données différentes, qui sont placées sur un même canal central pour être ensuite départagées et traitées par des composants adaptés. Nous pouvons voir dans cette figure que contrairement à notre approche précédente, les canaux n'ont bien qu'un seul type, le routage étant pris en compte par le type routé $r(\xi[String], \xi'[Integer])$: le canal central étant utilisé par des chaînes de caractères annotées par $\uparrow r$ et des entiers annotés par $\downarrow r$. Une propriété intéressante

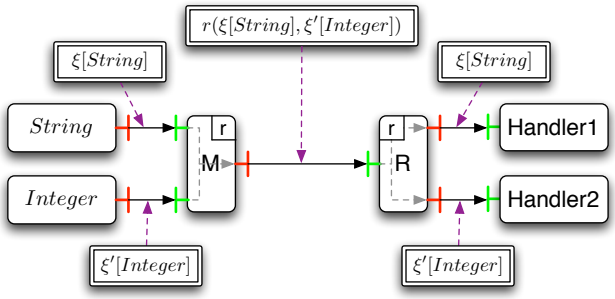


FIGURE 3.34

de notre système de type est que l'on peut combiner du sous-typage à la Java et notre méthode de routage, comme cela est illustré Figure 3.35. Cet exemple reprend notre chaîne de composants (Figure 3.30) en y intégrant une procédure de multiplexage afin de pouvoir traiter à la fois des entiers et des chaînes de caractères. De plus, cet assemblage est assez caractéristique d'un parallèle que l'on peut faire entre le typage à la Java (ou ML, c'est à dire un langage séquentiel classique) et notre approche : ce processus correspond en effet à deux créations de valeurs Java (un entier et une chaîne de caractères) sur lesquelles est appliquée une série de méthodes Java, comme cela est fait dans le programme suivant :

```
(new Integer(1)).getClass().getName().toString();
(new String()).getClass().getName().toString();
```

Au niveau du typage, Java fait donc deux dérivations de types, une pour chaque expression, permettant d'utiliser la règle de subsomption deux fois (entre *Integer* et *Object*, et entre *String* et *Object*). Dans notre cas, nous n'avons qu'une seule chaîne de composants (c'est à dire une seule expression), mais les annotations de routage ont un effet similaire à l'existence de deux expressions : elles permettent de créer un type routé à deux branches subissant, par congruence sur la structure du type, deux applications différentes de la règle de subsombtion

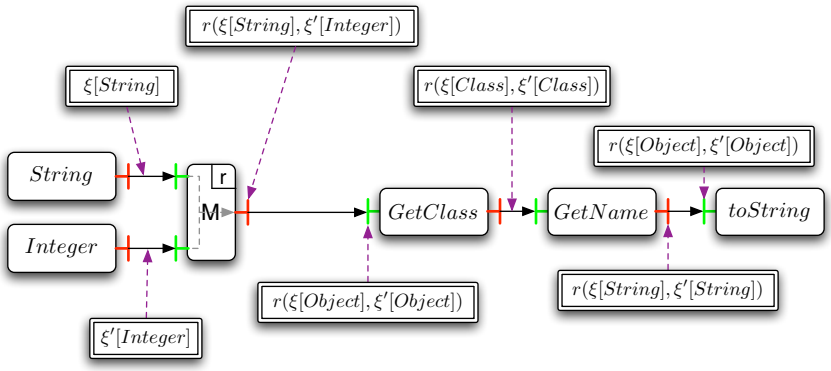


FIGURE 3.35

(lors de la validation du type de l'assemblage). Ainsi, les annotations de routage permettent de simuler dans notre calcul l'existence de plusieurs expressions dans un langage classique comme Java. Cette remarque est aussi valide avec le polymorphisme : les annotations de routage ont un effet similaire à l'utilisation de plusieurs instances d'une même fonction, permettant d'instancier plusieurs fois son type.

Portes et isolation. Notre quatrième exemple montre l'action qu'a un composant composite sur le type d'un assemblage. Notre assemblage Figure 3.36 consiste en une simple chaîne de composants manipulant les messages structurés que nous lui donnons en paramètre. Comme nous l'avons vu jusqu'à présent, le type d'une telle chaîne peut être assez conséquent et notre typage des composants composites permet de le simplifier : le type global est basé sur le type de la chaîne, auquel nous avons retiré toutes les informations strictement internes au composant composite, c'est à dire concernant les canaux qui ne font pas partie des portes entrantes ou sortantes de l'assemblage. En conséquence, le type de notre exemple est extrêmement simple et stipule juste que les messages en paramètres ne peuvent pas comporter de champs `IP` et `TTL`, et que l'assemblage retourne en résultat les messages reçus sans modification.

Les boucles. Nos trois figures suivantes présentent les types de boucles que nous pouvons typer avec notre approche. Contrairement à notre première approche, le système de type actuel est assez restrictif sur l'utilisation des annotations de routage : un nom de routage ne peut apparaître au plus qu'une seule fois dans une annotation. C'est pourquoi la plupart des assemblages formant des boucles que nous avons étudiés dans la partie sur le calcul étaient erronés. Nous présentons Figure 3.37 le seul composant de notre comparatif précédent

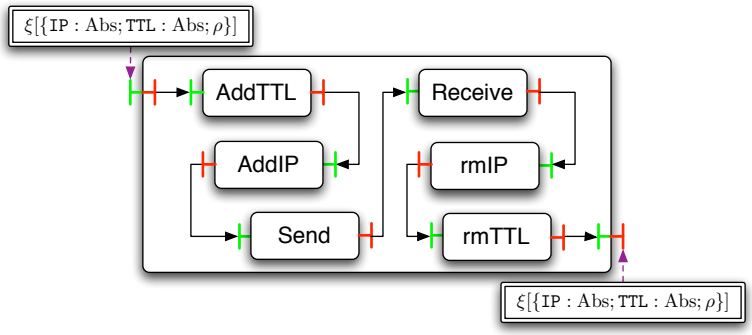


FIGURE 3.36

qui est typable. Ce composant possède une structure de boucle (sur le composant `AcsSub`) qui, en réalité, ne s'exécute qu'une seule fois. Ainsi le type de cet assemblage est similaire au type du composant `AcsSub` : notons toutefois que, contrairement au type de `AcsSub`, la variable ξ est annotée par l'ensemble $\{r\}$, signalant l'utilisation du nom de routage r dans notre assemblage. Les autres

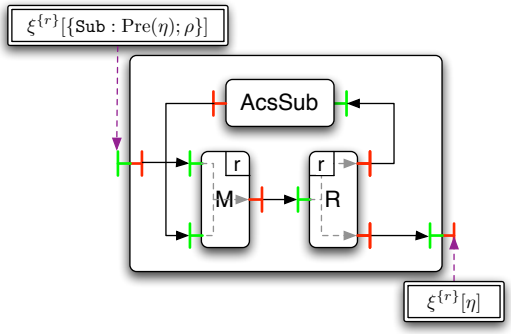


FIGURE 3.37

composants de notre comparatif précédent ne sont pas typables, généralement parce que leurs boucles tentent d'annoter plusieurs fois les messages par $\uparrow r$ ou $\downarrow r$, ce qui n'est pas autorisé dans le calcul. Le seul assemblage ayant une structure de boucle correcte pour le routage n'est en fait pas typable car il applique une infinité de fois le composant `AcsSub` sur les messages présents sur sa boucle. Néanmoins, en changeant ce composant par un autre ne modifiant pas la structure des messages manipulés, comme le composant $+1$ qui rajoute 1 aux entiers reçus, l'assemblage devient typable comme cela est montré Figure

3.38a. En fait, dans cet assemblage, ni le multiplexeur ni le routeur n'ont une part active dans la boucle, et l'assemblage a un comportement très similaire au composant proposé Figure 3.38b (seules l'utilisation du nom de routage r et par conséquent la restriction sur les variables de flux ne sont pas présentes dans le second composant). Nous pouvons donc remarquer que les routeurs et multi-

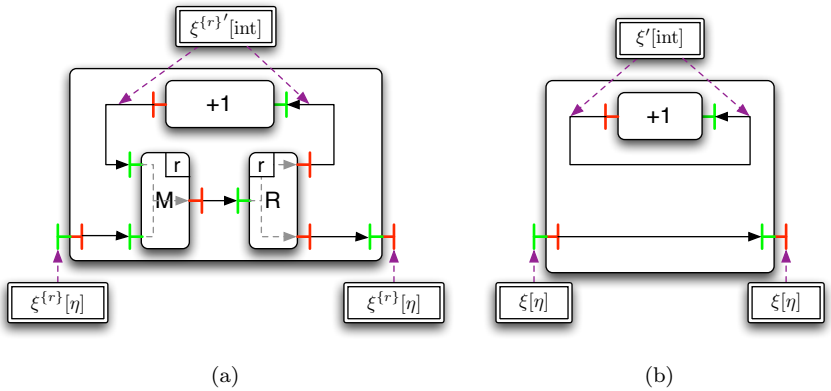


FIGURE 3.38

plexeurs ne sont pas bien acceptés au sein des boucles : soit elles sont construites de façon à produire une annotation de routage erronée, soit ni le routage ni le multiplexage ne prennent part à la boucle.

Les limitations de notre système de type

Le typage des annotations. Notre sixième exemple concerne le typage des messages annotés. En effet, nous avons vu dans les règles de typage que cette opération pouvait être assez pénible. Considérons par exemple le message $M \triangleq 1^{\uparrow r;\downarrow r'}$. Typer ce message consiste à construire un arbre de routage similaire à celui présenté Figure 3.39 où tous les points d'interrogation doivent être remplacés par un terme ne dépendant pas du message. Ces inconnues artificielles (car simplement causées par la structure d'arbre binaire de nos types) peuvent être remplacées dans notre type par une variable de flux fraîche (et non duplicable) et des types routés génériques de la forme $\xi[\eta]$. Un autre défaut de notre structure d'arbre binaire est que l'ordre à l'intérieur des annotations de routage est essentiel lors du typage du message. En effet, parce que $\uparrow r$ précède $\downarrow r'$ dans l'annotation de M, tous les messages transitant sur le même canal devront, pour que l'assemblage soit typable, être annotés soit par $\uparrow r$ soit par $\downarrow r$ sans qu'il n'y ait aucune contrainte concerant le nom de routage r'. Par contre, les messages transitant sur le même canal que $M' \triangleq 1^{\downarrow r';\uparrow r}$ devront, eux, être annotés soit par $\uparrow r'$ soit par $\downarrow r'$. C'est seulement par ce que notre système de

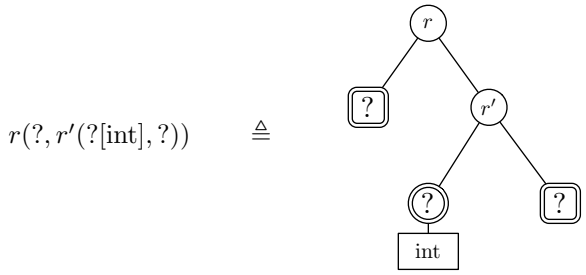


FIGURE 3.39 – Un type routé

type distingue M de M' que les déclarations à l'intérieur de nos annotations de routage ne sont pas permutables. La sémantique des routeurs et multiplexeurs tirerait bien partie d'une telle flexibilité, mais la structure d'arbre binaire de nos types routés nous en empêche. Néanmoins, les assemblages classiques ne comportent que peu de messages avant leur exécution, et aucun généralement ne comporte d'annotation de routage : ce défaut de notre système de type ne se remarque généralement pas en pratique.

La manipulation des annotations. Finalement, nous avons terminé notre partie concernant le calcul par un assemblage consistant en une boucle permettant d'appliquer trois fois le programme D aux messages entrants, non pas en utilisant les routeurs et multiplexeurs, mais d'autres composants manipulant plus finement les annotations des messages. Parmi ces composants, nous avons (i) le filtre qui agit comme du filtrage de motif en envoyant sur des sorties spécifiques des messages portant des annotations validant un certain motif ; (ii) l'ajout d'un nouvel élément dans une annotation ; et (iii) le retrait d'un élément $\uparrow r$ ou $\downarrow r$ de l'annotation des messages donnés en paramètre. Nous rappelons l'assemblage Figure 3.40. De fait, cet assemblage n'est, lui non plus, pas typable, car la structure en arbre binaire de nos types routés n'est pas suffisamment flexible pour typer les manipulations fines opérées sur les annotations. La première difficulté vient des composants de la forme `Sub(`$\uparrow r$`)` ôtant des annotations l'élément $\uparrow r$ et qui ne sont pas typables dans notre système. Ce composant n'accepte en effet en entrée que des messages dont l'annotation contient cet élément : s'il était typable, son type paramètre aurait donc la forme $r(T_1, T_2)$, ce qui impliquerait que le composant devrait aussi pouvoir prendre en compte les messages annotés par $\downarrow r$. Nous pouvons toutefois remplacer ce composant par `Sub(`r`)` qui prend en paramètre n'importe quel message annoté par $\uparrow r$ ou $\downarrow r$ et qui ôte cet élément de l'annotation. Nous pouvons en effet typer ce composant par $e : (r(\xi[\eta], \xi[\eta])) \to s : (\xi[\eta])$. Le second problème est alors le typage du canal central de la boucle où tout les composants de la forme `Add(`$\uparrow r$`)` envoient leurs messages. Ce canal transporte des messages annotés aussi bien

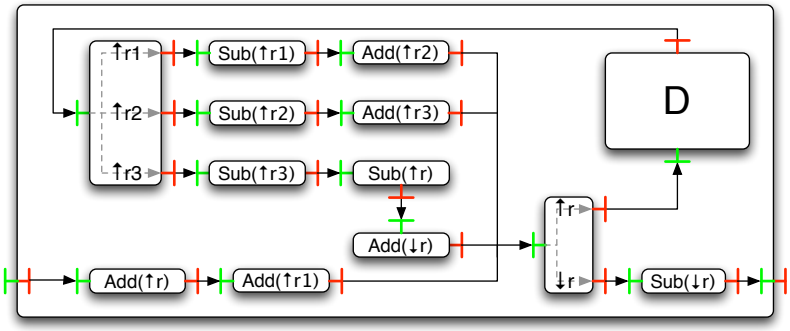


FIGURE 3.40

par $\uparrow r; \uparrow r1$ que par $\uparrow r; \uparrow r2$: il n'est donc pas typable, car la structure d'arbre de nos types routés impose que si un message est annoté par l'élément $\uparrow r1$, alors tous les autres messages passant par le même canal doivent avoir une annotation comportant le nom $r1$. En adaptant la manipulation des filtres et des manipulations faites sur les annotations, nous pouvons remédier au problème, et obtenir un assemblage typable. Néanmoins, l'assemblage résultant n'offre aucun avantage par rapport aux boucles classiques des langages séquentiels, car, toujours à cause de la structure d'arbre de nos types, il est impossible que le message traité puisse prendre un type différent à chaque itération de la boucle.

3.3 Discussion

Comme pour la plupart des systèmes de type, notre système définit une notion d'erreur, et permet de s'assurer qu'un programme typé ne comportera jamais au cours de son exécution une telle erreur. Pour les langages fonctionnels classiques [47] (ainsi que pour notre première approche), la notion d'erreur est construite par rapport à l'application entre une fonction et un mauvais argument. Pour les langages distribués [36, 41, 118], qui ne possèdent généralement pas de notion de fonction et de paramètre erroné, les erreurs sont définies par rapport à des spécifications annexes définissant le type de certains canaux de communication, ou leurs arités dans le cas où ils peuvent transporter plusieurs messages à la fois. Notre cas est assez unique, car bien que nous sommes dans un cadre distribué sans aucune application entre une fonction et un paramètre, nous avons une notion de paramètre valide pour nos composants primitifs, définie par la relation `match`. Cette relation donne pour chaque composant primitif p l'ensemble des motifs (un motif étant un ensemble de messages) que p accepte en entrée : tout message non présent dans ces motifs est donc un paramètre erroné pour le composant. Formellement, nous définissons notre notion d'erreur comme il suit :

Définition 8. *La* capacité entrante *d'un processus D, noté* $\mathtt{InM}(D)$ *est l'en-*

semble des envois de messages défini comme suit :

$$\mathtt{InM}(p) \triangleq \{\overline{e}\langle M\rangle \mid \exists J, \mathtt{match}(p, \overline{e}\langle M\rangle \mid J)\}$$

$$\mathtt{InM}(c[I\,/\,O][D]) \triangleq \bigcup_{e\in I}\{\overline{e}\langle M\rangle \mid \textit{For any } M\} \qquad \mathtt{InM}(D_1 \mid D_2) \triangleq \mathtt{Inc}(D_1)\cup\mathtt{Inc}(D_2)$$

Un processus D a une erreur *si et seulement si il existe un envoi de message $\overline{e}\langle M\rangle$, un processus D' et un contexte d'évaluation E tel que $D = E[\overline{e}\langle M\rangle \mid D']$ et $\overline{e}\langle M\rangle$ n'est pas un argument valide de D',* i.e. $\overline{e}\langle M\rangle \notin \mathtt{InM}(D')$.

Nous pouvons remarquer que cette définition permet au composant composite d'accepter n'importe quel message envoyé sur une de ses portes entrantes : seuls les composants primitifs spécifient explicitement quels sont leurs paramètres valides. Enfin, avec cette définition, nous pouvons donner les propriétés élémentaires de notre système de type :

Théorème 3 (Correction)**.** *Supposons donnés un assemblage D et un type de processus F tel que le jugement de type $D : F$ est valide. Alors D ne contient aucune erreur.*

Théorème 4 (Stabilité)**.** *Supposons donnés un jugement de type valide $D : \forall\overline{\alpha}.S_1 \to S_2$. Alors, pour tout assemblage D' tel que $D \rhd D'$, il existe S_2' avec $S_2' \subset S_2$ tel que le jugement de typage $D' : \forall\overline{\alpha}.S_1 \to S_2'$ est valide.*

Les preuves de ces deux théorèmes sont données dans l'annexe B.

Comparativement à nos premiers travaux, et même à beaucoup de calculs de processus existants, cette approche est assez innovante, aussi bien dans son calcul que dans son système de type. Au niveau syntaxique, nous avons un calcul extrêmement simple à base de composants primitifs et de messages s'abstrayant de toutes les opérations concernant la réception, l'envoi et la manipulation des messages. L'avantage principal de cette syntaxe, en plus de sa simplicité, est qu'elle est très proche de beaucoup d'ADL (*Architecture Description Language*) [17, 42, 80, 6, etc] qui omettent de décrire le comportement interne des composants [11]. Il est donc assez facile d'adapter notre calcul à ces langages afin d'y inclure sans beaucoup d'effort notre système de type. Par contre, il n'est pas toujours pratique de définir des exemples avec notre syntaxe, car dès que nous voulons avoir un comportement nouveau, nous devons étendre les relations `match` et `eval` au lieu de spécifier clairement ce comportement à l'aide d'un processus. Toutefois, dans le cadre de l'application de notre système à des assemblages réels définis par des ADLs, notre calcul est bien plus adapté que

11. Celui-ci est en effet spécifié soit dans des fichiers sources, Java ou C++, soit par une description formelle en vue de vérifier la validité de l'assemblage

notre première approche où le comportement des composants, défini dans un langage particulier, devait être traduit dans notre syntaxe.

Le routage

Dans le calcul. Notre routage *sémantique* à base d'annotation sur les messages est lui aussi une innovation majeure de notre approche. En effet, les seuls travaux existants s'apparentant à cette construction sont les variants de ML [92] : nous pouvons en effet considérer les étiquettes des variants comme des annotations sur des valeurs, et le filtrage de motifs comme une procédure de routage. Néanmoins, ces variants sont limités par rapport à nos annotations, car ils sont contraints à ne porter au plus qu'une seule annotation [12] et un filtrage de motif n'est valide que si chacune de ses branches a le même type. De fait, la motivation originale de la création de ces annotations était d'avoir un routage et un multiplexage dual. C'est cette recherche qui a introduit les noms identifiant les paires de routeur/multiplexeur, et donc les annotations de routage et a créé en conséquence ce que nous appelons le routage sémantique. Il est apparu dans notre étude sur le calcul que cette méthode était généralement plus adaptée que le routage structurel de notre première version, en partie car il est plus pratique à utiliser [13]. En effet, il suffit d'annoter deux messages de façons différentes pour qu'il puissent être distinguables, alors que pour avoir le même résultat dans notre première approche, il fallait que ces messages aient une structure différente. Il est des cas où le routage structurel est néanmoins intéressant, par exemple lorsqu'un composant ne prend en paramètre que des messages ayant un champ particulier et qu'il faille donc distinguer les messages que l'on peut envoyer à ce composant de ceux qui doivent être traités différemment. Nous avons vu qu'on pouvait décrire cette procédure dans notre calcul, mais le passage aux types est assez difficile. Ici, nos types encodent la structure des messages, et donc des messages de structures différentes doivent, pour être placés sur un même canal, porter des annotations différentes : le routage structurel est inclus dans le routage sémantique. Dans le cas où la structure des messages n'est pas décrite dans les types, alors les opérations spécifiques à chaque structure doivent être intégrées dans le routeur, qui ne peut avoir des types résultats de structures différentes. Nous avons finalement un cas intermédiaire, où la structure des messages est décrite dans les types, mais soumise à du sous-typage comme dans [19]. Dans ce cas, il n'est pas possible de typer la concaténation de messages [112, 91], mais il devient possible de placer deux messages différents sur un même canal avec une même annotation et de spécifier en sortie du routeur deux types résultats différents. Néanmoins, cette utilisation du sous-typage a le désavantage de perdre toutes les informations concernant la structure des mes-

12. Dans certaines extensions [89], il est possible d'avoir des étiquettes sous la forme d'ensemble de noms – que l'on pourrait considérer comme un ensemble d'annotations – mais il reste impossible de typer les fonctions de rajout ou de délétion de noms telles que nos composants `Add` et `Sub`

13. Il n'est toutefois pas plus expressif car, comme nous l'avons vu dans nos exemples, la plupart des boucles construites avec notre première approche ne sont pas exprimables avec notre routage sémantique

sages, et c'est pourquoi elle n'est généralement pas employée. Ainsi, il apparaît que ces deux formes de routage ont du mal à cohabiter.

Dans le système de type. La prise en compte du routage dans notre système de type est, de manière globale, assez positive. Le typage des chaînes de composants (sans boucle) est assez intuitive, et la manipulation des multiplexeurs et routeurs y est assez naturelle. La seule contrainte importante que notre routage sémantique apporte par rapport à notre première approche concerne le typage des boucles. En effet, nous avons vu dans nos exemples qu'il était impossible de définir des boucles construites sur une procédure de routage. Une des raisons de cette contrainte, mais pas la principale, réside dans la forme de nos types routés : leurs structures en arbre binaire restreint grandement leur capacité d'expression. La raison principale vient du fait que nos types associent à une annotation un type de message unique. Cette limitation n'est pas présente dans notre première approche, où n'importe quel message peut être sur n'importe quel canal, tant que cela ne cause pas d'erreur d'exécution. Néanmoins, notre limitation a une contrepartie : notre système de type possède un algorithme d'inférence (voir Chapitre 4). La contrainte que l'on impose sur les boucles du calcul est alors assez similaire à la contrainte sur les définitions récursives de ML : la récursion polymorphe n'est pas autorisée.

L'isolation

Dans le calcul. Notre calcul est aussi un outil d'étude pour la communication distante, et en particulier, sur les portes comme moyen pour accéder à ce genre de communication. À notre connaissance, peu de travaux se sont intéressés à cette problématique. La plupart des calculs [23, 98, 26] encodent la communication entre des parties éloignées d'un même programme en utilisant des programmes qui transmettent les messages de composants en composants jusqu'à leurs destinations finales. Les quelques exceptions [52, 39] soit (i) utilisent des références sur des composants distants afin de simuler leur présence localement ; ou (ii) mettent à plat la hiérarchie de composants afin que les communications distantes correspondent à des communications locales. Le système de porte, aussi utilisé dans [67] qui sera décrit dans le Chapitre 5, est donc le seul moyen à notre connaissance pour avoir une communication distante sans avoir à modifier la structuration en arbre des programmes à base de composants. Néanmoins, bien que ce système de porte soit relativement simple et élégant, il comporte quelques défauts importants. Tout d'abord, notre approche consiste en une communication asynchrone, ce qui simplifie les règles de communication : dans un cadre synchrone (voir le Chapitre 5) il n'est pas possible de faire avancer un message pas à pas vers sa destination, et les règles de réduction sont donc beaucoup plus complexes. L'autre défaut principal du système de porte est qu'il est impossible de spécifier précisément la destination d'un message : on peut seulement spécifier sur quel canal il transite. C'est ensuite au fur et à mesure de son cheminement dans l'assemblage qu'il pourra être consommé par un composant écoutant sur ce canal, si un tel composant se trouve sur son

chemin. Comme il peut y avoir plusieurs chemins pour un même canal, il peut y avoir plusieurs composants pouvant prendre ce message en argument : il n'existe aucun moyen pour le composant émetteur de spécifier vers quelle possible destination le message doit se diriger.

Dans le système de type. Néanmoins, ce genre de considérations, même si intéressantes et importantes dans le cadre de la définition d'un langage distribué, ne sont pas les plus importantes dans ce travail : nous nous concentrons principalement sur le typage, et plus particulièrement, à améliorer la prise en compte de l'isolation et du routage par rapport à notre première approche. De ce point de vue, les portes nous ont donné une construction du calcul permettant la mise en place simple de l'isolation au niveau des types. En fait, le principe des portes est d'afficher sur la frontière d'un composant ses capacités entrantes et sortantes. C'est grâce à cette connaissance que l'on peut restreindre les types et ainsi cacher au reste d'un programme le comportement interne d'un composant. Notons que dans le kell-calcul ainsi que dans notre première approche, l'isolation dépend du père de chaque composant, et plus précisément, des canaux qu'il peut utiliser pour communiquer avec ses fils : en rajoutant aux règles de typage de notre approche précédente un contexte donnant les canaux utilisés par le père de chaque composant, il est possible d'y inclure de l'isolation. Ainsi, bien que les portes aient quelques défauts dans leur sémantique d'échange de messages, leur étude nous a apporté quelques lumières sur comment intégrer de l'isolation dans un système de type pour un calcul distribué.

3.4 Conclusion

Dans ce chapitre, nous avons introduit un calcul novateur impliquant un système simple de composants et de portes permettant la communication entre plusieurs entités distantes. De plus, nous avons introduit un concept d'annotation permettant une nouvelle forme de routage et de multiplexage : le routage *sémantique*. Le système de type que nous avons défini sur ce calcul est lui-même assez complet, en prenant en compte à la fois la nouvelle forme de routage, l'isolation entre les composants impliquée par les portes et du sous-typage. Ce travail comporte quelques défauts, et n'est donc pas applicable à la définition d'un réel langage de programmation, mais nous trouvons qu'il est toutefois très intéressant et nous donne de nombreuses pistes pour des améliorations futures. Le défaut principal de notre calcul sont les portes : trouver un moyen d'avoir de l'isolation tout en permettant la communication à distance est un problème difficile, et notre système de porte n'est pas satisfaisant. Nous envisageons de remplacer ces éléments par des constructions proches des sessions [111] qui joueraient le même rôle que les interfaces dans les ADLs, et qui permettraient alors de connaître précisément son interlocuteur. L'autre défaut de notre approche se trouve dans notre système de type : la structure d'arbre binaire de nos types routés est extrêmement contraignante. Néanmoins, ce problème est proche d'être résolu, car comme nous l'avons fait remarqué, nos annotations sont très simi-

laires aux variants à la ML : nous avons donc généralisé – dans un travail encore non publié – le travail fait dans [89] pour pouvoir y exprimer les opérations de rajout et de retrait d'annotation. Il ne nous reste donc plus qu'a intégrer ce travail à notre calcul pour avoir un routage sémantique totalement fonctionnel et typable. Concernant l'ajout de primitives pour la mobilité de code et leur typage, nous n'avons pas encore eu de résultat satisfaisant (voir le Chapitre 5), mais beaucoup de pistes de recherche restent ouvertes, et nous considérons les types de sessions comme des travaux intéressants pour encoder la modification de la structure d'un programme durant son exécution. Finalement, nous concluons ce chapitre par un résultat important : nous système de type admet un algorithme d'inférence qui est implémenté sur deux ADLs existant : Dream [65] et Click [80]. Cet algorithme ainsi que son implémentation sont présentés dans le chapitre suivant.

Chapitre 4

Inférence et implémentation

Ce chapitre présente tout d'abord notre algorithme d'inférence de type, basé sur la génération et la résolution de contraintes, suivi par l'implantation de cet algorithme sur deux modèles, Fractal (ou plus précisement, Julia qui est son implantation en Java) et Click.

4.1 L'inférence de type

L'inférence de type [58, 102] est la tâche consitant à calculer automatiquement si un programme est typable ou non. Dans notre cas, nous parlons d'inférence *totale* car cet algorithme n'utilise aucune annotation de type de la part du programmeur pour l'aider dans son calcul. L'algorithme que nous présentons dans cette partie est basé sur des *contraintes* [83, 89] construites sur des types de messages, des types routés et des types de processus : à partir d'un programme en entrée, il calcule tout d'abord un ensemble de contraintes qu'un type doit satisfaire pour être valide pour le programme. Un solveur de contraintes est ensuite appliqué sur l'ensemble calculé, ce qui permet de savoir s'il est satisfiable, ce qui signifierait que le programme est bien typable. Ce solveur est inspiré de [93] pour gérer les types de messages, et est étendu aux types routés et aux types ensemble, ce qui forme la nouveauté de cet algorithme d'inférence par rapport à ceux existant, en particulier nous avons des règles de résolutions spécifiquement pour gérer la *duplication* des types routés.

Notre présentation est divisée en quatre parties : (i) nous introduisons tout d'abord la syntaxe de nos contraintes, ainsi que leur sémantique ; (ii) nous décrivons ensuite l'algorithme qui calcule l'ensemble de contrainte ; (iii) vient ensuite la présentation de notre solveur, juste avant (iv) les différentes propriétés que vérifient ces algorithmes, et qui rendent l'inférence de type suffisamment abordable pour être utilisée en pratique.

4.1.1 La syntaxe des contraintes

La grammaire présentant la syntaxe de nos contraintes est décrite Figure 4.1. Informellement, nous avons trois types de contraintes : (i) nous avons des contraintes de sous-typage simple entre des types ensemble ($S_1 \precsim S_2$) et entre des types routés ($T_1 \leq T_2$) ; (ii) nous avons des contraintes *locales* à une variables de routage pour gérer la localités des variables de types élémentaires et des types rangés ; (iii) des contraintes entre des *formes* pour manipuler correctement la duplication des variables de routage. La structure de base des contraintes est

C	$::=$		Une contrainte
		$\mathbf{true} \mid \mathbf{false} \mid S_1 \precsim S_2 \mid T_1^R \leq T_2^R$	Contrainte simple
	$\mid$	$[\xi^R : \tau_1 \leq \tau_2] \mid \exists\alpha.C \mid C_1 \wedge C_2$	Autre contrainte
	$\mid$	$\xi = \xi \mid \xi = r(\xi_1, \xi_2)$	Contrainte de duplication

FIGURE 4.1 – La syntaxe des contraintes

donnée par **true**, $C_1 \wedge C_2$ qui font de ces contraintes des ensembles de contraintes de base : **true** correspondant à l'ensemble vide, $C_1 \wedge C_2$ correspondant à l'union de deux ensemble, et les contraintes de base étant comme des singletons. Nous pouvons aussi considérer la contrainte **false** comme l'ensemble contenant toute les contraintes, et donc qui n'est par définition pas satisfiable. Les deux premières contraintes de base sont des contraintes de sous-typage : $S_1 \precsim S_2$ signifiant que S_1 doit être partiellement inclus dans S_2, et $T_1 \leq T_2$ signifiant que T_2 doit être supérieur, dans la relation de sous-typage, à T_1. $[\xi : E_1 \leq E_2]$ et $[\xi : W_1 \leq W_2]$ sont des contraintes de sous-typage sur les types élémentaires et types rangés, qui ne s'appliquent que localement à la variable de routage ξ. Cela permet de gérer sans erreur la localité des variables. Les variables non locales sont aussi placées dans des contraintes locales, mais sont manipulées différemment. Enfin, nous avons deux sortes de contraintes de duplication, $\xi = \xi'$ et $\xi = r(\xi_1, \xi_2)$. Ces contraintes servent à vérifier que l'on ne duplique pas de variable de routage non-duplicable, et pour ce faire, utilisent des contraintes d'égalité. En effet, utiliser seulement les contraintes de sous-typage sur les types routés n'est pas suffisant pour s'assurer que les kinds des variables de routage sont bien respectées. Considérons par exemple la contrainte $r(T_1, T_2) \leq T \wedge r'(T_1', T_2') \leq T$: comme il n'y a pas de relation de sous-typage entre les deux arbres $r(T_1, T_2)$ et $r'(T_1', T_2')$, on ne peut exprimer, sans nos contraintes spécifiques, le fait que le premier doit être duplicable par r' et que le second duplicable par r. Nous utilisons donc des contraintes d'égalité entre des formes (nous n'avons ainsi pas

de référence aux types élémentaires qui demanderaient du sous-typage) donnant quand deux variables sont égales, ou quand une variable doit être dupliquée.

Leur équivalence structurelle. Elle est définie comme la plus petite relation transitive et réflexive vérifiant les règles Figure 4.2, et présente simplement que les contraintes sont vues, comme dit précédemment, comme des ensembles de contraintes de base.

$$\mathbf{true} \wedge C \equiv C \qquad C_1 \wedge (C_2 \wedge C_3) \equiv (C_2 \wedge C_1) \wedge C_3 \qquad \frac{\alpha \notin fv(C')}{\exists \alpha.C \wedge C' \equiv \exists \alpha.(C \wedge C')}$$

FIGURE 4.2 – L'équivalence structurelle des contraintes

Leur sémantique. Elle est définie pour chaque contrainte C comme étant l'ensemble des substitutions *validant* cette contrainte. Ce paragraphe définit le prédicat $\vDash$ entre une substitution σ et une contrainte C, tel que $\sigma \vDash C$ est valide si et seulement si σ valide C.

Définition 9. *Nous définissons inductivement Figure 4.3 le prédicat* $\vDash$ *donnant lorsqu'une substitution valide une contrainte. De plus, nous notons* $C_1 \vDash C_2$ *lorsque toutes les substitutions validant* C_1 *valident aussi* C_2*, et nous notons* $C_1 \equiv C_2$ *lorsque ces deux contraintes ont la même sémantique, c.a.d. toutes les substitutions validant* C_1 *valident* C_2*, et réciproquement.*

4.1.2 La génération des contraintes

Cette partie de l'inférence est présentée Figures 4.4 et 4.5, qui donnent respectivement les règles de génération de contraintes pour les messages routés et pour les processus. Informellement, cet algorithme prend en entrée un programme, et essaye de trouver son type. Quand cette recherche demande un certain calcul, comme par exemple trouver une substitution unifiant deux types, l'algorithme génère en plus du type assigné au programme, une contrainte correspondant au calcul demandé. Le résultat de la génération de contraintes est donc un type assigné au programme en paramètre, et un ensemble de contraintes que ce type doit vérifier pour être un type correct du programme. Le solveur, présenté en troisième partie, est le processus manipulant les contraintes, et calculant si les contraintes générée sont satisfiables ou non.

$$\sigma \vDash \textbf{true} \qquad \frac{\sigma(S_1) \precsim \sigma(S_2)}{\sigma \vDash S_1 \precsim S_2} \qquad \frac{\sigma(T_1) \leq \sigma(T_2)}{\sigma \vDash T_1 \leq T_2} \qquad \frac{\sigma(\xi[E_1]) \leq \sigma(\xi[E_2])}{\sigma \vDash [\xi : E_1 \leq E_2]}$$

$$\frac{l = \{a_1, \ldots, a_n\} \quad \sigma(\xi[\{a_1 : \text{Abs}; \ldots ; a_n : \text{Abs}; W_1^l\}]) \leq \sigma(\xi[\{a_1 : \text{Abs}; \ldots ; a_n : \text{Abs}; W_2^l\}])}{\sigma \vDash [\xi : W_1^l \leq W_2^l]}$$

$$\frac{\sigma[\alpha \to \tau] \vDash C}{\sigma \vDash \exists \alpha . C} \qquad \frac{\sigma \vDash C_1 \qquad \sigma \vDash C_2}{\sigma \vDash C_1 \wedge C_2} \qquad \frac{\sigma(\xi[\bullet]) = \sigma(\xi'[\bullet])}{\sigma \vDash \xi = \xi'}$$

$$\frac{\sigma(\xi[\bullet]) = \sigma(r(\xi_1[\bullet], \xi_2[\bullet]))}{\sigma \vDash \xi = r(\xi_1, \xi_2)}$$

FIGURE 4.3 – La validation des contraintes

Dans les règles présentées ici, nous utilisons le terme fresh pour introduire dans les contraintes et les types construits des variables qui ne sont pas déjà utilisées par notre algorithme. Ces variables fraîches sont généralement utilisées avec les mêmes kinds, qui sont $\mathfrak{D}$ pour les variables de routages, et $\mathfrak{L}$ pour les types élémentaires et les types rangés. Ainsi, nous ne précisons la kind des variables fraîches que lorsqu'elle est différente de ce que nous avons présenté.

Les messages routés. Comme les messages ne contiennent pas d'opérateurs (dans le sens qu'il n'y a pas d'application dans notre calcul), la génération de contraintes pour ces éléments du calcul ne demande pas de définition de contraintes. Ainsi, cet algorithme appliqué aux messages (resp. messages routés) prend la forme $v : E$ (resp. $\delta \vdash M : T$ où δ est une annotation de routage). Les règles de génération de contraintes sont alors similaires que les règles de typage, la seule différence étant que l'on donne des variables fraîches pour les parties inconnues de l'arbre de routage d'un message routé.

Les processus. La génération de contraintes pour les processus prend la forme $D : F\,[C]$ où D est le programme en paramètre, F est le type que l'on lui assigne, et C est la contrainte calculée. La façon dont nous gérons le sous-typage est assez proche de ce qui est fait dans [88]. En effet, pour chaque composant primitif et message envoyé dans le programme, nous introduisons des variables fraîches qui typent ces éléments du programme. Ces variables sont ensuite contraintes par une relation de sous-typage avec le type donné du composant ou précédemment calculé du message envoyé. Considérons par exemple la règle de génération (G :ENVOI). Le message routé a déjà un type T calculé par les règles

$$\text{G :Constante}\quad c : \Psi(c)$$

$$\text{G :Message}\quad \frac{\forall 1 \leq i \leq n,\ v_i : E_i \qquad \forall 1 \leq i \neq j \leq n,\ a_i \neq a_j}{\{a_1 = v_1; \ldots; a_n = v_n\} : \{a_1 : \mathrm{Pre}(E_1); \ldots; a_n : \mathrm{Pre}(E_n); \mathrm{Abs}\}}$$

$$\text{G :Vide}\quad \frac{v : E \qquad \xi^R \text{ fresh} \wedge \mathtt{kind}(\xi^R) = \mathfrak{F}}{R \vdash v^{\emptyset} : \xi^R[E]}$$

$$\text{G :Droit}\quad \frac{R \uplus \{r\} \vdash v^{\delta} : T \qquad \eta, \xi^R \text{ fresh}}{\delta \vdash v^{\uparrow r;\delta} : r(T, \xi^R[\eta])}$$

$$\text{G :Gauche}\quad \frac{R \uplus \{r\} \vdash v^{\delta} : T \qquad \eta, \xi^R \text{ fresh}}{R \vdash v^{\downarrow r;\delta} : r(\xi^R[\eta], T)}$$

Figure 4.4 – La génération de contraintes : messages routés

de génération pour les messages, et pourtant $\overline{e}\langle M\rangle$ est typé par $\emptyset \to e : (\xi[\eta])$. De cette façon, avec la contrainte générée, nous pouvons instancier par la suite ces variables en n'importe quel sur-type de T : les règles de sous-typage du système de type sont ainsi directement intégrées dans cette règle de génération et dans (G :Primitif), qui est formée de la même façon. En effet, dans cette règle, nous remplaçons le type du composant primitif p par un type générique [1] $S'_1 \to S'_2$ que nous contraignons pour ne pouvoir être instancié que par un sur-type de p. Comme le sous-typage est concentré dans ces deux règles, nous avons pu simplifier la règle manipulant les composants, qui, contrairement à la règle de typage, définit totalement le type d'un composant en fonction du type de son processus interne : nous exposons seulement les interfaces d'entrées et de sortie dans le type. Finalement, la règle pour la composition parallèle correspond directement à la règle de typage, où les prérequis sur les types sont simplement transformés en contraintes.

4.1.3 La résolution de contrainte

Cette étape est la seconde et dernière partie de notre algorithme d'inférence. Suivant l'approche usuelle, elle consiste en un ensemble de règles de fermetures ajoutant à un un ensemble de contraintes de nouvelles jusqu'à stabilité de l'ensemble. Typiquement, soit nous obtenons durant cet ajout la contrainte **false**, auquel cas la contrainte n'est pas satisfiable, et le processus n'est pas

1. ce type ne contient en effet que des types de message routé de la forme $\xi[\eta]$ avec ξ et η frais.

G :Nul

$$\emptyset \,:\, \emptyset \to \emptyset\,[\mathbf{true}]$$

G :Envoi

$$\frac{\emptyset \vdash M \,:\, T \qquad \eta, \xi \text{ fresh}}{\overline{e}\langle M\rangle \,:\, \emptyset \to e : (\xi[\eta])\,[T \leq \xi[\eta]]}$$

G :Primitif

$$\frac{\begin{array}{c} p \,:\, \forall\overline{\alpha}.S_1 \to S_2 \qquad S_1 = \bigcup e_i : (T_i) \qquad S_2 = \bigcup e'_j : (T'_j) \\ \overline{\alpha}' \text{ fresh} \qquad \xi_i, \xi'_j, \eta_i, \eta'_j \text{ fresh} \qquad S'_1 \triangleq \bigcup e_i : (\xi_i[\eta_i]) \qquad S'_2 \triangleq \bigcup e'_j : (\xi'_j[\eta'_j]) \end{array}}{p \,:\, S'_1 \to S'_2\,[S_2\{^{\overline{\alpha}'}/_{\overline{\alpha}}\} \precsim S'_2 \wedge S'_1 \precsim S_1\{^{\overline{\alpha}'}/_{\overline{\alpha}}\}]}$$

G :Boîte

$$\frac{D' \,:\, S_1 \to S_2\,[C]}{c[I\,/\,O][D'] \,:\, (S_1 \cap I) \to (S_2 \cap O)\,[C]}$$

G :Parallèle

$$\frac{D \,:\, S_1 \to S_2\,[C] \qquad D' \,:\, S'_1 \to S'_2\,[C'] \qquad dc(S_1) \cap dc(S'_1) = \emptyset}{D \mid D' \,:\, (S_1 \cup S'_1) \to (S_2 \cup S'_2)\,[C \wedge C' \wedge S'_2 \precsim S_1 \wedge S_2 \precsim S'_1]}$$

Figure 4.5 – La génération des contraintes : processus

typable, soit nous obtenons un ensemble de contraintes de base stable par rapport aux règles de fermeture, auquel cas l'ensemble est satisfiable, le programme typable, et nous pouvons calculer une substitution rendant le type assigné au programme valide pour ce programme. Nous avons dans cette partie un grand nombre de règles, que nous structurons en quatre ensembles distincts : (i) simplification des contraintes $S_1 \precsim S_2$; (ii) simplification des contraintes sur les types routés $T_1 \leq T_2$; (iii) simplification des contraintes locales $[\xi : E_1 \leq E_2]$ et $[\xi : W_1 \leq W_2]$; et finalement (iv) propagation des contraintes locales, et manipulation des contraintes de duplication. Chacun de ces ensembles est présenté et expliqué séparément, afin de faciliter la compréhension du fonctionnement de l'algorithme de résolution. Notons néanmoins que ces règles s'appliquent sans contrainte d'ordonnancement durant l'exécution de l'algorithme. Finalement, une étape de résolution est notée $C \blacktriangleright C'$, ce qui signifie que si une contrainte contient C, alors nous lui ajoutons la contrainte C'.

Les types ensembles. Une des étapes importantes de la résolution est d'extraire des contraintes sur les types ensembles des contraintes plus simple sur les types routés contenu dans ces derniers. Le règle de fermeture correspondant à cette extraction est présentée Figure 4.6, et correspond directement à la définition du prédicat $\precsim$ entre deux types ensemble.

$$e : (T) \cup S_1 \precsim e : (T') \cup S_2 \blacktriangleright T \leq T'$$

FIGURE 4.6 – Simplification des contraintes $S_1 \precsim S_2$

Les types routés. Le principe de la résolution des contraintes sur les types routés (présenté Figure 4.7) est d'explorer leur structure d'arbre afin de construire les contraintes de duplication correspondant à cette structure, et les contraintes locales correspondant aux différentes spécifications que ces arbres ont sur leur feuilles. Ces règles d'exploration sont présentée 4.7. La première est relativement directe, et explore deux arbres de même racine. La seconde nécessite plus de travail, les deux arbres n'ayant pas la même racine. La façon dont elle gère la différence de structure des deux arbres s'inspire de l'inférence pour les types rangés [104] : considérons que ces deux arbres sont unifiables en un même arbre T. Comme cet arbre contient $r(T_1, T_2)$ et $r'(T_3, T_4)$, il a une structure similaire à $r(r'(T'_1, T'_2), r'(T'_3, T'_4))$. Nous introduisons ainsi dans la règles les quatre sous-arbres T'_i sous forme d'arbres génériques afin qu'il soient instanciés durant le reste de la résolution. La règle finalement donne les contraintes définissant T en fonction des deux arbres dans la contraintes en entrée, et plus précisément, celles qui lient les sous-arbres T'_i aux sous-arbres T_i. Une des particularités de cette règle est qu'elle introduit des variables fraîches, et peut ainsi est appliquée indéfiniment. Pour éviter ce phénomène, nous identifions les contraintes modulo α-conversion afin que cette règle, et toutes celles introduisant de nouvelles variables, ne puissent être appliquées qu'une seule fois avec le même paramètre. La troisième règle présente l'instanciation d'une feuille de type routé en un arbre plus complexe. Cette règle utilise la notation $\asymp$ signifiant aussi bien $\leq$ ou $\geq$, afin de limiter la duplication des règles de résolution. Finalement, la dernière règle gère l'unification entre deux feuilles, en créant une contrainte d'égalité entre les deux variables de routage, et une contrainte locale manipulant les deux types élémentaires.

La simplification des contraintes locales. Les règles de résolution des contraintes locales manipulant les types élémentaires et les types rangés sont présentées 4.8. Excepté la référence aux variables de routages, elles sont similaires aux règles classiques de résolution de contraintes sur les types de base. Les deux premières vérifient que les contraintes de sous-typage sur les constructeurs de types sont bien valides. Les deux suivantes permettent de passer des types élémentaires aux types rangés. Celles qui suivent explorent des types rangés de

$$r(T_1, T_2) \leq r({T_1}', {T_2}') \blacktriangleright T_1 \leq {T_1}' \wedge T_2 \leq {T_2}'$$

$$\frac{r \neq r' \qquad R' \triangleq R \uplus \{r, r'\} \qquad \forall 1 \leq i \leq 4,\ (\eta_i, \xi_i^{R'}) \text{ fresh} \wedge T'_i \triangleq \xi_i^{R'}[\eta_i])}{\begin{array}{c} r(T_1^{R\uplus\{r\}}, T_2^{R\uplus\{r\}}) \leq r'(T_3^{R\uplus\{r\}}, T_4^{R\uplus\{r\}}) \\ \blacktriangleright \exists(\eta_i, \xi_i^{R'}).\left(\begin{array}{l} \wedge T_1^{R\cup\{r\}} \leq r'(T'_1, T'_2) \wedge T_2^{R\cup\{r\}} \leq r'(T'_3, T'_4) \\ \quad r(T'_1, T'_3) \leq T_3^{R\cup\{r'\}} \wedge r(T'_2, T'_4) \leq T_4^{R\cup\{r'\}} \end{array}\right) \end{array}}$$

$$\frac{\begin{array}{c} \forall 1 \leq i \leq 2,\ \xi_i^{R\uplus\{r\}} \text{ fresh} \\ \xi^R[E] \succeq r(T_1, T_2) \blacktriangleright \exists \xi_1, \xi_2. \bigwedge(\xi_i[E] \succeq T_i) \wedge \xi^R = r(\xi_1, \xi_2) \end{array}}{T_1 \leq \xi[E] \wedge \xi[E] \leq T_2 \blacktriangleright T_1 \leq T_2}$$

FIGURE 4.7 – Simplification des contraintes $T_1 \leq T_2$

même structure, alors que les quatre suivantes manipulent les types rangés de structure différente, et donne en résultat la contrainte **false** lorsque l'unification des structures est impossible. Finalement, la dernière règle introduit dans les contraintes la transitivité du sous-typage.

Simplification des contraintes de duplication et propagation. Ce paragraphe présente le dernier ensemble de règles de fermeture de contraintes que nous avons dans cet algorithme de résolution. La gestion des contraintes de duplication est similaire à celle des contraintes de sous-typage entre les types routés. Les deux différences, comme notées précédemment, sont l'absence de type élémentaire dans les feuilles (les contraintes d'égalité entre deux feuilles ne génère donc pas de contrainte locale) et une transitivité plus forte, permettant de tester l'égalité d'arbre de routage qui ne sont pas en relation de sous-typage (mais qui doivent tout de même avoir la même forme). Les autre règles de ce paragraphe décrivent ce que nous appelons la *propagation*, et consiste, sous certaines conditions, à prendre une contrainte locale, et à l'appliquer à une autre variable de routage. Ces règles de propagation sont par exemple utilisées pour les contraintes impliquant des variables globales, ou lorsqu'une variable de routage est unifiée à un autre arbre de routage. Considérons par exemple l'assemblage décrit Figure 4.9. Cet assemblage propose un composant `Transmit_Int` prenant en entrée des entiers, et les transmettant en sortie. Ce composant est connecté à un `Conduit`, qui renvoi les messages reçus à un routeur `Router`. Comme les messages reçus par `Transmit_Int` peuvent avoir été déjà traités par un multiplexeur, cet assemblage est correct et typable. Du point de vue de l'inférence,

$$\frac{s \leq s'}{[\xi : s(E_1, \ldots, E_n) \leq s'(E'_1, \ldots, E'_n)] \blacktriangleright \bigwedge([\xi : E_i \leq E'_i])}$$

$$\frac{s \not\leq s'}{[\xi : s(E_1, \ldots, E_n) \leq s'(E'_1, \ldots, E'_n)] \blacktriangleright \textbf{false}}$$

$$[\xi : s(E_1, \ldots, E_n) \not\geq \{W\}] \blacktriangleright \textbf{false} \qquad [\xi : \{W_1\} \leq \{W_2\}] \blacktriangleright [\xi : W_1 \leq W_2]$$

$$[\xi : a : \mathrm{Pre}(E_1); W_1 \leq a : \mathrm{Pre}(E_2); W_2] \blacktriangleright [\xi : W_1 \leq W_2] \wedge [\xi : E_1 \leq E_2]$$

$$[\xi : a : \mathrm{Abs}; W_1 \leq a : \mathrm{Abs}; W_2] \blacktriangleright [\xi : W_1 \leq W_2]$$

$$\frac{\rho_1^{l \uplus \{a,b\}}, \rho_2^{l \uplus \{a,b\}} \text{ fresh}}{\begin{array}{l}[\xi : a : \mathrm{Pre}(E_1); W_1^{l \uplus \{a\}} \leq b : \mathrm{Pre}(E_2); W_2^{l \uplus \{b\}}] \\ \blacktriangleright \exists \rho^{l \uplus \{a,b\}}. \left(\wedge \begin{array}{l}[\xi : a : \mathrm{Pre}(E_1); \rho^{l \uplus \{a,b\}} \leq W_2^{l \uplus \{b\}}] \\ [\xi : W_1^{l \uplus \{a\}} \leq b : \mathrm{Pre}(E_2); \rho^{l \uplus \{a,b\}}]\end{array} \right)\end{array}}$$

$$\frac{\rho_1^{l \uplus \{a,b\}}, \rho_2^{l \uplus \{a,b\}} \text{ fresh}}{\begin{array}{l}[\xi : a : \mathrm{Abs}; W_1^{l \uplus \{a\}} \leq b : \mathrm{Pre}(E_2); W_2^{l \uplus \{b\}}] \\ \blacktriangleright \exists \rho^{l \uplus \{a,b\}}. \left(\wedge \begin{array}{l}[\xi : a : \mathrm{Abs}; \rho^{l \uplus \{a,b\}} \leq W_2^{l \uplus \{b\}}] \\ [\xi : W_1^{l \uplus \{a\}} \leq b : \mathrm{Pre}(E_2); \rho^{l \uplus \{a,b\}}]\end{array} \right)\end{array}}$$

$$\frac{\rho_1^{l \uplus \{a,b\}}, \rho_2^{l \uplus \{a,b\}} \text{ fresh}}{\begin{array}{l}[\xi : a : \mathrm{Abs}; W_1^{l \uplus \{a\}} \leq b : \mathrm{Abs}; W_2^{l \uplus \{b\}}] \\ \blacktriangleright \exists \rho^{l \uplus \{a,b\}}. \left(\wedge \begin{array}{l}[\xi : a : \mathrm{Abs}; \rho^{l \uplus \{a,b\}} \leq W_2^{l \uplus \{b\}}] \\ [\xi : W_1^{l \uplus \{a\}} \leq b : \mathrm{Abs}; \rho^{l \uplus \{a,b\}}]\end{array} \right)\end{array}}$$

$$[\xi : a : \mathrm{Pre}(E_1); W_1 \not\geq a : \mathrm{Abs}; W_2] \blacktriangleright \textbf{false}$$

$$[\xi : \tau \leq \alpha] \wedge [\xi : \alpha \leq \tau'] \blacktriangleright [\xi : \tau \leq \tau']$$

FIGURE 4.8 – Simplification des contraintes locales

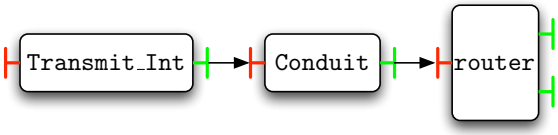


FIGURE 4.9 – Illustration de la propagation

nous avons le composant `Transmit_Int` dont le type de sortie $\xi[\mathrm{int}]$ doit être unifié avec le type d'entrée $\xi'[\eta]$ du composant `Conduit`. Ainsi, avec l'application des règles de résolution pour le sous-typage entre les types routés, nous obtenons une contrainte contenant $[\xi' : \mathrm{int} \leq \eta]$. Enfin, comme le conduit est

branché directement au routeur, nous avons aussi la contrainte que la variable ξ' doit être dupliquée : $\xi'[\eta] \leq r(\xi_1[\eta_1], \xi_2[\eta_2])$ ce qui nous donne, avec les règles de résolution $\xi' = \xi_1(\xi_2, \wedge)[\xi_1 : \eta \leq \eta_1] \wedge [\xi_2 : \eta \leq \eta_2]$. Logiquement, il faudrait avoir que les variables η_1 et η_2 soient instanciées en int, comme les seuls messages pouvant passer dans l'assemblage sont des entiers. Cette instanciation est calculée par nos règles de propagations en deux étapes :

1. Notre algorithme remarque que nous avons la contrainte $[\xi' : \text{int} \leq \eta]$, alors que ξ' se duplique en $\xi_1(\xi_2,)$. Les contraintes sur les messages de ξ' s'appliquent donc sur ξ_1 et ξ_2, et la propagation génère alors les contraintes $[\xi_1 : \text{int} \leq \eta]$ et $[\xi_2 : \text{int} \leq \eta]$.
2. Nous appliquons ensuite la transitivité dans les contraintes locales, pour avoir $[\xi_1 : \text{int} \leq \eta_1]$ et $[\xi_2 : \text{int} \leq \eta_2]$, ce qui nous donne l'instanciation voulue.

Les règles de ce paragraphe sont présentées Figure 4.10. La première génère les contraintes correspondantes à deux feuilles de routage en relation de sous-typage : les arbres de routage doivent avoir la même forme, est les messages doivent vérifier les contraintes de sous-typage. Les quatre règles suivantes mettent en place la transitivité de l'égalité des contraintes de duplication, et gèrent l'exploration des arbres de routages de façon similaires aux règles manipulant les contraintes sur les types routés. La sixième règle présente la propagation des contraintes impliquant variables globales. La gestion des contraintes manipulant de concert des variables locales et globales, présentée dans la règle suivant, nécessite l'utilisation de variables fraîches et globales afin de pouvoir propager les contraintes. Les deux règles suivantes présente la propagation due aux égalités entre variables de routage. Et finalement, les deux dernières règles vérifient que les kinds des variables de routage sont correctement manipulées. En effet, une variable non duplicable ne peut pas être instanciée en un arbre complexe, et il faut aussi vérifier qu'un arbre ne comporte pas sur une branche deux fois un même nom de routeur.

4.1.4 Les propriétés de base de l'inférence

Le but de l'algorithme d'inférence que nous venons de présenter est de décider automatiquement si un programme donné est typable ou non. Comme cet algorithme est construit en deux parties, nous prouvons pour chacune de ces parties les caractéristiques nécessaires pour que l'on soit assuré que l'ensemble ait le comportement souhaité. Tout d'abord, nous prouvons que la partie génération de contraintes calcule bien des types et des contraintes correspondant au typage du programme en paramètre :

Théorème 5 (Correction)**.** *Supposons donnée une génération de contrainte $D : F\,[C]$. Alors, pour toute substitution σ validant la contrainte C, $\sigma(F)$ est un type valide pour D.*

De plus, tout les types valides pour D correspondent plus ou moins à ce que nous génère l'algorithme :

$$\xi[E] \leq \xi'[E'] \blacktriangleright [\xi : E \leq E'] \wedge \xi = \xi' \qquad \xi = \xi' \wedge \xi' = \xi'' \blacktriangleright \xi = \xi''$$

$$\xi = \xi' \wedge \xi' = r(\xi_1, \xi_2) \blacktriangleright \xi = r(\xi_1, \xi_2) \qquad \xi = r(\xi_1, \xi_2) \wedge \xi = r(\xi'_1, \xi'_2) \blacktriangleright \bigwedge \xi_i = \xi'_i$$

$$\frac{r \neq r' \qquad R' \triangleq R \uplus \{r, r'\} \qquad \forall 5 \leq i \leq 8,\ \xi_i^{R'} \text{ fresh}}{\begin{array}{c}\xi = r(\xi_1^{R \uplus \{r\}}, \xi_2^{R \uplus \{r\}}) \wedge \xi = r'(\xi_3^{R \uplus \{r'\}}, \xi_4^{R \uplus \{r'\}}) \\ \blacktriangleright \exists \xi'_i.(\xi_1 = r'(\xi_5, \xi_6) \wedge \xi_2 = r'(\xi_7, \xi_8) \wedge r(\xi_5, \xi_7) = \xi_3 \wedge r(\xi_6, \xi_8) \leq \xi_4)\end{array}}$$

$$\frac{\mathtt{kind}(\alpha) = \mathtt{kind}(\tau) = \mathfrak{G} \qquad \xi \in \mathit{fv}(C)}{C \wedge [\xi_2 : \alpha \asymp \tau] \blacktriangleright [\xi : \alpha \asymp \tau]}$$

$$\frac{\begin{array}{c}\tau \notin \mathcal{V} \\ \mathtt{kind}(\alpha) = \mathfrak{G} \qquad \overline{\beta} = \{\beta \mid \beta \in \mathit{fv}(\tau) \wedge \mathtt{kind}(\beta) = \mathfrak{L}\} \qquad \overline{\beta}' \text{ fresh} \wedge \mathtt{kind}(\beta') = \mathfrak{G}\end{array}}{[\xi : \alpha \asymp \tau] \blacktriangleright \exists \overline{\beta}'. \bigwedge([\xi : \beta' \asymp \beta]) \wedge [\xi : \alpha \asymp \tau\{^{\overline{\beta}'}/_{\overline{\beta}}\}]}$$

$$[\xi : \tau_1 \leq \tau_2] \wedge \xi = \xi' \blacktriangleright [\xi' : \tau_1 \leq \tau_2]$$

$$[\xi : \tau_1 \leq \tau_2] \wedge \xi = r(\xi_1, \xi_2) \blacktriangleright \bigwedge [\xi_i : \tau_1 \leq \tau_2] \qquad \frac{(\mathtt{kind}(\xi) = \mathfrak{F}) \vee \{r\} \not\vdash \xi}{\xi = r(\xi_1, \xi_2) \blacktriangleright \textbf{false}}$$

$$\frac{\begin{array}{c}R_1 \not\subset R_2 \wedge R_2 \not\subset R_1 \\ \xi^{R_1 \cup R_2} \text{ fresh} \qquad (\mathtt{kind}(\xi_1^{R_1}) = \mathfrak{F} \vee \mathtt{kind}(\xi_2^{R_2}) = \mathfrak{F}) \Leftrightarrow \mathtt{kind}(\xi^{R_1 \cup R_2}) = \mathfrak{F}\end{array}}{\xi_1^{R_1} = \xi_2^{R_2} \blacktriangleright \exists \xi^{R_1 \cup R_2}. \xi_1^{R_1} = \xi^{R_1 \cup R_2} \wedge \xi_2^{R_2} = \xi^{R_1 \cup R_2}}$$

FIGURE 4.10 – La propagation et la duplication

Théorème 6 (Complétude)**.** *Supposons donné un programme D et un type F valide pour D. Alors, pour toute génération de contrainte $D : F'[C]$, il existe une substitution σ validant C et un ensemble de variable $\overline{\alpha}$ tels que $\forall \overline{\alpha}.\sigma(F') = F$.*

Ces deux théorèmes ensemble nous donnent qu'un programme D est typable si et seulement si lors d'une génération de contrainte $D : F[C]$, la contrainte C est satisfiable. Il ne nous reste plus qu'a montrer que notre solveur peut décider automatiquement de la satisfiabilité d'une contrainte pour prouver notre algorithme d'inférence. Pour ce faire, nous montrons tout d'abord que cette résolution par fermeture se termine dans tout les cas, qu'il ne modifie pas, en rajoutant de nouvelles contraintes de base, la sémantique de la contrainte originale, et que le résultat est satisfiable si et seulement si il contient la contrainte de base **false**. Le processus de résolution se terminant, son résultat est fini, et donc il est facile de vérifier sa satisfiabilité.

Définition 10. *Supposons donné une contrainte C : nous notons* $\mathtt{close}(C)$ *la plus petite (w.r.t $\subset$) contrainte contenant C et telle que s'il existe $C_1 \subset C$ avec $C_1 \blacktriangleright C_2$, alors C_2 est contenu dans* $\mathtt{close}(C)$.

Théorème 7 (Terminaison)**.** *Pour toute contrainte C, la fonction* `close` *possède un plus petit point fixe contenant C, appelé la* fermeture *de C.*

Théorème 8 (Stabilité)**.** *Pour toute contrainte C, C et* `close`(C) *sont logicalement équivalentes.*

Théorème 9 (Satisfiabilité)**.** *Supposons donnée une contrainte C telle que* `close`$(C) \subset C$ *et ne contenant pas* **false**. *Nous avons alors que C est satisfiable.*

Ces trois théorèmes sont démontrés en anglais dans l'annexe C.

4.2 L'implémentation de l'inférence de type

Avec l'algorithme d'inférence que nous venons de présenter, nous avons donc un second système de type adapté aux composants, permettant de typer des comportements tels que le routage et le multiplexage, et d'utilisation simplifiée grâce à notre inférence, qui rend obsolète la nécessité d'annoter par des types les différentes parties d'un programme. Du fait que ce travail théorique est relativement abouti, nous avons décidé de l'implanter sur des modèles à composants existant, afin de vérifier la pertinence de notre approche sur des exemples concrets. En effet, comme nous utilisons un calcul spécifique pour construire nos systèmes de types, il se peut que le portage de ce dernier aux modèles à composant se fasse mal. De plus, le système de type peut avoir des défauts que seule son utilisation peut nous faire remarquer. Nous présentons dans cette partie l'implantation que nous avons faite de notre inférence, que nous avons portée sur Fractal, en la testant sur des assemblages Dream, et aussi sur le modèle à composant Click, afin d'évaluer l'adaptation de notre travail à d'autres modèles que Dream. Nous avons choisi le modèle Click car bien que différent de Dream, son objectif est aussi la manipulation de messages structurés tels qu'utilisés dans les protocoles TCP et UDP.

Cette partie est structurée comme suit : nous faisons tout d'abord un rapide survol de Click et de Dream que nous représentons brièvement ici ; notons que nous faisons dans cette partie un amalgame entre Dream et Fractal afin de simplifier nos explications ; vient ensuite la description de l'agencement interne de notre implémentation avec la description de quelques modules annexes en introduction à notre programme d'inférence ; et finalement, nous présentons les quelques expérimentations que nous avons effectuées avec notre programme d'inférence, et concluons par une discussion sur les améliorations possible de notre travail.

4.2.1 Click et Dream

Ces deux structures sont construites en deux parties : (i) une librairie de composants, que des utilisateurs peuvent utiliser et assembler en des programmes de complexité arbitraire, et typiquement des MOMs qui est la forme de programme

pour laquelle ces librairies ont été conçues ; (ii) une chaîne de compilation qui permet, à partir d'une spécification assez abstraite d'un assemblage décrit à l'aide d'un ADL, de générer le code, la compilation, et le programme correspondant. Néanmoins, ces deux structures ont aussi leurs propres spécificités, concernant autant les fonctionnalités apportées par la librairie de composants que dans le fonctionnement de la chaîne de compilation.

Click. Cet outillage utilise un modèle rudimentaire et efficace de composants, ne comportant que des composants primitifs, écrits en C++, des interfaces (appelées *ports*[2]) et des liaisons. À notre connaissance, ce modèle ne permet aucune modification de la structure d'un programme à l'exécution, ni d'exploration de sa structure. Les messages échangés entre les composants Click sont de très bas niveau, et sont considéré comme de simples tableaux d'octets. Les opérations de manipulation de base de ces messages s'en ressent, comme avec le composant `StoreData` qui remplace une donnée placée après un certain nombre de bits, par un autre tableau d'octets de même taille, ou les composants `Strip` et $\widehat{\text{U}}$nStrip qui diminuent et étende la taille d'un message d'un certain nombre d'octets . Ces opérations de base niveau sont complétées par un ensemble de composants au fonctionnalités plus évoluées, notamment concernant la manipulation des protocoles TCP et UDP. Un assemblage utilisant la librairie Click généralement les composants suivants : *InfiniteSource* qui génère un nombre infini de messages, *classifier* qui route les messages en fonction des données qu'ils contiennent, *queue* qui stocke un nombre fini de messages en attendant d'être transmis à d'autres composants et *StoreData* qui modifie le contenu des messages.

L'ADL de Click permet de définir dans un fichier quels composants sont utilisé dans un assemblage, quels sont leurs *paramètres* (comme le nombre d'octet à enlever aux différents message par le composant `Strip`), et comment ils sont assemblés. De plus, cet ADL permet de *nommer* des composants afin de pouvoir s'en servir plus tard dans le document. Une fois que le fichier ADL contient toutes les informations nécessaires à la spécification de l'assemblage voulu, il est traité par la chaîne de compilation Click. Cet outil traduit ce fichier en un programme C++, qui est ensuite assemblé avec les composants de Click, compilé et exécuté. De fait, bien que tout les éléments de Click soient implantés en C++, les messages n'ont ni structure ni types, ce qui permet une très grande flexibilité dans leur manipulation, mais aussi autant de chance de commettre des erreurs lors de la définition d'un assemblage. Néanmoins, dans tout les exemples d'utilisation de cette librairie que nous avons étudiés, les messages sont vus et manipulés comme des structures tels que les messages de Dream. Ainsi, l'adaptation de notre système de type à base de types rangés a un sens ici, et peut combler le manque d'outils de vérification de Click.

Dream. Comme nous l'avons vu, cette librairie est elle aussi basée sur un modèle à composant, mais plus évolué que celui de Click. Ce modèle permet

2. contrairement à Dream, ces ports ne sont pas nommé et sont référencés par des indices

en effet de construire un assemblage, et de le placer dans un composant *composite*, ce qui permet de structurer un programme complexe. Il possède aussi des capacités d'introspection de la structure de programmes en cours d'exécution, ainsi que des opérateurs de base afin de modifier cette structure comme bon il nous semble. DREAM, contrairement à CLICK, est implanté en Java, et ne manipule que des messages structurés et utilisant des objets Java. Nous sommes donc en présence de messages de haut niveau, avec typage et sous-typage basé sur la hiérarchie des classes. Néanmoins, ce typage donné par Java reste incapable de vérifier la correction des assemblages DREAM, principalement parce qu'il ne comporte aucun élément permettant de vérifier les manipulation de messages structurés, et aussi car il n'est pas conçu pour typer des assemblages de composants. Notre système de type et notre algorithme d'inférence ont donc un réel intérêt ici, et peuvent être appliqués tout en étant quasi-conforme avec le sous-typage de Java qui peut être considérée comme étant la relation de sous-typage entre constructeurs de types [3]. Finalement, la librairie DREAM met à disposition des composants dédiés à la gestion des ressources, afin de pouvoir construire des assemblages pour une grande variété d'environnements matériels, tout en offrant la simplicité de programmation qu'offre les composants et Java.

4.2.2 La structure de notre implantation

La conception du portage de notre système de type aux librairies CLICK et DREAM est basée sur les différentes données dont notre algorithme d'inférence a besoin pour fonctionner. Tout d'abord, nous avons besoin de connaître les types des composants primitifs, qui sont une donnée essentielle de notre système de type (comme les fichier ADL ne décrivent que des assemblages de composants sans messages en cours d'échange, nous n'avons pas besoin de connaître les types des constantes). Nous avons ensuit besoin de traduire les configurations à typer en des termes de notre calcul. En effet, même si notre calcul est construit pour suivre au maximum les lignes directrices du modèle FRACTAL, nous avons vu qu'il possède quelques différences de conception, afin de faciliter la construction d'une sémantique formelle, d'un système de type, et les preuves des propriétés de ce dernier. Néanmoins, il est relativement aisé de passer d'un assemblage CLICK et DREAM à un programme équivalent de notre calcul. Une fois le programme CLICK ou DREAM traduit dans notre modèle, et les types des composants primitifs donnés, il nous reste plus qu'a implanter notre algorithme de génération de contrainte et de résolution pour décider si le programme donné est typable ou non. Cette implantation doit toutefois respecter certains critères, comme être relativement efficace, ou donner des messages d'erreur, lors de problèmes de typage, compréhensibles et utiles pour la correction des problèmes dans l'assemblage.

Dans cette approche de l'implantation, nous pouvons constater que la partie génération de contraintes et résolution est commune aux deux librairies CLICK

3. Il faut toutefois légèrement adapter le sous-typage de Java, car il ne forme pas un treillis entre les classes

et Dream, alors que les composants primitifs et les ADL sont spécifiques à chacun de ces modèles. Nous avons donc conçu notre outil comme un ensemble de trois modules distincts mais collaborant : (i) un module intégré dans la chaîne de compilation de Dream recueillant les informations sur les types des composants primitifs et sur les configurations afin de les transformer en un terme de notre calcul ; (ii) un programme parsant les fichier ADL de Click et de comportement similaire au module précédent ; (iii) et un programme principal qui reçoit en entrée un treillis de constructeurs de types constituant la relation sur laquelle est construite notre propres relation de sous-typage, une liste de composant primitifs avec leurs types, et un terme de notre calcul, et qui vérifie la validité de ce terme par rapport à notre système de type. Nous pouvons remarquer que la partie concernant Click n'est pas, contrairement à celle pour Dream, intégrée dans la chaîne de compilation, mais constitue un programme indépendant. La raison principale de ce choix est notre méconnaissance de cet outil de Click, mais il n'y a a priori aucune difficulté majeur à l'intégration de notre programme à cette chaîne.

Le module pour Dream

Nous entrons ici un peu plus dans les détails de l'implémentation : ce qui a été réalisé et les modifications apportées à la chaîne de compilation de Dream [64] afin d'intégrer les différentes données nécessaires au bon fonctionnement de notre outil. Ces modifications, ainsi que l'intégration du module à cette chaîne ont été extrêmement facilitées par la structure flexible de cette dernière.

La chaîne de compilation. La Figure 4.11 présente la forme générale de cet outil, qui est assemblé comme une file de composants Fractal, chacun opérant un traitement particulier sur son entrée. Ces fonctionnalités sont regroupées en trois composants principaux, que nous décrivons brièvement par la suite.

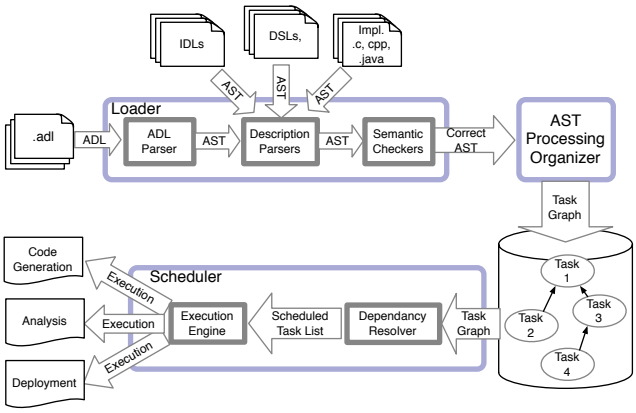


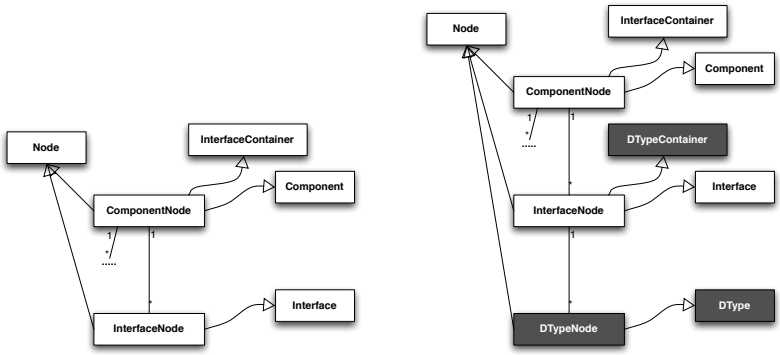
FIGURE 4.11 – Une vue d'ensemble de la chaîne de compilation

Le composant *Loader* a pour but de parser des fichiers de configurations et de description d'assemblage (e.g. ADL, IDL, DSL, etc., afin de construire un arbre (appelé AST pour Abstract Syntax Tree) représentant ces différentes données. Il permet aussi de vérifier certaines données critiques mémorisées dans cet arbre. Ce composant est aisément extensible par l'ajout de nouveau sous-composant permettant la prise en compte de nouveaux types de fichier, ou de nouvelles données dans des formats existant, et étendu. Le composant *organizer* parcourt l'arbre construit par le *Loader*, et génère un graphe de *tâches*. Ces dernières correspondent à la création, par génération de code source et compilation, des différentes partie du programme décrit dans l'AST. De plus, elles permettent aussi de mettre en place les phases d'exécution et de tests du programme construit, eux aussi mentionnés dans l'AST. Enfin, le composant *scheduler* ordonnance et exécute toutes les tâches construites, en garantissant que les relations de dépendance spécifiées dans le graphe données par le *organizer* sont bien respectées.

Nous avons choisi de placer notre module pour la vérification de types dans le composant *Loader*. C'est en effet l'emplacement le plus naturel, où l'on peut charger les données concernant les assemblages et les autres informations nécessaires à la vérification de type, notamment les types des composants primitifs. Nous avons choisi de placer ces derniers directement dans la description des assemblages, lorsque les composants primitifs sont utilisés. Cela nous a demandé de modifier légèrement la syntaxe ADL (basée dur su XML) utilisée pour la description des assemblages DREAM, ainsi que l'AST afin d'y intégrer ces nouvelles données.

Modification des AST et XML. L'AST utilisé dans la chaîne de compilation DREAM a une structure proche à du XML, où chaque noeud représente un élément de l'ADL. Tout les noeuds de l'arbre héritent de la classe `Node`, auquel est adjoint des fonctions de manipulation spécifiques aux données contenues

dans le noeud. Pour être prises en compte par l'AST, ces fonctions doivent être déclarées dans une interface que le noeud doit implanter. Enfin, si un noeud possède des fils, il doit aussi implanter l'interface `Container` associé à tout les types de fils qu'il peut avoir. Nous illustrons cette hiérarchie de classes et d'interfaces par la Figure 4.12a qui donne un aperçu de l'organisation originale de l'AST. Nous avons, en haut à droite, l'interface Node dont doit hériter tout



(a) Quelques noeuds de l'AST (b) Notre modification de l'AST

FIGURE 4.12 – Une partie de l'AST de FRACTAL

les noeuds de l'arbre AST. Nous pouvons ainsi voir que cet exemple présente deux noeuds, `ComponentNode`, correspondant à la définition d'un composant, et `InterfacceNode` correspondant à une interface dans un composant. La classe `componentNode` implémente l'interface `Component`, signifiant ainsi qu'elle s'occupe bien des composants, mais aussi `InterfaceContainer`, car les composants comportent des interfaces : les noeuds correspondant aux composants peuvent alors avoir des fils, correspondant à leurs interfaces. De manière similaire, la classe `InterfaceNode` implante l'interface `Interface`, mais comme une interface est seulement un point de rattachement dans le modèle FRACTAL, cette classe n'implante aucune autre interface, et en particulier, aucune interface `Container`. Nous pouvons remarquer avec cet exemple, qu'au niveau des classes, il n'y a pas de distinction entre les composants composite et les composants primitifs : un composant est composite pour l'AST lorsqu'il possède un fils qui est lui-même un composant. La Figure 4.12b montre comment nous avons modifié la structure de l'AST afin d'y inclure les types des composants primitifs. Nous avions plusieurs possibilités pour permettre cette inclusion, et nous avons choisi de placer les informations de typage dans la définition des interfaces. En effet, le type d'un composant (qui est un type de processus) consiste simplement en la définition de types routés pour chacune de ses interfaces. Comme les interfaces sont déjà définies dans l'AST, nous avons préféré associer directement les types routés aux interfaces plutôt que de mettre un type de processus dans le composant, et ensuite vérifier que ce type contenait bien les bonnes interfaces. L'ajout des informations de types à l'AST s'est fait par la création d'une nouvelle classe

DTypeNode, et deux nouvelles interfaces DType et DTypeContainer. La classe DTypeNode est un noeud de l'AST contenant un type routé comme il est spécifié dans l'interface DType dont hérite la classe. Ce noeud est inclus dans un noeud d'interface, qui maintenant implante aussi l'interface DTypeContainer.

Au niveau de la syntaxe ADL, la modification que nous avons apportée est relativement similaire. En effet, cette syntaxe est du XML contrôlée par un fichier DTD spécifiant les différentes balises du langage, ainsi que leur organisation. Nous avons légèrement modifié le fichier DTD en ajoutant une sous-balise optionnelle, nommée <dtype>, dans la balise de définition des interfaces. Cette balise ne comporte aucune option, et ne sert qu'à définir le type routé d'une interface. Notons finalement que comme il n'y a pas de différence, ni pour l'AST, ni pour le DTD, entre un composant composite et un composant primitif, que la balise <dtype> est optionnelle, ainsi que la définition d'un type dans un noeud d'interface. Le module d'inférence de type se charge alors de vérifier que tout les types sont donnés pour les composants sans composant fils (et donc primitif), et ne tient pas compte des annotations de typage donnés pour les composants composites.

Le module. La Figure 4.13 présente la structure interne du composant Loader de la chaîne de compilation FRACTAL. Ce Loader est formé par une chaîne de sous-composants, les premiers chargeant les différents fichiers ADL et les transformant en un AST, les suivants manipulant et vérifiant la bonne formation de cet arbre.

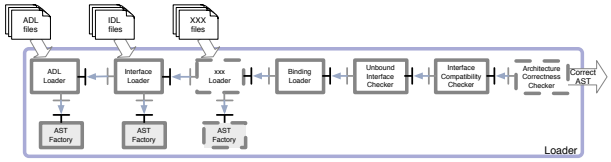


FIGURE 4.13 – La structure du composant Loader

La modification du fichier DTD a pour effet d'étendre le comportement du chargeur principal, qui rajoute maintenant, et automatiquement, les noeuds de typage sur les interfaces correspondantes. La chaîne étant conçue de manière totalement modulaire, cet extension n'a pas d'incidence sur les autres composants de la chaîne. Nous avons donc simplement à rajouter un nouveau module dans cette chaîne qui (i) récupère les différentes données de l'AST concernant notre système de types, comme la structure du programme, et les annotation de typage sur les composants primitifs ; (ii) traite ces données (comme vérifier que les composants primitifs sont bien annotés, ou transformer les interfaces des composants en canaux de communications) ; (iii) envoie ces informations traités au programme principal d'inférence de type ; et (iv) récupère le résultat de l'inférence, et affiche les messages, de validation ou d'erreur, correspondant. Nous avons rajouté ce module en fin de chaîne de chargement, après que tout

les traitements des autres modules aient été effectués, car beaucoup de ces modules manipulent la structure de l'AST, pour par exemple prendre en compte l'héritage entre composant et obtenir un AST final correspondant totalement à la configuration spécifiée.

Le module que nous avons conçu est assez élémentaire, et consiste principalement en un parcours de l'AST pour la récupération de données. Sa partie la plus complexe concerne les traitements, en particulier la création des canaux de communication, et la gestion du sous-typage Java. Pour la première difficulté il y a en effet une différence entre FRACTAL dont les composants communiquent sur via des interfaces reliées, et notre modèle où les composants communiquent via des canaux de même nom. Notre fonction de traduction crée un nom de canal pour chaque interface de chaque composant, et unifie les noms des interfaces reliées. En ce qui concerne le sous-typage de Java, nous utilisons l'introspection donnée par la machine virtuelle pour construire le graphe de dépendance entre les différentes classes et interfaces utilisées par l'assemblage. Ce graphe est ensuite complété pour en faire treillis par une classe `TOP` jouant le rôle de borne supérieure, et des relations permettant aux interfaces d'hériter de la classe `Object`. Ce dernier ajout n'a pas d'incidence sur le typage de Java, car les messages manipulés par DREAM sont déjà des objets. La structure de l'assemblage, ainsi que la relation de sous-typage et les types des composants primitifs sont ensuite envoyés pour traitement au programme principal d'inférence de type. Ce programme renvoi des messages d'erreurs si le programme n'est pas typable, qui sont alors simplement affichées à l'écran. Finalement, comme notre module reste un outil d'essai, il transmet l'AST au reste de la chaîne sans modification, même lorsque l'assemblage n'est pas typable : l'utilisateur peut arrêter le processus de compilation en cas d'erreur s'il le désir, mais ce n'est pas le comportement par défaut de la chaîne pour le moment.

Le programme pour CLICK

Pour le portage de notre système de type à CLICK nous avons du créer un programme annexe à la chaîne de compilation, car notre connaissance de cette outil, son organisation interne et comment il fonctionne, est relativement faible. La thèse [62] présentant toute la conception de CLICK donne entre autre un aperçu de l'outil de compilation, la syntaxe utilisée pour les fichiers ADL, etc, mais nous n'avons pas eu le temps de lire en détail ce document. Mais heureusement, les fichiers ADL courant de CLICK ont une syntaxe relativement simple et facile à parcourir. Notre programme possède donc un chargeur de fichier que nous avons conçu nous même, et manipule les données de façon similaire au module de DREAM. Une différence par rapport à DREAM concerne le chargement des configuration et des types. En effet, les composants de CLICK ont la particularité d'avoir des paramètres, qui influent sur leur type, et certain ont aussi un nombre arbitraire d'interfaces. Typer ce genre de composant est assez délicat, et demande souvent de typer aussi les paramètres, comme dans le cas du composant `InfiniteSource` où le paramètre est le messages envoyé par le composant. Mais, pas manque de temps, tout ce travail de typage fin des

composants à du être reporté à une date ultérieure. Afin de ne pas modifier la syntaxe des fichier ADL de Click, nous avons inclus nos types dans un fichier annexe, donnant pour chaque nom de composant son type, qui ne concerne que ses interfaces de base. Ces types sont étendus par un algorithme assez simple pour prendre en compte toutes les interfaces optionnelles qui sont utilisées dans l'assemblage. Finalement, comme Click ne comporte pas de types de donnés et ne considère ses messages que comme des tableaux d'octets, nous n'avons pas inclus dans notre programme de moyen de définir une relation de sous-typage entre des constructeurs de types. Nous créons par défaut une simple structure de treillis tel que les types de l'assemblages n'ont qu'une seule borne inférieur et supérieur, `Bot` et `Top`, ces deux types ne faisant pas partie des types utilisés par l'assemblage.

Nous terminons la présentation du programme pour Click par deux figures. La première 4.14a expose deux utilisations possibles du composant `InfiniteSource`. Comme nous pouvons le voir, ce composant possède différents paramètres, dont trois sont utilisés ici (les trois autres paramètres valides de ce composant gardent une valeur par défaut). La première utilisation de `InfiniteSource` est directe, c'est à dire que ce composant est utilisé directement dans l'assemblage spécifié par le fichier ADL. La seconde utilisation crée un nouveau composant, nommé `source`, qui se comporte comme le précédent, mais qui ne fait pas partie de l'assemblage. Ce composant pourra être utilisé par la suite, par référence à son nom. Rappelons que la création de ce nom peut grandement aider le typage, car on peut alors spécifier le type de ce nouveau composant. Sans ce nom, il est impossible avec notre implantation actuelle de donner un

```
InfiniteSource(DATA \<00
00 c0 ae 67 ef 08 00 45 00 00
28 00 00 00 00 40 11 77 c3 01 00
00 01 02 00 00 02 13 69 13 69 00
14 d6 41 55 44 50 2070 61 63 6b
65 74 21 0a>, LIMIT 5, STOP
true)
```

(a) Composant anonyme

```
source : : InfiniteSource(DATA
\<00 00 c0 ae 67 ef 08 00 45 00
00 28 00 00 00 00 40 11 77 c3
01 00 00 01 02 00 00 02 13 69
13 69 00 14 d6 41 55 44 50 2070
61 63 6b 65 74 21 0a>, LIMIT
5, STOP true)
```

(b) Composant nommé 'source'

Figure 4.14 – Deux utilisation du composant `InfiniteSource`

type précis à une instance du composant `InfiniteSource`, qui possède alors un type de sortie générique (une variable locale de type élémentaire routée par une variable de routage duplicable). La seconde figure 4.15 présente l'algorithme d'assignation des types aux composants Click. En effet, nous avons les composants nommés (qui ont déjà un type), et ceux utilisés directement, auxquels nous donnons un type approximatif.

if c n'est pas nommé **then**
  $c \leftarrow$ le composant de base de c
end if
$F \leftarrow$ le type de c
$F \leftarrow$ la complétion de F pour typer tous les ports utilisés par c

FIGURE 4.15 – L'extension des types CLICK

4.2.3 Le programme central d'inférence

Ce programme est dédié à la vérification de type d'assemblages, qui peuvent lui être donné de deux façons différentes, ce qui correspond à deux modes d'exécution du programme. Lorsqu'il est lancé en mode *fichier*, le programme attend sur son entrée standard un fichier décrivant toutes les données nécessaires à la vérification de type d'un assemblage. En mode *ligne de commande*, le programme affiche un message d'accueil, et attend une commande de l'utilisateur pour charger des données, ou effectuer une opération liée à l'inférence de type. Ce dernier mode offre donc plus de souplesse d'exécution, et est surtout utilisé pour tester le bon fonctionnement du programme. Néanmoins, l'utilisation classique de notre outil suit toujours toujours la même suite de commandes : (i) charger les constructeurs de types ; (ii) charger la relation de sous-typage entre ces constructeurs ; (iii) charger les types des composants primitifs ; (iv) charger l'assemblage ; (v) générer les contraintes à partir de l'assemblage donné ; (vi) résoudre ces contraintes ; et (vii) afficher les messages d'erreur s'il y en a, ou afficher le type du composant.

Son Implémentation

Celle-ci est réalisée dans le langage OCaml, et consiste en treize modules différents. La Figure 4.16 présente ces différents modules, ainsi que leur relation de dépendance. Le module principal est **pluto**, qui est aussi le nom de notre programme d'inférence.

MapString et SetString. Ces deux modules implantent respectivement de simples instanciations des signatures OCaml **Map.S** et **Set.S**, avec string comme type de base, et y rajoute quelques fonctionnalités utiles et récurrentes pour le reste de notre programme. **MapString** sert surtout à l'implantation des types ensembles, et **SetString** sert pour la mise en place des kinds.

Informations. Ce module contient toutes les informations de base d'un type, comme par exemple, sa (pour les types élémentaires) ou ses (pour les types routés et les types rangés) kinds. D'autres informations y sont aussi stockées, comme le composant et l'interface où le type est utilisé dans l'assemblage, afin

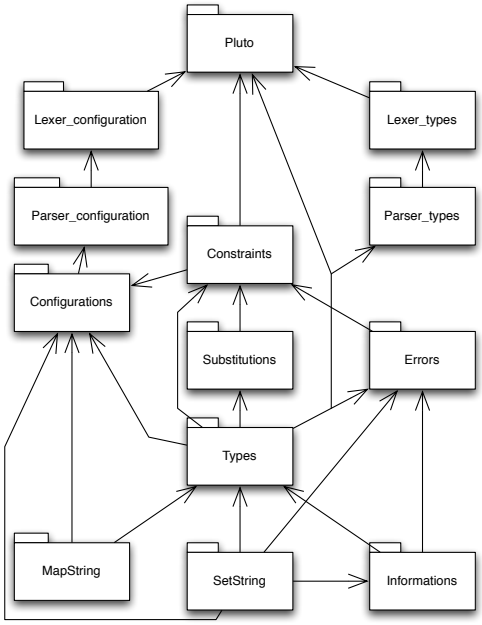


FIGURE 4.16 – Les dépendances entre les modules

de connaître l'endroit où un problème de typage survient, et le signaler dans les messages d'erreur.

Types. C'est dans ce module que nous définissons l'implantation de tous les types de notre système, ainsi que la relation de sous-typage et les types des composants primitifs. Il contient donc deux table pour stocker ces deux dernières données, et propose un large choix de fonctionnalités de manipulation de types, comme leur création, la manipulation de leurs informations, afin de rendre facile la définition de l'inférence.

Errors. Lorsqu'une erreur de typage survient dans notre programme, celui-ci lève une exception qui contient quelques informations utiles pour résoudre le problème. Ces différentes exceptions sont définies dans ce modules, qui contient aussi différentes fonctions pour afficher dans un message clair les différentes informations contenues dans chaque exception.

Substitutions. Comme son nom l'indique, ce module implante et gère les substitutions en définissant le résultat d'une application d'un type à une substitution.

Constraints. Ce module définit l'implantation de tout les types de contraintes de notre inférence de type. Il implante aussi l'algorithme de résolution tel qu'il

est définit dans la partie 4.1.3. De plus, lorsque l'algorithme de résolution termine sans avoir levé aucune erreur, ce module permet de calculer une substitution validant la contrainte.

Configurations. C'est ici que sont définit les différentes structures représentant les composants et processus. Nous avons omis d'inclure les envois de messages dans notre implantation, car de tels éléments ne sont généralement pas déclaré dans un ADL, et peuvent facilement être encodé par un composant primitif. La réduction du calcul n'est pas non plus implantée, n'étant pas utile pour le mécanisme d'inférence. Par contre, ce module contient l'implantation de l'algorithme de génération de contrainte, à partir d'une configuration, et des types des composants primitifs donnés dans le module **Types**.

Lexers et Pasers. Ces outils servant à parcourir les différentes informations textuelles données en entrée de notre programme sont classiquement construits avec `camlyacc` et `camllex`.

Pluto. Ce module principal utilise tout les modules sus-mentionnés en assemblant toutes leurs fonctionnalités, permettant ainsi les deux modes d'exécution du programme. Par exemple, il contient une table associant chaque nom de commande à l'appel d'une fonction particulière, comme la commande "load type" qui exécute le parseur chargeant les informations sur les types, ou la commande "solve" qui active la fonction de résolution des contraintes générées. En mode *fichier*, l'exécution du programme est pré-définie, et charge tout d'abord les types, puis la configuration, génère les contraintes qu'il résout, et affiche le résultat.

La syntaxe d'entrée de notre programme

Dans le chapitre précédent, nous avons donné une syntaxe pour les assemblages et les types, relativement épurée afin de simplifier la définition de noter système de type. Nous avons voulus garder un langage similaire pour la description de ces éléments pour notre programme d'inférence, mais quelques adaptations ont été néanmoins nécessaires, en particulier pour les types où il peut être intéressant de pouvoir spécifier les kinds des variables. D'autre légères modification ont été apportées à la syntaxe, inspirées du langage OCaml, afin d'avoir un résultat plus esthétique et lisible. Dans la présentation de notre syntaxe, les mots clefs sont écrits en gras, et '*ident*' correspond à toute chaîne de caractères alphanumériques qui n'est pas un mot clef.

La syntaxe des assemblages. Avec notre simplification du calcul, qui ne prend pas en compte les envois de messages, une configuration est simplement un ensemble de composant mis en parallèle. Dans notre syntaxe (décrite Figure 4.17), un processus est donc une liste de composants séparés par un point-virgule et terminée par deux point-virgules. Un composant peut être primitif, et dans ce cas, est identifié par son nom avec *ident*, ou composite, auquel cas nous spécifions

sont nom avec *ident*, ses capacités d'entrées et de sortie, ainsi que son processus interne. Notons que nous avons une troisième façon d'utiliser un composant, de la forme *ident* **with** channelMod, ce qui permet d'utiliser des composants avec des canaux différents de ceux spécifiés dans le type. Cela est extrêmement utile

main	::=	componentList ; ;
componentList	::=	component | component ; componentList
component	::=	*ident* | *ident* **with** channelMod
	|	*ident* = restriction **begin** componentList **end**
restriction	::=	(**input** : stringList ; **output** : stringList)
channelMod	::=	*ident* -> *ident* | *ident* -> *ident* , channelMod
stringList	::=	0 | *ident* | *ident* , stringList

FIGURE 4.17 – Les assemblages

pour placer deux instances d'un même composant dans différents endroits d'un assemblage, ou, dans le cas de DREAM et CLICK, de mettre facilement en place le mécanisme de remplacement d'interfaces en canaux. Considérons par exemple le composant `InfiniteSource` avec le port 1 connecté au port 2 du composant `Idle`. Notre algorithme de transformation des ports unifie ces deux ports en remplaçant par exemple les deux ports par le canal a. Ainsi, l'assemblage dans notre langage aura la forme suivante

`InfiniteSource` **with** 1 -> a; `Idle` **with** 2 -> a; ;

Ainsi, avec ces annotations de remplacement de canaux, les deux composants gardent le même comportement, mais sont maintenant connectés, ce qui est pris en compte aussi au niveau des types. Finalement, ce remplacement de canal est aussi très pratique lorsque nous écrivons des assemblages à la main : nous pouvons spécifier les composants ayant des interfaces avec des noms génériques, comme `entrée` ou `sortie`, et connecter ces interfaces dans l'assemblage avec la construction **with**. L'entrée 'restriction' de la grammaire présente comment les capacités d'entrée et de sortie des composants sont définies. Ces deux capacités consistent en des liste de noms de canaux séparés par une virgule, 0 étant la liste vide.

La syntaxe des types. Cette syntaxe, présentée Figure4.18, est plus complexe que la précédente, car nous avons besoin de plus de constructions pour définir toutes les données concernant les types. En effet, en plus des types tels

que nous les avons présentés le chapitre précédent, nous avons aussi besoin de déclarer les constructeurs de types, la relation de sous-typage, et le type de chaque composant primitif utilisé. La définition de ces différentes données se présente dans notre syntaxe sous la forme d'une liste de déclarations terminée par deux point-virgules. Nous avons trois types de déclaration : (i) **type** *ident* définit *ident* comme étant un nouveau constructeur de type ; (ii) **rel** *ident ident* instaure une relation de sous-typage entre deux constructeurs de type, le premier étant alors sous-typge du second ; et (iii) **val** *ident* : fType qui sert à définir le type d'un composant primitif (*ident* étant le nom du composant, et fType son type). Le reste de la syntaxe est relativement similaire à celle décrite Partie 3.2, avec quelques adaptations au niveau de la syntaxe, comme par exemple l'union, qui est représentée par '+' ou l'ensemble vide qui est représenté par zéro. La syntaxe des variables est toutefois étendue, afin de prendre en compte

main	::=	;; | decl main
decl	::=	**type** *ident* | **rel** *ident ident* | **val** *ident* : fType
fType	::=	**forall** stringList.fType | cType -> cType
cType	::=	0 | *ident* : rType | *ident* : rType + cType
rType	::=	[routingVar | mType] | [*ident* : rType | rType]
mType	::=	simpleVar | *ident* | *ident*(mTypeList)
	|	{*row*} | (mType)
row	::=	simpleVar | **Abs**
	|	*ident* : **Abs** ; row | *ident* : **Pre** mType ; row
mTypeList	::=	mType | mType , mTypeList

routingVar	::=	'*ident*
	|	@(*ident* : **Kind** = stringList) | @(*ident* : **Kindf** = finality)
	|	@(*ident* : **Kind** = stringList | **Kindf** = finality)
simpleVar	::=	'*ident* | @(*ident* : **Kindf** = locality)
locality	::=	&**local** | &**global**
finality	::=	&**duplic** | &**final**
stringList	::=	0 | *ident* | *ident* , stringList

FIGURE 4.18 – Les types

la spécification des kinds. Cette spécification est en effet quelques fois nécessaire, en particulier pour le typage des routeurs, où les variables de routages ont une kind non vide. La façon basique d'utiliser une variable est similaire à OCaml,

où l'on utilise la syntaxe `'`*ident*. Dans ce cas, notre algorithme de chargement essaye de donner les kinds correspondant le mieux à cette variable, mais échoue tout de même quelques fois. Dans ces conditions, nous pouvons donner explicitement la ou les kinds des variables, avec @(*ident* : **Kind** = ... | **Kindf** = ...). Le mot clef **Kind** correspond à la déclaration de la kind de routage (resp. de rangée) des variables de routages (resp. des variables de type rangé), et **Kindf** sert à spécifier si une variable est locale ou globale, ou bien duplicable ou non dans le cas d'une variable de routage. Le reste de la syntaxe propose quelques modulations par rapport à la définition des kinds, permettant de n'en donner qu'une si l'autre est correctement calculée.

Les messages de sortie

Ces messages ne respectent pas de syntaxe pour le moment. Les messages d'erreur commencent tous par '`* Error` ', et quand aucune erreur ne survient, le programme affiche le type de l'assemblage en utilisant la syntaxe des types que nous venons de présenter.

4.2.4 Nos résultats

Nous avons tout d'abord testé notre programme sur des exemples simples, écrits à la main, afin de s'assurer qu'il fonctionnait correctement, et que les messages d'erreur correspondaient bien aux problèmes de typage rencontrés. A partir de cela, nous avons pu faire évoluer ces messages pour qu'ils donnent des informations utiles sur comment corriger les problèmes rencontrés. Nous nous sommes alors concentré sur le typage d'assemblages Click et d'une grosse configuration Dream, nommée Cosmos [30]. Le typage de ces assemblages n'avait pas pour but de vérifier si notre programme capturait bien les erreurs souhaitées, les exemples étant largement utilisés, et donc très certainement, sans erreur, mais nous a permis de juger la difficulté de typer ce genre d'assemblage, en particulier quelle est la complexité d'assigner des types aux composants primitifs, et quelles sont les modifications à apporter aux assemblages fonctionnant pour être typables. Enfin, cela nous a permis de voir si notre implantation avait une exécution rapide ou non (notre algorithme ayant une complexité exponentielle). Ces temps d'exécution sont présentés Figure 4.19, et nous semblent assez raisonnables.

Assemblage	Composants	Primitifs	Canaux	Temps (sec)
COSMOS (Dream)	439	340	662	11.714
dnsproxy (Click)	9	8	7	0.288
fromhost-tunnet (Click)	24	22	24	0.103
mazu-nat (Click)	60	56	54	0.286

Figure 4.19 – Le temps d'exécution de notre programme

Notre outil fonctionne donc relativement rapidement, est bien adapté pour capturer les erreurs de manipulation de messages, mais son utilisation au sein de Click et de Dream s'est avérée plus difficile que prévu, et ce, pour des raisons différentes pour chaque librairie. Chacune des difficultés a pu être résolu par de simples modifications des fichiers ADL, mais nous considérons ces altérations comme des solutions temporaires, en attendant d'avoir les moyens de corriger élégamment ces problèmes.

Lors du typage des assemblages Click, nous nous sommes heurtés à deux problèmes majeurs : le typage des messages, et la gestion du routage. Contrairement à ce que l'on pourrait croire à première vue, ce n'est pas le manque de structure des messages qui nous a posé problème. En effet, comme nous l'avons dit, ces messages, même s'ils ne sont considérés par la libraire comme des tableaux d'octets, correspondent néanmoins à des messages structurés, comme cela est souvent le cas dans les protocoles de communication. Ceux que nous avons rencontrés étaient des messages TCP/IP ou UDP/IP, et ne posaient donc pas de problème de structuration. Par contre, c'est avec le routage, et donc la définition des types routés que nous avons eu des difficultés. Click possède en effet grand nombre de routeurs (généralement des instanciations du composant `Classifier`). Parmi eux, certains composants routent les messages par rapport au protocole utilisé. Ces derniers sont la source du problèmes, car en encodant chaque message, non pas comme un message structuré, mais comme un message annoté par le protocole utilisé, nous pouvons donner aux composants un type correspondant exactement à leur comportement. Une telle approche ne convient pas toutefois aux routages qui se font par rapport aux adresses IP, qui sont en trop grand nombre pour être correctement traitées. Nous avions donc le choix entre encoder les protocoles comme des champs dans un message, et ainsi respecter la structure des messages, et encoder les protocoles comme des annotations, permettant ainsi un typage des routeurs, mais perdant du coup une partie de la structure stockée dans les messages. Dans nos exemples, nous avons choisi la seconde solution, le routage étant assez important dans ces assemblages. Mais nous pouvons remarquer, que pour avoir des types encore plus précis, nous pouvons utiliser à la fois les annotations de protocole ainsi que la structure correspondante. Le second problème survenu lors de la définition des types des composants primitifs de Click concerne les multiplexeurs. En effet, notre système de type impose que les routeurs et les multiplexeurs aillent de paire, les uns ôtant les annotations de routage, les autres les rajoutant. Or, cette dualité n'est pas respectée dans les assemblages Click. Par exemple, dans `dnsproxy`, un routeur possédant trois ports de sortie (pour les protocoles TCP, UDP, et les autres) envoie les messages des deux premiers ports vers un composant, qui est lui-même suivit d'un routeur dissociant les protocoles UDP et TCP. Ce dernier routeur ne correspond donc pas à un multiplexeur joignant les deux sorties du premier, ce qui rend l'assemblage non-typable. Afin de résoudre le problème, nous avons rajouté dans l'assemblage un composant qui n'existe pas en réalité, et qui joue le rôle du multiplexeur manquant.

La librairie Dream possède les mêmes types de routeurs, et peut donc causer des problèmes similaires pour la définition des types. Néanmoins, l'assemblage

que nous avons typé ne comporte pas de routeur, et ce sont d'autre difficultés que nous avons rencontrées, principalement liées à certains outils de définition des assemblages FRACTAL. Tout d'abord, rappelons que FRACTAL (et donc DREAM) est implanté en Java, et donc les librairies de composants, entre autre celle de DREAM, est mise à disposition sous forme de fichiers *.jar*. Notre premier travail fut donc d'extraire de ce fichier les différents fichier ADL et Java correspondant à la bibliothèque DREAM, les modifier de façon à y inclure les types de tous les composants de cette librairie, et de tout ré-assembler en le fichier *.jar* original. Nous nous sommes ensuite penché sur la définition des types des composants primitifs de COSMOS, et nous nous sommes heurtés à une seconde difficulté. En effet, COSMOS utilise l'outil *Fraclet* [95] permettant de construire des composants avec de la programmation par aspect, en utilisant des annotations Java. En conséquence, une grande partie des composants primitifs et des assemblages de COSMOS sont définis par des annotations Java, et non pas des fichiers ADL : il nous était impossible donc d'annoter les composants primitifs par leur type. Afin de résoudre ce problème, nous avons un peu étudié le fonctionnement de l'outil *Fraclet*. Il est apparu que la compilation Java des annotations *Fraclet* générait les fichiers ADL correspondant à la définition des composants primitifs et des assemblages. Nous avons donc généré ces fichiers, que nous avons pu annoter par nos types. Une troisième difficulté est alors apparue, et concerne l'*héritage* entre les composants. En effet, FRACTAL permet d'un composant d'hériter d'un autre, ce qui a pour conséquence que le premier possède toute les interfaces du second, facilitant ainsi la définition de composants de structure semblable. Par contre, dans notre cas où les interfaces ont des types, le composant fils hérite aussi des types des interfaces du composant père, ce qui génère dans nos assemblages des composants primitifs ayant plusieurs types et des composants composites typés. De plus, le type hérité, provenant d'un composant ayant un code différent et donc un comportement différent, ne correspond pas au comportement du composant fils. Ainsi, cet héritage défini par FRACTAL, et ne prenant en compte que la structure des composants, et non pas leur comportement, n'est pas valide du point de vu de nos type. Nous avons résolu ce problème en remplaçant tous les héritages entre composants par les interfaces correspondantes, que nous avons ainsi pu typer librement.

4.3 Conclusion

Dans ce chapitre, nous avons présenté un algorithme d'inférence pour le système de type décrit dans le chapitre précédent, ainsi que l'implantation que nous en avons faites. Cette dernière est structurées en trois modules différents, afin d'avoir une plate-forme de typage souple et extensible : il suffit en effet de rajouter un nouveau module de traduction pour permettre la vérification de type dans un nouveau modèle à composant. Du fait des règles de duplication, notre algorithme a une complexité exponentielle en la taille de l'assemblage à typer, mais en pratique, il garde un temps d'exécution raisonnable, cette composante exponentielle étant principalement causée par des chaînes de rou-

tage : ces chaînes restent de taille très limitée dans les assemblages courants. Enfin, ce travail nous a permis de voir que même si notre outil de typage capture bien les erreurs de manipulation de message, il reste néanmoins quelques améliorations à lui apporter, aussi bien du côté du système de type que de son implantation elle-même, pour le rendre aisément utilisable par Click et Dream. En effet, le système de type possède des types routés, qui permettent d'avoir de l'inférence de type en présence de composant de routage, ce qui n'était pas possible avec notre première approche, mais aussi qui ne sont pas assez souples pour typer de manière intuitive des assemblages comportant de tels routeurs. Une généralisation de ces types serait donc grandement appréciable. Une première possibilité serait d'assouplir la structure d'arbre binaire de ces types, en les rendant n-aires, ce qui pourrait encore être étendu par leur inclusion directement dans les types de messages. De cette façon, nous pourrions avoir des messages qui serait des composants, permettant ainsi une certaine forme de reconfiguration dynamique telle que dans le calcul HO-π. Une dernière possibilité d'amélioration, que nous n'avons pas encore vraiment étudié, serait d'étendre notre système de type par des types *comportementaux* tels qu'utilisé par [60, 61, 25] pour capturer les inter-blocages. Dans notre cadre d'utilisation, ce genre de types permettraient de typer avec plus de précision les reconfigurations durant l'exécution des assemblages. Du côté de notre implantation, les deux modules spécifiques à Click et à Dream pourrait être étendus. Dans le cas de Click, deux améliorations nous semblent particulièrement importantes :

1. Ne pas typer les parties de bas niveau des assemblages. Comme nous l'avons fait souvent remarquer, les messages de Click sont des messages structurés, sauf dans certaines rares circonstances. Un exemple qui nous a été présenté est le protocole de *slicing*, où les messages sont découpés en plusieurs paquets avant d'être envoyé à un autre composant, et recomposé par la suite. Dans ce type de protocole, les messages perdent temporairement leur structure pour ne devenir que des tableaux d'octets, ce qui pose un gros problème de typage. Il serait donc intéressant de pouvoir spécifier que certaines partie d'un assemblage est de bas niveau, que son comportement interne n'est alors pas typable, tout en permettant de définir les types des ports d'entrée et de sortie de cette partie, afin de pouvoir vérifier correctement le type du reste de l'assemblage.
2. Pouvoir typer les paramètres des composants. Comme nous l'avons vu précédemment, la plupart des composants Click ont des paramètres qui influent sur leur comportement, les exemples les plus flagrant étant les composants `InfiniteSource` dont un des paramètres est le message envoyé par le composant (et donc qui influe directement sur son type), ou `Classifier` dont les paramètres sont des motifs de routage. Permettre d'annoter par des types les messages et les motifs, afin de générer un typage précis des composants, nommé ou non, serait un apport appréciable à notre module qui reste pour le moment relativement simpliste. Cet ajout demanderait cependant la modification de la syntaxe de Click par l'ajout d'annotations de types, et la définition dans le module d'un premier ou-

til d'inférence, calculant à partir du nom d'un composant de base de la librairie CLICK et de ses paramètre, le type du composant résultant.

Finalement, il serait intéressant aussi de rajouter quelques outils autour de notre module pour DREAM afin de résoudre les deux difficultés majeures que nous avons rencontré lors du typage de l'assemblage COSMOS. Ainsi, nous pourrions étendre l'outil *Fraclet* par de nouvelles annotations, permettant de spécifier directement dans les fichiers Java, et avec les annotations concernant les composants, les types des interfaces des composants primitifs. Enfin, du côté de l'héritage entre composant, nous pensons qu'une simple modification de la syntaxe XML des fichiers ADL, permettant la re-définition du type des interfaces, et un traitement adapté par notre module pourrait facilement résoudre le problème.

Chapitre 5

Le langage Oz/K

Comme nous l'avons vu jusqu'à présent, la partie principale de nos travaux consiste en la définition de calculs de processus relativement simples, la création de systèmes de type associés et à l'étude de leurs différentes propriétés. Néanmoins, en parallèle à ces travaux, nous avons aussi développé un langage noyau [1] nommé *Oz/K*. La motivation principale pour la création de ce langage est l'étude des primitives permettant la mobilité de code telle qu'elle est définie par exemple dans les Ambients, le modèle Fractal ou le kell-calcul. En effet, le but de notre travail de thèse est de définir un système de type pour un calcul de processus comportant de telles primitives. Or, parmis tous les travaux qui ont servis de base à notre étude [7, 17, 23, 26, 98, 52, 40, 70, 9, 53, dfusion], il est nous a été impossible de voir émerger un consensus concernant les opérateurs à utiliser pour mettre en place la mobilité de code. Nous avons donc repris ces différents travaux et essayé de trouver par nous-même une solution acceptable aux difficultés que soulève la définition de ces opérateurs. Notons que le but de cette étude n'est pas de trouver un langage qui à la fois comporte des primitives de *reconfiguration* et puisse être typable : nous ne nous intéressons ici qu'aux primitives et à leur sémantique, avant de pouvoir tester une méthode permettant de les typer.

Nous proposons dans ce chapitre une évolution des structures du kell-calcul avec partage [52] comme base de reconfiguration de notre langage. Nous avons choisi ce calcul comme point de départ pour notre approche pour trois raisons principales : (i) comme quelques autres calculs de processus, il permet d'encapsuler des composants dans d'autres composants, ce qui permet de structurer finement un programme ; (ii) il possède un opérateur de *passivation* permettant de suspendre l'exécution d'un composant et de stocker son état pour pouvoir soit le détruire, soit le relancer plus tard et à dans un autre environnement ; et (iii)

1. Le terme *noyau* signifie que le langage ne met à disposition que les primitives minimales pour la déclaration d'une sémantique cohérente.Un tel langage est conçu pour être le coeur d'un langage plus évolué contenant des opérateurs plus complexes se traduisant en une séquence de commandes primitives : ces opérateurs sont donc considérées comme du sucre syntaxique au dessus de notre langage de base.

il possède une capacité de partage permettant des communications entre composants distant (c'est à dire, séparés par plusieurs frontières de composants). Néanmoins, ce calcul possède toutefois quelques défauts, le principal concernant le partage. Ce dernier est en effet encodé dans ce calcul via des sortes de pointeurs que les processus peuvent s'échanger. Ces pointeurs référencent des composants distants et permettent de simuler leur présence en local. Ainsi, il est possible, une fois que l'on possède un pointeur, de l'envoyer à n'importe quel processus sans aucun contrôle, ouvrant ainsi d'énormes failles de sécurité. Dans notre approche, nous avons alors remplacé le partage par un système de porte très similaire à celui présenté lors de notre deuxième calcul, Chapitre 3. Un autre défaut mineur du kell-calcul concerne la passivation. Cette opération est en effet très intéressante pour la reconfiguration, car elle permet, entre autre, de déplacer un composant durant son exécution. Par contre, il est impossible dans ce calcul de configurer un composant passivé en vue de son redéploiement dans un environement différent de celui dans lequel il s'exécutait à l'origine. C'est pourquoi nous testons dans cette approche quelques primitives permettant de manipuler des composants passivés : nous avons par exemple une commande permettant de modifier les canaux de communication qu'utilisait un composant, afin de pouvoir le reconnecter dans son nouveau site d'éxécution.

Nous avons de plus profité de ce travail pour intégrer nos primitives de reconfiguration dans un langage noyau. Cette approche a l'avantage, par rapport à la simple définition d'un calcul ne comportant que les primitives étudiées, de montrer quelles sont les contraintes qu'impose l'utilisation de ces opérations de mobilité de code dans des paradigmes plus classiques, comme la programmation impérative ou fonctionnelle. La structure de composant offre de plus des capacités de concurrence, d'isolation et d'abstraction, qu'il est intéressant de comparer aux constructions plus classiques, comme les processus légers et les modules. Afin de pouvoir tester l'intérêt de l'apport de nos primitives dans un langage de programmation réel, nous n'avons pas défini nous même un langage noyau, mais nous en avons utilisé un existant : Oz [45, 110, Chapitre 13]. Le choix de ce langage comme base pour nos travaux n'est pas anodin. Tout d'abord, Oz est un langage noyau, formellement défini, intégrant dans une syntaxe minimale des constructions telles que les modules ou les processus légers. Il est alors possible de comparer ces constructions avec nos composants, ce qui est difficile à faire pour des langages plus complexes, et même impossible dans des langages comme le C ou le Java, où les primitives de concurrence n'ont pas une sémantique précise. Oz est de plus un langage multi-paradigme. En effet, ce langage est construit sur un coeur de programmation logique d'ordre supérieur (les procédures pouvant être utilisées comme paramètres), auquel ont été rajoutés plusieurs primitives pour y intégrer de la programmation impérative, concurrente et aussi paresseuse. Le paradigme fonctionnel est lui considéré comme du sucre syntaxique au dessus de l'appel de procédure. Ainsi avec Oz, nous pouvons tester l'intégration de nos primitives dans un large panel de paradigmes et étudier les possibles conflits ou similarités entre des constructions d'un langage classique et notre approche pour la mobilité de code. Une autre particularité intéressante de ce langage est qu'il utilise une sémantique très précise, proche

de celle d'une machine virtuelle. Notre approche consiste alors en la définition de nos opérateurs dans la même sémantique, ce qui permet de bien comprendre leur fonctionnement, leur complexité ainsi que leurs limitations. Finalement, le langage Oz ne comporte pas de sémantique distribuée (seulement concurrente), et notre approche est une étude intéressante pour l'extension du langage. Notons qu'il existe déjà une extension de Oz comportant une sémantique distribuée : le langage Mozart [29]. Le but de cette extension est de fournir un langage de programmation complet au-dessus de Oz, tout en permettant l'exécution répartie des programmes développés dans ce langage. Ainsi, Mozart possède un environnement d'exécution distribué, et permet d'assigner des politique de placement aux différents éléments manipulés par un programme Oz. Il est toutefois possible dans ce langage de laisser l'environnement d'exécution répartir librement les éléments d'un programme sur toute l'architecture matérielle disponible, et ainsi obtenir ce qui est appelé *la transparence réseau* dans [27] : c'est à dire de permettre à un programmeur de définir un logiciel réparti sans qu'il ait à se soucier de son mode de répartition, et ce, tant qu'il n'y a pas de défaillance de l'architecture. Néanmoins, l'approche de ce langage se limite à la distribution, alors que nous nous intéressons ici à la mobilité de code : il est donc intéressant de comparer nos deux approches, en étudiant à la fois les capacités de placement de nos deux langages, leurs différentes complexité d'implémentation, ainsi que la mise en place de la transparence réseau.

5.1 La syntaxe

Le langage Oz/K est construit en conservant intégralement le langage Oz comme il est décrit dans le chapitre 13 de [110], et en rajoutant à ce langage de base quelques constructions lui permettant de manipuler des structures à composant. Nous présentation de la syntaxe de notre syntaxe reprend cette structuration, en introduisant tout d'abord Oz, ses différents principes et constructions de base. Ensuite vient l'extension que nous étudions ici, que nous appelons K, car issue du kell-calcul.

5.1.1 La partie Oz du langage

Oz [49, 96, 110] est un langage complet, et malgré sa définition en un langage noyau, comporte beaucoup de constructions. Une présentation extensive de ce langage demanderait donc plusieurs chapitres, ce qui n'est pas le but de ce document. Nous nous contentons ici de ne présenter que les points principaux du langage, c'est à dire son coeur fonctionnel ainsi que sa construction de processus pour la concurrence. Ces différents éléments suffisent en effet à créer des exemples illustrant les différentes propriétés de notre extension. Nous reprenons ici les conventions syntaxique de Oz en distinguant trois catégories de mots dans le langage :

1. Les *variables* forment tous les éléments manipulés par un programme Oz possédant une valeur. Elle sont écrites par des mots commençant par une

$\mathcal{S}$	::=		Instruction
		`skip`	L'instruction vide
	|	`local X1 ... Xn in` $\mathcal{S}$ `end`	Création de variables
	|	`X =` $\mathcal{P}$	Affectation
	|	`proc{P X1... Xn}` $\mathcal{S}$ `end`	Création de procédure
	|	`{P X1... Xn}`	Appel de procédure
	|	`if X then` $\mathcal{S}1$ `else` $\mathcal{S}2$ `end`	Conditionnelle
	|	`case X of` $\mathcal{P}$ `then` $\mathcal{S}1$ `else` $\mathcal{S}2$ `end`	Filtrage de motif
	|	$\mathcal{S}1\ \mathcal{S}2$	Séquence
	|	`thread{X}` $\mathcal{S}$ `end`	Création de processus
$\mathcal{P}$	::=		Motifs
		`X` | *'atome'*	Variables et atomes
	|	`l(X1: X1' ... Xn: Xn')`	Message structuré

FIGURE 5.1 – La syntaxe OZ

majuscule.

2. Les *atomes* sont les valeurs de base du langage, au même titre que les entiers ou les chaînes de caractères et servent par exemple à noter les noms des champs d'un message structuré. Ils consistent en une chaîne de caractère en italique placée entre apostrophe, comme dans *'atome'*.
3. Finalement, les mots clefs du langage sont écrit normalement et ne commencent pas par une majuscule.

Les concepteurs de OZ préconisent aussi l'utilisation de certaines facilités d'écriture qui permettent de contourner plusieurs lourdeurs de la syntaxe du langage. En effet, cette syntaxe, présentée Figure 5.1, fait un usage intensif des variables qui forment le passage obligé pour pouvoir donner un paramètre à une procédure, ou pour définir la valeur d'un champ d'un message structuré. Par exemple, la définition de la valeur de la variable `Res`, s'écrivant classiquement `Res = 5 * (1 + 2)` en C, ML, ou d'autres langages classiques, s'écrit en OZ :

```
local Tmp1 Tmp2 Tmp3 Tmp4 in
  Tmp1 = 1
  Tmp2 = 2
  Tmp3 = 5
  {+ Tmp1 Tmp2 Tmp3}
  {* Tmp3 Tmp4 Res}
```

Afin d'éviter cette lourdeur d'écriture, nous nous permettons d'utiliser directement des valeurs dans un appel de procédure, et de considérer, quand cela est

possible, les procédures comme des fonctions dont leur résultat est donné par leur dernier paramètre. Par exemple, dans {`+ 1 2 Tmp3`}, `Tmp3` est le résultat de l'addition de 1 avec 2, et nous définissons alors {`+ 1 2`} comme étant la valeur 3. De plus, lorsqu'une expression a une valeur, nous pouvons la donner à une variable, comme dans `Tmp3 =` {`+ 1 2`}. Enfin, en utilisant la notation infixe pour les opérateurs classiques sur les entiers, l'expression `Res = 5 * (1 + 2)` devient valide dans notre syntaxe étendue. Finalement, nous avons la notation _ qui correspond à une variable anonyme, généralement mise à la place du résultat inutile d'une procédure.

Après ces quelques considérations syntaxiques sur le langage, nous passons à la présentation dans les paragraphes suivant des différentes constructions du langage.

Les variables et les valeurs. Le langage possède des variables logiques (ou simplement variables) qui peuvent être *liées* ou non. Un variable non liée n'a pas de valeur, alors qu'une liée réfère à une valeur non modifiable qui peut être un atome *'valeur'* ou un message structuré. Les messages structurés de Oz `l(X1:X1' ... Xn:Xn')` sont plus généraux que ceux que nous avons vu jusqu'à présent : ils consistent en une liste d'association entre des variables Xi et Xi' (où Xi correspond généralement à un atome du langage définissant le champ du message, et Xi' donnant la valeur de ce champ), et possède aussi un nom `l`. Néanmoins, les fonctions de manipulation de ces messages sont similaires à celles que nous avons vues, avec l'accès à un champ, sa délétion ou son rajout. Ces fonctions sont codées comme des procédures du langage, mais à des fins de présentation, nous noterons leur utilisation comme nous l'avons dans dans nos deux calculs précédents.

Oz étant un langage déclaratif, les nouvelles variables sont introduites par une déclaration de la forme `local X1 ... Xn in` $\mathcal{S}$ `end` où $\mathcal{S}$ est un programme quelconque et `X1 ... Xn` sont les variables fraîchement introduites. Par défaut, les variables fraîches ne sont pas liées et n'ont donc pas de valeur. Pour lier une variable, nous utilisons l'expression `X =` $\mathcal{P}$, où `X` est la variable que l'on lie, et $\mathcal{P}$ représente la valeur donnée à `X`. Lorsque le motif $\mathcal{P}$ est un atome ou un message structuré `l(X1:X1' ... Xn:Xn')`, `X` est directement liée à cette valeur, et lorsque ce motif est une variable `Y`, `X` est liée à la valeur de `Y`. Notons toutefois que si `Y` n'est pas liée lors de l'exécution de l'expression `X=Y`, `X` reste non liée : les deux variables prendrons la même valeur dès qu'une affectation sera appliquée soit à `X` soit à `Y`. Notons aussi que lorsque `X` possède déjà une valeur, l'expression d'affectation unifie les deux valeurs données [2].

Les procédures. L'expression `proc {P X1 ... Xn} in` $\mathcal{S}$ `end` crée une nouvelle procédure, où `P` est la variable contenant la procédure, `X1 ... Xn` sont

2. Cette sémantique d'unification implicite est propre à Oz et n'est pas reprise dans Oz/K, car une simple procédure d'affectation peut dans ce cas impliquer un calcul assez conséquent.Nous avons donc choisi dans OzK de ne pas permettre l'unification via l'affectation, mais d'avoir une nouvelle procédure dédiée à l'unification

ses paramètres et $\mathcal{S}$ son code interne. Comme cela est généralement le cas dans la programmation logique, les paramètres servent à la fois pour les entrées et les sorties, et et rendent les procédures très polyvalentes, pouvant ainsi servir à la fois de prédicats, de fonctions et de simples procédures sans effets de bords. Par exemple, la procédure `proc {P X} in local X1 X2 in X=l(a1:X1 a2:X2) end` peut être considérée soit (i) comme une fonction prenant en paramètre une variable non liée et lui donnant la forme d'un message structuré ; soit (ii) comme un prédicat vérifiant que la variable *liée* en paramètre est bien un message structuré avec les champs `a1` et `a2`. Un appel de procédure s'écrit `{P A1 ... An}` où `P` est la procédure exécutée et les `Ai` sont les paramètres de l'appel. Notons que les procédures sont liés à des variables comme les autres valeurs, et sont donc des paramètres valides d'une autre procédure.

Les conditionnelles. Notre présentation du langage inclue deux expressions permettant d'exécuter du code en fonction de certaines données. Nous avons la construction conditionnelle classique`if X then` $\mathcal{S}1$ `else` $\mathcal{S}2$ `end`, qui exécute $\mathcal{S}1$ si `X` est liée à *'true'* ou $\mathcal{S}2$ lorsque `X` est liée à *'false'*. La construction de filtrage de motif `case X of` $\mathcal{P}$ `then` $\mathcal{S}1$ `else` $\mathcal{S}2$ `end` teste si `X` est unifiable avec $\mathcal{P}$, auquel cas, elle exécute $\mathcal{S}1$, ou sinon, ce sera $\mathcal{S}2$ qui sera lancé. Par exemple, si la variable `X` est liée au message structuré `rec(a:X1 b:X2)`, alors l'expression

```
case X of rec(a:Y1 b:Y2)
  then {P Y1 Y2} else skip end
```

se réduira en `{P X1 X2}` (les variables du motifs `Y1` et `Y2` sont respectivement liées aux variables `X1` et `X2` durant l'unification).

La concurrence. Par défaut, Oz est un langage séquentiel : $\mathcal{S}1$ $\mathcal{S}2$ signifie que $\mathcal{S}1$ sera exécuté avant $\mathcal{S}2$. Il existe néanmoins une commande pour créer un nouveau flux d'exécution : `thread{T}` $\mathcal{S}$ `end` crée un nouveau processus (ou *thread*) qui exécute la série d'instructions $\mathcal{S}$ en parallèle aux autres processus du programme. Notons que la variable `T` est liée au statut du thread lors de sa création. Cela permet par exemple de savoir quand le processus a terminé son calcul, ou si une erreur s'est produite.

5.1.2 La partie K du langage

La partie K du langage consiste en l'introduction d'une nouvelle structure de composant dans le langage, que nous nommons *kell* par héritage du kell-calcul. Cette structure vient avec deux ensembles d'opérateurs permettant de manipuler les deux sémantiques indépendantes de cette structure.

La sémantique d'isolation. Les kells forment des frontières entre les différentes parties d'un programme, permettant ainsi de structurer, d'abstraire, et de contrôler les interactions entre les différents constituants un logiciel. Par défaut, les capacités de communication d'un kell sont les même que dans [98] : il ne peut

communiquer qu'avec ses composants proches, c'est à dire celui qui le contient (son *père*) et ceux qu'il contient (ses *fils*). Il est toutefois assez facile de voir que cette isolation forte est extrêmement pénalisante lorsque l'on veut communiquer avec des composants distants : dans le kell-calcul, il faut encoder une telle communication distante entre les kells a et b par une chaîne de transmetteurs qui font passer les messages d'un composant à un autre, au travers de toute la structure séparant les deux kells a et b. C'est pourquoi nous avons introduit dans notre extension des *portes* très similaires à celles que nous avons présentées dans le Chapitre 3. Leur principe est similaire : elles ouvrent la frontière des composants pour certains canaux de communication, et les communications utilisant ces canaux ne sont alors plus limitées à être entre composants père et fils. Néanmoins, quelques modifications ont été apportées sur nos portes : dans le chapitre 3, les communications étaient asynchrones et les messages pouvaient franchir les frontières où la porte était ouverte. Ici, la communication est synchrone et, afin de permettre une communication à la kell-calcul même lorsque les portes sont fermées, nous autorisons à toute communication de franchir au plus une frontière qui n'est pas ouverte. Nous donnons de plus la capacité aux composants de contrôler l'ouverture et la fermeture des portes de leurs composants fils, rendant dynamique la frontière qui dans notre travail précédent était statique. Enfin, les portes ne sont pas de simples noms ici, comme des atomes, mais des structures indépendantes qu'il faut créer. Ainsi, en créant une porte personnelle, on peut s'assurer que seul des entités de confiance peuvent recevoir des messages sur ce canal sécurisé où peuvent circuler des données sensibles.

La sémantique de mobilité. Les kells forment aussi des entités d'exécution indépendantes que l'on stopper, dupliquer et déplacer. Le principe fondamental de ces différentes opérations est la *passivation* [98]. La passivation est une commande permettant de prendre un kell et de le détruire en sauvegardant son état dans une valeur spécifique. Ainsi, il devient possible de détruire cette valeur, auquel cas le composant est définitivement perdu, ou de l'envoyer vers un autre kell qui pourra relancer son exécution. Plus précisément, la partie K de notre langage contient une procédure de base, `Pack K X`, qui prend le contenu du kell `K`, et le passive dans une valeur liée à `X`. Notons que la structure du composant `K` est toujours présente, bien que vide, afin que les accès ultérieurs à la variable `K` ne causent pas d'erreur. Une telle structure peut de plus être utilisée de nouveau en y injectant un nouvel ensemble de commande via une procédure de remplacement de kell. Les valeurs issues de la passivation d'un composant, que nous nommons *valeurs packées*, sont sujettes à deux types d'opérations : des opérations de modifications et une opération de déploiement. Cette dernière prend le contenu de la valeur (c'est à dire tout les processus et les kells fils) et les places dans le kell courant, prêt à être exécuté à nouveau. Néanmoins afin que toutes ces structures puissent s'exécuter convenablement, il peut être nécessaire des les adapter au préalable à leur nouvel environnement. C'est pourquoi nous offrons la possibilité de modifier les valeurs packées, permettant ainsi de mettre à jour certaines références et données en vu de leur future déploiement. L'en-

semble de ces manipulations sur les composants composent un *cycle de vie*, que nous résumons Figure 5.2. Finalement, un composant possède une variable de

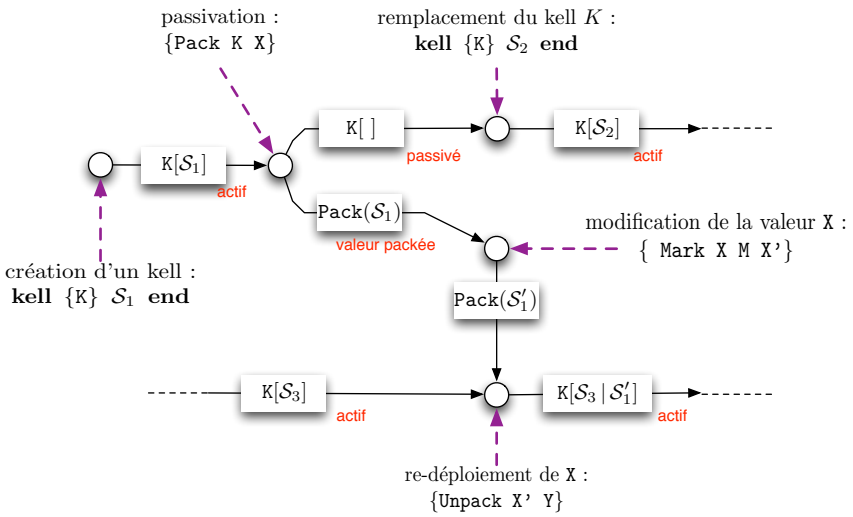


FIGURE 5.2 – Le cycle de vie d'un composant

statut qui est non liée lors de l'exécution normale du kell, et prend une variable lors de l'arrêt du composant : soit *'packed'* si l'arrêt est dû à une commande de passivation, soit *'failed'* si l'arrêt est causé par une erreur d'exécution.

Après cette brève introduction, nous présentons la syntaxe de notre extension, qui se limite à la définition d'une structure de kell, ainsi que différentes procédures de base manipulant portes, kells et valeurs packées.

Nous rajoutons au constructions précédentes trois nouveaux éléments : les *kells*, les *portes* et la *passivation*. La syntaxe de ces commandes pour manipuler ces objets est donnée Figure 5.3, où les termes `K`, `X`, `Y`, `Z`, `G` sont de simples variables.

Les kells. Ces composants sont créés avec la commande `kell`{`K`} $\mathcal{S}$ `end` où `K` est la variable liée au kell créé, et $\mathcal{S}$ le code exécuté dans un nouveau processus par le composant. De plus, la commande {`Status K X`} permet de monitorer les composants Oz/K, et de vérifier leur état de fonctionnement. Il est ainsi assez facile de détecter une erreur d'exécution dans un kell, et de lancer le code de gestion d'erreur approprié. Toute cette gestion est expliquée plus en détail dans la Partie 5.2. Une des problématique de la structure de composant vient de sa capacité à être passivé, son exécution momentanément suspendue et reprise dans un environnement pouvant être totalement différent de celui

$\mathcal{S}$	::=	...	
	|	`kell{K}` $\mathcal{S}$ `end`	Création d'un kell
	|	`{NewGate X}`	Création de porte
	|	`{Send G X}`	Envoi d'un message
	|	`{Receive G X}`	Recpetion d'un message
	|	`{Open K G}`	Ouvre la porte `G`
	|	`{Close K G}`	Ferme la porte `G`
	|	`{Pack K X}`	Stoppe l'exécution d'un kell
	|	`{Unpack X Y}`	Continue l'execution d'un kell
	|	`{Mark X Y Z}`	Remplacement des portes d'un kell stoppé
	|	`{Status K X}`	Lecture du status d'un kell

FIGURE 5.3 – La syntaxe K

d'origine. Une telle opération peut alors être source d'incohérences dans l'état du système lorsque l'exécution du composant passivé dépend de son environnement. C'est pourquoi nous imposons dans notre approche une isolation forte de nos composants qui ne peuvent communiquer qu'au travers des portes. Une première contrainte que nous avons afin d'assurer cette isolation est que le code $\mathcal{S}$ ne doit pas compoter de référence à des variables non liées (à part `K` qui sera liée durant la création du kell). En effet, si $\mathcal{S}$ comporte des variables non liées provenant de l'extérieur du composant, cela impliquerait que le composant et son environnement peuvent communiquer via ces variables, empêchant ainsi tout contrôle des communications associés aux portes. Notons que cette contrainte n'est pas difficile à implémenter, car elle demande des structures similaires à celles utilisées pour l'unification, qui est déjà présente dans Oz.

Les portes comme outils de communications. La commande {`NewGate G`} crée une nouvelle porte (qui est une valeur), et la lie à la variable `G`. Cette variable peut ensuite être utilisée pour les envois de messages ({`Send G X`}) et leur réception ({`Receive G X`}). Les échanges de messages se font par rendez-vous atomiques, comme pour le π-calcul synchrone : les envois et réceptions de message sont bloquantes, et ne réussissent que lorsqu'une réception ou un envoi correspondant a lieu. Cette sémantique, avec l'isolation des composants, permettent d'assurer le bon fonctionnement [3] de la commande `Pack` qui stoppe un composant et sauvegarde son état dans une variable. Notons que l'isolation

3. En effet, avec une communication asynchrone telle que celle que nous avons dans la Chapitre 3, la commande `Pack` peut capturer un message en cours de transmission et ainsi causer un état incohérent dans le système.

des composants est assurée par le fait que leur code ne comporte pas de variable libre pouvant être utilisée comme moyen de communication vers le reste du programme. Dans le paragraphe précédent, nous avons donné une contrainte dans ce sens, spécifiant que le code initial d'un composant ne pouvait comporter de variable non liée (à part le nom du composant lui-même). Et afin de garder cette propriété au cours de l'exécution du composant, nous contraignons les portes à n'échanger que des valeurs *strictes*, c'est à dire ne comportant aucune variable non liée. Ces deux contraintes seront décrites en détails dans la partie discutant de la sémantique opérationnelle du langage.

Ainsi, les seules primitives de communication entre composants sont les portes, et cette communication est de plus contrôlée par les frontières des composants. Cette nouvelle contrainte a pour but de limiter les capacités d'interactions entre composants, et ainsi de sécuriser des portions de programmes traitant des données sensibles, ou ayant des fonctions vitales pour l'ensemble de l'application (et qui ne doivent alors être corrompues). Par défaut, les communications directe entre un composant et ses fils, qui ne traversent ainsi qu'une barrière d'isolation, sont les seules à être autorisées. Cette capacité peut être étendue, et réduite par la suite, grâce au primitives `Open` et `Close` qui ouvrent et ferment des portes sur les frontières des composants : un composant père peut ainsi ouvrir la porte `G` d'un composant fils `K` avec la commande {`Open K G`}, ou la refermer avec {`Close K G`}. Avec ce principe de porte, nous pouvons étendre nos capacités initiale, et, lors d'une communication, un message envoyé sur la porte `G` peut franchir toute frontière ayant cette porte ouverte, plus une unique frontière avec cette porte de fermée. Les communications distantes, traversant plusieurs limites de composants, doivent être autorisée par l'ouverture des portes servant à la communication. Supposons par exemple que deux composants frères `K1` et `K2` veulent communiquer sur la porte `G`. Leur échange traverse leurs deux frontières, et il faut donc ouvrir la porte `G` pour `K1` ou `K2` pour qu'ils puissent communiquer. Cette opération est effectuée par le composant père `K`, avec la commande {`Open K1 G`} ou {`Open K2 G`}. Enfin, les opérations d'ouverture et de fermeture de porte sont étendues par l'atome *'all'* permettant de spécifier tout les sous-composants d'un composant père, ou toutes les portes. Ainsi, {`Open K 'all'`} ouvre toute les portes du composant `K`, et {`Close 'all' G`} ferme la porte `G` de tous les sous-composants du kell courant.

La passivation. La commande {`Pack K V`} stoppe le composant `K` ainsi que tout ses sous-composants, et stocke son état (c'est à dire les différents processus qu'il exécutait ainsi que les valeurs qu'il utilisait) dans une valeur donnée à la variable `V`. Cet valeur est appelé une *valeur packée*, et supporte quelque opération de modification via la commande `{Mark V1 R V2}`. Dans cette opération, `V1` est la valeur packée à modifier, `R` est la commande, et `V2` est le résultat. Il existe trois formes de commandes : (i) `gate(G1 G2)` remplace les références vers la porte `G1` par des références vers `G2` dans `V2`; (ii) `proc(P Q` remplace la procédure `P` par `Q`; et (iii) `top(K)` remplace le composant englobant de la valeur packée par `K`. De plus, les modifications apportées à la valeur packée sont *marquées*, ce

qui sera pris en compte lors du déploiement de la valeur packée en un nouveau composant s'exécutant. En effet, les valeurs packées peuvent être utilisée avec la commande `{Unpack V R}`, qui crée un nouveau composant similaire à celui stocké dans `V`, et l'exécute. Afin qu'il n'y ait aucun conflit entre les variables utilisée par le programme, et celles venant du composant déployé, la procédure `Unpack` renomme tous les objets contenu dans la valeur packée, sauf ceux qui ont été marquées (et qui correspondent donc à des objets voulus par le programmeur). Ces nouveaux noms sont aussi mis à disposition dans `R`, qui est un message structuré, avec comme champs, les anciens noms des différents objets de la valeur packée, et comme valeurs, les nouveaux noms correspondant.

5.2 La sémantique opérationnelle du langage

Contrairement aux langages purement fonctionnels, ou aux calculs de processus tels que nous en avons définis précédemment, notre langage Oz/K possède des commandes de création d'éléments, comme des processus ou des kells, qui peuvent être utilisé plus tard dans le programme. Afin de se souvenir de tout ces éléments créés, nous suivons la même approche que les concepteurs de Oz en définissant notre sémantique opérationnelle à l'aide de *stores* (notés σ) qui stockent tout ces éléments en les associe à des noms unique pour faciliter leur utilisation ultérieure. De manière similaire, la sémantique concurrente des threads et des kells est mise en place par des *tâches* (notées $\mathcal{T}$) qui associent des termes du langage aux différentes structures d'exécution en parallèle. Ainsi, la sémantique opérationnelle du langage Oz/K s'exprime sous la forme d'une relation de réduction $\triangleright$ entre des paires $(\sigma, \mathcal{T})$. Une telle paire est nommée *structure d'exécution*.

Nous supposons par la suite donné des ensembles dénombrables et disjoints `Ident`, `Var`, `Nom` et `Atome`.

Les stores. Ils sont décrits par la grammaire donée Figure 5.4. Un store consiste en un ensemble de déclarations simples, combinées par la construction $\sigma \wedge \sigma$. Ces déclaration peuvent être de différents types : nous avons tout d'abord les décarations de variable x, ainsi que les liaisons de variables $x = V$, où V est une valeur. Viennent ensuite les éléments nommés $\xi : E$, et quelques autres expressions que nous ne détaillons pas en entier dans ce document. Nous donnons simplement deux exemples, qui sont : $\texttt{in}(\xi_1, \xi_2)$ spécifie que le composant ξ_2 est fils de ξ_1 ; et un processus léger ξ_2 s'exécutant dans un kell ξ_1 est noté $\texttt{inth}(\xi_1, \xi_2)$. Nous avons cin q types de valeurs dans notre store : les variables x (pour avoir des contraintes d'égalité entre des variables), les noms ξ (qui correspondent à des éléments de haut niveau, comme des procédures ou des kells), les messages structurés $l(a_1 : x_1, \ldots, a_n : x_n)$, les erreurs d'unification $\texttt{failed}(x)$ et les valeurs packées $\texttt{pack}(\xi, \mathcal{T}, \sigma, M)$. Dans ces dernières, ξ correspond au nom du kell qui a été passivé, $\mathcal{T}$ est la tâche qu'il était entrain d'exécuter, σ est l'ensemble des données utilisées par le kell, et M est

σ	$::=$	x	Une variable
	$\mid$	$x = V$	Une variable liée
	$\mid$	$\xi : E$	Un élément nommé
	$\mid$	$\texttt{in}(\xi_1, \xi_2)$	Un composant fils
	$\mid$	$\texttt{inth}(\xi_1, \xi_2)$	Un thread dans un kell
	$\mid$	$\sigma \wedge \sigma$	
V	$::=$	x	Une variable
	$\mid$	u	Une valeur de base
	$\mid$	ξ	Un nom
	$\mid$	$l(a_1 : x_1, \ldots, a_n : x_n)$	Un message structuré
	$\mid$	$\texttt{failed}(x)$	Une valeur d'erreur
	$\mid$	$\texttt{pack}(\xi, \mathcal{T}, \sigma, M)$	Une valeur packée
E	$::=$		Un élément
		$\texttt{proc}\{\$\ X_1 \ldots X_n\}\mathcal{S}\ \texttt{end}$	Une procédure
	$\mid$	$\texttt{thread}(x)$	Un processus léger
	$\mid$	$\texttt{kell}(O, x)$	Un kell
	$\mid$	$\texttt{gate}$	Une porte
O	$::=$	$\{\xi_1 \cdot \xi_2\}$	Ouverture de portes
	$\mid$	$\xi \cdot \texttt{G} \mid \texttt{K} \cdot \xi \mid \texttt{K} \cdot \texttt{G}$	
	$\mid$	$O \cup O \mid O \setminus O$	

FIGURE 5.4 – La grammaire des stores

l'ensemble des noms qui ont été marqués par la commande `Mark`. Les éléments associés aux noms du store peuvent être de quatre types différents : nous avons tout d'abord les procédures $\texttt{proc}\{\$\ X_1 \ldots X_n\}\mathcal{S}\ \texttt{end}$ avec $X_1 \ldots X_n$ ses paramètres et $\mathcal{S}$ son code ; Les procéssus léger sont notés $\texttt{thread}(x)$, où x est la variable contenant le *status* du processus, ce status n'étant lié qu'à la terminaison du processus, et signalant à ce moment si cet arrêt est normal, ou causé par une erreur survenue durant son exécution ; les composants ont une structure similaire à laquelle sont rajoutées les informations O concernant les ouvertures de portes : $\texttt{kell}(O, x)$. Ces informations sont construites comme des ensembles finis ou co-finis de déclaration $\{\xi_1 \cdot \xi_2\}$, qui signifie que la porte ξ_2 est ouverte pour le composant fils ξ_1. Le terme spécial `K` (resp. `G`) est utilisée pour construits les ensembles infinis, et correspond à tout les composants fils du composant courant (resp. toutes les portes). Les notations $O \cup O$ et $O \setminus O$ ont la signification usuelle des opérations sur les ensembles. Dans la suite, et afin de simplifier la présentation de la sémantique de notre langage, nous ne ferons pas de distinction entre la syntaxe de O, et l'ensemble qu'il représente.

Notons que dans le cas d'un kell, la variable de status peut aussi prendre

la valeur *packed*, signifiant que le composant a été passivé. Enfin, un nom peut correspondre à une porte `gate`, qui ne contient aucune donnée annexe.

Les tâches. Ces éléments sont les structures d'exécution de notre sémantique opérationnelle. De manière similaire aux calculs distribués, ces structures forment les différents processus en parallèle du programme en cours d'exécution. Ces processus contiennent le code qu'ils exécutent, qui se présente comme une extension de la syntaxe présentée Figure 5.1 et 5.3. Cette extension consiste en le simple remplacement des variables du langage par des variables du store. Un terme $\mathcal{S}$ étendu est notée par la suite S. Les tâches sont construites à partir de cette extension, grâce à la grammaire présentée Figure 5.5. Intuitivement, une tâche

$$\begin{array}{rcl} \mathcal{T} & ::= & \xi T \mid \mathcal{T}\ \mathcal{T} \quad \text{Les tâches} \\ T & ::= & \langle\rangle \mid \langle S \quad T\rangle \quad \text{Les processus légers} \end{array}$$

FIGURE 5.5 – La syntaxe des tâches

est un multi-ensemble de threads nommé ξT. Ces processus légers sont soit terminé, soit encore en cours d'exécution avec une commande S à accomplir, et une continuation T.

5.2.1 La relation de réduction

Comme nous l'avons introduit précédemment, la sémantique opérationnelle du langage Oz/K est définie par une relation entre des structures d'exécution $(\sigma, \mathcal{T})$. Comme pour les calculs précédemment, cette relation est introduite par un ensemble de règles de réduction, présentant de manière formelle le comportement de chacune des commandes du langage. Nous adoptons dans cette partie la même approche envers la sémantique opérationnelle que le chapitre 13 de [110], et en particulier, nous adoptons la même notation pour les règles de réduction. En effet, cette formulation des règles, définie pour le langage Oz, est relativement bien adaptée pour gérer à la fois des stores et des tâches. De plus, le fait d'utiliser le même formalisme que celui dans lequel a été défini la base de notre langage Oz/K, montre avec clarté que ce langage est bien une extension qui ne modifie en aucun cas les fondements de Oz, et qui ne pourrait être encodée avec les opérations de base de Oz. Les règles d'inférence pour définir la réduction du langage Oz/K prend la forme suivante, cette règle signifiant que la structure

d'exécution $(\sigma, \mathcal{T})$ se réduit en $(\sigma', \mathcal{T}')$ sous la condition C :

$$\frac{\mathcal{T} \parallel \mathcal{T}'}{\sigma \parallel \sigma'} \quad \texttt{si} \quad C$$

Quelques définitions préliminaires

Comme dans les calculs précédents, nous introduisons la sémantique opérationelle de notre langage par un ensemble de définitions donnant les bases sur lesquelles sont fondées nos règles de réduction.

L'équivalence structurelle. Cette relation de réduction fontionne modulo une relation d'éqauivalence structurelle sur les termes manipulés (tâches et stores) qui identifie les éléments de sémantique semblable. La Figure 5.6 présente l'équivalence structurelle entre les tâches, qui identifie les termes du calcul modulo α conversion. Les tâches sont identifiées modulo associativité et commutativité des processus légers : ces derniers s'exécutant en parallèle, l'ordre dans lequel ils sont déclaré n'a pas d'importance.

$$\frac{S_1 \equiv_\alpha S_2}{S_1 \equiv S_2} \qquad \mathcal{T}_1\ \mathcal{T}_2 \equiv \mathcal{T}_2\ \mathcal{T}_1 \qquad \mathcal{T}_1\ (\mathcal{T}_2\ \mathcal{T}_3) \equiv (\mathcal{T}_1\ \mathcal{T}_2)\ \mathcal{T}_3 \qquad \frac{\mathcal{T}_1 \equiv \mathcal{T}_2}{\mathcal{T}_1\ \mathcal{T} \equiv \mathcal{T}_2\ \mathcal{T}}$$

FIGURE 5.6 – La relation d'équivalence entre tâches

L'équivalence structurelle pour les stores, présentée Figure 5.7, est elle aussi assez concise. Là encore, les éléments du store, déclarant simplement les différents objets construits par le programme durant son exécution, peuvent permuter librement. De plus, les égalités impliquant des termes du calcul équivalents sont elles aussi équivalentes.

La sémantique des stores. Afin de simplifier la manipulation des stores, nous définissons une syntaxe simple de contrainte, donnant quels éléments sont déclarés si une variable est liée ou non, si deux variables sont unifiable, etc. Cette syntaxe est donnée Figure 5.8. Afin de ne pas avoir de doublons, nous utilisons dans cette figure le symbole ϵ pour dénoter soit une variable x soit un nom ξ. à partir de ces contraintes, nous construisons Figure 5.9 le prédicat $\models$ entre un store σ et une contrainte ϕ, définissant quand un store valide une telle

$$\frac{\sigma_1 \equiv \sigma_3 \qquad \sigma_2 \equiv \sigma_4}{\sigma_1 \wedge \sigma_2 \equiv \sigma_4 \wedge \sigma_3} \qquad \frac{\mathcal{T}_1 \equiv \mathcal{T}_2 \qquad \sigma_1 \equiv \sigma_2}{x = \texttt{pack}(\xi, \mathcal{T}_1, \sigma_1, M) \equiv x = \texttt{pack}(\xi, \mathcal{T}_2, \sigma_2, M)}$$

$$\frac{S_1 \equiv S_2}{\xi : \texttt{proc}\{P\ X_1 \ldots X_n\}\ S_1\ \texttt{end} \equiv \xi : \texttt{proc}\{P\ X_1 \ldots X_n\}\ S_2\ \texttt{end}} \qquad x = y \equiv y = x$$

FIGURE 5.7 – La relation d'équivalence entre les stores

$$\phi \quad ::= \quad \sigma \quad | \quad x = \bot \quad | \quad V \bowtie V \quad | \quad \neg\phi \quad | \quad \exists\epsilon.\phi \quad | \quad \forall\epsilon.\phi \quad | \quad \phi \wedge \phi \quad | \quad \phi \vee \phi$$

FIGURE 5.8 – Les contraintes de store

contrainte. Informellement, ce prédicat est réflexif et transitif (dans le sens où un store est une contrainte). De plus, la déclaration de valeur dans un store est transitive : si $x = y$ et $y = V$ sont dans le même store, ce la signifie que $y = V$, mais aussi que $x = V$. Les autres règles présentées dans la Figure définissent la signification de chaque terme de contrainte. Par exemple, la contrainte $x = \bot$ signifie que la variable x est déclarée dans le store, mais n'est pas liée à une valeur. De manière similaire, la contrainte $V \bowtie V'$ signifie que ces deux valeurs sont unifiables. Finalement, les six dernières règles montre comment les lieurs classiques de la logique et les opérateurs de négation $\neg$ et de disjonction $\vee$ sont interprétés dans notre sémantique. La sémantique de ces lieurs est construite à partir de deux fonctions sur les stores : v et n qui donnent respectivement les variables et les noms déclarés dans le store en paramètre. Plus précisement, nous pouvons définir les ensembles $n(\sigma)$ et $v(\sigma)$ comme suit :

$$n(\sigma) \triangleq \{\xi \mid \exists E,\ \sigma \vDash x = \xi \vee \xi : E\}$$
$$v(\sigma) \triangleq \{x \mid \exists V, O,\ \sigma \vDash x \vee x = V \vee \texttt{failed}(x) \vee \texttt{thread}(x) \vee \texttt{kell}(O, x)\})$$

Les stores non valides. Les stores de notre langage ont un grand pouvoir d'expressions, ce qui nous permet de connaître beaucoup de chose sur l'exécution

$$\sigma \vDash \sigma \qquad \frac{\sigma_1 \vDash \sigma_2 \qquad \sigma_2 \vDash \sigma_3}{\sigma_1 \vDash \sigma_3} \qquad \frac{\sigma \vDash \phi_1 \wedge \phi_2}{\sigma \vDash \phi_1} \qquad \frac{\sigma \vDash \phi_1 \qquad \sigma \vDash \phi_2}{\sigma \vDash \phi_1 \wedge \phi_2}$$

$$\frac{\sigma \vDash y = V \wedge x = y}{\sigma \vDash x = V} \qquad \frac{\sigma \vDash x \qquad \forall V, \sigma \nvDash x = V}{\sigma \vDash x = \bot} \qquad \frac{V \equiv V'}{\sigma \vDash V \bowtie V'}$$

$$\frac{\sigma \vDash V_1 \bowtie V_2 \wedge V_2 \bowtie V_3}{\sigma \vDash V_1 \bowtie V_3} \qquad \frac{\sigma \vDash x = \bot}{\sigma \vDash x \bowtie V} \qquad \frac{\sigma \vDash x = V \wedge V \bowtie V'}{\sigma \vDash x \bowtie V'}$$

$$\frac{\sigma \vDash x \bowtie y}{\sigma \vDash \texttt{failed}(x) \bowtie \texttt{failed}(y)}$$

$$\frac{\sigma \vDash x_1 \bowtie x'_1 \wedge \cdots \wedge x_n \bowtie x'_n}{\sigma \vDash l(a_1 : x_1, \ldots, a_n : x_n) \bowtie l(a_1 : x'_1, \ldots, a_n : x'_n)} \qquad \frac{\sigma \nvDash \phi}{\sigma \vDash \neg\phi} \qquad \frac{\sigma \vDash \phi}{\sigma \vDash \phi \vee \phi'}$$

$$\frac{\forall x \in v(\sigma), \sigma \vDash \phi}{\sigma \vDash \forall x.\phi} \qquad \frac{\forall \xi \in v(\sigma), \sigma \vDash \phi}{\sigma \vDash \forall \xi.\phi} \qquad \frac{\exists x \in v(\sigma), \sigma \vDash \phi}{\sigma \vDash \exists x.\phi} \qquad \frac{\exists \xi \in v(\sigma), \sigma \vDash \phi}{\sigma \vDash \exists \xi.\phi}$$

FIGURE 5.9 – La sémantique des stores

en cours d'un programme. Nous pouvons en effet connaître l'état des threads, les valeurs liées à chaque variables, savoir quelles contraintes d'égalité entre différentes variables ont été déclarées. Mais ce pouvoir d'expression a aussi l'inconvénient de permettre à un store d'être *incohérent*. En effet, il est possible dans un store σ de lier une variable x à deux valeurs différentes, d'avoir une structure de kell qui forme des cycles, etc. Nous présentons Figure 5.10 une contrainte ϕ qui définit quand un store contient une incohérence :

Définition 11. *Un store σ est dit* incohérent *ou* invalide *s'il existe V, V', et pour tout O, O' nous avons $\sigma \vDash \phi$ (avec ϕ définie Figure 5.10).*

La contrainte ϕ est composée de six disjonctions, chacune d'elles précisant une condition d'invalidité du store. La première cause d'invalidité est, tel que nous l'avons signalé, une variable liée à différentes valeurs non unifiables. La seconde et la troisième cause concerne la structure des kells. En effet, ces composants doivent impérativement être assemblés dans une structure d'arbre enraciné, et un store validant une de ces clauses de la contrainte aurait soit un cycle dans l'assemblage, soit un composant avec deux pères, ce qui est interdit. La quatrième clause est semblable à la troisième, et spécifie qu'un *thread* cette fois ne peut avoir deux composants englobant. Les deux dernières clauses donnent que les déclarations d'inclusion ne peuvent être faites qu'entre des noms correspondant à des kells, ou à des threads dans des kells.

$$
\begin{array}{lll}
 & \exists x,\ x = V \wedge x = V' \wedge \neg(V \bowtie V') & (1) \\
\vee & \exists \kappa_1, \kappa_2,\ \mathtt{in}(\kappa_1, \kappa_2) \wedge \mathtt{in}(\kappa_2, \kappa_1) & (2) \\
\vee & \exists \kappa_1 \neq \kappa_2, \kappa,\ \mathtt{in}(\kappa_1, \kappa) \wedge \mathtt{in}(\kappa_2, \kappa) & (3) \\
\vee & \exists \kappa_1 \neq \kappa_2, \xi,\ \mathtt{inth}(\kappa_1, \tau) \wedge \mathtt{in}(\kappa_2, \tau) & (4) \\
\vee & \exists \kappa, \kappa', \forall x, x',\ \mathtt{in}(\kappa, \kappa') \wedge \neg(\kappa : \mathtt{kell}(O, x) \wedge \kappa' : \mathtt{kell}(O', x')) & (5) \\
\vee & \exists \kappa, \tau, \forall x, x',\ \mathtt{inth}(\kappa, \tau) \wedge \neg(\kappa : \mathtt{kell}(O, x) \wedge \tau : \mathtt{thread}(x')) & (6)
\end{array}
$$

FIGURE 5.10 – La contrainte ϕ définissant les stores invalides

L'extension de la sémantique des stores. Afin de d'avoir une présentation lisible de la sémantique des stores, nous avons donné précédemment une syntaxe des contraintes de store ϕ relativement simple et suffisante pour décrire quand un store est invalide. Cette syntaxe est maintenant étendue Figure 5.11 par deux nouvelles constructions, spécifiques à la manipulation des kells et des échanges de messages entre composants. La première construction que nous rajoutons est

$$
\phi \quad ::= \quad \ldots \quad | \quad \mathtt{strict}(V) \quad | \quad \mathtt{access}(\xi, \xi, \xi)
$$

FIGURE 5.11 – L'extension des contraintes de store

$\mathtt{strict}(V)$, qui est valide si et seulement si la valeur V est *stricte*. Une valeur est dite *stricte* quand elle est totalement définie, qu'elle ne possède aucune référence vers des variables non liées, comme cela est précisé Figure 5.12. Ce prédicat est fort utile dans les règles de communications entre composants, pour s'assurer que les portes n'échangent que des valeurs strictes, sans variable non liée qui pourraient conduire à la perte de l'isolation et du contrôle des communications entre ces composants. Le second prédicat est $\mathtt{access}(\xi, \kappa_1, \kappa_2)$. Il définit quand la porte ξ peut être utilisée pour la communication entre κ_1 et κ_2. Comme cela a été précisé précédemment, les communications sont toujours autorisées à l'intérieur d'un même composant, et entre un composant et ses fils. Le reste de la

$$\sigma \vDash \texttt{strict}(u) \qquad \sigma \vDash \texttt{strict}(\texttt{pack}(\xi, \mathcal{T}, \sigma, M)) \qquad \frac{\sigma \vDash \xi : \texttt{gate} \vee \neg\xi : E}{\sigma \vDash \texttt{strict}(\xi)}$$

$$\frac{\sigma \vDash \texttt{strict}(v) \wedge x = v}{\sigma \vDash \texttt{strict}(x)} \qquad \frac{\sigma \vDash \texttt{strict}(x)}{\sigma \vDash \texttt{strict}(\texttt{failed}(x))}$$

$$\frac{\sigma \vDash \texttt{strict}(x_1) \wedge \cdots \wedge \texttt{strict}(x_n)}{\sigma \vDash \texttt{strict}(l(a_1 : x_1, \ldots, a_n : x_n))}$$

$$\frac{\sigma \vDash \xi : \texttt{proc}\{\$\ X_1 \ldots X_n\} S\ \texttt{end} \qquad \forall x \in v(S),\ \sigma \vDash \texttt{strict}(x)}{\sigma \vDash \texttt{strict}(\xi)}$$

FIGURE 5.12 – La sémantique de `strict`

sémantique de ce prédicat suit ce même principe : toute communication ne peut franchir qu'une seule frontière de composant. L'ouverture d'une porte a un effet similaire à la destruction d'une frontière pour les communications passant sur cette porte, et permet ainsi d'avoir des communications distantes. La sémantique de ce prédicat est ainsi basé sur une notion de distance définissant combien de frontières sont placées entre deux composants, pour une porte donnée. Cette distance est donnée Figure 5.13 comme une relation $\sigma \vDash \texttt{dist}(\xi, \kappa_1, \kappa_2, n)$. Une telle relation définit qu'il existe dans le store σ un chemin de communication sur la porte ξ, entre les composants κ_1 et κ_2, et qui ne franchit que n frontières. Bien sûr, il existe plusieurs chemins entre deux composants. Par exemple, entre deux composants frères, il existe un chemin passant simplement par le père, mais aussi un autre passant par le père, le grand-père, puis de nouveau par le père. Il existe néanmoins un plus court chemin, qui correspond au nombre minimal de frontières à franchir. Nous pouvons donc définir le prédicat $\texttt{access}(\xi, \kappa_1, \kappa_2)$ comme suit :

$$\frac{\sigma \vDash \texttt{dist}(\xi, \kappa_1, \kappa_2, n) \qquad n \leq 1}{\sigma \vDash \texttt{access}(\xi, \kappa_1, \kappa_2)}$$

Les sous-threads et les sous-kells. Nous utilisons deux nouvelles fonctions sur les stores, calculant tous les sous-composants d'un composant donné, et tout les threads que ces composants exécutent. La fonction `subk`, calculant l'ensemble des sous composants à partir d'une paire (σ, ξ), est définie récusivement comme

$$\frac{\sigma \vDash \kappa = \texttt{kell}(O, x)}{\sigma \vDash \texttt{dist}(\xi, \kappa, \kappa, 0)}$$

$$\frac{\sigma \vDash \texttt{dist}(\xi, \kappa_1, \kappa_2, n) \qquad \sigma \vDash \texttt{in}(\kappa_2, \kappa_3) \qquad \sigma \vDash \kappa_2 = \texttt{kell}(O, x) \qquad \kappa_3 \cdot \xi \in O}{\sigma \vDash \texttt{dist}(\xi, \kappa_1\kappa_3, n)}$$

$$\frac{\sigma \vDash \texttt{dist}(\xi, \kappa_1, \kappa_2, n) \qquad \sigma \vDash \texttt{in}(\kappa_2, \kappa_3) \qquad \sigma \vDash \kappa_2 = \texttt{kell}(O, x) \qquad \kappa_3 \cdot \xi \notin O}{\sigma \vDash \texttt{dist}(\xi, \kappa_1\kappa_3, n+1)}$$

$$\frac{\sigma \vDash \texttt{dist}(\xi, \kappa_1, \kappa_2, n) \qquad \sigma \vDash \texttt{in}(\kappa_3, \kappa_2) \qquad \sigma \vDash \kappa_3 = \texttt{kell}(O, x) \qquad \kappa_2 \cdot \xi \in O}{\sigma \vDash \texttt{dist}(\xi, \kappa_1\kappa_3, n)}$$

$$\frac{\sigma \vDash \texttt{dist}(\xi, \kappa_1, \kappa_2, n) \qquad \sigma \vDash \texttt{in}(\kappa_3, \kappa_2) \qquad \sigma \vDash \kappa_3 = \texttt{kell}(O, x) \qquad \kappa_2 \cdot \xi \notin O}{\sigma \vDash \texttt{dist}(\xi, \kappa_1\kappa_3, n+1)}$$

FIGURE 5.13 – La relation de distance entre deux composants

suit :

$$\frac{K \triangleq \{\xi' \mid \sigma \vDash \xi : \texttt{kell}(O, x) \wedge \texttt{in}(\xi, \xi') \wedge \xi' : \texttt{kell}(O', x') \wedge x' = \bot\}}{\texttt{subk}(\sigma, \xi) \triangleq \bigcup_{\xi' \in K} \texttt{subk}(\sigma, \xi') \cup K}$$

Cette définition est assez claire : les descendants d'un composant sont ses fils (définis par l'ensemble K), et les descendants de ces fils. Il est facile, à partir de cette définition de sous-composants, de construire la fonction `subth` qui donne les threads s'exécutant dans un composant. Cette fonction prend elle aussi une paire (σ, ξ) en paramètre, où ξ est le composant contenant les threads voulus :

$$\frac{T(\xi') \triangleq \{\xi_t \mid \sigma \vDash \texttt{inth}(\xi', \xi_t)\}}{\texttt{subth}(\sigma, \xi) \triangleq \bigcup_{\xi' \in \texttt{subk}(\sigma, \xi)} T(\xi')}$$

Les abréviations. Finalement, la dernière notion que nous introduisons avant la présentation des règles de réduction sont les abréviations que nous utilisons pour simplifier nos règles. Ces facilités d'écriture sont résumées Figure 5.14. Nous utilisons une dernière abréviation dans nos règles : $S\,|^{\sigma}_{\kappa}$. Ce terme n'est valide que s'il existe un thread ξ tel que $\sigma \vDash \texttt{inth}(\kappa, \xi)$, et correspond à la tâche $\xi\langle S \quad T\rangle$, pour un T qui n'est pas modifié par la règle de réduction.

La règle	est une abréviation pour
$\dfrac{S}{\sigma} \Vert \dfrac{S'}{\sigma'}$ `si` C	$\dfrac{\xi\langle S \quad T\rangle}{\sigma} \Vert \dfrac{\xi\langle S' \quad T\rangle}{\sigma'}$ `si` C
$\dfrac{\mathcal{T}}{\sigma \models \phi} \Vert \dfrac{\mathcal{T}'}{\sigma'}$ `si` C	$\dfrac{\mathcal{T}}{\sigma} \Vert \dfrac{\mathcal{T}'}{\sigma'}$ `si` $C \wedge \sigma \models \phi$

FIGURE 5.14 – Les deux abréviations principales

Les règles de réduction

Nous donnons finalement dans cette partie les règles *principales* définissant la sémantique opérationnelle de notre langage. Nous ne donnons pas par exemple toutes les règles d'erreur, qui s'appliquent quand celles données ici ne peuvent l'être, et dont le seul effet est la levée d'une exception correspondant à l'erreur rencontrée. Dans la suite, nous présentons chaque règle individuellement, afin d'être clair sur tout les détails les concernant.

Les règles structurelles. Nous avons deux règles simple définissant dans quel contexte une tâche peut s'exécuter. La première montre que ces processus sont bien en parallèle, et s'exécutent de manière indépendante :

$$\frac{\mathcal{T} \quad \mathcal{T}'}{\sigma} \Vert \frac{\mathcal{T}' \quad \mathcal{T}}{\sigma'} \quad \texttt{si} \quad \frac{\mathcal{T}}{\sigma} \Vert \frac{\mathcal{T}'}{\sigma'}$$

La seconde est encore plus simple, et établit que la relation de réduction est bien définie modulo la relation d'équivalence entre tâches et entre les stores :

$$\frac{\mathcal{T}_1}{\sigma_1} \Vert \frac{\mathcal{T}_2}{\sigma_2} \quad \texttt{si} \quad \frac{\mathcal{T}_3}{\sigma_3} \Vert \frac{\mathcal{T}_4}{\sigma_4} \ \texttt{et}\ \mathcal{T}_1 \equiv \mathcal{T}_3 \wedge \mathcal{T}_2 \equiv \mathcal{T}_4 \wedge \sigma_1 \equiv \sigma_3 \wedge \sigma_2 \equiv \sigma_4$$

Les règles d'exécution séquentielle. Nous avons trois règles définissant comment se comportent les différentes éléments de la sémantique séquentielle du langage. La première donne comment le mot clef `skip` se comporte : ce mot clef correspond à l'instruction vide, et n'a aucun effet sur le store ou les autres tâches.

$$\frac{\xi\langle \texttt{skip} \quad T\rangle}{\sigma} \Vert \frac{\xi T}{\sigma}$$

La seconde règle montre comment les différentes déclarations d'un terme S sont décomposées pour être ensuite exécutées :

$$\frac{\xi\langle (S_1\ S_2) \quad T\rangle}{\sigma} \Vert \frac{\xi\langle S_1 \quad \langle S_2 \quad T\rangle\rangle}{\sigma}$$

Et finalement, la troisième règle présente comment se passe l'arrêt normal d'une tâche. La variable de status, supposée non liée se voit alors affecter la valeur `terminated`, signifiant ainsi que la tâche s'est exécuté sans problème.

$$\frac{\xi\langle\rangle}{\sigma \models \xi : \texttt{thread}(x) \wedge x = \bot} \;\Big\Vert\; \frac{}{\sigma \wedge x = \texttt{terminated}}$$

La règle de création de thread. Un thread est créée par la commande de construction de thread $\texttt{thread}\{x\}\ S\ \texttt{end}$. Durant cette création, la règle utilise un nouveau nom ξ' qui représente le nouveau thread, et une nouvelle variable x' qui est la variable de status de ce thread. Ce nouveau thread est mis en place par la nouvelle tâche $\xi'\langle S\ \langle\rangle\rangle$, qui est placée dans le même kell que la tâche d'origine.

$$\frac{\xi\langle\texttt{thread}\{x\}\ S\ \texttt{end}\quad T\rangle}{\sigma \models x = \bot \wedge \texttt{inth}(\kappa,\tau)} \;\Big\Vert\; \frac{\xi T \quad \xi'\langle S\ \langle\rangle\rangle}{\sigma \wedge x = \xi' \wedge \xi' : \texttt{thread}(x') \wedge x' \wedge \texttt{inth}(\kappa,\xi')}$$

$$\texttt{si}\quad \xi', x' \text{ fresh}$$

La règle de création de variables. Cette règle est relativement simple, et a pour effet de remplacer les variables formelles du langage par des nouvelles variables du store.

$$\frac{\texttt{local}\ \ X_1 \ldots X_n\ \texttt{in}\ \ S\ \texttt{end}}{\sigma} \;\Big\Vert\; \frac{S\{^{x_i}/_{X_i}\}}{\sigma \wedge x_1 \wedge \cdots \wedge x_n} \quad \texttt{avec}\quad x_i \text{ fresh}$$

Les règles d'affectation de variable. Contrairement à la plupart des langages logiques, et en particulier, contrairement à Oz, les déclarations d'égalité ne font pas intervenir d'unification. Ce choix est motivé par plusieurs problèmes que peut poser l'application automatique de l'unification lors de l'utilisation du prédicat d'égalité. Par exemple, l'unification est une opération très couteuse, alors que l'affectation d'une valeur à une variable est quasi-instantanée. De plus, ne pas avoir d'unification permet d'éviter des erreurs de programmation, telles que donner une valeur V à une variable x, pensant qu'elle est libre, alors qu'elle ne l'est pas. Sans unification, l'erreur lève une exception durant l'exécution du programme, et peut ainsi être facilement corrigée. Avec une unification, il est possible que V soit unifiable avec celle que contient déjà x. L'erreur ne sera donc pas visible, et pourra même causer des disfonctionnement du programme par la suite, x ne contenant pas V, mais l'unification de cete valeur avec une autre. L'unification, opération de base des langages logiques, est tout de même intégré au langage, sous forme d'une procédure `UnifyS`. L'effet de cette procédure n'est pas présenté ici, car il est identique à l'unification telle qu'elle est présentée dans le chapitre 13 de [110].

Nous avons trois règles d'affectation. La permière concerne la liaison d'une variable à une valeur, telle qu'un entier ou un message structuré. Comme nous l'avons précisé, cette règle ne peut s'appliquer si x possède déjà une valeur :

$$\frac{x = V}{\sigma \models x = \bot} \;\Big\|\; \frac{\texttt{skip}}{\sigma \wedge x = v} \quad \texttt{si} \quad V \notin \texttt{Var}$$

La seconde présente la sémantique de la déclaration $x = y$. Afin d'éviter l'unification, cette déclaration est ajoutée au store si et seulement si au maximum une variable parmis x et y possède une valeur :

$$\frac{x = y}{\sigma \models x = \bot \vee y = \bot} \;\Big\|\; \frac{\texttt{skip}}{\sigma \wedge x = y}$$

Finalement, la dernière règle présente comment l'accès aux champs d'un message structuré s'effectue :

$$\frac{x = x'.z}{\sigma \models x = \bot \wedge x' = l(a_1 : x_1, \ldots, a_n : x_n) \wedge a = f_i} \;\Big\|\; \frac{\texttt{skip}}{\sigma \wedge x = x_i}$$

La règle de lecture du status. Nous proposons ici de montrer la sémantique d'une procédure de base, la procédure `Status`, qui retourne le status d'un processus léger. Il est à noter qu'il est seulement possible de connaître le status d'un thread s'exécutant dans le composant courant, afin de garantir la séparation entre les kells d'un même programme.

$$\frac{\xi\langle\{\texttt{Status}\ x\ y\} \quad T\rangle}{\sigma} \;\Big\|\; \frac{\xi\langle T\rangle}{\sigma \wedge y = z} \quad \texttt{si} \quad \sigma \models \begin{array}{l} y = \bot \wedge x = \xi' \wedge \xi' : \texttt{thread}(z) \\ \wedge \texttt{inth}(\kappa, \xi) \wedge \texttt{inth}(\kappa, \xi') \end{array}$$

Les règles pour la conditionnelle. La sémantique de la commande conditionnelle `if` x `then` S_1 `else` S_2 `end` est classique, identique à celle présentée pour le langage Oz. Elle est construite à partir de deux règles de réduction. La première est appliquée lorsque la condition de la commande est validée :

$$\frac{\texttt{if}\ x\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}}{\sigma \models x = \texttt{true}} \;\Big\|\; \frac{S_1}{\sigma}$$

La seconde règle est utilisée lorsque la condition de la commande n'est pas validée :

$$\frac{\texttt{if}\ x\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}}{\sigma \models x = \texttt{false}} \;\Big\|\; \frac{S_2}{\sigma}$$

Les règles pour le filtrage de motif. La commande pour le fitrage de motif `case` x `of` J `then` S_1 `else` S_2 du langage Oz est similaire à la conditionnelle présentée précédemment, dans le sens où nous avons deux cas : soit le filtrage fonctionne, auquel cas S_1 est exécuté, soit il ne fonctionne pas (la condition n'est pas validée), et c'est S_2 qui sera exécuté dans ce cas. Nous avons une fonction, `match`, qui prend en paramètre un triplet (σ, x, J) et qui renvoie

une substitution dans le cas où le filtrage fonctionne (la substitution définit les valeurs des différentes variables non liées du motif). Si le filtrage ne fonctionne pas, la fonction renvoie alors $\perp$. Les cas ou le filtrage est validée sont présentés Figure 5.15 (les cas non présentés dans cette figure correspondent aux echecs du filtrage). Cette figure est constituée de trois colonnes : la première donne quelle condition le store doit valider pour que le fitrage soit valide. La seconde donne la forme du filtre, et la troisième donne la substitution calculée par la fonction `match`. Notons que la substitution `Id` est l'identité qui ne modifie aucune variable. Avec cette fonction `match` nous pouvons aisément définir la règle de

ϕ	J	$\texttt{match}(\sigma, x, J)$
$x = v$	v	`Id`
$x = v \wedge y = v$	y	`Id`
$x = v \wedge y = \perp$	X	$[X \mapsto v]$
$\bigwedge_{i=1}^{n} a_i = a'_i \wedge l = l'$ $\wedge x = l(a_1 : x_1, \ldots, a_n : x_n)$	$l'(a'_1 : X_1, \ldots, a'_n : X_n)$	$[X_i \mapsto x_i]$

FIGURE 5.15 – La fonction `match`

réduction pour un filtrage valide :

$$\frac{\texttt{case}\ x\ \texttt{of}\ J\ \texttt{then}\ S_1\ \texttt{else}\ S_2}{\sigma} \Bigg\| \frac{\theta(S_1)}{\sigma} \quad \texttt{si}\ \ \texttt{match}(\sigma, x, J) = \theta$$

La règle pour un echec de filtrage est aussi relativement simple :

$$\frac{\texttt{case}\ x\ \texttt{of}\ J\ \texttt{then}\ S_1\ \texttt{else}\ S_2}{\sigma} \Bigg\| \frac{S_2}{\sigma} \quad \texttt{si}\ \ \texttt{match}(\sigma, x, J) = \perp$$

Les règles pour la création et l'utilisation des procédures. Notre sémantique opérationnelle comprend trois règles pour la manipulation des procédures. La première montre comment les procédures sont créées, et ajoutées dans le store.

$$\frac{\texttt{proc}\{x\ X_1 \ldots X_n\}\ S\ \texttt{end}}{\sigma \models x = \perp} \Bigg\| \frac{\texttt{skip}}{\sigma \wedge x = \xi \wedge \xi : \texttt{proc}\{\$\ X_1 \ldots X_n\} S\ \texttt{end}} \quad \texttt{avec}\ \ \xi \text{ fresh}$$

Cette règle s'assure que x ne possède aucune valeur, puis crée un nouveau nom ξ, et associe x à ξ et ξ à la procédure fraichement créée. La seconde règle permet de *remplacer* une procédure par une nouvelle. Cette règle est un rajout fait à Oz, pour prendre en compte des opérations intéressantes, comme l'héritage des objets. Ce remplacement ne peut se faire que sous deux conditions : il faut que la nouvelle procédure ait le même nombre d'argument que l'originale, et que l'originale n'ait pas de référence vers des variables extérieures (signalé par la contrainte $\texttt{strict}(\xi)$). Ces variables pourraient en effet constituer des effets de bords qui ne seraient plus assurés, et qui pourraient ainsi compromettrent le

bon fonctionnement de l'intégralité du programme.

$$\frac{\texttt{proc}\{x\ X_1 \ldots X_n\}\ S'\ \texttt{end}}{\sigma \wedge \xi : \texttt{proc}\{\$\ X_1 \ldots X_n\} S\ \texttt{end}} \Bigg\| \frac{\texttt{skip}}{\sigma \wedge \xi : \texttt{proc}\{\$\ X_1 \ldots X_n\} S'\ \texttt{end}} \quad \texttt{si} \quad \sigma \models x = \xi \wedge \texttt{strict}(\xi)$$

Finalement, l'utilisation d'une procédure se fait classiquement par l'application de ses arguments, comme décrit dans la règle qui suit :

$$\frac{\{x\ x_1 \ldots x_n\}}{\sigma \models x = \xi \wedge \xi : \texttt{proc}\{\$\ X_1 \ldots X_n\} S\ \texttt{end}} \Bigg\| \frac{S\{^{x_i}/_{X_i}\}}{\sigma}$$

Les règles pour les exceptions. Les exceptions sont gérées par un ensemble de cinq règles montrant la sémantique du bloc `try` ... `catch` et de la commande `raise`. La première règle gère l'exécution normale d'un bloc `try` ... `catch` : le code S_1 est exécuté normalement, tout en gardant la garde `catch` au cas où un probl'eme surviendrait.

$$\frac{\texttt{try}\ \ S_1\ \texttt{catch}\ \ X\ \texttt{then}\ \ S_2\ \texttt{end}}{\sigma} \Bigg\| \frac{S_1 \quad \langle \texttt{catch}\ \ X\ \texttt{then}\ \ S_2\ \texttt{end} \rangle}{\sigma}$$

Dans le cas où aucune exception n'a été levée durant l'exécution de S_1, la garde est simplement effacée :

$$\frac{\texttt{catch}\ \ X\ \texttt{then}\ \ S_2\ \texttt{end}}{\sigma} \Bigg\| \frac{\texttt{skip}}{\sigma}$$

Par contre, supposons qu'une exception soit levée : l'exécution de S_1 est alors stoppée, et la garde est appelée pour traiter l'erreur. Cela est pris en compte par les deux règles suivantes : la première termine l'exécution de S_1 en passant toute les instructions qui ne sont pas des gardes.

$$\frac{\xi\langle \texttt{raise}\ \ x\ \texttt{end} \quad \langle S \quad T \rangle\rangle}{\sigma} \Bigg\| \frac{\xi\langle \texttt{raise}\ \ x\ \texttt{end} \quad T\rangle}{\sigma} \quad \texttt{si} \quad S \not\equiv \texttt{catch}\ \ldots \texttt{end}$$

Une fois que la garde est trouvée le traitement de l'erreur se fait de la même façon que dans les autres langages :

$$\frac{\xi\langle \texttt{raise}\ \ x\ \texttt{end} \quad \langle \texttt{catch}\ \ X\ \texttt{then}\ \ S_2\ \texttt{end} \quad T\rangle\rangle}{\sigma} \Bigg\| \frac{\xi\langle S_2\{^{x}/_{X}\} \quad T\rangle}{\sigma}$$

Finalement, il est possible qu'une exception ne soit jamais récupérée par une garde. Dans ce cas, le thread courant ne peut plus s'exécuter, à cause de l'exception, et est donc stoppé, tout en signalant par $y = \texttt{failed}(x)$ que cet arret est causé par une erreur.

$$\frac{\xi\langle \texttt{raise}\ \ x\ \texttt{end} \quad \langle\rangle\rangle}{\sigma \models \xi : \texttt{thread}(y) \wedge y = \bot} \Bigg\| \frac{}{\sigma \wedge y = \texttt{failed}(x)}$$

les règles pour l'utilisation des portes. Les portes peuvent être créées, ouvertes, refermées, et utilisées pour l'échange de messages entre différents composants. La règle pour la création de porte est relativement simple. Elle demande, comme d'habitude, que x ne soit pas liée, et crée un nouveau nom correspondant à la porte, qui est lié par la suite à x :

$$\frac{\{\texttt{NewGate}\quad x\}}{\sigma \vDash x = \bot} \;\Bigg\|\; \frac{\texttt{skip}}{\sigma \wedge x = \xi \wedge \xi : \texttt{gate}} \quad \texttt{avec} \quad \xi \text{ fresh}$$

La communication entre composant est régulée par les commandes d'ouverture et de fermeture de portes. Ces commandes prennent en paramètre le nom de la porte à ouvrir, avec le nom du sous-composant pour lequel la porte sera ouverte. Néanmoins, la porte et le composant peuvent être remplacés par l'atome `all`, signifiant que toute les portes ou tous les sous composants sont affectés par l'opération. Par conséquent, les deux règles d'ouverture et la fermeture de portes ont chacune quatre cas d'utilisation dépendant de la forme de leurs deux paramètres. Afin de factoriser ces différents cas, nous utilisons une fonction Ω définissant quelles portes sont à ouvrir pour quels composants, en fonction de quatre paramètres : (i) le store, (ii) le composant demandant l'ouverture ou la fermeture des portes, (iii) le ou les composants affectés par cette opération, et (iv) les portes mises en jeu dans l'opération.

$$\Omega(\sigma, \kappa, k, g) \triangleq \begin{cases} \kappa' \cdot \xi & \text{si } \sigma \vDash g = \xi \wedge \xi : \texttt{gate} \wedge k = \kappa' \wedge \texttt{in}(\kappa, \kappa') \\ \texttt{K} \cdot \xi & \text{si } \sigma \vDash g = \xi \wedge \xi : \texttt{gate} \wedge k = \texttt{all} \\ \kappa' \cdot \texttt{G} & \text{si } \sigma \vDash g = \texttt{all} \wedge k = \kappa' \wedge \texttt{in}(\kappa, \kappa') \\ \texttt{K} \cdot \texttt{G} & \text{si } \sigma \vDash g = \texttt{all} \wedge k = \texttt{all} \end{cases}$$

Ainsi, la règle d'ouverture de porte rajoute à l'ensemble des portes ouvertes le résultat de la fonction Ω, alors que la règle de fermeture, elle, le soustrait :

$$\frac{\{\texttt{Open}\ k\ g\}\,|^{\sigma}_{\kappa}}{\sigma \wedge \kappa : \texttt{kell}(O, x)} \;\Bigg\|\; \frac{\texttt{skip}}{\sigma \wedge \kappa : \texttt{kell}(O \cup \Omega(\sigma, \kappa, k, g), x)}$$

$$\frac{\{\texttt{Close}\ k\ g\}\,|^{\sigma}_{\kappa}}{\sigma \wedge \kappa : \texttt{kell}(O, x)} \;\Bigg\|\; \frac{\texttt{skip}}{\sigma \wedge \kappa : \texttt{kell}(O \setminus \Omega(\sigma, \kappa, k, g), x)}$$

Finalement, comme nous pouvons maintenant ouvrir et fermer des portes, les composants peuvent échanger des messages. Nous avons déjà introduit les constructions principales pour la définition de la règle de communication : le prédicat `access` qui est valide lorsque deux composants peuvent communiquer via une porte donnée et le prédicat `strict`, valide seulement lorsque la valeur en paramètre ne contient pas de référence non liée [4]. Ainsi, nous pouvons définir avec la règle suivante la sémantique des opérateurs `Send` et `Receive`.

$$\frac{\{\texttt{Send}\ g\ x\}\,|^{\sigma}_{\kappa} \quad \{\texttt{Receive}\ g'\ x'\}\,|^{\sigma}_{\kappa'}}{\sigma \vDash x' = \bot \wedge g = \xi \wedge g' = \xi \wedge \xi : \texttt{gate}} \;\Bigg\|\; \frac{\texttt{skip}\,|^{\sigma}_{\kappa} \quad \texttt{skip}\,|^{\sigma}_{\kappa'}}{\sigma \wedge y = x}$$

$$\texttt{si} \quad \sigma \vDash \texttt{strict}(x) \wedge \texttt{access}(\xi, \kappa, \kappa')$$

4. Rappelons que ce dernier prédicat est utile, car lorsque les valeurs échangées entre deux composants sont strictes, il assure que les portes restent le seul moyen de communiquer pour ces deux composants ;et c'est seulement avec cette assurance que la passivation est correcte et ne cause aucune incohérence dans le store

Les règles pour la manipulation des kells. Notre sémantique comporte deux règles pour manipuler les composants. La première consiste en la création d'un kell, et est assez semblable à la celle décrivant la création d'un processus léger : nous utilisons tout d'abord deux variables fraîches pour définir une nouvelle structure de composant comportant un nom κ' et un status y, et le reste de la règle décrit la création du processus exécutant le code de notre nouveau composant. Notons que le corps du processus doit être strict, toujours dans le but d'éviter les communications entre composants en dehors des portes.

$$\frac{\texttt{kell}\{x\}\ S\ \texttt{end}|_{\kappa}^{\sigma}}{\sigma \vDash x = \bot} \Bigg\| \frac{\texttt{skip}|_{\kappa}^{\sigma} \quad \xi\langle S \quad \langle\rangle\rangle}{\sigma'}$$

$$\texttt{si} \begin{cases} \kappa', \xi', y, y' \text{ fresh} \wedge \sigma' \vDash \texttt{strict}(S) \\ \wedge\, \sigma' \triangleq \sigma \wedge \left(\begin{array}{c} x = \kappa' \wedge \kappa' : \texttt{kell}(\emptyset, y) \wedge y \\ \wedge\, \xi' : \texttt{thread}(y') \wedge y' \wedge \texttt{inth}(\kappa', \xi') \wedge \texttt{in}(\kappa, \kappa') \end{array} \right) \end{cases}$$

La règle pour le remplacement de kell est similaire à son homologue pour les procédures. cette règle permet de remplacer une structure vide de composant (où son contenu a été passivé) par un composant complet. Ainsi, le composant, sans changer de nom, peut de nouveau s'exécuter et être dans l'état actif (c'est à dire que sa variables d'état y' n'est pas liée).

$$\frac{\texttt{kell}\{x\}\ S\ \texttt{end}|_{\kappa}^{\sigma}}{\sigma \wedge \kappa' : \texttt{kell}(O, y)} \Bigg\| \frac{\texttt{skip}|_{\kappa}^{\sigma} \quad \xi'\langle S \quad \langle\rangle\rangle}{\sigma'}$$

$$\texttt{si} \begin{cases} \xi', y', y'' \text{ fresh} \wedge \sigma \vDash \texttt{strict}(S) \wedge x = \kappa' \wedge y = \texttt{packed} \wedge \texttt{in}(\kappa, \kappa') \\ \wedge \sigma' \triangleq \sigma \wedge \kappa' : \texttt{kell}(O, y') \wedge y' \wedge \xi' : \texttt{thread}(y'') \wedge y'' \wedge \texttt{inth}(\kappa', \xi') \end{cases}$$

Les règles de manipulation des valeurs packées. Les valeurs packées sont la forme que prennent les composants lorsqu'ils sont passivés. Ainsi, ele contiennent des informations essentielles pour permettre le re-déploiement du composant dans un nouvel environnement. Typiquement, la valeur $\texttt{pack}(\kappa, \mathcal{T}, \sigma, M)$ correspond au composant κ exécutant les processus $\mathcal{T}$ avec σ comme store. M quant à lui est un ensemble de noms dit *marqués*, et est utilisé pour éviter les conflits de nommage pouvant survenir lors de la remise en place du composant packé. Étudions tout d'abord la règle de passivation d'un composant x : elle utilise tout d'abord les prédicats `subk` et `subth` pour récupérer l'ensemble des sous-composants de x ainsi que les processus s'y exécutant. Elle passive alors tout ces composants en donnant une valeur à leur état, et crée la structure $\texttt{pack}(\kappa_0, \mathcal{T}, \sigma, \emptyset)$ stockant le nom du composant passivé κ_0, l'ensemble des processus précédemment calculé $\mathcal{T}$, et le store σ contenant toute les informations concernant la structure du composant passivé ainsi que les données qu'il

contient. Notons que l'ensemble des noms marqués est initialement vide.

$$\frac{\{\texttt{Pack}\ x\ y\}|^{\sigma}_{\kappa} \quad \mathcal{T}}{\sigma \vDash y = \bot} \Bigg\| \frac{\texttt{skip} \quad \emptyset}{\sigma \wedge \sigma'}$$

$$\texttt{si} \quad \begin{cases} \quad \texttt{subk}(\sigma, \kappa_0) = \{\kappa_1, \ldots, \kappa_m\} \wedge \texttt{subth}(\sigma, \kappa_0) = \{\xi_1, \ldots, \xi_n\} \\ \wedge\, \sigma \vDash \bigwedge_0^m (\texttt{kell}(O_i, x_i) \wedge x_i = \bot) \wedge x = \kappa_0 \wedge \texttt{in}(\kappa, \kappa_0) \\ \wedge\, \mathcal{T} \triangleq \xi_1 : T_1 \ldots \xi_n : T_n \wedge (\sigma' \triangleq \bigwedge_0^m x_i = \texttt{packed} \wedge y = \texttt{pack}(\kappa_0, \mathcal{T}, \sigma, \emptyset)) \end{cases}$$

La règle suivante donne la sémantique de l'opérateur `Unpack` servant à re-créer la structure d'un composant passivé. La première remarque que l'on peut faire sur cette règle est qu'elle utilise une substitution θ assez conséquente. En effet, nous avons mentionné le fait qu'il puisse y avoir des conflits de nommage lors du re-déploiement d'un composant : ce dernier utilisait lors de son fonctionnement des noms liés à certaines valeurs qui ont pu changer durant le temps que le composant à passé sous forme de valeur. Ainsi, recréer le composant et son store directement pourrait avoir la conséquence fâcheuse d'associer plusieurs valeur à un même nom et causer une incohérence au niveau du store global du programme. C'est pour éviter cela que nous utilisons la substitution θ : nous remplaçons tout les noms pouvant causer un conflit par un nom frais. Notons que ni le nom du composant accueillant le composant dé-passivé ni les noms contenu dans M ne sont affectés par la substitutions. La raison pour laquelle le nom du composant n'est pas modifié est relativement clair, et quant à M, il sert à lier le composant dé-passivé à son nouvel environnement. En effet, il est naturel de remplacer les noms et variable du store d'un composant packé par des éléments frais, mais cela empêche toute communication entre le composant reconstruit et son environnement dont il ne possède aucune référence. C'est pourquoi nous avons l'opérateur `Mark` : en remplaçant les données internes d'une valeur packée par des éléments du store courant, on s'assure que ces références sont valides et correspondent bien à des éléments de l'environnement de la valeur. Nous n'avons donc pas besoin de remplacer ces références, que nous stockons dans M. Finalement la procédure relance les processus contenu dans $\mathcal{T}$ et retourne dans x une liste de paires associant les anciennes porte du composant packé avec les nouvelles, issues de la substitution par θ (cette liste sert dans le cas où `Mark` n'a pas été utilisé pour re-connecter le composant avant son re-déploiement).

$$\frac{\{\texttt{Unpack}\ y\ x\}|^{\sigma}_{\kappa}}{\sigma \vDash \left(\begin{array}{c} \kappa : \texttt{kell}(O, z) \wedge x = \bot \\ \wedge\, y = \texttt{pack}(\kappa', \mathcal{T}, \sigma' \wedge \kappa' : \texttt{kell}(O', z'), M) \end{array} \right)} \Bigg\| \frac{\texttt{skip}|^{\sigma}_{\kappa} \quad \theta(\mathcal{T})}{\sigma \wedge \sigma''}$$

$$\texttt{si} \quad \begin{cases} \quad \theta \triangleq ([\xi \mapsto \xi']_{\xi \in n(\sigma)} [x \mapsto x']_{x \in v(\sigma)})_{|\mathbb{C}M} [\kappa_0 \mapsto \kappa] \wedge \Im(\theta) \setminus \{\kappa\} \text{ fresh} \\ \wedge\, \texttt{subth}(\sigma', \kappa') = (\xi_i^t)_1^n \wedge (\xi_i^g)_1^m \triangleq \{\xi \mid \sigma' \vDash \xi : \texttt{gate}\} \\ \wedge\, \sigma'' \triangleq \theta(\sigma') \wedge x = [(\xi_i^g, \theta(\xi_i^g))]_1^m \wedge (\bigwedge_1^n \texttt{inth}(\kappa, \theta(\xi_i^t)) \end{cases}$$

Comme nous l'avons vu précédemment, les valeurs packées peuvent être modifiées par l'intermédiaires de la primitive `Mark`. Cet opérateur prend en paramètre la valeur à modifierm une commande donnant quelle est la nature de la modification à effectuer et retourne dans son dernier paramètre le résultat de la modification. Dans sa version actuelle, `Mark` accepte deux commandes différentes : $\texttt{gate}(x\ y)$ remplace la porte x par y et $\texttt{prc}(x\ y)$ remplace la procédure x par y. Nous avons donc deux règles décrivant la sémantique de cet opérateur. Notons que dans les deux règles, l'ensemble des noms marqués

est étendu pour prendre en compte la nouvelle référence valide pointée par y. La première que nous présentons concerne le remplacement d'une porte, et est relativement simple, avec le simple contrôle que les éléments impliqués sont bien des portes, et l'application de la substitution de remplacement θ sur le store de la valeur packée résultat.

$$\frac{\{\texttt{Mark} \quad z \quad \texttt{gate}(x\ y) \quad p\}}{\sigma \models \wedge p = \bot} \;\Bigg\|\; \frac{\texttt{skip}}{\sigma \wedge p = \texttt{pack}(\kappa, \theta(\mathcal{T}), \theta(\sigma'), M \cup \{\xi'\})}$$

$$\texttt{si} \quad \left\{ \begin{array}{l} \theta \triangleq [\xi \mapsto \xi'] \wedge \sigma' \models \xi : \texttt{gate} \\ \wedge\, \sigma \models z = \texttt{pack}(\kappa, \mathcal{T}, \sigma', M) \wedge x = \xi \wedge \xi : \texttt{gate} \wedge y = \xi' \wedge \xi' : \texttt{gate} \end{array} \right.$$

La règle concernant le remplacement de procédure dans une valeur packé fonctionne de la même manière que la règle précédente, avec un simple contrôle et un remplacement par une substitution. Notons toutefois que comme pour la règle de remplacement de procédure présentée précédemment, nous demandons que le corps de la nouvelle fonction soit stricte afin d'éviter des captures de nom non voulues.

$$\frac{\{\texttt{Mark} \quad z \quad \texttt{prc}(x\ y) \quad p\}}{\sigma \models p = \bot} \;\Bigg\|\; \frac{\texttt{skip}}{\sigma \wedge p = \texttt{pack}(\kappa, \theta(\mathcal{T}), \theta(\sigma') \wedge \xi : E, M \cup \{\xi'\})}$$

$$\texttt{si} \quad \left\{ \begin{array}{c} \theta \triangleq [\xi \mapsto \xi'] \wedge E \triangleq \texttt{proc}\{\$\ X_1 \ldots X_n\} S' \ \texttt{end} \\ \wedge\, \sigma \models \left(\begin{array}{c} z = \texttt{pack}(\kappa, \mathcal{T}, \sigma', M) \wedge x = \xi \\ \wedge\, \xi : \texttt{proc}\{\$\ X_1 \ldots X_n\} S \ \texttt{end} \\ \wedge\, y = \xi' \wedge \xi' : E \wedge \texttt{strict}(S) \end{array} \right) \end{array} \right.$$

5.3 Exemples

Nous illustrons notre langage par une série d'exemples, dont le but et de montrer l'intérêt des constructions de composants, de portes et de mobilité de code. Nous structurons notre présentation en trois parties : (i) la première décrit le principe d'isolation des composants et des portes ; (ii) la seconde discute de l'encodage de la programmation distribuée dans notre modèle, et discute de la relation entre notre travail et Mozart ; et (iii) nous terminons par une discussion sur la mobilité de code et les capacités, les limitations de la passivation introduite dans ce langage. Notons toutefois que les exemples qui sont présentés ici sont assez différents de ceux que nous avons vu dans les chapitres précédents. En effet, nous restons ici avec des assemblages simples sans boucle ni routage, pour deux raisons. Tout d'abord, contrairement à nos autres recherches, celle présentée ici a pour but d'étudier des constructions spécifiques d'un langage, ce que l'on peut exprimer avec et avec quelle facilité. Cela demande donc des exemples ciblés, contrairement à nos approches précédentes qui se focalisaient sur l'expressivité générale d'un calcul afin de savoir quelle genre de construction nous avions à typer. Enfin, comme nous avons besoin d'exemples ciblés, nous les décrivons entièrement dans la syntaxe de notre langage. Construire des boucles et des assemblages complexes est bien sûr assez facile ici, mais n'apporte pas beaucoup à notre discussion, et demande tout de même un certain nombre de lignes de code.

5.3.1 Portes et isolation

Le principe de base des communications dans Oz/K est la localité, comme cela est fait dans le kell-calcul. En effet, par défaut, les composants pères et fils peuvent communiquer en toute liberté, et il faut utiliser des portes pour étendre cette capacité. Néanmoins, ce mode de communication n'est pas souhaitable dans certaines circonstances, typiquement lorsque le composant père manipule des données qu'il veut secrètes pour ses fils. Ainsi, supposons qu'un composant père `K1` veuille communiquer avec son fils `K` seulement sur la porte `G`, et empêcher toute communication sur les autres portes. La méthode pour permettre cette isolation est d'utiliser un nouveau composant (nommé `Server`) isolant `K` de son père, et qui ne transmette que les messages envoyés sur `G` : le principe de localité implique donc que seuls les messages sur `G` peuvent être échangé entre `K` et `K1`. Ensuite, la méthode de transfert peut prendre différentes formes impliquant ou non l'ouverture de portes. Dans cet exemple, nous identifions Figure 5.16 les trois méthodes possibles pour le transfert des messages envoyés par `K` (resp. `K'`) vers `K'` (resp. `K`). La première est identique à celle utilisée pour le kell-calcul : nous

(a)	(b)	(c)

`kell {Server}`
`  local K in`
`  kell {K}`
`  ...`
`  end`
`  {Relay G G}`
`end end`
	`kell {Server}`
	`  local K in`
	`  kell {K}`
	`  ...`
	`  end`
	`  end`
	`{Open Server G} end`
		`kell {Server}`
		`  local K in`
		`  kell {K}`
		`  ...`
		`  end`
		`  {Open K G}`
		`end end`

FIGURE 5.16

avons une procédure `Relay` dans le composant `Server` récupérant les messages envoyé sur `G` et les transmettant sur la même porte. De cette façon, les messages envoyé par `K` sur `G` peuvent être récupérés par la procédure `Relay` et renvoyés vers `K'` (et réciproquement). Cette méthode n'est pas sans défauts. En effet,

```
proc{Relay G1 G2}
  local M in
    {Receive G1 M} {Send G2 M}
    {Relay G1 G2}
end end
```

FIGURE 5.17 – La procédure `Relay`

comme nous utilisons une procédure de relais, la communication entre `K'` et `K` devient asynchrone, alors que notre sémantique est construite pour permettre la

synchronisation par échange de messages. De plus, il est impossible, une fois que la procédure `Relay` est lancée, d'interdire par la suite la communication de `K` sur `G`, ce qui est géré par les portes avec la commande `Close`. Enfin, il peut arriver que lorsque `K` exécute la suite de commandes {`Send G X`} {`Recieve G Y`} pour communiquer avec `K'`, le message soit en fait retransmis vers `K` qui reçoit alors son propre message. Les deux autres méthodes sont basées sur les portes, et ouvrent `G` soit dans `K'` soit dans `Server`, ce qui a différentes conséquences. À la base, ces deux méthodes permettent la communication entre `K'` et `K`, ce dernier ne pouvant communiquer qu'avec `K'`. Maintenant, supposons que `K` offre sur `G` un service intéressant pour le reste du programme, et qu'il faille que `K` puisse communiquer avec le parent de `K'`. Dans la première méthode, il faut que ce soit le parent lui-même qui ouvre la porte `G` pour pouvoir accéder aux messages de `K`. Par contre, dans notre dernière méthode, `K'` peut ouvrir la porte `G` pour obtenir le même résultat : dépendamment de où est placée la commande d'ouverture de porte dans notre isolation, les capacités des composants n'intervenant pas dans cette isolation en sont changées. Cela est dû à la sémantique assez subtile de communication, où tout message peut franchir au plus une frontière fermée. Ainsi, l'intérêt des portes de ce langage est mitigé par rapport à celles proposées Chapitre 3. En effet, malgré la sémantique synchrone, qui est un réel apport de ce langage, la manipulation des portes reste un peu trop subtile pour être réellement utilisable en pratique. Comme nous avons vu dans cet exemple, il est assez difficile de savoir qui doit ouvrir une porte lors d'une communication entre deux composants distants, et de plus, ces portes ont le même problème que leur version précédente : il est impossible de contrôler la destination d'un message. En effet, si `K` peut envoyer des messages au père de `K'` sur la porte `G`, il est impossible de faire en sorte que `K'` ne puisse recevoir les messages de `K` et traiter ceux qu'il reçoit à la place de son père. et ce, sans que ni `K` ni le père de `K'` ne puissent le remarquer.

5.3.2 Distribution

Notre langage n'a pas de support explicite de la distribution comme le fait Mozart [29], mais il peut être encodé : les kells servant alors comme substituts aux localités. Notre premier exemple de distribution, présenté Figure 5.18, est relativement simple. Il met en oeuvre deux localités (ou *sites*) `Site1` et `Site2` s'échangeant des messages via le réseau, encodé lui même par un composant appelé `Net`. Ce composant décrit les connections existantes entre les localités présentes dans le système au travers de deux processus transmettant les messages envoyés vers leur destination. La procédure utilisée lors du transfert est celle que nous avons déjà vu dans notre premier exemple, `Relay`, qui permet une communication asynchrone entre deux localités. Enfin, si nous considérons que `Out1` (resp. `Out2`) est la porte d'envoi de messages de `Site1` (resp. `Site2`) et que `In1` (resp. `In2`) est sa porte d'entrée, il est clair que le réseau `Net` connecte les deux sites de notre assemblage. Notons que nous ouvrons les portes du composant `Net` afin qu'il puisse recevoir les messages des deux localités et aussi les

```
kell{Site1} {P1} end
kell{Site2} {P2} end

kell{Net}
    thread {Relay Out1 In2} end
    thread {Relay Out2 In1} end
end
{Open Net 'all'}
```

FIGURE 5.18

transmettre vers leur destination. Notons aussi que nous n'avons pas ouvert les portes d'entrée et de sortie des deux localités `Site1` et `Site2`, car cela leur aurait permis de communiquer directement, sans passer par le composant `Net`, et donc, sans passer par le réseau.

Dans ce premier exemple, le réseau connecte les deux localités `Site1` et `Site2`, mais aucune modification des communications n'est possible. Nous proposons donc un second exemple Figure 5.19 introduisant un composant de réseau quelque peu plus complexe que précédemment et permettant l'ajout et le retrait de connections entre sites. Le principe de base de cet exemple est relativement simple : pour chaque connexion, nous avons un kell dont l'unique but est d'assurer l'échange de messages correspondant. Ainsi, nous pouvons créer une nouvelle connexion en créant un nouveau kell, et la supprimer avec l'opérateur de passivation. Techniquement, nous avons besoin d'une structure permettant de stocker la relation entre les connexions et les kells effectuant les transmissions de messages au sein du réseau. Pour ce faire, nous utilisons une variable, appelée `Map` et différentes procédures permettant de manipuler cette variables et de l'utiliser comme une fonction à domaine fini. Plus précisément, nous avons [5] (i) {`NewMap Map`} qui crée une fonction vide et la stocke dans `Map`, (ii) {`AddMap Map X I`} qui étend le domaine de la fonction, en donnant que l'image de `X` par `Map` est `I`, (iii) {`FindMap Map X I`} qui donne dans `I` l'image de `X` par la fonction `Map` et (iv) {`RemoveMap Map X`} qui ôte `X` du domaine de `Map`. Notre composant réseau, nommé encore une fois `Net`, comporte trois procédures. La première, {`Connect K1 G1 K2 G2`} permet de créer une connexion entre la porte de sortie `G1` de la localité `K1` et la porte d'entrée `G2` de la localité `K2`. Cette connexion est assurée par un nouveau composant `K` exécutant la procédure `Relay` La seconde détruit cette connexion en passivant le composant `K` : ce dernier ne s'exécute plus et donc, la transmission de message est interrompue. Pour ce faire, nous utilisons la variable `Map` stockant la correspondance entre connexion et composant, ainsi que la procédure `FindMap` qui trouve `K`. Finalement, après la passivation du composant, la connexion est effacée de `Map`. Notre dernière procédure est `Run`, et elle permet aux différentes localités présentes de profiter des fonctionnalités

5. Ces différentes procédures peuvent être facilement définissable dans le langage OZ complet, qui inclue par exemple de la programmation impérative

```
kell{Net}
  local Map Connect Disconnect Run in
    {NewMap Map}
    {NewGate Service}

    proc Connect{K1 G1 K2 G2}
      local K in kell{K} {Relay G1 G2} end
        {Open K G1} {Open K G2}
        {AddMap Map (K1,G1,K2,G2) K}
    end end

    proc Disconnect{K1 G1 K2 G2}
      local K in
        {FindMap Map (K1,G1,K2,G2) K}
        {Pack K _}
        {RemoveMap Map (K1,G1,K2,G2)}
    end end

    proc {Run}
      local M in
        {Receive Service X}
        case X of
          `connect'(K1:G1 , K2:G2) then
            {Connect K1 G1 K2 G2}
        else case X of
          `disconnect'(K1:G1 , K2:G2) then
            {Disconnect K1 G1 K2 G2}
        else skip
        {Run}
    end end

    {Run}
end end
{Open Net `all'}
```

FIGURE 5.19

du réseau, c'est à dire du composant `Net` et de ses deux procédures de connexion. Cette procédure utilise une porte `Service` pour recevoir les différentes requêtes de connexion et de déconnexion demandées par les localités. Une demande de connexion est un message de la forme *'connect'*`(K1:G1 , K2:G2)` où `G1` est la porte de sortie de `K1` qu'il faut relier à la porte d'entrée `G2` de la localité `K2`. Une demande de déconnexion a la forme *'disconnect'*`(K1:G1 , K2:G2)` : les deux kells `K1` et `K2` ne veulent plus communiquer au travers des portes spécifiées dans ce message. Finalement, nous ouvrons toutes les portes de `Net` afin qu'il puisse effectivement converser avec toutes les localités présentes dans l'assemblage. Une alternative à ce dernier exemple serait, au lieu d'utiliser des kells pour assurer

le transfert des messages, de demander à de simples processus de réaliser cette tâche. Afin de pouvoir couper une connexion, le processus correspondant testerai alors régulièrement une variable qui prendrait une valeur spéciale lors d'une demande de déconnexion. Cette alternative, bien que très similaire à l'exemple que nous avons donné, n'a pas tout à fait le même comportement : alors que nous passivons les kells dans notre exemple, ce qui pourrait interrompre brutalement le transfert d'un message qui serait alors perdu, cette alternative transmet tout les messages reçus avant la demande de déconnexion. Ainsi, les kells permettent aussi d'encoder les erreurs pouvant survenir sur le réseau, avec la perte aléatoire de message.

D'après ces deux exemples, il semble possible d'encoder dans Oz/K un grand nombre de réseaux de propriétés différentes. L'intérêt de cet encodage est double. Tout d'abord, cela permet de prouver des algorithmes en fonction de certaines hypothèses sur leur environnement d'exécution. Mais cela permet aussi de faire une implémentation assez subtile d'une machine virtuelle distribuée pour ce langage, où chaque instance de la machine construit, en fonction de son environnement matériel, une abstraction de kells et de procédures modélisant les possibilités de communication disponibles. Ainsi, les programmes Oz/K peuvent manipuler le réseau comme n'importe quel autre kell, et communiquer avec d'autres ordinateurs via les abstractions proposés par la machine virtuelle. De plus, bien que nous ayons un store partagé par toutes les localités, le fait que celles-ci n'échangent que des valeurs strictes permet de l'implémenter de manière assez efficace : chaque localité ne possède, lors de sa création, que ses données propres, et les envois de messages sont implémentés par la copie de la donnée dans le store local de localité réceptrice. En effet, comme les messages échangés sont strictes, ils ne possèdent pas de références modifiables et un message n'est alors pas distinguable d'une de ses copies. Ainsi, bien que l'utilisation des portes est contrainte, cette dernière permet en fait d'avoir un modèle de distribution assez efficace, sans nécessiter des références entre différents ordinateurs comme c'est le cas pour Mozart [29].

5.3.3 Mobilité de code

Comme nous venons de le voir, il est assez facile d'échanger des valeurs strictes au travers d'un réseau dans Oz/K. Il suffit en effet d'avoir une machine virtuelle qui crée les abstraction encodant le réseau pour pouvoir l'utiliser. De fait, il est aussi possible d'envoyer des valeurs non strictes (dans notre sémantique actuelle, seulement des procédures, mais une extension à n'importe quelle valeur est facile) à une localité distante, mais cela demande un certain encodage. Considérons en effet la procédure `Pack` : elle prend un kell, le passive avec une copie du store courant et met le tout dans une nouvelle valeur. Cette valeur étant stricte (même si le store contient une variable non stricte), elle peut être envoyé au travers d'une porte, donc à une autre localité qui relance l'exécution du kell et récupère la copie du store de la localité d'origine. Considérons maintenant le programme donné Figure 5.20, dont le but est d'envoyer une procédure `P` non-stricte à une localité `L` que l'on peut joindre par la

porte G. Cet exemple construit tout d'abord un composant temporaire K conte-

```
local K X Y Tmp1 Tmp2 Tmp3 in
  kell{K}
    Tmp1 = Tmp2 + Tmp3
  end
  {Pack K X}
  {Mark X proc(P P) Y}
  {Send G Y}
end
```

FIGURE 5.20

nant un code qui ne finira jamais : l'addition de deux variables sans valeur ne peut se réduire. Il passive ensuite ce kell, faisant par conséquent une copie du store contenant P, et marque P afin que cette procédure ne soit pas renommée lors du re-déploiement du store dans la localité L. Finalement, il envoi la valeur packée résultante sur G, afin que L puisse la recevoir, la re-déployer et utiliser P[6]. Cet exemple illustre comment utiliser la procédure Pack pour envoyer tout type de donnée statique au travers d'un réseau. De plus, l'objectif original de la commande Pack est de transformer les kells et ainsi les processus en valeurs prêtes à être modifiées et envoyées vers une localité distante.

```
proc {ManagePlugin}
  local Plugin M in
    {Receive In M}
    case M of msg(`plugin':PM `gate':GM) then
      kell{Plugin}
          {Unpack {Mark PM gate(GM G)} _}
      end
      {Open Plugin G}
    else skip end
    {ManagePlugin}
end end
```

FIGURE 5.21

Nous illustrons cette capacité dans notre second exemple (Figure 5.21), où nous définissons une procédure recevant des messages contenant un kell ainsi que la porte sur laquelle il communique, et qui le re-déploie dans un nouveau composant. Ce genre de procédure est typiquement utilisée dans le cadre d'un programme évolutif où l'on peut rajouter de nouvelles fonctionnalités durant

6. Notons que pour éliminer le kell temporaire K, L doit re-déployer Y dans un nouveau kell que l'on détruit par la suite : la commande Tmp1 = Tmp2 + Tmp3 est ainsi détruite

son exécution. De fait, cette procédure a un fonctionnement assez simple. Elle consiste en une boucle infinie, où elle attend un nouveau message de la forme `msg(`*'plugin'*`:PM` *'gate'*`:GM)` donnant à la fois le kell à déployer et la porte utilisée pour communiquer avec lui. Tout message ayant une autre forme est ignoré par notre procédure. Une fois le message reçus, la procédure crée un nouveau kell `Plugin` où il place le nouveau code, sans oublier de remplacer dans celui-ci sa porte de communication `PM` par une porte locale `G` prévue pour ce nouveau composant. Enfin, nous ouvrons la porte `G` afin de permettre au nouveau composant de communiquer avec son environnement. Notons que dans cet exemple, nous n'avons pas adopté la méthode présentée Figure 5.16, dans le cas où le code n'est pas de confiance et nécessite d'être isolé du reste du programme (à part pour la porte `G`). En effet, une telle isolation n'est pas nécessaire, car le comportement par défaut de `Unpack` est de renommer toutes les variables du kell re-déployé : celui-ci n'a donc aucun moyen, à part la porte `G`, de communiquer avec son environnement, et donc d'accéder à des informations sensibles.

Néanmoins, il peut arriver que l'on veuille tout de même contrôler finement les communications effectuées par le nouveau composant. Nous illustrons les capacités de contrôle dans la Figure 5.22 où nous reprenons notre exemple précédent, en rajoutant une procédure `Filter` qui intercepte toutes les communications provenant et à destination du greffon. Nous pouvons remarquer que

```
local Filter M Plugin in
  proc{Filter G1 G2} ... end
  {Receive In M}
  case M of msg(`plugin`:PM `gate`:GM) then
    kell{Plugin} local K LG in
      {NewGate LG}
      kell{K}
        {Unpack {Mark PM gate(GM LG)} _}
      end
      thread {Filter LG G} end
    end
  else skip end end
```

FIGURE 5.22

nous utilisons une porte locale, `LG` qui sert à la communication entre le greffon et la procédure `Filter` qui communique, elle, vers le reste du programme via la porte `G`. De plus, le code de cette procédure peut être simple comme très complexe, et a la possibilité de détruire le greffon dans le cas où il s'avère dangereux.

The encapsulation realized by the kell construct allows in particular to build wrappers as in the Boxed-π calculus [101]. For instance, we can build a simple *filtering wrapper* for some untrusted plugin, where the used service is made

```
proc {UpdateManager}
  local M in
    {Receive In M}
    case M of msg(`kell':K `update':Z) then
      {Pack K _}
      kell {K}
        {Unpack Z _}
      end
    else skip end
    {UpdateManager}
end end
```

FIGURE 5.23

available on gate SV.

5.3.4 Gestion des erreurs

La gestion des erreurs dans Oz/K comporte de fortes similarités avec celle qui est faite dans Erlang [10], et aussi avec les évolutions récentes de Oz [28]. Les unités de base pour la gestion des erreurs dans notre langage sont les processus et les composant. Dans notre premier exemple, Figure 5.24 nous définissons une procédure NewThread créant un processus, et contrôlant son état en envoyant un message sur la porte Gate si celui-ci échoue durant son exécution. Cette

```
proc {NewThread Code G} local Proc Moniteur in
    thread{Proc} {Code} end
    thread{Moniteur} local S in
        {Status Proc S}
        case S of failed(Z) then {Send G failed(Proc Z)}
        else skip end
    end end
end end
```

FIGURE 5.24

procédure crée en fait deux processus : Proc est le processus de base qui exécute le code Code en paramètre, alors que Moniteur sert au contrôle de l'état de Proc. Ce contrôle se fait de la manière suivante : Moniteur, avec la commande {Status Proc S}, teste la valeur de la variable d'état de Proc, et donc se bloque jusqu'à l'arrêt de ce processus. Ainsi, lorsque Proc se termine, Moniteur continue son exécution, et dans le cas où la valeur d'état du processus est une valeur d'erreur, envoie un message sur la porte G.

Nous pouvons aussi encoder les *processus dépendants* de Erlang en utilisant

les kells et la procédure de passivation. Dans Erlang, les processus dépendants forment un ensemble de processus qui, soit terminent tous leurs exécution correctement, soit, lorsqu'un des processus de l'ensemble échoue, tous les autres échouent avec lui. Avec Oz/K, nous pouvons créer un tel ensemble en plaçant les processus dans un kell que l'on détruit lorsqu'une erreur survient durant leurs exécutions. Cela est illustré dans notre dernier exemple décrit Figure 5.25. Cet exemple crée, avec l'appel de la procédure `NewThread`, un ensemble de deux

```
local K G M X in
  {NewGate G}
  kell{K}
    {NewThread Code1 G} {NewThread Code2 G}
  end
  {Receive G M}
  case M of failed(T Z) then
    {Pack K X} {Send MG failed(K Z)}
  else skip end
end
```

FIGURE 5.25

processus liés dans un kell `K`. Comme l'état de ces deux processus est contrôlé, il ne suffit plus que d'attendre un message d'erreur sur la porte locale `G` pour passiver le composant `K` et ainsi stopper l'exécution des deux processus [7]. Enfin, nous envoyons la valeur packée résultante sur la porte `MG` afin que l'erreur puisse être traitée.

5.4 Discussion

Une propriété centrale du travail que nous avons entrepris Oz/K est le grand nombre de notion que nous manipulons dans un unique langage. Nous avons déjà discuté en détail de la plupart des aspects de l'isolation et de la mobilité de code (et nous y reviendrons dans le reste de cette partie), et, du fait que nous avons travaillé sur une extension du langage Oz, nous nous sommes aussi penché sur des aspects tels que la conception de langages de programmation et leurs implémentations. Nous avons aussi étudié les similarités qu'il y a entre notre travail et d'autres approches [11, 39, 114, 63, 16, 5, 71, 29]. En particulier, nous nous sommes intéressé aux rapprochements possibles entre les kells et des constructions plus classiques tels que les processus ou les modules.

7. Notons que dans le cas où les deux processus terminent correctement, notre exemple attend infiniment un message sur `G`. Il est possible, avec un encodage un peu plu complexe, d'éviter ce problème

5.4.1 Isolation et implémentation.

Tout au long de notre présentation des portes et des kells, nous avons souligné l'importance des valeurs strictes afin d'assurer l'isolation entre nos composants. Dans notre approche, l'isolation signifie que le seul moyen de communication entre les composants sont les portes : la communication classique de Mozart via le partage de variables entre différentes localités est interdite.

Le partage des variables de Mozart**.** Ce langage structure le store de Oz en différentes parties, avec un store local à chaque localité, plus un store global renseignant entre autre sur les connections existantes entre les localités. Par conséquent, bien que basé sur Oz, Mozart possède des formes de store et de tâche différentes de son langage d'origine, ce qui nécessite la redéfinition complète de Oz dans ce nouveau formalisme. Ce langage comprend huit algorithmes de partage de données, ce qui correspond à trois méthodes différentes : (i) aucun partage, (ii) partage par copie, et (iii) partage par transfert. Le programmeur peut spécifier pour chaque variable du langage quel algorithme utiliser via un mécanisme d'annotation, le mode par défaut étant aucun partage. En conséquence, la gestion du transfert des données entre différentes localités dans Mozart est élégant et souple d'utilisation, mais aussi trop souple, et d'un autre côté, contraint. En effet, Mozart est contraint du fait que le programmeur est limité à huit algorithmes de partage, et il est impossible d'en définir d'autres sans changer le langage. De plus, il est trop souple, car il permet qu'une variable que l'on peut partager entre différents stores fasse référence à une autre qui doit rester locale : cela génère lors de l'exécution du programme des blocages pénibles et difficiles à détecter.

La comparaison avec Oz/K**.** Dans ce travail, nous interdisons le partage des variables non-liées que fait Mozart grâce à l'utilisation des portes et des valeurs strictes. Cette isolation est formalisée dans le théorème suivant (sa preuve est disponible dans [68]) :

Théorème 10. *Supposons donnée une structure d'exécution* $(\sigma, \xi_1 T\ \xi_2 T'\ \mathcal{T}')$ *issue de l'exécution normal d'un terme* Oz/K *et telle qu'il existe* $\kappa_1 \neq \kappa_2$ *avec* $\sigma \vDash \texttt{inth}(\kappa_1, \xi_1) \wedge \texttt{inth}(\kappa_2, \xi_2)$*. Alors, toute variable* x *mentionnée dans* T *telle que* $\sigma \vDash x = \bot$ *n'est pas présente dans* T'*.*

Cette restriction a plusieurs avantages. Tout d'abord, elle empêche la création de blocage présent dans Mozart dû aux demandes de déplacement de variables locales. De plus, cela permet d'avoir une implémentation de la distribution bien plus simple et plus efficace de la distribution dans une machine virtuelle. En effet, comme les valeurs échangées entre kells (et donc entre localités) sont strictes, il est impossible de les distinguer de leurs copies : un envoi d'une référence à une localité distante peut donc être implémenté par la copie en local de la valeur référencée, simplifiant et accélérant du fait les accès futur à la valeur.

Enfin, pour terminer cette discussion sur l'isolation et les portes, nous reprenons les différentes remarques que nous avons faites dans nos exemples : nous

avons étudié ici une nouvelle forme de communication entre composant distant. La plupart des travaux précédents impliquant des localités ou des composants ne permettent pas l'échange de messages entre des composants éloignés, et l'encodent avec des processus de transmission de messages [114, 63, 98, 23]. Parmi les quelques travaux où la communication distante est possible [39, 52], notre utilisation des portes est assez novatrice : elle permet d'avoir une communication synchrone sans utiliser de pointeur. Néanmoins, cette solution n'est pas vraiment satisfaisante, car, comme nous l'avons dit Chapitre 3, l'utilisation des portes est assez difficile en pratique pour la définition d'exemples. De plus, un composant ne peut connaître quelle est la destination d'un message qu'il envoie.

5.4.2 Composants

La structure de composant que nous avons développé dans Oz/K offre de nombreuses fonctionnalités. Nous avons déjà discuté dans nos exemples de l'encodage de la distribution et de la mobilité de code avec nos composants et notre primitive de passivation. Il nous parait donc clair que les capacités de distributions de calculs ou langages tels que [39, 98, 52, 23, 114, 63, 16, 29] sont facilement exprimables dans notre approche. De plus, nos composants peuvent (i) s'organiser en une hiérarchie, ce qui est seulement possible dans [39, 23, 98, 52], (ii) avoir des communication distantes ([39, 52]) et (iii) isoler des parties non-certifiées d'un logiciel afin de se protéger de leurs comportements pouvant être hostiles. Il existe plusieurs approches pour permettre l'isolation entre composants, et comme nous en avons discuté dans la partie précédente, nous pensons que les solutions existantes, y compris nos portes, ne sont pas satisfaisantes et demandent plus d'études.

L'approche que nous avons mené dans ce langage nous a aussi permis de comparer nos kells avec d'autres structures que les localités et composants d'autres approches. En effet, en intégrant les kells dans un langage complet tel que Oz, nous avons pu étudier leurs similarités avec des constructions classiques des langages de programmation, telles que les processus, les modules et les messages structurés.

Les processus. Les kells partagent avec les processus le fait de constituer une structure d'exécution séparée du flot d'exécution principal du programme. Chaque kell, chaque processus s'exécute indépendamment les uns des autres. Il existe toutefois trois différences fondamentales entre un kell et un processus. Tout d'abord, les kells forment une structure hiérarchique alors que les processus sont totalement indépendants les uns des autres. Cette structure est utile en particulier dans le cadre de de la passivation : les processus interdépendants sont assemblés en une unique structure que l'on peut passiver sans mettre en péril la cohérence du programme global. Les composants servent aussi comme structure d'isolation alors que les processus peuvent partager leurs données sans contraintes. Et enfin, les composants possèdent un *nom*, ce qui n'est généralement pas le cas des processus. Les calculs de processus ne nomment en effet pas leur processus, seuls les langages de programmations tels que [29, 48,

11, 108] créent un identifiant unique pour chaque structure d'exécution créée. De fait, le nom des composants permet de les manipuler et en particulier, rend possible la passivation.

Les modules. Les composants sont similaires à des modules dans le sens où ils structurent les programmes en entités distinctes et collaborantes. De même que les modules et les objets, les kells offrent une capacités d'abstraction en n'affichant qu'un ensemble de services (au travers de leurs portes) tout en cachant comment ceux-ci sont implémentés. Ainsi, en considérant les kells comme des modules, notre approche permet de construire des foncteurs, mais ne possède aucune capacité quant à la création de mixin [37]. Néanmoins, les kells ont trois avantages par rapport aux simples modules : il est possible de les passiver et de manipuler leur contenu ; et ils forment des structures d'exécution isolées et indépendantes du flot d'exécution principal du programme, ce qui permet par exemple de contrôler leur exécution dans le cas où le modules est un greffon d'origine inconnue et peut-être dangereuse.

Les messages structurés. Finalement, comme il est possible des passiver un kell, nous pouvons aussi le considérer comme une valeur, et en particulier un message structuré. En effet, un kell offre des services sur ses portes, correspondant aux différents champs d'un message structuré. De plus, les capacités de remplacement des portes et des procédures associées dans nos valeurs packées correspondent assez bien au remplacement de champ dans un message. Considérant l'inspection de la structure d'un composant passivé, il est aussi possible de voir chaque sous-composant comme un champ de message, distinct des autres et contenant lui-même une structure interne. Nous nous rapprochons ainsi du modèle à objet, avec les portes correspondant aux méthodes et les sous-composants constituant l'état de chaque kell passivé. La comparaison reste toutefois assez limitée, car est encore impossible dans notre approche d'ôter ou de rajouter un champ d'un composant passivé : ce genre de modification n'est possible que lorsque le composant est actif. Ainsi, les kells n'offrent encore que peu de fonctions d'exploration et de manipulation (une remarque que nous avons d'ailleurs déjà faite), et il serait intéressant d'étudier de telles opérations. Du côté des messages, il est encore impossible de les transformer en structure d'isolation et d'exécution.

Ainsi, les kells, bien que conçus pour assurer un ensemble de tâches spécifiques (structurer, isoler et passiver), ont beaucoup de points communs avec des constructions plus classiques des langages de programmation. Il serait intéressant d'étudier chaque tâche, chaque similarité individuellement, afin de bien comprendre la notion que nous appelons *composant* [107].

5.4.3 Langage

Si l'on compare maintenant globalement le langage Oz/K à l'existant, nous pouvons voir qu'il existe déjà plusieurs travaux tentant de rassembler dans un

unique modèle des notions de structuration, d'isolation et de manipulation de code.

Lisp. Déjà dans les années soixante-dix, le langage Lisp [99] considérait le code comme une liste que l'on pouvait parcourir et interpréter, offrant ainsi une notion très puissante de manipulation de code. Comparativement à notre approche, cette notion est bien plus expressive mais ne permet aucun contrôle quant aux objets que l'on peut modifier.

Java et ses extensions. Plus récemment, le langage Java [11] offre lui aussi des opérations permettant (i) l'ajout de nouvelles classes dans un programme en cours d'exécution, (ii) l'utilisation de threads (iii) et une certaine forme d'isolation avec les méthodes privées et publiques. Néanmoins, les capacités de ce langage concernant ces trois points sont encore très limités. Par exemple, l'isolation n'est testée qu'a la compilation du code source, et peut très facilement être contournée avec les méthodes de la classe `Class`. De plus, la méthode permettant la destruction ou la reprise de l'exécution d'un thread a été ôtée de l'API Java (entre les version 1.4.2 et 1.5.0 [8]) car elle pouvait causer des incohérences dans l'état des objets (les verrous en particulier). Enfin, java ne permet pas la passivation.

ArchJava [5] et Classages [71] offrent une notion très classique de composants que l'on peut considérer comme des éléments distincts d'un programme, organisés en une hiérarchie, et communicant via des *connecteurs* similaires aux interfaces d'autres modèles. Néanmoins, en comparaison avec Oz/K, ces deux langages ne considèrent pas leur composants comme des entités d'exécution indépendantes : ils se rapprochent ainsi de la notion de module avec des limitations similaires. En particulier, une erreur se produisant dans un composant de ArchJava utilisé comme greffon peut condamner l'exécution de tout un programme. Finalement, aucun de ces deux langages n'offre de réelles opérations de passivation, de mobilité de code et d'isolation.

Scala [84] est un langage différent de Java, mais s'exécutant sur la même machine virtuelle. Ce langage propose une notion d'acteur assez similaire à nos kells du point de vue de la concurrence et des communications. En effet, chaque acteur s'exécute indépendamment les uns des autres, et communiquent via des envois de messages. Néanmoins, les acteurs ne constituent pas une hiérarchie, et donc ne permettent aucune communication à distance et aucune isolation. De plus, la passivation n'est pas non plus supportée par le langage.

Les extensions de ML. Alice [94] peut être considéré comme une extension de ML [76] qui offre des modules pouvant s'inclure les uns dans les autres et des primitives de concurrences avec futurs et exécution paresseuse. La notion de module de ce langage permet aussi de créer des foncteurs plus généraux que

8. Dans cette version du langage, il est seulement possible de suspendre l'exécution d'un thread avec la méthode `interrupt`, sans aucune possibilité de la relancer

OCaml, mais reste toutefois assez éloignée des aspects concurrentiels et d'isolation des composants de Oz/K. L'intérêt principal de ce langage toutefois est son typage fort qu'il serait intéressant d'étudier pour l'adapter à nos constructions.

Acute [100] est aussi un langage de la famille ML, lui aussi fortement typé, et avec un support important pour la gestion des modules, leur rajout à chaud et la manipulation de leurs différentes versions. De plus, ce langage inclue de la concurrence via des processus et aussi une forme de passivation où il est possible de capturer l'état d'un processus en cours d'exécution. Néanmoins les mécanismes que Acute utilise sont relativement complexes, avec en particulier la notion de *marque* pour contrôler la visibilité d'un module dynamique. Ainsi, notre approche comportant trois notions simples (les composants, les portes et la passivation) nous semble plus simple à appréhender que Acute, et permet en plus de ce langage un notion de distribution, des communications distantes et des moyens d'isolation.

D'autres langages s'intéressent à des problématiques similaires à celles traitées dans ce chapitre. Nous citons entre autres le langage E [74] qui se focalise sur la sécurité, et Sing# [4] qui est développé dans le cadre du projet Singularity [53]. Ces deux langages offrent un bon support pour les notions classiques de processus communicant par l'intermédiaire de messages, mais n'intègrent pas d'opérations de structuration, de manipulation dynamique de code que nous développons dans Oz/K. Il est aussi intéressant de noter qu'une autre approche [55] que Mozart à déjà été faite pour intégrer des localités dans Oz : celle-ci s'inspire du kell-calcul pour ajouter à Oz une notion de *membrane* dont le seul rôle est de contrôler les communications entre différentes parties d'un programme.

5.5 Conclusion

Dans ce chapitre, nous avons présenté le langage noyau Oz/K. La définition de ce langage avait plusieurs objectifs : (i) le premier était de donner une définition claire à la notion encore assez vague de composant ; (ii) la seconde était d'étudier les constructions permettant de créer, d'assembler et de modifier les composants ; et enfin (iii), nous voulions aussi tester l'intégration de ces éléments dans des paradigmes de programmation classiques, afin de voir de degré de corrélation entre nos composants et les constructions déjà présentes dans les langages existants. Ainsi, Oz/K se base sur le langage Oz [110], auquel nous rajoutons (i) une structure de composant, (ii) des portes pour permettre la communication distante entre deux composants, et (iii) la passivation qui est étendue par rapport à [98, 51] avec des opérations de modification de composants passivés. Le résultat est un langage très expressif qui associe à la fois de l'isolation, de la mobilité de code et une notion implicite de distribution, et qui dispose d'un sémantique précise, proche de celle d'une machine virtuelle. Malgré ses différents avantages, nous ne considérons pas ce langage comme abouti, et beaucoup de travail reste encore à accomplir pour obtenir un résultat satisfai-

sant. Le système de porte a en effet besoin d'être changé. Nous pensons par la suite étudier des constructions similaires à des *interfaces* [17, 7, 80] et à la primitive **connect** des sessions [111]. Les interfaces sont en effet largement utilisées en pratique, et les sessions ont la propriété d'assurer une communication entre deux processus bien identifiés. Concernant les composants, il serait certainement très intéressant d'étudier plus en détail les caractéristiques des kells de notre langage. En effet, les kells sont proches des processus (car ils forment des entités d'exécution distinctes du reste du programme), mais aussi des modules (car ils offrent des services) et des messages structurés (car on peut les échanger). Bien que ce travail a permis une première étude de la notion de composant, il serait intéressant de continuer cette étude pour aboutir à une définition formelle satisfaisante de ce qu'est un composant. Finalement, un dernier sujet d'amélioration est l'opération de passivation. Pour le moment, les opérations de manipulations de kells passivés restent assez élémentaires. Il serait intéressant de rajouter des opérations permettant la passivation non pas que de kells, mais aussi d'autres valeurs (ce qui est déjà possible de faire dans Oz/K avec un encodage). De plus, l'exploration de la structure d'un kell packé ainsi que sa modification à la [105] seraient des opérations intéressantes à rajouter au langage.

Chapitre 6

Conclusion

Dans ce document, nous avons présenté les quatre travaux principaux que nous avons entrepris dans notre thèse. Le premier s'est tout d'abord intéressé à la construction d'un langage à composant simple, basé sur le kell-calcul [98] et agrémenté d'une procédure de routage basé sur la structure des messages transmis. Nous avons aussi défini sur ce calcul un système de type intuitif permettant de contrôler la manipulation des messages par les composants, et ainsi s'assurer qu'un programme typable n'opère que des opérations valides sur les messages durant son exécution. Notre seconde approche consiste ensuite en une évolution de ce premier travail où le calcul et le système de type sont largement modifiés pour y inclure beaucoup d'éléments novateurs. Le calcul est maintenant totalement basé sur les composants, ce qui le rend assez proche des ADLs existant [17, 80, 6], comporte des portes permettant la communication à distance et aussi une évolution du système précédent de routage, basé sur des annotations portées par les messages. Le système quant à lui inclue du sous-typage et possède une nouvelle forme de type, les types routés, afin de gérer convenablement le routage de ce nouveau calcul. Ce système de type admettant une d'inférence décidable, notre troisième travail fut de construire un algorithme d'inférence et de l'implémenter sur deux architecture à composant : Dream [65] et Click [80]. Enfin, en parallèle à ces différents travaux, nous nous sommes intéressés aux modèles à composant et avons construit un langage noyau basé sur le langage Oz. Notre approche reprend intégralement Oz en lui ajoutant trois nouvelles constructions : (i) les composants appelés *kells*, (ii) les portes pour le support des communications distantes et (iii) la passivation avec un ensemble réduit de primitives pour la modification des composants passivés.

6.1 Contributions

Le routage structurel. La première nouveauté qu'apporte ce travail de thèse est le *routage structurel*, l'étude de son expressivité ainsi que de son typage. Le principe de cette forme de routage est de différencier les messages reçus

en fonction de leurs structures, de leurs types. Ainsi, des messages de types différents peuvent être envoyés sur des canaux différents pour y être traités par des processus adaptés. À notre connaissance, il n'existe aucune étude dans les calculs de processus concernant le routage, et ce travail est donc innovant, même s'il reste assez proche des travaux sur l'analyse intentionnelle de type faite sur le langage ML. Du côté du système de type étend les travaux précédent sur les type de processus en permettant d'avoir plusieurs type de message par canal. Cette approche permet une gestion souple du routage, et nous avons aussi prouvé que son inférence de type est indécidable.

Le routage sémantique. La seconde innovation de notre travail est l'introduction d'une nouvelle forme de routage : le *routage sémantique*. Son principe est d'annoter chaque message par un ensemble modifiable de tags. Le routage se base alors sur ces annotations pour différencier les messages et les envoyer sur des canaux différents pour y être traités par des processus adaptés. Cette approche du routage vient avec un système de type adapté introduisant une nouvelle catégorie de types : les *types routés*. Cette forme de type est bien adaptée pour la gestion du routage sémantique, et reste suffisamment contrainte pour que l'inférence de type soit décidable. De fait, nous avons défini un algorithme d'inférence pour ce système de type, et nous l'avons implémenté pour deux systèmes de composants (Click et Dream).

La notion de composant. Finalement, les travaux entrepris dans cette thèse nous ont permis d'étudier la notion de composant et de tester certaines idées concernant ce modèle. Concernant l'isolation entre composant et la communication distante, nous avons introduit le concept de *portes*. Chaque porte correspond à un canal de communication que l'on peut ouvrir ou fermer sur une frontière de composant, permettant ainsi aux messages envoyé sur ce canal de franchir ou non cette frontière. Cette notion permet donc d'avoir à la fois des frontières de composant isolantes (lorsque les portes sont fermées) et des communications distantes lorsque la même portes est ouvertes sur toute une hiérarchie de composants. De plus, cette notion de communication distante nous permet d'exprimer dans notre modèle de composant localités et réseaux, avec leurs différentes caractéristiques.

Nous avons de plus étendu le concept de la passivation avec la gestion des conflits de noms de canaux, ainsi que des fonctions de manipulation des composants passivés. Il est apparu que cette extension permet une gestion simplifiée des greffons et de la mise à jours de programmes.

6.2 Perspectives

Le travail que nous avons effectué dans cette thèse n'apporte pas de solution définitive ni à la gestion du routage, ni à la notion de composant, mais apporte quelques pistes de recherche intéressantes pour le futur. Concernant le modèle de composant, nous avons vu que le concept de portes n'était pas

satisfaisant pour gérer les communications à distance ou modéliser le partage. Une approche intéressante à étudier serait d'adapter le modèle de sessions pour modéliser les interfaces des modèles à composant courant. Cette approche a l'intérêt d'assurer qu'un échange de messages entre deux composants ne puisse être intercepté par un tiers écoutant sur le même canal de communication. Un autre sujet d'étude est la structure associée aux composants. En effet, comme nous avons pu le voir dans le Chapitre 5, la notion de composant est liée à beaucoup d'éléments différents, comme la structuration du code, l'isolation des communications, l'indépendance d'exécution ou la structure des composant passivés. Il est encore assez difficile de voir comment ces différentes notions interagissent, et il n'existe encore aucune façon claire et satisfaisante de les définir ou de les manipuler. Il serait donc intéressant dans un premier temps d'identifier précisément les éléments impliqués dans la structure de composant, avant de définir les bonnes opérations pour les manipuler. Enfin, concernant les systèmes de type, il nous reste deux sujets d'étude principaux à investiguer. Tout d'abord, nous devons assouplir le typage du routage sémantique qui est contraint par la forme de nos types routés. Ce travail est déjà bien avancé. Notre second sujet est l'extension de notre approche actuelle en y intégrant des éléments nouveaux, comme le typage de primitive de manipulation de configurations (par exemple, la passivation ou l'ouverture et la fermeture de porte), ou bien l'ajout de tests sur la structures des composants afin de vérifier par exemple que les interfaces sont bien toutes connectées, ou qu'un composant donné possède une structure interne spécifique. Les types de sessions [111] semblent être un bon point de départ pour typer la manipulation des assemblages car ils intègrent une notion d'évolution de capacités. Finalement, les travaux sur le XML [43] ou la logique spatiale des Ambients [18] peuvent nous guider quant à la définition d'un système de type vérifiant la correction de la structure d'un assemblage de composant.

Bibliographie

[1] A. Aiken. Set constraints : Results, applications, and future directions. http ://theory.stanford.edu/∼aiken/publications/publications.html.

[2] A. Aiken, D. Kozen, and E. Wimmers. Decidability of systems of set constraints with negative constraints. Technical report, Ithaca, NY, USA, 1993.

[3] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[4] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *MSPC '06 : Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 1–10, New York, NY, USA, 2006. ACM.

[5] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings 16th European Conf. on Object-Oriented Programming*, 2002.

[6] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology, vol. 6, no 3*, 1997.

[7] OSGi Alliance. *Osgi Service Platform, Release 3*. IOS Press, Inc., 2003.

[8] Farhad Arbab. Abstract behavior types : A foundation model for components and their composition. *Formal Methods for Components and Objects*, pages 33–70, 2003///.

[9] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[10] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, Stockholm, Sweden, 2003.

[11] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, 4th edition*. Addison-Wesley, 2005.

[12] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote : A system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4), 1998.

[13] P. Bidinger, M. Leclercq, V. Quéma, A. Schmitt, and J.B. Stefani. Dream Types - A Domain Specific Type System for Component-Based Message-Oriented Middleware. In */4th Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05), in association with ESEC/F-SE'05/*, 2005.

[14] Philippe Bidinger and Jean-Bernard Stefani. The Kell calculus : operational semantics and type system. In *Proceedings 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 03)*, 2003.

[15] G. Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, INRIA, 1992.

[16] Gérard Boudol. Ulm : A core programming model for global computing : (extended abstract). In David A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2004.

[17] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[18] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part i). *Inf. Comput.*, 186(2) :194–235, 2003.

[19] L. Cardelli. A semantics of multiple inheritance. *Inf. Comput.*, 76(2-3), 1988.

[20] L. Cardelli. Structural subtyping and the notion of power type. In *POPL '88 : Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–79, New York, NY, USA, 1988. ACM.

[21] L. Cardelli. Program fragments, linking, and modularization. In *POPL '97 : Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 266–277, New York, NY, USA, 1997. ACM.

[22] L. Cardelli. Types for mobile ambients. In *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1999.

[23] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures, M. Nivat (Ed.), Lecture Notes in Computer Science, Vol. 1378*. Springer Verlag, 1998.

[24] L. Cardelli and J. Mitchell. Operations on records. In *Mathematical Foundations of Programming Semantics*, pages 22–52, 1990.

[25] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for web services. *Web Services and Formal Methods*, pages 148–162, 2006///.

[26] G. Castagna, J. Vitek, and F. Zappa Nardelli. The seal calculus. *Inf. Comput.*, 201(1), 2005.

[27] R. Collet. *The Limits of Network Transparency in a Distributed Programming Language*. PhD thesis, Université catholique de Louvain, Belgium, 2007.

[28] R. Collet and P. Van Roy. Failure Handling in a Network-Transparent Distributed Programming Language. In *Recent Advances in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*. Springer, 2006.

[29] Raphael Collet. *The Limits of Network Transparency in a Distributed Programming Language*. PhD thesis, Université catholique de Louvain, December 2007.

[30] Denis Conan, Romain Rouvoy, and Lionel Seinturier. Scalable processing of context information with cosmos. In Jadwiga Indulska and Kerry Raymond, editors, *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'07)*, Lecture Notes in Computer Science, Paphos, Cyprus, June 2007. Springer-Verlag.

[31] S. Conchon and F. Pottier. Join(x) : Constraint-based type inference for the join-calculus. In *10th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2001.

[32] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. OpenCOM v2 : A Component Model for Building Systsms Software. In *Proceedings of IASTED Software Engineering and Applications (SEA '04)*, 2004.

[33] K. Crary and S. Weirich. Flexible type analysis. In *ICFP '99 : Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 233–248, New York, NY, USA, 1999. ACM.

[34] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP '98 : Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 301–312, New York, NY, USA, 1998. ACM.

[35] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82 : Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[36] Benjamin Pierce Davide and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Mathematical Structures in Computer Science*, pages 376–385, 1996.

[37] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *ICFP '96 : Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 262–273, New York, NY, USA, 1996. ACM.

[38] Johan Eker, JW. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. pages 127–144, New York, NY, USA, 2003. IEEE Computer Society.

[39] C. Fournet and G. Gonthier. The join calculus : A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal*, pages 268–332. Springer-Verlag, 2002.

[40] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *In Proceedings 7th International Conference on Concurrency Theory (CONCUR '96), Lecture Notes in Computer Science 1119*. Springer Verlag, 1996.

[41] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ml for the join-calculus. In *In Proceedings 8th International Conference on Concurrency Theory (CONCUR '97), Lecture Notes in Computer Science 1243*. Springer Verlag, 1997.

[42] D. Garlan, R. Monroe, and D. Wile. *Acme : Architectural Description of Component-Based Systems*, chapter 3. Cambridge University Press, 2000.

[43] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07 : Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 342–351, New York, NY, USA, June 2007. ACM Press.

[44] Jean-Yves Girard. The system f of variable types, fifteen years later. *Theor. Comput. Sci.*, 45(2) :159–192, 1986.

[45] D. Grolaux, K. Glynn, and P. Van Roy. A fault tolerant abstraction for transparent distributed programming. In *Multiparadigm Programming in Mozart/Oz, 2nd International Conference, MOZ 2004*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.

[46] Object Management Group. *CORBA Components – OMG document orbos/99-02-01*. 1999.

[47] Carl A. Gunter. *Semantics of programming languages : structures and techniques*. MIT Press, Cambridge, MA, USA, 1992.

[48] Samuel P. Harbison and Guy L. Steele. *C, a Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[49] S. Haridi, P. Van Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka. Efficient logic variables for distributed computing. *ACM Trans. Program. Lang. Syst.*, 21(3), 1999.

[50] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2) :253–289, 1993.

[51] Thomas Hildebrandt, Jens Chr. Godskesen, and Mikkel Bundgaard. Bisimulation congruences for homer — a calculus of higher order mobile embedded resources. Technical Report TR-2004-52, IT University of Copenhagen, 2004.

[52] D. Hirschkoff, T. Hirschowitz, D. Pous, A. Schmitt, and J.B. Stefani. Component-Oriented Programming with Sharing : Containment is not Ownership. In *4th International Conference on Generative Programming and Component Engineering (GPCE)*, Tallinn, Estonia, September 2005.

[53] Galen C. Hunt and James R. Larus. Singularity : rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2) :37–49, 2007.

[54] V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment : Experience with the Aster Prototype. In *4th Int. Conf. on Configurable Distributed Systems*, 1998.

[55] Yves Jaradin, Fred Spiessens, and Peter Van Roy. Capability confinement by membranes. Technical Report RR2005-03, Department of Computing Science and Engineering, Université catholique de Louvain, 2005.

[56] L. Jategaonkar and J. Mitchell. Ml with extended pattern matching and subtypes. In *LFP '88 : Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 198–211, New York, NY, USA, 1988. ACM.

[57] B. Jay and D. Kesner. *Pure Pattern Calculus*, pages 100 – 114. 2006.

[58] Wells J.B. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98 :111–156(46), 30 June 1999.

[59] A. Joolia, T. Batista, G. Coulson, and A. Gomes. Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In *5th IEEE/IFIP Conference on Software Architecture (WICSA'05)*. IEEE Computer Society, 2005.

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

[61] Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 439–453. Springer, 2003.

[62] E. Kohler. *The Click Modular Router*. PhD thesis, MIT, MA, USA,, 2000.

[63] L.Bettini, V.Bono, R.De Nicola, G.Ferrari, D.Gorla, M.Loreti, E.Moggi, R.Pugliese, E.Tuosto, and B.Venneri. The klaim project : Theory and practice. In *Global Computing : Programming Environments, Languages, Security and Analysis of Systems*, number 2874 in Lecture Notes in Computer Science. Springer-Verlag, 2003.

[64] M. Leclercq, A. E. Ozcan, V. Quema, and J.B. Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE '07 : Proceedings of the 29th International Conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[65] M. Leclercq, V. Quema, and J.B. Stefani. DREAM : a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9), 2005.

[66] C. Lhoussaine. Type inference for a distributed π-calculus. *Sci. Comput. Program.*, 50(1-3), 2004.

[67] M. Lienhardt, A. Schmitt, and J-B. Stefani. Oz/k : A kernel language for component-based open programming. In *GPCE'07 : Proceedings of the 6th international conference on Generative Programming and Component Engineering*, pages 43–52, New York, NY, USA, 2007. ACM.

[68] M. Lienhardt, A. Schmitt, and J-B. Stefani. Oz/k : A kernel language for component-based open programming. Technical Report RR-6202, INRIA - France, 2007.

[69] M. Lienhardt, A. Schmitt, and J.-B. Stefani. A type system for the Dream framework, 2007. http ://sardes.inrialpes.fr/papers/files/rr-dreamtypes.pdf.

[70] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In *ACM Symposium on Operating Systems Principles*, 1999.

[71] Y.D. Liu and S. Smith. Interaction-Based Programming with Classages. In *Proceedings OOPSLA*, 2005.

[72] D. MacQueen. Modules for standard ml. In *LFP '84 : Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM.

[73] Sergio Maffeis. Sequence types for the pi-calculus. In *ITRS'04*, volume 136 of *ENTCS*, pages 117–132. Elsevier, 2005.

[74] M. Miller. *Robust Composition : Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, baltimor, Maryland, USA, 2006.

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

[76] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[77] Robin Milner. *Communicating and mobile systems : the π-calculus.* Cambridge University Press, New York, NY, USA, 1999.

[78] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.

[79] H. Miranda, A. S. Pinto, and L. Rodrigues. Appia : A flexible protocol kernel supporting multiple coordinated channels. In *21st International Conference on Distributed Computing Systems (ICDCS 2001).* IEEE Computer Society, 2001.

[80] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *ACM Symposium on Operating Systems Principles*, 1999.

[81] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228, London, UK, 1984. Springer-Verlag.

[82] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1) :35–55, 1999.

[83] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9 :49–67, 1997.

[84] Haller Philipp and Odersky Martin. Actors that unify threads and events. In *Coordination Models and Languages*, pages 171 – 190, 2007.

[85] B. Pierce and D. Turner. Pict : A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction : Essays in Honour of Robin Milner*. MIT Press, 2000.

[86] Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52 :264–268, 1946.

[87] F. Pottier. Type Inference in the Presence of Subtyping : from Theory to Practice. Technical report, CRISTAL - INRIA Rocquencourt - INRIA, 1998.

[88] F. Pottier. Simplifying subtyping constraints : A theory. *INFCTRL : Information and Computation (formerly Information and Control)*, 170, 2001.

[89] F. Pottier. A constraint-based presentation and generalization of rows. *LICS*, 0 :331, 2003.

[90] F. Pottier and D. Rémy. The Essence of ML Type Inference. In *B. Pierce (ed), Advanced Topic in Types and Programming Languages*. MIT Press, 2005.

[91] François Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4) :312–347, 2000.

[92] D. Rémy. Type checking records and variants in a natural extension of ml. In *POPL '89 : Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–88, New York, NY, USA, 1989. ACM.

[93] D. Rémy. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[94] A. Rossberg. The Missing Link – Dynamic Components for ML. In *Int. Conf. Functional Programming (ICFP)*, 2006.

[95] Romain Rouvoy and Philippe Merle. Leveraging component-based software engineering with fraclet. *Annales des Télécommunications*, 64(1-2) :65–79, 2009.

[96] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in distributed oz. *ACM Trans. Program. Lang. Syst.*, 19(5), 1997.

[97] A. Schmitt and J.B. Stefani. The M-calculus : A Higher-Order Distributed Process Calculus. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[98] A. Schmitt and J.B. Stefani. The Kell Calculus : A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*. Springer, 2005.

[99] Peter Seibel. *Practical Common Lisp.* APress, 2004.

[100] P. Sewell, J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Willians, P. Habouzit, and V. Vafeiadis. Acute : High-level programming language design for distributed computation. In *Int. Conf. Functional Programming*, 2005.

[101] Peter Sewell and Jan Vitek. Secure Composition of Untrusted Code : Box pi, Wrappers, and Causality. *Journal of Computer Security*, 11(2), 2003.

[102] Vincent Simonet. An extension of hm(x) with bounded existential and universal data-types. In *ICFP '03 : Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 39–50, New York, NY, USA, 2003. ACM.

[103] Vincent Simonet. Type inference with structural subtyping : A faithful formalization of an efficient constraint solver. In *APLAS*, pages 283–302, 2003.

[104] Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Program. Lang. Syst.*, 29(1) :1, 2007.

[105] Gareth Stoyle, Michael W. Hicks, Gavin M. Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis : safe and predictable dynamic software updating. In *POPL*, pages 183–194, 2005.

[106] Sun Microsystems. JSR 220 : Enterprise JavaBeans, Version 3.0 – EJB Core Contracts and Requirements, 2006.

[107] C. Szyperski. *Component Software, 2nd edition.* Addison-Wesley, 2002.

[108] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries : International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1 (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[109] A. Unyapoth and P. Sewell . 2001. Nomadic Pict : Correct Communication Infrastructures for Mobile Computation. In *Proceedings ACM Int. Conf. on Principles of Programming Languages (POPL)*, 2001.

[110] P. van Roy and S. Haridi. *Concepts, Techniques and Models of Computer Programming.* MIT Press, 2004.

[111] Vasco T. Vasconcelos. *9th International School on Formal Methods for the Design of Computer, Communication and Software Systems : Web Services (SFM 2009)*, volume 5569 of *LNCS*, chapter Fundamentals of Session Types, pages 158–186. Springer, 2009.

[112] M. Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual Symposium on Logic in Computer Science*, pages 92–97, Asilomar Conference Center, Pacific Grove, CA, 1989. IEEE Computer Society Press.

[113] S. Weirich. Higher-order intensional type analysis. In *11th European Symposium on Programming Languages and Systems.* Springer-Verlag, 2002.

[114] P. Wojciechowski and P. Sewell. Nomadic Pict : Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.

[115] N. Yoshida. Channel dependent types for higher-order mobile processes. *SIGPLAN Not.*, 39(1) :147–160, 2004.

[116] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher-order processes. In *Proceedings CONCUR 99, Lecture Notes in Computer Science no 1664*. Springer, 1999.

[117] N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 334–345, 2000.

[118] N. Yoshida and M. Hennessy. Assigning types to processes. *Inf. Comput.*, 174(2), 2002.

[119] N. Yoshida and V. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited : Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4), 2007.

Annexe A

Annexe du chapitre 2

A.1 Les types principaux

Définition 12. *Une substitution* généralisée Σ *est un ensemble fini de substitution* σ*. Nous définissons l'application d'une substitution généralisée* Σ *à un type de processus* S *de manière inductive comme il suit :*

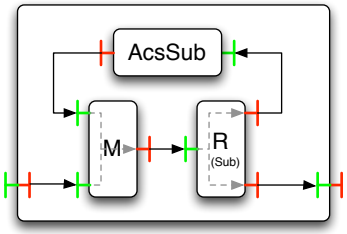
$$\begin{array}{l}
\{\sigma_i \mid i \in I\}(e : (T)) \triangleq \begin{cases} e : (T) & \textit{lorsque } fv(T) = \emptyset \\ \bigcup_{i \in I} \sigma_i(e : (T)) & \textit{sinon} \end{cases} \\
\Sigma(e) \triangleq e \\
\Sigma(S \cup S') \triangleq \Sigma(S) \cup \Sigma(S')
\end{array}$$

Nous supposons dans le reste de cette partie que le système de type utilise les substitutions généralisées dans le prédicat d'instantiation. Comme les substitutions généralisées incluent les substitutions, les processus typables avec les simples substitutions le sont toujours. Il en découle que la propriété de *stabilité* du système de type est conservée, et il est facile de voir que la *correction* est elle aussi toujours valide.

Définition 13. *Supposons donné un environement de typage* Γ *et un processus* D*. Un type* S *est* principal *pour* D *et* Γ *s'il existe une dérivation de* $S, \Gamma \vdash D$ *et si, pour tout type* S' *tels que* $S', \Gamma \vdash D$*, nous avons* $\Gamma \vdash S' \Leftarrow S$*. De manière générale, nous disons qu'un processus admet un type principal s'il existe* S*, principal pour* D *et* $\emptyset$*.*

Avec les substitutions généralisées, nous pouvons facilement donner des types principaux aux composants que nous avons présentés jusqu'à présent. Le cas du composant **TheLoop** (que nous présentons une dernière fois Figure A.1 [1]) est assez intéressant. Nous avons en effet trouvé un type principal pour ce composant, alors que son comportement particulier nous laissait penser

1. remarquons que le code du processus est une simplification de l'assemblage présenté, avec le même comportement



$b[e(x).\mathrm{IfPre}(\mathtt{Sub}, x, i, s) \mid i(x).\overline{e}\langle x.\mathtt{Sub}\rangle]$

FIGURE A.1 – Le composant TheLoop

qu'un tel type n'existait pas. Considérons les types $T \triangleq \forall\eta, \rho.\{\mathtt{Sub} : \mathrm{Pre}(\eta); \rho\}$, $T' \triangleq \forall\rho.\{\mathtt{Sub} : \mathrm{Abs}; \rho\}$ et $S \triangleq e : (T) \cup e : (T') \cup i : (T) \cup s : (T')$. Il est assez facile de voir (informellement) que ce type est suffisamment général pour pouvoir s'instancier en tout les types possibles de l'assemblage **TheLoop**. La preuve d'un tel résultat n'est pas développée ici. Par ailleurs, ce type est aussi un type valide pour **TheLoop**, comme nous le montre la dérivation de type J présentée Figure A.2. Nous pouvons remarquer dans cette dérivation, et plus particulièrement dans la sous dérivation J_2, une particularité intéressante. Dans cette dérivation, nous typons le composant `AcsSub` par $S_s \triangleq i : (T) \cup e : (T')$ en instanciant la variable η en T' lors du typage du canal e : notre système de type permet de faire cette substitution de manière locale, sans changer η dans le context de typage, car elle est gardée par un schéma de type. En conséquence, le type S_s cache la relation pourtant évidente qu'il y a entre l'entrée et la sortie du composant `AcsSub`. Ce genre de manipulation permet de donner un type principal à notre assemblage **TheLoop**, mais au coût de rendre nos types inutiles quant à la caractérisation des comportements des composants typés. De manière générale, tout assemblage sans message sur un canal est typable, même lorsque son comportement est incohérent. Nous considérons cela comme une mauvaise propriété de notre système de type, d'autant plus que malgré son expressivité, nous n'avons pas de types principaux. En effet, considérons le composant **Plus1** $\triangleq b[e(x).\overline{s}\langle x+1\rangle]$. Comme tout assemblage typable et sans boucle, ce composant admet un type évident $S_1 \triangleq e : (\mathrm{int}) \cup s : (\mathrm{int})$ et un type minimal $S_2 \triangleq e$. Raisonnons maintenant par contradiction, et supposons que ce composant admet un type principal S. Comme $S_1(e)$ est non vide, il est clair que $S(e)$ est lui non plus, non vide. Il est facile de vérifier alors, avec une dérivation de type assez simple, que $S(s) = \{\mathrm{int}\}$. Par conséquence, S n'est pas instanciable en S_2, ce qui contredit notre hypothhèse. Ainsi **Plus1** n'admet pas de type principal, et ce pour un raison très simple : notre système de type, avec nos substitutions généralisées, permet de manipuler les variables de types de manière très flexible, mais il en est totalement différents pour les types totalement instanciés. Dans notre cas, la sortie du composant **Plus1** est obligatoirement de type int qui ne comporte aucune variable. Il est alors impossible d'éliminer ce type pour unifier S avec le type minimal S_2. Une possibilité se-

$$J_1 \triangleq \frac{x : T \vdash x : T \qquad x : T \vdash T \Leftarrow \forall \eta, \rho^{\{\mathtt{Sub}\}}.\{\mathtt{Sub} : \mathrm{Pre}(\eta); \rho^{\{\mathtt{Sub}\}}\} \qquad T \in S(i)}{S, x : T \vdash \mathrm{IfPre}(\mathtt{Sub}, x, i, s)}$$

$$J_1' \triangleq \frac{x : T' \vdash x : T' \qquad x : T' \vdash T' \Leftarrow \forall \rho^{\{\mathtt{Sub}\}}.\{\mathtt{Sub} : \mathrm{Abs}; \rho^{\{\mathtt{Sub}\}}\} \qquad T' \in S(s)}{S, x : T' \vdash \mathrm{IfPre}(\mathtt{Sub}, x, i, s)}$$

$$J_2 \triangleq \left(\frac{\dfrac{\dfrac{\dfrac{x;T \vdash x : T}{x : T \vdash x : \{\mathtt{Sub} : \mathrm{Pre}(\eta); \rho\}} \qquad \begin{array}{c} \dfrac{x : T \vdash .\mathtt{Sub} : \forall \eta, \rho.\{\mathtt{Sub} : \mathrm{Pre}(\eta); \rho\} \to \eta}{x : T' \vdash .\mathtt{Sub} : \{\mathtt{Sub} : \mathrm{Pre}(\eta); \rho\} \to \eta} \\ \vdots \end{array}}{x : T \vdash x.\mathtt{Sub} : \eta}}{x : T \vdash x.\mathtt{Sub} : T'} \qquad T' \in S(e)}{S, x : T \vdash \overline{e}\langle x.\mathtt{Sub}\rangle} \right)$$

$$J \triangleq \left(\frac{\dfrac{\dfrac{J_1 \qquad J_1'}{S, \emptyset \vdash e(x).\mathrm{IfPre}(x, \mathtt{Sub}, i, s)} \qquad \dfrac{J_2}{S, \emptyset \vdash i(x).\overline{e}\langle x.\mathtt{Sub}\rangle}}{S, \emptyset \vdash e(x).\mathrm{IfPre}(x, \mathtt{Sub}, i, s) \mid i(x).\overline{e}\langle x.\mathtt{Sub}\rangle}}{S, \emptyset \vdash \mathbf{TheLoop}} \right)$$

FIGURE A.2

rait d'assouplir encore les substitutions pour permettre l'élimination des types concrets, Mais avec ce genre de substitution, nous pourrions éliminer les types des messages envoyés sur les canaux, et ainsi, le système de type ne serait plus correct : tout assemblage serait typable. Une autre possibilité serait d'introduire explicitement dans nos types une dépendence entre les entrés d'un composant et ses sorties, et permettre ainsi d'éliminer les types de sortie du composant lorsque ses types d'entrés sont vides. Cette solution est en fait étudiée dans notre seconde approche, Chapitre 3, où les dépendences sont représentées par les variables de flux. Notons que dans cette approche, nous ne permettons pas l'élimination des types comme suggéré ici, afin de pouvoir détécter directement les composants au comportement incohérent.

A.2 L'indécidabilité de l'inférence de type

Nous prouvons dans cette partie l'équivalence entre l'inférence de type pour notre système de type, et le problème de correspondance de Post (ou simplement PCP). Nous rappelons dans un premier temps la définition de ce problème, avant de montrer comment nous l'encodons dans notre inférence de type.

Définition 14. *Supposons donné un alphabet $\mathcal{A}$ contenant au moins deux lettres, et une liste finie $(v_i, u_i)_i$ de paires de mots construits sur l'alphabet $\mathcal{A}$. Le* problème de correspondance de Post *consiste en la recherche d'une suite finie et non vide d'indices $(i_j)_{1 \leq j \leq n}$ telle que les mots $v_{i_1} \dots v_{i_n}$ et $u_{i_1} \dots u_{i_n}$ soient égaux. Ce problème est indécidable, c'est à dire qu'il n'existe pas d'algorithme qui, à partir d'une liste finie de paires de mots, peut calculer si une solution $(i_j)_{1 \leq j \leq n}$ existe.*

Nous supposons par la suite que l'alphabet $\mathcal{A}$ est l'ensemble $\{a, b\}$. Le principe de notre encodage est relativement simple : chaque paramètre l possible de PCP est traduit en un programme D, typable si et seulement si l admet une solution pour PCP. Il est alors clair qu'un algorithme d'inférence pour notre système de type pourrait résoudre PCP : un tel algorithme n'existe donc pas. Considérons maintenant un paramètre $l \triangleq (v_i, u_i)_{1 \leq i \leq n}$ de PCP. Le progamme D_l que nous construisons à partir de l consiste en fait en une simple recherche en largeur d'une solution pour PCP : ce programme termine si et seulement si le problème a une solution. De plus, les mots sont encodés dans notre programme par des messages structurés : si PCP a une solution pour l, le programme D_l ne calcule qu'un nombre fini de messages tous typable, et donc, le programme est typable. Par contre, si PCP n'a pas de solution pour l, D_l aura une exécution infinie avec le calcul d'un nombre infini de messages de structure différente : D_l n'est alors pas typable (les types infinis ne sont pas autorisés dans notre système de type). Le reste de cette partie présente la construction de D_l à partir de l.

L'encodage des structures de base. Nous faisons la supposition raisonnable que tout les éléments de l'ensemble

$$\mathcal{A} \uplus \{\text{tail}, \text{head}, \text{first}, \text{sec}, \text{third}, \text{tmp}\}$$

sont des labels valides pour nos messages structurés. Nous pouvons alors définir les notations suivantes, définissant comment les structures de PCP sont encodés dans nos programmes :

- Si u est un mot, nous notons $\|u\|_D$ son encodage qui est défini inductivement comme il suit (ε étant le mot vide et d une lettre quelconque de $\mathcal{A}$) :

  $$\|\varepsilon\|_D \triangleq \{\} \qquad\qquad \|d.u\|_D \triangleq \{d = \|u\|_D\}$$

  De plus, si M est un message et u un mot, nous notons $u \bullet M$ le message construit inductivement comme il suit (ce qui nous donne une opération de concaténation sur les mots) :

  $$\varepsilon \bullet M \triangleq M \qquad\qquad d.v \bullet M \triangleq \{d = (v \bullet M)\}$$

- Si M_1 et M_2 sont deux messages, nous notons $M_1 :: M_2$ le message $\{\text{head} = M_1; \text{tail} = M_2\}$. De plus, nous utiliserons aussi dans le reste de reste de ce chapitre, à quelques endroits particuliers, la notation [] pour le message vide. Ces notations montrent commentnous encodons les listes dans nos messages structurés.
- Si M_1 et M_2 sont deux messages, nous notons (M_1, M_2) le message $\{\text{first} = M_1; \text{sec} = M_2\}$. Si M_3 est aussi un message, nous notons (M_1, M_2, M_3) le message $\{\text{first} = M_1; \text{sec} = M_2; \text{third} = M_3\}$.
- Finalement, si M est un message, nous notons

$$\mathit{fst}(M) \triangleq M.\text{first} \qquad \mathit{snd}(M) \triangleq M.\text{sec} \qquad \mathit{trd}(M) \triangleq M.\text{third}$$
$$\mathit{hd}(M) \triangleq M.\text{head} \qquad \mathit{tl}(M) \triangleq M.\text{tail}$$

La structure de notre programme. Comme nous l'avons déjà signalé, le principe de l'encodage de l est de calculer l'ensemble L_l de mots que PCP peut former à partir de l, afin de voir si ce problème admet une solution. Plus précisement, notre algorithme consiste en une boucle calculant itérativement un nouveau sous-ensemble de L et testant si cet ensemble contient une solution. De plus, cette boucle comporte un unique message M encodant dans une liste toutes les paires de mots générés à chaque tour de boucle. Comme M contient beaucoup d'informations, ce message est assez difficile à manipuler, et il aurait été beaucoup plus facile de construire notre programme si nous avions choisi une approche comportant plusieurs messages. Néanmoins, notre approche a l'avantage de définir un flux de donné totalement séquenciel, ce qui simplifie grandement la preuve de la correspondance entre l'exécution du programme et l'existence d'un type pour celui-ci. Finalement, afin d'illustrer comment le message M est généré à chaque tour de boucle, nous définissons une famille d'ensemble de paires de mots :

Définition 15. *Soit $m \in \mathbb{N}$. L'ensemble $L_l(m)$ est défini comme suit :*

$$L_l(m) \triangleq \{(u_{i_1} \dots u_{i_m}, v_{i_1} \dots v_{i_m}) \mid \forall 1 \leq j \leq m,\ 1 \leq i_j \leq n\}$$

Il est évident que l'union infinie $\bigcup_i^\infty L_l(m)$ donne bien l'ensemble L_l de tout les mots pouvant être une solution de PCP pour l. De plus, les ensembles $L_l(m)$ peuvent être facilement construits récursivement par la formule suivante :

$$L_l(0) \triangleq \{(\varepsilon, \varepsilon)\} \qquad L_l(m+1) \triangleq \{(u.u_1, v.v_1) \mid (u, v) \in l \wedge (u_1, v_1) \in L_l(m)\}$$

La construction inductive des ensembles $L_l(m)$ correspond directement au calcul générant à chaque tour de boucle le message M : à l'étape m du calcul, le message M correspond à l'ensemble $L_l(m)$, et est calculé en fonction de la version précédente de M qui encodait l'ensemble $L_l(m-1)$.

La phase de test est plus simple que la phase de construction de M, et consiste simplement en le parcours de la liste de paires. Chaque paire de mots (u, v) est traîtée, et nous vérifions que u est égal à v. Si c'est le cas, nous avons trouvé une solution pour PCP, et nous envoyons un message de réussite sur le

canal de sortie de notre programme : le calcul s'arrête donc. Si aucune paire ne comporte deux fois le même mots, nous renvoyons M à la partie de génération pour un nouveau tour de boucle. Comme $\bigcup_1^\infty L_l(m) = L_l$, nous sommes assurés de trouver en un nombre fini d'étapes une solution pour PCP lorsqu'elle existe.

La construction de M. Le message M est construit selon le principe de récurrence présenté Définition 15 : à partir d'une liste L donnée en entrée, nous devons construire une liste correspondant à l'ensemble $\{(u.u_1, v.v_1) \mid (u, v) \in l \wedge (u_1, v_1) \in L\}$ Nous décomposons cette construction en plusieurs sous-étapes, chacune calculant l'ensemble $\{(u.u_1, v.v_1) \mid (u_1, v_1) \in L\}$ pour une paire (u, v) donnée : la sous étape correspondant à cette paire de mots est définie par le programme nommé $D_a(u, v)$. Ce programme écoute sur le canal $\text{input}_{u,v}$ et attend un triplet contenant (i) la liste correspondant à L ; (iii) une donnée globale utilisée par d'autres parties du programme (en fait, un duplicat de la liste L) ; et (iii) le resultat partiel calculé par les autres sous-étapes. La première chose que le programme $D_a(u, v)$ fait est de tester si la liste L est vide. Comme la procédure de routage ne permet pas d'inspection en profondeur des messages structurés, nous utilisons une petite astuce pour pouvoir accéder à L sans perdre les information contenu dans le message : nous construisons le message temporaire $\mathit{fst}(x) + (\text{tmp} = x)$ qui correspond à la liste que l'on peut donc tester, étendue par un nouveau champ tmp contenant tout le message d'entré. Ce test passé, nous récupérons le triplet initial, et le traîtons différemment en fonction de l'état de la liste : si elle est vide, il n'y a pas de paire à rajouter au résultat et le message est envoyé alors directement sur le canal de sortie $\text{output}_{u,v}$. Si elle n'est pas vide, nous envoyons le triplet sur le canal $\text{tmp}_{u,v}$ pour rajouter une nouvelle paire au résultat déjà calculé.

$$
\begin{aligned}
&D_a(u, v) \triangleq \\
&\quad !\text{input}_{u,v}(x).\text{IfPre}(\text{head}, \mathit{fst}(x) + (\text{tmp} = x), \text{loop}_{u,v}, \text{finish}_{u,v}) \\
&\quad \mid !\text{loop}_{u,v}(x).\overline{\text{tmp}_{u,v}}\langle x.\text{tmp}\rangle \\
&\quad \mid !\text{finish}_{u,v}(x).\overline{\text{output}_{u,v}}\langle x.\text{tmp}\rangle \\
&\quad \mid !\text{tmp}_{u,v}(x).\overline{\text{input}_{u,v}}\langle \begin{pmatrix} \mathit{tl}(\mathit{fst}(x)), \mathit{snd}(x), \\ (u \bullet \mathit{fst}(\mathit{hd}(\mathit{fst}(x))), v \bullet \mathit{snd}(\mathit{hd}(\mathit{fst}(x)))) :: (\mathit{trd}(x)) \end{pmatrix}\rangle
\end{aligned}
$$

Nous assemblons les programmes $D_a(u, v)$ en une chaîne relativement simple pour obtenir le programme D_n construisant M à chaque étape m en fonction de la version précédente du message. Ce programme prend son paramètre sur le canal input, et envoie le résultat du calcul sur le canal ouput :

$$
\begin{aligned}
&D_n(l) \triangleq \\
&\quad !\text{input}(x).\overline{\text{input}_{u_1,v_1}}\langle(x, x, [])\rangle \\
&\quad \mid D_a(u_1, v_1) \\
&\quad \mid !\text{output}_{u_1,v_1}(x).\overline{\text{input}_{u_2,v_2}}\langle(\mathit{snd}(x), \mathit{snd}(x), \mathit{trd}(x))\rangle \\
&\quad \mid \ldots \\
&\quad \mid D_a(u_n, v_n) \\
&\quad \mid !\text{output}_{u_n,v_n}(x).\overline{\text{ouput}}\langle \mathit{trd}(x)\rangle
\end{aligned}
$$

Nous pouvons remarquer que le second élément des triplets échangés entre les programmes $D_a(u, v)$ correspond bien à une copie de la liste L. Nous avons besoin de cette copie, car les programmes $D_a(u, v)$ consomment leur instance de la liste lors du calcul de la liste résultat.

Le test du message M. Le message M calculé par le programme D_n est une liste de la forme $(w_1, w'_1) :: (w_2, w'_2) :: \ldots :: []$ et contenant l'ensemble des paires de mots de l'ensemble $L_l(m)$ pour un m donné. La partie test de notre programnme consiste à vérifier si l'ensemble calculé ne contient pas une solution pour PCP. Pour cela, nous parcourons simplement la liste M, et pour chaque paire de mots (w, w'), nous testons si w est égal à w'. Ce test d'égalité est assuré par le programme D_{eq}, Ce programme prend sur son canal d'entré input_t une paire de mots à tester, étendue par une troisième composantes contenant les données annexes utilisées par le reste de notre encodage. À l'aide de plusieurs procédure de routage et de notre astuce utilisée dans les programmes $D_a(u, v)$, nous testons si les deux mots sont vides ou commence par la même lettre. Dans le premier cas, la troisième composante du message est envoyée sur le canal de sortie output_{ok}, signalant ainsi que les deux mots sont egaux. Dans le second cas, le message d'entré est envoyé sur le canal t_a (si la première lettre deux deux mots est un a) ou t_b (resp. un b). Cette première lettre est alors otée des mots et la paire résultante plus la donnée annexe sont renvoyé sur input_t pour continuer le test.

$$
\begin{aligned}
D_{eq} \triangleq & \\
& !\text{input}_t(x).\text{IfPre}(a, \mathit{fst}(x) + (\text{tmp} = x), \text{t}_{ua}, \text{t}_{na}) \\
& \mid !\text{t}_{ua}(x).\text{IfPre}(a, \mathit{snd}(x.\text{tmp}) + (\text{tmp} = x.\text{tmp}), \text{t}_a, \text{t}_{\text{err}}) \\
& \mid !\text{t}_{na}(x).\text{IfPre}(a, \mathit{snd}(x.\text{tmp}) + (\text{tmp} = x.\text{tmp}), \text{t}_{\text{err}}, \text{t}_{tb}) \\
& \mid !\text{t}_{tb}(x).\text{IfPre}(b, \mathit{fst}(x.\text{tmp}) + (\text{tmp} = x.\text{tmp}), \text{t}_{ub}, \text{t}_{u:\varepsilon}) \\
& \mid !\text{t}_{ub}(x).\text{IfPre}(b, \mathit{snd}(x.\text{tmp}) + (\text{tmp} = x.\text{tmp}), \text{t}_b, \text{t}_{\text{err}}) \\
& \mid !\text{t}_{u:\varepsilon}(x).\text{IfPre}(b, \mathit{snd}(x.\text{tmp}) + (\text{tmp} = x.\text{tmp}), \text{t}_{\text{err}}, \text{t}_\varepsilon) \\
& \mid !\text{t}_a(x).\overline{\text{input}_t}\langle((\mathit{fst}(x.\text{tmp})).a, (\mathit{snd}(x.\text{tmp})).a, \mathit{trd}(x.\text{tmp}))\rangle \\
& \mid !\text{t}_b(x).\overline{\text{input}_t}\langle((\mathit{fst}(x.\text{tmp})).b, (\mathit{snd}(x.\text{tmp})).b, \mathit{trd}(x.\text{tmp}))\rangle \\
& \mid !\text{t}_\varepsilon(x).\overline{\text{output}_{ok}}\langle \mathit{trd}(x.\text{tmp})\rangle \\
& \mid !\text{t}_{\text{err}}(x).\overline{\text{output}_{fail}}\langle \mathit{trd}(x.\text{tmp})\rangle
\end{aligned}
$$

Finalement, dans le cas où les premières lettres des deux mots ne sont pas les mêmes, la donnée annexe est envoyée sur le canal de sortie output_{fail}, signalant ainsi que les deux mots en paramètre ne sont pas égaux. Nous pouvons noter que nous faisons un large usage du champ tmp dans ce programme, mais qu'il contient en fait toujours la même valeur : le message initial donné en entré au programme de test.

La partie gérant le parcours de la liste est nommé D_{eqp}. Ce programme prend en entré (sur le canal eq) une liste de paire de mots telle que la génère le programme D_n. La première chose que fait ce programme est de dupliquer cette liste dans une paire, au cas où elle ne contienne pas de solution, et qu'il faille continuer le calcul et la renvoyer au programme D_n. Tout le traitement se

fait donc sur la première composante de cette paire (qui, rapelons le, est la liste de paire de mots à tester).

$$
\begin{aligned}
D_{eqp} \triangleq & \\
& !\mathrm{eq}(x).\overline{\mathrm{eq}_{\mathrm{test}}}\langle(x,x)\rangle \\
& \mid !\mathrm{eq}_{\mathrm{test}}(x).\mathrm{IfPre}(\mathrm{head}, \mathit{fst}(x) + (\mathrm{tmp} = x), \mathrm{send}_{eqp}, \mathrm{list}_r) \\
& \mid !\mathrm{send}_{eqp}(x).\overline{\mathrm{input}_t}\langle(\mathit{hd}(x) + (\mathrm{third} = x.\mathrm{tmp}))\rangle \\
& \mid D_{eq} \\
& \mid !\mathrm{output}_{fail}(x).\overline{\mathrm{eq}_{\mathrm{test}}}\langle(\mathit{tl}(\mathit{fst}(x)), \mathit{snd}(x))\rangle \\
& \mid !\mathrm{output}_{ok}(x).\overline{\mathrm{eq}_{\mathrm{s}}}\langle[]\rangle \\
& \mid !\mathrm{list}_r(x).\overline{\mathrm{eq}_{\mathrm{fail}}}\langle \mathit{snd}(x.\mathrm{tmp})\rangle
\end{aligned}
$$

Le principe de ce programme est similaire à celui du programme précédent, et nous pensons donc inutile de détailler comment la boucle de test fonctionne. Notons que si le programme trouve une paire de mots identique dans la liste, il envoie la liste vide sur le canal eq_{s}, signalant ainsi que le cette instance de PCP admet une solution. Dans le cas contraire, le programme renvoie la liste en paramètre sur son canal d'erreur $\mathrm{eq}_{\mathrm{fail}}$.

L'assemblage des différentes parties. Maintenant que les deux parties de notre encodage sont définies, nous pouvons les assembler, et trouver le message adapté pour démarrer l'exécution du programme résultant. Par construction, il est naturel de connecter la sortie de D_n à l'entrée de D_{eqp}, et la sortie d'erreur de ce dernier à l'entrée de D_n, formant ainsi la boucle de calcul des ensembles $L_l(m)$. Finalement, pour démarrer ce calcul, nous plaçons sur le canal input l'ensemble $L_l(0)$. Ainsi, l'exécution du programme commencerant par la génération et le test de l'ensemble $L_l(1)$, et continuera jusqu'à ce qu'une solution pour PCP soit trouvée.

$$
\begin{aligned}
D_{pcp}(l) \triangleq & \\
& \overline{\mathrm{input}}\langle(\|\varepsilon\|_D, \|\varepsilon\|_D) :: []\rangle \\
& \mid D_n(l) \\
& \mid !\mathrm{ouput}(x).(\overline{\mathrm{eq}}\langle x\rangle) \\
& \mid D_{eqp} \\
& \mid !\mathrm{eq}_{\mathrm{fail}}(x).(\overline{\mathrm{input}}\langle x\rangle)
\end{aligned}
$$

Les propriétés de notre encodage

Théorème 11. *Supposons donné un paramètre valide l pour PCP et admettant une solution pour ce problème. Alors, l'encodage $D_{pcp}(l)$ que nous proposons de l est un programme typable.*

Théorème 12. *Supposons donné un paramètre valide l pour PCP et n'admettant pas de solution pour ce problème. Alors, l'encodage $D_{pcp}(l)$ que nous proposons de l ne possède pas de type dans notre système.*

Les preuves de ces théorèmes sont extrèmement longues et techniques, mais pas vraiment difficiles, car le typage (ou non) du programme $D_{pcp}(l)$ consiste

plus ou moins en le comptage du nombre de paires de mots que ce programme génère durant son exécution.

En conclusion, comme PCP est indécidable, il n'existe pas d'algorithme pouvant décider si un programme $D_{pcp}(l)$ est typable. Par conséquence, l'inférence de type est indécidable dans notre cas :

Corollaire 1. *L'inférence est indécidable pour notre système de type.*

Annexe B

Annexe du chapitre 3

Dans cette annexe, nous proposons les preuves des théorèmes de correction et de stabilité du système de type décrit dans la partie 3.2. Ces théorèmes sont introduits par plusieurs lemmes préliminaires.

B.1 Les propriétés du sous-typage

La relation de sous-typage entre les types peut être vue comme l'intersection d'une famille dénombrable de relation $(\leq_i)_{0\leq i}$, où $\leq_0$ met tous les types en relation, et $\leq_k$ est construit par induction, comme cela est montré Figure B.1.

$$\frac{s \leq s' \qquad \forall 1 \leq i \leq \min(m,n),\ E_i \leq_k E'_i}{s(E_1,\ldots,E_n) \leq_{k+1} s'(E'_1,\ldots,E'_m)}$$

$$\frac{\forall 1 \leq i \leq n,\ E_i \leq_k E'_i}{\{a_1 : \mathrm{Pre}(E_1);\ldots;a_n : \mathrm{Pre}(E_n);W\} \leq_{k+1} \{a_1 : \mathrm{Pre}(E'_1);\ldots;a_n : \mathrm{Pre}(E'_n);W\}}$$

$$\frac{E \leq_k E'}{\xi[E] \leq_{k+1} \xi[E']} \qquad \frac{T_1 \leq_k T'_1 \qquad T_2 \leq_k T'_2}{r(T_1,T_2) \leq_{k+1} r(T'_1,T'_2)} \qquad \frac{T \leq_k T'}{e : (T) \leq_{k+1} e : (T')}$$

$$\frac{S_1 \leq_k S'_1 \qquad S_2 \leq_k S'_2}{S_1 \cup S_2 \leq_{k+1} S'_1 \cup S'_2} \qquad \frac{S_1 \leq_k S'_1 \qquad S_2 \leq_k S'_2}{S'_1 \to S_2 \leq_{k+1} S_1 \to S'_2} \qquad \frac{F \leq_k F'}{\forall\alpha.F \leq_{k+1} \forall\alpha.F'}$$

FIGURE B.1 – La relation de sous-typage

Rappelons que l'ensemble des constructeurs de type $\mathcal{C}$ forme un treillis avec

les opérateurs $\sqcap$ et $\sqcup$, et que nous avons :

$$\forall S \subset \mathcal{C} \forall s \in S,\ a(\sqcap(S)) \leq a(s) \wedge a(\sqcup(S)) \geq a(s)$$

Lemme 2. *La relation de sous-typage construit une structure de treillis sur l'ensemble des types de messages.*

Démonstration. Il est facile de voir que les relations $\leq_k$ sont des pré-ordres pour tous les $k \in \mathbb{N}$: par construction, toutes ces relations sont réflexives. Supposons maintenant que $\leq_k$ est transitif et prouvons qu'il est de même pour $\leq_{k+1}$. Considérons trois types τ_1, τ_2 et τ_3 tels que $\tau_1 \leq_{k+1} \tau_2$ et $\tau_2 \leq_{k+1} \tau_3$. Par construction de $\leq_{k+1}$, cela implique que $\tau_1' \leq_k \tau_2'$ et $\tau_2' \leq_k \tau_3'$ pour les sous-termes τ_i' de τ_i. Ainsi, par transitivité, nous avons $\tau_1' \leq_k \tau_3'$, et donc, par construction, nous avons aussi $\tau_1 \leq_{k+1} \tau_3$. De plus, comme la relation sur les constructeurs de type est un ordre, le noyau de la relation de sous-typage $\leq$ est l'identité, et donc cette relation est un ordre.

Finalement, nous pouvons de plus définir l'opérateur $\sqcap$ (resp. $\sqcup$) comme suit (resp.de manière similaire à ce qui suit) :

$$\sqcap(s_i(E_i^j)) \triangleq \sqcap(s_i)((\sqcap(E_i))^j) \qquad \sqcap(\{W_i\}) \triangleq \{\sqcap(W_i)\}$$

$$\sqcap(a : \mathrm{Pre}(E_i); W_i) \triangleq a : \mathrm{Pre}(\sqcap(E_i); \sqcap(W_i) \qquad \sqcap(a : \mathrm{Abs}; W_i) \triangleq a : \mathrm{Abs}; \sqcap(W_i)$$

$$\sqcap(\mathrm{Abs}) \triangleq \mathrm{Abs}$$

$\sqcap(E_i) \triangleq \sqcap$ pour tout ensemble de types de message non pris en compte par les définitions précédentes

Par induction sur k, il est facile de voir que $\sqcap(E_i)$ (resp. $\sqcup(E_i)$) est la borne minimale (resp. la borne maximale) de E_i. $\square$

B.1.1 La forme normale d'une dérivation de type

Dans cette partie, nous prouvons que toute dérivation de type peut se réécrire sous une *forme normale*. Nous définition tout d'abord ce qu'est une forme normale pour les dérivations, puis par une succession de lemmes, transformons petit à petit les dérivations de types pour aboutir à des dérivations en forme normale.

Définition 16. *Supposons donné une dérivation de type J. J est en* forme normale *si aucune des règles* T :Inst, T :Gen, T :SubD *ou* T :Inst *suit les règles* T :Parallel *ou* T :Box *durant cette dérivation de type.*

Lemme 3. *Supposons donnés un programme D et un type $F \triangleq \forall \overline{\alpha}.S_1 \to S_2$ tels qu'il existe une dérivation de $D : F$. Alors il existe une dérivation de $D : S_1 \to S_2$.*

Démonstration. Le résultat est évident avec la règle de typage T :Inst. □

Lemme 4. *Supposons donné un programme D et un type F tel que le typage $D : F$ est valide. Alors, il existe une dérivation de type J' se concluant par $D : F$ telle que si sa dernière règle est* T :Inst*, alors elle ne contient pas d'application des règles* T :Parallèle *or* T :Boîte*.*

Démonstration. Par induction sur la dérivation J du typage $D : F$:

- Cas J ne termine pas par la règle T :Inst. Avec $J' \triangleq J$, nous avons le résultat.
- Cas J termine avec la règle T :Inst mais ne comporte aucune utilisation des règles T :Parallèle et T :Boîte. Avec $J' \triangleq J$, nous avons encore une fois le résultat.
- Considérons maintenant le cas où J a la forme suivante, avec J_k pouvant contenir des applications des règles T :Parallèle et T :Boîte :

$$\text{T :Inst}\quad \dfrac{\dfrac{J_k}{D : \forall\alpha.F}}{D : F}$$

En utilisant la règle d'induction sur J_k, il existe une dérivation J'_k de $D : \forall\alpha.F$ qui ne termine pas par une instance de la règle T :Inst. De plus, par construction, la règle de typage T :Inst ne peut être appliquée à la suite de T :Boîte ni à la suite de T :Parallèle. Nous avons donc trois cas :

1. Cas J'_k est de la forme :

$$\dfrac{\dfrac{J''_k}{D : \forall\alpha.F'}}{D : \Sigma(\forall\alpha.F')}$$

Par construction, nous avons $\alpha \notin \Im(\Sigma)$, comme cette variable est liée dans $\forall\alpha.F'$. De plus, nous avons $\Sigma_{|\complement\{\alpha\}}(\forall\alpha.F') = \Sigma(\forall\alpha.F')$ pour la même raison. Par conséquent, nous pouvons remarquer que $\Sigma_{|\complement\{\alpha\}}(F') = F$. Ainsi, nous pouvons construire J' comme suit, ce qui nous donne le résultat dans ce cas :

$$J' \triangleq \dfrac{\dfrac{\dfrac{J''_k}{D : \forall\alpha.F'}}{D : F'}}{D : \Sigma_{|\complement\{\alpha\}}(F')}$$

2. Cas J'_k est de la forme :

$$\dfrac{\dfrac{J''_k}{D : F'}}{D : \forall\beta.F'}$$

Tout d'abord, si $\alpha = \beta$, nous avons $F' = F$. Comme J''_k contient une instance d'une des règles T :Parallèle ou T :Boîte, en utilisant l'hypothèse de récurrence, il existe une dérivation J' de $D : F$ qui ne termine pas par T :Inst. Dans le cas où $\alpha \neq \beta$, nous pouvons définir J' comme suit, ce qui nous donne le résultat dans ce cas (avec $F' = \forall\alpha.F''$) :

$$J' \triangleq \dfrac{\dfrac{\dfrac{J''_k}{D : F'}}{D : F''}}{D : \forall\beta.F''}$$

3. Cas J'_k est de la forme :

$$\dfrac{\dfrac{J''_k}{D : F_1} \qquad F_1 \leq F_2}{D : F_2}$$

Par construction, nous avons $F_2 = \forall\alpha.F$. De plus, d'après la définition de la relation de sous-typage $\leq$ entre les types de processus, il existe F_3 avec $F_1 = \forall\alpha.F_3$ et $F_3 \leq F$. Ainsi, nous pouvons définir J' comme suit, ce qui nous donne le résultat dans ce dernier cas :

$$\dfrac{\dfrac{\dfrac{J''_k}{D : F_1}}{D : F_3} \qquad F_3 \leq F}{D : F}$$

□

Lemme 5. *Supposons donnés un programme D, un type de processus $S_1 \to S_2$ et une dérivation de type valide J se concluant par $D : S_1 \to S_2$. Alors il existe une dérivation* en forme normale *de $D : S_1 \to S_2$.*

Démonstration. Nous construisons J' par récurrence sur la taille de J (remarquons au préalable que J ne peut terminer par la règle T :Gen par construction) :

– Cas J est d'une des deux forme suivante ;

$$\dfrac{\dfrac{J_k}{M : T}}{\overline{e}\langle M\rangle : e : (T)} \quad \text{ou} \quad \dfrac{\Omega(p) = F}{p : F}$$

  Ce cas est évident.
– Cas J termine par soit la règle T :Parallèle ou la règle T :Boîte. En utilisant l'hypothèse de récurrence, nous avons le résultat.

– Cas J est de la forme :

$$\dfrac{\dfrac{J_k}{D \,:\, F}}{D' \,:\, \Sigma(F)}$$

Par définition de la règle de substitution, F est de la forme $S'_1 \to S'_2$. Nous pouvons donc utiliser l'hypothèse de récurrence pour obtenir une dérivation en forme normale J'_k de $D \,:\, F$. Si J'_k termine par une des règles T :Inst, ou T :Sub ou encore T :Subst, la construction suivante nous donne le résultat dans ce cas :

$$\dfrac{\dfrac{J'_k}{D \,:\, F}}{D' \,:\, \Sigma(F)}$$

Nous avons deux autres sous-cas :

1. Cas J'_k est de la forme :

$$\dfrac{\dfrac{J_1}{D_1 \,:\, S_1^1 \to S_2^1} \qquad \dfrac{J_2}{D_2 \,:\, S_1^2 \to S_2^2}}{D_1 \mid D_2 \,:\, (S_1^1 \cup S_1^2) \to (S_2^1 \cup S_2^2)}$$

Considérons les deux dérivations suivantes :

$$J'_1 \triangleq \dfrac{\dfrac{J_1}{D_1 \,:\, S_1^1 \to S_2^1}}{D_1 \,:\, \Sigma(S_1^1 \to S_2^1)} \quad \text{et} \quad J'_2 \triangleq \dfrac{\dfrac{J_2}{D_2 \,:\, S_1^2 \to S_2^2}}{D_2 \,:\, \Sigma(S_1^2 \to S_2^2)}$$

Comme ces dérivations sont plus courte que J, nous pouvons appliquer l'hypothèse de récurrence : il existe une dérivation J''_1 (resp J''_2) en forme normale de $D_1 \,:\, \Sigma(S_1^1 \to S_2^1)$ (resp. $D_2 \,:\, \Sigma(S_1^2 \to S_2^2)$). Nous pouvons de plus remarquer que nous avons toujours $\Sigma(S_2^1) \precsim \Sigma(S_1^2)$ et $\Sigma(S_2^2) \precsim \Sigma(S_1^1)$. Par conséquent, la construction suivante nous donne le résultat dans ce cas :

$$\dfrac{\dfrac{J''_1}{D_1 \,:\, \Sigma(S_1^1 \to S_2^1)} \qquad \dfrac{J''_2}{D_1 \,:\, \Sigma(S_1^2 \to S_2^2)}}{D_1 \mid D_2 \,:\, S_1 \to S_2}$$

2. Cas J'_k est de la forme :

$$\dfrac{S'_2 \precsim S_2 \qquad \dfrac{J_1}{D' \,:\, S'_1 \to S'_2} \qquad S_1 \precsim S'_1 \qquad dc(S'_1) = I \wedge dc(S'_2) = O \qquad S'_1 \to S'_2 \text{ is consistent}}{c[I \,/\, O][D'] \,:\, S_1 \to S_2}$$

Nous pouvons facilement voir que la dérivation de type suivante est valide, et nous donne ainsi le résultat :

$$\dfrac{\dfrac{\dfrac{J_1}{D' : S'_1 \to S'_2}}{D' : \Sigma(S'_1 \to S'_2)} \qquad \Sigma(S_1) \precsim \Sigma(S'_1) \qquad \Sigma(S'_2) \precsim \Sigma(S_2) \qquad dc(\Sigma(S'_1)) = I \wedge dc(\Sigma(S'_2)) = O \qquad \Sigma(S'_1 \to S'_2) \text{ is consistent}}{D : \Sigma(S'_1 \to S'_2)}$$

– Cas J est de la forme :

$$\dfrac{\dfrac{J_k}{D : F'} \qquad F' \leq F}{D : F}$$

En utilisant la même approche que le cas précédent, nous avons le résultat.

– Cas J est de la forme :

$$\dfrac{\dfrac{J_k}{D : \forall\alpha.F}}{D : F}$$

Si J est en forme normale, nous avons le résultat avec $J' \triangleq J$. Sinon, il existe une application d'une des règles T :Parallèle ou T :Boîte dans J. Nous pouvons donc appliquer le lemme 4 pour avoir une dérivation J'_k de $D : F$ qui ne termine pas par une instance de la règle T :Inst. Nous revenons ainsi à un des cas précédents, qui nous donne le résultat.

□

B.2 L'extraction de type

Lemme 6. *Supposons donnés un programme $D \triangleq \overline{e}\langle M\rangle \mid D_k$ et un type de processus $F \triangleq \forall\overline{\alpha}.S_1 \to S_2$ tels que le typage $D : F$ est valide. Alors, il existe un type routé $T \in S_2(e)$ tel que le typage $\overline{e}\langle M\rangle : \emptyset \to e : (T)$ est valide lui aussi.*

Démonstration. Nous raisonnons par récurrence sur la dérivation de type J de $D : F$:

– Cas J est de la forme :

$$\dfrac{\dfrac{J'}{M : T}}{\overline{e}\langle M\rangle : \emptyset \to e : (T)}$$

Le résultat est direct dans ce cas.

– Cas J est de la forme :

$$\dfrac{\dfrac{J_1}{D_1 \,:\, S_1 \to S_2} \qquad \dfrac{J_2}{D_2 \,:\, S_1' \to S_2'}}{D_1 \mid D_2 \,:\, (S_1 \cup S_1') \to (S_2 \cup S_2')}$$

Avec l'hypothèse de récurrence, nous pouvons supposer qu'il existe $T \in S_2(e)$ tel que le typage $\overline{e}\langle M\rangle \,:\, \emptyset \to e : (T)$ est valide. Nous avons par conséquent que $T \in (S_2 \cup S_2')(e)$.

– Cas J est de la forme :

$$\dfrac{\dfrac{J'}{D' \,:\, F'}}{D' \,:\, \sigma(F')}$$

Avec l'hypothèse de récurrence, si nous notons $F' \triangleq \forall\overline{\beta}.S_1' \to S_2'$, il existe $T \in S_2'(e)$ tel que le typage $\overline{e}\langle M\rangle \,:\, \emptyset \to e : (T)$ est valide. Notons de plus $S_2' \triangleq S_3 \cup e : (T)$. Nous pouvons alors voir que par construction $fv(T) = \emptyset$, ce qui implique, avec l'α-conversion, que $\sigma(F') = \forall\overline{\beta}.\sigma(S_1') \to \sigma(S_3) \cup e : (T)$. Le résultat est alors direct dans ce cas.

– Cas J est de la forme :

$$\dfrac{\dfrac{J'}{D' \,:\, F'}}{D' \,:\, \forall\alpha.F'}$$

Si nous notons $F' \triangleq \forall\overline{\beta}.S_1' \to S_2'$, en utilisant l'hypothèse de récurrence, il existe $T \in S_2'(e)$ tel que le typage $\overline{e}\langle M\rangle \,:\, \emptyset \to e : (T)$ est valide. Le résultat est alors direct dans ce cas.

– Cas J est de la forme :

$$\dfrac{\dfrac{J'}{D \,:\, \forall\alpha.F'}}{D \,:\, F'}$$

Si nous notons $\forall\alpha.F' \triangleq \forall\overline{\beta}.S_1' \to S_2'$, en utilisant l'hypothèse de récurrence, il existe $T \in S_2'(e)$ tel que le typage $\overline{e}\langle M\rangle \,:\, \emptyset \to e : (T)$ est valide. Le résultat est alors direct dans ce cas.

– Cas J est de la forme :

$$\dfrac{\dfrac{J'}{D' \,:\, F'} \qquad F' \leq F''}{D' \,:\, F''}$$

Avec l'hypothèse de récurrence, si nous notons $F' \triangleq \forall\overline{\beta}.S_1' \to S_2'$, il existe $T \in S_2'(e)$ tel que le typage $M \,:\, T$ est valide. Nous pouvons donc utiliser la définition de la relation de sous typage entre deux types de processus pour avoir $F'' = \forall\overline{\beta}.S_1'' \to S_2''$ avec $S_1'' \leq S_1'$ et $S_2' \leq S_2''$. Ainsi, il existe $T' \in S_2''(e)$ tel que $T \leq T'$. Par conséquent, comme $\overline{e}\langle M\rangle \,:\, \emptyset \to e : (T)$ est valide, nous pouvons appliquer T :Sub pour obtenir une dérivation de type de $\overline{e}\langle M\rangle \,:\, \emptyset \to e : (T')$.

□

Lemme 7. *Supposons donnés un programme $D \triangleq p \mid D_k$ et un type de processus F tels que le typage $D : F$ est valide. Alors il existe $\overline{\alpha}$, S_1, S'_1, S_2 et S'_2 tels que le typage $p : \forall\overline{\alpha}.S_1 \to S_2$ est valide, et $F = \forall\overline{\alpha}.S_1 \cup S'_1 \to S_2 \cup S'_2$.*

Démonstration. Nous raisonnons par récurrence sur la longueur de la dérivation de type $D : F$:

- Cas J est de la forme : $\dfrac{\Omega(p) = F}{p : F}$ Le résultat est trivial dans ce cas.
- Cas J est de la forme :

$$\dfrac{\dfrac{J_1}{D_1 : S_1 \to S_2} \qquad \dfrac{J_2}{D_2 : S'_1 \to S'_2}}{D_1 \mid D_2 : (S_1 \cup S'_1) \to (S_2 \cup S'_2)}$$

  En utilisant l'hypothèse de récurrence, nous pouvons supposer qu'il existe S_3, S'_3, S_4 et S'_4 tels que le typage $p : S_3 \to S_4$ est valide et $S_1 = S_3 \cup S'_3$ et $S_2 = S_4 \cup S'_4$. Ainsi, nous pouvons définir $S_5 \triangleq S'_3 \cup S'_1$ et $S_6 \triangleq S'_4 \cup S'_2$, ce qui nous donne le résultat : $S_1 \cup S'_1 = S_3 \cup S_5$ et $S_2 \cup S'_2 = S_4 \cup S_6$.
- Cas J est de la forme :

$$\dfrac{\dfrac{J'}{D' : F'}}{D' : \sigma(F')}$$

  En utilisant l'hypothèse de récurrence, nous pouvons supposer qu'il existe $\overline{\alpha}$, S_1, S'_1, S_2 and S'_2 tels que le typage $p : \forall\overline{\alpha}.S_1 \to S_2$ est valide et $F' = \forall\overline{\alpha}.S_1 \cup S'_1 \to S_2 \cup S'_2$. Ainsi, en utilisant l'$\alpha$-conversion, nous pouvons supposer que $\sigma(F') = \forall\overline{\alpha}.\sigma(S_1) \cup \sigma(S'_1) \to \sigma(S_2) \cup \sigma(S'_2)$. De plus, comme le typage $p : \forall\overline{\alpha}.S_1 \to S_2$ est valide, nous pouvons appliquer la règle de typage T :Subst pour obtenir une dérivation de type de $p : \forall\overline{\alpha}.\sigma(S_1) \to \sigma(S_2)$. Nous avons alors le résultat.
- Cas J est de la forme :

$$\dfrac{\dfrac{J'}{D' : F'}}{D' : \forall\alpha.F'}$$

  En utilisant l'hypothèse de récurrence, nous pouvons supposer qu'il existe $\overline{\alpha}$, S_1, S'_1, S_2 and S'_2 tels que le typage $p : \forall\overline{\alpha}.S_1 \to S_2$ est valide et $F' = \forall\overline{\alpha}.S_1 \cup S'_1 \to S_2 \cup S'_2$. Nous pouvons appliquer la règle de typage T :Gen pour obtenir une dérivation de type de $p : \forall\alpha.\forall\overline{\alpha}.S_1 \to S_2$. Nous avons alors le résultat.
- Cas J est de la forme :

$$\dfrac{\dfrac{J'}{D : \forall\alpha.F'}}{D : F'}$$

En utilisant l'hypothèse de récurrence, nous pouvons supposer qu'il existe $\overline{\beta}$, S_1, S'_1, S_2 and S'_2 tels que le typage $p : \forall\overline{\beta}.S_1 \to S_2$ est valide et $\forall\alpha.F' = \forall\overline{\beta}.S_1 \cup S'_1 \to S_2 \cup S'_2$. Nous pouvons appliquer la règle de typage T :Inst pour obtenir $\overline{\beta} \setminus \{\alpha\}$. Nous avons alors le résultat.

– Cas J est de la forme :

$$\frac{\dfrac{J'}{D' : F'} \qquad F' \leq F''}{D' : F''}$$

En utilisant l'hypothèse de récurrence, nous pouvons supposer qu'il existe $\overline{\alpha}$, S_1, S'_1, S_2 and S'_2 tels que le typage $p : \forall\overline{\alpha}.S_1 \to S_2$ est valide et $F' = \forall\overline{\alpha}.S_1 \cup S'_1 \to S_2 \cup S'_2$. Then, as $F' \leq F''$, we have $F'' = \forall\overline{\alpha}.S_3 \cup S'_3 \to S_4 \cup S'_4$ with $S_3 \leq S_1$, $S'_3 \leq S'_1$, $S_2 \leq S_4$ and $S'_2 \leq S'_4$. Nous pouvons appliquer la règle de typage T :Sub pour obtenir une dérivation de type de $p : \forall\overline{\alpha}.S_3 \to S_4$. Nous avons alors le résultat.

□

B.3 Correction

Rappelons tout d'abord ce qu'est une d'erreur avant de prouver qu'un programme bien typé ne comporte aucune de ces erreurs :

Définition 17. *Un processus D a une* erreur *si et seulement si il existe un envoi de message $\overline{e}\langle M\rangle$, un processus D' et un contexte d'évaluation E tel que $D = E[\overline{e}\langle M\rangle \mid D']$ et $\overline{e}\langle M\rangle$ n'est pas un argument valide de D',* i.e. $\overline{e}\langle M\rangle \notin \mathtt{InM}(D')$.

Théorème 13. *Supposons donnés un programme D et un type de processus F tels que le typage $D : F$ est valide. Alors D ne contient aucune erreur.*

Démonstration. Nous raisonnons par récurrence sur la taille de D :

– Les cas où $D = p$ ou $D = \overline{e}\langle M\rangle$ sont trivaux.
– Cas $D = c[I\,/\,O][D']$. De par la construction du système de type, D possède un type valide si et seulement si D' en possède un aussi. Ainsi, en utilisant l'hypotèse de récurrence, D' ne contient pas d'erreur, ce qui implique que D non plus n'en possède pas.
– Cas $D = (D_1 \mid D_2)$. En utilisant la même approche que dans le cas précédent, D_1 (resp. D_2) possède un type valide $S_1 \to S_2$ (resp. $S'_1 \to S'_2$), et ne contient donc aucune erreur. Supposons maintenant que D possède une erreur : en utilisant la commutativité de la composition parallèle, nous pouvons donc supposer qu'il existe un envoi de message $\overline{e}\langle M\rangle$ dans D_1 et un composant primitif p dans D_2 tels que $e \in \mathtt{Inc}(p)$ et $\overline{e}\langle M\rangle \notin \mathtt{Inm}(p)$. Comme l'envoi de message $\overline{e}\langle M\rangle$ est dans D_1, nous pouvons utiliser le lemme 6 qui nous donne l'existence de $T \in S_2(e)$ tel que le typage $M : T$ est valide. De plus, comme le composant primitif p est dans D_2, nous pouvons utiliser le lemme 7 qui nous donne l'existence de S_3, S'_3, S_4 et S'_4

tels que $S'_1 = S_3 \cup S_3$, $S'_2 = S_4 \cup S'_4$ et le typage $p : S_3 \to S_4$ est valide. Ainsi, comme $S_2 \precsim S'_1$ et $S'_2 \precsim S_1$, nous avons $S_4 \precsim \emptyset$ et $e : (T) \precsim S_3$. Par conséquent, nous avons $\overline{e}\langle M \rangle \in \mathtt{Inm}(p)$ par définition du type d'un composant primitif, ce qui contredit notre hypothèse $\overline{e}\langle M \rangle \notin \mathtt{Inm}(p)$. Nous pouvons donc en conclure qu'il n'existe pas d'erreur dans D.

□

B.4 Stabilité

Dans cette partie, nous ne considérons des types de processus sans variable liée, afin de simplifier les preuves. Remarquons toutefois que le lemme 3 nous assure que les résultats présentés ici sont généralisable à n'importe quel type de processus.

Définition 18. *Supposons donnés deux programmes D_1 et D_2. Nous écrivons $D_1 \sqsubseteq D_2$ si et seulement si pour tout type de processus $F \triangleq S_1 \to S_2$ tel que $D_1 : F_1$, il existe $F' \triangleq S_1 \to S'_2$ avec $D_2 : F_2$ et $S'_2 \subset S_2$.*

Lemme 8. *Pour tout contexte d'exécution E et paires de processus (D, D'), $D \sqsubseteq D'$ implique $E[D] \sqsubseteq E[D']$.*

Démonstration. Nous raisonnons par récurrence sur la structure du contexte E :

- Cas $E = []$. Ce cas est trivial.
- Cas $E = D_k \mid E'$. Considérons le typage valide $E[D] : S_1 \to S_2$. Nous pouvons utiliser le lemme 5 qui nous donne l'existence d'une dérivation en forme normale J de $E[D] : S_1 \to S_2$. Par construction, J est de la forme :

$$\frac{\dfrac{J_1}{D_k : S_1^1 \to S_2^1} \qquad \dfrac{J_2}{E'[D] : S_1^2 \to S_2^2}}{E[D] : (S_1^1 \cup S_1^2) \to (S_2^1 \cup S_2^2)}$$

  En utilisant l'hypothèse de récurrence, il existe un type ensemble S_3 avec $S_3 \subset S_2^2$ et une dérivation de type J_3 de $E'[D'] : S_1^2 \to S_3$. De plus, la définition du prédicat $\precsim$ nous donne que $S_3 \precsim S_1^1$, ce qui implique que la dérivation de type suivante est valide, nous donnant alors le résultat attendu :

$$\frac{\dfrac{J_1}{D_k : S_1^1 \to S_2^1} \qquad \dfrac{J_3}{E'[D'] : S_1^2 \to S_3}}{E[D] : (S_1^1 \cup S_1^2) \to (S_2^1 \cup S_3)}$$

- Cas $E = c[I\,/\,O][E']$. Considérons le typage valide $E[D] : S_1 \to S_2$. Utilisant le lemme 5, nous pouvons supposer qu'il existe une dérivation en forme normale J de $E[D] : S_1 \to S_2$. Par construction, J est de la forme :

$$\frac{\dfrac{J_1}{D : S'_1 \to S'_2} \qquad S_1 \precsim S'_1 \qquad S'_2 \precsim S_2}{c[I\,/\,O][D] : S_1 \cap I \to S_2 \cap O}$$

En utilisant l'hypothèse de récurrence, il existe un type ensemble S_3 avec $S_3 \subset S_2^2$ et une dérivation de type J_2 de $E'[D'] : S'_1 \to S_3$. Nous construisons alors le type ensemble S'_3 tel que $S'_3 = S_3 \cap O$ et la dérivation de type suivante, qui nous donne le résultat (en effet, comme $S_3 \subset S'_2$ et $S'_2 \precsim S_2$, nous avons par construction $S'_3 \subset S_2$) :

$$\frac{S_3 \precsim S'_3 \qquad dc(S'_1) = I \wedge dc(S'_3) = O \qquad \dfrac{J_2}{D' : S'_1 \to S_3} \quad S_1 \precsim S'_1 \qquad S'_1 \to S'_3 \text{ is consistent}}{c[I\,/\,O][D'] : S_1 \to S'_3}$$

□

Théorème 14. *Supposons donnés deux programmes D_1 et D_2 tels que $D_1 \rhd D_2$. Nous avons alors que $D_1 \sqsubseteq D_2$.*

Démonstration. Nous raisonnons par cas sur la règle de réduction utilisée dans $D_1 \rhd D_2$:

- Cas R :Contexte : $D_1 = E[D]$ et $D_2 = E[D']$. En utilisant l'hypothèse de récurrence, nous avons $D \sqsubseteq D'$. Nous pouvons alors utiliser le lemme 8 pour avoir le résutat.
- Cas R :Entrée. Supposons que D_1 est typable. Avec le lemme 5, il existe une dérivation de type valide et en forme normale J typant l'assemblage $D_1 : J$ est donc de la forme suivante.

$$\frac{\dfrac{J_1}{\overline{e}\langle M\rangle : \emptyset \to e : (T)} \qquad \dfrac{\dfrac{J_2}{D' : S_1 \to S_2}}{c[I\,/\,O][D'] : S_1 \cap I \to S_2 \cap O}}{D_1 : (S_1 \cap I) \to (S_2 \cap O) \cup e : (T)}$$

La dérivation suivante nous donne alors clairement le résultat (nous avons en effet $(S_2 \cup e : (T)) \cap O \subset (S_2 \cap O) \cup e : (T)$) :

$$\frac{\dfrac{\dfrac{J_1}{\overline{e}\langle M\rangle : \emptyset \to e : (T)} \qquad \dfrac{J_2}{D' : S_1 \to S_2}}{D' \mid \overline{e}\langle M\rangle : S_1 \to S_2 \cup e : (T)}}{D_2 : S_1 \cap I \to (S_2 \cup e : (T)) \cap O}$$

- Case R :Sortie. Supposons que D_1 est typable. Avec le lemme 5, il existe une dérivation de type valide et en forme normale J typant l'assemblage $D_1 : J$ est donc de la forme suivante.

$$\frac{\dfrac{\dfrac{J_1}{\overline{e}\langle M\rangle : \emptyset \to e : (T)} \qquad \dfrac{J_2}{D' : S_1 \to S_2}}{D' \mid \overline{e}\langle M\rangle : S_1 \to S_2 \cup e : (T)}}{D_1 : S_1 \cap I \to (S_2 \cup e : (T)) \cap O}$$

De plus, comme la règle de réduction R :Sortie peut s'appliquer sur D_1, nous avons $e \in O$, et donc $(S_2 \cup e : (T)) \cap O = (S_2 \cap O) \cup e : (T)$. La dérivation suivante nous donne alors clairement le résultat :

$$\dfrac{\dfrac{J_1}{\overline{e}\langle M\rangle \ : \ \emptyset \to e : (T)} \qquad \dfrac{\dfrac{J_2}{D' \ : \ S_1 \to S_2}}{c[I\,/\,O][D'] \ : \ S_1 \cap I \to S_2 \cap O}}{D_2 \ : \ S_1 \cap I \to (S_2 \cap O) \cup e : (T)}$$

– Case R :Primitif. Nous avons le résultat trivialement, en utilisant la définition de la fonction Ω (voir la partie 3.2.3 pour plus d'informations).

□

Annexe C

Annexe du chapitre 4

This last annex is in english and presents the proofs of our inference algorithm described section 4.1.

C.1 Constraint Generation

In this subsection, we present the properties of the constraint generation algorithm, the first step of the inference algorithm. These properties are correction and completeness, meaning that the algorithm generates a satisfiable constraint if and only if the input program has a valid type.

C.1.1 Correction

Lemme 9. *Let suppose given a valid inference statement of $v : \tau$. Then there exist a type derivation of $v : \tau$.*

Démonstration. By induction on the inference derivation of $e : \tau$, the result is evident. □

Lemme 10. *Let suppose given a valid inference statement of $R \vdash M : T$. Then there exist a type derivation of $R \vdash M : T$.*

Démonstration. By induction on the inference derivation of $R \vdash M : T$, the result is evident. □

Théorème 15. *Let suppose given a valid derivation of $D : F[C]$. Then, for every substitution σ such that $\sigma \vDash C$, there exist a type derivation of $D : \sigma(F)$.*

Démonstration. By induction on D :

- Case $D = p$. This case is evident with the typing rules T :Primitive, T :Inst, T :Subst and T :Sub.

– Case $D = \overline{e}\langle M\rangle$. Because of the definition of the constraint generation algorithm, the generation derivation has the form :

$$\frac{\dfrac{J}{\emptyset \vdash M \,:\, T} \qquad \eta, \xi \text{ fresh}}{\overline{e}\langle M\rangle \,:\, \emptyset \to e : (\xi[\eta])\,[T \leq \xi[\eta]]}$$

Using the previous lemma, the typing statement $\emptyset \vdash M \,:\, T$ holds : let note J' a typing derivation for this statement. We can then have the result in this case with the following derivation :

$$\text{T :Sub}\ \frac{\text{T :Subst}\ \dfrac{\text{T :Channel}\ \dfrac{\dfrac{J}{\emptyset \vdash M \,:\, T}}{\overline{e}\langle M\rangle \,:\, \emptyset \to e : (T)}}{\overline{e}\langle M\rangle \,:\, \sigma(\emptyset \to e : (T))}}{\overline{e}\langle M\rangle \,:\, \sigma(\emptyset \to e : (\xi[\eta]))}$$

– Case $D = D_1 \mid D_2$. The inference derivation has the form :

$$\frac{D_1 \,:\, S_1 \to S_2\,[C_1] \qquad D_2 \,:\, S'_1 \to S'_2\,[C_2]}{D \,:\, S_1 \cup S'_1 \to S_2 \cup S'_2\,[C_1 \wedge C_2 \wedge S_2 \precsim S'_1 \wedge S'_2 \precsim S_1]}$$

As $\sigma \vDash C$, we have $\sigma \vDash C_1$ and $\sigma \vDash C_2$. Thus, per the induction hypothesis, there exist a type derivation for both statement $D_1 \,:\, \sigma(S_1) \to \sigma(S_2)$ and $D_2 \,:\, \sigma(S'_1) \to \sigma(S'_2)$. Moreover, as $\sigma \vDash S_2 \precsim S'_1 \wedge S'_2 \precsim S_1$, we have $\sigma(S_2) \precsim \sigma(S'_1)$ and $\sigma(S_2) \precsim \sigma(S'_1)$. We can thus apply the typing rule for parallel composition : there exists a type derivation of $D \,:\, \sigma(S_1 \cup S'_1) \to \sigma(S_2 \cup S'_2)$.

– Case $D = c[I\,/\,O][D']$. The inference derivation has the form :

$$\frac{D' \,:\, S_1 \to S_2\,[C]}{c[I\,/\,O][D'] \,:\, (S_1 \cap I) \to (S_2 \cap O)\,[C]}$$

Per induction, as $\sigma \vDash C$, $D' \,:\, \sigma(S_1 \to S_2)$ holds. Moreover, per construction, we have $(S_1 \cap I) \precsim S_1$ and $S_2 \precsim (S_2 \cap O)$. This implies that $\sigma(S_1 \cap I) \precsim \sigma(S_1)$ and $\sigma(S_2) \precsim \sigma(S_2 \cap O)$. Thus, we can apply the component typing rule to have a type derivation of $D \,:\, \sigma(F)$.

□

C.1.2 Completeness

Lemme 11. *Let consider a type T^R, a routing variable $\xi^{R'}$ and a message variable η such that $R' \subset R$. Then we can construct σ such that* $\mathrm{dom}(\sigma) = \{\xi, \eta\}$ *and $\sigma(\xi[\eta]) = T$.*

Démonstration. By induction on T. □

Lemme 12. *Let suppose given a valid type statement $v : E$. Then, there exist a valid constraint generation derivation of $v : E$.*

Démonstration. By induction on the type derivation of $v : \tau$. □

Lemme 13. *Let suppose given a valid type statement $R \vdash M : T$ and an inference derivation $R : M : T'$. Then there exist a substitution σ with* $\mathrm{dom}(\sigma) \cap \Im(\sigma) = \emptyset$ *and $\sigma(T') = T$.*

Démonstration. By induction on the type derivation J of $M : T$:

– Case J has the form :

$$\frac{\dfrac{J'}{v : E} \qquad R \subset R' \qquad \mathtt{kind}(\xi^{R'}) = \mathfrak{F}}{R \vdash v^{\emptyset} : \xi^{R'}[E]}$$

Using the previous lemma, the inference statement $v : E$ holds. We can then trivially construct an inference derivation of $R \vdash v^{\emptyset} : \xi^{R}[E]$, with ξ fresh and not duplicable. Finally, the substitution $\xi^{R} \to \xi[^{]}R'\bullet$ gives us the result.

– Case J has the form :

$$\frac{\dfrac{J'}{R \cup \{r\} \vdash v^{\delta} : T} \qquad R \cup \{r\} \vdash T_k}{R \vdash v^{\mathrm{r};\delta} : r(T, T_k)}$$

Using the induction hypothesis, the inference statement $R \cup \{r\} \vdash v^{\delta} : T'$, with a substitution σ_1 such that $\sigma_1(T') = T$. We can then trivially construct an inference derivation of $R \vdash v^{\mathrm{r};\delta} : r(T', \xi^{R\cup\{r\}}[\eta])$, with ξ and η fresh. Let then construct the substitution σ_2 such that $\sigma_2(\xi[\eta]) = T_k$: the substitution $\sigma \triangleq \sigma_2 \circ \sigma_1$ gives us the result.

– Case J has the form :

$$\frac{R \cup \{r\} \vdash v^{\delta} : T \qquad R \cup \{r\} \vdash T_k}{R \vdash M^{\downarrow r;\delta} : r(T_k, T)}$$

Using the same approach as before, we have the result.

□

Théorème 16 (Completeness)**.** *Let suppose given a valid type statement $D : \forall\overline{\beta}.S_1 \to S_2$. Then, for all substitution σ' and valid type F' with $\forall\overline{\beta}.\sigma'(S_1 \to S_2) \leq F'$, there exist an inference derivation of $D : S_1' \to S_2'\,[C]$, some variables $\overline{\alpha}$ and a substitution σ such that* $\mathrm{dom}(\sigma) \cap \Im(\sigma) = \emptyset$*, $\sigma \vDash C$ and $\forall\overline{\alpha}.\sigma(S_1' \to S_2') = F'$*

Démonstration. By induction on the type derivation J of $D : S_1 \to S_2$:

– Case J has the form :

$$\frac{\emptyset \vdash v^{\delta} : T_1}{\overline{e}\langle v^{\delta}\rangle : \emptyset \to e : (T_1)}$$

Let consider a substitution σ' and a valid type F' such that $\sigma'(\emptyset \to e : (T_1)) \leq F'$. With the previous lemma, there exist a substitution σ_1 and a type T' such that the inference statement $\delta : v^\delta : T'$ holds and $\sigma(T') = T_1$. We can then construct the inference derivation of $\overline{e}\langle v^\delta \rangle : \emptyset \to e : (\xi[\eta])\,[T \leq \xi[\eta]]$ with ξ and η fresh. Let now remark that because we have $\emptyset \to e : (T) \leq F'$, there exists T_2 such that $\sigma'(T_1) \leq T_2$ and $F' = \emptyset \to e : (T_2)$. We can then construct the substitution σ_2 such that $\sigma_2(\xi[\eta]) = T_2$, $\mathrm{dom}(\sigma_2) = \{\xi, \eta\}$ and the substitution σ such that $\sigma \triangleq \sigma_2 \circ \sigma' \circ \sigma_1$. We trivially have that $\sigma \vDash T \leq \xi[\eta]$ and $\sigma(\emptyset \to e : (\xi[\eta])) = F'$

– Case J has the form :

$$\frac{\Omega(p) = \forall\overline{\beta}.S_1 \to S_2}{p : \forall\overline{\beta}.S_1 \to S_2}$$

Let consider a substitution σ' and a valid type F' such that $\forall\overline{\beta}.\sigma'(S_1 \to S_2) \leq F'$. Let then consider the valid inference statement

$$p : S_1' \to S_2'\,[S_1' \precsim S_1\{^{\overline{\alpha}'}/_{\overline{\beta}}\} \wedge S_2\{^{\overline{\alpha}'}/_{\overline{\beta}}\} \precsim S_2'] \qquad \text{with} \quad S_1 = \bigcup e_i : (T_i)$$
$$S_2 = \bigcup e_j : (T_j) \quad S_1' = \bigcup e_i : (\xi_i[\eta_i]) \quad S_2' = \bigcup e_j : (\xi_j[\eta_j])$$

Because we have $\forall\overline{\beta}.\sigma'(S_1 \to S_2) \leq F'$ there exist T_i' and T_j' such that

$$F' = \forall\overline{\beta}.\bigcup e_i : (T_i') \to e_j : (T_j') \quad \sigma(T_i) \leq T_i' \quad \sigma(T_j) \leq T_j'$$

Let consider the substitutions σ_i (resp. σ_j) such that $\sigma_i(\xi_i[\eta_i]) = T_i'$ (resp. $\sigma_j(\xi_j[\eta_j]) = T_j'$) and $\sigma_k = \overline{\beta} \to \overline{\alpha}'$. Let finally construct $\sigma \triangleq \bigodot \sigma_i \circ \bigodot \sigma_j \circ \sigma_k$: this substitution with $\overline{\alpha} = \overline{\beta}$ gives us trivially the result.

– Case J is of the form :

$$\frac{\dfrac{J_1}{D_1 : S_1 \to S_2} \qquad \dfrac{J_2}{D_2 : S_3 \to S_4}}{D_1 \mid D_2 : (S_1 \cup S_3) \to (S_2 \cup S_4)}$$

Because we have $\sigma'(F) \leq F'$ there exists S_1', S_2', S_3' and S_4' such that

$$F' = (S_1' \cup S_3') \to (S_2' \cup S_4') \qquad S_1' \leq \sigma'(S_1) \quad \sigma'(S_2) \leq S_2' \quad S_3' \leq \sigma'(S_3) \quad \sigma'(S_4) \leq S_4'$$

Then, using the induction hypothesis, there exists an valid inference statement $D_1 : S_1'' \to S_2''\,[C_1]$ (resp. $D_2 : S_3'' \to S_4''\,[C_2]$) and a substitution σ_1 (resp. σ_2) such that $\sigma_1(S_1'' \to S_2'') = S_1' \to S_2'$ (resp. $\sigma_2(S_3'' \to S_4'') = S_3' \to S_4'$). Let define $\sigma = \sigma_2 \circ \sigma_1$. Per construction, we have $\mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset$ and $\Im(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset$, so $\sigma(F_1) = S_1' \to S_2'$ and $\sigma(F_2) = S_3' \to S_4'$ hold. Finally, as F' is valid, $\sigma \vDash C_1 \wedge C_2 \wedge S_4'' \precsim S_1'' \wedge S_2'' \precsim S_3''$, which gives us the result.

– Case J has the form :

$$\frac{\dfrac{J'}{D : F}}{D : \sigma_1(F)}$$

Let define $F = \forall\overline{\alpha}.S_1 \to S_2$: using α conversion, we can suppose that $\overline{\alpha} \cap (\mathrm{dom}(\sigma_1) \cup \Im(\sigma_1)) = \emptyset$. We thus have $\sigma_1(F) = \forall\overline{\alpha}.\sigma_1(S_1 \to S_2)$. Let also set a substitution σ' and a valid type F' such that $\forall\overline{\alpha}.\sigma'(\sigma_1(S_1 \to S_2)) \leq F'$, i.e. $\forall\overline{\alpha}.\sigma \circ \sigma_1(S_1 \to S_2) \leq F'$. Thus, using the induction hypothesis, there exist a valid inference statement $D : F_1\,[C]$ and a substitution σ' such that $\sigma' \vDash C$ and $\sigma'(F_1) = F'$: we then have the result.

– Case J has the form :

$$\frac{\dfrac{J'}{D\ :\ F} \qquad F \leq F''}{D\ :\ F''}$$

Let consider $F'' = \forall\overline{\alpha}.S_1 \to S_2$, a substitution σ' and a valid type F' such that $\forall\overline{\alpha}.\sigma'(S_1 \to S_2) \leq F'$. Moreover, as $F \leq F''$, we have $F = \forall\overline{\alpha}.S_1' \to S_2'$ because the subtyping relation is transitive and stable through substitution, we then have $\forall\overline{\alpha}.\sigma'(S_1' \to S_2') \leq F'$. Thus, using the induction hypothesis, there exist a valid inference statement $D : F_1\,[C]$ and a substitution σ' such that $\sigma' \vDash C$ and $\sigma'(F_1) = F'$: we then have the result.

– Case J has the form :

$$\frac{\dfrac{J'}{D\ :\ F}}{D\ :\ \forall\alpha.F}$$

Let define $F = \forall\overline{\beta}.S_1 \to S_2$ and consider a substitution σ' and a valid type F' with $\forall\alpha.\forall\overline{\beta}.\sigma'(S_1 \to S_2) \leq F'$. Moreover, per definition of the subtyping relation, there exist a valid type F'' with $F' = \forall\alpha.F''$ and $\forall\overline{\beta}.\sigma'(S_1 \to S_2) \leq F''$. Using the induction hypothesis, there exists $\overline{\beta}'$, a valid inference statement $D : F_1\,[C]$ and a substitution σ such that $\sigma \vDash C$ and $\forall\overline{\beta}'.\sigma(F_1) = F''$. We then have $\forall\alpha.\forall\overline{\beta}'.F_1 = F'$, which gives us the result.

– Case J has the form :

$$\frac{\dfrac{J}{D\ :\ \forall\alpha.F}}{D\ :\ F}$$

Let define $F = \forall\overline{\beta}.S_1 \to S_2$ and consider a substitution σ' and a valid type F' with $\forall\overline{\beta}.\sigma'(S_1 \to S_2) \leq F'$. With the definition of the subtyping relation, we have $\forall\alpha.\forall\overline{\beta}.\sigma'(S_1 \to S_2) \leq \forall\alpha.F'$. Using the induction hypothesis, there exists $\overline{\beta}'$, a valid inference statement $D : F_1\,[C]$ and a substitution σ such that $\sigma \vDash C$ and $\forall\overline{\beta}'.\sigma(F_1) = \forall\alpha.F'$. We thus have $\forall\overline{\beta}' \setminus \{\alpha\}.\sigma(F_1) = F'$, which gives us the result.

– Case J_k' has the form :

$$\frac{\dfrac{J_1}{D'\ :\ S_1' \to S_2'} \qquad S_1 \precsim S_1' \qquad S_2' \precsim S_2 \qquad dc(S_1') = I \wedge dc(S_2') = O}{c[I\,/\,O][D'] :\ S_1 \to S_2}$$

We can use the induction hypothesis to get a substitution σ giving the result for the statement $D' : S'_1 \to S'_2$. Using the definition of the $\precsim$ predicate, we have the result : $S_1 \subset \sigma(S'_1)$ and $S_2 \subset \Sigma(S'_2)$. We use the same approach to define σ'.

□

C.2 Constraint solving

The proofs of the god properties of the constraint solving algorithm are based on the definition of a function `close` defined on constraint. We first recall the definition of this function, before proving the different properties of the algorithm.

Définition 19. *Given a constraint C, let* $\texttt{close}(C)$ *be the least (w.r.t $\subset$) constraint such that :*

- $C \subset \texttt{close}(C)$ *and*
- *if* $C_1 \subset C$ *and* $C_1 \blacktriangleright C_2$*, then* $C_2 \subset \texttt{close}(C)$.

Théorème 17. *Given a constraint C, the function* `close` *has a least fixed point containing C, called the* closure *of C.*

Démonstration. One can remark that every rules produce a finite set of smaller term than the input constraint. Moreover, none of them introduce new routing names, nor new row label, nor new type constructors. Finally, the rules that introduce fresh variables can be applied at most once with a given input, implying that the number of possibly created terms is finite. So, the sequence $\texttt{close}(C), \texttt{close}^2(C), \ldots$ must converge to the least fixed point containing C. □

Théorème 18. *C and* $\texttt{close}(C)$ *are logically equivalent.*

Démonstration. As we have $C \subset \texttt{close}(C)$, we trivially have $\texttt{close}(C) \vDash C$. To prove the converse, we show that for all closure rules $C_1 \blacktriangleright C_2$, we have $C_1 \vDash C_2$. Let suppose given σ such that $\sigma \vDash C_1$:

- Case $C_1 = e : (T) \cup S_1 \precsim e : (T') \cup S_2$. As $\sigma \vDash C_1$, we have $\sigma(e : (T) \cup S_1) \leq \sigma(e : (T') \cup S_2)$, implying that $\sigma(T) \leq \sigma(T')$.. Hence, $\sigma \vDash C_2$ which gives us the result is this case.
- Case $C_1 = r(T_1, T_2) \leq r({T_1}', {T_2}')$. As $\sigma \vDash C_1$, we have $\sigma(r(T_1, T_2)) \leq \sigma(r({T_1}', {T_2}'))$, which implies that $\sigma(T_1) \leq \sigma(T'_1)$ and $\sigma(T_2) \leq \sigma(T'_2)$. Hence, $\sigma \vDash C_2$ which gives us the result is this case.
- Case $C_1 = r(T_1^{R \uplus \{r\}}, T_2^{R \uplus \{r\}}) \leq r'(T_3^{R \uplus \{r\}}, T_4^{R \uplus \{r\}})$. As $\sigma \vDash C_1$, we have $\sigma(r(T_1^{R \uplus \{r\}}, T_2^{R \uplus \{r\}})) \leq \sigma(r'(T_3^{R \uplus \{r\}}, T_4^{R \uplus \{r\}}))$. Thus, there exist T'_1, T'_2, T'_3 and T'_4 such that $\sigma(r(T_1^{R \uplus \{r\}}, T_2^{R \uplus \{r\}})) = r'(T'_1, T'_2)$ and $\sigma(r'(T_3^{R \uplus \{r\}}, T_4^{R \uplus \{r\}})) = r'(T'_3, T'_4)$. Moreover, we have $T'_i \leq T_i$, for all $1 \leq i \leq 2$ and $T_i \leq T'_i$, for all $3 \leq i \leq 4$. We can then construct the substitution σ' that extends σ with η_i, ξ_i, $1 \leq i \leq 4$ such that $\sigma'(\xi_i[\eta_i]) = T'_i$. With this substitution, we can see that $\sigma \vDash C_2$, which gives us the result in this case.

- Case $C_1 = \xi^R[E] \leq r(T_1, T_2)$. Per definition, we have $\sigma(\xi^R[E]) \leq \sigma(r(T_1, T_2))$, which means that there exist T_1' and T_2' such that $\sigma(\xi^R[E]) = r(T_1', T_2')$ and $T_1' \leq T_1$ and $T_2' \leq T_2$. We can then construct the substitution σ' that extends σ with η_i, ξ_i, $1 \leq i \leq 2$ such that $\sigma'(\xi_i[\eta_i]) = T_i'$. With this substitution, we can see that $\sigma \vDash C_2$, which gives us the result in this case.
- The case where $C_1 = \xi^R[E] \leq r(T_1, T_2)$ can be solved using the same approach as the previous case.
- Case $C_1 = T_1 \leq \xi[E] \wedge \xi[E] \leq T_2$. As we have $\sigma(T_1) \leq \sigma(\xi[E]) \leq \sigma(T_2)$, and because $\leq$ is transitive, we have $\sigma(T_1) \leq \sigma(T_2)$. We thus have the result : $\sigma \vDash C_2$.
- Case $C_1 = [\xi : s(E_1, \ldots, E_n) \leq s'(E_1', \ldots, E_n')]$ with $s \leq s'$. Per definition of the $\vDash$ predicate, we have $\sigma(\xi[s(E_1, \ldots, E_n)]) \leq \sigma(\xi[s'(E_1', \ldots, E_n')])$. Per definition of the sub-typing relation, we have $\sigma(\xi[E_i]) \leq \sigma(\xi[E_i'])$. We thus have $\sigma \vDash C_2$, which gives us the result in this case.
- Case $C_1 = [\xi : s(E_1, \ldots, E_n) \leq s'(E_1', \ldots, E_n')]$ with $s \not\leq s'$. In this case, given the definition of the sub-typing relation, it is evident that C_1 is not satisfiable. Thus C_1 and C_2 are equivalent.
- Case $C_1 = [\xi : \{W_1\} \leq \{W_2\}]$. As $\sigma(\xi[\{W_1\}]) \leq \sigma(\xi[\{W_2\}])$, we have per definition of the sub-typing relation $\sigma(\xi[W_1]) \leq \sigma(\xi[W_2])$. Thus, $\sigma \vDash C_2$, and we have the result.
- Cases $C_1 = [\xi : a : \mathrm{Pre}(E_1); W_1 \leq a : \mathrm{Pre}(E_2); W_2]$ and $C_1 = [\xi : a : \mathrm{Pre}(E_1); W_1^{l \uplus \{a\}} \leq b : \mathrm{Pre}(E_2); W_2^{l \uplus \{b\}}]$. These two cases are similar to the previous one, and their proof are just based on the definition of the sub-typing relation.
- Case $C_1 = [\xi : a : \mathrm{Pre}(E_1); W_1 \succeq a : \mathrm{Abs}; W_2]$. This constraint is trivially not satisfiable, as it requires that an absent field should be present. We then easily have that C_1 is logically equivalent to C_2.
- Case $C_1 = [\xi : \tau \leq \alpha] \wedge [\xi : \alpha \leq \tau']$. Because the sub-typing relation is transitive, this case is trivial.
- Case $C_1 = [\xi : \tau_1 \leq \tau_2] \wedge \xi[E] \leq \xi'[E']$. Per definition of the $\vDash$ predicate, We have $\sigma(\xi[\tau_1]) \leq \sigma(\xi[\tau_2])$ and $\sigma(\xi[E]) \leq \sigma(\xi'[E'])$. This second relation is possible only if $\sigma(\xi) = \sigma(\xi')$, per definition of the sub-typing relation. Thus, as we have $\sigma(\xi) = \sigma(\xi')$, we have $\sigma(\xi[\tau_1]) = \sigma(\xi'[\tau_1])$ and $\sigma(\xi[\tau_2]) = \sigma(\xi'[\tau_2])$. Hence, we have $\sigma \vDash [\xi' : \tau_1 \leq \tau_2]$.
- case $C_1 = [\xi : \tau_1 \leq \tau_2] \wedge \xi[E] \leq r(\xi_1[E_1], \xi_2[E_2])$. Per definition of the $\vDash$ predicate, We have $\sigma(\xi[\tau_1]) \leq \sigma(\xi[\tau_2])$ and $\sigma(r(\xi_1[E_1], \xi_2[E_2]))$. This second relation is possible only if $\sigma(\xi) = r(\sigma(\xi_1), \sigma(\xi_2))$, per definition of the sub-typing relation. Thus, we have $\sigma(\xi[\tau_1]) = r(\sigma(\xi_1[\tau_1]), \sigma(\xi_2[\tau_1]))$ and $\sigma(\xi[\tau_2]) = r(\sigma(\xi_1[\tau_2]), \sigma(\xi_2[\tau_2]))$. We then have $r(\sigma(\xi_1[\tau_1]), \sigma(\xi_2[\tau_1])) \leq r(\sigma(\xi_1[\tau_2]), \sigma(\xi_2[\tau_2]))$. Hence, using the definition of the sub-typing relation, we have $\sigma(\xi_1[\tau_1]) \leq \sigma(\xi_1[\tau_2])$ and $\sigma(\xi_2[\tau_1]) \leq \sigma(\xi_2[\tau_2])$. So $\sigma \vDash C_2$, which gives us the result in this case.
- Case $C_1 = [\xi_1 : \tau_1 \leq \tau_1'] \wedge [\xi_2 : \alpha \succeq \tau]$ with $\mathtt{kind}(\alpha) = \mathtt{kind}(\tau) = \mathfrak{G}$. Per definition of $\sigma \vDash C_1$, we have $\sigma(\xi_2[\alpha]) \leq \sigma(\xi_2[\tau])$. But because α and τ are global, for all variables $\beta \in fv(\tau) \cup \{\alpha\}$ and all routing variable ξ, the

image of these variables through σ is the same. So per definition, we have $\sigma(\xi_1[\alpha]) \leq \sigma(\xi_1[\tau]) : \sigma \vDash C_2$. We then have the result in this case.

- Case $C_1 = [\xi : \alpha \leq \tau]$ with $\tau \notin \mathcal{V}$ and $\texttt{kind}(\alpha) = \mathfrak{G}$. Let consider $\{\xi'_i \mid 1 \leq i \leq n\} = fv(\sigma(\xi))$. As τ is local, for each variable $\beta \in fv(\tau)$ and all $1 \leq i \leq n$ there exist τ_i^β such that $\sigma(\xi_i[\beta]) = \tau_i^\beta$. The sub-typing relation defines a lattice structure on routing types, so we can define $\tau^\beta \triangleq \sqcap \tau_i^\beta$ for all $\beta \in fv(\tau)$, and $\tau' \triangleq \tau\{^{\tau^\beta}/_\beta\}$. Let define τ_k the image of α by σ. As for all $1 \leq i \leq n$, we have $\sigma(\xi_i[\alpha]) \leq \sigma(\xi_i[\tau])$, we have $\tau_k \leq \sqcap\tau\{^{\tau_i^\beta}/_\beta\}$. And thus, we have $\tau_k \leq \tau'$. Now, let's introduce for all $\beta \in fv(\tau)$ β' fresh and global, and extend σ with $\sigma(\beta') = \tau^\beta$. One can now see that this extension of σ validates $\bigwedge([\xi : \beta' \leq \beta]) \wedge [\xi : \alpha \leq \tau\{^{\overline{\beta}'}/_{\overline{\beta}}\}]$. So, σ validates C_2, which gives us the expected result.
- Case $C_1 = [\xi : \tau \leq \alpha]$ with $\tau \notin \mathcal{V}$ and $\texttt{kind}(\alpha) = \mathfrak{G}$. Using the same approach as the previous case, we get the result in this case.
- Case $C_1 = \xi[E] \leq r(\xi_1[E_1], \xi_2[E_2])$. As we have $\sigma(\xi[E]) \leq \sigma(r(\xi_1[E_1], \xi_2[E_2]))$, using the definition of the sub-typing relation, we have $\sigma(\xi) = r(\sigma(\xi_1), \sigma(\xi_2))$. So, σ validates C_2, and we have the result in this case.
- Case $C_1 = \xi[E] \leq r(\xi_1[E_1], \xi_2[E_2])$. Using the same approach as the previous case, we get the result in this case.
- Case $C_1 = \xi_1[E_1] \leq \xi_2[E_2] \wedge \texttt{duplic}(\xi_1 \to \texttt{r}(\xi, \xi'))$. As $\sigma \vDash C_1$, we have $\sigma(\xi_1[E_1]) \leq \sigma(\xi_2[E_2])$, which implies, from the definition of the sub-typing relation that $\sigma(\xi_1) = \sigma(\xi_2)$. Thus, per definition of $\texttt{duplic}(\xi_1 \to \texttt{r}(\xi, \xi'))$, we have $\sigma \vDash \texttt{duplic}(\xi_2 \to \texttt{r}(\xi, \xi'))$. So, σ validates C_2 and thus gives us our result.
- Case $C_1 = \xi_2[E_2] \leq \xi_1[E_1] \wedge \texttt{duplic}(\xi_1 \to \texttt{r}(\xi, \xi'))$. Using the same approach as the previous case, we get the result in this case.
- Case $\texttt{duplic}(\xi \to \texttt{r}(\xi, \xi'))$ with $\texttt{kind}(\xi) = \mathfrak{F}$. This constraint is trivially not satisfiable, and thus, is equivalent to C_2.

□

C.2.1 Constraint Satisfiability

Définition 20. *A* routing equation set *is a set C_{eq} of constraints of the form $\xi = \xi'$ or $\xi = r(\xi_1, \xi_2)$ such that :*

- *$\xi = \xi' \in C_{eq} \wedge \xi' = \xi'' \Rightarrow \xi = \xi''$ (the '$=$' raises an equivalence relation between routing variables) ;*
- *$\xi^R = \xi_1^{R_1} \in C_{eq} \Rightarrow (R \subset R_1) \vee (R_1 \subset R)$;*
- *For all ξ, there is at most one constraint of the form $\xi = r(\xi_1, \xi_2)$ in C_{eq} ;*
- *Moreover, if $\xi^R = r(\xi_1, \xi_2) \in C_{eq}$, for all $\xi^R = \xi_1^{R_1} \in C_{eq}$, $R_1 \subset R$ and no constraint of the form $\xi_1^{R_1} = r(\xi_1, \xi_2)$ is in C_{eq}.*

Lemme 14. *Let suppose given a routing equation set C_{eq}. Then there exist a substitution σ such that : (i) $\forall \xi_0 \neq \xi_n$, $\sigma(\xi_0) = \xi_n \Rightarrow \exists(\xi_i)_i$, $\bigwedge(\xi_i = \xi_{i+1}) \in C_{eq}$, (ii) $\sigma(\xi_0) = r(s_1, s_2) \Rightarrow (\exists(\xi_i)_i, \bigwedge(\xi_i = \xi_{i+1})\xi_n = r'(\xi'_1, \xi'_2) \in C_{eq})$ and $\sigma(\xi'_1) = s_1$, $\sigma(\xi'_2) = s_2$, (iii) $\mathrm{dom}\sigma) \cap \Im(\sigma) = \emptyset$ and (iv) $\sigma \vDash C_{eq}$.*

Démonstration. By induction on C_{eq} :

- Case $C_{eq} = \mathbf{true}$. Using the identity substitution, we have the result.
- Case $C_{eq} = (\xi = \xi' \wedge C'_{eq})$. Using the induction hypothesis, there exist σ_1 such that (i) $\forall \xi_0 \neq \xi_n$, $\sigma_1(\xi_0) = \xi_n \Rightarrow \exists (\xi_i)_i$, $\bigwedge(\xi_i = \xi_{i+1}) \in C'_{eq}$, (ii) $\sigma_1(\xi_0) = r(s_1, s_2) \Rightarrow (\exists (\xi_i)_i$, $\bigwedge(\xi_i = \xi_{i+1})\xi_n = r'(\xi'_1, \xi'_2) \in C'_{eq})$ and $\sigma_1(\xi'_1) = s_1$, $\sigma_1(\xi'_2) = s_2$, (iii) dom$\sigma_1) \cap \Im(\sigma_1) = \emptyset$ and (iv) $\sigma_1 \vDash C'_{eq}$. We have two cases (using the symetry of $=$) :
  1. $\sigma_1(\xi) = \xi_3^{R_3}$ and $\sigma_1(\xi') = \xi_4^{R_4}$. Using renaming when necessary, we can suppose that $R_3 \subset R_4$. We can then define $\sigma_2 \triangleq \xi_3 \to \xi_4$, $\sigma \triangleq \sigma_2 \circ \sigma_1$: we trivially have that $\sigma \vDash C_{eq}$. Moreover, because dom$\sigma_1) \cap \Im(\sigma_1) = \emptyset$, we have $\xi_3^{R_3} \notin \mathrm{dom}(\sigma_1)$. Thus, we also have dom$\sigma) \cap \Im(\sigma) = \emptyset$. The two other properties of σ can also be easily proved.
  2. $\sigma_1(\xi) = r(s_1, s_2)$ and $\sigma_1(\xi') = \xi^{R''}$. Let first prove that there exist a family of routing variables (ξ_i) such that $\bigwedge(\xi_i = \xi_{i+1})\xi_n = r(\xi'_1, \xi'_2) \in C_{eq})$ with $\sigma_1(\xi'_1) = s_1$, $\sigma_1(\xi'_2) = s_2$ and $\xi_1 = \xi''$ for some ξ'_1 and ξ'_2. First, because $\sigma_1(\xi) = r(s_1, s_2)$, there exist (ξ_i) such that $\bigwedge(\xi_i = \xi_{i+1})\xi_n = r'(\xi'_1, \xi'_2) \in C_{eq})$ with $\sigma_1(\xi'_1) = s_1$, $\sigma_1(\xi'_2) = s_2$ and $\xi_1 = \xi$ Then, if $\xi'' = \xi'$, we can add create the family $(\xi', \xi_1, \ldots, \xi_n)$ to get the result. And if $\xi'' \neq \xi'$ there exist (ξ'_i) such that $\xi'_1 = \xi'$, $\xi'_m = \xi''$ and $\bigwedge(\xi'_i = \xi'_{i+1}) \in C'_{eq}$: the family $(\xi'_1, \ldots, \xi'_m, \xi_1, \ldots, \xi_n^{R_n})$ gives us the result.

     With this and the property of C_{eq}, we have that $R \subset R_n$: the substitution $\sigma_2 \triangleq \xi'' \to r(s_1, s_2)$ is valid. Let finally define $\sigma \triangleq \sigma_2 \circ \sigma_1$: one can see that σ gives us the solution as before.

  The last case one can think of is $\sigma_1(\xi) = r(s_1, s_2)$ and $\sigma_1(\xi') = r'(s_3, s_4)$, but it is not possible, because of the restriction on C_{eq} and σ.
- Case $C_{eq} = (\xi^R = r(\xi_3, \xi_4) \wedge C'_{eq})$. Using the induction hypothesis, there exist σ_1 such that (i) $\forall \xi_0 \neq \xi_n$, $\sigma_1(\xi_0) = \xi_n \Rightarrow \exists (\xi_i)_i$, $\bigwedge(\xi_i = \xi_{i+1}) \in C'_{eq}$, (ii) $\sigma_1(\xi_0) = r(s_1, s_2) \Rightarrow (\exists (\xi_i)_i$, $\bigwedge(\xi_i = \xi_{i+1})\xi_n = r'(\xi'_1, \xi'_2) \in C'_{eq})$ and $\sigma_1(\xi'_1) = s_1$, $\sigma_1(\xi'_2) = s_2$, (iii) dom$\sigma_1) \cap \Im(\sigma_1) = \emptyset$ and (iv) $\sigma_1 \vDash C'_{eq}$.
  Let first suppose that $\sigma_1(\xi) = \xi$, and define $\sigma_2 \triangleq \xi \to r(\sigma_1(\xi_3), \sigma_1(\xi_4))$, $\sigma \triangleq \sigma_2 \circ \sigma_1$: one can see that σ gives us trivially the solution.
  Let now suppose that $\sigma_1(\xi) \neq \xi$. Thus, there exist $\xi_1^{R_1}$ such that $\sigma_1(\xi) = \xi_1$ by construction of C_{eq} and σ_1. Moreover, we have that $R_1 \subset R$, also because of the constraints on C_{eq} and σ_1. Thus the substitution $\sigma_2 \triangleq \xi_1 \to r(\sigma_1(\xi_3), \sigma_1(\xi_4))$ is valid. Let finally define $\sigma \triangleq \sigma_2 \circ \sigma_1$: one can see that σ gives us trivially the solution.

□

Théorème 19. *If C is closed and does not contain* **false**, *then C is satisfiable.*

Démonstration. The principle of this proof is to construct a substitution that is supposed to validate C, using some specific constraints in C. Then, we show that the substitution validates all simple constraints in C, using the properties

of the closure algorithm, and how it is constructed. As C is not modified in this proof, we can suppose without loose of generality that there is no bounded variables in C.

Let continue with some definitions : (i) when $\alpha \in fv(C)$ and $\texttt{kind}(\alpha) = \mathfrak{G}$, let $lb(\alpha)$ be the set of lower bound of α in C, that is

$$lb(\alpha) \triangleq \{\tau \mid [\xi : \tau \leq \alpha] \in C \wedge \tau \notin \mathcal{V} \wedge \xi \in fv(C)\}$$

(ii) when $\alpha, \xi \in fv(C)$ and $\texttt{kind}(\alpha) = \mathfrak{L}$, let $lb(\xi, \alpha)$ be the set of lower bound of $\xi[\alpha]$ in C, that is

$$lb(\xi, \alpha) \triangleq \{\tau \mid [\xi : \tau \leq \alpha] \in C \wedge \tau \notin \mathcal{V}\}$$

(iii) when $\xi \in fv(C)$, let $ev(\xi)$ the routing variables equal to ξ in C, that is

$$ev(\xi) \triangleq \{\xi' \mid \xi = \xi' \in C\}$$

and (iv) when $\xi \in fv(C)$, let $er(\xi)$ the forms equal to ξ in C, that is

$$er(\xi) \triangleq \{r(\xi_1, \xi_2) \mid \xi = r(\xi_1, \xi_2) \in C\}$$

Now let consider the following system of equation, whose unknown is a substitution σ_1 :

$$\begin{array}{rl} \sigma_1(\alpha) = \sqcup\sigma_1(lb(\alpha)) & \text{if } \texttt{kind}(\alpha) = \mathfrak{G} \\ \sigma_1(\xi[\alpha]) = \sqcup\sigma_1(\xi[lb(\xi, \alpha)]) & \text{if } \texttt{kind}(\alpha) = \mathfrak{L} \end{array}$$

With a minor generalization of [89] to handle local variable, one can see that this equation system is constractive, and thus admit a solution σ_2.

Let consider a routing variable $\xi \in fv(C)$ and note $\sqcap ev(\xi)$ the set $\{\xi_1^{R_1} \mid \xi_2^{R_2} \in ev(\xi) \Rightarrow R_2 \subset R_1\} \cap ev(\xi)$. Because of the last rule of the constraint propagation, this set is empty iff $ev(\xi)$ is empty : let construct the routing equaltion set $C_{eq} \triangleq \{\xi = \xi' \mid \xi \in fv(C) \wedge \xi' \in \sqcap ev(\xi)\}$. Moreover, because C is stable under the transitivity of the '=' constraint, if $ev(\xi) \cap ev(\xi') \neq \emptyset$, then $\sqcap ev(\xi) = \sqcap ev(\xi')$. Hence, we can consider the equivalence relation $\mathcal{R}$ defined by $\xi \mathcal{R} \xi'$ iff $\sqcap ev(\xi) = \sqcap ev(\xi')$. For each equivalence class, let take a representative ξ, and with this representative, let choose $\xi' \in \sqcap ev(\xi)$. Then, if $er(\xi') \neq \emptyset$, let choose $r(\xi_1, \xi_2) \in er(\xi')$ and add the constraint $\xi' = r(\xi_1, \xi_2)$ to C_{eq}. We can see that the resulting constraint still validate the definition of *routing equaltion set*. Using the previous lemma, there exist a substitution σ_1 that validate C_{eq}, and let construct $\sigma \triangleq \sigma_2 \circ \sigma_1$.

We now need to verify that σ validate C, and not just some sub-constraints of C. In order to ensure this validation, we use the preorders $\leq_k$, and define from it a $k-$validation predicate. We then show that σ $k-$validate C then it $k+1-$validates the constraint. And as $\infty-$validation is the actual validation predicate, we have the result. The induction proof is done testing each simple constraint in C :

- $S_1 \precsim S_2$. As C is closed with the first rule, for each $e \in dc(S_1)$ we have for all $T \in S_2(e)$ $T \leq_k S_1(e)$: we thus have the result.

- $\xi[E] \leq r(T_1, T_2)$. As C is closed, there exist ξ_1, E_1, ξ_2 and E_2 such that $\xi[E] \leq r(\xi_1[E_1], \xi_2[E_2]) \wedge \xi_1[E_1] \leq T_1 \wedge \xi_2[E_2] \leq T_2 \in C$. Then, as we have $\xi_1[E_1] \leq_k T_1$ and $\xi_2[E_2] \leq_k T_2$, we have $r(\xi_1[E_1], \xi_2[E_2]) \leq_{k+1} r(T_1, T_2)$. So, we have $\xi[E] \leq_{k+1} r(T_1, T_2)$, which gives us the result.
- $r(T_1, T_2) \leq r(T_1', T_2')$. The result is evident from the definition of k-validity.
- $r(T_1, T_2) \leq r'(T_1', T_2')$ with $r \neq r'$. Using the same approach as in the second case, we have the result.
- $[\xi^R : \tau_1 \leq \tau_2]$. This case deal with the validity of constraint over message types. Such constraints, dealing with type constructors and rows were studied a lot in previous works. We then won't present the proof of the validity of our algorithm here, but invite the interested reader to check [89]
- $\xi = r\xi_1\xi_2$. If this constraint is part of the equation set used to create σ, this constraint is trivially validated. Then, let consider that $\xi = r\xi_1\xi_2 \notin C_{eq}$: there exist $\xi' \in \sqcap ev(\xi)$ and $r'(\xi_1, \xi_2) \in er(\xi')$ such that $\xi = \xi' \wedge \xi' = r'(\xi_1, \xi_2) \in C_{eq}$. Because C is closed, we have $\xi = r'(\xi_1, \xi_2) \in C$. Then, using the same approach as in the second case, we get the result.
- $\xi = \xi'$. This case is evident from the transitivity of '=' in a closed constraint, and the construction of C_{eq}.

□