Programmation Réactive Synchrone, Langage et Contrôle des Ressources

Frederic Dabrowski

UNIVERSITÉ PARIS 7 - DENIS DIDEROT

THÈSE

pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PARIS 7

Spécialité : INFORMATIQUE

présentée par

Frédéric Dabrowski

Programmation Réactive Synchrone

Langage et Contrôle des Ressources

Thèse encadrée par Roberto Amadio
Soutenue le 22 juin 2007 devant le jury composé de :

Rapporteurs :		
Jean-Yves Marion	Professeur	Ecole des mines de Nancy
Marc Pouzet	Professeur	Université Paris 11
Examinateurs :		
Roberto Amadio	Professeur	Université Paris 7
Gérard Boudol	Directeur de recherche	INRIA Sophia-Antipolis
Frédéric Boussinot	Maître de recherche	Ecole des Mines de Paris
Frédéric Loulergue	Professeur	Université d'Orléans

Table des matières

1 Introduction **9**
- 1.1 Programmation Réactive . . . 10
- 1.2 Contributions . . . 13
  - 1.2.1 Contrôle des ressources pour la réactivité . . . 13
  - 1.2.2 Programmation multi-cores . . . 13

I Contrôle des Ressources et Réactivité dans un π-calcul Synchrone **15**

2 Contrôle de la complexité **17**
- 2.1 Systèmes de réécriture et terminaison . . . 18
  - 2.1.1 Systèmes de réécriture . . . 18
  - 2.1.2 Terminaison . . . 18
- 2.2 Langage fonctionnel du premier ordre . . . 21
  - 2.2.1 Syntaxe . . . 21
  - 2.2.2 Typage . . . 21
  - 2.2.3 Sémantique . . . 22
  - 2.2.4 Terminaison par Size Change Principle . . . 22
- 2.3 Complexité des Programmes fonctionnels . . . 23
  - 2.3.1 Récursion bornée sur la notation. . . . 23
  - 2.3.2 Caractérisation syntaxique de PTIME . . . 23
  - 2.3.3 Mobile resource Guarantees . . . 24
  - 2.3.4 Quasi-interprétations . . . 25
- 2.4 Concurrence . . . 26
  - 2.4.1 Hume . . . 26
  - 2.4.2 Programmation réactive . . . 27

3 Le π-Calcul Synchrone **29**
- 3.1 Syntaxe . . . 30
- 3.2 Sémantique Intuitive . . . 31
- 3.3 Typage . . . 31
- 3.4 Opérateurs Dérivés . . . 32
- 3.5 Comparaison avec le π-calcul . . . 32
- 3.6 Exemples de programmes . . . 33
- 3.7 Sémantique Formelle . . . 34

4 Le $S\pi$-calcul à contrôle fini **37**
4.1 Contribution . . . 38
4.2 Réactivité . . . 39
4.3 Annotations . . . 41
4.3.1 Borne sur le nombre de threads . . . 41
4.3.2 Annotation des identificateurs de threads . . . 41
4.3.3 La condition read-once . . . 42
4.3.4 Annotations des paramètres . . . 44
4.3.5 Annotations de statut . . . 44
4.3.6 Annotations des signaux . . . 45
4.4 Analyse statique . . . 45
4.4.1 Inégalités pour la terminaison des instants . . . 45
4.4.2 Inégalités pour le contrôle de la taille des paramètres au début du cycle . . . 46
4.4.3 Inégalités pour le contrôle le taille des valeurs calculées dans un cycle . . . 47
4.4.4 Quasi-interprétations . . . 49

5 Sémantique annotée **53**
5.1 Sémantique . . . 53
5.1.1 Sémantique de réduction . . . 53
5.1.2 Inégalités et sémantique annotée . . . 56
5.2 Résultats . . . 57

6 Preuves **59**

II Programmation Multi-Cores Sûre **69**

7 Programmation des architectures Multi-Core **71**

8 PACT(PArtially Cooperative Threads) **75**
8.1 Définition du langage . . . 75
8.1.1 Valeurs . . . 76
8.1.2 Expressions et filtres . . . 77
8.1.3 Threads, programmes et sémantique informelle . . . 77
8.1.4 Déterminisme, ordonnancement et fairness . . . 79
8.1.5 Constructions dérivées . . . 80
8.1.6 Comparaison avec les FAIR THREADS . . . 80
8.2 Sémantique . . . 81
8.2.1 Définitions . . . 81
8.2.2 Relation de réduction . . . 83

9 Atomicité **93**
9.1 Atomicité en PACT . . . 94
9.2 Propriété et droit d'accés . . . 96
9.3 Analyse Statique . . . 98
9.4 Correction . . . 102

10 Preuves **105**
10.1 Preuve de la proposition 4 105
10.2 Preuve du lemme 2 105
10.3 Preuve du lemme 3 107
10.4 Preuve du théorème 2 108

11 Conclusion **109**

Remerciements

Je tiens avant tout à remercier les membres de ma famille, et plus particulièrement ma grand-mère, sans qui le parcours qui mène à la rédaction de ce manuscrit n'aurait pas été possible.

Merci à l'équipe enseignante de l'université Paris 12 qui a su m'initier aux plaisirs de l'informatique théorique. Merci plus particulièrement à Frédéric Loulergue dont les travaux ont motivés mon intérêt pour la recherche, et qui m'a fait le plaisir d'accepter d'être président du jury. Merci également à l'équipe enseignante du DEA Programmation de l'université Paris 7 pour l'année passionante passée à suivre leurs cours.

Merci aux membres de l'équipe MIMOSA de l'INRIA Sophia-Antipolis pour la qualité de l'ambiance de travail dont j'ai bénéficier au cours de ces quelques années. Merci tout particulièrement à Frédéric Boussinot pour avoir accepté de faire partie du jury et pour nos discussions passionantes dont certaines, qui avaient l'informatique comme sujet, ont beaucoup comptées dans la rédaction de ce manuscript.

Merci à Jean-Yves Marion et à Marc Pouzet d'avoir accepter de rapporter cette thèse et surtout d'avoir su le faire dans des délais aussi courts.

Merci enfin à Roberto Amadio et Gérard Boudol pour la qualité de leur encadrement et surtout pour leur patience. Je tiens tout particulièrement à remercier Gérard pour ses remarques toujours pertinentes et Roberto pour sa précieuse implication dans la première partie de ce manuscrit.

Chapitre 1

Introduction

La notion de *système réactif* [57] désigne des systèmes qui maintiennent une interaction permanente avec un environnement donné ; en particulier, un système réactif doit être en mesure de *réagir* aux sollicitations successives de son environnement. La famille des *langages synchrones* [50] regroupe un ensemble de langages de programmation dédiés à la conception de tels systèmes. De manière générale, ces langages permettent de décrire le comportement de composants parallèles qui s'exécutent de manière synchrone, relativement à une horloge logique sur laquelle repose un mécanisme de diffusion instantanée de l'information. Parmi les langages synchrones, on distingue principalement deux paradigmes de programmation : la programmation orientée *flots de données* (LUSTRE [51], SIGNAL[70],...) et la programmation orientée *contrôle*(Esterel[16],...). Dans tous les cas, ces langages sont conçus de manière à permettre un ordonnancement statique des composants parallèles et une compilation des programmes vers du code séquentiel, des automates à états finis ou des circuits. En particulier, sous certaines contraintes, on peut assurer que, pour toute sollicitation de l'environnement, la réaction du système existe (*réactivité*) et est unique (*déterminisme*). En contrepartie de l'obtention de propriétés aussi fortes, les contraintes imposées sur ces langages limitent leur utilisation à des domaines très spécifiques. Comme exemples de fonctionnalités non supportées par les langages synchrones, on peut citer : l'introduction dynamique de composants parallèles (e.g. par création de threads ou mobilité du code), la mobilité des noms (au sens des algèbres de processus), la définition de fonctions récursives et la manipulation de types de données dynamiques.

La *programmation réactive*[90] désigne un paradigme de programmation qui repose sur une relaxation des contraintes imposées dans les langages synchrones. Ce paradigme de programmation, inspiré plus particulièrement par le langage Esterel, propose un type de programmation concurrente plus générale ayant pour objectif la réalisation, par exemple, de contrôleurs "event-driven", d'interfaces graphiques, de simulations physiques, de services web et de jeux multi-joueurs (voir [90] et [74]). Plusieurs implantations de ce paradigme, pour différents langages, ont été proposées : Java [24], C[25, 26], Scheme [92] et Objective Caml[75]. En particulier, ces implantations permettent la création dynamique de comportements et la définition de fonctions récursives et de types de données complexes.

Le travail présenté ici s'intéresse à la notion de *logiciel sûr* dans le cadre de la programmation réactive. Intuitivement, un logiciel sûr (ou *logiciel de confiance*) peut être présenté comme dans [56] :

> Tout comme une "personne de confiance", il s'agit d'un logiciel que l'on *croit* : ses réactions vis-à-vis de l'environnement sont prévisibles, contrôlées ; non-dangereuses, les résultats de son utilisation sont garantis.

En pratique, dans le cadre d'un langage de programmation, cette notion de logiciel sûr suppose la

définition formelle de la sémantique du langage et des propriétés de sûreté souhaitées et, si nécessaire, la définition et la validation (pour un degré de confiance donné) des outils nécessaires pour garantir ces propriétés. De manière générale, on distingue deux approches complémentaires permettant de garantir la satisfaction des propriétés souhaitées. La *vérification dynamique*, d'une part, qui présente l'avantage d'une expressivité maximale. L'*analyse statique*, d'autre part, qui évite une surcharge de calcul à l'exécution et qui, de manière plus primordiale, permet une détection à priori des comportements jugés inacceptables. Sans négliger l'importance de la première approche, on se concentre ici sur la seconde pour traiter de deux propriétés rattachées à la programmation réactive. Premièrement, on s'intéresse au problème du contrôle statique des ressources nécessaires à l'exécution des programmes. Contrairement aux langages synchrones et du fait de l'introduction de caractéristiques dynamiques, les langages réactifs (relatifs à la programmation réactive) n'offrent aucune garantie en la matière. On se propose de définir une notion de réactivité, et plus précisément de *réactivité efficace*, dans le cadre d'une algèbre de processus synchrone, le $S\pi$-calcul[6], qui capture les caractéristiques essentielles de la programmation réactive. Deuxièmement, on s'intéresse à l'utilisation de la programmation réactive, et en particulier d'un modèle à base de threads coopératifs, pour le développement d'applications adaptées aux architectures multicores. Afin de mieux situer les contributions de ce document (sections 1.2), on commence par présenter succinctement les origines et les concepts de la programmation réactive (section 1.1).

1.1 Programmation Réactive

Dans le paradigme synchrone[14, 15], l'exécution d'un programme repose sur le calcul d'*instants* successifs. Chaque instant est défini comme une *réaction* du système à une sollicitation de l'environnement ou, plus précisément, par un calcul des *signaux de sortie* et du prochain état du système à partir des *signaux d'entrée* et de l'état courant du système. A un niveau plus abstrait, l'*hypothèse synchrone* suppose qu'une réaction ne prend pas de temps ; les entrées et les sorties du système sont simultanées (synchrones). En particulier, "au cours" de l'instant, les temps de calcul et de communication sont négligés et le programme n'interagit pas avec son environnement. L'instant représente une *unité de temps logique*, à laquelle est associée, de manière cohérente pour tous les composants du système, un état des signaux (présence ou absence). A chaque instant, un signal est absent par défaut ; il est présent s'il est donné par l'environnement ("au début" de l'instant) ou émis par le programme ("au cours" de l'instant). Comme nous l'avons déjà noté, les différents langages de programmation basés sur le paradigme synchrone peuvent être regroupés en deux familles. Dans les langages de programmation orientés *flots de données*, chaque réaction repose sur la résolution d'équations décrivant les signaux de sortie en fonction des signaux d'entrée et des valeurs de l'instant précédent. Dans les langages de programmation orientés *flots de contrôle*, chaque réaction repose sur l'exécution d'instructions qui permettent de manipuler l'état du système et de calculer les signaux de sortie en fonction de cet état et des signaux d'entrée.

La programmation réactive est dérivée de la classe des langages orientés flots de contrôle et en particulier du langage Esterel que nous survolons maintenant. Bien que les implantations du langage Esterel reposent sur des langage hôtes (principalement, le langage C), les propriétés de réactivité et de déterminisme ne concernent que le noyau du langage ; c'est plus particulièrement à ce dernier que nous nous intéressons. Le langage Esterel propose un style de programmation impérative ; les instructions de base (test, boucle, émission, suspension, préemption,...) sont combinées par compositions séquentielle et parallèle synchrone. Les *threads* du système communiquent par *diffusion instantanée* de signaux, éventuellement associés à des valeurs. Ces valeurs sont typiquement des types de données

de taille bornée (booléens, entiers, flottants,...). Comme dans le cas des autres langages synchrones, sous certaines conditions, l'utilisation de techniques d'exécution symbolique permet une compilation des programmes, et en particulier du parallélisme, vers des équations, des automates ou des circuits. Intuitivement, en Esterel, calculer une réaction revient à déterminer une valuation des signaux (présents ou absents) cohérente avec le programme à exécuter. Plus précisément, une telle valuation réalise des hypothèses de présence ou d'absence des signaux devant respecter la *loi de cohérence*, qui stipule qu'un signal est présent si et seulement si il est émis par le programme (ou donné par l'environnement). Un programme correct doit admettre, pour chaque réaction possible du système, une et une seule valuation des signaux (propriétés de réactivité et de déterminisme). Pour illustrer ce propos, on donne ci-dessous deux exemples de programmes incorrects ; le premier n'admet aucune valuation des signaux alors que le second en admet deux. Considérons le programme Esterel suivant :

$$\texttt{present}\ S\ \texttt{then nothing else emit}\ S\ \texttt{end} \tag{1.1}$$

Ce programme teste la présence du signal S, ne fait rien si S est présent et émet S si S est absent. Intuitivement, en supposant que la définition de S est locale, il n'est pas possible d'associer un état (présent ou absent) à S. En effet, si S est supposé présent, alors S n'est pas émis et on a une contradiction. Si S est supposé absent alors il est émis et on a à nouveau une contradiction. L'exécution du programme n'est pas possible ; celui-ci n'est donc pas réactif. Maintenant, considérons le programme suivant :

$$\texttt{present}\ S\ \texttt{then emit}\ S\ \texttt{else nothing end} \tag{1.2}$$

Ce programme teste la présence du signal S, émet S si S est présent et ne fait rien si S est absent. Intuitivement, en supposant que la définition de S est locale, il est possible d'associer n'importe quel état (présent ou absent) à S. En effet, si S est supposé présent, alors S est émis, Si S est considéré comme absent alors il n'est pas émis. Deux exécutions du programme sont possibles ; celui-ci n'est donc pas déterministe. La sémantique constructive[16] d'Esterel se base sur une approche plus intuitive que celle de la loi de cohérence. Intuitivement, la sémantique constructive rejette toute justification d'une hypothèse par un élément dépendant de cette hypothèse. Par exemple, dans le cas du programme (1.1), la vérification de l'hypothèse de présence de S par l'émission présente dans le programme dépend de la présence de S. Dans la terminologie d'Esterel, on appelle ce type de problème un cycle de causalité[49]. Quelque soit la sémantique considérée, une analyse statique est nécessaire pour rejeter ces cycles de causalité et celle-ci est, en partie, responsable des limitations du langages. En particulier, l'analyse permettant le rejet des cycles de causalité en Esterel n'est pas stable par composition parallèle. De manière générale, ces problèmes de causalité rendent difficile l'utilisation d'Esterel en tant que langage de programmation généraliste.

Le langage SL[27] (Synchronous Language) reprends , à une différence sémantique près, les caractéristiques du langage Esterel. En SL, les hypothèses d'absence et de présence des signaux sont rejetées au profit d'une réaction retardée à l'absence. Concrètement, cela veut dire que, lorsqu'un signal est absent, l'évaluation de la branche *else* d'une construction de test de présence est *retardée* à l'instant suivant le test. Cette modification a pour effet d'éliminer, par définition de la sémantique, les cycles de causalité. En particulier, en SL, le programme (1.1) est cohérent : si le signal S est absent alors il est émis *à l'instant suivant*. De la même manière, le programme (1.2) est déterministe : sans hypothèse de présence, le signal S ne peut être considéré que comme absent, la branche *else* est choisie et S n'est pas émis.

La réaction retardée à l'absence proposée par SL permet d'envisager l'utilisation du style de programmation introduit par les langages synchrones dans le cadre de langages de programmation concurrente plus généraux. Dans [7, 4], les auteurs proposent un modèle de programmation, appelons

le SLR, qui revisite le langage SL en introduisant, en particulier, la création dynamique de comportements.

Contrairement à Esterel, ni SL ni SLR ne considèrent la question des signaux portant des valeurs (on dit dans ce cas que les signaux sont purs). En Esterel, les signaux peuvent *porter* des valeurs, i.e. l'information associée à un signal au cours de l'instant n'est plus seulement un booléen dénotant la présence ou l'absence du signal mais une valeur. La valeur d'un signal, pour un instant donné, dépend des valeurs associée à (émises sur) ce signal et peut être lue *au cours* de cet instant. Si plusieurs valeurs sont émises sur un signal au cours du même instant, la valeur associée au signal pour cet instant dépend du choix d'une fonction de combinaison qui doit être associative et commutative pour conserver le déterminisme. Pour déterminer la valeur d'un signal au cours de l'instant on retrouve les problèmes de cohérence liés aux statuts des signaux. Les programmes suivants, où S est un signal portant des entiers et $?S$ dénote la valeur de S dans l'instant, sont rejetés :

$$(a) \quad \texttt{emit}\ S\ (?S) \qquad\qquad (b) \quad \texttt{emit}\ S\ (?S + 1) \tag{1.3}$$

En effet, supposons que, pour ces deux programmes, la définition du signal S soit locale. Dans les deux cas, la valeur lue doit être identique à la valeur émise. Dans le programme (a), la valeur du signal S doit satisfaire l'équation $Valeur(S) = Valeur(S)$ qui admet une infinité de solutions (le programme n'est pas déterministe). Dans le programme (b) la valeur du signal S doit satisfaire l'équation $Valeur(S) = Valeur(S) + 1$ qui n'admet aucune solution (le programme n'est pas réactif).

Bien que les signaux ne portent pas de valeur en SL et SLR, il serait légitime de supposer ,du fait de la réaction retardée à l'absence, que la valeur d'un signal pour un instant donné ne peut être déterminée qu'à la fin de l'instant et n'est donc disponible qu'à l'instant suivant (ce qui n'empêche pas de réagir dans l'instant à la présence d'un signal). En effet, déterminer la valeur d'un signal suppose de disposer de toutes les valeurs émises et donc de savoir que le signal ne sera plus émis au cours de l'instant ; ce qui est une forme de réaction à l'absence. Le $S\pi$-calcul[6], une algèbre de processus synchrone, peut être vu comme une extension de SLR dans laquelle les signaux portent des valeurs de types inductifs à la ML. Cette algèbre de processus sera présentée en détail dans la partie I ; nous verrons, en particulier, qu'une information partielle sur la valeur d'un signal peut être utilisée au cours de l'instant.

Plusieurs paradigmes de programmation ont été inspirés par le modèle de SL[22, 29, 28, 2] dont certains proposent une approche graphique de la programmation[23, 32]. Plusieurs implantations, que nous regrouperons sous le nom de FAIR THREADS, reposent sur la notion de threads coopératifs [24, 92, 26, 75, 73]. Dans le paradigme coopératif, par opposition au paradigme préemptif, un thread conserve le contrôle jusqu'à ce qu'il le rende explicitement. Ce type d'ordonnancement facilite grandement la programmation puisque la protection des données est assurée par construction entre deux points de coopération. En contrepartie, si un thread ne coopère pas, les autres threads risquent d'être bloqués. Les différentes implantations des FAIR THREADS se distinguent principalement par les solutions choisies pour traiter certaines opérations ad hoc qui peuvent soulever ce problème, comme la gestion des entrées et sorties bloquantes. Dans L'implantation des FAIR THREADS en Scheme[92], des *services asynchrones* permettent de réaliser ce type d'opérations. Par exemple, pour lire une socket, un thread fait appel au service asynchrone associé, se *suspend* et reprend la main lors d'une coopération (d'un autre thread) intervenant après la réalisation du service. Dans L'implantation des FAIR THREADS en C[26], plusieurs ordonnanceurs coopératifs (ou zones synchrones) peuvent être définis et les threads ont la possibilité de passer dynamiquement de l'un à l'autre. L'ordonnancement de ces zones synchrones, les unes par rapport aux autres est préemptif alors que l'ordonnancement des threads attachés à une même zone synchrone est coopératif. Un thread peut également se *détacher* temporairement de

l'ordonnanceur coopératif afin d'être géré par un thread "système" ; l'exécution des autres threads se poursuit sans ce dernier jusqu'à son *retour*. Cette notion de détachement peut être vue comme une généralisation des services asynchrones dont le comportement peut alors être défini par l'utilisateur.

Finalement, le langage ULM[21], inspiré par les FAIR THREADS, montre que le passage à l'échelle distribuée est possible. ULM introduit pour les threads la possibilité de migrer au sein d'un réseau (asynchrone) de sites distants. L'originalité de ce langage est de se baser sur les notions d'instant et de présence et d'absence de la programmation réactive pour prendre en compte le caractère non permanent des ressources accessibles au travers d'un réseau. En particulier, ce caractère non permanent des ressources est attribué aux références. Par exemple, supposons qu'un thread crée une référence et communique celle-ci à un autre thread avant que ce dernier ne migre vers un autre site. Si le second thread, une fois arrivé sur le nouveau site, tente d'accéder à cette référence il sera suspendu. Il pourra alors, par exemple, réagir à l'absence de cette référence et adapter son comportement. A noter que cette particularité du langage suppose une vérification dynamique de la présence des références lors des accès ; dans [21], un système de type permet cependant de détecter statiquement un sous ensemble des accès réalisés par un programme pour lesquels aucun test n'est nécessaire (e.g. la lecture par un thread d'une référence créée par un autre thread lorsqu'aucun des deux ne peut migrer). Une implantation d'une version non typée du langage ULM en SCHEME est présentée dans [43].

1.2 Contributions

Ce document est structuré en deux parties correspondant à deux propriétés que l'on cherche à associer aux langages réactifs. La première de ces propriétés, à savoir la réactivité effective des programmes, concerne l'ensemble des langages réactifs. La seconde, appelée atomicité, porte plus précisément sur l'utilisation des FAIR THREADS dans le cadre de la programmation d'architectures multi-cores.

1.2.1 Contrôle des ressources pour la réactivité

Dans les langages réactifs, on suppose que les programmes sont effectivement réactifs ; i.e., pour chaque activation provenant de l'environnement, on suppose que le calcul de l'instant termine toujours en produisant une sortie. Contrairement au cas des langages synchrones, cette propriété n'est pas garantie et plusieurs caractéristiques des programmes peuvent corrompre la réactivité d'un système. Par exemple, une erreur d'exécution telle qu'un *core dumps* compromet la réactivité d'un programme. De ce point de vue, le choix d'une approche typée, comme dans ReactiveML[75], représente une première étape. Une autre caractéristique essentielle d'un programme réactif est l'utilisation raisonnable (d'un point de vue quantitatif) des ressources (unités de calcul et mémoire). Dans la première partie, nous étudions la propriété de *réactivité* des programmes dans le cadre du $S\pi$-calcul. Plus précisément, nous proposons une définition formelle de la notion de *réactivité* (et plus précisément de réactivité *efficace*) adaptée à ce modèle et des outils d'analyse statique permettant de vérifier qu'un programme satisfait cette propriété.

1.2.2 Programmation multi-cores

Un des avantages du choix d'un ordonnancement purement coopératif dans le cadre de la programmation à base de threads est que la protection des données est assurée par construction. Entre deux points de coopération, un thread est assuré d'être le seul à accéder aux données. Malheureusement,

une approche purement coopérative ne permet pas de tirer parti du parallélisme des nouvelles architectures à base de processeurs multi-cores. Les FAIR THREADS en C, qui mélangent les styles coopératifs et préemptifs, permettent de dépasser cette limitation. Dans [26], les ordonnanceurs (ou zones synchrones) sont présentés de la manière suivante :

> A synchronized area can, quite naturally, be defined to manage some shared data that has to be accessed by several threads. In order to get access to the data, a thread first has to link to the area, and then it becomes scheduled by the area and can thus get safe access to the data. Indeed, as the scheduling is cooperative, there is no risk to the thread of being preempted during an access to the data.[...]
> Data shared by unlinked threads have to be protected by locks in the standard way.

Malheureusement, cette approche ne permet pas de conserver les propriétés d'atomicité de la partie purement coopérative. Les données accédées par des threads détachés doivent être protégées et on suppose implicitement que les threads attachés à des zones synchrones différentes n'accèdent pas aux mêmes données. En particulier, aucune garantie n'est offerte sur la protection effective des données et donc sur l'atomicité (au moins à un niveau logique) du code exécuté par un thread entre deux points de coopération. Dans la seconde partie de ce document nous revisitons le style de programmation des FAIR THREADS afin : (1) de définir formellement la sémantique du parallélisme pour ce modèle (seule la partie purement coopérative des FAIR THREADS est considérée dans [25]) et (2) de proposer une notion d'atomicité adaptée au modèle et garantie par analyse statique. On se basera pour cela sur une extension minimale du $S\pi$-calcul, appelée PACT.

Première partie

Contrôle des Ressources et Réactivité dans un π-calcul Synchrone

Chapitre 2

Contrôle de la complexité

Le contrôle (quantitatif) des ressources (CPU/mémoire) nécessaires à l'exécution d'un programme est une propriété essentielle du point de vue de la sûreté du logiciel. Dans certains domaines comme les systèmes embarqués ou le code mobile, et de manière générale dans les systèmes réactifs, le système doit disposer de ressources suffisantes pour remplir sa tâche ; de même, l'arrivée d'un nouveau composant ne doit pas compromettre le reste du système en consommant trop de ressources.

Pour obtenir de telles garanties, une première approche consiste à vérifier *statiquement* (i.e. à la compilation) que l'exécution d'un programme pourra se faire pour une quantité de ressources donnée. Dans le cadre du code mobile, par exemple, cette approche permet d'envisager une approche proof-carrying code[85], c'est à dire la vérification, lors du chargement du code, d'un certificat obtenu à la compilation ; ce qui permet au système hôte de s'assurer que le code en question respecte une politique de sécurité donnée. Une autre approche consiste à vérifier *dynamiquement* (i.e. au cours de l'exécution) qu'un programme ne consomme pas plus d'une quantité donnée de ressources, par exemple en mettant en place des moniteurs d'exécution. La seconde approche présente l'avantage d'offrir plus de liberté quand à l'écriture des programmes (la première relève toujours d'un compromis entre contrôle et expressivité). En revanche, celle-ci ne permet pas de détecter à priori les programmes fautifs et peut se révéler limitée dans le cadre des systèmes embarqués, par exemple, pour lesquels la modification du code est difficile.

Dans les prochains chapitres, nous nous baserons sur la première approche (contrôle statique) pour nous intéresser au problème de la réactivité des programmes dans le cadre d'une algèbre de processus synchrones inspirée par le paradigme réactif. En particulier, nous serons amenés à borner statiquement la taille des systèmes et à garantir la terminaison des instants. L'algèbre de processus en question repose sur un sous langage fonctionnel du premier ordre. Les programmes de ce sous-langage peuvent être représentés par (un sous-ensemble) des systèmes de réécriture pour lesquels le problème de la terminaison a fait l'objet de nombreuses études. Dans ce chapitre, nous rappelons les bases des systèmes de réécriture ; en particulier, nous présentons certaines méthodes de preuve de terminaison proches des techniques qui seront utilisées par la suite (2.1). Nous rappelons également un certain nombre de concepts et de résultats permettant, dans le cadre d'un langage fonctionnel du premier ordre, de contrôler la complexité des programmes (section 2.3). Finalement, nous profitons de ce chapitre pour rappeler des résultats antérieurs(2.4).

2.1 Systèmes de réécriture et terminaison

On rappelle succinctement quelques éléments de base des systèmes de réécriture ainsi que quelques résultats de terminaison. Pour une étude poussée des systèmes de réécriture le lecteur pourra se référer à [40, 68, 12].

2.1.1 Systèmes de réécriture

Un système de réécriture définit un ensemble de termes et une relation de réécriture sur ces termes. Ces notions sont définies formellement ci-dessous.

Définition 1. *Un* système de réécriture *est la donnée d'un triplet* (X, S, R) *où :*

- X *est un ensemble de variables,*
- S *est un ensemble de symboles,*
- R *est un ensemble de règles de réécritures.*

L'ensemble $T(X, S)$ *des* termes *est le plus petit ensemble tel que* $X \subseteq T(X, S)$ *et, si h est un symbole,* $h(t_1, \ldots, t_n) \in T(X, S)$ *dès que* $h \in S$ *et* $t_1, \ldots, t_n \in T(X, S)$.
Chaque règle de réécriture *est une paire* $(h(t_1, \ldots, t_n), t)$, *que l'on notera* $h(t_1, \ldots, t_n) = t$, *où h est un symbole et* $t, t_1, \ldots, t_n$ *sont des termes et tels que les variables du membre droit apparaîssent dans le membre gauche de l'équation.*
Les règles de réécriture R d'un système de réécriture (X, S, R) *induisent une La* relation de réécriture *induite par R sur* $T(X, S) \times T(X, S)$ *, notée* $\rightarrow_R$ *ou plus simplement* $\rightarrow$, *est définie par les règles suivantes :*

$$\frac{t \rightarrow t'}{b(\ldots, t, \ldots) \rightarrow b(\ldots, t', \ldots)} \qquad \frac{b(t_1, \ldots, t_n) = t \in R}{b(t_1, \ldots, t_n) \rightarrow t} \qquad \frac{t \rightarrow t' \qquad \sigma : X \mapsto T(X, S)}{\sigma t \rightarrow \sigma t'}$$

où σ *est une substitution. La seconde règle dénote l'application directe d'une règle de réécriture alors que les deux autres dénotent la stabilité par* contexte *et par* substitution *d'un système de réécriture. Une* chaîne de réduction *est une suite finie ou infinie de termes* $t_1, t_2, \ldots$, *telle que* $t_1 \rightarrow t_2$, $t_2 \rightarrow t_3$,...

Exemple 1. *A titre d'exemple, considérons le système de réécriture* $(\{x\}, \{f, g, a\}, \{f(g(x)) = x, g(x) = g(f(g(x)))\})$. *Un exemple de chaîne de réduction est :*

$$f(g(a)) \rightarrow f(g(f(g(a)))) \rightarrow f(g(a)) \rightarrow a$$

La première réduction utilise la seconde équation et la stabilité par contexte et les deux autres réductions utilisent la première équation.

2.1.2 Terminaison

Définition 2. *Un système de réécriture termine s'il n'admet aucune chaîne de réduction infinie.*

Par exemple, le programme donné précédemment ne termine pas car on peut construire la chaîne infinie :

$$f(g(a)) \rightarrow f(g(f(g(a)))) \rightarrow f(g(f(g(f(g(a)))))) \rightarrow \ldots$$

Décider si un système de réécriture termine ou non est un problème indécidable ; pour s'en convaincre on pourra se référer au codage des machines de Turing par des systèmes de réécriture donné dans [12]. Cependant, plusieurs méthodes, capturant des classes de programmes plus ou moins grandes,

permettent de prouver la terminaison d'un système de réécriture. L'idée de base pour prouver la terminaison d'un système de réécriture est d'exhiber un ordre $>$ bien fondé (i.e. sans chaîne infinie décroissante) sur les termes tels que pour tout termes s, t l'inégalité $s > t$ est vérifiée dès que $s \to t$. En pratique, on ne cherche pas à vérifier directement que cette inégalité est satisfaite pour toute réduction ; la notion d'ordre de réduction permet de limiter l'étude de la terminaison d'un système à une analyse de ses règles de réécriture.

Définition 3. *Une relation d'ordre stricte $<$ sur les termes est un ordre de réduction si elle est*

1. *stable par contexte, i.e., si $t < t'$ alors $f(\ldots, t, \ldots) < f(\ldots, t', \ldots)$ pour tout symbole f,*
2. *stable par substitution, i.e., si $t < t'$ alors $\sigma t < \sigma t'$ pour toute substitution σ des variables vers les valeurs,*
3. *bien fondée*

Propriété. *Un système de réécriture termine si et seulement si il existe un ordre de réduction $>$ tel que $l > r$ pour toute règle de réécriture $l \to r$ du système.*

Pour prouver la terminaison d'un système de réécriture, il est parfois plus pratique de "plonger" les termes dans une autre structure ; cette méthode s'appelle l'*interprétation*. Un exemple classique d'interprétation est l'interprétation polynomiale[69] qui consiste à interpréter les symboles par des polynômes.

Définition 4. *Étant donné un système de réécriture, une interprétation polynômiale i est une fonction qui associe à chaque symbole h d'arité n un polynôme $i_h : \mathbb{N}^n \to \mathbb{N}$ tel que*

1. *$i_h(x_1, \ldots, x_n) \geq x_i$, pour $i = 1, \ldots, n$,*
2. *$a_i \geq b_j$ pour $j = 1, \ldots, n$ implique $i_h(a_1, \ldots, a_n) \geq i_h(b_1, \ldots, b_n)$,*
3. *si h est un constructeur d'arité 0 alors i_h est une constante positive.*
4. *pour toute règle $f(t_1, \ldots, t_n) = t$ on a $i_{\sigma h(t_1,\ldots,t_n)} > i_{\sigma t}$ pour toute substitution close σ.*

où i est étendue à l'ensemble des termes par i_x et $i_{h(a_1,\ldots,a_n)} = i_h(i_{a_1}, \ldots, i_{a_n})$.

Du point de vue du contrôle des ressources nécessaire à l'exécution d'un programme, l'interprétation polynômiale permet non seulement de prouver la terminaison d'un système de réécriture mais offre également, sous certaines conditions, une garantie sur le nombre d'étapes de réduction possibles.

Propriété. *Si i est une interprétation polynômiale d'un système de réécriture, alors l'ordre $<_i$ sur les termes clos défini par $t <_i s$ si et seulement si $i_t < i_s$ est un ordre de réduction. L'ensemble des systèmes de réécriture* confluents *admettant une interprétation polynômiale pour des polynômes de la forme $P(x_1, \ldots, x_n) = \Sigma_{i=1,\ldots,n} x_i + d$, où d est une constante positive, caractérisent la classe PTIME [18].*

Remarque 1. *On renvoie à [18] pour la caractérisation d'autres classes de complexité, fonctions de la forme des polynômes et de la confluence du système.*

De manière générale, prouver qu'un ordre sur les termes est bien fondé peut être une tâche difficile. La notion d'ordre de simplification remplace cette condition par une condition plus forte (condition 3 dans la définition ci-dessous), plus facile à déterminer.

Définition 5. *Un ordre strict $<$ sur les termes est un* ordre de simplification *si et seulement si*

1. *$<$ est stable par contexte, i.e., si $t < t'$ alors $f(\ldots, t, \ldots) < f(\ldots, t', \ldots)$.*

2. *$<$ est stable par substitution, i.e., si $t < t'$ alors $\sigma t < \sigma t'$ pour toute substitution σ*
3. *pour tout termes s, t tels que t est un sous terme strict de s on a $s > t$*

Propriété. *Un ordre de simplification est un ordre de réduction.*

Les ordres récursifs sur les chemins sont des exemples classiques d'ordres de simplification [41]. Ces ordres sont particulièrement intéressants du fait qu'il est possible de caractériser la classe des fonctions qui s'expriment par des programmes dont la terminaison peut être prouvée grâce à ces derniers. Afin de présenter ces ordres, on commence par rappeler les notions d'ordres lexicographique et multi-ensemble.

Définition 6. *Soit $<$ un ordre strict sur un ensemble A. L'ordre* lexicographique *$<_{lex}$ induit par $<$ sur les listes d'éléments de A est défini par $a_1, \ldots, a_n <_{lex} b_1, \ldots, b_m$ si et seulement si l'une des conditions suivantes est satisfaite :*

- $a_1 < b_1$
- $a_1 = b_2$ *et* $(a_2, \ldots, a_n) <_{lex} (b_2, \ldots, b_m)$

Propriété. *Si $<$ est un ordre strict bien fondé sur A et n un entier. La relation $<_{lex}$ est un ordre bien fondée sur les listes d'éléments de A de longueur inférieure ou égale à n.*

Définition 7. *Soit $<$ un ordre strict sur un ensemble A. L'ordre multi-ensemble $<_{mset}$ induit sur A par $<$ est la clôture transitive de la relation $<^1_{mset}$ définie par*

$$\mathcal{M} \uplus \{\!\{y_1, \ldots, y_n\}\!\} <^1_{mset} \mathcal{M} \uplus \{\!\{x\}\!\}$$

où $n \geq 0$ et pour $i = 1, \ldots, n$ on a $y_i < x$.

Propriété. *Si $<$ est un ordre strict bien fondé sur A. La relation $<_{mset}$ est un ordre bien fondé sur les multi-ensembles d'éléments de A.*

On peut maintenant présenter les ordres récursifs sur les chemins associés aux ordres lexicographique et multi-ensemble et présenter les résultats associés. Pour celà, on suppose donné un pré-ordre $<_{\mathcal{F}}$ sur les symboles.

Définition 8. *L'ordre MPO (Multiset Path Ordering) $<_{mpo}$ est la plus petite relation telle que $t = f(t_1, \ldots, t_n) <^{mpo} g(s_1, \ldots, s_m) = s$ si l'une des conditions suivantes est réalisée :*

- *il existe j tel que $t \leq^{mpo} s_j$*
- *$f <_{\mathcal{F}} g$ et $t_i <^{mpo} s$ pour tout i*
- *$f =_{\mathcal{F}} g$ et $\{\!\{t_1, \ldots, t_n\}\!\} <^{mpo}_{mset} \{\!\{s_1, \ldots, s_m\}\!\}$*

Propriété. *MPO est un ordre de simplification([39]). La classe des fonctions calculables par un système de réécriture terminant par MPO est exactement la classe des fonctions primitives récursives [36, 60].*

Définition 9. *L'ordre LPO (Lexicographic Path Ordering) $<^{lpo}$ est la plus petite relation telle que $t = f(t_1, \ldots, t_n) <^{mpo} g(s_1, \ldots, s_m) = s$ si l'une des conditions suivantes est réalisée :*

- *il existe j tel que $t \leq^{lpo} s_j$*
- *$f <_{\mathcal{F}} g$ et $t_i <^{lpo} s$ pour tout i*
- *$f =_{\mathcal{F}} g$, $(t_1, \ldots, t_n) <^{lpo}_{lex} (s_1, \ldots, s_m)$ et $t_i \leq^{lpo} s$ pour tout i.*

Propriété. *LPO est un ordre de simplification([67]). La classe des fonctions calculables par un système de réécriture terminant par LPO est exactement la classe des fonctions multiples récursives [96].*

Nous nous somme contenté ici d'esquisser des méthodes de terminaison proches de ce que l'on proposera par la suite. La littérature sur le problème de la terminaison des systèmes de réécriture est très vaste et il serait impossible d'en faire ici le détail. Comme nous venons de le voir, certaines méthodes de preuve de terminaison des systèmes de réécriture permettent de caractériser des classes de fonctions importantes. Cependant, ces classes de fonctions sont assez éloignées de l'idée que l'on peut se faire de fonctions à la complexité raisonnable. La classe des fonctions calculables en temps polynômiale est généralement acceptée comme une classe de fonction raisonnable. Nous verrons dans les sections suivantes que certaines méthodes permettent de capturer cette classe.

2.2 Langage fonctionnel du premier ordre

Dans ce manuscrit, on considérera à plusieurs reprises un langage fonctionnel du premier ordre simplement typé, définissant des fonctions sur des types de données définies inductivement et reposant sur une stratégie d'appel par valeur. Comme nous l'avons déjà mentionné, les programmes de ce type de langages peuvent être représentés par des systèmes de réécriture. Pour fixer les idées, on présente ici un tel langage. Par la suite, on supposera donnée une définition axiomatique des fonctions du sous-langage considéré mais le langage présenté ici est celui que l'on aura à l'esprit.

2.2.1 Syntaxe

Les valeurs du langage considéré sont des valeurs de types de données inductif donné par des déclarations à la ML[34] de la forme :

$$\texttt{type}\ t\ =\ \ldots \mid \texttt{c of} t_1 * \ldots * t_n) \mid \ldots$$

où $\texttt{c}$ est appelé un constructeur et $t_1, \ldots, t_n$ pour $n \geq 0$ sont des types. Par exemple, le type *nat* représentant les entiers naturels est défini par $\texttt{type}\ nat\ =\ \texttt{z} \mid \texttt{s of}\ nat$. Le type $list(nat)$ des listes d'entiers est défini par $\texttt{type}\ nat\ =\ \texttt{nil} \mid \texttt{cons of}\ nat * list(nat)$. Les fonctions d'un programme sont définies par filtrage. Un filtre p est soit une variable soit un terme $\texttt{c}(p_1, \ldots, p_m)$ où $\texttt{c}$ est un constructeur $m \geq 0$ et $p_1, \ldots, p_m$ sont des filtres. Une expression e est soit une variable soit un terme $h(e_1, \ldots, e_n)$ où h est soit un symbole de fonction soit un constructeur et $e_1, \ldots, e_n$ sont des expressions. Finalement, un programme est la donnée

- d'un ensemble de déclarations de types de données,
- d'un ensemble d'équations de la forme $f(p_1, \ldots, p_n) = e$ où $p_1, \ldots, p_n$ sont des filtres et e est une expression dont les variables sont dans $p_1, \ldots, p_n$.

2.2.2 Typage

Pour chaque symbole de fonction f, on suppose donné un type $\tau_1, \ldots, \tau_n \rightarrow \tau$ où $\tau_1, \ldots, \tau_n$ dénotent les types des paramètres de f et τ dénote le type de la valeur calculée par f. On supposera toujours, par la suite, que l'on considère des programmes bien typés. On supposera également que pour tout identificateur f et pour toute équation $f(p_1, \ldots, p_n) = e$:

- chaque variable apparaît au plus une fois dans $p_1, \ldots, p_n$,
- si $f(v_1, \ldots, v_n)$ est bien typé alors il existe une et une seule règle $f(p_1, \ldots, p_n)$ telle que $\sigma t_i = v_i$ pour une certaine substitution σ des variables vers les valeurs.

2.2.3 Sémantique

L'évaluation d'un programme est le résultat de la réduction d'un terme de la forme $f(v_1, \ldots, v_n)$ (l'appel initial) par les règles données ci dessous. Ces règles définissent une sémantique de réduction (avec évaluation gauche-droite et appel par valeur).

$$\frac{e_i \Downarrow v_i,\ i = 1, \ldots, n}{\mathtt{c}(e_1, \ldots, e_n) \Downarrow c(v_1, \ldots, v_n)}$$

$$\frac{e_i \Downarrow v_i,\ i = 1, \ldots, n \qquad f(p_1, \ldots, p_n) = e \qquad \sigma p_i = v_i,\ i = 1, \ldots, n \qquad \sigma e \Downarrow v}{\mathtt{f}(e_1, \ldots, e_n) \Downarrow \sigma e}$$

où σ une substitution des variables vers les valeurs.

Exemple 2. *Dans cet exemple, on considère les fonctions add et mult qui définissent respectivement l'addition et la multiplication sur le type nat :*

$$\begin{array}{lcl} add(\mathtt{z}, y) & = & y \\ add(\mathtt{s}(x), y) & = & \mathtt{s}(add(x, y)) \end{array}$$

$$\begin{array}{lcl} mult(\mathtt{z}, y) & = & \mathtt{z} \\ mult(\mathtt{s}(x), y) & = & add(y, mult(x, y)) \end{array}$$

Remarque 2. *Il existe une correspondance entre tout programme du langage considéré ici et un système de réécriture ; en particulier, la terminaison d'un tel programme peut être prouvée à l'aide des méthodes présentées dans la section précédente.*

2.2.4 Terminaison par Size Change Principle

Avant d'introduire dans la section suivante plusieurs résultats concernant le contrôle de la complexité des programmes, il nous semble intéressant de noter l'existence d'une méthode de preuve de terminaison appelée le *Size Change Principle*[71]. Cette méthode est plus souple que celle basée sur les ordres récursifs sur les chemins. En effet, ces dernièrs opèrent de proche en proche sur les appels récursifs. Au contraire, le Size Change Principle opère sur le graphe d'appel du programme. En particulier, celui-ci cherche à construire, pour chaque boucle du graphe, un chemin capturant une décroissance stricte. Par exemple, considérons un programme associé aux équations suivantes :

$$f(x) = g(x) \qquad g(\mathtt{s}(x)) = f(x) \qquad g(0) = 0$$

De manière évidente, le calcul de $f(n)$ termine pour tout élément n de *nat*. Cependant, les ordres récursifs sur les chemins ne permettent pas de prouver la terminaison car l'appel $g(x)$ dans l'équation $f(x) = g(x)$ ne met pas de décroissance en évidence. En revanche, le Size Change Principle permet de détecter une suite infinie décroissante du seul paramètre des fonctions f et g dans toute suite infinie d'appels. On anticipe sur les chapitres précédent pour noter que, bien que le Size Change Principle nous semble une approche très attractive, les techniques que nous utiliserons par la suite sont plus proches des ordres récursifs sur les chemins (bien qu'elles s'inspirent légérement du Size Change Principle). En effet, nous aurons besoin déduire, à partir d'une borne polynomiale sur la taille des valeurs calculées et d'une preuve de terminaison, une borne polynomiale sur la complexité en temps des programme. Dans le cas du Size Change Principle , l'existence d'un tel résultat similaire est une question ouverte.

2.3 Complexité des Programmes fonctionnels

2.3.1 Récursion bornée sur la notation.

Le premier exemple de caractérisation de PTIME est [37]. Dans cette approche, on considère une algèbre de fonctions C construite à partir d'un ensemble de fonctions initiales et close par composition et par récursion bornée sur la notation. Les fonctions initiales sont la fonction nulle, les projections, les fonctions successeur $s_0(x) = 2x$ et $s_1(x) = 2x + 1$ et une fonction d'amorce $x\#y = 2^{|x|.|y|}$ où $|x| = \lceil log_2(x + 1)\rceil$. La récursion bornée par la notation est définie par le schéma suivant :

$$\begin{array}{lcl} f(0, \mathbf{x}) & = & g(\mathbf{x}),\ g \in C \\ f(s_0(x), \mathbf{y}) & = & h(x, \mathbf{y}, f(x, \mathbf{y})),\ h \in C \\ f(s_1(x), \mathbf{y}) & = & h(x, \mathbf{y}, f(x, \mathbf{y})), \qquad f(x, \mathbf{y}) \leq k(x, \mathbf{y}), k, h \in C \end{array}$$

Propriété. *La classe C est exactement la classe PTIME des fonctions calculables en temps polynômial.*

Les caractéristiques principales de la classe C sont la présence d'une fonction d'amorce et le besoin de donner la fonction k (dans le schéma de récursion) pour borner la récursion. Du point de l'analyse statique de la complexité des programmes la seconde condition est évidemment problématique.

2.3.2 Caractérisation syntaxique de PTIME

Dans [13], les auteurs présentent une classe de programmes qui permet de caractériser PTIME sans les contraintes posées par la classe C. Cette approche constitue une caractérisation purement syntaxique de PTIME (ce résultat est à rapproché de celui de [72] où une approche similaire est suivie). On considère une algèbre de fonctions B construite à partir de fonctions initiales et close par composition *sûre* et par récursion *sûre*. Les fonctions initiales sont la fonction nulle, les projections et les fonctions successeurs s_0 et s_1 ; cette fois il n'y a pas de fonction d'amorce. Dans la signature des fonctions, on distingue deux types de paramètres : *normaux* et *sûrs*. Intuitivement, les paramètres subissant la récursion sont toujours normaux alors que le résultat partielle d'une récursion est toujours un paramètre sûr. La récursion sûre est définie par le schéma ci-dessous, où les paramètres à gauche (resp à droite) de ; sont les paramètres normaux (resp. sûrs).

$$\begin{array}{lcl} f(\varepsilon, \mathbf{x}; \mathbf{y}) & = & g(\mathbf{x}; \mathbf{y}),\ g \in B \\ f(s_0(x), \mathbf{x}; \mathbf{y}) & = & h_0(x, \mathbf{x}; \mathbf{y}, f(x, \mathbf{x}; \mathbf{y})),\ h_0 \in B \\ f(s_1(x), \mathbf{x}; \mathbf{y}) & = & h_1(x, \mathbf{x}; \mathbf{y}, f(x, \mathbf{x}; \mathbf{y})),\ h_1 \in B \end{array}$$

Le principe de base de la class B est que le résultat partiel d'une récursion ne doit pas lui-même être soumis à une récursion. Le schéma de composition assure simplement qu'un paramètre sûr n'est pas utilisé en position normale.

Exemple 3. *L'addition add et la multiplication mul peuvent être définies de la manière suivante :*

$$\begin{array}{lcl} add(0; \mathbf{y}) & = & y \\ add(succ(x); y) & = & succ(add(x; y)) \\ mul(0, y;) & = & 0 \\ mul(succ(x), y;) & = & add(y; mul(x, y;)) \end{array}$$

Propriété. *La classe B est exactement la classe PTIME des fonctions calculables en temps polynômial[13].*

On peut également noter l'utilisation d'u principe similaire associé aux ordres de terminaison dans [77]. Dans cette approche l'ordre récursif sur les chemins LPO est adapté à ce principe. L'odre obtenu, appelé LMPO, permet également de capturer PTIME tout en capturant une plus vaste gamme d'algorithmes.

Propriété. *L'ensemble des programmes terminant par LMPO capturent la classe PTIME[77].*

2.3.3 Mobile resource Guarantees

Le projet MRG (mobile ressource guarantees [65]) fournit des outils permettant de garantir que l'espace nécessaire à l'exécution d'un programme fonctionnel du premier ordre est linéaire en la taille de l'entrée du programme. Visant plus particulièrement les systèmes embarqués [11], cette approche s'inscrit dans le paradigme du *proof-carrying code* [85] en s'appuyant sur un langage de haut niveau appelé Camelot compilé vers un langage de bas niveau appelé Grail. Ces travaux reposent principalement sur une approche à base de systèmes de types [61, 62]. Plus précisément, les propriétés sur les ressources sont obtenues à l'aide d'un type ressource $\diamond$ et de règles de typage linéaires. Intuitivement, un objet de type ressource peut être vu comme une cellule mémoire. Un constructeur de type inductif *consomme* un objet de type ressource et les fonctions peuvent recevoir ces objets comme paramètre.

Exemple 4. *Considérons par exemple le programme suivant qui réalise le tri par insertion. La fonction insert est de type* $\diamond, nat, list(nat) \rightarrow list(nat)$ *et la fonction sort est de type* $list(nat) \rightarrow list(nat)$.

$$
\begin{array}{lcl}
insert(d, a, \texttt{nil}) & = & \texttt{cons}(d, a, \texttt{nil}) \\
insert(d, a, \texttt{cons}(d', b, t)) & = & if\ a \leq b\ then\ \texttt{cons}(d, a, \texttt{cons}(d', b, t)) \\
 & & else\ \texttt{cons}(d', b, insert(d, a, t)) \\
 & & \\
sort(\texttt{nil}) & = & \texttt{nil} \\
sort(\texttt{cons}(d, a, t)) & = & insert(d, a, sort(t))
\end{array}
$$

Intuitivement, le système de type assure qu'un élément de type ressource ne peut être utilisé qu'au plus une fois pour permettre l'application d'un constructeur. Le pattern matching permet de récupérer *un élément de type ressource. Dans cet exemple, la fonction insert reçoit un paramètre de type ressource. Les paramètres d, a et l sont respectivement un élément de type ressource, l'élément à insérer et la liste initiale. Comme on peut le voir dans la définition de insert, tant que la liste initiale est non vide, un élément* d' *de type ressource est récupéré et réutilisé pour rétablir la liste ; lorsque le nouvel élément doit être inséré, on utilise également d.*

Exemple 5. *L'exemple suivant ne peut pas être typé car leparamètre de type ressource est utilisé plus d'une fois.*

$$
\begin{array}{lcl}
twice(\texttt{nil}) & = & \texttt{nil} \\
twice(\texttt{cons}(d, x, l)) & = & \texttt{cons}(d, \texttt{tt}, \texttt{cons}(d, \texttt{tt}, twice(l))))
\end{array}
$$

Avec cette approche, les programmes sont non size increasing, ceux-ci peuvent être compilés vers du code, par exemple en *C*, n'utilisant pas d'allocation dynamique. La limite de cette approche est que l'on souhaiterait pouvoir inférer la présence des éléments de types ressources plutôt que de les donner explicitement.

2.3.4 Quasi-interprétations

On s'intéresse maintenant aux *quasi-interprétations* que nous utiliserons par la suite. Les *quasi-interprétations*[76, 20] permettent de borner la taille des valeurs calculées par les programmes décrits dans un langage fonctionnel du premier ordre.

Définition 10. *Une quasi-interprétation* polynômiale *associe à chaque symbole du programme (constructeur ou fonction) d'arité n une fonction* $q_h : (\mathbb{R}^+)^n \to \mathbb{R}^+$ *telle que*

- q_h *est bornée par un polynôme,*
- $q_h(x_1, \ldots, x_n) \geq x_i$ *pour* $i = 1, \ldots, n$,
- *si* $x_i \geq y_i$ *pour* $i = 1, \ldots, n$ *alors* $q_h(x_1, \ldots, x_n) \geq q_h(y_1, \ldots, y_n)$,
- *pour un constructeur c,* $q_c(x_1, \ldots, x_n) = \Sigma_{i=1}^{n} x_i + d$ *où d est une constante strictement positive.*

L'interprétation d'une expression est obtenue par extension : $q_x = x$ et $q_{h(e_1,\ldots,e_n)} = q_h(q_{e_1}, \ldots, q_{e_n})$.

Définition 11. *On dit que le programme admet une quasi-interprétation q si pour toute équation* $f(p_1, \ldots, p_n) = e$ *on a* $q_{f(e_1,\ldots,e_n)} \geq q_e$.

Propriété. *Si un programme admet une quasi-interprétation, alors pour tout symbole de fonction* f *du programme et pour toutes valeurs* $v_1, \ldots, v_n$, *la taille des valeurs calculées lors de l'évaluation de* $f(v_1, \ldots, v_n)$ *est bornée par* $P(max_{i=1,\ldots,n}|v_i|)$ *pour un certain polynôme P (voir [20]).*

En pratique, l'intérêt des quasi-interprétations est double :

1. D'une part, celles-ci permettent de caractériser des classes importantes de complexité. Par exemple, combinés avec certaines preuves de terminaison, les quasi-interprétations permettent de caractériser la classe PTIME des fonctions calculables en temps polynomial [78].
2. D'autre part, les quasi-interprétations permettent d'aborder le problème de la complexité des programmes d'un point de vue algorithmique. L'existence de quasi-interprétations peut en effet être décidée pour une large classe de programmes ce qui en fait un outil intéressant pour contrôler par analyse statique les ressources nécessaires à l'exécution d'un programme.

Propriété. *L'ensemble des programmes qui terminent par LPO et qui admettent une quasi-interprétation capture la classe PSPACE[19] des fonctions calculables en espace polynômial.*

On peut citer l'existence de deux logiciels liés aux quasi-interprétations. Le système ICAR[84] se base sur des variantes des ordres récursifs pour prouver la terminaison des programmes et permet de vérifier qu'un candidat est effectivement une quasi-interprétation. Dans le cas des programmes considérés, le problème de terminaison est dans PTIME. Le système Crocus[38] quand à lui, s'intéresse au problème de la synthèse des quasi-interprétations. A l'heure actuelle, ce programme étant en développement, il est difficile de faire une présentation des résultats obtenus.

Comme noté dans [3] l'utilisation des quasi-interprétations dans le cadre d'une analyse statique (en particulier dans le cas d'une approche proof-carrying code) pour contrôler les ressources nécessaires à l'exécution d'un programme pose deux problèmes :

- comment synthétiser une quasi-interprétation
- étant donnée une quasi-interprétation, comment vérifier que celle-ci correspond bien à un programme donné.

Si les polynômes considérés sont définis sur les entiers naturels, même le problème de la vérification est indécidable (10^{eme} problème de Hilbert). Si les polynômes considérés sont définis sur les réels, ce même problème est décidable (à l'aide de la procédure de décision de Tarski) mais sa complexité est

très élevée. Dans [3, 5], l'auteur suggère de considérer les polynômes de l'algèbre max-plus $((\mathbb{Q}^+ \cup \{-\infty\}, max, +))$. Il est démontré[5] que le problème de la synthèse est alors NP-difficile et dans NP pour le cas particulier des d'une certaine classe de polynômes dits multi-linéaires. En particulier, ette approche permet déjà de capturer les programmes caractérisés par l'approche de la section précédente.

Finalement, on peut noter que les quasi-interprétations, comme l'approche non-size increasing, peuvent être utilisés dans le cadre d'une approche *Proof-carrying code*. Dans [8] les auteurs ont adapté la notion de quasi-interprétation à une machine à pile dédié à l'exécution d'un bytecode pour un langage fonctionnel du premier ordre. Les vérifications de type, de contrôle de la taille des valeurs calculées et de terminaison des programmes sont réalisées au niveau de ce bytecode.

2.4 Concurrence

2.4.1 Hume

Le langage Hume[79, 54, 53] est un langage de programmation concurrente dédiés aux systèmes nécessitant un contrôle des ressource (espace/temps) tels que les systèmes embarqués temps réels. L'approche suivie pour assurer les caractéristiques temps réel des systèmes est un compromis entre analyse statique (pour le contrôle de l'occupation mémoire) et vérification dynamique (pour le contrôle des temps d'exécution). D'une part le langage vise à permettre le calcul de bornes statiques sur l'occupation mémoire, d'autre part une construction de *time-out* permet de forcer l'existence de bornes sur les temps de calcul. La définition de Hume repose sur plusieurs couches (*layers*) offrant une expressivité croissante du langage au prix d'une augmentation des difficultés d'analyse des programmes. Ce découpage du langage en plusieurs couches vise à permettre, au terme d'étapes successives de complexité croissante, de contrôler la complexité des programmes de la couche la plus élevée (le langage Hume complet) qui dispose de caractéristiques évoluées telles que l'ordre supérieur, le polymorphisme ou encore la gestion des exceptions. La version actuelle du langage permet d'obtenir des garanties fortes sur la complexité des programmes d'une couche de plus bas niveau, définissant un sous langage appelé FSM-HUME [81, 80] (pour Finite State Machine Hume).

En Hume, l'unité de base du calcul est la boite (*box*). Un programme est défini par la description de types de données et par un nombre fini de boites connectées par des buffers une-place appelés fils (*wires*) pour former un réseau de processus statique. Chaque boite définie une fonction (au sens large) de ses entrées vers ses sorties. Plus précisément, une boite est définie par un ensemble de règles de la forme $pattern- > function(pattern)$ (voir exemple plus loin). Chaque couche du langages est définie par un ensemble de restrictions sur les règles utilisables pour la définition des boites. Le sous language FSM-HUME est obtenu en se limitant aux structures de données non récursives et aux fonctions du premier ordre non récursives. Pour les programmes écrits dans ce sous langage le compilateur de Hume, qui repose sur la définition d'une machine abstraite[52], réalise le calcul d'une borne sur la quantité de mémoire nécessaire à l'exécution du programme. Une *sémantique de coût en espace* définie le comportement des programmes en terme d'occupation mémoire. Un exemple de

programme écrit en FSM-HUME et emprunté à [54] est donné ci-dessous.

```
type bit = word 1; type parity = boolean;

stream input from "/dev/sensor";
stream output from "std_out";

box even_parity
in (b :: bit, p :: parity)
out (p' :: parity, show :: string)
match
 (0, true) -> (true, "true")
| (1, true) -> (false, "false")
| (0, false) -> (false, "false")
| (1, false) -> (true, "true");

wire input to even_parity.b
wire even_parity.p' to even_parity.p initiallytrue
wire even_parity.show to output
```

Ce programme vérifie la parité d'une suite de booléens. La boite définie dans cet exemple, reçoit à chaque activation la valeur 0 ou 1 sur son entrée *b* et son état courant sur son entrée *p* (*p* est relié à la sortie p' de la boite, ce qui lui permet de conserver son état). Le résultat, déterminé par filtrage à l'aide de la construction `match`, est alors affiché sur la sortie standard par l'intermédiaire de la sortie *show*. Les déclarations *stream* et *wire* définissent les entrées/sorties et les branchements des fils. On renvoie le lecteur à [52] pour une description plus complète du langage, en particulier en ce qui concerne le mode d'ordonnancement des processus.

2.4.2 Programmation réactive

Dans cette section on survole brièvement le cadre de recherche dans lequel s'inscrit le chapitre suivant. Ce cadre de recherche pose pour objectif la dérivation de bornes sur les ressources nécessaires à l'exécution de code mobile afin d'éviter des attaques de type déni de service.

Dans [10] un langage concurrent du premier ordre à base de threads est étudié sous l'angle du contrôle des ressources. En particulier, certaines restrictions sont considérées afin de pouvoir définir le comportement d'un programme dans l'instant comme une fonction de ses paramètres au début de l'instant. Le langage proposé est à base de threads coopératifs partageant une notion d'instant et communiquant par variables partagées (pas de création dynamique). Au début de chaque instant, les variables partagées sont réinitialisées à une valeur par défaut. Les programmes manipulent des valeurs de types inductifs à la ML et disposent de constructions de filtrage sur ces valeurs. Le flot des données est restreint par le biais d'un condition, appelée *read-once*, qui stipule qu'un thread ne peut réaliser une opération de lecture donnée (relativement au code) qu'au plus une fois dans l'instant. Cette restriction permet de caractériser le comportement d'un thread dans l'instant comme une fonction de ses paramètres au début de l'instant et des valeurs lues sur les variables dans l'instant. L'analyse statique proposée assure que le calcul de chaque instant est réalisé en temps et en espace polynomiaux en la taille des paramètres du systèmes (*pile* des threads et valeurs des variables) au début de l'instant. La description d'une machine virtuelle permettant l'exécution d'un Byte Code obtenu par compilation du langage est également donnée. Cette description permet d'envisager l'adaptation des techniques

présentées dans [8] à ce langage concurrent. L'analyse statique proposée dans [10] ne permet pas de prédire la taille du système après un nombre arbitraire d'instants. Celle-ci repose sur une combinaison de l'analyse statique proposée et d'une vérification dynamique de la taille du système au début de chaque instant afin de décider, par exemple, d'arrêter certains threads susceptibles de consommer trop de ressources. Dans [9], nous avons considéré une extension de cette analyse statique qui permet de borner pour un nombre arbitraire d'instants les ressources nécessaires au calcul des instants. En particulier, nous avons introduit un mécanisme de masquage des paramètres qui permet de distinguer ceux utilisés strictement pour le calcul de l'instant en cours de ceux nécessaires pour le calcul d'un nombre arbitraire d'instants. Cette méthode permet de poser des restrictions sur les seconds afin de les forcer à être *Non-Size Increasing* ce qui a pour effet de borner la taille du système pour un nombre arbitraire d'instants.

Chapitre 3

Le π-Calcul Synchrone

Une opposition classique des langages de programmation concurrente distingue les langages synchrones d'une part et les langages asynchrones d'autre part. Le π-calcul[82] (et ses dérivés) peuvent être vus comme des modèles abstraits typiques de la programmation concurrente asynchrone. A l'opposé, le π-calcul synchrone (ou $S\pi$-calcul)[6], que l'on présente dans ce chapitre, est une algèbre de processus synchrone, inspirée par SL. En particulier, l'exécution des programmes du $S\pi$-calcul est réglée par la notion d'instant et la réaction à l'*absence* d'un signal ne peut se produire qu'à l'instant suivant.

Les langages synchrones (ESTEREL, LUSTRE, SIGNAL,...) ont été conçus de manière à permettre un ordonnancement statique et une compilation vers des automates finis. En conséquence, les modèles de programmation associés sont beaucoup plus restreints que le π-calcul et supportent mal des concepts tels que les types de données généraux, la mobilité des noms, les définitions récursives et la création dynamique de threads. Dans [7, 4], les auteurs proposent un modèle de programmation basé sur SL(cf Introduction) ; ce dernier permet la création dynamique de threads et les définitions récursives et propose une traduction CPS du langage vers une forme récursive terminale. Comme le langage SL, ce nouveau modèle suppose que les signaux sont *purs* au sens où ils ne transportent pas de valeurs (un signal pur peut être vu comme un booléen dénotant la présence ou l'absence du signal) ce qui permet, en particulier, de rendre les calculs déterministes. Cependant, les langages de programmation dérivés de SL incluent des types de données complexes portés par les signaux et cette extension rend les calculs *non-déterministes* si des restrictions conséquentes ne sont pas imposées.

Le $S\pi$-calcul peut être vu comme une extension minimale du modèle récursif terminal présenté dans [4] dans lequel les signaux peuvent porter des valeurs du premier ordre, noms de signaux inclus. La complexité du langage ainsi obtenu est comparable à celle du π-calcul. Le $S\pi$-calcul repose sur un sous-langage d'expressions fonctionnels du premier ordre dont les valeurs sont les valeurs de types de données inductifs. Le comportement des threads est défini par des équations récursives reposant sur l'utilisation d'une opération de filtrage à la ML et la création dynamique de nom de signaux et la composition parallèle de comportements sont possibles. Contrairement au langage SL, le $S\pi$-calcul n'est pas déterministe ; en revanche, le non-déterminisme est clairement isolé. Plus précisément, le non déterminisme du langage est uniquement lié à l'ordre de lecture des valeurs émises, au cours du même instant, sur les signaux.

3.1 Syntaxe

Dans cette section, on présente la syntaxe du $S\pi$-calcul. Pour cela, on commence par introduire plusieurs catégories syntaxiques. De manière générale, la notation $\mathbf{m}$ dénote un n-uplet $m_1 * \ldots * m_n$.

Les variables *Var* contiennent les noms de signaux ainsi que des variables d'autres types.

$$\begin{array}{lcll} \mathit{Sig} & ::= & s \mid t \mid \cdots & \text{(noms de signaux)} \\ \mathit{Var} & ::= & \mathit{Sig} \mid x \mid y \mid z \mid \cdots & \text{(variables)} \end{array}$$

Les valeurs *Val* sont des termes construits à partir des constructeurs et des noms de signaux. La *taille d'une valeur v*, notée $|v|$ est définie par $|s| = |\mathsf{c}| = 0$ où s est un signal et c est une constante, et $|\mathsf{c}(v_1, \ldots, v_n)| = 1 + \Sigma_{i=1,\ldots,n}|v_i|$ si $n \geq 1$. Les filtres *Pat* sont des termes construits à partir des constructeurs et des variables (y compris les noms de signaux). On suppose qu'un filtre p contient exactement un constructeur et que toutes les variables apparaissant dans p sont deux à deux distinctes.

$$\begin{array}{lcll} \mathit{Cnst} & ::= & * \mid \mathsf{nil} \mid \mathsf{cons} \mid \mathsf{c} \mid \mathsf{d} \mid \cdots & \text{(constructeurs)} \\ \mathit{Val} & ::= & \mathit{Sig} \mid \mathit{Cnst}(\mathit{Val}, \ldots, \mathit{Val}) & \text{(valeurs } v, v', \ldots) \\ \mathit{Pat} & ::= & \mathit{Var} \mid \mathit{Cnst}(\mathit{Pat}, \ldots, \mathit{Pat}) & \text{(filtres } p, p', \ldots) \end{array}$$

Si P et p sont un programme et un filtre alors on note $fn(P)$ (resp. $fn(p)$) l'ensemble des noms de signaux libres apparaissant dans P (resp. p). On note également $FV(P)$ (resp. $FV(p)$) l'ensemble des variables libres (y compris les noms de signaux) apparaissant dans P (resp. p). Les filtres seront utilisés pour la manipulation de valeurs de types inductifs ; la substitution de filtrage $match(v, p)$, où v est une valeur et p un filtre, est définie de la manière suivante :

$$match(v, p) = \begin{cases} \sigma & \text{si } dom(\sigma) = FV(p), \sigma p = v \\ \uparrow & \text{sinon.} \end{cases}$$

On suppose donnés des symboles de fonctions du premier ordre $f, g, \ldots$ dont le comportement sera défini de manière axiomatique. Les expressions *Exp* sont des termes construits à partir de variables, de constructeurs et de symboles de fonctions.

$$\begin{array}{lcll} \mathit{Fun} & ::= & f \mid g \mid \cdots & \text{(symboles de fonctions du premier ordre)} \\ \mathit{Exp} & ::= & \mathit{Var} \mid \mathit{Cnst}(\mathit{Exp}, \ldots, \mathit{Exp}) \mid \mathit{Fun}(\mathit{Exp}, \ldots, \mathit{Exp}) & \text{(expressions } e, e', \ldots) \end{array}$$

Les expressions étendues *Rexp* sont des expressions pouvant contenir la valeur associée avec un signal s à la fin de l'instant (ce que l'on note $!s$, en suivant la convention ML pour la déréférenciation). Intuitivement, cette valeur est une *liste de valeurs* représentant l'ensemble des valeurs émises sur un signal pendant l'instant.

$$\begin{array}{lcll} \mathit{Rexp} & ::= & !\mathit{Sig} \mid \mathit{Var} \mid \mathit{Cnst}(\mathit{Rexp}, \ldots, \mathit{Rexp}) & \\ & & \mid \mathit{Fun}(\mathit{Rexp}, \ldots, \mathit{Rexp}) & \text{(exp. avec déréf. } r, r', \ldots) \end{array}$$

Finalement, les programmes $P, Q, \ldots$ du $S\pi$-calcul sont définis par la grammaire suivante :

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

Comme dans le π-calcul, les noms de signaux représentent à la fois des constantes comme celles générées par l'opérateur ν et des variables comme celles des paramètres formel de l'opérateur *present*.

3.2 Sémantique Intuitive

La sémantique intuitive du $S\pi$-calcul est définie par induction structurelle sur les threads.

- 0 dénote le thread dont l'exécution est terminée.
- Chaque identificateur de thread est associé à une unique équation de la forme $A(\mathbf{x}) = P$. $A(\mathbf{e})$ dénote le thread dont le comportement est celui de $[\mathbf{v}/\mathbf{x}]P$ si $\mathbf{e}$ s'évalue en $\mathbf{v}$.
- $\overline{s}e$ dénote le thread dont le comportement consiste simplement à diffuser sur le signal s la valeur dénotée par e. Cette valeur est instantanément disponible (pour l'instant courant) sur le signal s pour l'ensemble des threads.
- $s(x).P, K$ dénote le test de la présence du signal s et correspond à la construction *present* qui est l'opérateur fondamental du modèle SL. Si les valeurs $\{v_1, \ldots, v_n\}$ sont les valeurs émises sur le signal s alors $s(x).P, K$ se comporte comme $[v_i/x]P$ pour $1 \leq i \leq n$. Le choix de i est non déterministe. Si aucune valeur n'est émise sur s au cours de l'instant alors le thread est suspendu jusqu'à la fin de l'instant et la *continuation* K est exécutée à l'instant suivant.
- $[s_1 = s_2] \cdot P_1,\ P_2$ correspond à la fonction de test habituelle du π-calcul et se comporte comme P_1 si $s_1 = s_2$ et comme P_2 sinon. A noter que s_1 et s_2 sont libres dans P_1.
- $[u \trianglerighteq p]\ P_1, P_2$ est la fonction de filtrage des valeurs. u est soit une variable x soit une valeur v et que p à la forme $\mathsf{c}(\mathbf{p})$ où c est un constructeur et $\mathbf{p}$ est un vecteur de filtres. A l'exécution, u est toujours une *valeur* ; on évalue σP_1 si le résultat σ du filtrage de u contre p est défini et P_2 sinon. De manière standard, les variables apparaissant dans p sont liées dans P_1.
- $\nu s\ P$ se comporte comme P où s dénote un nouveau nom de signal.
- $P_1 \mid P_2$ dénote le comportement de P_1 et P_2 en parallèle et termine quand P_1 et P_2 terminent.

Remarque 3. *Pour illustrer la différence entre les deux types de filtrage, supposons que s et s' sont deux noms de signaux distincts et considérons les programmes suivants :*

$$P = [s = s'] \cdot P_1,\ P_2 \qquad P' = [\mathsf{cons}(s, \mathsf{nil}) \trianglerighteq \mathsf{cons}(s', \mathsf{nil})]\ P_1, P_2$$

Le programme P se réduit en P_2 alors que le programme P' se réduit en $[s/s']P_1$. Dans le premier cas s' est une constante alors que dans le second c'est une variable liée.

3.3 Typage

L'ensemble des types contient :

- le type de base 1 dont l'unique valeur est la constante $*$,
- si t est un type, le type $Sig(t)$ des signaux portant des valeurs de type t,
- si t est un type, le type $List(t)$ des listes de valeurs de type t construites à l'aide des constructeurs nil et cons.

Les types 1 et $List(t)$ sont des exemples de types inductifs. D'autres types inductifs (booléens, entiers naturels, arbres,...) peuvent être ajoutés à l'aide de constructeurs supplémentaires. On suppose que

les variables (y compris les noms de signaux), les constructeurs, les symboles de fonctions et les identificateurs de threads sont fournis avec leur type du *premier ordre*. Par exemple, un constructeur c peut avoir le type $t_1, t_2 \rightarrow t$ signifiant qu'il attend deux arguments respectivement de types t_1 et t_2 et retourne une valeur de type t. Il est alors trivial de déterminer lorsqu'un programme est bien typé et de vérifier que cette propriété est conservée par la sémantique de réduction donnée plus loin. On note simplement que pour un signal s de type $Sig(t)$ la valeur déréférencée $!s$ doit avoir le type $List(t)$. Par la suite, on suppose que l'on a à faire à des programmes, expressions, substitutions,... bien typés.

3.4 Opérateurs Dérivés

On introduit plusieurs opérateurs dérivés et des abréviations en se basant pour l'instant sur la sémantique informelle de la section 3.1. On note $s(x).P$ pour $s(x).P, A()$ où $A() = 0$.
Un opérateur de choix interne peut être dérivé de la manière suivante :

$$P_1 \oplus P_2 \quad = \quad \nu s\ (s(x).[x \trianglerighteq 0]\ P_1, P_2 \mid \overline{s}0 \mid \overline{s}1)$$

où l'on pose, par exemple, $0 = \mathsf{nil}$ et $1 = \mathsf{cons}(*, \mathsf{nil},)$.
La construction `pause`.K suspend l'exécution jusqu'à la fin de l'instant et évalue K à l'instant suivant. Cette construction peut être définie par

$$\texttt{pause}.K \quad = \quad \nu s\ s.0, K$$

La construction `await` $s(x).P$ attend une valeur v sur le signal s et se comporte comme P où v est liée à x. Cette construction peut être définie par

$$\texttt{await}\ s(x).P = s(x).P, A(\mathbf{x})$$

où $\{\mathbf{x}\} = \{s\} \cup FV(P)$ et $A(\mathbf{x}) = s(x).P, A(\mathbf{x})$.

On anticipe sur le fait que la sémantique ne fait pas de distinction en fonction du nombre d'émissions d'une même valeur sur un signal. S'il est nécessaire de compter le nombre d'émissions il suffit de combiner (à l'aide d'un constructeur) un nouveau nom de signal à chaque valeur émise.

Remarque 4. *Les calculs avec signaux purs considérés dans [27, 4, 7] peuvent être retrouvés en supposant que tous les signaux sont de type $Sig(1)$. Dans ce cas, on note simplement $\overline{s}$ pour $\overline{s}*$ et $s.P, K$ pour $s(x).P, K$ où $x \notin FV(P)$.*

3.5 Comparaison avec le π-calcul

Afin de faciliter la comparaison, la syntaxe du $S\pi$-calcul est similaire à celle du π-calcul. Les seules nouveautés syntaxiques sont la présence d'une branche *else* dans la construction de réception des messages et la possibilité de *déréférencer* un signal. Cependant, il y a des différences sémantiques significatives à garder à l'esprit.

Deadlock vs. Fin de l'instant. Que se passe-t-il lorsque tous les threads sont soit terminés soit en attente d'un événement ? Dans le π-calcul, le calcul s'arrête. Dans le $S\pi$-calcul (et de manière plus générale dans le modèle SL) cette situation est détectée et marque la fin de l'instant courant. Les threads suspendus et les signaux sont alors réinitialisés et le calcul se poursuit dans l'instant suivant.

Canaux vs. Signaux. Dans le π-calcul, un message est consommé par son récepteur. Dans le $S\pi$-calcul, une valeur émise sur un signal *persiste* dans l'instant et est réinitialisé à la fin de celui-ci.

Types de données. Le π-calcul (polyadique [83]) possède des *n-uplets* comme type de données de base, alors que le $S\pi$-calcul possède des *listes*. La motivation pour l'introduction des listes plutôt que des *n*-uplets comme type de base du modèle est qu'à la fin de l'instant l'ensemble des valeurs émises sur un signal est transformé en une structure adaptée (une liste) représentant cet ensemble et pouvant être traitée à l'instant suivant.

Exemple 6. *On présente un exemple illustrant cette discussion. Supposons que $v_1 \neq v_2$ sont deux valeurs distinctes et considérons le programme suivant :*

$$P \quad = \quad \nu s_1, s_2\ (\overline{s_1}v_1 \mid \overline{s_1}v_1 \mid \overline{s_1}v_2 \mid s_1(x).(s_1(y).(s_2(z).0, A(x, y, !s_1)), 0), 0)$$

Dans le $S\pi$-calcul, P se réduit en :

$$P_1 \quad = \quad \nu s_1, s_2\ s_2(z).0, A(\sigma(x), \sigma(y), !s_1)$$

où $\sigma(x), \sigma(y) \in \{v_1, v_2\}$. La réduction du même programme P dans le π-calcul (en ignorant les branches else*), donnerait :*

$$P_2 = \nu s_1, s_2\ (s_2(z).0, A(\sigma'(x), \sigma'(y), !s_1))$$

où $\sigma'(x), \sigma'(y) \in \{v_1, v_2\}$ et de plus soit $\sigma'(x) \neq v_2$ soit $\sigma'(y) \neq v_2$. La différence dans le calcul de la substitution est lié au fait que l'émission d'un signal dans le $S\pi$-calcul est persistante (dans l'instant) alors qu'un message est consommé dans le π-calcul de telle manière que la valeur v_2 ne peut pas être lue deux fois.

Les deux programmes P_1 et P_2 semblent ne pas pouvoir subir de nouvelle réduction. Cela est détecté dans le $S\pi$-calcul et un calcul supplémentaire est effectué pour préparer le programme à réaliser l'instant suivant. Dans notre cas, P_1 est initialisé à

$$P'_1 = \nu s_1, s_2\ A(\sigma(x), \sigma(y), \sigma(\ell))$$

où, à nouveau, $\sigma(x), \sigma(y) \in \{v_1, v_2\}$ et $\sigma(\ell) \in \{\mathsf{cons}(v_1, \mathsf{cons}(v_2, \mathsf{nil})), \mathsf{cons}(v_2, \mathsf{cons}(v_1, \mathsf{nil}))\}$. On voit ici que la déréférenciation du signal s_1 à la fin de l'instant revient à collecter selon une permutation arbitraire la liste des valeurs (distinctes) émises sur ce signal durant l'instant. De plus, une valeur émise persiste durant l'instant mais disparaît à la fin de celui-ci.

3.6 Exemples de programmes

On introduit plusieurs exemples de programmes sur lesquels on se basera par la suite pour illustrer nos techniques d'analyse statique.

Exemple 7. *Le modèle synchrone est particulièrement bien adapté à la simulation de divers types de systèmes (on renvoie à [90] pour une liste d'exemples). On décrit ici le comportement d'une cellule d'un automate cellulaire générique. La définition de chaque cellule repose sur trois paramètres : son propre signal d'activation s, son état q et la liste l des signaux d'activation des cellules voisines. La cellule réalise les actions suivantes de manière cyclique :*

1. *émettre son état pour l'instant courant sur les signaux d'activation des cellules voisines*
2. *se suspendre pour l'instant courant*
3. *collecter les valeurs émises par ses voisines et calculer son nouvel état.*

Ce comportement peut être décrit de la manière suivante :

$$\begin{array}{lcll} Cell(s,q,l) & = & Send(s,q,l,l) & \\ Send(s,q,l,l') & = & [l' \trianglerighteq \mathsf{cons}(s',l'')] & (\overline{s'}q \mid Send(s,q,l,l'')), \\ & & & \mathtt{pause}.Cell(s,next(q,!s),l) \end{array}$$

où next est une fonction qui calcule l'état suivant d'une cellule en fonction de son état courant et de l'état des cellules voisines. On suppose donné un type énuméré state contenant une constante pour chaque état possible. Le type des signaux s et s' est $Sig(state)$, le type des listes l et l' est $List(Sig(state))$ et le type de la fonction next est $state, List(state) \to state$.

Exemple 8. *Cet exemple décrit un* serveur *($Server(s)$) répondant à chaque instant à une liste de requêtes émises dans l'instant précédent sur un signal s. Le type treq des requêtes contient le constructeur* req *de type $Sig(t'), t \to treq$. Pour chaque requête de la forme* req(s', x)*, le serveur produit une réponse, qui est une fonction de x, sur le signal s'.*

$$\begin{array}{lcll} Server(s) & = & \mathtt{pause}.Handle(s,!s) & \\ Handle(s,l) & = & [l \trianglerighteq \mathsf{cons}(\mathsf{req}(s',x),l')] & (\overline{s'}\,f(x) \mid Handle(s,l')), \\ & & & Server(s) \end{array}$$

On suppose que la fonction f est de type $t \to t'$. Le paramètres s est de type $Sig(treq)$ et les listes l et l' sont de type $List(treq)$.

Exemple 9. *L'exemple suivant décrit deux threads : à chaque instant, le thread $A(s)$ émet sur s les valeurs émises sur s à l'instant précédent alors que le thread $C(s)$ produit une (nouvelle) valeur sur s.*

$$\begin{array}{lcl} A(s) & = & \mathtt{pause}.B(s,!s) \\ B(s,l) & = & [l \trianglerighteq \mathsf{cons}(n,l')]\,(\overline{s}n \mid B(s,l')), A(s) \\ C(s) & = & \nu n\, \overline{s}n \mid \mathtt{pause}.C(s) \end{array}$$

En supposant que n est de type $Sig(1)$, s est de type $Sig(Sig(1))$ et la liste l est de type $List(Sig(1))$.

3.7 Sémantique Formelle

Dans cette section, on définit formellement la sémantique de réduction du $S\pi$-calcul. On suppose donnée une relation $\Downarrow$ telle que pour tout symbole de fonction f et pour toutes valeurs $v_1, \ldots, v_n$, bien typées, il existe une unique valeur v telle que $f(v_1, \ldots, v_n) \Downarrow v$. On suppose de plus que cette valeur peut être calculée en temps polynomial en la taille des valeurs $v_1, \ldots, v_n$. Comme mentionné précédemment, les techniques pour définir des programmes fonctionnels du premier ordre satisfaisant à ces conditions ont été bien étudiées. La relation d'évaluation $\Downarrow$ est étendue aux expressions de manière standard par :

$$\frac{}{s \Downarrow s} \qquad \frac{e_i \Downarrow v_i,\ i = 1, \ldots, n}{\mathsf{c}(e_1, \ldots, e_n) \Downarrow \mathsf{c}(v_1, \ldots, v_n)} \qquad \frac{e_i \Downarrow v_i,\ i = 1, \ldots, n \qquad f(v_1, \ldots, v_n) \Downarrow v}{f(e_1, \ldots, e_n) \Downarrow v}$$

Par la suite, on notera plus simplement $\mathbf{e} \Downarrow \mathbf{v}$ pour $e_1 \Downarrow v_1, \ldots, e_n \Downarrow v_n$

Le comportement (interne) d'un programme est donné par

- un *système de réduction* $\to$ décrivant les réductions possibles du programme *pendant un instant*,
- une *relation d'évaluation* $\mapsto$ décrivant la façon dont un programme évolue *à la fin de chaque instant*.

Ces définitions reposent sur une relation d'équivalence structurelle $\equiv$ que l'on commence par introduire.

Équivalence structurelle. L'équivalence structurelle $\equiv$ est la plus petite relation sur les programmes qui identifie les programmes à α-renommage près et qui satisfait les équations standards suivantes :

$$P \mid 0 \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$

$$\nu s\, P \equiv P \text{ if } s \notin fn(P) \qquad \nu s\, P_1 \mid P_2 \equiv \nu s\, P_1 \mid P_2 \text{ if } s \notin fn(P_2)$$

Réduction dans l'instant. De manière standard, on se base sur la notion de *contexte statique* pour définir la relation de réduction. La relation de réduction $\rightarrow$ est alors définie, à l'aide de la relation auxiliaire $\twoheadrightarrow$, par les règles de la figure 3.1 où un contexte statique C est défini par $C ::= [] \parallel \nu s\, C \mid \mid (C \mid P)$

$$\frac{e \Downarrow v}{\overline{s}e \mid s(x).P, K \twoheadrightarrow \overline{s}e \mid [v/x]P} \qquad \frac{A(\mathbf{x}) = P \qquad \mathbf{e} \Downarrow \mathbf{v}}{A(\mathbf{e}) \twoheadrightarrow [\mathbf{v}/\mathbf{x}]P}$$

$$\frac{}{[s = s] \cdot P_1,\ P_2 \twoheadrightarrow P_1} \qquad \frac{s \neq s'}{[s = s'] \cdot P_1,\ P_2 \twoheadrightarrow P_2}$$

$$\frac{match(v, p) = \sigma}{[v = p] \cdot P_1,\ P_2 \twoheadrightarrow \sigma P_1} \qquad \frac{match(v, p) = \uparrow}{[v = p] \cdot P_1,\ P_2 \twoheadrightarrow P_2}$$

$$\frac{P \equiv C[P'] \qquad P' \twoheadrightarrow Q' \qquad C[Q'] \equiv Q}{P \rightarrow Q}$$

FIG. 3.1 – Réduction dans l'instant

Calcul de fin d'instant. On note $P \downarrow$ si $\neg\exists Q.(P \rightarrow Q)$ et on dit alors que le programme P est suspendu. Lorsque P est suspendu l'instant termine et un calcul additionnel a lieu pour passer à l'instant suivant. Ce calcul présente trois étapes :

1. collecter sous forme de listes les ensembles de valeurs émises sur chacun des signaux
2. Extruder les noms de signaux contenus dans des valeurs visibles à la fin de l'instant
3. initialiser les continuations des opérateurs *present*.

On commence par introduire plusieurs notations, en notant qu'un programme suspendu est structurellement équivalent à

$$\nu \mathbf{s}\ (S \mid In) \tag{3.1}$$

où les noms de signaux $\mathbf{s}$ sont tous distincts et où pour $m, n \geq 0$:

$$S = \overline{s_1}e_1 \mid \ldots \mid \overline{s_n}e_n \qquad In = t_1(x_1).P_1, A_1(\mathbf{r}_1) \mid \ldots \mid t_m(x_m).P_m, A_m(\mathbf{r}_m)$$

Par convention, une composition parallèle vide est égale au programme 0. On note $\overline{s}e \in S$ si $\overline{s}e$ apparaît dans la composition parallèle S. Les étapes 1 à 3 sont alors formalisées de la manière suivante :

1. Soit V une fonction des noms de signaux vers les listes de valeurs. On dit que V représente S et on note $V \models S$ si, pour tout signal s, si $\{v_1, \ldots, v_n\} = \{v \mid \overline{s}e \in S,\ e \Downarrow v\}$ alors $V(s) = [v_{\pi(1)}; v_{\pi(2)}; \ldots; v_{\pi(n)}]$ pour une certaine permutation π.

2. On définit $Free(\nu\mathbf{s}\ S)$ comme le plus petit ensemble de noms de signaux tel que $Free(\nu\mathbf{s}\ S) \supseteq fn(\nu\mathbf{s}\ S)$ et si $s \in Free(\nu\mathbf{s}\ S)$, $\overline{s}e \in S$, $e \Downarrow v$ et $s' \in fn(v)$ alors $s' \in Free(\nu\mathbf{s}\ S)$. Par exemple, $Free(\nu s_1, s_2\ (\overline{s}s_1 \mid \overline{s_1}s_2)) = \{s, s_1, s_2\}$.
3. Si r est une expression avec déréférenciation alors $V(r)$ est l'expression résultant du remplacement de tout signal déréférencé $!s$ par $V(s)$. Si $A(\mathbf{r})$ est la continuation d'un opérateur *present*, où $\mathbf{r}$ est une liste d'expressions closes, alors $Eval(A(\mathbf{r}), V) = A(\mathbf{v})$ si $V(\mathbf{r}) \Downarrow \mathbf{v}$. Finalement, si In est défini comme dans (3.1) alors $Eval(In, V) = Eval(A_1(\mathbf{r}_1), V) \mid \ldots \mid Eval(A_m(\mathbf{r}_m), V)$.

En suivant ces conventions, la règle d'évaluation à la fin de l'instant est donnée par :

$$\frac{P \downarrow \qquad P \equiv \nu\mathbf{s}\ (S \mid In) \qquad V \models S \qquad \{\mathbf{s}'\} = \{\mathbf{s}\} \backslash Free(\nu\mathbf{s}\ S) \qquad P' \equiv \nu\mathbf{s}'\ Eval(In, V)}{P \mapsto P'}$$

FIG. 3.2 – Évaluation à la fin de l'instant

Intuitivement, si le programme est suspendu, i.e. $P \downarrow$ alors on procède de la manière suivante :

1. dans P, on distingue les émissions S des programmes suspendus en attente d'un signal In,
2. on calcule une représentation V des émissions S
3. on calcule les noms de signaux extrudés
4. on retire les émissions et on initialise les continuations des opérateurs *present*.

Chapitre 4

Le $S\pi$-calcul à contrôle fini

Dans le paradigme réactif, un programme doit réagir (ou émettre des signaux de sortie) à une collection d'événements (où signaux d'entrée) à chaque instant. Cette spécification intuitive repose sur une notion *bien fondée* d'instant ; en d'autres termes, cela suppose que les réactions d'un programme terminent toujours. Pour des programmes écrits dans un langage suffisamment expressif, cette hypothèse n'est pas, en général, satisfaite ce qui conduit à l'introduction d'une notion spécifique de correction que l'on appelle *réactivité*. Intuitivement, on dit qu'un programme est réactif si chaque réaction calculée par ce programme termine. Dans ce chapitre on s'intéresse plus particulièrement à une notion plus effective de réactivité que l'on appelle *réactivité efficace*. D'après [33],

> Reactivity means the ability to react to external events, and, quite obviously, requires bounded memory and reaction time

On considère ici un sous ensemble du $S\pi$-calcul pour lequel on développe des méthodes d'analyse statique permettant de garantir qu'un programme calcule ses réactions en temps et espace polynomiaux en la taille de ses entrées. Comme nous l'avons déjà noté, une complexité polynomiale est généralement considérée comme une bonne approximation de la notion de *complexité raisonnable*.

Une réaction produit des sorties en réponse à des entrées fournies par l'environnement. En pratique, si l'on considère des langages de programmation suffisamment expressifs, le calcul d'une réaction peut dépendre de l'histoire du calcul (les instants précédents) et non seulement des entrées fournies par l'environnement pour cette réaction. De manière évidente, un programme nécessitant de conserver une trace complète de l'histoire du calcul n'est pas raisonnable. Dans des travaux précédents [4] nous avons développé des méthodes d'analyse statique permettant de garantir que la quantité de ressources nécessaire au calcul des réactions d'un programme est polynômiale en la taille de ses entrées. Bien que le langage présenté dans ces travaux dispose de valeurs de types de données dynamiques, le calcul sur celles-ci était fortement confiné dans l'instant ; l'histoire du calcul (plus précisément l'histoire des communications) devant être représentée par des valeurs de types de données finis. Un exemple typique est celui d'un booléen dénotant la présence d'un signal à l'instant précédent (à noter que les entiers 32-bits, les flottants,... sont également des types de données finis). De nombreux programmes synchrones supportent cette restriction. Cependant, d'autres programmes ont besoin de plus d'un instant pour réaliser certains calculs nécessitant un accès à l'histoire du calcul plus complexe. Par exemple, on peut considérer un serveur recevant une liste de requêtes auquel celui-ci doit répondre lors du premier instant au cours duquel un signal donné est émis.

Dans ce chapitre, on considère un sous-ensemble du $S\pi$-calcul dans lequel la récursion est restreinte

de manière à ne pas passer sous la composition parallèle (intuitivement, cela revient à dire que l'on n'autorise pas la création dynamique de threads).

4.1 Contribution

La contribution de ce chapitre est de proposer une méthodologie pour annoter les programmes du $S\pi$-calcul et de développer des méthodes d'analyse statique garantissant la réactivité efficace des programmes. La restriction principale apportée au $S\pi$-calcul dans nos développements suppose que l'on considère des programmes impliquant un nombre borné de threads. Pour assurer cette propriété, on se basera sur une condition standard qui stipule que la récursion ne passe pas sous la composition parallèle. En se basant sur cette hypothèse, on suppose qu'un programme est donné avec deux types d'*annotations* concernant les identificateurs de threads et les signaux.

Annotation des identificateurs de thread

Une caractéristique des programmes synchrones est que chaque thread réalise un certain ensemble d'opérations de manière cyclique. Un cycle est différent d'un instant car son exécution peut se dérouler sur un nombre arbitraire d'instants (éventuellement un nombre non borné). De plus, un cycle est propre à un thread, contrairement à la notion d'instant qui est commune à l'ensemble des threads. On demande à ce qu'un sous ensemble des identificateurs de thread dénotent explicitement la fin d'un cycle et le début d'un nouveau cycle. Cette annotation n'a pas d'effet sur la sémantique opérationnelle mais sera utilisée pour produire certaines contraintes de l'analyse statique. Intuitivement, on demande que chaque thread lors de chaque cycle ne puisse effectuer qu'un nombre fini de lectures. Une implication technique de cette restriction est que le comportement d'un thread au cours d'un cycle est une *fonction* de ses paramètres au début du cycle et des valeurs lues (en nombre fini) sur les signaux au cours de ce cycle.

Les identificateurs de thread disposent de deux annotations additionnelles :

- Un de nos objectifs pour garantir la réactivité efficace des programmes sera de montrer que les paramètres d'un thread (au début de ses cycles) sont d'une certaine manière *non-size increasing*. Pour ce faire il est parfois utile de ne pas considérer l'ensemble des paramètres d'un thread au cours de l'analyse. En conséquence, on associe explicitement avec chaque identificateur de thread un sous ensemble de ses paramètres qui sera considéré dans l'analyse de la taille de ce thread.
- Un second objectif sera de montrer que chaque instant termine. Il est alors naturel de *comparer* les identificateurs de threads et leurs paramètres selon un ordre bien fondé. Pour cette raison, on suppose que chaque identificateur de thread est annoté par un *statut* décrivant la manière dont ses paramètres doivent être comparés (typiquement, selon un ordre lexicographique ou un ordre multi-ensemble).

Pour résumer, un identificateur de threads possède trois types d'annotations :

- une spécifiant s'il dénote ou non la fin (le début) d'un cycle,
- une spécifiant le sous-ensemble de ses paramètres qui sont significatifs dans l'analyse de sa taille,
- une spécifiant le type de comparaison à utiliser sur ses paramètres pour l'analyse de terminaison.

Annotations des noms de signaux

D'une part, un programme devrait pouvoir émettre sur un signal des valeurs lues sur d'autres signaux. D'autre part, on cherche à éviter des situations où, par exemple, un programme lit et émet sur un

signal de manière répétitive une valeur de plus en plus grosse. Pour éviter ce type de problème, on suppose que l'ensemble des noms de signaux est partitionné en *régions* $\rho_1, \rho_2, \ldots$ ordonnées. Plus précisément, on raffine le système de type de manière à ce que les types des signaux soient annotés par une région comme dans le type $Sig_\rho(t)$. Le type d'un nom de signal porte explicitement une information sur la région à laquelle ce nom de signal appartient. A nouveau, cette annotation n'affecte pas la sémantique opérationnelle mais sera utile pour la générations de conditions lors de l'analyse statique. Intuitivement, ces conditions imposent que la taille d'une valeur émise sur un signal de région ρ soit bornée par une fonction de la taille des valeurs lues sur des signaux de régions strictement plus petite que ρ.

A partir des annotations que l'on vient de présenter, on donne une description informelle des conditions statiques imposées aux programmes.

Quasi-interprétations

En premier lieu, nous avons besoin d'un moyen approprié pour *abstraire la taille des valeurs*. Dans ce but, on adapte la notion de quasi-interprétation présentée plus tôt. On anticipe sur le fait que si v est une valeur alors on notera $|v|$ la taille de cette valeur. Comme on peut le voir de manière plus précise dans le chapitre 2 les quasi-interprétations sont similaires aux interprétations utilisées pour prouver la terminaison des systèmes de réécriture. Cependant leur but initial n'est pas de prouver la terminaison mais de borner la taille des valeurs calculée. Dans le travail présenté ici, nous utiliserons également les quasi-interprétations comme outil d'approximation de la taille des paramètres des appels récursifs pour prouver la terminaison des instants.

Inégalités

Nous décrivons une méthode pour associer à un programme un ensemble fini d'inégalités sur des termes du premier ordre et nous prouvons que lorsque ces inégalités, interprétées à l'aide d'une quasi-interprétation, sont satisfaites alors le programme considéré est réactif efficace. Ces inégalités peuvent être classées en trois catégories associées à des buts bien précis :

1. assurer la terminaison des instants
2. assurer que la taille des paramètres des threads au début des cycles est non-size increasing
3. assurer que la taille des valeurs calculées durant un cycle est polynômiale en la taille des paramètres du thread au début du cycle et des valeurs lues sur les signaux durant ce cycle.

L'ensemble de ces contraintes dépendent des annotations que l'on a décrit précédemment.

On anticipe sur les développements à venir qu'une caractéristique essentielle de notre approche est qu'elle fait abstraction dans une large mesure des valeurs émises sur les signaux et de l'ordonnancement des threads. Cela signifie que chaque thread peut être analysé séparément, que la complexité de l'analyse croit linéairement en le nombre de threads, et qu'une analyse incrémentale d'un système de threads changeant dynamiquement est possible.

4.2 Réactivité

Au début de chaque instant, un programme reçoit une entrée que l'on peut représenter par un nouvel identificateur de thread *Env* défini par une équation :

$$Env() = \overline{s_1}v_1 \mid \ldots \mid \overline{s_n}v_n$$

On écrit alors

$$P \overset{Env}{\mapsto} P' \text{ si } P \mapsto P'' \text{ et } P'' \equiv (P' \mid Env())$$

Par les propriétés du modèle on peut, sans perte de généralité, supposer que dans une entrée toutes les valeurs émises sur un signal s sont distinctes.

Définition 12. *Un calcul d'un programme P est une suite infinie dénombrable de programmes $P_1, P_2, \ldots$ telle que*

$$P \equiv P_{i_0+1} \overset{*}{\to} P_{i_1} \overset{Env_1}{\mapsto} P_{i_1+1} \overset{*}{\to} P_{i_2} \overset{Env_2}{\mapsto} P_{i_2+1} \overset{*}{\to} \ldots$$

Intuitivement, P_{i_k+1} est le programme au début de l'instant k. De manière générale, la réduction de $P_1, P_{i_1+1}, P_{i_2+1}, \ldots$ peut ne pas atteindre la fin de l'instant et on appelle *réactifs* les programmes qui se suspendent à chaque instant. L'ensemble des programmes réactifs est défini par co-induction à partir des programmes qui *terminent*.

Définition 13. *L'ensemble Ter des programmes qui* terminent *est le plus petit ensemble de programmes tels que*

$$Si\ \forall P'(P \to P'\ impl.\ P' \in Ter)\ alors\ P \in Ter$$

L'ensemble React des programmes réactifs est le plus grand ensemble de programmes tels que :

$$\forall P \in React, S, Env. \begin{cases} (P \mid S) \in Ter \\ ((P \mid S) \overset{*}{\to} P' \overset{Env}{\mapsto} P'')\ impl.\ P'' \in React \end{cases}$$

Exemple 10. *Dans l'exemple (9), un calcul possible du programme $A(s) \mid C(s)$ est donné par :*

$$\begin{aligned} A(s) \mid C(s) \quad & \overset{*}{\to} \mathsf{pause}.B(s, !s) \mid \nu n_0\ \overline{s} n_0 \mid \mathsf{pause}.C(s) \\ & \overset{Env}{\mapsto} B(s, \mathsf{cons}(n_0, \mathsf{nil})) \mid C(s) \\ & \overset{*}{\to} \overline{s} n_0 \mid \mathsf{pause}.B(s, !s) \mid \nu n_1\ \overline{s} n_1 \mid \mathsf{pause}.C(s) \\ & \overset{Env}{\mapsto} B(s, \mathsf{cons}(n_0, \mathsf{cons}(n_1, \mathsf{nil}))) \mid C(s) \\ & \ldots \end{aligned}$$

Dans ce cas, on suppose que l'entrée au début de chaque instant est vide, i.e. $Env() = 0$.

On suppose que, initialement, un programme a la forme

$$\nu \mathbf{s}\ A_1(\mathbf{v}_1) \mid \ldots \mid A_n(\mathbf{v}_n) \tag{4.1}$$

pour $n \geq 0$. Par définition de la construction `present` et de l'entrée au début de l'instant, un programme a cette forme, à équivalence structurelle près, au début de chaque instant. La définition de *réactivité efficace* est relative à la taille du programme initial et à la taille de la (plus grande) entrée. Par convention, la taille d'un programme de la forme (4.1) est n plus la somme de la taille des valeurs $\mathbf{v}_1, \ldots, \mathbf{v}_n$. La taille d'une entrée Env définie par une équation $Env() = \overline{s_1} v_1 \mid \ldots \mid \overline{s_n} v_n$ est la somme des tailles des valeurs $v_1, \ldots, v_n$.

Définition 14 (réactivité efficace). *Un programme P de la forme (4.1) est réactif efficace s'il existe un polynôme Q tel que pour tout calcul*

$$P \equiv P_{i_0+1} \overset{*}{\to} P_{i_1} \overset{Env_1}{\mapsto} P_{i_1+1} \overset{*}{\to} P_{i_2} \overset{Env_2}{\mapsto} P_{i_2+1} \overset{*}{\to} \ldots$$

si d borne la taille de P et la taille des entrées $Env_1, Env_2, \ldots, Env_k$ pour $k \geq 0$ alors

1. *la taille de P_{i_k+1} (le programme au début de l'instant k) est bornée par $Q(d)$.*
2. *P_{i_k+1} se suspend en temps au plus $Q(d)$.*

Par exemple, le programme de l'exemple (9) n'est pas réactif efficace car la taille du paramètre de *B,* et donc du programme, augmente de 1 à chaque instant.

4.3 Annotations

Un programme est associé à un système fini d'équations. Notre analyse statique se concentre sur ce système et est indépendante du programme particulier utilisé pour initialiser le calcul. Le lecteur devra garder à l'esprit que l'analyse d'un programme est en fait l'analyse du système associé. Pour des raisons techniques, il sera utile de supposer qu'un programme ne contient pas de filtrage triviaux tels que $[v \trianglerighteq p]\ P_1, P_2$ (filtrage d'une valeur) $[s \trianglerighteq s]\ P_1, P_2$ (comparaison de deux noms de signaux identiques). De la même manière, dans $[x \trianglerighteq p]\ P_1, P_2$ on suppose que x n'est pas filtrée à nouveau dans P_1. De tels filtrages peuvent être éliminés par une simple exécution symbolique. Modulo cette simplification, l'analyse statique fait abstraction des noms de signaux réels.

4.3.1 Borne sur le nombre de threads

On définit une fonction *tc* qui compte le nombre de threads dans un programme P :

$tc(P)$	=	`case` P `of`
0		0
$B(\mathbf{e})$		1
$\overline{s}e$		0
$[s_1 = s_2] \cdot P_1,\ P_2$		$max\{tc(P_1), tc(P_2)\}$
$[u \trianglerighteq p]\ P_1, P_2$		$max\{tc(P_1), tc(P_2)\}$
$s(x).P, B(\mathbf{r})$		$max\{tc(P), 1\}$
$\nu s\ P'$		$tc(P')$
$P_1 \mid P_2$		$tc(P_1) + tc(P_2)$

On demande que dans toute équation récursive $A(\mathbf{x}) = P$ la valeur $tc(P)$ soit au plus 1. Il est immédiat que cette condition implique l'existence d'une borne sur le nombre de threads présents dans tout programme accessible. Le lecteur pourra vérifier que cette condition est satisfaite par les exemples (7), (8) et (9)

4.3.2 Annotation des identificateurs de threads

Annotations de ré-initialisation

On note *Reset* un sous-ensemble des identificateurs de threads contenant les identificateurs correspondant au début d'un nouveau *cycle*. Un identificateur de thread A dans *Reset* doit vérifier une des conditions suivantes :

- soit A est définie par une équation de la forme $A(\mathbf{x}) =$ `pause`$.K$
- soit toutes les occurrences de A dans le programme sont dans la branche *else* d'une construction *present*.

Par ces conditions syntaxiques, on garantit en particulier que la fin d'un cycle pour un thread implique toujours la fin de son calcul pour l'instant courant. Par exemple, dans l'exemple (8) il est naturel de considérer que $Server \in Reset$ et $Handle \notin Reset$.

Remarque 5. *A noter que du fait de la condition sur le nombre de threads present dans un programme accessible, il n'y a pas d'ambiguïté sur ce qu'est la fin du cycle d'un thread. Si la création de nouveaux threads en parallèle était possible, il serait nécessaire de* synchroniser *les points de ré-initialisation des différents (sous-)threads.*

4.3.3 La condition read-once

Comme nous l'avons vu, un programme peut lire un signal soit dans l'instant soit par l'opération de déréférenciation à la fin de l'instant. La condition *read-once* est l'hypothèse selon laquelle, pour chaque thread et dans tout cycle, il existe une borne (une constante quelconque) sur le nombre de fois que la lecture d'un signal peut être réalisée. En particulier, on demande et on vérifie statiquement sur le *graphe d'appel* (définie ci-dessous) que le calcul réalisé en partant de tout identificateur de thread ne peut exécuter chaque instruction de lecture qu'au plus une fois dans un même cycle.

1. **(étiquettes).** On associe à chaque construction *present* et à chaque déréférenciation dans le texte d'un programme une nouvelle étiquette (variable) y et on collecte toutes ces étiquettes sous la forme d'une suite ordonnée $y_1, y_2, \ldots, y_m$. Par la suite, on utilisera les notations

$$s^y(x).P, K \qquad !^y s$$

   pour rendre les étiquettes explicites dans le texte du programme. Si $\mathbf{r}$ est un vecteur d'expressions avec déréférenciation, on note $Lab(\mathbf{r})$ l'ensemble des étiquettes apparaissant dans $\mathbf{r}$.

2. **(noeuds).** A chaque identificateur de thread A défini par une équation $A(\mathbf{x}) = P$ on associe un noeud du graphe. On introduit également un nouvel identificateur O et un noeud associé jouant le rôle de *puit*.

3. **(arcs).** On défini une fonction *Call* prenant en entrée un programme et un ensemble fini d'étiquettes et produisant en sortie un ensemble fini de paires composées d'un identificateur de threads et d'un ensemble d'étiquettes. La fonction *Call* est définie par :

$$\begin{array}{lcl}
Call(0, L) & = & \{(O, L)\} \\
Call(\overline{s}e, L) & = & \{(O, L)\} \\
Call(s^y(x).P, A(\mathbf{r}), L) & = & \begin{cases} Call(P, L \cup \{y\}) \cup \{(A, L \cup Lab(\mathbf{r}))\} & \text{si } A \notin Reset \\ Call(P, L \cup \{y\}) \cup \{(O, L \cup Lab(\mathbf{r}))\} & \text{sinon} \end{cases} \\
Call(A(\mathbf{e}), L) & = & \begin{cases} \{(A, L)\} & \text{si } A \notin Reset \\ \{(O, L)\} & \text{sinon} \end{cases} \\
Call([s_1 = s_2] \cdot P_1,\ P_2, L) & = & Call(P_1, L) \cup Call(P_2, L) \\
Call([x = p] \cdot P_1,\ P_2, L) & = & Call(P_1, L) \cup Call(P_2, L) \\
Call(P_1 \mid P_2, L) & = & Call(P_1, L) \cup Call(P_2, L) \\
Call(\nu s\ P, L) & = & Call(P, L)
\end{array}$$

   Supposons que A est un identificateur de threads définie par une équation $A(\mathbf{x}) = P$ et que $C = Call(P, \emptyset)$. On introduit un arc de A vers un identificateur B (éventuellement O) si $(B, L) \in C$. Dans ce cas, l'étiquette de l'arc est l'ensemble $\bigcup\{L \mid (B, L) \in C\}$.

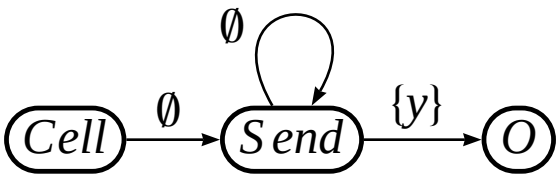
On note $R(A)$ l'union des ensembles d'étiquettes des arcs accessibles de A et $\mathbf{y}_A$ la séquence ordonnée des étiquettes dans $R(A)$. La définition de *Call* est telle que pour toute suite d'appels dans l'exécution d'un thread dans un cycle on peut trouver un chemin correspondant dans le graphe d'appel.

Définition 15 (condition read-once). *Un programme satisfait la condition* read-once *si dans le graphe d'appel il n'y a pas de boucle passant par un arc dont l'étiquette est non vide.*

Remarque 6. *A noter que, bien que le nombre de lectures réalisée dans un même cycle soit borné par une constante, la quantité d'information lue ne l'est pas. Par exemple, un thread* Serveur *peut lire un signal sur lequel est stocké l'ensemble des requêtes produites jusque là et parcourir alors la liste pour répondre à l'ensemble de ces requêtes dans l'instant courant.*

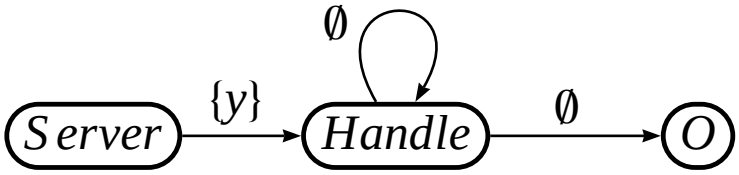
Par la suite, on supposera toujours que les programmes vérifient la condition *read-once*. Pour de tels programmes, on introduit pour chaque identificateur de thread A, avec $\mathbf{x}$ pour paramètres, un nouvel identificateur de thread A^+ dont les paramètres sont ceux de A augmentés des paramètres $\mathbf{y}_A$ pouvant être lus dans un cycle. Le comportement généré par un identificateur de thread dans un cycle peut être décrit par une fonction de ses paramètres $\mathbf{x}$ déterminés au début du cycle et des valeurs $\mathbf{y}_A$ lues sur les signaux pendant le cycle.

Exemple 11. *On considère l'exemple (7) en supposant que Cell marque la fin d'un cycle* ($Reset = \{Cell\}$) *et que l'étiquette associée avec la déréférenciation est y. Le graphe d'appel du programme est :*



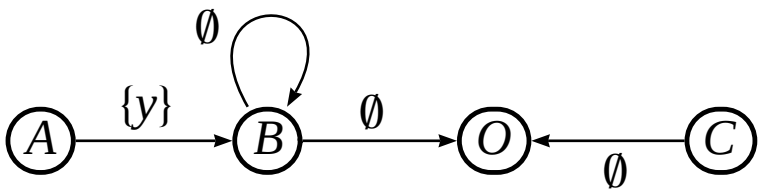
Le programme satisfait la condition read-once *puisque la seule boucle possible, celle de Send vers Send est un d'arc dont l'étiquette est l'ensemble vide. Les identificateurs de threads $Send^+$ et $Cell^+$ ont un paramètre supplémentaire y.*

Exemple 12. *On considère l'exemple (8) en supposant que* $Reset = \{Server\}$ *et que l'étiquette associée avec la déréférenciation est y. Le graphe d'appel du programme est :*



A nouveau, la condition read-once *est satisfaite. L'identificateur $Server^+$ a un paramètre supplémentaire y alors que $Handle^+$ n'en a pas.*

Exemple 13. *Enfin, on considère l'exemple (9) en supposant que* $Reset = \{A, C\}$ *et que l'étiquette associée avec la déréférenciation est y. Le graphe d'appel du programme est :*



Ici aussi, la condition read-once *est satisfaite. L'identificateur de thread A^+ a un paramètre supplémentaire y alors que B^+ et C^+ n'en ont pas.*

4.3.4 Annotations des paramètres

Comme nous l'avons déjà mentionné, un de nos buts est de contrôler la taille des paramètres des threads. Cependant, il est parfois approprié de *négliger* certains de ces paramètres. Par exemple, considérons l'exemple (8). Un des paramètres de l'identificateur *Handle* est une liste l lue sur un signal s dont la taille n'est pas liée à la taille du paramètre s de l'identificateur de thread $Server$. On peut noter que l est utilisée par *Handle* pour réaliser un calcul et que ce paramètres est ensuite négligé à la fin du cycle. On introduit alors un mécanisme permettant de *masquer* des paramètres tels que l. On note $\mathbb{0}$ une nouvelle constante représentant un paramètre de taille 0 (indépendemment du type réel du paramètre). Si h est une fonction d'arité n et si $I \subseteq \{1,\dots,n\}$ est un sous ensemble des positions de ces paramètres, on note

$$h(e_1,\dots,e_n)_I = h(e'_1,\dots,e'_n) \qquad \text{où } e'_i = e_i \text{ si } i \in I \text{ et } e'_i = \mathbb{0} \text{ sinon}$$

Intuitivement, dans $h(e_1,\dots,e_n)_I$ on fixe à 0 tous les arguments dont la position n'est pas dans I. Pour chaque identificateur de thread définissant un comportement d'arité n on suppose donné un ensemble $I_A \subseteq \{1,\dots,n\}$ avec comme condition que si A marque la fin d'un cycle dans le programme ($A \in Reset$) alors $I_A = \{1,\dots,n\}$ (i.e. dans ce cas, aucun paramètre ne peut être fixé à 0).

Remarque 7. *On anticipe sur le fait qu'étant donné un identificateur A l'ensemble I_A sera utilisé pour masquer les paramètres de la signature étendue A^+. En particulier, les paramètres supplémentaires $\mathbf{y}_A$ sont toujours masqués dans $A^+(e_1,\dots,e_n,e_{n+1},\dots,e_{n+m})_{I_A}$, où $n, n+m \geq 0$ sont respectivement l'arité de A et l'arité de A^+, puisque $I_A \subseteq \{1,\dots,n\}$. On verra également un autre cas d'utilisation de cette technique de masquage (concernant l'orientation des flux d'information) dans lequel les paramètres supplémentaires ne sont pas systématiquement masqués.*

4.3.5 Annotations de statut

On associe avec chaque identificateur de thread un *statut* qui est soit *lex* (pour lexicographique) soit *mset* (pour multi-ensemble). On suppose que deux threads équivalents selon un certain pré-ordre $\geq_F$, que l'on définit ci-dessous, ont le même statut et la même arité. En particulier, on note que si A et B sont équivalents selon le pré-ordre, alors A^+ et B^+ ont également la même arité. Pour définir le pré-ordre $\geq_F$ on introduit un *graphe d'appel dans l'instant* en modifiant la définition de la section 4.3.3 de la manière suivante :

$$Call(s(x).P,K,L) = Call(P,L) \qquad Call(A(\mathbf{e}),L) = \{(A,L)\}$$

où le graphe ne contient pas de noeud puit. En particulier, on ignore les appels ne se produisant pas dans l'instant mais on prend en compte les appels dans l'instant concernant des identificateurs de threads marquant la fin d'un cycle. Ainsi, il existe un noeud de A vers B si, $A(\mathbf{x}) = P$ et s'il est possible d'appeler B dans P au cours de l'instant où A est appelé. On construit le plus petit pré-ordre $\geq_F$ sur les identificateurs de threads tel que $A \geq_F B$ s'il existe un arc de A vers B dans le graphe d'appel dans l'instant. On note $A =_F B$ si $A \geq_F B$ et $B \geq_F A$ et $A >_F B$ si $A \geq_F B$ et $A \neq_F B$. Le rang d'un identificateur de threads A, noté $rank(A)$, est la longueur de la plus longue chaine $A >_F B >_F \ldots$.

4.3.6 Annotations des signaux

Le but des annotations de signaux est de rejeter des programmes tels que celui de l'exemple (9) Considérons en particulier le thread A. A chaque instant, ce thread émet sur un signal s les valeurs émises sur ce même signal durant l'instant précédent. On désire rejeter ce type de comportement qui peut faire grandir la taille du système de manière raisonnable ; on souhaite cependant conserver la possibilité, sous certaines conditions, d'accepter des comportements dans lesquels un thread émet sur un signal s une série de valeurs qui dépend de valeurs lues sur un signal s' *différent*. Par exemple, on souhaite pouvoir programmer un *serveur* (cf. exemple (8)) qui reçoit une série de requêtes à la fin d'un instant et produit en réponse une série de valeurs à l'instant suivant. Intuitivement, l'idée est de partitionner l'ensemble des noms de signaux en une collection finie de régions. Les régions sont alors ordonnées et le comportement du serveur décrit précédemment est autorisé si le signal s (réception des requêtes) est associé à une région strictement inférieure à celle des signaux s' (réponses). Formellement, on suppose donné un ensemble infini dénombrable de région $\mathcal{R} = \{\rho_1, \rho_2, \ldots\}$ doté d'un ordre strict $>_{\mathcal{R}}$. On suppose alors que le type d'un signal est raffiné de manière à contenir cette information supplémentaire. c'est à dire que le type d'un signal est maintenant de la forme $Sig_\rho(t)$ ce qui dénote l'appartenance du signal à la région ρ. Dans ce qui suit, on se basera sur ce type d'annotations pour dériver des inégalités garantissant que la taille des valeurs associées à un signal associé à une région ρ peut être bornée par une fonction de la taille des paramètres initiaux du programme et des valeurs lues sur des signaux associés à des régions strictement inférieures. Dans l'exemple (8) on émet sur un signal s' une valeur dépendant d'une valeur lue sur un signal s. Si l'on admet que cette valeur est de taille arbitraire (voir remarque ci-dessous) il est alors nécessaire d'exiger que le signal s soit associé à une région strictement inférieure à la région associée à s'.

Remarque 8. *On anticipe sur la forme des contraintes générées pour faire remarquer que celles-ci ne restreignent le flot d'information que dans le cas de données de taille arbitraire. Par exemple, dans le cas de l'exemple (7), les signaux s (signal d'activation de la cellule) et s'(signal d'activation d'une voisine) sont nécessairement associées (pour des raisons évidentes de typage) à une seule et même région. Le nouvel état d'une cellule next*$(q, !s)$ *est émis par cette cellule sur le signal s'. Si on suppose que l'état d'une cellule est une valeur d'un type énuméré (e.g : un type state contenant les valeurs* alive *et* dead*) les contraintes définies par notre analyse statique permettent d'accepter un tel programme.*

4.4 Analyse statique

4.4.1 Inégalités pour la terminaison des instants

Afin de garantir la terminaison des instants, on produit des contraintes sur les appels récursifs pouvant se produire dans l'instant. Considérons par exemple la définition du thread *Send* de l'exemple

(8). On peut extraire de cette équation une inégalité de la forme :

$$Send(s, q, \ell, \mathsf{cons}(s', \ell'')) > Send(s, q, \ell, \ell'')$$

Si cette contrainte est satisfaite pour un ordre bien fondé approprié alors on sera à même de conclure que la branche `pause` du thread finira par être empruntée lors du calcul. Suivant cette intuition, on définit une fonction C_0 qui extrait du texte du programme un ensemble d'inégalités. Ces inégalités portent l'index 0 correspondant aux contraintes de terminaison. La fonction C_0, définie par cas sur les paires de la forme $(P, A^+(\mathbf{p}))$, est donnée dans la figure 4.1. L'ensemble des contraintes d'index

FIG. 4.1 – Contraintes d'index 0

$$
\begin{array}{lcll}
C_0(P, A^+(\mathbf{p})) & = & \mathsf{case}\ P\ \mathsf{of} & \\
 & 0 & : & \emptyset \\
 & \overline{s}e & : & \emptyset \\
 & [x \trianglerighteq p]\ P_1, P_2 & : & C_0([p/x]P_1, A^+([p/x]\mathbf{p})) \cup C_0(P_2, A^+(\mathbf{p})) \\
 & [s_1 = s_2] \cdot P_1,\ P_2 & : & \bigcup_{i=1,2} C_0(P_i, A^+(\mathbf{p})) \\
 & (P_1 \mid P_2) & : & \bigcup_{i=1,2} C_0(P_i, A^+(\mathbf{p})) \\
 & \nu s\ P' & : & C_0(P', A^+(\mathbf{p})) \\
 & B(\mathbf{e}) & : & \begin{cases} \{A^+(\mathbf{p}) >_0 B^+(\mathbf{e}, \mathbf{y}_B)\} & \text{si } A =_F B \\ \emptyset & \text{sinon} \end{cases} \\
 & s^y(x).P', K & : & C_0([y/x]P', A^+(\mathbf{p}))
\end{array}
$$

0 associées à un programme est obtenu de la manières suivante : pour tout identificateur de thread A défini par une équation $A(\mathbf{x}) = P$, on calcule $C_0(P, A^+(\mathbf{x}, \mathbf{y}_A))$.

Exemple. *En se basant sur la définition des graphes d'appel dans l'instant respectifs, on obtient*

- *pour l'exemple (7) l'ensemble de contraintes :*

$$\{Send^+(s, q, \ell, \mathsf{cons}(s', \ell''), y) >_0 Send^+(s, q, \ell, \ell'', y)\}$$

- *pour l'exemple (8) l'ensemble de contraintes :*

$$\{Handle^+(s, \mathsf{cons}(\mathsf{req}(s', x), \ell')) >_0 Handle^+(s, \ell'))\}$$

- *pour l'exemple (9) l'ensemble de contraintes :*

$$\{B^+(s, \mathsf{cons}(n, \ell')) >_0 B^+(s, \ell')\}$$

4.4.2 Inégalités pour le contrôle de la taille des paramètres au début du cycle

De manière évidente, un programme réagissant en temps polynomial ne devrait pas, par exemple, produire de valeurs de taille exponentielle. Les techniques standard de preuve de terminaison ne permettent pas de garantir cette propriété. Ainsi, une tendance générale consiste à combiner les techniques de preuves de terminaison à des arguments supplémentaires permettant de contrôler la taille

des valeurs calculées. Selon cette tendance, comme première étape, on commence par générer des contraintes d'index 1 dont le but est d'assurer que la taille des paramètres d'un thread au début de chaque cycle est polynomiale en la taille du programme initial. Ces contraintes serviront de base au contrôle de la taille des valeurs calculées par un programme dont on discutera dans la sous-section suivante. Si $\mathbf{r}$ est un vecteur d'expressions avec déréférenciation, on note $\overline{\mathbf{r}}$ le vecteur d'expressions obtenu à partir de $\mathbf{r}$ en remplaçant chaque déréférenciation $!^y s$ par l'étiquette y. La fonction C_1 qui génère les contraintes d'index 1 à partir des équations d'un programme et qui est définie sur les paires de la forme $(P, A^+(\mathbf{p})$ est définie dans la figure 4.2. Les inégalités d'index 1 associées à un programme

FIG. 4.2 – Contraintes d'index 1

$$
\begin{array}{lcll}
C_1(P, A^+(\mathbf{p})) & = & \text{case } P \text{ of} & \\
 & & & \\
 & & 0 & : \quad \emptyset \\
 & & \overline{s}e & : \quad \emptyset \\
 & & [x \trianglerighteq p]\, P_1, P_2 & : \quad C_1([p/x]P_1, A^+([p/x]\mathbf{p})) \cup C_1(P_2, A^+(\mathbf{p})) \\
 & & [s_1 = s_2] \cdot P_1,\ P_2 & : \quad \bigcup_{i=1,2} C_1(P_i, A^+(\mathbf{p})) \\
 & & (P_1 \mid P_2) & : \quad \bigcup_{i=1,2} C_1(P_i, A^+(\mathbf{p})) \\
 & & \nu s\, P' & : \quad C_1(P', A^+(\mathbf{p})) \\
 & & B(\mathbf{e}) & : \quad \{A^+(\mathbf{p})_{I_A} \geq_1 B^+(\mathbf{e}, \mathbf{y}_B)_{I_B}\} \\
 & & s^y(x).P', B(\mathbf{r}) & : \quad C_1([y/x]P', A^+(\mathbf{p})) \cup \{A^+(\mathbf{p})_{I_A} \geq_1 B^+(\overline{\mathbf{r}}, \mathbf{y}_B)_{I_B}\}
\end{array}
$$

sont définies de la même manière que celles d'index 0, i.e. : pour chaque identificateur de thread défini par une équation $A(\mathbf{x}) = P$, on calcule $C_1(P, A^+(\mathbf{x}, \mathbf{y}_A))$.

Exemple. *Pour l'exemple 7, en supposant* $I_{Cell} = \{1, 2, 3\}$ *et* $I_{Send} = \{1, 2, 3, 4\}$ *on obtient :*

$$
\begin{array}{l}
Cell^+(s, q, \ell, 0) \geq_1 Send^+(s, q, \ell, \ell, 0) \\
Send^+(s, q, \ell, \mathsf{cons}(s', \ell''), 0) \geq_1 Send^+(s, q, \ell, \ell'', 0) \\
Send^+(s, q, \ell, \ell', 0) \geq_1 Cell^+(s, next(q, y), \ell, 0)\ .
\end{array}
$$

Pour l'exemple 8, en supposant $I_{Server} = I_{Handle} = \{1\}$ *on obtient :*

$$
\begin{array}{l}
Server^+(s, 0) \geq_1 Handle^+(s, 0) \\
Handle^+(s, 0) \geq_1 Handle^+(s, 0) \\
Handle^+(s, 0) \geq_1 Server^+(s, 0)\ .
\end{array}
$$

Pour l'exemple 9, en supposant $I_A = I_B = I_C = \{1\}$*, on obtient :*

$$
\begin{array}{ll}
A^+(s, 0) \geq_1 B^+(s, 0) & B^+(s, 0) \geq_1 B^+(s, 0) \\
B^+(s, 0) \geq_1 A^+(s, 0) & C^+(s) \geq_1 C^+(s)\ .
\end{array}
$$

4.4.3 Inégalités pour le contrôle le taille des valeurs calculées dans un cycle

Finalement, on introduit les contraintes d'index 2 dont l'objet est de garantir que la taille de chaque valeur calculée durant un cycle est bornée par une fonction (un polynôme) de la taille des paramètres au début du cycle et de la taille des valeurs lues sur les signaux durant ce cycle.

1. Étant donné un thread A, on calcule une sur-approximation de l'ensemble des régions associées avec les émissions réalisées durant un cycle partant de A. Dans ce but, on utilise le graphe d'appel défini précédemment et on calcule l'ensemble des identificateurs de threads accessibles depuis A dans un même cycle. On examine alors la définition de chaque identificateur de thread (diffèrent de O) et on détermine les régions associées avec les émissions apparaissant dans la définition. On note $\mathcal{W}(A)$ cet ensemble.
2. Soit A un identificateur de thread d'arité n associé aux paramètres auxiliaires $\mathbf{y}_A = y_1, \ldots, y_m$. On associe à chaque position $i \in 1, \ldots, m$ de la liste des paramètres auxiliaires une unique région $\gamma(i)$ qui est la région de l'instruction de lecture correspondante.
3. Étant donné un ensemble M de régions, on note

$$M_{Inf} = \{\rho \in \mathcal{R} \mid \forall \rho' \in \mathcal{R}.(\rho' \in M \Rightarrow \rho' >_{\mathcal{R}} \rho)\}$$

   En particulier, on a $\emptyset_{Inf} = \mathcal{R}$ et $\mathcal{R}_{Inf} = \emptyset$. Étant donné un ensemble M de régions, on introduit la notation $A^+(\mathbf{p})_M$ pour $A^+(\mathbf{p})_I$ où $I = \{1, \ldots, n\} \cup \{n + i \mid \gamma(i) \in M\}$. Intuitivement, cela revient à mettre à 0 tous les paramètres auxiliaires dont la région n'est pas dans M. A noter que ce masquage n'affecte que les paramètres auxiliaires des identificateurs de threads.

Les inégalités d'index 2 sont définies comme précédemment en calculant $C_2(P, A^+(\mathbf{x}, \mathbf{y}_A))$ pour toute équation $A(\mathbf{x}) = P$ du programme. La fonction C_2 est définie dans la figure 4.4.3. A noter que les inégalités générées dénotent deux types de contraintes. Premièrement, celles-ci permettent de contrôler la taille d'une valeur émise sur un signal indépendemment des valeurs lues sur des signaux dont la région n'est pas strictement plus petite. Deuxièmement, le même type de restriction porte sur les paramètres d'un appel pouvant générer des émissions sur un signal. En effet, une valeur lue par déréférenciation étant la liste des valeurs émises, il est également nécessaire de contrôler le nombre d'émissions. Sans cette dernière contrainte, la stratification des signaux serait brisée.

Exemple. *Considérons l'exemple 7, en supposant que tous les signaux du programme appartiennent à la même région ρ. Dans ce cas, $\mathcal{W}(Cell) = \mathcal{W}(Send) = \{\rho\}$ et les inégalités associées sont :*

$$\begin{aligned}
&Cell^+(s, q, \ell, y) \geq_2 Send^+(s, q, \ell, \ell, y)\\
&Cell^+(s, q, \ell, 0) \geq_2 Send^+(s, q, \ell, \ell, 0)\\
&Send^+(s, q, \ell, \mathsf{cons}(s', \ell''), y) \geq_2 Send^+(s, q, \ell, \ell'', y)\\
&Send^+(s, q, \ell, \mathsf{cons}(s'\ell''), 0) \geq_2 Send^+(s, q, \ell, \ell'', 0)\\
&Send^+(s, q, \ell, \mathsf{cons}(s'\ell''), 0) \geq_2 q
\end{aligned}$$

Maintenant, considérons l'exemple 8, en supposant que la région ρ du signal sur lequel Server reçoit la requête est plus petite que la région ρ' sur laquelle il retourne la réponse. Dans ce cas, $\mathcal{W}(Server) = \mathcal{W}(Handle) = \{\rho'\}$ et les inégalités associées sont :

$$\begin{aligned}
&Server^+(s, y) \geq_2 Handle^+(s, y)\\
&Handle^+(s, \mathsf{cons}(\mathsf{req}(s', x), \ell')) \geq_2 Handle^+(s, \ell')\\
&Handle^+(s, \mathsf{cons}(\mathsf{req}(s', x), \ell')) \geq_2 f(x)
\end{aligned}$$

Finalement, considérons l'exemples 9. Ici, nous avons un seul signal appartenant, par exemple, à la région ρ. Dans ce cas, $\mathcal{W}(A) = \mathcal{W}(B) = \mathcal{W}(C) = \{\rho\}$ et les inégalités associées sont :

$$\begin{aligned}
&A^+(s, y) \geq_2 B^+(s, y) && A^+(s, 0) \geq_2 B^+(s, y)\\
&B^+(s, \mathsf{cons}(n, \ell')) \geq_2 B(s, \ell') && B^+(s, \mathsf{cons}(n, \ell')) \geq_2 n\\
&C^+(s) \geq_2 C^+(s) && C^+(s) \geq_2 n
\end{aligned}$$

On anticipe sur la suite en notant que l'inégalité $A^+(s, 0) \geq_2 B^+(s, y)$ ne sera pas satisfiable puisque A ne dépend pas de y qui correspond à une liste de taille arbitraire.

$$
\begin{array}{lcl}
C_2(P, A^+(\mathbf{p})) = \text{case } P \text{ of} & & \\
0 & : & \emptyset \\
\overline{s}e & : & \{A^+(\mathbf{p})_{\{\rho\}_{Inf}} \geq_2 e\} \qquad s : Sig_\rho(t) \\
[x = p] \cdot P_1,\ P_2 & : & C_2([p/x]P_1, A^+([p/x]\mathbf{p})) \cup C_2(P_2, A^+(\mathbf{p})) \\
[s_1 = s_2] \cdot P_1,\ P_2 & : & \bigcup_{i=1,2} C_2(P_i, A^+(\mathbf{p})) \\
(P_1 \mid P_2) & : & \bigcup_{i=1,2} C_2(P_i, A^+(\mathbf{p})) \\
\nu s\ P' & : & C_2(P', A^+(\mathbf{p})) \\
B(\mathbf{e}) & : & \begin{cases} \{A^+(\mathbf{p})_{M_{Inf}} \geq_2 B^+(\mathbf{e}, \mathbf{y}_B)_{M_{Inf}} \mid M \subseteq \mathcal{W}(B)\} & B \notin Reset \\ \emptyset & \text{sinon} \end{cases} \\
s^y(x).P', B(\mathbf{r}) & : & C_2([y/x]P', A^+(\mathbf{p})) \\
 & & \cup \\
 & & \begin{cases} \{A^+(\mathbf{p})_{M_{Inf}} \geq_2 B^+(\overline{\mathbf{r}}, \mathbf{y}_B)_{M_{Inf}} \mid M \subseteq \mathcal{W}(B)\} & B \notin Reset \\ \emptyset & \text{sinon} \end{cases}
\end{array}
$$

FIG. 4.3 – Contraintes d'index 2

4.4.4 Quasi-interprétations

Dans cette section on introduit une structure dans laquelle on interprètera les contraintes de la section précédente.

Définition 16. *Une* affectation *q est une fonction qui associe à chaque symbole h d'arité n (un constructeur c, un symbole de fonction f ou un identificateur de thread A^+) d'un programme une fonction notée $q_h : \mathbb{N}^n \to \mathbb{N}$ et satisfait les conditions suivantes :*

1. *si h est un constructeur c et $n = 0$ alors $q_h = 0$*
2. *si h est un constructeur c et $n > 0$ alors*

$$q_h(x_1, \ldots, x_n) = d_\mathsf{c} + \Sigma_{i=1}^n x_i$$

   pour une certaine constante $d_\mathsf{c} \geq 1$.

3. *la fonction q_h vérifie la* propriété du sous-terme*, i.e.*

$$q_h(x_1, \ldots, x_n) \geq x_i,\ i = 1, \ldots, n$$

4. *la fonction q_h est monotone, i.e.*

$$a_j \geq b_j \text{ pour } j = 1, \ldots, n \text{ implique } q_h(a_1, \ldots, a_n) \geq q_h(b_1, \ldots, b_n)$$

5. *Pour tout symbole de fonction f d'arité n on a*

$$f(v_1, \ldots, v_n) \Downarrow v \text{ implique } q_v \leq q_{f(v_1, \ldots, v_n)}.$$

Afin de montrer comment cette structure est utilisée pour l'interprétation des contraintes de la section précédente on introduit d'abord quelques notations. On note E une formule pouvant être

- soit une expression e
- soit l'application d'un identificateur de thread à des expressions, i.e. $A^+(e_1, \ldots, e_n)$.

Soit E une telle formule et soient $x_1, \ldots, x_n$ les variables de E. Étant donnée une affectation q, on associe à E une fonction numérique d'arité n, notée q_E, définie sur la structure de E de la manière suivante :

$$q_{x_i} = x_i \qquad q_{h(e_1,\ldots,e_n)} = q_h(q_{e_1}, \ldots, q_{e_n})$$

Remarque 9. *On note que si v est une valeur alors q_v est une constante numérique. Par convention, un nom de signal sera considéré comme un constructeur d'arité 0.*

Une substitution de base est une substitution associant des valeurs aux variables (en respectant les types). Etant données deux formules E_1 et E_2, on note

$$q \models E_1 > E_2 \text{ (resp. } q \models E_1 \geq E_2)$$

si pour toute substitution de base σ on a : $q_{\sigma E_1} > q_{\sigma E_2}$ (resp. $q_{\sigma E_1} \geq q_{\sigma E_2}$). On comparera également des vecteurs d'expressions formelles.

– Pour la comparaison *lexicographique*, on note

$$q \models E_1, \ldots, E_n >_{lex} E'_1, \ldots, E'_n$$

s'il existe $1 \leq i < n$ tel que $q \models E_j \geq E'_j$ pour tout $j = 1, \ldots, i-1$ et $q \models E_i > E'_i$. Pour la comparaison multi-ensemble, on note

$$q \models E_1, \ldots, E_n >_{mset} E'_1, \ldots, E'_n$$

si pour toute substitution de base σ, on a $\{\!|q_{\sigma E_1}, \ldots q_{\sigma E_n}|\!\} >^{\mathbb{N}}_{mset} \{\!|q_{\sigma E'_1}, \ldots q_{\sigma E'_n}|\!\}$ où $\{\!|...|\!\}$ est notre notation pour les multi-ensembles et $>^{\mathbb{N}}_{mset}$ est l'ordre multi-ensemble sur les entiers naturels.

Remarque 10. *On note les points suivants relatifs à la notion d'affectation :*

1. *Du fait de la condition (1), il existe une constante $k \geq 1$ (pouvant être pris comme la plus grande constante additive d_c) telle que $|v| \leq q_v \leq k.|v|$ pour toute valeur v.*
2. *Du fait de la condition (1), la condition (2) est toujours vérifiée pour les constructeurs.*
3. *La définition d'une affectation assure que pour toute expression close e si $e \Downarrow v$ alors $q_v \leq q_e$.*

On dit qu'une fonction $U : \mathbb{N} \to \mathbb{N}$ borne l'affectation q si pour tout symbole h et pour tout entier naturel n on a

$$q_h(n, \ldots, n) \leq U(n)$$

On dit qu'une affectation q est bornée polynomialement si il existe un polynôme U bornant q. Par la suite, on ne considérera que des affectations polynomialement bornées.

Définition 17. *Une affectation q est une quasi-interprétation d'un programme si et seulement si les conditions suivantes sont vérifiées :*

1. *Pour tout contrainte de la forme $A^+(p_1, \ldots, p_n) >_0 B^+(e_1, \ldots, e_n)$, où $A =_F B$ et st est le statut de A et B, on a :*
$$q \models (p_1, \ldots, p_n) >_{st} (e_1, \ldots, e_n) .$$
2. *Pour tout contrainte de la forme $A^+(p_1, \ldots, p_n) \geq_i B^+(e_1, \ldots, e_m)$ $(i = 1, 2)$ et $A^+(p_1, \ldots, p_n) \geq_2 e$ on a :*
$$q \models A^+ p_1, \ldots, p_n \geq B^+ e_1, \ldots, e_m \text{ et } q \models A^+(p_1, \ldots, p_n) \geq e .$$

Théorème 1. *Un programme admettant une quasi-interprétation polynomial est réactif efficace.*

Afin d'illustrer la notion de quasi-interprétation, on revient sur les exemples de programmes que nous avons déjà étudiés.

Exemple. *Considérons l'exemple 7 et supposons que l'on attribue un statut lexicographique à Cell et Send. On note que Cell* $>_F$ *Send. L'inégalité d'index 0 est satisfaite car l'interprétation de* $\mathsf{cons}(s', \ell'')$ *est toujours strictement plus grande que la quasi-interprétation de* ℓ''*. Pour satisfaire les contraintes d'index* 1 *et* 2 *il suffit d'interpréter Cell*$^+$ *et Send*$^+$ *comme la fonction max, en notant que next*(q, y) *est toujours un état représenté par une constante de taille 0.*

Exemple. *Pour l'exemple 8 on attribue un statut lexicographique aux identificateurs de threads. On note que Handle* $>_F$ *Server. A nouveau, l'inégalité d'index 0 est satisfaite car l'interprétation de* $\mathsf{cons}(\mathsf{req}(s', x), \ell')$ *est toujours plus grande que la quasi-interprétation de* ℓ'*. Pour satisfaire les inégalité d'index* 1, 2 *et* 3 *il suffit de supposer que la quasi-interprétation de Handle*$^+$ *et Server*$^+$ *est une fonction* $g : \mathbb{N}^2 \to \mathbb{N}$ *telle que* $g(0, x)$ *est plus grande, point à point, que la quasi-interprétation de la fonction* f*.*

Exemple. *Finalement, considérons l'exemple 9. On note que* $A >_F B$*. Les inégalités d'indice 0 et 1 peuvent être satisfaite mais comme mentionné précédemment il n'est pas possible de satisfaire l'inégalité* $A^+(s, 0) \geq_2 B^+(s, y)$ *puisque y est une liste de taille arbitraire.*

Chapitre 5

Sémantique annotée

Dans ce chapitre on introduit un certain nombre de définitions techniques nécessaires à la validation de l'analyse statique présentée dans le chapitre précédent. En particulier, on introduit une sémantique de réduction annotée que l'on rattache aux inégalités associées à un programme. Considérons à nouveau la définition du thread *Send* de l'exemple 7 et supposons que l'on souhaite prouver que le thread finira par choisir la branche pause qui assure la fin du calcul pour l'instant courant. Pour ce faire, on aura à comparer l'appel $Send(s, q, \ell, \ell')$ avec l'appel récursif $Send(s, q, \ell, \ell'')$ et cela nous amène à comparer les listes ℓ' et ℓ'' et à noter que $\ell' = \mathsf{cons}(s', \ell'')$. De manière plus générale, il semble utile de conserver une trace de la relation entre les paramètres des identificateurs de threads et les valeurs lues sur les signaux lors des appels récursifs. En suivant ce principe, on revisite la sémantique de réduction présentée dans le chapitre précédent en rendant explicite ce type d'information.

5.1 Sémantique

La sémantique de réduction est maintenant définie (à équivalence structurelle près, voir plus loin) sur des triplets de la forme $(P, A^+(\mathbf{p}), \sigma)$ où :

- P est le programme à exécuter,
- $A^+(\mathbf{p})$ indique que le calcul P provient d'un appel à un identificateur de thread A et que, de plus, la forme des paramètres initiaux et des valeurs lues sur les signaux correspond aux filtres $\mathbf{p}$,
- σ est une substitution finie associant des valeurs aux variables.

On suppose que les variables libres de P qui ne dénotent pas des signaux apparaissent dans les filtres $\mathbf{p}$ et que la substitution σ leur affecte une valeur. L'ensemble des *programmes annotés* est donné par la grammaire suivante :

$$R ::= 0 \mid (P, A^+(\mathbf{p}), \sigma) \mid (R \mid R) \mid \nu s\, R$$

où A est un identificateur de thread défini par une équation du programme. Les définitions du chapitre précédent sont adaptées aux programmes annotés et l'ensemble des développements techniques de ce chapitre se réfèrent à ces nouvelles définitions.

5.1.1 Sémantique de réduction

On définit l'équivalence structurelle $\equiv$ sur les programmes annotés comme la plus petite relation d'équivalence identifiant les programmes égaux à α-renommage prés et satisfaisant les équations

suivantes :

$$0 \equiv (0, A^+(\mathbf{p}), \sigma), \quad R \mid 0 \equiv R, \quad R_1 \mid R_2 \equiv R_2 \mid R_1, \quad R_1 \mid (R_2 \mid R_3) \equiv (R_1 \mid R_2) \mid R_3,$$

$$\nu s\, R_1 \mid R_2 \equiv \nu s\, (R_1 \mid R_2) \quad \text{si } s \notin fn(R_2), \qquad \nu s\, R \equiv R \quad \text{si } s \notin fn(R),$$

$$(\nu s\, P, A^+(\mathbf{p}), \sigma) \equiv \nu s\, (P, A^+(\mathbf{p}), \sigma) \quad s \text{ 'fresh'},$$

$$(P_1 \mid P_2, A^+(\mathbf{p}), \sigma) \equiv (P_1, A^+(\mathbf{p}), \sigma) \mid (P_2, A^+(\mathbf{p}), \sigma)\ .$$

Remarque 11. *Le but des deux dernières équations est simplement de commuter les annotations avec la génération de noms de signaux et la composition parallèle. La condition s 'fresh' signifie que s n'apparaît pas dans les filtres* $\mathbf{p}$ *et la substitution* σ*.*

Relation de réduction. Comme précédemment, la définition de la sémantique repose sur la notion de contexte statique. Ces contextes statiques sont maintenant définis par $C ::= [\ ] \mid \nu s\, C \mid (C \mid R)$. La relation de réduction $\rightarrow$, définie à partir de la relation auxiliaire $\twoheadrightarrow$, est la plus petite relation binaire sur les programmes annotés telle que :

$$\frac{\sigma_1(s_1) = \sigma_2(s_2) \quad \sigma_1 e_1 \Downarrow v_1}{\begin{array}{c}(\overline{s_1} e_1, A_1^+(\mathbf{p}_1), \sigma_1) \mid (s_2^y(x).P_2, K, A_2^+(\mathbf{p}_2), \sigma_2) \twoheadrightarrow \\ (\overline{s_1} e_1, A_1^+(\mathbf{p}_1), \sigma_1) \mid ([y/x]P_2, A_2^+(\mathbf{p}_2), [v_1/y] \circ \sigma_2)\end{array}} \qquad \frac{A'(\mathbf{x}) = P \quad \sigma\mathbf{e} \Downarrow \mathbf{v}}{\begin{array}{c}(A'(\mathbf{e}), A^+(\mathbf{p}), \sigma) \twoheadrightarrow \\ (P, A'^+(\mathbf{x}, \mathbf{y}_{A'}), [\mathbf{v}/\mathbf{x}])\end{array}}$$

$$\frac{\sigma(s_1) = \sigma(s_2)}{\begin{array}{c}([s_1 = s_2]P_1, P_2, A^+(\mathbf{p}), \sigma) \twoheadrightarrow \\ (P_1, A^+(\mathbf{p}), \sigma)\end{array}} \qquad \frac{\sigma(s_1) \neq \sigma(s_2)}{\begin{array}{c}([s_1 = s_2]P_1, P_2, A^+(\mathbf{p}), \sigma) \twoheadrightarrow \\ (P_2, A^+(\mathbf{p}), \sigma)\end{array}}$$

$$\frac{match(\sigma x, p) = \sigma'}{\begin{array}{c}([x \trianglerighteq p]\ P_1, P_2, A^+(\mathbf{p}), \sigma) \twoheadrightarrow \\ ([p/x]P_1, A^+([p/x]\mathbf{p}), \sigma' \circ \sigma)\end{array}} \qquad \frac{match(\sigma x, p) \uparrow}{\begin{array}{c}([x = p] \cdot P_1,\ P_2, A^+(\mathbf{p}), \sigma) \twoheadrightarrow \\ (P_2, A^+(\mathbf{p}), \sigma)\end{array}}$$

$$\frac{R \equiv C[R_1] \qquad R_1 \twoheadrightarrow R_2 \qquad R_2 \equiv C[R']}{C[R] \rightarrow C[R']}$$

Exemple. *Pour illustrer les annotations de la sémantique, considérons le programme*

$$A(s_1, [s_3; s_4]) \mid B(s_1, s_2, \mathtt{nil})$$

où A et B sont définis par les équations suivantes :

$$A(s, l) = [l \trianglerighteq \mathtt{cons}(x, l')]\ (\overline{s}x \mid A(s, l')), \mathtt{pause}.A(s, [s_3; s_4])$$

$$B(s, s', x) = \overline{s'}x \mid \mathtt{pause}.B(s, s', !s)$$

Intuitivement, à chaque instant, A émet les noms s_3 *et* s_4 *sur* s_1 *et B récupère la liste correspondante, à la fin de l'instant, pour l'émettre sur* s_2 *à l'instant suivant. Après la réduction des appels* $A(s_1, [s_3; s_4])$

et $B(s_1, s_2, \texttt{nil})$ *et la communication sur* s_1*, le programme a la forme*

$$\begin{array}{ll} & (\overline{s}x, A^+(s, \texttt{cons}(x, l')), \{s \mapsto s_1, x \mapsto s_3, l' \mapsto s_4\}) \\ | & (\overline{s}x, A^+(s, \texttt{cons}(x, l')), \{s \mapsto s_1, x \mapsto s_4, l' \mapsto \texttt{nil}\}) \\ | & (\texttt{pause}.A(s, [s_3; s_4]), A^+(s, l), \{s \mapsto s_1, l \mapsto \texttt{nil}\}) \\ | & (\overline{s'}x, B^+(s, s', x), \{s \mapsto s_1, s' \mapsto s_2, x \mapsto \texttt{nil}\}) \\ | & (\texttt{pause}.B(s, s', !s), B^+(s, s', x), \{s \mapsto s_1, s' \mapsto s_2, x \mapsto \texttt{nil}\}) \end{array}$$

Suspension et évaluation à la fin de l'instant. On note $R \downarrow$ si $\neg\exists R'$ $(R \to R')$ et on dit que le programme (annoté) R est *suspendu*. Un programme suspendu R est structurellement équivalent à un programme :

$$\nu\mathbf{s}(S \mid In) \tag{5.1}$$

où les noms de signaux $\mathbf{s}$ sont tous distincts,

$$S \equiv (\overline{s_1}e_1, A_1^+(\mathbf{p}_1), \sigma_1) \mid \cdots \mid (\overline{s_n}e_n, A_n^+(\mathbf{p}_n), \sigma_n), \qquad \sigma_i e_i \Downarrow v_i \text{ pour } i = 1, \ldots, n,$$

$$In \equiv (t_1(x_1).P_1, B_1(\mathbf{r_1}), C_1^+(\mathbf{p}'_1), \sigma'_1) \mid \cdots \mid (t_m(x_m).P_m, B_m(\mathbf{r_m}), C_m^+(\mathbf{p}'_m), \sigma'_m),$$

et $n, m \geq 0$ (par convention une composition parallèle vide est égale au programme 0). Le calcul de la fin d'instant suit les mêmes étapes que celles de la sémantique du chapitre précédent. On commence par introduire un certain nombre de notations.

1. On note $\hat{S}$ la composition parallèle des émissions $\overline{s'}v$ telle que $(\overline{s}e, A^+(\mathbf{p}), \sigma)$ apparaît dans la composition parallèle S, $\sigma(s) = s'$, et $\sigma(e) \Downarrow v$.
2. En supposant $V \Vdash \hat{S}$ et $V(\sigma'_i \mathbf{r}_i) \Downarrow \mathbf{v}_i$, on définit pour $i = 1, \ldots, m$:

$$Eval(B_i(\mathbf{r}_i), \sigma'_i, V) = (B_i(\mathbf{x}_i), B_i^+(\mathbf{x}_i, \mathbf{y}_{B_i}), [\mathbf{v}_i/\mathbf{x}_i]) \,.$$

3. On définit également $Eval(In, V)$ comme la composition parallèle de $Eval(B_i(\mathbf{r}_i), \sigma'_i, V)$ pour $i = 1, \ldots, m$.

Avec ces conventions, la règle d'évaluation à la fin de l'instant est donnée par :

$$\frac{R \downarrow \quad R \equiv \nu\mathbf{s}\,(S \mid In) \quad V \Vdash \hat{S} \quad \{\mathbf{s}'\} = \{\mathbf{s}\} \backslash Free(\nu\mathbf{s}\,\hat{S}) \quad R' \equiv \nu\mathbf{s}' Eval(In, V)}{R \mapsto R'}$$

Exemple. *Pour illustrer l'utilisation des annotations dans le calcul de la fin d'instant, on revient sur le programme obtenu précédemment après réduction des appels et réalisation de la communication. Le programme obtenu est bien suspendu, et on a pour In le programme suivant :*

$$\begin{array}{ll} & (\texttt{pause}.A(s, [s_3; s_4]), A^+(s, l), \{s \mapsto s_1, l \mapsto \texttt{nil}\}) \\ | & (\texttt{pause}.B(s, s', !s), B^+(s, s', x), \{s \mapsto s_1, s' \mapsto s_2, x \mapsto \texttt{nil}\}) \end{array}$$

qui devient, par exemple, à l'instant suivant :

$$\begin{array}{l} (A(s, l), A^+(s, l), \{s \mapsto s_1, l \mapsto [s_3; s_4]\}) \\ (B(s, s', x), B^+(s, s', x), \{s \mapsto s_1, s' \mapsto s_2, x \mapsto [s_4; s_3]\}) \end{array}$$

Calculs (avec annotations). Comme dans la sémantique originale, un programme reçoit au début de chaque instant une entrée que l'on représente par un nouvel identificateur de threads *Env* défini par une équation :

$$Env() = \overline{s_1}v_1 \mid \ldots \mid \overline{s_n}v_n$$

On écrit alors, pour un calcul, :

$$R \overset{Env}{\mapsto} R' \text{ si } R \mapsto R'' \text{ et } R' = (R'' \mid (Env, Env^+, \sigma_\emptyset))$$

où $\sigma_\emptyset$ dénote la substitution vide. Suivant la convention fixée dans le chapitre précédent, on suppose que dans une entrée toutes les valeurs émises sur un signal s sont distinctes.

Définition 18. *Un* calcul *d'un programme R est une suite dénombrable de programmes* $R_1, R_2, \ldots$ *telle que :*

$$R \equiv R_{i_0+1} \overset{*}{\rightarrow} R_{i_1} \overset{Env_1}{\mapsto} R_{i_1+1} \overset{*}{\rightarrow} R_{i_2} \overset{Env_2}{\mapsto} R_{i_2+1} \ldots$$

On suppose que, initialement, un programme a la forme :

$$\nu\mathbf{s}\ ((A_1(\mathbf{x}_1), A_1^+(\mathbf{x}_1, \mathbf{y}_{A_1}), [\mathbf{v}_1/\mathbf{x_1}]) \mid \ldots \mid (A_n(\mathbf{x}_n), A_n^+(\mathbf{x}_n, \mathbf{y}_{A_n}), [\mathbf{v}_n/\mathbf{x_n}])) \tag{5.2}$$

Par définition de l'instruction *present* et des entrées un programme a cette forme (à équivalence structurelle près) au début de chaque instant.

Suppression des annotations. Comme nous l'avons déjà suggéré, la sémantique présentée ici est une version annotée de la sémantique du chapitre précédent, en particulier il est facile de vérifier que :

1. Étant donné un programme annoté *R*, on peut construire un programme *P* en remplaçant chaque triplet $(P, A^+(\mathbf{p}), \sigma)$ dans *R* par σP. Intuitivement, cela revient simplement à oublier les annotations du programme. De plus, les calculs se projettent. En particulier, la taille d'un programme annoté est simplement définie comme étant la taille du programme obtenu par cette traduction.
2. Étant donné un calcul $P_1, P_2, \ldots$ d'un programme (non annoté) $P = \nu\mathbf{s}\ (A_1(\mathbf{v}_1) \mid \ldots \mid A_n(\mathbf{v}_n))$ on peut trouver un calcul $R_1, R_2, \ldots$ du programme (annoté)

   $$R = \nu\mathbf{s}\ ((A_1(\mathbf{x}_1), A_1^+(\mathbf{x}_1, \mathbf{y}_{A_1}), [\mathbf{v}_1/\mathbf{x_1}]) \mid \ldots \mid (A_n(\mathbf{x}_n), A_n^+(\mathbf{x}_n, \mathbf{y}_{A_n}), [\mathbf{v}_n/\mathbf{x_n}]))$$

   tel que pour tout $i \geq 0$ chaque P_i est le programme obtenu à partir de R_i par traduction et tel que $R_i \downarrow$ si et seulement si $P_i \downarrow$. En particulier, pour prouver que *P* est réactive efficace, il suffit de prouver que *R* est réactif efficace.

5.1.2 Inégalités et sémantique annotée

Intuitivement, lorsqu'un programme évolue en un programme de la forme $C[(P, A^+(\mathbf{p}), \sigma)]$ pour un certain contexte *C* et un certain programme annoté $(P, A^+(\mathbf{p}), \sigma)$, on se base sur les inégalités associées avec les équations de ce programme pour fournir certaines garanties sur les calculs possibles de σP. Par exemple, si le programme décrivant le comportement d'un automate cellulaire évolue en

$$(Send(s, q, \ell, \ell'), Send^+(s, q, \ell, \mathsf{cons}(\mathsf{s}', \ell''), \mathsf{y}), \sigma)$$

pour une certaine substitution σ alors, pour des considérations de terminaison, on devrait être en mesure de conclure que

$$Send^+(s, q, \ell, \mathsf{cons}(\mathsf{s}', \ell''), \mathsf{y}) >_0 \mathsf{Send}(\mathsf{s}, \mathsf{q}, \ell, \ell'', \mathsf{y})$$

est *effectivement* une inégalité associée avec les équations du programmes.

Définition 19. *Étant donné un programme R admettant une quasi-interprétation q et un calcul*

$$R \equiv R_{i_0+1} \overset{*}{\rightarrow} R_{i_1} \overset{Env_1}{\mapsto} R_{i_1+1} \overset{*}{\rightarrow} R_{i_2} \overset{Env_2}{\mapsto} R_{i_2+1} \overset{*}{\rightarrow} \dots$$

de R, on dit que le couple $(P, A^+(\mathbf{p}))$ *est un point de contrôle si :*

- *Pour toute contrainte de la forme :* $A^+(p_1, \dots, p_n) >_0 B^+(e_1, \dots, e_n) \in C_0(P, A^+(\mathbf{p}))$ *on a :*

$$q \models p_1, \dots, p_n >_{st} e_1, \dots, e_n \qquad \text{où st est le statut de A et B}$$

- *Pour toute contrainte de la forme* $A^+(p_1, \dots, p_n) \geq_i B^+(e_1, \dots, e_m) \in C_i(P, A^+(\mathbf{p}))$ $(i = 1, 2)$ *et pour toute contrainte de la forme* $A^+(p_1, \dots, p_n) \geq_2 e \in C_2(P, A^+(\mathbf{p}))$ *on a :*

$$q \models A^+(p_1, \dots, p_n) \geq B^+(e_1, \dots, e_m) \qquad et \qquad q \models A^+(p_1, \dots, p_n) \geq e$$

Un programme R est une instance des points de contrôle si dès que $R \equiv C[(P, A^+(\mathbf{p}), \sigma)]$*, où A est un identificateur de thread,* $(P, A^+(\mathbf{p}))$ *est un point de contrôle.*

Lemme 1. *Soit R un programme admettant une quasi-interprétation q. Pour tout calcul* $R_1, R_2, \dots$ *de R et pour tout* $k \geq 1$ *si* $R' \equiv R_k$ *alors* R' *est une instance des points de contrôle.*

5.2 Résultats

Dans cette section, on introduit un certain nombre de résultats auxiliaires qui permettent de prouver qu'un programme admettant une quasi-interprétation polynômiale est réactif efficace. La présentation du résultat suit le schéma des contraintes introduites dans le chapitre précédent et repose sur la sémantique annotée.

Terminaison des instants

Notre premier résultat stipule, à partir des inégalités d'index 0, que le calcul d'un instant est polynomial en la taille des valeurs calculées durant cet instant.

Proposition 1. *Soit R un programme admettant une quasi-interprétation polynomiale. Il existe un polynôme* Q_0 *tel que, pour tout calcul de R, si c borne la taille des valeurs calculées durant l'instant k alors le programme (du début de l'instant k) se suspend en temps au plus* $Q_0(c)$*.*

Débuts de cycle et systèmes *non-size increasing*

Intuitivement, au cours d'un cycle les tailles des valeurs calculées par un thread dépendent des tailles des paramètres de ce thread au début du cycle et des valeurs lues par ce thread au cours du cycle. Les contraintes d'index 1 présentées dans le chapitre précédent ont pour but d'assurer que les paramètres des threads au début de leurs cycles sont *non-size increasing* et en particulier ne dépendent (de manière polynomiale) que de la taille du programme initial.

Proposition 2. *Soit R un programme admettant une quasi-interprétation polynomiale. Il existe un polynôme* Q_1 *tel que si c borne la taille de R et si* $A \in Reset$ *alors, dans tout calcul de R, les tailles des paramètres dans tout appel de A sont bornés par* $Q_1(c)$*.*

Bornes sur la taille des valeurs calculées

Le résultat précédent permet effectivement de calculer une borne sur la taille des paramètres des threads au début de leurs cycles. Les contraintes d'index 2 nous permettent d'assurer que la taille des valeurs calculées lors d'un cycle ne dépendent (de manière polynomiale) que des paramètres de ce thread au début du cycle et des valeurs lues sur les signaux par ce thread durant le cycle. Cela est formalisé par la proposition suivante :

Proposition 3. *Soit R un programme admettant une quasi-interprétation polynomiale. il existe un polynôme Q_2 tel que dans tout calcul*

$$R \equiv R_{i_0+1} \xrightarrow{*} R_{i_1} \overset{Env_1}{\mapsto} R_{i_1+1} \xrightarrow{*} R_{i_2} \overset{Env_2}{\mapsto} R_{i_2+1} \dots$$

si d borne la taille de R et des entrées $Env_1, \dots, Env_k$ pour $k \geq 0$ alors les tailles des valeurs calculées durant l'instant k sont bornées par $Q_2(d)$.

En particulier, en combinaison avec la proposition précédente, ce résultat nous permet d'assurer que la taille de toute valeur calculée par un programme est polynomiale en la taille du programme initial et des valeurs lues sur les signaux. Ce résultat repose fortement sur la stratification des régions qui empêche un groupe de threads de faire grossir la taille d'une valeur au travers d'un nombre arbitraire de communications.

Réactivité efficace

Les preuves des trois propositions que l'on vient d'introduire seront données dans le chapitre suivant. On verra également que la preuve du théorème 1 découle directement de ces trois propositions.

Remarque 12. *On peut noter que la définition de la taille d'un programme considérée ignore la génération de noms de signaux. Il semble cependant légitime de s'interroger sur les implications de celle-ci au niveau d'une implantation du $S\pi$-calcul. Par exemple, quelle est l'occupation mémoire d'un programme tel que*

$$A(s_0) \qquad \text{où } A(s) = \nu s\, \texttt{pause}.A(s)$$

Selon la sémantique, après un nombre arbitraire d'instants le programme, de la forme $\nu s \dots \nu s\, A(s)$, implique l'imbrication d'un nombre arbitraire d'opérateurs new. Deux remarques peuvent être faites à ce sujet. L'opérateur new peut être implémenté à la manière d'une fonction gensym retournant un nouveau nom de signal et terminant. Dans le cas d'une implantation utilisant par exemple des arbres pour représenter les programmes, le nombre de new imbriqués influe sur la taille du programme. Cependant, grâce à la règle d'équivalence $\nu s\, R \equiv R$ si $s \notin fn(R)$ il est possible de raisonner en supposant la présence d'un garbage collector. *En effet, il est tout à fait légal de supprimer lors de l'exécution les générations de noms de signaux non utilisées par le programme. Dans ce cas une notion de taille prenant en compte la génération des noms de signaux est* équivalente *à celle proposée ici puisque le nombre de noms de signaux utilisés par le programme ne dépend que de la pile, i.e des appels qui sont pris en compte par notre mesure.*

Chapitre 6

Preuves

Afin de simplifier la présentation des résultats, on supposera toujours que les instants commencent par le développement du premier appel de chaque thread et de l'entrée. On définit la transformation $Eval'$ par $Eval'(B(\mathbf{r}), \sigma, V) = (P, B^+(\mathbf{x}, \mathbf{y}_B), [\mathbf{v}/\mathbf{x}])$ si et seulement si $B(\mathbf{x}) = P$ et $Eval(B(\mathbf{r}), \sigma, V) = (B(\mathbf{x}), B^+(\mathbf{x}, \mathbf{y}_B), [\mathbf{v}/\mathbf{x}])$. Par la suite, on notera simplement $Eval$ pour $Eval'$. De la même manière, pour tout $k \geq 1$, si $Env_k = \overline{s_1}v_1 \mid \ldots \mid \overline{s_n}v_n$, R est le programme à la fin de l'instant $k-1$ et $R \mapsto R'$ alors le programme au début de l'instant k est $R' \mid (\overline{s_1}v_1 \mid \ldots \mid \overline{s_n}v_n, Env_k^+, \sigma_\emptyset)$. Par un simple examen de la sémantique, il est facile de se convaincre que ce choix n'a aucune influence sur les propriétés considérées. Dans la suite, on considérera implicitement cette sémantique modifiée. On appelera identificateur de thread un symbole A qui ne dénote pas une entrée. Si σ est une substitution, on notera σ^+ la substitution définie par $\sigma^+(x) = \sigma(x)$ si $\sigma(x)$ est définie et $\mathbb{0}$ sinon. Etant données deux substitutions σ et σ', on note $\sigma \subseteq \sigma'$ si $Dom(\sigma) \subseteq Dom(\sigma')$ et $\sigma(x) = \sigma'(x)$ pour tout $x \in Dom(\sigma)$.

Avant de prouver les résultats du chapitre précédent, on introduit un résultat auxiliaire sur les ordres lexicographique et multi-ensemble qui sera utile pour établir une borne polynômiale sur le nombre de réductions possibles au cours d'un instant. Pour $n \geq 0$ et $a_0, a_1, \ldots, a_{n-1}, c \in \mathbb{N}$, on note

$$B_{a_{n-1},\ldots,a_0}(c)(lex) = \sum_{i=1,\ldots,n} a_i.x^i \qquad B_{a_{n-1},\ldots,a_0}(c)(mul) = \sum_{i=1,\ldots,n} a_{\sigma(i)}.x^i$$

où σ est une permutation sur $\{0, 1, \ldots, n-1\}$ telle que $a_{\sigma(i+1)} \geq a_{\sigma(i)}$ pour $i = 0, 1, \ldots, n-1$. Si q est une quasi-interprétation (donnée par le contexte), $st \in \{lex, mul\}$ et $v_1, \ldots, v_n$ sont des valeurs, on notera simplement $B_{v_1,\ldots,v_n}(c)(st)$ pour $B_{q_{v_1},\ldots,q_{v_n}}(c)(st)$.

Lemme (2). *Pour tout $n \geq 0$, si $\mathbf{a}$ et $\mathbf{b}$ sont des n-uplets d'éléments de $\mathbb{N}$ bornés par $c \in \mathbb{N}$ et si $st \in \{lex, mset\}$ alors $B_{\mathbf{a}}(c)(st), B_{\mathbf{b}}(c)(st) \leq c^n$ et $(\mathbf{a} >_{st} \mathbf{b} \Rightarrow B_{\mathbf{a}}(c)(st) > B_{\mathbf{b}}(c)(st))$.*

Preuve du lemme 2

Si $st = lex$, le résultat est immédiat. On note simplement que les deux n-uplets peuvent être vus comme les coefficients de deux entiers naturels exprimés en base c. Si $st = mul$, alors le résultat est immédiat pour $n = 0$. Soient $(a_{n-1}, \ldots, a_0)$ et $(b_{n-1}, \ldots, b_0)$ deux n-uplets d'entiers bornés par une constante c. Pour $n \geq 1$, on prouve, par induction sur n, que $\{\!|a_{n-1}, \ldots, a_0|\!\} >_{mul} \{\!|b_{n-1}, \ldots, b_0|\!\}$ implique $a_{\sigma(n-1)}, \ldots, a_{\sigma(0)} >_{lex} b_{\sigma'(n-1)}, \ldots, b_{\sigma'(0)}$ où σ et σ' sont deux permutations telles que $a_{\sigma(i+1)} \geq a_{\sigma(i)}$ et $b_{\sigma'(i+1)} \geq b_{\sigma'(i)}$ pour $i = 0, \ldots, n-1$ et où $\{\!|\ |\!\}$ est notre notation pour les multi-ensembles. Si $n = 1$ alors le résultat est immédiat puisque $\{\!|a_{n-1}|\!\} >_{mul} \{\!|b_{n-1}|\!\}$ si et seulement si $a_{n-1} > b_{n-1}$ (i.e. $(a_{n-1}) >_{lex} (b_{n-1})$). Si $n > 1$, par définition de σ et de $>_{mul}$ il existe $j \in \{0, \ldots, n-1\}$ tel que

$a_{\sigma(n-1)} \geq a_j \geq b_{\sigma(n-1)}$. On distingue deux cas. Si $a_{\sigma(n-1)} > b_{\sigma'(n-1)}$ alors, par définition de $>_{lex}$, il est immédiat que $(a_{\sigma(n-1)}, \ldots, a_{\sigma(0)}) >_{lex} (b_{\sigma'(n-1)}, \ldots, b_{\sigma'(0)})$. Si $a_{\sigma(n-1)} = b_{\sigma'(n-1)}$, alors on note simplement que $\{\!|a_{\sigma(n-2)}, \ldots, a_{\sigma(0)}|\!\} >_{mul} \{\!|b_{\sigma(n-2)}, \ldots, b_{\sigma(0)}|\!\}$ et on conclut par hypothèse d'induction.

Preuve du lemme 1

Lemme. *Soit R un programme admettant une quasi-interprétation. Pour tout calcul $R_1, R_2, \ldots$ de R et pour tout $i \geq 0$, le programme R_i est une instance des points de contrôle.*

Premièrement, on remarque qu'être une instance des points de contrôle est invariant par équivalence structurelle. Ce résultat est immédiat par définition de la relation $\equiv$, en notant que $(0, A^+(\mathbf{p}))$ est un point de contrôle ($C_i(0, A^+(\mathbf{p})) = \emptyset$ pour $i = 0, 1, 2$) et qu'être une instance des points de contrôle est invariant par α-renommage (q identifie les noms de signaux).

Soient R un programme admettant une quasi-interprétation q et $R_1, R_2, \ldots$ un calcul de R. On prouve que, pour tout $i \geq 0$, R_i est une instance des points de contrôle. La preuve est par induction sur i. Par définition d'une quasi-interprétation, il est évidement que pour $k \geq 0$, le programme R_{i_k+1} (le programme au début de l'instant k) est une instance des points de contrôle (R_{i_k+1} est une composition parallèle de programmes de la forme $(P, A^+(\mathbf{p}), [\mathbf{v}/\mathbf{x}])$ où $A(\mathbf{x}) = P$). Pour prouver le résultat il est donc suffisant de vérifier que la propriété est conservée par la relation de réduction dans l'instant. Supposons que $R \to R'$ et que R est une instance des points de contrôle. Par définition de la sémantique, il existe un contexte C et des programmes R_0 et R'_0 tels que $R \equiv C[R_0]$, $R' \equiv C[R'_0]$ et $R_0 \twoheadrightarrow R'_0$. Pour prouver que R' est une instance des points de contrôle, il est suffisant de prouver que $(P, A^+(\mathbf{p}))$ est un point de contrôle pour tout $(P, A^+(\mathbf{p}), \sigma)$ dans R'_0. On procède par cas sur la règle utilisée pour la réduction en notant que si $(P, A^+(\mathbf{p}))$ est un point de contrôle et si $C_i(P', A'^+(\mathbf{p}')) \subseteq C_i(P, A^+(\mathbf{p}))$ pour $i = 0, 1, 2$ alors $(P', A'^+(\mathbf{p}))$ est un point de contrôle.

- Si la réduction est une lecture, i.e.

$$R_0 = R_1 \mid (s_2^y(x).P, K, A_2^+(\mathbf{p}_2), \sigma_2) \qquad R'_0 = R_1 \mid ([y/x]P, A_2^+(\mathbf{p}_2), [v/y] \circ \sigma_2)$$

  où $R_1 = (\overline{s_1}e, A_1^+(\mathbf{p}_1), \sigma_1)$, $\sigma_1 s_1 = \sigma_2 s_2$ et $\sigma_1 e \Downarrow v$. Par hypothèse d'induction, il suffit de prouver que $([y/x]P, A_2^+(\mathbf{p}_2))$ est un point de contrôle, ce qui est immédiat puisque $C_i([y/x]P, A_2^+(\mathbf{p}_2)) \subseteq C_i(s_2^y(x).P, K, A_2^+(\mathbf{p}_2))$ pour $i = 0, 1, 2$.
- Si la réduction est un appel, i.e.

$$R_0 = (A_1(\mathbf{e}), A_0^+(\mathbf{p}), \sigma) \qquad R'_0 = (P, A_1^+(\mathbf{x}, \mathbf{y}_{A_1}), [\mathbf{v}/\mathbf{x}])$$

  où $A_1(\mathbf{x}) = P$ et $\sigma\mathbf{e} \Downarrow \mathbf{v}$. Puisque $A_1(\mathbf{x}) = P$ est une équation du programme, le résultat est immédiat par définition d'une quasi-interprétation.
- Si la réduction est le filtrage réussi d'une valeur, i.e.

$$R_0 = ([x \trianglerighteq p]\ P_1, P_2, A^+(\mathbf{p}), \sigma) \qquad R'_0 = ([p/x]P_1, A^+([p/x]\mathbf{p}), \sigma' \circ \sigma)$$

  où $\sigma' = match(\sigma x, p)$. Il suffit de montrer que $([p/x]P_1, A^+([p/x]\mathbf{p}))$ est un point de contrôle, ce qui est immédiat puisque $C_i([p/x]P_1, A^+([p/x]\mathbf{p})) \subseteq C_i([x \trianglerighteq p]\ P_1, P_2, A^+(\mathbf{p}))$ pour $i = 0, 1, 2$. Les autres cas sont similaires.

Preuve de la proposition 1

Proposition. *soit R un programme admettant une quasi-interprétation polynômiale. Il existe un polynôme Q_0 tel que, pour tout calcul de R, si c borne la taille de R et des valeurs calculées durant l'instant k alors le programme (au début de l'instant k) se suspend en temps au plus $Q_0(c)$.*

Soit R un programme admettant une quasi-interprétation et soit u un entier naturel tel que $|v| \leq q_v \leq u.|v|$ pour toute valeur v. On note $|.|$ la fonction sur les programmes (non annotés) définie par :

$$|0| = |\overline{s}e| = |A(\mathbf{e})| = 0 \qquad |[x \trianglerighteq p]\ P_1, P_2| = |[s_1 = s_2] \cdot P_1,\ P_2| = 1 + max\{|P_1|, |P_2|\}$$
$$|s(x).P, K| = 1 + |P| \qquad |P_1 \mid P_2| = |P_1| + |P_2| \qquad |\nu s\ P| = |P|$$

On note K pour $max\{|P| \mid A(\mathbf{x}) = P\}$ et on note respectivement r et a le rang maximal et l'arité maximales des équations du programme. Soient $R \equiv R_{i_0+1} \overset{*}{\rightarrow} R_{i_1} \overset{Env_1}{\mapsto} R_{i_1+1} \overset{*}{\rightarrow} R_{i_2} \overset{Env_2}{\mapsto} R_{i_2+1} \overset{*}{\rightarrow} \ldots$ un calcul de R, $k \geq 0$ et c un entier naturel tels que c borne la taille des valeurs calculées au cours de l'instant k. On prouve que R_{i_k+1} (le programme au début de l'instant k) se suspend en temps au plus $Q_0(c)$ où Q_0 est le polynôme défini par

$$Q_0(x) = x.(K+1).r.(u.x+1)^a$$

On note $m = i_k + 1, i_k + 2, \ldots$, la suite maximale telle que $\neg(R_{i_k+1} \downarrow), \neg(R_{i_k+2} \downarrow), \ldots$ et on prouve que m est une suite finie de longueur bornée par $Q_0(c)$. Pour cela, on procède en trois étapes :

(a) on définit une fonction μ des programmes (annotés) vers les entiers naturels,

(b) on prouve que $\mu(R_{i_k+1}) \leq Q_0(c)$,

(c) on prouve que pour tout $k' \in m$ l'inégalité $\mu(R_{k'}) > \mu(R_{k'+1})$ est vérifiée.

(a) On note $b = u.c + 1$ et $(\!|A(\mathbf{v})|\!) = (K+1)((rank(A) - 1).b^a + B_{q_{v_1},\ldots,q_{v_n}}(b)(st)$ où st est le statut de A. La fonction μ est définie, en posant $\mu(P, Env_k^+, \sigma_\emptyset) = 0$ pour tout P par convention, de la manière suivante :

$$\mu(0) = 0 \qquad \mu(R \mid R') = \mu(R) + \mu(R') \qquad \mu(\nu s\ R) = \mu(R)$$

$$\mu(P, A^+(p_1, \ldots, p_{n+m}), \sigma) = tc(P).(\!|A(\sigma p_1, \ldots, \sigma p_n)|\!) + |P|, \text{ où } n \text{ est l'arité de } A$$

Il est facile de vérifier que μ identifie les programmes structurellement équivalents et que $\mu(R_{k'})$ est définie sur $\mathbb{N}$ pour tout $k' \in m$. Par hypothèse sur la forme des équations d'un programme et par définition de K, si $A(\mathbf{x}) = P$, alors

$$\forall \mathbf{v}.(\mu(P, A^+(\mathbf{x}, \mathbf{y}_A), [\mathbf{v}/\mathbf{x}]) \leq (\!|A(\mathbf{v})|\!) + K) \tag{6.1}$$

(b) Immédiat, par la forme d'un programme au début de l'instant et par (6.1), en notant que le nombre de threads dans toute configuration accessible est bornée par c.
(c) Par hypothèse, $\neg(R_{k'} \downarrow)$ et il existe C_0, R_ε et R' tels que $R_{k'} \equiv C_0[R_\varepsilon]$, $R_{k'+1} \equiv C_0[R']$ et $R_\varepsilon \rightarrowtriangle R'$. Il est facile de définir une fonction μ' des contextes vers les entiers naturels telle que $\mu(C_1[R_1]) = \mu'(C_1) + \mu(R_1)$ pour tout C_1 et R_1. Puisque μ identifie les programmes structurellement équivalents, il est suffisant de montrer que $\mu(R_\varepsilon) > \mu(R')$. On procède par cas sur la règle dont la réduction $R_\varepsilon \rightarrowtriangle R'$ est une instance.

- Supposons que la réduction soit une lecture, i.e.

$$R_\varepsilon = (s_2^y(x).P_2, K, A_2^+(\mathbf{p}_2), \sigma_2) \mid R'' \qquad R' = ([y/x]P_2, A_2^+(\mathbf{p}_2), [v_1/y] \circ \sigma_2) \mid R''$$

où $R'' = (\overline{s_1}e_1, A_1^+(\mathbf{p}_1), \sigma_1)$, $\sigma_1 s_1 = \sigma_2 s_2$ et $\sigma_1 e_1 \Downarrow v_1$. Par définition de μ, il suffit de vérifier que la mesure du membre gauche de la composition parallèle de R_ε est strictement plus grande que celle du membre gauche de la composition parallèle de R'. On conlut en notant que, si n est l'arité de A_2 et $\mathbf{p}_2 = p_1, \ldots, p_{n+m}$ alors $\sigma_2(p_1, \ldots, p_n) = ([v_1/y] \circ \sigma_2)(p_1, \ldots, p_n)$, $tc(s_2^y(x).P_2, K) \geq tc([y/x]P_2)$ et $|s_2^y(x).P_2, K| > |[y/x]P_2|$.
- Supposons que la réduction soit un appel, i.e.

$$R_\varepsilon = (A_1(\mathbf{e}_1), A_0^+(\mathbf{p}_0), \sigma_0) \qquad R' = (P_1, A_1^+(\mathbf{x_1}, \mathbf{y}_{A_1}), [\mathbf{v}_1/\mathbf{x}_1])$$

où $\sigma_0 \mathbf{e}_1 \Downarrow \mathbf{v}_1$ et $A_1(\mathbf{x}_1) = P_1$. Soient st_0 (resp. st_1) le statut de A_0 (resp. A_1), n l'arité de A_0 et $\mathbf{v}_0$ la liste des n premières valeurs de $\sigma_0 \mathbf{p}_0$. Par définition de μ et par (6.1), il vient

$$\begin{array}{rcl} \mu(R_\varepsilon) & = & (K+1).((rank(A_0)-1).b^a + B_{\mathbf{v}_0}(b)(st_0) \\ \mu(R') & \leq & (K+1).((rank(A_1)-1).b^a + B_{\mathbf{v}_1}(b)(st_1) + K \end{array}$$

On distingue deux cas, selon que $A_0 >_F A_1$ ou $A_0 =_F A_1$

•• Supposons que $A_0 >_F A_1$. Par hypothèse sur la taille des valeurs calculées dans l'instant et par le lemme 2, on a :

$$\begin{array}{rcl} \mu(R') & \leq & (K+1).((rank(A_1)-1).b^a + b^a - 1) + K \\ & \leq & (K+1).rank(A_1).b^a - 1 \\ & \leq & (K+1)((rank(A_0)-1).b^a) - 1 \\ & < & \mu(R_\varepsilon) \end{array}$$

•• Supposons que $A_0 =_F A_1$. Par hypothèse, A_0 et A_1 ont le même statut $st = st_0 = st_1$ et la même arité n. Par le lemme 1, on sait que $q \models \mathbf{p}_0 >_{st} \mathbf{e}_1, \mathbf{y}_{A_1}$. Par définition du graphe d'appel et de $\leq_F$, il existe un cycle passant par A_0 et A_1 ; en particulier, on a $\mathbf{y}_{A_0} = \mathbf{y}_{A_1}$. De plus, par la condition read-once, $\mathbf{p}_0$ est de la forme $p_1, \ldots, p_n, \mathbf{y}_{A_0}$. Il vient $q_{\mathbf{v}_0} >_{st} q_{\sigma \mathbf{e}_1}$ et donc $q_{\mathbf{v}_0} >_{st} q_{\mathbf{v}_1}$. Par hypothèse sur la taille des valeurs calculées au cours de l'instant et par le lemme 2, on a :

$$\begin{array}{rcl} \mu(R') & \leq & (K+1).((rank(A_0)-1).b^a + B_{\mathbf{v}_0}(b)(st) - 1) + K \\ & \leq & (K+1).((rank(A_0)-1).b^a + B_{\mathbf{v}_0}(b)(st) - 1 \\ & < & \mu(R_\varepsilon) \end{array}$$

- Supposons que la réduction soit une opération de filtrage acceptée, i.e.

$$R_\varepsilon = ([x \trianglerighteq p]\ P_1, P_2, A_0^+(\mathbf{p}_0), \sigma_0) \qquad R' = ([p/x]P_1, A_0^+([p/x]\mathbf{p}_0), \sigma_1 \circ \sigma_0)$$

où $\sigma_1 = match(\sigma_0(x), p)$. Par définition de *match*, on a $A_0^+(\sigma_0 \mathbf{p}_0) = A_0^+((\sigma_1 \circ \sigma_0)[p/x]\mathbf{p}_0)$. On conclut en notant que $tc([x \trianglerighteq p]\ P_1, P_2) \geq tc([p/x]P_1)$ et $|[x \trianglerighteq p]\ P_1, P_2| > |[p/x]P_1|$. Les autres cas sont similaires.

Preuve de la proposition 2

Proposition. *Soit R un programme admettant une quasi-interprétation polynômiale. Il existe un polynôme Q_1 tel que si c borne la taille de R et si $A \in$ Reset alors, dans tout calcul de R, les tailles des paramètres dans tout appel de A sont bornés par $Q_1(c)$.*

Soit R un programme admettant une quasi-interprétation polynômiale q. Soient U un polynôme et u un entier naturels tels que U borne q et $|v| \leq q_v \leq u.|v|$. Soit c un entier naturel tel que c borne la taille de R. On note Q_1 le polynôme défini par $Q_1(x) = U(u.x)$ et on prouve que dans tout calcul de R la taille des paramètres des appels dont l'identicateur est dans *Reset* est bornée par $Q_1(c)$. Soient $R \equiv R_{i_0+1} \overset{*}{\to} R_{i_1} \overset{Env_1}{\mapsto} R_{i_1+1} \overset{*}{\to} R_{i_2} \overset{Env_2}{\mapsto} R_{i_2+1} \overset{*}{\to} \ldots$ un calcul de R et $A_1(\mathbf{v}_1)$, pour $A_1 \in Reset$, un appel de ce calcul, i.e. il existe $k \geq 0$ tel que

$$\begin{array}{lcl} R_k & \equiv & C_0[(A_1(\mathbf{e}_1), A_0^+(\mathbf{p}_0), \sigma_0)] \\ R_{k+1} & \equiv & C_0[(P_1, A_1^+(\mathbf{x}_1, \mathbf{y}_{A_1}), [\mathbf{v}_1/\mathbf{x}_1]) \quad \text{où } A_1(\mathbf{x}_1) = P_1,\ \sigma_0\mathbf{e}_1 \Downarrow \mathbf{v}_1 \end{array}$$

On prouve que les valeurs $\mathbf{v}_1$ sont bornées par $Q_1(c)$. Pour cela, on procède en deux étapes :

(1) on prouve que $q_{\sigma_0^+ A_0^+(\mathbf{p}_0)_{I_{A_0}}}$ borne la taille des valeurs $\mathbf{v}_1$,

(2) on prouve que l'inégalité $Q_1(c) \geq q_{\sigma_0^+ A_0^+(\mathbf{p}_0)_{I_{A_0}}}$ est vérifiée.

(1) Par le lemme 1, on sait que R_k est une instance des points de contrôle. En particulier, on a $q_{\sigma_0^+ A_0^+(\mathbf{p}_0)_{I_{A_0}}} \geq q_{\sigma_0^+ A_1^+(\mathbf{e}_1, \mathbf{y}_{A_1})_{I_{A_1}}}$. De plus, si n est l'arité de A_1, alors $I_{A_1} = \{1, \ldots, n\}$ (A_1 est dans *Reset*). En particulier, on a $\sigma_0^+ A_1^+(\mathbf{e}_1, \mathbf{y}_{A_1})_{I_{A_1}} = \sigma_0^+ A_1^+(\mathbf{e}_1, \mathbb{0}, \ldots, \mathbb{0})$ et il vient $q_{\sigma_0^+ A_1^+(\mathbf{e}_1, \mathbf{y}_{A_1})_{I_{A_1}}} = q_{\sigma_0^+ A_1^+(\mathbf{e}_1, \mathbb{0}, \ldots, \mathbb{0})} \geq q_{A_1^+(\mathbf{v}_1, \mathbb{0}, \ldots, \mathbb{0})}$ par hypothèse sur les quasi-interprétations des expressions et par monotonicité de $q_{A_1^+}$. On conclut en notant que, par définition d'une affectation, $q_{A_1^+(\mathbf{v}_1, \mathbb{0}, \ldots, \mathbb{0})}$ borne la quasi-interprétation, et donc la taille, de chacune des valeurs $\mathbf{v}_1$.

(2) Pour prouver ce résultat, on prouve que pour $k' = i_0+1, \ldots, k$, si $R_{k'} \equiv C[(P, A^+(\mathbf{p}), \sigma)]$ et $\neg(P \equiv 0)$ alors $Q_1(c) \geq q_{\sigma^+ A^+(\mathbf{p})_{I_A}}$. La preuve est par induction sur k'.

- Supposons que $k' = i_0 + 1$. Si $R_{k'} \equiv C[(P, A^+(\mathbf{p}), \sigma)]$ et $\neg(P \equiv 0)$ alors, par définition de la sémantique, $(P, A^+(\mathbf{p}), \sigma)$ est de la forme $(P_1, A_1^+(\mathbf{x}_1, \mathbf{y}_{A_1}), [\mathbf{v}_1/\mathbf{x}_1])$ où $A_1(\mathbf{x}_1) = P_1$. Par définition de la taille d'un programme, la taille des valeurs $\mathbf{v}_1$ est bornée par c. Par définition de Q_1, U et de u et par monotonicité de $q_{A_1^+}$, il vient

$$Q_1(c) \geq q_{[\mathbf{v}_1/\mathbf{x}_1]^+ A_1^+(\mathbf{x}_1, \mathbf{y}_{A_1})_{I_{A_1}}}$$

- Supposons que $k' > i_0 + 1$. On distingue deux cas selon que $R_{k'-1} \downarrow$ (a) ou non (b)

  (a) Si $R_{k'-1} \downarrow$, alors il existe j tel que $k' = i_j + 1$ et $R_{k'-1} \overset{Env_j}{\mapsto} R_{k'}$. En particulier, il existe R' tel que $R_{k'-1} \mapsto R'$, i.e., $R_{k'-1} \equiv \nu\mathbf{s}\ (S \mid In)$ et $R' \equiv \nu\mathbf{s}'\ Eval(In, V)$ pour $\mathbf{s}, \mathbf{s}', S, In, V$ donnés. Si $R_{k'} \equiv C[(P, A^+(\mathbf{p}), \sigma)]$, A est un identificateur de thread et $\neg(P \equiv 0)$ alors, par définition de la sémantique, on a

$$(P, A^+(\mathbf{p}), \sigma) = (P_1, A_1^+(\mathbf{x}_1, \mathbf{y}_{A_1}), [\mathbf{v}_1/\mathbf{x}_1]) \qquad A_1(\mathbf{x}_1) = P_1$$

$$R_{k'-1} \equiv C'[(s^y(x).P, A_1(\mathbf{r}_1), A_0^+(\mathbf{p}_0), \sigma_0)]$$

  On note σ_V la plus petite substitution telle que $\sigma_0 \subseteq \sigma_V$ et $\sigma_V\overline{\mathbf{r}_1} = V(\sigma_0 r_1)$. Par le lemme 1, on a :

$$q_{\sigma_V^+ A_0^+(\mathbf{p}_0)_{I_{A_0}}} \geq q_{\sigma_V^+ A_1^+(\overline{\mathbf{r}_1}, \mathbf{y}_{A_1}))_{I_{A_1}}} \geq q_{\sigma_V^+ A_1^+(\mathbf{v}_1, \mathbf{y}_{A_1}))_{I_{A_1}}} = q_{[\mathbf{v}_1/\mathbf{x}_1]^+ A_1^+(\mathbf{x}_1, \mathbf{y}_{A_1})_{I_{A_1}}}$$

  On conclut par hypothèse d'induction, en notant que $q_{\sigma_0^+ A_0^+(\mathbf{p}_0)_{I_{A_0}}} = q_{\sigma_V^+ A_0^+(\mathbf{p}_0)_{I_{A_0}}}$.

  (b) Si $\neg(R_{k-1} \downarrow)$ alors il existe C_0, R_ε et R' tels que $R_{k-1} \equiv C_0[R_\varepsilon]$, $R_k \equiv C_0[R']$ et $R_\varepsilon \rightarrowtriangle R'$. Supposons que $R_{k'} \equiv C[(P, A^+(\mathbf{p}), \sigma)]$ où $\neg(P \equiv 0)$ et A est un identificateur de thread. On suppose que $(P, A^+(\mathbf{p}), \sigma)$ n'apparaît pas dans R_{k-1} (sinon le résultat est immédiat par hypothèse d'induction) et on procède par cas sur la règle utilisée pour la réduction $R_\varepsilon \rightarrowtriangle R'$.

- Si $R_\varepsilon = R'_1 \mid (s_2^y(x).P_2, K, A_2^+(\mathbf{p_2}), \sigma_2)$ et $R' = R'_1 \mid ([y/x]P_2, A_2^+(\mathbf{p_2}), [v_1/y] \circ \sigma_2)$ où $R'_1 = (\overline{s_1}e_1, A_1^+(\mathbf{p_1}), \sigma_1), \sigma_1 s_1 = \sigma_2 s_2$ et $\sigma_1 e_1 \Downarrow v_1$. Alors $(P, A^+(\mathbf{p}), \sigma) = ([y/x]P_2, A_2^+(\mathbf{p_2}), [v_1/y] \circ \sigma_2)$ On conclut par hypothèse d'induction en notant que $\sigma_2^+ A_2^+(\mathbf{p_2})_{I_{A_2}} = ([v_1/y] \circ \sigma_2)^+ A_2^+(\mathbf{p_2})_{I_{A_2}}$.
- Si $R_\varepsilon = (A_1(\mathbf{e_1}), A_0^+(\mathbf{p_0}), \sigma_0)$ et $R' = (P_1, A_1^+(\mathbf{x_1}, \mathbf{y}_{A_1}), [\mathbf{v_1}/\mathbf{x_1}])$ où $A_1(\mathbf{x_1}) = P_1$ et $\sigma_0 \mathbf{e_1} \Downarrow \mathbf{v_1}$. Alors $(P, A^+(\mathbf{p}), \sigma) = (P_1, A_1^+(\mathbf{x_1}, \mathbf{y}_{A_1}), [\mathbf{v_1}/\mathbf{x_1}])$. Par le lemme 2, on a $q_{\sigma_0^+ A_0^+(\mathbf{p_0})_{I_{A_0}}} \geq q_{\sigma_0^+ A_1^+(\mathbf{e_1}, \mathbf{y}_{A_1})_{I_{A_1}}}$. Par hypothèse d'induction et par monotonicité de $q_{A_1^+}$ il vient :

$$Q_1(c) \geq q_{\sigma_0^+ A_1^+(\mathbf{e_1}, \mathbf{y}_{A_1})_{I_{A_1}}} \geq q_{\sigma_0^+ A_1^+(\mathbf{v_1}, \mathbf{y}_{A_1})_{I_{A_1}}} = q_{[\mathbf{v}_1/\mathbf{x}_1]^+ A_1^+(\mathbf{x_1}, \mathbf{y}_{A_1})_{I_{A_1}}}$$

- Si $R_\varepsilon = ([x \trianglerighteq p]\ P_1, P_2, A_0^+(\mathbf{p_0}), \sigma_0)$ et $R' = ([p/x]P_1, A_0^+([p/x]\mathbf{p_0}), \sigma_1 \circ \sigma_0)$ où $\sigma_1 = match(\sigma_0 x, p)$. Alors $(P, A^+(\mathbf{p}), \sigma) = ([p/x]P_1, A_0^+([p/x]\mathbf{p_0}), \sigma_1 \circ \sigma_0)$ On conclut par hypothèse d'induction, en notant que $\sigma_0^+ \mathbf{p}_0 = (\sigma_1 \circ \sigma_0)^+([p/x]\mathbf{p_0})$ (par définition de *match*). Les autres cas sont similaires.

Preuve de la proposition 3

Proposition. *Soit R un programme admettant une quasi-interprétation polynomiale. il existe un polynôme Q_2 tel que dans tout calcul*

$$R \equiv R_{i_0+1} \overset{*}{\to} R_{i_1} \overset{Env_1}{\mapsto} R_{i_1+1} \overset{*}{\to} R_{i_2} \overset{Env_2}{\mapsto} R_{i_2+1} \ldots$$

si c borne la taille de R et des entrées $Env_1, \ldots, Env_k$ pour $k \geq 1$ alors les tailles des valeurs calculées durant l'instant k sont bornées par $Q_2(c)$.

Soit R un programme admettant une quasi-interprétation q. Soit U un polynôme et soit u un entier naturel tels que U borne q et $|v| \leq q_v \leq u.|v|$ pour toute valeur v. On note $|.|$ la fonction sur les programmes non annotés définie par :

$$|0| = |A(\mathbf{e})| = 0 \qquad |\overline{s}e| = 1$$
$$|[s_1 = s_2] \cdot P_1,\ P_2| = |[x \trianglerighteq p]\ P_1, P_2| = max\{|P_1|, |P_2|\}$$
$$|s(x).P, K| = |P| \qquad |(P_1 \mid P_2)| = |P_1| + |P_2| \qquad |vs\ P| = |P|$$

On note K pour $max\{|P| \mid A(\mathbf{x}) = P\}$ et on note respectivement r et a le rang maximal et l'arité maximale des équations du programme. Etant donnée une région $\rho \in \mathcal{R}$, on définit $level(\rho)$ comme la longueur de la plus grande chaine telle que $\rho >_{\mathcal{R}} \rho' >_{\mathcal{R}} \ldots$.
On note P_n pour tout $n \geq 1$ et Q_2 les polynômes définis par :

$$\begin{aligned} P_1(x,y) &= u + y.x.(K.r.(y+1)^a + 1) \\ P_{n+1}(x,y) &= u + U(P_n(x,y)).x.(K.r.(U(P_n(x,y)) + 1)^a + 1),\ n \geq 1 \end{aligned}$$

$$Q_2(x) = U(P_\ell(x, u.Q_1(x)))$$

où Q_1 est le polynôme de la proposition 1 et ℓ est le plus haut niveau associé à la région d'un signal dans les équations du programme (sans perte de généralité, on peut supposer que $\ell \geq 1$.
Soit $R_{i_0+1} \overset{*}{\to} R_{i_1} \overset{Env_1}{\mapsto} R_{i_1+1} \overset{*}{\to} R_{i_2} \overset{Env_2}{\mapsto} R_{i_2+1} \ldots$ un calcul de R ; supposons que c borne la taille de R et les tailles des entrées $Env_1, \ldots, Env_k$ pour $k \geq 1$. On prouve que la taille de toute valeur calculée durant les k premiers instants est bornée par $Q_2(c)$. Pour cela, on procède en quatre étapes.

(1). On prouve que pour $k' = i_0 + 1, \ldots, i_{k+1}$, si $R_{k'}$ est structurellement équivalent à $C[(P, A^+(\mathbf{p}), \sigma)]$, où A est un identificateur de thread :
- Si $P = A'(\mathbf{e})$, où $A' \notin \mathit{Reset}$ et $\sigma\mathbf{e} \Downarrow \ldots, v, \ldots$, alors $q_{\sigma^+ A^+(\mathbf{p})_{M_{Inf}}} \geq q_v$ pour tout $M \subseteq \mathcal{W}(A')$ et $\mathcal{W}(A') \subseteq \mathcal{W}(A)$
- Si $P = \overline{s}e$, où s est un nom de signal de région ρ et $\sigma e \Downarrow v$, alors $q_{\sigma^+ A^+(\mathbf{p})_{\lfloor\rho\rfloor_{Inf}}} \geq q_v$ et $\{\rho\} \subseteq \mathcal{W}(A)$

(2). Pour tout ensemble M de régions, on note d_M le max des quasi-interprétations des valeurs lues sur des signaux dont la région est dans M au cours des k premiers instants ; on pose $d_M = 0$ si aucune valeur n'est lue. On prouve que pour $k' = i_0 + 1, \ldots, i_{k+1}$, si $R_{k'} \equiv C[(P, A^+(\mathbf{p}), \sigma)]$ où $\neg(P \equiv 0)$ et A est un identificateur de thread alors :

$$U(max\{u.Q_1(c), d_{M_{Inf}}\}) \quad \geq \quad q_{(\delta\circ\sigma)A^+(\mathbf{p})_{M_{Inf}}}, \text{ pour tout } M \subseteq \mathcal{W}(A)$$

où δ est la subsitution associant les variables $fv(\sigma\mathbf{p})$ aux valeurs à lire dans $(P, A^+(\mathbf{p}), \sigma)$ avant la fin du cycle et avant la fin de l'instant k s'il elles existent et à $\mathbb{0}$ sinon.

(3). On prouve que pour toute région ρ de niveau n les quasi-interprétations des valeurs lues sur les signaux associés à cette région durant les k premiers instants sont bornées par $P_n(c, u.Q_1(c))$.

(4). On conclut par (1), (2) and (3).

[(1)] - Soit $k' \in i_{i_0}+1, i_{i_0}+2, \ldots, i_{k+1}$; supposons que $R_{k'}$ est structurellement équivalent à $C[(P, A_0^+(\mathbf{p}_0), \sigma_0)]$ où A_0 est un identificateur de thread. Supposons que $P = A_1(\mathbf{e}_1)$, $A_1 \notin \mathit{Reset}$ et $\sigma_0\mathbf{e}_1 \Downarrow \ldots, v, \ldots$. Par le lemme 1, $R_{k'}$ est une instance des points de contrôle et, en particulier, on a $q_{\sigma_0^+ A_0^+(\mathbf{p}_0)_{M_{Inf}}} \geq q_{\sigma_0^+ A_1^+(\mathbf{e}_1, \mathbf{y}_{A_1})_{M_{Inf}}}$ pour tout $M \subseteq \mathcal{W}(A_1)$. Par définition d'une affectation, il vient $q_{\sigma_0^+ A_0^+(\mathbf{p}_0)_{M_{Inf}}} \geq q_v$ pour tout $M \subseteq \mathcal{W}(A_1)$. Puisque $A_1 \notin \mathit{Reset}$, il est immédiat par définition de $\mathcal{W}$ que $\mathcal{W}(A_1) \subseteq \mathcal{W}(A_0)$. Le raisonnement est similaire si P est de la forme $\overline{s}e$ où s est un signal de région ρ en notant que, par définition, $\{\rho\} \subseteq \mathcal{W}(A_0)$.

[(2)] La preuve de (2) est par induction sur la longueur de la dérivation et suit le même schéma que la preuve de la proposition 2.

[(3)] - Pour tout $n \geq 0$ on note $\mathcal{R}_n = \{s \in \mathcal{R} \mid s : Sig_\rho(t),\ level(\rho) = n\}$. On prouve que, pour tout $n \geq 0$, $s \in \mathcal{R}_n$ implique $q_v \leq P_n(c, u.Q_1(c))$ pour toute valeur v lue sur le signal s durant les k premiers instants. La preuve est par induction sur n. Si $n = 0$ alors le résultat est immédiat puisque $\mathcal{R}_0 = \emptyset$ par définition de *level*. Si $n > 0$ alors supposons que $\mathcal{R}_n \neq \emptyset$ (sinon le résultat est immédiat) et prenons s un nom de signal dans $\mathcal{R}_n$. Soit $0 \leq k' < k$ et soit v une valeur lue sur s dans l'instant k'. Soit v est émise sur s dans l'instant k' soit v est lue par déréférenciation de s à la fin de l'instant k'. Par définition de la sémantique, une valeur émise sur s persiste dans l'instant. Il existe donc un programme $v\mathbf{s}(S \mid In)$ structurellement équivalent à $R_{i_{k'+1}}$ tel que $\{v_1, \ldots, v_m\} = \{v \mid \overline{s}v \in \hat{S}\}$ et soit $v \in \{v_1, \ldots, v_m\}$ soit $v = [v_{\pi(1)}; v_{\pi(2)}; \ldots; v_{\pi(m)}]$ pour une certaine permutation π. On prouve que $P_n(c, u.Q_1(c)) \geq q_{[v_{\pi(1)}; v_{\pi(2)}; \ldots; v_{\pi(m)}]}$. Pour cela, on note b l'entier naturel $U(P^{n-1}(c, u.Q_1(c)) + 1$ et on procède en quatre étapes.
(a) on définit une fonction μ des programmes (annotés) vers les entiers naturels.
(b) on prouve que pour tout k' dans $0, 1, \ldots, k$ on a $\mu(R_{i_{k'+1}}) \leq c.(K.r.b^a + 1)$
(c) on prouve que pour tout k' dans $0, 1, \ldots, k$ et pour tout $i_{k'} + 1 \leq k'' < i_{k'+1}$ on a $\mu(R_{k''}) \geq \mu(R_{k''+1})$
(d) on conclut par (b) et (c).

[(3.a)] - On note $(\!|A(\mathbf{v})|\!) = K.((rank(A)-1).b^a + B_{\sigma p_1,\ldots,\sigma p_n}(b)(st))$ où st est le statut de A. On définit la fonction μ des programmes annotés vers les entiers naturels, en posant $\mu(P, Env_k, \sigma_\emptyset) = |P|$ pour tout P, de la manière suivante :

$$\mu(0) = 0 \qquad \mu(R' \mid R'') = \mu(R') + \mu(R'') \qquad \mu(\nu s\ R) = \mu(R)$$

$$\mu(P, A^+(p_1, \ldots, \ldots, p_{m+m'}), \sigma) = \begin{cases} tc(P).(\!|A(\sigma p_1, \ldots, \sigma p_m)|\!) + |P| & \text{si } \rho \in \mathcal{W}(A), level(\rho) = n \\ 0 & \text{sinon} \end{cases}$$

où m est l'arité de A

Il est facile de vérifier que, pour tout $i_{k'} + 1 \leq k'' < i_{k'+1}$ si $R' \equiv R_{k''}$ alors $\mu(R')$ est défini dans $\mathbb{N}$ et que μ identifie les programmes structurellement équivalents. Si A est défini par une équation $A(\mathbf{x}) = P$ et $\rho \in \mathcal{W}(A)$, $level(\rho) = n$ alors (par hypothèse sur la forme des équations et par définition de K) il est immédiat que :

$$\forall \mathbf{v}.(\mu(P, A^+(\mathbf{p}), [\mathbf{v}/\mathbf{x}]) \leq (\!|A(\mathbf{v})|\!) + K) \tag{6.2}$$

[(3.b, 3.c)] - Les preuves suivent le schéma de celle de la proposition 1.

[(3.d)] - Premièrement, on prouve que $U(P_{n-1}(c, u.Q_1(c)))$ borne les quasi-interprétations des valeurs $v_1, \ldots, v_m$. Si $v \in \{v_1, \ldots, v_m\}$ alors, par un simple examen des règles de la sémantique, on sait que $R_{i_{k'+1}}$ (le programme à la fin de l'instant k') est structurellement équivalent à un programme $C[(\overline{s'}e, A^+(\mathbf{p}), \sigma)]$ où $\sigma s' = s$ et $\sigma e \Downarrow v$. Par (1), on a $\{\rho\} \subseteq \mathcal{W}(A)$ et, par définition, $\{\rho\}_{Inf} \subseteq \bigcup_{i=1}^{n-1} \mathcal{R}_{n-1}$. Par hypothèse d'induction, les q.i. des valeurs lues sur les signaux, dont la région est dans $\{\rho\}_{Inf}$, sont bornées par $P_{n-1}(c, u.Q_1(c))$. Par (2), on a donc

$$U(P_{n-1}(c, u.Q_1(c))) \geq q_{(\delta\circ\sigma)A^+(\mathbf{p})_{\{\rho\}_{Inf}}}$$

où δ est la substitution associant les variables de $fv(\sigma\mathbf{p})$ aux valeurs à lire dans $(\overline{s'}e, A^+(\mathbf{p}), \sigma)$ avant la fin du cycle et durant les k premiers instants si elles existent et $\mathbb{0}$ sinon. Par (1) et par monotonicitéde q_{A^+}, il vient

$$U(P_{n-1}(c, u.Q_1(c))) \geq q_{(\delta\circ\sigma)A^+(\mathbf{p})_{\{\rho\}_{Inf}}} \geq q_{\sigma^+A^+(\mathbf{p})_{\{\rho\}_{Inf}}} \geq q_v$$

Finalement, en notant que (par (3.b) et (3.c)) on a $m \leq c.(K_e.r.b^a + 1)$ (la mesure μ d'un programme borne le nombre de constructions *emit* de ce programme) et que $q_{\mathsf{cons}} \leq u$ il vient :

$$q_{[v_{\pi(1)};v_{\pi(2)};\ldots;v_{\pi(m)}]} \leq u + U(P_{n-1}(c, u.Q_1(c))).c.(K.r.b^a + 1) = P_n(c, u.Q_1(c))$$

[(4)] - Finalement, si v est une valeur calculée par le programme durant l'instant k alors il existe $k' \in \{i_k + 1, i_k + 2, \ldots, i_{k+1}\}$ tel que $R_{k'}$ est structurellement équivalent à un programme $C[(P, A^+(\mathbf{p}), \sigma)]$ et soit $P = A'(\mathbf{e})$ et $\sigma\mathbf{e} \Downarrow \ldots, v, \ldots$ soit $P = \overline{s}e$ et $\sigma e \Downarrow v$. Par (1), on sait que $q_{\sigma^+A^+(\mathbf{p})_{M_{Inf}}} \geq q_v$ où M est un ensemble de régions tel que $M \subseteq \mathcal{W}(A)$. En effet, par définition, soit $P = A'(\mathbf{e})$, $A' \notin Reset$ et $\mathcal{W}(A') \subseteq \mathcal{W}(A)$ soit $P = \overline{s}e$ et la région de s et dans $\mathcal{W}(A)$). Par (2) et (3), en particulier, on a $U(P_\ell(u.c, u.c, Q_1(u.c))) \geq q_{\sigma' A^+(\mathbf{p})_{M_{Inf}}} \geq q_v \geq |v|$.

Preuve du théorème 1

Théorème. *Un programme admettant une quasi-interprétation polynomiale est réactif efficace.*

La preuve découle immédiatement des propositions 1 et 3. D'une part, la taille d'un programme au début de chaque instant est polynômiale en la taille du programme initial et des entrées puisque le nombre de thread est borné par hypothèse et la taille des paramètres des threads est bornée par la proposition 3. D'autre part, le temps de calcul de chaque réaction est également polynômial par les propositions 1 et 3.

Deuxième partie

Programmation Multi-Cores Sûre

Chapitre 7

Programmation des architectures Multi-Core

La difficulté fondamentale de la programmation concurrente, en comparaison avec la programmation séquentielle, découle du partage de l'information par plusieurs flots d'exécution. Parmi les modèles de programmation les plus courants on peut distinguer les processus (mémoire non partagée avec rendez-vous, passage de messages,...) des threads (mémoire partagée avec verrous, moniteurs,...). De manière générale, la synchronisation de ces flots d'exécution représente un problème complexe tant pour la conception que pour la maintenance des programmes. Aucun de ces deux modèles de programmation concurrente ne s'impose réellement ; alors que la conception de langages associant efficacité, simplicité de développement et de maintenance et sûreté reste un sujet qui est, plus que jamais, d'actualité. D'une part, la conception de tels langages est nécessaire à la fois pour exprimer la nature intrinsèquement concurrente des systèmes réactifs et pour séparer au niveau logique des flots d'exécution indépendants (modularité du code). D'autre part, les besoins de l'industrie, liée aux changements récents dans l'évolution de l'architecture des machines grand public, se font de plus en plus pressants.

Depuis l'apparition des machines grand public, les développeurs et les utilisateurs se sont habitués à une évolution rapide et régulière de la puissance des configurations suivant de près la *loi de Moore*. Une caractéristique essentielle de cette évolution (augmentation du nombre de transistors, de la fréquence,...) est la stabilité de l'architecture (modulo l'introduction de nouveaux jeux d'instructions) par rapport au logiciel. En effet, lors de l'augmentation de la puissance des machines, les logiciels ont bénéficié jusqu'ici, à moindre frais, des gains successifs de puissance. Cependant, une évolution récente de l'architecture des machines compromet cette progression. Depuis peu de temps, les constructeurs se sont tournés vers de nouvelles techniques pour accroître la puissance des configurations. Dans un premier temps, l'apparition de l'*Hyper-Threading* [1] a permis de légers gains en termes d'efficacité. Dans un second temps, les architectures à base de processeurs *Dual Core* [2] ont placé pour de bon la *vraie* concurrence au sein des machines grand public. Actuellement, certaines configurations combinent ces deux techniques et la multiplication des *cores* au sein des processeurs est annoncée comme une solution incontournable, au moins à court et moyen termes. Tout cela préfigure du potentiel des machines à venir en terme de parallélisme et, en particulier, des besoins au niveau logiciel pour leur exploitation.

[1] un processeur supportant l'Hyper-Threading se présente au système d'exploitation comme deux processeurs virtuels
[2] deux unités physiques de calculs effectives au sein de la même puce

On s'intéresse ici au cas de la programmation par threads. En toute généralité, la programmation concurrente à base de threads repose :

1. sur la possibilité de composer plusieurs flots d'exécution séquentiels (les threads),
2. sur des primitives de synchronisation permettant de contrôler l'accès des threads à un espace mémoire partagé.

Dans les langages à base de threads, on peut principalement distinguer deux types de partage du contrôle entre les threads : coopératif et préemptif. Dans un système basé sur un mode d'exécution coopératif, un flot d'exécution reçoit le contrôle, réalise de manière atomique une série d'opérations et retourne explicitement le contrôle ; le choix du prochain thread peut être explicite (*co-routines*) ou résulter d'un algorithme d'ordonnancement. L'ordonnancement coopératif présente l'avantage d'être simple (pas de mécanisme de protection des données) et intuitif (chacun son tour). Malheureusement, celui-ci souffre d'une mauvaise réputation dont l'origine se trouve dans son utilisation par d'anciens systèmes d'exploitation *multi-tâches* tels que Windows (≤ 3.11) ou Mac OS (≤ 9). Ces systèmes souffraient d'une instabilité chronique souvent imputée, en partie à raison [3], à un manque de robustesse inhérent à tout système d'exploitation multi-tâches basé sur ce principe (possibilité de blocage de l'ensemble du système dès qu'un thread ne retourne pas le contrôle). Les versions plus récentes de ces systèmes d'exploitations utilisent des algorithmes d'ordonnancement préemptifs. Afin d'éviter un blocage du système, ce type d'algorithme repose sur la possibilité de retirer le contrôle à un thread en un point quelconque de son exécution. En contrepartie l'utilisateur dispose de mécanismes de synchronisation lui permettant de maintenir des invariants sur les données partagées entre deux exécutions successives du même thread. L'exemple le plus courant d'un tel mécanisme, appelé *verrous*, trouve son origine dans les notions de sémaphore[42] et de moniteurs[59]. Si les systèmes d'exploitation bénéficient effectivement, en terme de robustesse, de l'introduction de ce type de programmation, l'expérience montre que la complexité des mécanismes de synchronisation mis en jeu rend celle-ci extrêmement difficile à maîtriser. A ce sujet, le tutoriel[17] permet de se faire une idée assez précise du niveau de complexité atteint. Bien que ce tutoriel ne soit pas récent, il est intéressant de noter que les méthodes de programmation présentées ne sont pas différente de la pratique actuelle.

Dans cette partie, on présente un noyau de langage, appelé PACT, pour PArtially Cooperative Threads, dédié à un style de programmation concurrente à base de threads. La conception du langage PACT repose principalement sur le choix de primitives de synchronisation basées sur un ordonnancement coopératif et, de manière plus générale, sur l'approche réactive. Le nom choisi suggère un mécanisme de coopération régissant l'accès aux données partagées par un ensemble de threads. La terminologie "PArtially Cooperative Threads" dénote un compromis entre le préemptif et le coopératif. Le point de vue défendu ici est dual :

1. l'abandon d'un ordonnancement coopératif de l'ensemble des composantes d'un système d'exploitation était justifié,
2. la notion d'ordonnancement coopératif comme moyen simple et intuitif de synchronisation, au niveau utilisateur, mérite d'être considérée.

En PACT, un programme décrit des threads se synchronisant par le biais de *zones synchrones* :

– des threads soumis à l'*influence* d'une même zone synchrone adoptent collectivement un mode d'ordonnancement coopératif.

[3] L'absence de mécanismes de protection mémoire telle que ceux des processus UNIX était également largement en cause

– un ordonnancement préemptif classique est en charge de répartir le contrôle entre les différentes zones synchrones et les threads en dehors de l'influence de toute zone synchrone (on parle alors de *threads libres*).

On peut également noter, qu'au sein de chaque zone synchrone, les threads bénéficient d'une notion commune de temps logique (les instants) basée sur la présence et l'absence d'événements valués. Afin de respecter l'esprit du modèle coopératif lors de l'accès à la mémoire partagée (atomicité de l'exécution entre deux points de coopération) une analyse statique des programmes est introduite. Plus précisément, on définit une politique de droit d'accès à la mémoire basée sur l'attribution à chaque adresse mémoire d'une notion de propriété relative soit à une zone synchrone soit à un thread. Intuitivement, cette politique stipule que :

- le droit d'accès à une adresse associée à un thread est réservé à ce seul thread,
- le droit d'accès à une référence associée à une zone synchrone est partagé par l'ensemble des threads soumis à son influence.

Ici, le terme accès désigne aussi bien la lecture que l'écriture ; on ne fait pas de distinction entre ces deux opérations. Ensuite, on introduit un système de types permettant de s'assurer qu'un programme vérifie cette politique.

Le langage PACT se place directement dans la continuité de la librairie des FAIR THREADS en C qui mélange déjà au sein d'un même langage les approches coopérative et préemptive afin de contourner les limitations d'une approche purement coopérative (gestion des entrées sorties bloquantes, prise en charge des architectures multiprocesseur,...). L'utilisateur est alors libre de choisir (à tout moment de l'exécution d'un programme et indépendamment pour chacun des threads) de bénéficier de l'une ou l'autre. Malheureusement, en introduisant une part de concurrence préemptive dans le modèle, les FAIR THREADS héritent des problèmes classiques liés à ce type de programmation. En particulier, la protection des données manipulées par un thread entre deux points de coopération n'est plus garantie structurellement (la présence de la notion de verrous dans ce langage en est symptomatique). Le travail présenté ici résulte avant tout d'une volonté de parvenir à une meilleure compréhension de ce mélange des approches coopérative et préemptive. Directement dérivé des FAIR THREADS, le langage PACT en diffère néanmoins sur plusieurs points ; ces différences sont motivées pour la plupart par la volonté d'assurer statiquement un certain nombre de propriétés, principalement :

– il nous semble qu'un langage de haut niveau pour la programmation concurrente doit bénéficier de propriétés de sûreté de l'évaluation telles que celles assurées par un typage fort comme celui, par exemple, d'OBJECTIVE CAML.
– le contrôle statique de l'accès à la mémoire mentionné précédemment à également conduit à introduire certaines modifications.
– Une propriété plus ambitieuse, non développé ici mais souhaitée à terme, concerne le contrôle statique des ressources nécessaires à l'exécution d'un programme et la garantie de la coopération effective des threads soumis à l'influence d'une zone synchrone. Les recherches en cours sur ce sujet dans le cadre du $S\pi$-calcul (voir Partie I) justifient également certains aspects du langages.

D'autres changements sont motivés par des choix plus personnels. Dans tous les cas, ces points seront, pour les plus significatifs d'entre eux, discutés lors de la définition du langage.

Chapitre 8

PACT
(PArtially Cooperative Threads)

Dans ce chapitre, nous présentons la syntaxe et la sémantique du langage PACT qui propose un style de programmation concurrente à base de threads. De manière standard, ces threads sont des flots d'exécution séquentiels partageant la mémoire. En revanche, le langage PACT se distingue par le choix :

(1) d'un *compromis* entre les styles coopératif et préemptif

(2) des mécanismes de *protection des données* et de *synchronisation* proposés.

Le modèle de programmation introduit par le langage raffine celui proposé par la librairie des FAIR THREADS. En particulier, nous présenterons dans les chapitres suivants une discipline de programmation et une analyse statique dont l'objectif est de combiner *de manière consistante* les avantages des styles coopératif et préemptif.

En PACT, un programme définit le comportement de threads pouvant se synchroniser au moyen de (se placer sous l'influence de) zones synchrones. Les threads soumis à l'influence d'une même zone synchrone partagent la même notion d'instant et adoptent collectivement un ordonnancement coopératif. En particulier, la visibilité des signaux (présence et absence) est associée à la notion locale (zone synchrone) d'instant. Plusieurs zones synchrones composent le système et les threads peuvent se lier dynamiquement à chacune d'entre elles (une à la fois) ou opter pour une exécution autonome (on parle alors de thread *libres*). L'ordonnancement des différentes zones synchrones et des threads libres est réalisé selon un mode préemptif.

Une représentation graphique du modèle d'exécution de PACT est donnée dans la figure (8.1) Les composantes $T(1)$ à $T(n)$ représentent des threads libres. Durant l'exécution, certains de ces threads (ici, $T(3)$,$T(4)$,...) se synchronisent au moyen d'une zone synchrone (ici, $S(1)$, choisie parmi les zones $S(1)$ à $S(m)$). Ces threads partagent alors la même notion d'instant ; un instant est représenté par deux bandes noires reliées par une flèche dénotant le flot d'exécution au cours de l'instant. L'exécution des threads est réalisée selon un mode coopératif et ceux-ci peuvent communiquer par signaux valués. A noter que, dans la figure (8.1), la mémoire est partagée ; à tout moment, les threads peuvent accéder à la mémoire sans restriction. Nous reportons au chapitres suivant la description des restrictions apportées par PACT afin de garantir l'atomicité de l'exécution des threads entre deux points de coopération.

8.1 Définition du langage

Dans un premier temps, on présente la syntaxe du langage, ainsi qu'une description informelle de sa sémantique. Ce langage repose, d'une part, sur un ensemble de primitives de communication

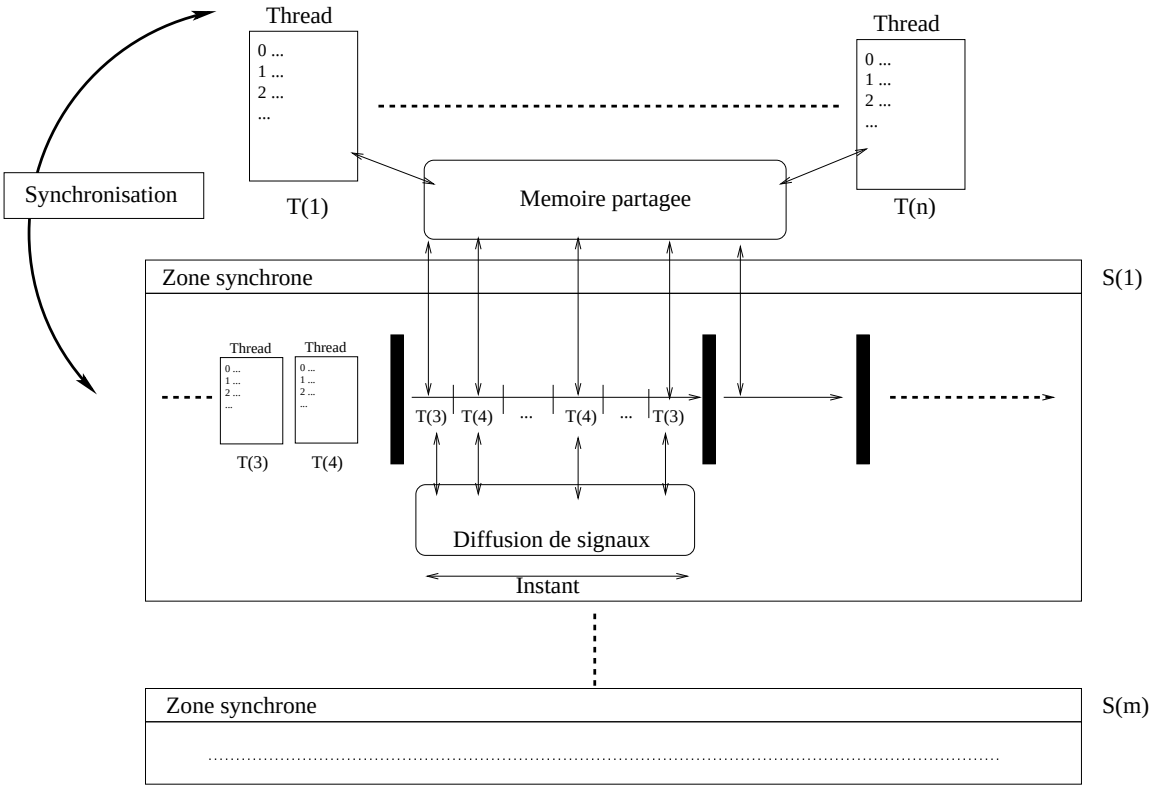


FIG. 8.1 – Modèle d'exécution

et de synchronisation et, d'autre part, sur l'existence d'un sous langage fonctionnel du premier ordre (on retrouve ici une restriction liée aux travaux de la partie I). Le comportement des threads est décrit au moyen d'équations récursives basées sur une opération de filtrage des valeurs à la ML. On définit successivement les catégories syntaxiques suivantes : les valeurs, les expressions, les filtres et les threads. Par la suite, les éléments d'une catégorie syntaxique donnée pourront être indicés par des entiers pour plus de clarté. De manière générale, on note **a** une liste d'éléments $a_1, \ldots, a_n$ et $\overline{\alpha}$ un produit cartésien $\alpha_1 \times \ldots \times \alpha_n$.

8.1.1 Valeurs

On suppose donnés deux ensembles dénombrables, les *noms de références* $Ref = \{r, r', \ldots\}$ et les *noms de signaux* $Sig = \{s, s', \ldots\}$ [1]. Les valeurs du langages et les types de données associés sont définis de la manière suivante :

- la constante $*$ est l'unique valeur du type 1,
- Les références (resp. les signaux) portant des valeurs de type τ sont les valeurs du type τ *ref* (resp. τ *sig*) (ces valeurs sont dites *dynamiques*),
- Les listes de valeurs de type τ forment le type τ *list*.

$\mathcal{V}$ (resp. $\mathcal{T}$) dénote l'ensemble des valeurs (resp. des types) et $\mathcal{V}_\tau$ dénote l'ensemble des valeurs de type τ. D'autres types de données de base ou structures de données (entiers, booléens, arbres...) peuvent être ajoutés.

[1] ces éléments seront appelés références et signaux par la suite.

8.1.2 Expressions et filtres

Le langage PACT est un langage de coordination d'atomes que sont les expressions d'un langage fonctionnel du premier ordre. Ces expressions sont formées de variables ($Var = \{x, y, \ldots\}$), de valeurs et de symboles de fonction $F = \{f, g, \ldots\}$ définis axiomatiquement. Elles sont définies par la grammaire suivante :

$$Exprs \quad e \quad ::= \quad x \mid r \mid s \mid \mathsf{nil} \mid \mathsf{cons}(e, e) \mid f(\mathbf{e})$$

Les symboles nil et cons sont les constructeurs de liste ; par la suite, on notera plus simplement $[v_1; \ldots; v_n]$ la liste $\mathsf{cons}(v_1, \mathsf{cons}(v_2, \ldots \mathsf{cons}(v_n, \mathsf{nil})))$. On note $var(e)$ (resp. $ref(e)$ et $sig(e)$) l'ensemble des variables (resp. références et signaux) apparaissant dans e.

Pour chaque symbole de fonction f, on suppose donnés : un domaine $\mathcal{V}_{\overline{\tau}}$, un co-domaine $\mathcal{V}_\tau$ et une fonction $[\![f]\!]$ de $\mathcal{V}_{\overline{\tau}}$ vers $\mathcal{V}_\tau$; on associe alors la fonction f à son type par la notation $f : \overline{\tau} \to \tau$. On suppose que, pour tout symbole de fonction f et pour toutes valeurs $\mathbf{v}$, si $[\![f]\!](\mathbf{v}) = v$ alors $ref(v) \subseteq ref(\mathbf{v})$ et $sig(v) \subseteq sig(\mathbf{v})$.
Les *filtres* sont des termes définis à partir des variables et des constructeurs. Les filtres *Pat* des filtres sont définis par la grammaire suivante :

$$Pat \quad p, p', \ldots \quad ::= \quad x \mid \mathsf{nil} \mid \mathsf{cons}(p, p')$$

Une substitution close, notée $\sigma, \sigma', \ldots$, est une fonction partielle des variables vers les valeurs. On note $[v/x]$ la substitution qui à x associe v et qui est non définie pour $y \neq x$. Si σ et σ' sont deux substitutions telles que $Dom(\sigma) \cap Dom(\sigma') = \emptyset$, on note $(\sigma \cdot \sigma')$ la substitution définie par l'union point à point de σ et σ'. Intuitivement, le filtrage consiste à déterminer si une valeur v correspond à un filtre p. Si cela est le cas, on note $match(v, p)$ la plus petite substitution telle que $\sigma p = v$. Dans le cas contraire, on dit que le filtrage échoue, ce que l'on note $match(v, p) = \uparrow$.

8.1.3 Threads, programmes et sémantique informelle

Les threads représentent la catégorie syntaxique principale du langage. Etant donné un ensemble dénombrable $\mathcal{L} = \{\ell, \ell', \ldots\}$ de noms de *zones synchrones*, les threads sont définis par la grammaire suivante :

$$\begin{aligned} P, P', \ldots \quad ::= \quad & 0 \mid A(\mathbf{e}) \mid [e \trianglerighteq p]\, P, P' \\ & \mid (x = \mathtt{ref}\ e) \cdot P \mid \mathtt{set}\ e\ e \mid (x = \mathtt{get}\ e) \cdot P \\ & \mid (x = \mathtt{sig}) \cdot P \mid \mathtt{emit}\ e\ e \mid (x = \mathtt{await}\ e) \cdot P \mid (x = \mathtt{pre}\ e) \cdot P \\ & \mid P; P' \mid \mathtt{watch}\ e\ \cdot P, P' \mid \mathtt{link}\ \ell\ P \mid \mathtt{unlink}\ P \\ & \mid \mathtt{run}\ A(\mathbf{e}) \end{aligned}$$

A noter ici que l'on introduit directement des primitives pour la gestion des références. En effet, la définition des signaux dans le paradigme réactif, par exemple dans le π-calcul synchrone, ne permet pas de coder ces objets (contrairement à ce que l'on pourrait faire en π-calcul). Simuler des signaux valués à partir de signaux purs et de références ne serait pas non plus intéressant dans notre cas car nous introduirons une distinction assez forte entre signaux et références. On étend la notion de substitution aux threads en notant σP le thread obtenu par application de σ aux expressions de P après renommage des variables liées (voir la description informelle ci-dessous) dans P. On note $fv(P)$ l'ensemble des variables libres de P.

Un programme est la donnée d'un ensemble d'équations de la forme $A(\mathbf{x}) = P$ et d'un thread initial P clos (sans variables libres). Durant son exécution un thread est soit *libre* soit soumis à l'*influence*

d'une zone synchrone. L'exécution de l'ensemble des threads soumis à une zone synchrone se déroule sous deux contraintes :

1. mode coopératif : seul le thread ayant le contrôle peut s'exécuter, il conserve le contrôle jusqu'à ce qu'il soit suspendu.
2. mode synchrone : l'exécution au sein d'une zone synchrone se déroule par instants successifs. A chaque instant un signal est soit absent soit présent pour cette zone (un signal est présent si au moins une valeur est émise sur ce signal). Lorsqu'aucun thread ne peut plus progresser, l'instant suivant commence. La suspension d'un thread a lieu lorsque celui-ci attend un signal absent ou souhaite quitter l'influence de la zone synchrone.

Ci-dessous, on donne une définition informelle de la sémantique du langage. Dans cette description, lorsque l'on fait référence à la présence ou à l'absence d'un signal il est entendu que cela est relatif à la zone synchrone à laquelle le thread est soumis. En particulier, les opérations associées à la notion de signal ne sont définies que lorsque un thread est soumis à l'influence d'une zone synchrone. Intuitivement, le comportement des threads peut être décrit en trois parties : (1) contrôle, (2) mémoire partagée et (3) signaux et synchronisation.

1. **Contrôle** La constante 0 dénote un thread dont l'exécution est terminée. L'application $A(\mathbf{e})$ évalue $[\mathbf{v}/\mathbf{x}]P$ si $\mathbf{e}$ dénote $\mathbf{v}$ et $A(\mathbf{x}) = P$. On suppose que les symboles apparaissant dans un programme sont bien définis par une équation. La construction $[e \trianglerighteq p]\ P_1, P_2$ évalue P_1 si p filtre la valeur dénotée par e. Dans ce cas, les variables de p sont associées dans P_1 au résultat du filtrage. Dans le cas contraire, P_2 est évalué. On suit ici la convention utilisée en Objective Caml selon laquelle les filtres sont linéaires, i.e chaque variable apparaît au plus une fois dans un filtre [34]. De manière standard, la composition séquentielle de deux threads P_1 et P_2 est notée $P_1; P_2$. La construction `run` $A(\mathbf{e})$ démarre l'exécution du nouveau thread $A(\mathbf{v})$, où $\mathbf{v}$ sont les valeurs dénotées par $\mathbf{e}$, et termine immédiatement. Un nouveau thread hérite de la zone synchrone du thread qui le lance ; en particulier, il est libre si ce dernier est libre.
2. **Mémoire partagée.** La construction $(x = \texttt{ref}\ e) \cdot P$ dénote le thread P où x est liée à une nouvelle référence dont la valeur initiale est la valeur dénotée par e. La construction `set` $e_1\ e_2$ affecte à la référence dénotée par e_1 la valeur dénotée par e_2. La construction $(x = \texttt{get}\ e) \cdot P$ dénote le thread P où x est liée à la valeur associée à la référence dénotée par e.
3. **Signaux et synchronisation** La construction $(x = \texttt{sig}) \cdot P$ dénote le thread P où x est liée à un nouveau signal. La construction `emit` $e_1\ e_2$ génère l'émission de la valeur dénotée par e_2 sur le signal dénoté par e_1. Le langage propose deux constructions pour la lecture de la valeur associée à un signal. Intuitivement, la valeur d'un signal pour un instant donné est le multi-ensemble des valeurs émises sur ce signal pendant cet instant. Au cours de l'instant, la construction $(x = \texttt{await}\ e) \cdot P$ permet de lire une information partielle (une des valeurs du multi-ensemble) ; dans le thread P, x est liée à une des valeurs émises sur le signal si elle existe, dans le cas contraire le thread est suspendu. L'attente d'un signal absent par cette construction peut prendre un nombre arbitraire d'instants. La construction $(x = \texttt{pre}\ e) \cdot P$ permet de lire une liste représentant le multi-ensemble des valeurs émises sur un signal à l'instant précédent. La construction `watch` $e \cdot P_1, P_2$ évalue P pour l'instant courant de la zone synchrone courante. Si P est suspendu à la fin de l'instant et si le signal dénoté par e est présent alors l'exécution de P_1 est abandonnée au profit de celle de la continuation P_2 à l'instant suivant. La construction `link` $\ell\ P$ attache le thread à la zone synchrone ℓ pour l'exécution de P. La construction `unlink` P libère le thread pour l'exécution de P.

8.1.4 Déterminisme, ordonnancement et fairness

Pour conclure la description informelle de la sémantique que nous venons de donner, on apporte quelques précisions sur l'ordonnancement coopératif des threads au sein d'une même zone synchrone. Dans [25], une sémantique de la partie purement coopérative de la librairie des FAIR THREADS (une seule zone synchrone et pas de threads libres) est donnée. Dans cette sémantique, l'ordonnancement des threads est décrit de manière déterministe. Plus précisément, l'algorithme d'ordonnancement est un *round robin*, i.e. les threads sont placés dans une file d'attente de type *Firt-In-First-Out*. L'ordre d'exécution des threads coopératifs attachés à un scheduler est conservé au cours de l'exécution. En pratique, ce type d'ordonnancement devient inefficace pour l'exécution de programmes impliquant un grand nombre de threads. Dans l'implémentation actuelle des FAIR THREADS, l'ordonnancement (au sein d'une zone synchrone) est basé sur un mécanisme plus complexe de files d'attente associées aux signaux. Plus précisément, les deux types d'ordonnancement existent dans l'implémentation ; cependant l'avantage du premier (déterminisme) ne reste valable que pour les programmes de la partie purement coopérative. La sémantique présentée dans la section suivante ne prend pas en compte le mécanisme de files d'attente mis en oeuvre dans les FAIR THREADS. Il nous semble en effet que, dans la sémantique et du fait de la complexité des synchronisations mises en oeuvre, un mécanisme d'élection basé sur un choix non déterministe dans la sémantique soit plus judicieux. A noter cependant que dans le cadre d'une implémentation du langage nous choisirions probablement ce mécanisme de files d'attentes. Dans le cadre de la partie purement coopérative, celui-ci conduit à des comportements reproductibles ce qui représente déjà un avantage pour l'utilisateur.

Intuitivement, dans un modèle à base de threads préemptifs, l'ordonnancement des threads doit être équitable (*fair*), i.e. chaque thread doit pouvoir bénéficier raisonnablement des processeurs. Cependant, cette spécification reste vague ; les comportements observés à l'exécution en témoignent. Dans un modèle à base de threads coopératifs, un thread doit toujours rendre le contrôle après un temps fini. On fait ici une hypothèse plus forte qui est que tout instant termine après un temps fini (en pratique, cela suppose d'étendre les résultats de l'analyse statique développée dans la partie I pour le $S\pi$-calcul). Une conséquence de cette hypothèse est que l'ordonnancement coopératif des threads au sein d'une zone synchrone est relativement libre. En effet, il n'est pas nécessaire de garantir une propriété de *fairness* puisque la fin de l'instant suppose que chaque thread a eu l'opportunité de mener à bien son exécution. Dans le cas d'un modèle à base de threads préemptifs, garantir que chaque thread ne détient un verrou que pendant un temps fini ne suffit pas à assurer cette propriété de *fairness*. De la même manière, en PACT, une telle propriété est bien sûr toujours nécessaire au niveau préemptif (entre zones synchrones et thread libres) et devrait être vérifiée par toute implémentation du langage.

Sans création dynamique de signaux ni de threads, les deux hypothèses (chaque thread coopère en temps fini et terminaison des instants) sont très probablement équivalentes (cela serait toutefois à vérifier formellement). En revanche, grâce à la création dynamique de signaux on peut écrire le programme $\texttt{link}\ \ell\ A()$ où :

$$A() = \texttt{yield}_\ell; A() \qquad \texttt{yield}_\ell \equiv (x = sig) \cdot ((\texttt{run}\ \texttt{link}\ \ell\ \texttt{emit}\ x\ *); \texttt{await}\ \texttt{x})$$

qui est tel que A coopère un nombre infini de fois dans le même instant. En particulier, en l'absence de création dynamique de signaux et de threads la non terminaison de l'instant implique l'existence d'un thread ne coopérant pas. En revanche, dans le cas contraire, la non terminaison de l'instant peut résulter soit d'un protocole plus ou moins complexe mis en oeuvre par plusieurs threads soit d'une chaîne infinie de création de threads.

8.1.5 Constructions dérivées

Avant d'introduire la sémantique formelle de PACT nous introduisons quelques constructions dérivées à partir des primitives du langage. On note $(x_1, \ldots, x_n = sig) \cdot P$ pour $(x_1 = sig) \cdot \ldots (x_n = sig) \cdot P$. Un signal peut être utilisé pour une simple synchronisation ne nécessitant pas de lire où d'écrire une valeur ; dans ce cas, on utilisera les constructions suivantes :

$$\texttt{await}\ e \equiv (x = \texttt{await}\ e) \cdot 0 \qquad \texttt{emit}\ e \equiv \texttt{emit}\ e\ *$$

La construction `pause` permet à un thread de suspendre son exécution jusqu'à l'instant suivant ; elle est définie par :

$$\texttt{pause} \equiv (x, y = \texttt{sig}) \cdot \texttt{watch}\ x\ \texttt{emit}\ x; \texttt{await}\ y, 0$$

Le branchement conditionnel peut facilement être défini à partir du filtrage de la manière suivante :

$$\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \equiv [e \trianglerighteq True]\ P_1, P_2$$

où il est entendu que les filtres $False$ et $True$ sont, par convention, définis par $False = \mathsf{nil}$ et $True = \mathsf{cons}(*, x)$ pour une variable fraîche x.

8.1.6 Comparaison avec les FAIR THREADS

Dans les FAIR THREADS, les primitives d'accès aux zones synchrones n'ont pas la forme de blocs de contrôle. Le choix fait dans PACT, par rapport aux FAIR THREADS, est similaire à celui de [44], par rapport aux primitives usuelles `lock` et `unlock` de gestion des verrous ; ces dernières sont également remplacées par des blocs de contrôle de la forme `lock`{*...code...*}. Les primitives des FAIR THREADS offrent un pouvoir expressif plus important mais introduisent des mécanismes de synchronisation beaucoup plus difficiles à contrôler statiquement.

La construction `pre` n'existe pas dans les FAIR THREADS. En revanche, ceux-ci proposent une primitive permettant de lire la *i*-ème valeur émise dans l'instant courant ; cela permet par une simple boucle de récupérer l'ensemble des valeurs. Cette approche est difficilement compatible avec nos méthodes d'analyse statique pour le contrôle des ressources. Dans un environnement ouvert, le nombre de valeurs émises sur un signal durant un instant ne peut être déduit du code du programme.

Dans les FAIR THREADS, il existe une primitive qui permet d'émettre, depuis une zone synchrone, un signal sur une autre zone synchrone. Dans ce cas, la valeur est mise en attente pour être émise localement lorsque la zone destinataire démarrera son prochain instant. Cette construction est ici rejetée car elle nous poserait un problème en terme de contrôle des ressources : elle supposerait la mise en place d'un buffer pour stocker les valeurs. D'une part, un buffer non borné rendrait difficile l'évaluation des ressources nécessaires à l'exécution d'un programme. D'autre part, un buffer borné supposerait qu'un thread voulant émettre à distance soit suspendu si le buffer est plein. Dans le second cas, un comportement similaire peut être obtenu en synchronisant le thread avec la zone destinataire.

Dans les FAIR THREADS, un signal est associé par l'utilisateur, lors de sa création, à une zone synchrone donnée. L'émission (ou la réception) de ce signal par un thread n'est alors possible que si celui-ci est soumis à l'influence de cette zone. Dans le cas contraire, un code d'erreur est retourné et aucune valeur n'est émise (ou reçue). Dans PACT, un signal est vu comme une *clé* pouvant être utilisée pour communiquer au sein d'une zone synchrone quelconque. Ce choix nous semble plus modulaire.

8.2 Sémantique

Dans cette section, on définit la sémantique du langage PACT au moyen d'une relation de réduction. L'ordonnancement des threads défini par cette relation repose sur les deux niveaux déjà évoqués de manière informelle :

– Au plus haut niveau, le contrôle est partagé par *interleaving* entre les zones synchrones et les threads libres.
– Au sein de chaque zone le contrôle est transmis selon un mode coopératif aux threads soumis à l'influence de celle-ci. Ces threads partagent les instants et la visibilité des signaux associés à cette zone.

On donne en premier lieu plusieurs définitions (8.2.1) nécessaires à l'introduction de la relation de réduction (8.2.2).

8.2.1 Définitions

La relation de réduction est définie sur des *systèmes* et, de manière standard, repose sur les notions de contexte d'évaluation et d'*expression réductible*. Intuitivement, un système est un ensemble de zones synchrones et de threads libres tels que chaque zone synchrone contrôle l'exécution des threads soumis à son influence. A chaque thread est associée une *zone cible* (éventuellement aucune, dans ce cas le thread doit être *libéré*), pouvant changer au cours du temps ; cette zone cible est celle au sein de laquelle l'exécution du thread peut se poursuivre. L'exécution d'un thread peut être suspendue si ce thread est soit en attente de l'émission d'un signal soit en attente d'une nouvelle synchronisation. Finalement, lorsque la fin d'un instant est décidée pour une zone synchrone, une phase de préemption est réalisée afin de préparer le calcul de l'instant suivant. Ci-dessous, on définit successivement :

- la notion de système
- les contextes d'évaluation et l'évaluation des expressions
- les notions de zone cible, de suspension et de préemption

Un *système* est constitué de plusieurs zones synchrones et threads (libres), d'un état de la mémoire et d'un état des signaux. Ces différents éléments sont définis de la manière suivante :

- Un thread est une paire (P,t), noté P^t, telle que t est un *nom de thread*, i.e. un élément d'un ensemble infini dénombrable donné $\mathcal{T} = \{t, t', \ldots\}$ (les noms de threads ne font pas partie du langage et seront simplement utilisés par la suite pour définir certaines propriétés des programmes).
- Une *zone synchrone* est un triplet (T, T', ℓ) noté $\langle T, T' \rangle_\ell$. Intuitivement, ℓ est le nom de la zone synchrone, T et T' sont des listes de threads. On note $P^t \cdot T$ la liste constituée du programme P^t et de la liste T et $\emptyset$ la liste vide. Intuititvement, T est la liste des threads soumis à l'influence de la zone ℓ et T' est la liste des threads en attente de l'instant suivant de la zone ℓ.
- L'*état de la mémoire* est dénoté par une fonction partielle $\mathcal{H} : Ref \to \mathcal{V}$.
- L'*état des signaux* est dénoté par une fonction partielle $\mathcal{E} : Sig \times \mathcal{L} \to \mathcal{M}_f(\mathcal{V}) \times \mathcal{V}$ où $\mathcal{M}_f(\mathcal{V})$ dénote l'ensemble des multi-ensembles finis d'éléments de $\mathcal{V}$.

Un système, noté $S, S', \ldots$, est un triplet $R, \mathcal{H}, \mathcal{E}$ où R est défini par la grammaire suivante

$$R ::= \langle T, T' \rangle_\ell \mid P^t \mid (R \mid R)$$

et où $\mathcal{H}$ et $\mathcal{E}$ sont définis comme précédement. Intuitivement, si $\mathcal{E}(s, \ell) = (M, v)$ alors

- M est le multi-ensemble des valeurs émises pour l'instant courant de ℓ

- v est une liste de valeurs *représentant* le multi-ensemble des valeurs émises pour l'instant précédent de ℓ.

Si $\mathcal{E}(s,\ell) = (M, v)$, on notera plus simplement $\mathcal{E}_c(s,\ell)$ pour M et $\mathcal{E}_p(s,\ell)$ pour v. On notera $\mathcal{H}_\emptyset$ la fonction définie par $Dom(\mathcal{H}_\emptyset) = \emptyset$ et $\mathcal{E}_\emptyset$ la fonction définie par $Dom(\mathcal{E}_\emptyset) = \emptyset$ où $Dom(f)$ dénote le domaine de f. Finalement, on suppose que dans tout système, chaque nom de zone synchrone est associé à une zone synchrone au plus et on identifie les systèmes au moyen de la relation d'équivalence $\equiv$ qui est la plus petite relation vérifiant :

$$R \mid 0 \equiv R \qquad R_1 \mid R_2 \equiv R_2 \mid R_1 \qquad (R_1 \mid R_2) \mid R_3 \equiv R_1 \mid (R_2 \mid R_3)$$

La relation de réduction qui sera définie par la suite le sera modulo cette relation d'équivalence.

Contextes d'évaluation. Les contextes d'évaluations, notés $E, E', \ldots$ et les expressions réductibles, notées $pr, pr', \ldots$ sont donnés par les grammaires suivantes :

$$\begin{array}{rcl} E & ::= & [\,] \mid E; P \mid \texttt{watch}\ e\ E, P \mid \texttt{link}\ \ell\ E \mid \texttt{unlink}\ E \\ \\ pr & ::= & A(\mathbf{e}) \mid [e = p] \cdot P,\ P' \\ & & \mid (x = \texttt{ref}\ e) \cdot P \mid \texttt{set}\ e\ e \mid (x = \texttt{get}\ e) \cdot P \mid 0; P \\ & & \mid (x = \texttt{sig}) \cdot P \mid \texttt{emit}\ e\ e \mid (x = \texttt{await}\ e) \cdot P \mid (x = \texttt{pre}\ e) \cdot P \\ & & \mid \texttt{watch}\ e \cdot 0, P \mid \texttt{link}\ \ell\ 0 \mid \texttt{unlink}\ 0 \\ & & \mid \texttt{run}\ A(\mathbf{e}) \end{array}$$

On peut noter que dans la définition des redex les expressions ne sont pas nécessairement des valeurs. Cependant, par définition des expressions et par hypothèse sur les symboles de fonctions, il est facile de vérifier que pour toute expression close il existe une valeur v telle l'on a toujours $e \Downarrow v$ où la relation $\Downarrow$ est définie par :

$$\frac{}{s \Downarrow s} \qquad \frac{}{r \Downarrow r} \qquad \frac{}{\mathsf{nil}_\Downarrow \mathsf{nil}} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\mathsf{cons}(e_1, e_2) \Downarrow \mathsf{cons}(v_1, v_2)} \qquad \frac{\mathbf{e} \Downarrow \mathbf{v} \qquad [\![f]\!](\mathbf{v}) = v}{f(\mathbf{e}) \Downarrow v}$$

où, pour $\mathbf{e} = e_1, \ldots, e_n$, on note $\mathbf{e} \Downarrow \mathbf{v}$ la suite $e_1 \Downarrow v_1, \ldots, e_n \Downarrow v_n$. Le premier résultat donné ci-dessous est immédiat.

Synchronisation. La *cible* d'un thread P est un élément de $\mathcal{L} \cup \{free\}$ où $free$ est une constante spéciale dénotant l'absence de synchronisation. La relation $\odot\!\!\rightarrow\ \alpha$ $\alpha \in \mathcal{L} \cup \{free\}$ (lire α est la cible de P) est défini par :

$$0 \odot\!\!\rightarrow free \qquad E[r] \odot\!\!\rightarrow \alpha, \text{ si } E \odot\!\!\rightarrow \alpha$$

où $E \odot\!\!\rightarrow \alpha$ est définie par cas sur E par :

$$\alpha = \begin{cases} free & \text{si } E = [\,] \text{ ou } E = E'[\texttt{unlink}\ [\,]] \\ \ell & \text{si } E = E'[\texttt{link}\ \ell\ [\,]] \\ \alpha' & \text{si } E = E'[[\,]; P] \text{ et } E' \odot\!\!\rightarrow \alpha' \\ \alpha' & \text{si } E = E'[\texttt{watch}\ e \cdot [\,], P'] \text{ et } E' \odot\!\!\rightarrow \alpha' \end{cases}$$

On étend simplement la définition aux listes de thread (on a alors une liste de cible) par :

$$\emptyset \odot\!\!\rightarrow \emptyset \qquad (P^t \cdot T \odot\!\!\rightarrow \{\alpha\} \cup L) \text{ si } P \odot\!\!\rightarrow \alpha \text{ et } T \odot\!\!\rightarrow L$$

Suspension. Un thread soumis à l'influence d'une zone synchrone est *suspendu* si l'une des deux conditions suivantes est vérifiée :

1. il est en attente d'un signal absent,
2. son exécution doit se poursuivre sous l'influence d'une autre zone synchrone (ou en étant libre)

Cela s'exprime formellement par le prédicat $(P, \mathcal{E}, \ell) \downarrow$ (lire : P est suspendu en ℓ pour $\mathcal{E}$) défini par :

$$\frac{P \odot\!\!\rightarrow \ell \qquad P = E[(x = \texttt{await}\ e) \cdot P'] \qquad e \Downarrow s \qquad \mathcal{E}_c(s)(\ell) = \emptyset}{(P, \mathcal{E}, \ell) \downarrow} \qquad \frac{P \odot\!\!\rightarrow \alpha \qquad \alpha \neq \ell}{(P, \mathcal{E}, \ell) \downarrow}$$

On note $(T, \mathcal{E}, \ell) \downarrow$ si tous les éléments de T sont suspendus en ℓ pour $\mathcal{E}$.

Préemption. Intuitivement, la fin de l'instant pour une zone synchrone est décidée lorsque les deux conditions suivantes sont vérifiées :

- L'ensemble des threads soumis à l'influence de cette zone sont suspendus (ou en attente de l'instant suivant, pour les threads arrivant d'une autre zone synchrone).
- L'ensemble des threads qui sont en attente d'une *synchronisation* avec une autre zone synchrone (ou aucune) ont quitté la zone concernée.

Ainsi, lorsque la fin de l'instant est décidée pour une zone synchrone, l'ensemble des threads soumis à son influence sont de la forme $E[(x = \texttt{await}\ e) \cdot P]$. Dans ce cas, la continuation du thread pour l'instant suivant est :

- celle du bloc de préemption de E le plus externe tel que le signal associé soit présent s'il existe,
- le thread lui même dans le cas contraire.

Lorsque la fin de l'instant est décidée pour une zone synchrone, un calcul est effectué pour déterminer les continuations pour l'instant suivant de l'ensemble des threads soumis à son influence. Ce calcul est défini formellement par la fonction $(\!|P|\!)_{\mathcal{E},\ell}$ ci dessous.

$$(\!|E[pr]|\!)_{\mathcal{E},\ell} = \begin{cases} pr & \text{si } E = [\,] \\ (\!|E'[pr]|\!)_{\mathcal{E},\ell}; P' & \text{si } E = E'; P' \\ P' & \text{si } E = \texttt{watch}\ e \cdot E', P',\ e \Downarrow s \text{ et } \mathcal{E}_c(s)(\ell) \neq \emptyset \\ \texttt{watch}\ e \cdot (\!|E'[pr]|\!)_{\mathcal{E},\ell}, P' & \text{si } E = \texttt{watch}\ e \cdot E', P',\ e \Downarrow s \text{ et } \mathcal{E}_c(s)(\ell) = \emptyset \\ \texttt{link}\ \ell'\ (\!|E'[pr]|\!)_{\mathcal{E},\ell} & \text{si } E = \texttt{link}\ \ell'\ E' \\ \texttt{unlink}\ (\!|E'[pr]|\!)_{\mathcal{E},\ell} & \text{si } E = \texttt{unlink}\ E' \end{cases}$$

La définition est étendue naturellement aux listes de threads par $(\!|\emptyset|\!)_{\mathcal{E},\ell} = \emptyset$ et $(\!|P^t \cdot T|\!)_{\mathcal{E},\ell} = (\!|P|\!)^t_{\mathcal{E},\ell} \cdot (\!|T|\!)_{\mathcal{E},\ell}$.

8.2.2 Relation de réduction

L'exécution d'un programme est définie par une relation de réduction sur les systèmes. Cette relation de réduction est de la forme

$$R, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} R', \mathcal{H}', \mathcal{E}'$$

où M est soit l'ensemble vide $\emptyset$, soit un singleton $\{(t, r)\}$ noté plus simplement (t, r). Intuitivement, $\{(t, r)\}$ est une annotation dénotant un accès à la référence r par le thread dont le nom est t lors de la réduction. Ces annotations seront utiles par la suite pour la définition d'une propriété d'atomicité

sur les programmes et pour l'analyse statique associée. Celles-ci n'influent pas sur la sémantique des programmes et peuvent, en première lecture, être ignorées.

Étant donné un programme P associé aux zones synchrones $\{\ell_1, \ldots, \ell_n\}$, le système initial pour l'exécution du programme est :

$$(\langle\emptyset,\emptyset\rangle_{\ell_1} \mid \ldots \mid \langle\emptyset,\emptyset\rangle_{\ell_n} \mid P^t), \mathcal{H}_\emptyset, \mathcal{E}_\emptyset \qquad t \in \mathcal{T} \text{ quelconque}$$

Afin de définir la relation de réduction on procède en trois étapes :

1. on définit le comportement individuel des threads
2. on définit le contrôle synchrone, i.e. la coopération des threads et la fin des instants
3. on définit la coordination du système, i.e. des zones synchrones et des threads libres.

1) Comportement individuel des threads

On définit une relation auxiliaire sur les expressions réductibles, notée :

$$\vdash_{t,\alpha} P, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} P', \mathcal{H}', \mathcal{E}'$$

où t, α et M sont des annotations telles que :

- $\alpha \in \mathcal{L} \cup \{free\}$ est la cible de t au moment de la réduction. Cette annotation permet simplement d'associer le comportement d'un thread à l'état des signaux associé à sa cible.
- t est le nom du thread associé à la réduction et M est défini comme précédemment.

Cette relation auxiliaire est utilisée, pour la réduction des redex (exception faite de `run` A(**e**)), dans les règles (*context*$_1$) (threads liés à une zone synchrone) et (*context*$_2$) (threads libres). Le cas d'un redex de la forme `run` A(**e**) est traité séparément après la définition des règles auxiliaires (règles (*thread*$_1$) et (*thread*$_2$)).

$$\frac{E \odot\!\!\rightarrow \ell \qquad \vdash_{t,\ell} P, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} P', \mathcal{H}', \mathcal{E}'}{\langle E[P]^t \cdot T, T'\rangle_\ell \mid R, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} \langle E[P']^t \cdot T, T'\rangle_\ell \mid R, \mathcal{H}', \mathcal{E}'} \qquad (context_1)$$

$$\frac{E \odot\!\!\rightarrow free \qquad \vdash_{t,free} P, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} P', \mathcal{H}', \mathcal{E}'}{E[P]^t \mid R, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} E[P']^t \mid R, \mathcal{H}', \mathcal{E}'} \qquad (context_2)$$

On peut noter que, dans tous les cas, la réduction ne peut avoir lieu que si le thread est soumis à l'influence de sa cible (si celle-ci est $free$, le thread doit être libre).

Avant de présenter les règles de réduction auxiliaires, on commence par introduire certaines notations nécessaires à la définition des règles de celles-ci.réduction auxiliaires ; ces notations concernent la transformation de la mémoire et de l'environnement de signaux au cours de l'exécution.

a) Création d'un référence. Si $r \notin Dom(\mathcal{H})$, on note $(\mathcal{H} \cup \{r \mapsto v\})$ la fonction dont le domaine de définition est $Dom(\mathcal{H}) \cup \{r\}$ qui est égale à $\mathcal{H}$ sur $Dom(\mathcal{H})$ et qui vérifie $(\mathcal{H} \cup \{r \mapsto v\})(r) = v$.

b) Mise à jour d'une référence. Si $r \in Dom(\mathcal{H})$, on note $(\mathcal{H}[r := v])$ la fonction égale à $\mathcal{H}$ pour tout $r' \neq r$ et telle que $(\mathcal{H} \cup \{r := v\})(r) = v$.

c) Création d'un signal. Si $s \notin Dom(\mathcal{E})$, on note $(\mathcal{E} \cup \{s\})$ la fonction dont le domaine est $Dom(\mathcal{E}) \cup \{s\})$, qui est égale à $\mathcal{E}$ sur $Dom(\mathcal{E})$ et telle que pour tout $\ell \in \mathcal{L}$ on a $(\mathcal{E} \cup \{s\})(s)(l) = (\emptyset, \mathsf{nil})$ où $\emptyset$ dénote le multi-ensemble vide. Initialement, le multi-ensemble des valeurs émises par un signal et la liste

des valeurs émises à l'instant précédent pour l'instant courant sont vides pour l'ensemble des zones synchrones.

d) Émission d'un signal. Si $s \in Dom(\mathcal{E})$ et si ℓ est un nom de zone synchrone, on note $\mathcal{E}[s \twoheadleftarrow_\ell v]$ la fonction telle que

$$\mathcal{E}[s \twoheadleftarrow_\ell v](s)(\ell) = \begin{cases} (\mathcal{E}_c(s)(\ell) \uplus \{v\}, \mathcal{E}_p(s)(\ell)) \\ \mathcal{E}(s')(\ell') \text{ si } s' \neq s \text{ ou } \ell \neq \ell' \end{cases}$$

e) Fin d'instant. On définit la relation $\mapsto_\ell$ par $\mathcal{E} \mapsto_\ell \mathcal{E}'$ si et seulement si : (1) $\mathcal{E}$ et $\mathcal{E}'$ ont le même domaine, (2) si $\ell' \neq \ell$ alors $(\mathcal{E}(s)(\ell') = \mathcal{E}'(s)(\ell'))$ pour tout $s \in Dom(\mathcal{E})$, et (3) pour tout $s \in Dom(\mathcal{E})$ on a $\mathcal{E}'(s)(\ell) = (\emptyset, v)$ où $v = [v_{\sigma(1)}; v_{\sigma(2)}; \ldots, v_{\sigma(n)}]$ avec σ une permutation quelconque sur $\{1, \ldots, n\}$ et $\mathcal{E}_c(s)(\ell) = \{v_1, \ldots, v_n\}$. Intuitivement, à la fin d'un instant, pour chaque signal le multi-ensemble des valeurs de l'instant courant est *vidé* et la valeur du signal pour l'instant précédent est un représentant (une liste), obtenu de manière non-déterministe, du multi-ensemble des valeurs émises au cours de celui-ci.

On peut maintenant présenter les règles de réduction auxiliaires ; ces règles peuvent être regroupées en trois groupes : (1) le contrôle, (2) la manipulation des références et (3) la manipulation des signaux.

Contrôle. Les règles (*call*) (appel de fonction), ($match_1$) (réussite du filtrage), ($match_2$) (échec du filtrage) et (*seq*) (séquence) sont définies comme on peut s'y attendre. Ces règles s'appliquent quelque soit l'état de synchronisation du thread concerné par la réduction.

$$\frac{A(\mathbf{x}) = P \qquad \mathbf{e} \Downarrow \mathbf{v} \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} A(\mathbf{e}), \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P[\mathbf{v}/\mathbf{x}], \mathcal{H}, \mathcal{E}} \qquad (call)$$

$$\frac{e \Downarrow v \qquad match(v, p) = \sigma \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} [e \trianglerighteq p]\ P_1, P_2, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P_1\sigma, \mathcal{H}, \mathcal{E}} \qquad (match_1)$$

$$\frac{e \Downarrow v \qquad match(v, p) = \uparrow \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} [e \trianglerighteq p]\ P_1, P_2, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P_2, \mathcal{H}, \mathcal{E}} \qquad (match_2)$$

$$\frac{\alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} 0; P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P, \mathcal{H}, \mathcal{E}} \qquad (seq)$$

En revanche, les règles (*watch*), (*link*) et (*unlink*) dépendent de la *synchronisation* courante du thread. La terminaison d'un bloc de préemption requiert que le thread concerné soit soumis à l'influence d'une zone synchrone. Cette contrainte pourrait être relaxé mais on choisit, de manière générale, d'imposer que les opérations associées à la notion de signal soit réalisées sous l'influence d'une zone synchrone.

$$\frac{e \Downarrow s \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} \texttt{watch}\ e\ 0, P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} 0, \mathcal{H}, \mathcal{E}} \qquad (watch)$$

La terminaison d'un bloc *link* associé à une zone synchrone ℓ ne peut se faire que si le thread concerné est soumis à l'influence de cette zone.

$$\frac{}{\vdash_{t,\ell} \texttt{link}\ \ell\ 0, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} 0, \mathcal{H}, \mathcal{E}} \quad (\textit{link})$$

De la même manière, la terminaison d'un bloc *unlink* ne peut se faire que lorsque le thread est libre.

$$\frac{}{\vdash_{t,free} \texttt{unlink}\ 0, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} 0, \mathcal{H}, \mathcal{E}} \quad (\textit{unlink})$$

Références. Les règles (*ref*), (*set*) et (*get*) définissent respectivement la création, la mise à jour et la lecture d'une référence ; ces règles s'appliquent quelque soit l'état de synchronisation du thread concerné et sont définies ci-dessous :

$$\frac{r \notin Dom(\mathcal{H}) \qquad e \Downarrow v \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} (x = \texttt{ref}\ e) \cdot P, \mathcal{H}, \mathcal{E} \underset{(t,r)}{\longrightarrow} P[r/x], \mathcal{H} \cup \{r \mapsto v\}, \mathcal{E}} \quad (\textit{ref})$$

$$\frac{e_1 \Downarrow r \quad r \in Dom(\mathcal{H}) \qquad e_2 \Downarrow v \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} \texttt{set}\ e_1\ e_2, \mathcal{H}, \mathcal{E} \underset{(t,r)}{\longrightarrow} 0, \mathcal{H}[r := v], \mathcal{E}} \quad (\textit{set})$$

$$\frac{e \Downarrow r \qquad \mathcal{H}(r) = v \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} (x = \texttt{get}\ e) \cdot P, \mathcal{H}, \mathcal{E} \underset{(t,r)}{\longrightarrow} P[v/x], \mathcal{H}, \mathcal{E}} \quad (\textit{get})$$

Signaux. Les règles (*sig*), (*emit*), (*await*) et (*pre*) définissent respectivement la création, l'émission, la lecture de la valeur (partielle) courante et la lecture de la valeur précédente d'un signal. Ces règles ne s'appliquent que lorsque le thread est soumis à l'influence d'une zone synchrone. Lors de la création d'un signal sa valeur pour l'instant courant (resp. pour l'instant précédent) est le multi-ensemble vide (resp. `nil`).

$$\frac{s \notin Dom(\mathcal{E}) \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} (x = \texttt{sig}) \cdot P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P[s/x], \mathcal{H}, \mathcal{E} \cup \{s\}} \quad (\textit{sig})$$

L'émission d'une valeur sur un signal par un thread soumis à l'influence d'une zone synchrone diffuse cette valeur à l'ensemble des threads soumis à cette même zone.

$$\frac{e_1 \Downarrow s \qquad e_2 \Downarrow v \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} \texttt{emit}\ e_1\ e_2, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} 0, \mathcal{H}, \mathcal{E}[s \twoheadleftarrow_\ell v]} \quad (\textit{emit})$$

Un thread soumis à l'influence d'une zone synchrone ℓ accède de manière non déterministe à l'une des valeurs émise sur un signal s, s'il en existe au moins une, par la construction `await`.

$$\frac{e \Downarrow s \qquad v \in \mathcal{E}_c(s)(\ell) \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} (x = \texttt{await}\ e) \cdot P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P[v/x], \mathcal{H}, \mathcal{E}} \quad (\textit{await})$$

Un thread soumis à l'influence d'une zone synchrone ℓ récupère une liste représentant le multi-ensemble des valeurs émises sur un signal s à l'instant précédent de ℓ par la construction `pre`.

$$\frac{e \Downarrow s \qquad \mathcal{E}_p(s)(\ell) = v \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} (x = \texttt{pre}\ e) \cdot P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P[v/x], \mathcal{H}, \mathcal{E}} \qquad (pre)$$

Création de threads. Pour terminer la description du comportement individuel des threads, il nous reste à voir comment un nouveau thread est crée ; cela est défini par les règles ($thread_1$) (thread soumis à l'influence d'une zone synchrone) et ($thread_2$) (thread libre) données ci-dessous ; ces règles suivent le même découpage que les règles ($context_1$) et ($context_2$) ; de la même manière, la réduction n'est définie que lorsque le thread est soumis à l'influence de sa cible (libre quand celle-ci est $free$).

$$\frac{E \odot\!\!\rightarrow \ell \qquad A(\mathbf{x}) = P \qquad \mathbf{e} \Downarrow \mathbf{v} \qquad t'\ fresh}{\langle E[\texttt{run}\ A(\mathbf{e})]^t \cdot T, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} \langle E[0]^t \cdot T \cdot (P[\mathbf{v}/\mathbf{x}])^{t'}, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E}} \qquad (thread_1)$$

$$\frac{E \odot\!\!\rightarrow free \qquad A(\mathbf{x}) = P \qquad \mathbf{e} \Downarrow \mathbf{v} \qquad t'\ fresh}{E[\texttt{run}\ A(\mathbf{e})]^t \mid R, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} E[0]^t \mid (P[\mathbf{v}/\mathbf{x}])^{t'} \mid R, \mathcal{H}, \mathcal{E}} \qquad (thread_2)$$

Lorsqu'un thread est créé par un thread soumis à l'influence d'une zone synchrone, le nouveau thread est également soumis à l'influence de cette zone. Cependant, si la cible du nouveau thread est une autre zone synchrone (ou aucune) son exécution est suspendue en attente d'une synchronisation avec cette cible.

2) Contrôle synchrone

La règle ($context_1$), présentée précédemment, décrit le comportement d'un thread au sein d'une zone synchrone lorsque celui-ci possède le contrôle. Lorsque, au sein d'une zone synchrone ℓ, le thread ayant le contrôle ne peut plus progresser on distingue trois cas :

- un autre thread peut progresser et reçoit le contrôle (règle ($cooperate$)),
- aucun thread ne peut progresser et l'ensemble des threads ont pour cible la zone synchrone concernés ; dans ce cas, l'instant termine (règle (eoi)),
- aucun thread ne peut progresser et certains threads ont pour cible une autre zone synchrone ; ce cas sera traité plus loin par les règles dédiées à la coordination du système.

On note $T \triangleq T'$ si T' est une permutation quelconque des éléments de T. Le passage du contrôle à un nouveau thread est alors défini par la règle ci-dessous :

$$\frac{(P, \mathcal{E}, \ell) \downarrow \qquad T \triangleq P' \cdot T'' \qquad \neg((P', \mathcal{E}, \ell) \downarrow)}{\langle P \cdot T, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} \langle P' \cdot T'' \cdot P, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E}} \qquad (cooperate)$$

Lorsque la fin de l'instant est décidée, plusieurs calculs sont nécessaires à la préparation de l'exécution du prochain instant :

- on vérifie que l'ensemble des threads ont pour cible la zone concernée,
- les continuations de ces threads pour l'instant suivant sont calculées et ajoutées à l'ensemble des threads en attente de l'instant suivant,
- l'environnement de signaux (pour cette zone synchrone) est réinitialisé.

$$\frac{(T,\mathcal{E},\ell)\downarrow \qquad T \odot\!\!\rightarrow \{\ell\} \qquad T'' \triangleq (\!|T|\!)_{\mathcal{E},\ell}\cdot T' \qquad \mathcal{E}\mapsto_\ell \mathcal{E}'}{\langle T,T'\rangle_\ell \mid R,\mathcal{H},\mathcal{E} \underset{\emptyset}{\rightarrow} \langle T'',\emptyset\rangle_\ell \mid R,\mathcal{H},\mathcal{E}'} \quad (\textit{eoi})$$

3) Coordination du système

La coordination des composantes d'un système est définie par les règle (*move*), (*out*) et (*in*). Ces règles décrivent respectivement :

- comment un thread passe d'une zone synchrone à une autre,
- comment un thread devient libre,
- comment un thread libre se soumet à l'influence d'une zone synchrone,

Deux points sont à noter sur la synchronisation des threads :

1. Si la zone d'influence d'un thread est une zone synchrone alors celui-ci doit attendre la fin de l'instant courant pour la quitter (règles (*move*) et (*out*)). Dans le cas de la règle (*move*), ce choix permet de placer simultanément sous l'influence d'une nouvelle zone synchrone l'ensemble des threads ayant désigné la même cible dans l'instant courant (tant que l'instant n'est pas terminé, de nouveaux threads peuvent désigner cette même cible). Dans le cas de la règle (*out*), cette restriction n'est là que par souci de cohérence et pourrait être relaxée.
2. L'entrée dans une nouvelle zone synchrone se fait par l'intermédiaire de la *file d'attente* associée à cette zone (règles (*move*) et (*in*)) ; cela permet de retarder l'exécution des threads entrants à l'instant suivant.

Dans la règle (*move*) on demande que l'ensemble des threads s'exécutant dans la zone synchrone ℓ_1 soient suspendus (prédicat $(T_1,\mathcal{E},\ell)\downarrow$). Les conditions supplémentaires dans la prémisse permettent de distinguer l'ensemble des threads ayant pour cible ℓ_2. En particulier, la condition $U' \neq \emptyset$ n'autorise qu'au plus un déplacement de threads vers ℓ_2 ; sans cette condition, la règle pourrait s'appliquer indéfiniment. Finalement, l'ensemble des threads ayant pour cible ℓ_2 est ajouté à la file d'attente de ℓ_2.

$$\frac{(T_1,\mathcal{E},\ell_1)\downarrow \qquad T_1 \triangleq U\cdot U',\ U'\neq\emptyset \qquad U \odot\!\!\rightarrow L \qquad \ell_2 \notin L \qquad U' \odot\!\!\rightarrow \{\ell_2\}}{\langle T_1,T_1'\rangle_{\ell_1} \mid \langle T_2,T_2'\rangle_{\ell_2} \mid R,\mathcal{H},\mathcal{E} \underset{\emptyset}{\rightarrow} \langle U,T_1'\rangle_{\ell_1} \mid \langle T_2,T_2'\cdot U'\rangle_{\ell_2} \mid R,\mathcal{H},\mathcal{E}} \quad (\textit{move})$$

La règle (*out*) est similaire mais s'applique à la libération d'un thread.

$$\frac{(T,\mathcal{E},\ell)\downarrow \qquad T \triangleq P^t\cdot T'' \qquad P \odot\!\!\rightarrow \mathit{free}}{\langle T,T'\rangle_\ell \mid R \underset{\emptyset}{\rightarrow} \langle T'',T'\rangle_\ell \mid P^t \mid R,\mathcal{H},\mathcal{E}} \quad (\textit{out})$$

La règle (*in*) décrit l'insertion d'un thread libre dans la file d'attente de la zone synchrone qu'il a pour cible.

$$\frac{P \odot\!\!\rightarrow \ell}{\langle T,T'\rangle_\ell \mid P^t \mid R,\mathcal{H},\mathcal{E} \underset{\emptyset}{\rightarrow} \langle T,T'\cdot P^t\rangle_\ell \mid R,\mathcal{H},\mathcal{E}} \quad (\textit{in})$$

L'ensemble des règles de la sémantique est résumé dans les figures 8.2 et 8.3.

Exemple 14. *Considérons l'exemple suivant. Intuitivement, le programme $A(s_1,s_2,s_3,s_4)$ se synchronise avec ℓ_2, émet le signal s_1 et attend une réponse sur le signal s_2. Si cette réponse arrive, alors le*

thread se synchronise avec ℓ_1 et transmet (localement) la valeur reçue sur s_2. Si cette réponse n'arrive pas, et que le signal s_3 et émis alors le thread termine à la fin de l'instant.

$$\begin{aligned} A(s_1, s_2, s_3) \quad = \quad & \texttt{link}\ \ell_2 \\ & \quad \texttt{emit}\ s_1; \\ & \quad \texttt{watch}\ s_3 \cdot y = \texttt{await}\ s_2 \cdot \texttt{link}\ \ell_1\ \texttt{emit}\ s_2\ y, 0 \end{aligned}$$

Regardons plus précisément comment se déroule l'exécution. Initialement, le thread est libre. il est en attente d'une synchronisation avec ℓ_2. Lorsque celle-ci est réalisée, le thread émet s_1 et se met en attente du signal s_2, il est donc suspendu si celle-ci n'est pas émise. On distingue deux cas :

1. *si le signal est émis, le thread réalise la lecture et est suspendu en attente de la fin de l'instant. A la fin de l'instant, une synchronisation avec ℓ_1 permet au thread de changer de zone synchrone, il est enregistré dans la file d'attente de s_1, il réalisera l'émission de la réponse à l'instant suivant et terminera.*
2. *Si le signal n'est pas émis, alors tant que s_3 n'est pas émis non plus, le thread est suspendu sur ℓ_2. Si s_3 est émis, à la fin de l'instant, la préemption de l'attente du signal est réalisée et le thread termine.*

$$\frac{A(\mathbf{x}) = P \qquad \mathbf{e} \Downarrow \mathbf{v} \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} A(\mathbf{e}), \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P[\mathbf{v}/\mathbf{x}], \mathcal{H}, \mathcal{E}} \quad (call)$$

$$\frac{e \Downarrow v \qquad match(v, p) = \sigma \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} [e \trianglerighteq p]\ P_1, P_2, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P_1\sigma, \mathcal{H}, \mathcal{E}} \quad (match_1)$$

$$\frac{e \Downarrow v \qquad match(v, p) = \uparrow \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} [e \trianglerighteq p]\ P_1, P_2, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P_2, \mathcal{H}, \mathcal{E}} \quad (match_2)$$

$$\frac{\alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} 0; P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P, \mathcal{H}, \mathcal{E}} \quad (seq)$$

$$\frac{e \Downarrow s \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} \texttt{watch}\ e\ 0, P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} 0, \mathcal{H}, \mathcal{E}} \quad (watch)$$

$$\frac{}{\vdash_{t,\ell} \texttt{link}\ \ell\ 0, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} 0, \mathcal{H}, \mathcal{E}} \quad (link)$$

$$\frac{}{\vdash_{t,free} \texttt{unlink}\ 0, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} 0, \mathcal{H}, \mathcal{E}} \quad (unlink)$$

$$\frac{r \notin Dom(\mathcal{H}) \qquad e \Downarrow v \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} (x = \texttt{ref}\ e) \cdot P, \mathcal{H}, \mathcal{E} \underset{(t,r)}{\longrightarrow} P[r/x], \mathcal{H} \cup \{r \mapsto v\}, \mathcal{E}} \quad (ref)$$

$$\frac{e_1 \Downarrow r \quad r \in Dom(\mathcal{H}) \qquad e_2 \Downarrow v \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} \texttt{set}\ e_1\ e_2, \mathcal{H}, \mathcal{E} \underset{(t,r)}{\longrightarrow} 0, \mathcal{H}[r := v], \mathcal{E}} \quad (set)$$

$$\frac{e \Downarrow r \qquad \mathcal{H}(r) = v \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} (x = \texttt{get}\ e) \cdot P, \mathcal{H}, \mathcal{E} \underset{(t,r)}{\longrightarrow} P[v/x], \mathcal{H}, \mathcal{E}} \quad (get)$$

$$\frac{s \notin Dom(\mathcal{E}) \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} (x = \texttt{sig}) \cdot P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P[s/x], \mathcal{H}, \mathcal{E} \cup \{s\}} \quad (sig)$$

$$\frac{e_1 \Downarrow s \qquad e_2 \Downarrow v \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} \texttt{emit}\ e_1\ e_2, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} 0, \mathcal{H}, \mathcal{E}[s \twoheadleftarrow_\ell v]} \quad (emit)$$

$$\frac{e \Downarrow s \qquad v \in \mathcal{E}_c(s)(\ell) \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} (x = \texttt{await}\ e) \cdot P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P[v/x], \mathcal{H}, \mathcal{E}} \quad (await)$$

$$\frac{e \Downarrow s \qquad \mathcal{E}_p(s)(\ell) = v \qquad \ell \in \mathcal{L}}{\vdash_{t,\ell} (x = \texttt{pre}\ e) \cdot P, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} P[v/x], \mathcal{H}, \mathcal{E}} \quad (pre)$$

FIG. 8.2 – Sémantique : règles auxiliaires

$$\frac{E \odot\!\!\rightarrow \ell \qquad \vdash_{t,\ell} P, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} P', \mathcal{H}', \mathcal{E}'}{\langle E[P]^t \cdot T, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} \langle E[P']^t \cdot T, T' \rangle_\ell \mid R, \mathcal{H}', \mathcal{E}'} \qquad (context_1)$$

$$\frac{E \odot\!\!\rightarrow free \qquad \vdash_{t,free} P, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} P', \mathcal{H}', \mathcal{E}'}{E[P]^t \mid R, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} E[P']^t \mid R, \mathcal{H}', \mathcal{E}'} \qquad (context_2)$$

$$\frac{E \odot\!\!\rightarrow \ell \qquad A(\mathbf{x}) = P \qquad \mathbf{e} \Downarrow \mathbf{v} \qquad t' \ fresh}{\langle E[\mathtt{run}\ A(\mathbf{e})]^t \cdot T, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} \langle E[0]^t \cdot T \cdot (P[\mathbf{v}/\mathbf{x}])^{t'}, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E}} \qquad (thread_1)$$

$$\frac{E \odot\!\!\rightarrow free \qquad A(\mathbf{x}) = P \qquad \mathbf{e} \Downarrow \mathbf{v} \qquad t' \ fresh}{E[\mathtt{run}\ A(\mathbf{e})]^t \mid R, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} E[0]^t \mid (P[\mathbf{v}/\mathbf{x}])^{t'} \mid R, \mathcal{H}, \mathcal{E}} \qquad (thread_2)$$

$$\frac{(P, \mathcal{E}, \ell) \downarrow \qquad T \triangleq P' \cdot T'' \qquad \neg((P', \mathcal{E}, \ell) \downarrow)}{\langle P \cdot T, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} \langle P' \cdot T'' \cdot P, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E}} \qquad (cooperate)$$

$$\frac{(T, \mathcal{E}, \ell) \downarrow \qquad T \odot\!\!\rightarrow \{\ell\} \qquad T'' \triangleq (\!|T|\!)_{\mathcal{E},\ell} \cdot T' \qquad \mathcal{E} \mapsto_\ell \mathcal{E}'}{\langle T, T' \rangle_\ell \mid R, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} \langle T'', \emptyset \rangle_\ell \mid R, \mathcal{H}, \mathcal{E}'} \qquad (eoi)$$

$$\frac{(T_1, \mathcal{E}, \ell_1) \downarrow \qquad T_1 \triangleq U \cdot U',\ U' \neq \emptyset \qquad U \odot\!\!\rightarrow L \qquad \ell_2 \notin L \qquad U' \odot\!\!\rightarrow \{\ell_2\}}{\langle T_1, T_1' \rangle_{\ell_1} \mid \langle T_2, T_2' \rangle_{\ell_2} \mid R, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} \langle U, T_1' \rangle_{\ell_1} \mid \langle T_2, T_2' \cdot U' \rangle_{\ell_2} \mid R, \mathcal{H}, \mathcal{E}} \qquad (move)$$

$$\frac{(T, \mathcal{E}, \ell) \downarrow \qquad T \triangleq P^t \cdot T'' \qquad P \odot\!\!\rightarrow free}{\langle T, T' \rangle_\ell \mid R \underset{\emptyset}{\rightarrow} \langle T'', T' \rangle_\ell \mid P^t \mid R, \mathcal{H}, \mathcal{E}} \qquad (out)$$

$$\frac{P \odot\!\!\rightarrow \ell}{\langle T, T' \rangle_\ell \mid P^t \mid R, \mathcal{H}, \mathcal{E} \underset{\emptyset}{\rightarrow} \langle T, T' \cdot P^t \rangle_\ell \mid R, \mathcal{H}, \mathcal{E}} \qquad (in)$$

FIG. 8.3 – Sémantique : règles de réduction

Chapitre 9

Atomicité

De manière générale, concevoir, valider et maintenir des programmes à base de threads est une tache difficile[63] ; principalement en raison des interactions, non maitrisées, des composantes. Le problème le plus connu est probablement celui des *data-races* [86] qui se produisent lorsque deux composantes accédent (une au moins en écriture) simultanément à une même donnée ; ce qui peut mener à une corruption de cette donnée.

Plusieurs travaux proposent des méthodes de détection des data-races, statiques[44, 45, 31, 30] ou dynamiques[91, 95, 35], ce qui permet l'élimination de nombreuses erreurs de programmation. Cependant, l'utilisateur recherche généralement une propriété plus forte que l'absence de data-races. En effet, même lorsque les accès simultanés sont évités, des interactions non controlées peuvent conduire à une configuration inattendue des données.

L'*atomicité* est une propriété qui, intuitivement, stipule que (au moins d'un point de observationel) l'exécution d'un bloc de code peut être considérée comme séquentielle. En particulier, les méthodes d'analyse standard de la programmation séquentielle peuvent être utilisée pour raisonner sur de tels fragments de code. Comme noté dans [48], l'absence de data-race ne suffit pas à garantir l'atomicité pour laquelle des outils spécifiques doivent être proposés.

Une première approche consiste à fournir au programmeur des primitives de synchronisation lui permettant d'assurer lui même les invariants nécessaires à la cohérence des données ; les verrous et les variables de condition sont des exemples, probablement les plus répandus, de telles primitives. Dans cette approche, l'utilisateur a la charge d'associer un verrou à chaque donnée devant être protégée. Pour assurer l'atomicité d'un fragment de code, les verrous associés aux données manipulés par ce fragment sont pris au début de celui-ci et relachés à la fin. Si ce protocole n'est pas respecté par l'ensemble des composantes du système, l'atomicité est compromise. Il y a aujourd'hui une tendance à reconnaître que l'utilisation de ces techniques tend plutôt à rendre le développement encore plus complexe. Les verrous sont en effet des primitives de bas niveau dont l'utilisation peut nécessiter la résolution de problèmes de synchronisation complexes (e.g. deadlocks). De manière générale, il y a aujourd'hui un accord sur la nécéssité d'introduire des primitives, de plus haut niveau, dédiées au problème de l'atomicité ; d'autant plus que l'atomicité est généralement la propriété recherchée par les utilisateurs lors de l'utilisation de primitives de bas niveau. Les restrictions apportées à la notion de verrou afin de garantir la mise en place de protocoles de synchronisation corrects [47, 48, 46] vont d'ailleurs dans ce sens.

L'exemple le plus caractéristique de primitive dédiée à la question de l'atomicité est certainement la *transaction*. Les transactions ont d'abord été utilisées dans le cadre des bases de données et des systèmes distribuées pour regrouper plusieurs opérations selon le principe du tout ou rien.

Intuitivement, si l'atomicité d'une transaction est compromise, e.g. parce qu'une autre transaction démarre avant que celle-ci ne soit achevée, la transaction est annulée (ces effets ne doivent pas être visibles) et son exécution sera tentée à nouveau par la suite. Plus récemment, l'utilisation des transactions comme primitive de synchronisation pour les langages de programmation concurrente a été proposée[93, 58, 55]. Les transactions ont par exemple été implantées pour Objective Caml[89] et Standard ML[97]. L'utilisation des transactions dans le cadre d'un langage de programmation généraliste est encore jeune et il faudra un certain temps pour se faire une idée précise de la validité de concept. En effet, dans le cadre d'un langage de programmation concurrente, les interactions sont beaucoup plus complexes que dans une base de donnée. Par exemple, une transaction trop longue a peu de chances de ne pas être interrompue ; une solution consiste alors a forcer l'exécution de cette transaction au détriment des autres au risque de bloquer le système. La gestion des entrées/sorties est également problèmatique (une transaction peut être annulée en cours d'exécution). La mise en place d'une mémoire tampon peut résoudre une partie des problèmes, mais toutes les entrées/sorties ne supportent pas cette approche. La désactivation des interruptions réalisée dans les systèmes d'exploitations pour garantir l'atomicité de certaines opérations (très rapides) est une approche similaire.

Pour finir cette discussion sur la notion d'atomicité, revenons aux FAIR THREADS et au langage PACT dans lesquels l'ordonnancement des threads soumis à l'influence d'une même zone synchrone est coopératif. Lorsque le parallèlisme n'est pas considéré, ce type d'ordonnancement offre un moyen simple et efficace de réaliser une série d'opérations de manière atomique. Entre deux points de coopérations, un thread est assuré d'être le seul à s'exécuter. Comme nous l'avons déja vu, les FAIR THREADS et PACT mélangent les styles de programmation coopératif et préemptif et cette propriété atomicité n'est plus garantie par construction. Dans [26], on peut d'ailleurs noter la présence de verrous permettant de maintenir (à la main) les invariants voulus sur les données lorsque celles-ci peuvent être manipulées par des threads soumis à différentes zones synchrones. Dans ce chapitre, nous proposons une discipline de programmation pour PACT qui permet de conserver l'atomicité de l'exécution du code entre deux points de coopération. Celle-ci sera présentée dans la section 9.2 après que nous ayons donné une définition formelle de l'atomicité pour ce langage dans la section 9.1. Dans la section 9.3, nous présentons une analyse statique qui permet de garantir qu'un programme respecte cette discipline.

9.1 Atomicité en PACT

Dans cette section, nous définissons une propriété d'atomicité adaptée au langage PACT. Intuitivement, cette propriété stipule que, entre deux points de coopération, l'exécution d'un thread peut être considérée comme atomique. Cette propriété repose sur une notion de séparation de la mémoire relative au non déterminisme des accès et sur la notion de thread actif. Ces deux notions sont définies formellement ci-dessous.

Définition 20. *Soit $S = R, \mathcal{H}, \mathcal{E}$ un système et soit δ une fonction de $\mathcal{T}$ (les noms de threads) vers $\mathcal{P}(Dom(\mathcal{H}))$(l'ensemble des parties finies de $\mathcal{H}$). On dit que δ* séparе *S si $t \neq t'$ implique $\delta(t) \cap \delta(t') = \emptyset$ et si $S \xrightarrow[(t,r)]{} S'$ implique $r \in \delta(t)$.*

Un thread actif est simplement un thread qui dispose du contrôle pour une zone synchrone donnée. Dans la définition ci-dessous, on demande qu'un thread actif ne soit pas suspendu ce qui assure que deux phases d'exécution successives d'un même thread ne peuvent pas être confondues.

Définition 21. *Soit S un système et soit ℓ le nom d'une zone synchrone de ce système. On dit que le thread t est actif en ℓ pour S si $S = R, \mathcal{H}, \mathcal{E}$, $R \equiv \langle P^t \cdot T, T' \rangle_\ell \mid R'$ et $\neg(P, \mathcal{E}, \ell)\downarrow$.*

Intuitivement, un programme est atomique si, en tout point de l'exécution, il est possible d'associer à chaque thread un fragment de la mémoire tel que, entre deux points de coopération, ce thread n'accède qu'à ce fragment et lui seul y accède. Cette propriété est définie formellement ci-dessous.

Définition 22. *Un programme est atomique si pour toute exécution $S_0, S_1, \ldots$ de ce programme, il existe une suite de fonctions $\delta_0, \delta_1, \ldots$ telle que*

1. *δ_i sépare S_i pour tout $i \geq 0$,*
2. *pour tout $i \geq 0$ et pour tout ℓ de P, si t est actif en ℓ pour S_i et S_{i+1} alors $\delta_i(t) = \delta_{i+1}(t) \cap Dom(\mathcal{H}_i)$.*

La notion de data-race, i.e. d'accès conflictuel, peut être caractérisée par un choix non déterministe de la forme :

$$S \xrightarrow[(t,r)]{} S' \text{ et } S \xrightarrow[(t',r)]{} S'' \text{ où } t \neq t'$$

Il est immédiat que, au cours de l'exécution d'un programme atomique, une telle configuration est impossible du fait de la première condition de la définition précédente. En particulier, la propriété d'atomicité proposée ici est bien une extension de l'absence de data-race dans laquelle la notion de conflit n'est pas *physique* mais *logique* au sens ou celle-ci est associée à une portion de code de longueur arbitraire.

Remarque 13. *La notion d'atomicité proposée ici ne distingue pas le type d'accès réalisée (création, lecture ou écriture d'une référence) L'ensemble des développements à venir peuvent sans difficulter être adaptés à une telle distinction.*

Les exemples 15 et 16, ci-dessous, illustrent la différence entre absence de data-race et atomicité.

Exemple 15. *Considérons le programme* $\texttt{run}\ A(r, s); \texttt{run}\ B(r)$ *où r (resp. s) est une référence (resp. un signal) et A et B sont définis par les équations suivantes :*

$$\begin{array}{lcl} A(x,y) & = & \texttt{link}\ \ell_1\ (\texttt{set}\ x\ \texttt{z}; (u = \texttt{get}\ x) \cdot \texttt{emit}\ y\ u) \\ B(x) & = & \texttt{link}\ \ell_2(\texttt{set}\ x\ \texttt{s(z)}) \end{array}$$

Dans cet exemple, les deux threads accèdent de manière concurrente à la référence r, il y a un choix non déterministe ; la valeur (un entier naturel) émise sur le signal s est soit z *soit* s(z).

Exemple 16. *Considérons le programme* $\texttt{run}\ A(r_1, r_2, r_2, s); \texttt{run}\ B(r_1, r_2, r_3)$.

$$\begin{array}{lcl} A(x,y,z,t) & = & \texttt{link}\ \ell_1\ (\texttt{set}\ x\ \texttt{z}; \texttt{set}\ y\ True; C(z); (u = \texttt{get}\ x) \cdot \texttt{emit}\ t\ u) \\ B(x,y,z) & = & \texttt{link}\ \ell_2(C(y); \texttt{set}\ x\ \texttt{s(z)}; \texttt{set}\ z\ True) \\ C(x) & = & (y = \texttt{get}\ x) \cdot \texttt{if}\ y\ \texttt{then}\ 0\ \texttt{else}\ C(x) \end{array}$$

Cet exemple est similaire à l'exemple précédent, à une différence près. Ici, l'équation C est utilisée par les deux threads pour réaliser des attentes actives *afin de synchroniser leurs accès. En particulier, aucun accès conflictuel n'est possible du fait de cette synchronisation. En revanche, ce programme n'est pas atomique.*

Remarque 14. *A noter que le codage de l'attente active de l'exemple précédent ne serait pas possible avec un contrôle de terminaison des instants similaire à celui de la partie I. Il serait intéressant de voir si, dans le cadre d'un tel contrôle, l'absence de data-race et l'atomicité sont des propriétés équivalentes.*

On termine cette section par un exemple de programme atomique.

Exemple 17. *Considérons le programme* $\texttt{run}\ A(r); \texttt{run}\ A(r)$ *où r est une référence et où A et B sont définis par les équations suivantes :*

$$A(x) \quad = \quad (\texttt{link}\ \ell_1\ (y = \texttt{get}\ x) \cdot \texttt{set}\ x\ \texttt{s}(y); \texttt{pause}); A(x)$$

Dans cet exemple, chaque thread incrémente la référence r de manière atomique.

9.2 Propriété et droit d'accés

Dans cette section, on propose une discipline de programmation, simple et intuitive, qui permet d'assurer qu'un programme est atomique. Cette discipline repose principalement sur la définition d'un *droit d'accès* basé sur l'attribution d'un *propriétaire*, un thread ou une zone synchrone, à chaque référence.

Selon ce principe, on commence par introduire un certain nombre de notations dans la syntaxe et la sémantique du langage. Une référence est maintenant une paire $(r, o) \in Ref \times (\mathcal{T} \cup \mathcal{L})$ où r est comme précédemment le nom de la référence et $o \in \mathcal{T} \cup \mathcal{L}$ est son propriétaire. Par la suite on notera plus simplement r_o au lieu de (r, o). De même, dans un système $R, \mathcal{H}, \mathcal{E}$, la fonction $\mathcal{H}$ est maintenant une fonction partielle de $Ref \times (\mathcal{L} \cup \mathcal{T})$ vers $\mathcal{V}$. On suppose que l'utilisateur annote les créations de références de la manière suivante :

$$(x = \texttt{ref}_\alpha\ e) \cdot P$$

où $\alpha \in \mathcal{L} \cup \{self\}$ et où $self$ est une constante dénotant le thread courant (celui qui crée la référence). Cette annotation est prise en compte dans la sémantique en divisant la règles (ref) en deux règles (ref_1) et (ref_2) pour plus de lisibilité, les autres règles de la sémantique sont inchangées, aux annotations près. :

$$\frac{r_\ell \notin Dom(\mathcal{H}) \qquad e \Downarrow v \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} (x = \texttt{ref}_\ell\ e) \cdot P, \mathcal{H}, \mathcal{E} \underset{t,r_\ell}{\rightarrow} [r_\ell/x]P, \mathcal{H} \cup \{r_\ell \mapsto v\}, \mathcal{E}} \quad (ref_1)$$

$$\frac{r_t \notin Dom(\mathcal{H}) \qquad e \Downarrow v \qquad \alpha \in \mathcal{L} \cup \{free\}}{\vdash_{t,\alpha} (x = \texttt{ref}_{self}) \cdot P, \mathcal{H}, \mathcal{E} \underset{(t,r_t)}{\longrightarrow} [r_t/x]P, \mathcal{H} \cup \{r_t \mapsto v\}, \mathcal{E}} \quad (ref_2)$$

On peut maintenant définir la notion de droit d'accès. Intuitivement, dans un système S, un thread possède le droit d'accès à une référence s'il en est le propriétaire ou s'il est actif pour la zone qui en est le propriétaire. Cela est formalisé par la définition donnée ci-dessous.

Définition 23. *Soit S un système et t un nom de thread. On dit que t possède le droit d'accès pour o dans S, pour* $o \in \mathcal{T} \cup \mathcal{L}$*, ce que l'on note* $S \rhd t : o$*, si et seulement si* $o = t$ *ou* $o = \ell$ *et t est actif en* ℓ*. On dit que le programme P est compatible avec les droits d'accès si pour toute exécution* $S_0, S_1 \ldots$ *de P on a*

$$\forall i \geq 0. (S_i \underset{(t,r_o)}{\longrightarrow} S_{i+1}) \Rightarrow (S_i \rhd t : o)$$

Cette notion de droit d'accès introduit un style de programmation qui est résumé dans la figure 9.1. Intuitivement, chaque thread dispose de sa propre mémoire locale accessible par lui seul quel que soit sont état de synchronisation. De la même manière, chaque zone synchrone est associée à une portion de la mémoire uniquement accessible par les threads soumis à son influence et dont l'exécution est

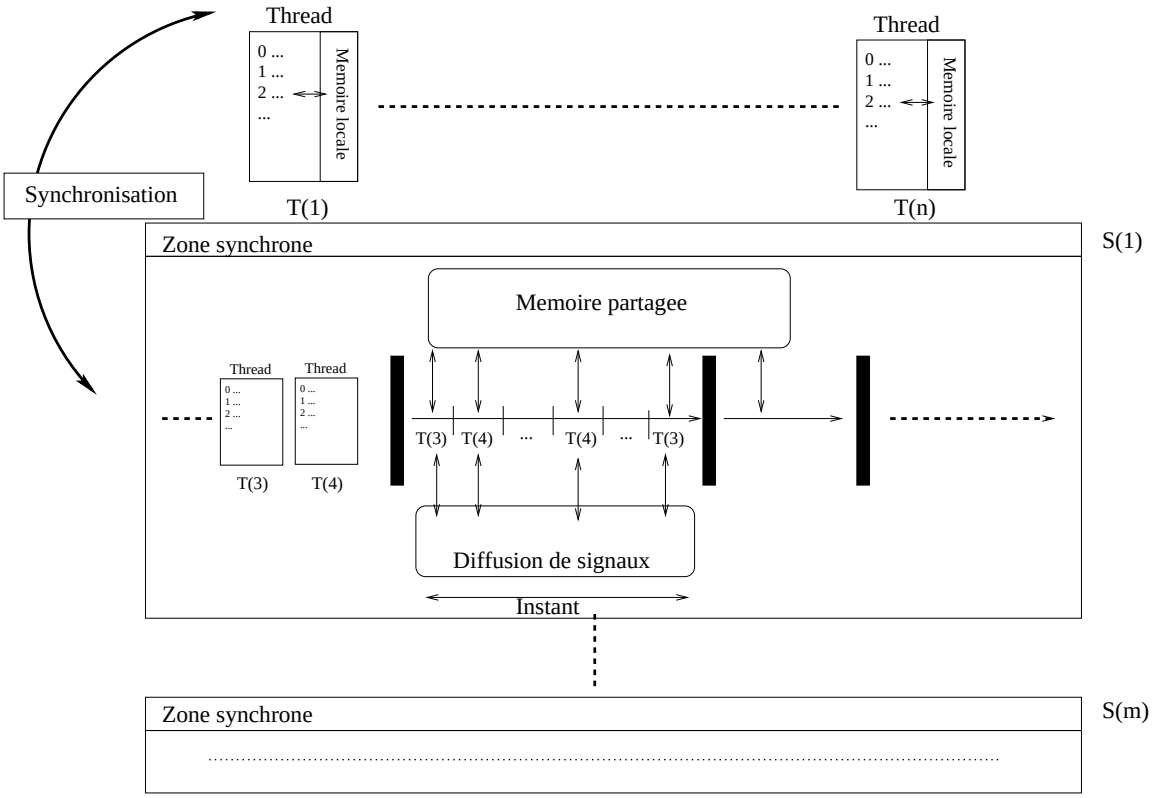


FIG. 9.1 – Droit d'accès

ainsi séquentialisée. Intuitivement, à tout moment, le droit d'accès à une référence est détenu par au plus un thread et les droits d'accès détenus par un thread entre deux points de coopération sont constants. Il est facile de vérifier que cette discipline de programmation assure qu'un programme est atomique.

Proposition 4. *Un programme compatible avec les droits d'accès est atomique.*

A noter que, pour communiquer, des threads doivent nécessairement être soumis à une même zone synchrone. Intuitivement, cette contrainte revient à imposer, dans le cadre d'un système purement préemptif, une synchronisation par verrou pour chaque accès à une référence partagée comme cela est fait dans [44]. [1] Une analogie intéressante existe entre le modèle de programmation considéré ici et la notion de processus. Dans les systèmes UNIX, par exemple, un processus dispose d'un espace mémoire protégé. Au sein de chaque processus plusieurs threads préemptifs partagent l'espace mémoire. Par analogie et grâce à la propriété d'atomicité, chaque zone synchrone peut être vue comme un processus au sein duquel des threads coopératifs partagent l'accès à une mémoire commune. Nous avons préféré ne pas trop mettre cette analogie en avant dans le travail présenté ici du fait que la notion de processus fait appel à une notion physique de protection de la mémoire. Plus précisément, un processus ne peut avoir connaissance de l'espace mémoire d'un autre processus. Dans le langage considéré ici, la possibilité pour les threads de passer d'une zone synchrone à l'autre ne correspond

[1] On peut d'ailleurs noter à ce propos que, dans l'implémentation des FAIR THREADS, un *scheduler* est lui même défi ni par un verrou.

pas tout à fait à ce cas de figure. Il nous semble cependant que l'analogie mérite d'être soulignée en tant que méthodologie de structuration des programmes. D'une certaine façon, le langage permet de combiner des modes de programmation en mémoire partagée et par "passage de messages" (envoi d'un thread d'une zone synchrone à l'autre). Une zone synchrone peut alors être vue comme le lieu d'exécution de composants concurrents nécessitant un haut niveau de communication et de synchronisation.

9.3 Analyse Statique

Dans la section précédente, nous avons défini une propriété d'atomicité et une condition suffisante sur les programmes qui garantit que cette propriété est vérifiée. Dans cette section, on introduit un système de type tel que tout programme bien typé vérifie cette condition. Ce système de type repose sur les notions de propriété et de droit d'accès définis précédemment. En particulier, le propriétaire d'une référence est partiellement défini par une annotation qui raffine le type de celle-ci. Le type d'une référence est maintenant de la forme :

$$\tau\ ref_{\alpha} \text{ où } \alpha \in \mathcal{L} \cup \{self\}$$

Intuitivement, une référence de type $\tau\ ref_{\ell}$ dénote une référence associée à la zone synchrone ℓ alors qu'une référence de type $\tau\ ref_{self}$ dénote une référence associé à un thread quelconque. Ces informations (partielles) sur les propriétaires des threads sont utilisées par le système de type pour garantir qu'aucun thread n'accède à une référence sans posséder le droit d'accès associé. Cette vérification repose sur deux points, selon que le propriétaire est une zone synchrone ou un thread.

1. Si le propriétaire d'une référence est une zone synchrone, alors un thread ne peut y accéder qu'en étant soumis à l'influence de cette zone. L'information de type associée à cette référence est utilisée pour forcer la synchronisation correspondante. On associe à un thread une *contrainte de synchronisation*, notée E, telle que :
   - E est `none` si l'exécution du thread peut se dérouler indépendamment de son état de synchronisation
   - E est `access`(ℓ) si l'exécution du thread ne peut se dérouler que sous l'influence de la zone synchrone ℓ.

   Pour simplifier les notations, on se donne une relation d'ordre partielle sur les contraintes de synchronisation, définie comme étant la plus petite relation vérifiant `none` $<$ `access`(ℓ) pour tout ℓ. Les contraintes de synchronisation utilisées sont similaires aux effets utilisées dans les systèmes de types et d'effets[64, 94].
2. Lorsqu'une référence est crée par un thread, le type de celle-ci ne permet pas de déterminer le thread en question. Afin d'assurer qu'une telle référence ne peut être manipulée que par le thread l'ayant créée, on interdit simplement à ce dernier de la "communiquer" à un autre thread. En particulier, les threads ne peuvent communiquer que par signaux ou en utilisant des références associées à des zones synchrones. Pour restreindre les communications, on introduit un prédicat sur les types, le prédicat *Public*, défini de la manière suivante :

   $$\frac{}{Public(1)} \qquad \frac{Public(\tau)}{Public(\tau\ ref_{\ell})} \qquad \frac{Public(\tau)}{Public(\tau\ sig)} \qquad \frac{Public(\tau)}{Public(\tau\ list)}$$

   Il est facile de vérifier qu'une valeur de type τ, telle que $Public(\tau)$, ne "contient" pas de référence dont le propriétaire soit un thread.

Exemple 18. *Dans l'exemple suivant un thread défini par l'équation de A reçoit en paramètre une référence x et met à jour le contenu de celle-ci. Dans le programme deux copies de A sont lancées en parallèle et reçoivent en paramètre la même référence définie comme* privée.

$$A(x) = (y = \texttt{get}\ x) \cdot \texttt{set}\ x\ \texttt{cons}(*, y)$$

$$(x = \texttt{ref}_{self}\ \texttt{nil}) \cdot (\texttt{thread}\ A(x); A(x)$$

Le système de type que nous introduisons par la suite rejettera un tel programme en détectant qu'une référence privée du thread initial est communiquée au nouveau thread. Pour qu'un tel partage soit possible, le propriétaire de la référence devrait être une zone synchrone à l'influence de laquelle les deux threads seraient soumis lors de ces accès.

Avant d'introduire le système de type, nous définissons plusieurs notations. Un *environnement de typage*, noté Γ, est une fonction partielle de Var vers $\mathcal{T}$. On note $x : \tau$ l'environnement de typage Γ tel que $\Gamma(x) = \tau$ et tel que Γ soit non défini partout ailleurs. On note $Dom(\Gamma)$ le domaine de définition de Γ. On note $\Gamma,\ \Gamma'$ la fonction définie par l'union point à point de Γ et Γ' où il est entendu que $Dom(\Gamma) \cap Dom(\Gamma') = \emptyset$. Le système de type est construit sur deux formes de jugements :

- des jugements de la forme $\Gamma \vdash e : \tau$, lire "le type de e sous l'environnement Γ est τ". Ce type de jugement permet de typer les expressions fonctionnelles du sous langage.
- des jugements de la forme $\Gamma \triangleright P : E$, lire "la contrainte de synchronisation de P sous l'environnement Γ est E". Ce type de jugement permet de contrôler la conformité du comportement d'un thread à notre politique de droits d'accès.

Typage des expressions. Le jugement $\Gamma \vdash e : \tau$ est défini par :

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \mathsf{nil} : \tau\ list} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ list}{\Gamma \vdash \mathsf{cons}(e_1, e_2) : \tau\ list} \qquad \frac{f : \overline{\tau} \to \tau \qquad \Gamma \vdash \mathbf{e} : \overline{\tau}}{\Gamma \vdash f(\mathbf{e}) : \tau}$$

où, si $\mathbf{e} = e_1, \ldots, e_n$ et $\overline{\tau} = \tau_1 \times \ldots \times \tau_n$, la notation $\Gamma \vdash \mathbf{e} : \overline{\tau}$ représente la liste de jugements $\Gamma \vdash e_1 : \tau_1, \ldots, \Gamma \vdash e_n : \tau_n$.

Typage des programmes On utilisera la notation $\Gamma, p : \tau$ pour l'environnement de typage $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n$ où $x_1, \ldots, x_n$ sont les variables de p et les types $\tau_1, \ldots, \tau_n$ sont tels que $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash p : \tau$. On rappelle qu'un programme est constitué d'un ensemble de noms de zone synchrone, d'un thread et d'un ensemble d'équations de la forme $A(\mathbf{x}) = P$. Pour chaque équation $A(\mathbf{x}) = P$, on suppose donnés $\overline{\tau}$ le type des paramètres de A et E la contrainte de synchronisation associée au membre droit de l'équation ; on note alors $A : \overline{\tau}/E$. Les règles de dérivation des jugements de la forme $\Gamma \triangleright P : E$ sont données ci-dessous et il est entendu que les noms de zones synchrones utilisés sont ceux associés au programme.
Le thread terminé ne nécessite pas de synchronisation particulière.

$$\frac{}{\Gamma \triangleright 0 : \mathsf{none}} \quad (TERM)$$

La contrainte de synchronisation associée à un appel est simplement celle donnée par le type de l'identificateur.

$$\frac{A : \overline{\tau}/E \qquad \Gamma \vdash \mathbf{e} : \overline{\tau}}{\Gamma \triangleright A(\mathbf{e}) : E} \quad (CALL)$$

La contrainte de synchronisation d'un filtrage doit être compatible avec (plus grande que) celles de ses deux branches.

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma,\ p : \tau \rhd P_1 : E_1 \qquad \Gamma \rhd P_2 : E_2 \qquad E_1, E_2 \leq E_3}{\Gamma \rhd [e \trianglerighteq p]\ P_1, P_2 : E_3} \quad (MATCH)$$

La création d'une référence associée à un thread ne nécessite pas de synchronisation particulière.

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma,\ x : \tau\ ref_{self} \vdash P : E}{\Gamma \rhd (x = \texttt{ref}_{self}\ e) \cdot P : E} \quad (REF_1)$$

La création d'une référence associée à une zone synchrone nécessite une synchronisation avec celle-ci. Le code sous la portée de la définition de cette nouvelle référence doit donc être compatible avec cette contrainte de synchronisation. De plus, une telle référence étant partagée, les valeurs portées par celles-ci ne doivent pas permettre l'échange de références privées (utilisation du prédicat *Public*).

$$\frac{\Gamma \vdash e : \tau \qquad Public(\tau) \qquad \Gamma,\ x : \tau\ ref_\ell \rhd P : E \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{ref}_\ell\ e) \cdot P : \texttt{access}(\ell)} \quad (REF_2)$$

La manipulation d'une référence associée à un thread ne nécessite pas de synchronisation particulière (règles (SET_1) et (GET_1)). En revanche, la manipulation d'une référence associée à une zone synchrone impose une synchronisation avec cette zone (règles (SET_2) et (GET_2)).

$$\frac{\Gamma \vdash e_1 : \tau\ ref_{self} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \rhd \texttt{set}\ e_1\ e_2 : \texttt{none}} \quad (SET_1)$$

$$\frac{\Gamma \vdash e : \tau\ ref_{self} \qquad \Gamma,\ x : \tau \rhd P : E}{\Gamma \rhd (x = \texttt{get}\ e) \cdot P : E} \quad (GET_1)$$

$$\frac{\Gamma \vdash e_1 : \tau\ ref_\ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \rhd \texttt{set}\ e_1\ e_2 : \texttt{access}(\ell)} \quad (SET_2)$$

$$\frac{\Gamma \vdash e : \tau\ ref_\ell \qquad \Gamma,\ x : \tau \rhd P : E \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{get}\ e) \cdot P : \texttt{access}(\ell)} \quad (GET_2)$$

La contrainte de synchronisation d'une séquence doit être compatible avec celles de ses deux membres.

$$\frac{\Gamma \rhd P_1 : E_1 \qquad \Gamma \rhd P_2 : E_2 \qquad E_1, E_2 \leq E_3}{\Gamma \rhd P_1; P_2 : E_3} \quad (SEQ)$$

La création d'un nouveau signal nécessite une synchronisation avec une zone quelconque. Comme dans le cas d'une référence associée à une zone synchrone, le prédicat *Public* doit être vérifié pour le type des valeurs portées par le signal.

$$\frac{\Gamma,\ x : \tau\ sig \rhd P : E \qquad Public(\tau) \qquad \ell \in \mathcal{L} \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{sig}) \cdot P : \texttt{access}(\ell)} \quad (SIG)$$

L'émission d'un signal nécessite une synchronisation avec une zone synchrone quelconque.

$$\frac{\Gamma \vdash e_1 : \tau\ sig \qquad \Gamma \vdash e_2 : \tau \qquad \ell \in \mathcal{L}}{\Gamma \rhd \texttt{emit}\ e_1\ e_2 : \texttt{access}(\ell)} \quad (EMIT)$$

Même principe pour les cas de la lecture d'un signal dans l'instant et de la lecture des valeurs émises à l'instant précédent.

$$\frac{\Gamma \vdash e : \tau\ sig \qquad \Gamma,\ x : \tau \rhd P : E \qquad \ell \in \mathcal{L} \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{await}\ e) \cdot P : \texttt{access}(\ell)} \quad (AWAIT)$$

$$\frac{\Gamma \vdash e : \tau\ sig \qquad \Gamma,\ x : \tau\ list \rhd P : E \qquad \ell \in \mathcal{L} \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{pre}\ e) \cdot P : \texttt{access}(\ell)} \quad (PRE)$$

L'exécution d'un bloc de préemption doit être synchronisée avec une zone synchrone, compatible avec l'exécution de ses deux branches.

$$\frac{\Gamma \vdash e : \tau\ sig \qquad \Gamma \rhd P_1 : E_1 \qquad \Gamma \rhd P_2 : E_2 \qquad \ell \in \mathcal{L} \qquad E_1, E_2 \leq \texttt{access}(\ell)}{\Gamma \rhd \texttt{watch}\ e\ P_1, P_2 : \texttt{access}(\ell)} \quad (WATCH)$$

La construction `link` lève (si elle existe) une contrainte de synchronisation associée à une zone synchrone donnée.

$$\frac{\Gamma \rhd P : E \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd \texttt{link}\ \ell\ P : \texttt{none}} \quad (LINK)$$

Le même principe s'applique pour la construction `unlink` pour laquelle la portion de code concernée ne doit nécessiter aucune synchronisation particulière.

$$\frac{\Gamma \rhd P : \texttt{none}}{\Gamma \rhd \texttt{unlink}\ P : \texttt{none}} \quad (UNLINK)$$

Finalement, pour la création d'un nouveau thread, aucune synchronisation ne doit être requise (un nouveau thread est, sauf demande explicite de sa part, *libre* par défaut).

$$\frac{A : \overline{\tau}/\texttt{none} \qquad Public(\overline{\tau}) \qquad \Gamma \vdash \mathbf{e} : \overline{\tau}}{\Gamma \rhd \texttt{run}\ A(\mathbf{e}) : \texttt{none}} \quad (THREAD)$$

L'ensemble des règles de typage est résumé dans la figure 9.2. Pour conclure la présentation de ce système de type, nous revenons sur certains exemples de programmes présentés plus tôt.

Exemple. *Considérons d'abord l'exemple 16. Dans cet exemple, les trois références considérées sont passées comme paramètres à des threads et doivent donc vérifier le prédicat Public à cause de la règle* (*THREAD*)*. Or ces trois références sont utilisées sur les deux zones synchrones ℓ_1 et ℓ_2 ce qui est impossible car les règles de typages associées à la manipulation de ce type de références stipulent que celles-ci devraient être associées aux deux zones. Il n'y a aucun moyen de typer ce programme quelque soient les propriétaires des références. Considérons maintenant l'exemple 18. Dans cet exemple, le thread crée une référence privée qu'il passe en paramètre à un nouveau thread. L'utilisation de la règle* (*THREAD*) *rend impossible le typage de ce programme.*

On peut maintenant introduire la notion de programme bien typé et notre résultat principal qui est donné par le théorème 2.

Définition 24. *Un programme est dit bien typé si,*

- $\rhd P : \texttt{none}$*, où P est le thread initial et où l'environnement de typage vide est simplement omis :*
- *si pour toute équation $A(\mathbf{x}) = P$ avec $A : \overline{\tau}/E$ on a $\mathbf{x} : \overline{\tau} \rhd P : E$.*

Théorème 2. *Un programme bien typé vérifie la propriété d'atomicité.*

Pour conclure cette section, on peut noter l'existence d'autres approches dont on aurait pu s'inspirer pour assurer la séparation mémoire des composants d'un système. La logique de séparation *separation logic*[88] permet de raisonner de manière indépendante sur des parties distinctes d'un programme ; en particulier, cette approche peut être utilisée dans le cadre de la concurrence[87]. L'approche suivie ici est différente, la logique de séparation "cloisonne" très fortement l'espace mémoire des composants ; dans notre approche, l'espace mémoire de chaque composant peut être modifié dynamiquement. On peut également noter la possibilité, dans les systèmes de types et d'effets, de définir localement une région. Par exemple, lors de la création d'un thread, une déclaration de la forme "*local* ρ *in* `run` $A(\mathbf{e})$". pourrait forcer les références de la région ρ à être locales au nouveau thread. En terme de modularité, cette approche nous semble limitée. Par exemple, dans notre approche, l'appel de "fonctions externes" peut également générer des références locales au thread.

9.4 Correction

Dans cette section, on introduit un certain nombre de définitions et de résultats auxiliaires nécessaires à la preuve du théorème de la section précédente.présente les détails techniques On commence par étendre la notion de contexte de typage afin de typer les expressions non statiques. Le domaine de définition d'un environnement de typage Γ est maintenant $\mathcal{V} \cup Ref \cup Sig$ et le système de type des expressions introduit dans la section précédente est étendu par les règles suivantes :

$$\frac{r \in Ref,\ \ell \in \mathcal{L}}{\Gamma, r_\ell : \tau \vdash r_\ell : \tau\ ref_\ell} \qquad \frac{r \in Ref,\ t \in \mathcal{T}}{\Gamma, r_t : \tau \vdash r_t : \tau\ ref_{self}} \qquad \frac{s \in Sig}{\Gamma, s : \tau \vdash s : \tau\ sig}$$

On procède ensuite en deux étapes correspondants aux deux types de contraintes du système présenté dans la section précédente (contraintes de synchronisation et prédicat *Public*).

Premièrement, on prouve un lemme de *subject reduction* reposant sur l'extension des jugements de typage aux systèmes. En particulier, cette extension permet d'assurer la cohérence entre le typage des threads et celui des valeurs associées aux références et aux signaux ; en particulier en ce qui concerne le prédicat *Public*.

Définition 25. *Etant donné un environnement de typage* Γ *et un système* $R, \mathcal{H}, \mathcal{E}$*, on note* $\Gamma \triangleright R, \mathcal{H}, \mathcal{E}$*, si les conditions suivantes sont vérifiées :*

1. *pour tout thread* P^t *de* R*, on a* $\Gamma \triangleright P : \mathsf{none}$.
2. $Dom(\Gamma) = Dom(\mathcal{H}) \cup Dom(\mathcal{E})$.
3. *pour tout* $r_o \in Dom(\mathcal{H})$ *si* $\Gamma = \Gamma', r_o : \tau$ *alors*
   - $\Gamma \vdash \mathcal{H}(r_o) : \tau$,
   - $o \in \mathcal{L} \Rightarrow Public(\tau)$
4. *pour tout* $s \in Dom(\mathcal{E})$ *si* $(\Gamma = \Gamma', s : \tau)$ *alors*
   - *pour tout* ℓ *et pour tout* $v \in \mathcal{E}_c(s, \ell)$ *on a* $\Gamma \vdash v : \tau$,
   - *pour tout* ℓ *on a* $\Gamma \vdash \mathcal{E}_p(s, \ell) : \tau\ list$,
   - $Public(\tau)$ *est vérifié*

Lemme 2. *Si* $\Gamma \triangleright R, \mathcal{H}, \mathcal{E}$ *et* $R, \mathcal{H}, \mathcal{E} \underset{M}{\rightarrow} R', \mathcal{H}', \mathcal{E}'$ *alors il existe* Γ' *tel que* $\Gamma \subseteq \Gamma'$ *et* $\Gamma' \triangleright R', \mathcal{H}', \mathcal{E}'$.

$$\frac{}{\Gamma \rhd 0 : \texttt{none}}\ (TERM) \qquad \frac{A : \overline{\tau}/E \qquad \Gamma \vdash \mathbf{e} : \overline{\tau}}{\Gamma \rhd A(\mathbf{e}) : E}\ (CALL)$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma,\ p : \tau \rhd P_1 : E_1 \qquad \Gamma \rhd P_2 : E_2 \qquad E_1, E_2 \leq E_3}{\Gamma \rhd [e \trianglerighteq p]\ P_1, P_2 : E_3}\ (MATCH)$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma,\ x : \tau\ ref_{self} \vdash P : E}{\Gamma \rhd (x = \texttt{ref}_{self}\ e) \cdot P : E}\ (REF_1)$$

$$\frac{\Gamma \vdash e : \tau \qquad Public(\tau) \qquad \Gamma,\ x : \tau\ ref_{\ell} \rhd P : E \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{ref}_{\ell}\ e) \cdot P : \texttt{access}(\ell)}\ (REF_2)$$

$$\frac{\Gamma \vdash e_1 : \tau\ ref_{self} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \rhd \texttt{set}\ e_1\ e_2 : \texttt{none}}\ (SET_1) \qquad \frac{\Gamma \vdash e : \tau\ ref_{self} \qquad \Gamma,\ x : \tau \rhd P : E}{\Gamma \rhd (x = \texttt{get}\ e) \cdot P : E}\ (GET_1)$$

$$\frac{\Gamma \vdash e_1 : \tau\ ref_{\ell} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \rhd \texttt{set}\ e_1\ e_2 : \texttt{access}(\ell)}\ (SET_2) \qquad \frac{\Gamma \vdash e : \tau\ ref_{\ell} \qquad \Gamma,\ x : \tau \rhd P : E \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{get}\ e) \cdot P : \texttt{access}(\ell)}\ (GET_2)$$

$$\frac{\Gamma \rhd P_1 : E_1 \qquad \Gamma \rhd P_2 : E_2 \qquad E_1, E_2 \leq E_3}{\Gamma \rhd P_1; P_2 : E_3}\ (SEQ)$$

$$\frac{\Gamma,\ x : \tau\ sig \rhd P : E \qquad Public(\tau) \qquad \ell \in \mathcal{L} \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{sig}) \cdot P : \texttt{access}(\ell)}\ (SIG)$$

$$\frac{\Gamma \vdash e_1 : \tau\ sig \qquad \Gamma \vdash e_2 : \tau \qquad \ell \in \mathcal{L}}{\Gamma \rhd \texttt{emit}\ e_1\ e_2 : \texttt{access}(\ell)}\ (EMIT)$$

$$\frac{\Gamma \vdash e : \tau\ sig \qquad \Gamma,\ x : \tau \rhd P : E \qquad \ell \in \mathcal{L} \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{await}\ e) \cdot P : \texttt{access}(\ell)}\ (AWAIT)$$

$$\frac{\Gamma \vdash e : \tau\ sig \qquad \Gamma,\ x : \tau\ list \rhd P : E \qquad \ell \in \mathcal{L} \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd (x = \texttt{pre}\ e) \cdot P : \texttt{access}(\ell)}\ (PRE)$$

$$\frac{\Gamma \vdash e : \tau\ sig \qquad \Gamma \rhd P_1 : E_1 \qquad \Gamma \rhd P_2 : E_2 \qquad \ell \in \mathcal{L} \qquad E_1, E_2 \leq \texttt{access}(\ell)}{\Gamma \rhd \texttt{watch}\ e\ P_1, P_2 : \texttt{access}(\ell)}\ (WATCH)$$

$$\frac{\Gamma \rhd P : E \qquad E \leq \texttt{access}(\ell)}{\Gamma \rhd \texttt{link}\ \ell\ P : \texttt{none}}\ (LINK) \qquad \frac{\Gamma \rhd P : \texttt{none}}{\Gamma \rhd \texttt{unlink}\ P : \texttt{none}}\ (UNLINK)$$

$$\frac{A : \overline{\tau}/\texttt{none} \qquad Public(\overline{\tau}) \qquad \Gamma \vdash \mathbf{e} : \overline{\tau}}{\Gamma \rhd \texttt{run}\ A(\mathbf{e}) : \texttt{none}}\ (THREAD)$$

FIG. 9.2 – Règles de typage

Deuxièmement, en se basant sur le premier lemme, on prouve que l'ensemble des références, dont le type est de la forme $\tau\ ref_{self}$, accessibles par un thread lors de son exécution sont des références crées par ce thread. En particulier, ce thread est le seul à accéder à ces références. Intuitivement, pour cela, on introduit la notion de réseau bien formé qui capture une propriété du graphe des références accessibles par un thread. Pour $o \in \mathcal{T} \cup \mathcal{L}$, on note Ref_o les références de la forme r_o.

Définition 26. *Un réseau $R, \mathcal{H}, \mathcal{E}$ est* bien formé *si pour tout P^t de R on a*

1. *$acc(P) \subseteq Ref_t$,*
2. *Pour tout $r_t \in Dom(\mathcal{H})$, $acc(\mathcal{H}(r_t)) \subseteq Ref_t$*

où $acc(P)$ (resp. $acc(e)$) est l'ensemble des références apparaissant dans les expressions de P (resp. dans e).

Lemme 3. *Soient $R, \mathcal{H}, \mathcal{E}$ un système bien formé et Γ un environnement tel que $\Gamma \triangleright R, \mathcal{H}, \mathcal{E}$. Si $R, \mathcal{H}, \mathcal{E} \rightarrow R', \mathcal{H}', \mathcal{E}'$ alors $R', \mathcal{H}', \mathcal{E}'$ est bien formé.*

Dans le chapitre suivant nous prouverons les résultats auxiliaires introduits dans cette section. Nous verrons également que le théorème de la section précédente se déduit facilement de ces résultats.

Chapitre 10

Preuves

Avant de présenter les preuves du chapitre précédent, on introduit trois résultats auxiliaires. La preuve du lemme 4 (resp 5) est une simple induction sur le programme (resp. le contexte).

Lemme 4. *Si* $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \triangleright P : E$ *et si* $\Gamma \vdash e_i : \tau_i$ *pour* $1 \leq i \leq n$ *alors* $\Gamma \triangleright P[e_1/x_1, \ldots, e_n/x_n] : E$.

Lemme 5. *Si* $\Gamma \triangleright E[P] : E_1$ *alors il existe* E_0 *tel que* $\Gamma \triangleright P : E_0$. *De plus, si* $\Gamma \triangleright P' : E'_0$ *et* $E'_0 \leq E_0$ *alors il existe* E'_1 *tel que* $E'_1 \leq E_1$ *et* $\Gamma \triangleright C[P] : E'_1$.

Lemme 6. *Si* $\Gamma \triangleright E[P] : \texttt{none}$ *et* $E \odot\!\!\rightarrow \alpha$ *alors soit* $\Gamma \triangleright P : \texttt{none}$ *soit* $\alpha = \ell$ *et* $\Gamma \triangleright P : \texttt{access}(\ell)$.

La preuve du lemme 6 est par induction sur la taille du contexte E. Il est immédiate que si $\Gamma \triangleright E[P] : E_0$ alors il existe E_1 tel que $\Gamma \triangleright P : E_1$. Supposons, que $E \odot\!\!\rightarrow \ell$ pour ℓ donné. Par définition, $E = E'[\texttt{link}\ \ell\ E'']$ où la construction de E'' ne repose ni sur la construction `link` ni sur la construction `unlink`. On a donc $\Gamma \triangleright \texttt{link}\ \ell\ E''[P] : \texttt{none}$ et, par la règle (*LINK*), $\Gamma \triangleright E''[P] : E_0$ avec $E_0 \leq \texttt{access}(\ell)$. Par construction de E'', il est immédiat que si $\Gamma \triangleright P : E_1$ alors $E_1 \leq E_0$. Si $E \odot\!\!\rightarrow \mathit{free}$, le raisonnement est similaire.

10.1 Preuve de la proposition 4

Soit $S_0 = (R_0, \mathcal{H}_0, \mathcal{E}_0), S_1 = (R_1, \mathcal{H}_1, \mathcal{E}_1), \ldots$ une exécution de P. On définit la suite de fonctions $\delta_0, \delta_1, \ldots$ par $\delta_i(t) = \{(r, o) \in \mathit{Dom}(\mathcal{H}_i) \mid S_i \triangleright t : o\}$. On vérifie d'abord que pour tout $i \geq 0$, δ_i est une fonction de séparation de $R_i, \mathcal{H}_i, \mathcal{E}_i$. Pour cela, il suffit de vérifier que $(t \neq t') \Rightarrow (S \triangleright t : \alpha \Rightarrow \neg(S \triangleright t' : \alpha))$; ce qui est immédiat. Pour tout $i \geq 0$, on a donc $\delta_i(t) \cap \delta_i(t') = \emptyset$ dès que $t \neq t'$. De plus, si $S_i \underset{t,r_o}{\rightarrow} S_{i+1}$, on a $S_i \triangleright t : o$ par hypothèse et donc $r_o \in \delta_i(t)$ par définition de δ_i. Il reste finalement à vérifier que pour tout $i \geq 0$, $\delta_i(t) = \delta_{i+1}(t) \cap \mathit{Dom}(\mathcal{H}_i)$ dès que t est actif en ℓ pour S_i et S_{i+1}. On note simplement que si $S \underset{(t,r_o)}{\longrightarrow} S'$ et si t est actif en ℓ pour S et S' alors $\forall \alpha.(S \triangleright t : \alpha \Leftrightarrow S' \triangleright t : \alpha)$.

10.2 Preuve du lemme 2

Démonstration. Supposons que $\Gamma \triangleright R, \mathcal{H}, \mathcal{E}$ et $R, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} R', \mathcal{H}', \mathcal{E}'$. On prouve qu'il existe Γ' tel que $\Gamma \subseteq \Gamma'$ et $\Gamma' \triangleright R', \mathcal{H}', \mathcal{E}'$. On note (1),(2),(3) et (4) (resp.(1′),(2′),(3′) et (4′)) les conditions associées à $\Gamma \triangleright R, \mathcal{H}, \mathcal{E}$ (resp. $\Gamma \triangleright R', \mathcal{H}', \mathcal{E}$). La preuve est par cas sur la règle utilisée pour la réduction.

- Supposons que la réduction soit une instance de la règle (*context*$_1$), i.e. $R \equiv \langle E[P]^t \cdot T, T' \rangle_\ell \mid R_1$, $R' \equiv \langle E[P']^t \cdot T, T \rangle'_\ell \mid R_1$ et $\vdash_{t,\ell} P, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} P', \mathcal{H}', \mathcal{E}'$ pour t et ℓ donnés. Par le lemme 5, pour prouver (1′), il suffit de prouver que si $\Gamma \rhd P : E_0$ alors il existe E'_0 tel que $E'_0 \leq E_0$ $\Gamma' \rhd P' : E'_0$. On procède par cas sur la règle utilisée pour la réduction $\vdash_{t,\ell} P, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} P', \mathcal{H}', \mathcal{E}'$.
  - Pour les règles (*call*), (*match*$_1$), (*match*$_2$), (*seq*) (*watch*),(*link*),(*unlink*),(*get*), (*await*) et (*pre*) on a $\mathcal{H}' = \mathcal{H}$ et $\mathcal{E}' = \mathcal{E}$. En particulier, pour $\Gamma' = \Gamma$, (2′), (3′) et (4′) sont vérifiées et on prouve (1′).
    - Supposons que la réduction est une instance de la règle (*call*), i.e. $P = A(\mathbf{e})$ et $P' = P''[\mathbf{v}/\mathbf{x}]$ avec $A(\mathbf{x}) = P''$ et $\mathbf{e} \Downarrow \mathbf{v}$. Si $\Gamma \rhd P : E_0$ alors, par la règle (*CALL*) et par définition, on a $\Gamma, \mathbf{x} : \overline{\tau} \rhd P'' : E_0$ et $\Gamma \vdash \mathbf{e} : \overline{\tau}$ pour $\overline{\tau}$ fixés. Par hypothèse sur le typage des expressions et par le lemme 4, il vient $\Gamma \rhd P' : E_0$.
    - Supposons que la réduction est une instance de la règle (*match*$_1$), i.e $P = [e \trianglerighteq p]\ P_1, P_2$ et $P' = P_1\sigma$ avec $e \Downarrow v$ et $\sigma = match(v, p)$. Si $\Gamma \rhd P : E_0$ alors, par la règle (*MATCH*), on a $\Gamma, p : \tau \rhd P_1 : E_1$, $\Gamma \vdash e : \tau$ et $E_1 \leq E_0$. On a $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \rhd P_1 : E_1$ et $\Gamma \vdash \sigma x_i : \tau_i$ pour $i = 1, \ldots, n$ avec $x_1, \ldots, x_n$ et $\tau_1, \ldots, \tau_n$ fixés. Par hypothèse sur le typage des expressions et par le lemme 4, il vient $\Gamma \rhd P' : E_1$. Le cas (*match*$_2$) est similaire.
    - Supposons que la réduction est une instance de la règle (*seq*), i.e. $P = 0; P''$ et $P' = P''$. Si $\Gamma \rhd P : E_0$ alors, par la règle (SEQ), on a $\Gamma \rhd P' : E_1$ et $E_1 \leq E_0$.
    - Si la réduction est une instance de l'une des règles (*watch*), (*link*) et (*unlink*) alors le résultat est immédiat puisque $\Gamma \rhd 0 : \texttt{none}$ et $\texttt{none} \leq E_0$ par définition de $\leq$.
    - Supposons que la réduction est une instance de la règle (*get*), i.e. $P = (x = \texttt{get}\ e) \cdot P''$ et $P' = P''[v/x]$ avec $e \Downarrow r_o$ et $\mathcal{H}(r_o) = v$. Si $\Gamma \rhd P : E_0$ et $E_0 = \texttt{access}(\ell)$ alors, par la règle (GET_2), on a $\Gamma, x : \tau \rhd P'' : E_1$ et $\Gamma \vdash e : \tau\ ref_\ell$ avec $E_1 \leq E_0$ pour τ fixé. Par hypothèse sur le typage des expressions on a $\Gamma \vdash r_o : \tau\ ref_\ell$ et par l'hypothèse (3) on a $\Gamma \vdash v : \tau$. Par le lemme 4 il vient $\Gamma \rhd P' : E_1$. Le cas $E_0 = \texttt{none}$ est similaire en utilisant la règle (GET_1).
    - Les cas (*await*) et (*pre*) sont similaires en utilisant les règles (*AWAIT*), (*PRE*) et l'hypothèse (4).
  - Supposons que la réduction est une instance de (ref_2), i.e. $P = (x = \texttt{ref}_\ell\ e) \cdot P''$, $P' = P''[r_\ell/x]$ et $\mathcal{H}' = \mathcal{H} \cup \{r_\ell = v\}$ où $e \Downarrow v$. Si $\Gamma \rhd P : E_0$ alors par la règle (REF_2) on a $E_0 = \texttt{access}(\ell)$, $\Gamma, x : \tau\ ref_\ell \rhd P'' : E_1$ et $\Gamma \vdash e : \tau$ avec $Public(\tau)$ et $E_1 \leq \texttt{access}(\ell)$. On prend $\Gamma' = \Gamma, r_\ell : \tau$. Puisque $\mathcal{E}' = \mathcal{E}$, il suffit de montrer (1′), (2′) et (3′). Par le lemme 4 on a $\Gamma' \vdash P' : E_1$ ((1') est vérifiée). Par l'hypothèse (2) et par définition de Γ', on a bien (2′), i.e. $Dom(\Gamma') = Dom(\mathcal{H}') \cup Dom(\mathcal{E}')$. Par (3) et par hypothèse sur le typage des expressions on a bien (3′), i.e. $\Gamma \vdash v : \tau$ et $Public(\tau)$. Si la réduction est une instance de (ref_1), la preuve est similaire.
  - Supposons que la réduction est une instance de la règle (*set*), i.e. $P = \texttt{set}\ e_1\ e_2$, $P' = 0$ et $\mathcal{H}' = \mathcal{H}[r_o := v]$ avec $e_1 \Downarrow r_o$ et $e_2 \Downarrow v$. On prend $\Gamma' = \Gamma$ et donc (2′) est vérifiée. Si $\Gamma \rhd P : E_0$ et $E_0 = \texttt{access}(\ell)$ alors, par la règle (SET_2), on a $\Gamma \vdash e_1 : \tau\ ref_\ell$ et $\Gamma \vdash e_2 : \tau$ et on en déduit que $o = \ell$. Par définition, $\Gamma \rhd P' : \texttt{none}$ ((1') est vérifiée) et par hypothèse sur le typage des expressions, on a $\Gamma \vdash v : \tau$ et $Public(\tau)$ est vérifiée par (3) (3') est vérifiée. La condition (4′) est immédiate. Le cas $o = t$ et $E = \texttt{none}$ est similaire.
  - Les cas (*sig*) et (*emit*) sont similaires aux cas (ref_2) et (set_2) en utilisant respectivement les règles (SIG) et ($EMIT$) et l'hypothèse (4).
- Supposons que la réduction soit une instance d'une des règles (*thread*$_1$) et (*thread*$_2$). Puisque ni $\mathcal{H}$ ni $\mathcal{E}$ ne sont modifiés, il suffit de vérifier (1') en prenant $\Gamma' = \Gamma$. En particulier, il suffit de vérifier que $\Gamma \rhd E[0] : \texttt{none}$ et $\Gamma \rhd A(\mathbf{e}) : \texttt{none}$. Puisque $\Gamma \rhd E[\texttt{run A(e)}] : \texttt{none}$ alors, par

un raisonnement identique au cas *call* et par la règle (*THREAD*), on a $\Gamma \rhd A(\mathbf{e}) : \texttt{none}$. Par définition, on a $\Gamma \rhd 0 : \texttt{none}$ et donc $\Gamma \rhd E[0] : \texttt{none}$.

– Dans le cas de la fin d'instant (2′) et (3′) sont immédiate.Pour la condition (1′), on note simplement que si $\Gamma \rhd \texttt{watch}\ e\ P_1\ P_2 : E$ alors, par la règle (*WATCH*), on a $\Gamma \rhd P_2 : E_2$ avec $E_2 \leq E$; on conclut par le lemme 5. La condition (3') est évidement vérifiée puisque pour tout $s \in Dom(\mathcal{E}') = Dom(\mathcal{E})$ on $\mathcal{E}'_c(s) = \emptyset$ et $\mathcal{E}'_p(s) = [v_1; \ldots; v_n]$ avec $v_i \in \mathcal{E}_c(s)$, et donc $\Gamma \vdash v_i : \tau$ par (3), pour $i = 1, \ldots, n$.

– Si la réduction est une instance de l'une des règles (*cooperate*), (*in*), (*out*) et (*move*), le résultat est immédiat puisque ces règles ne modifient pas le type des threads et que $\mathcal{H}' = \mathcal{H}$ et $\mathcal{E}' = \mathcal{E}$.

□

10.3 Preuve du lemme 3

Soit Γ un environnement de typage et $R, \mathcal{H}, \mathcal{E}$ un système tels que $\Gamma \rhd R, \mathcal{H}, \mathcal{E}$ et $R, \mathcal{H}, \mathcal{E}$ bien formé. On prouve que si $R, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} R', \mathcal{H}', \mathcal{E}'$ alors $R', \mathcal{H}', \mathcal{E}'$ est bien formé. La preuve est par cas sur la règle utilisée pour la réduction.

– Supposons que la réduction est une instance de la règle (*context*$_1$), i.e. $R \equiv \langle E[P]^t \cdot T, T' \rangle_\ell \mid R_1$, $R' \equiv \langle E[P']^t \cdot T, T' \rangle_\ell \mid R_1$ et $\vdash_{t,\ell} P, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} P', \mathcal{H}', \mathcal{E}'$. De manière évidente, pour tout thread différent de *t*, *acc* est inchangée. Il suffit donc de vérifier que $acc(P') \subseteq Ref_t$ et que pour tout thread t' de R et pour tout $r_{t'} \in Dom(\mathcal{H})$, on a $acc(\mathcal{H}(r_{t'})) \subseteq Ref_{t'}$. La preuve est par cas sur la règle utilisée pour la réduction $\vdash_{t,\ell} P, \mathcal{H}, \mathcal{E} \underset{M}{\longrightarrow} P', \mathcal{H}', \mathcal{E}'$.
  – Pour les règles (*call*), (*match*$_1$), (*match*$_2$), (*seq*), (*watch*), (*link*), (*unlink*), (*get*), (*sig*), (*await*) et (*pre*) la seconde condition est vérifiée puisque $\mathcal{H}' = \mathcal{H}$; il suffit donc de vérifier que $acc(P') \subseteq Ref_t$.
    – Supposons que la réduction est une instance de la règle (*call*), i.e. $P = A(\mathbf{e})$, $P' = P''[\mathbf{v}/\mathbf{x}]$ et $A(\mathbf{x}) = P''$. Par construction et par hypothèse sur l'évaluation des expressions on a $acc(P') \subseteq acc(\mathbf{v}) \subseteq acc(\mathbf{e})$. Or $acc(\mathbf{e}) \subseteq acc(P) \subseteq Ref_t$ par hypothèse.
    – De la même manière, si la réduction est une instance de l'une des règles (*match*$_1$), (*match*$_2$) (*seq*), (*watch*), (*link*), (*unlink*), on note simplement que $acc(P') \subseteq acc(P) \subseteq Ref_t$ par hypothèse.
    – Supposons que la réduction est une instance de la règle (*get*), i.e. $P = (x = \texttt{get}\ e) \cdot P''$ et $P' = P''[v/x]$ avec $e \Downarrow r_o$ et $\mathcal{H}(r_o) = v$. Par hypothèse, $acc(P) \subseteq Ref_t$ et il suffit de vérifier que $acc(v) \subseteq Ref_t$. Soit $o = t$ ($o = t'$ avec $t' \neq t$ n'est pas possible puisque $r_o \in acc(P)$) et la condition est vérifiée, soit $o \in \mathcal{L}$. Si $o = \ell$ pour $\ell \in \mathcal{L}$ alors, en particulier, $\Gamma \vdash \mathcal{H}(r_o) : \tau$ pour un certain τ tel que $Public(\tau)$, i.e. $acc(v) = \emptyset$.
    – Si la réduction est une instance de l'une des règles (*await*) et (*pre*) alors le raisonnement est similaire au cas (*get*) avec $o \in \mathcal{L}$.
  – Supposons que la réduction est une instance de la règle (ref_1), i.e $P = (x = ref_{self}\ e) \cdot P''$, $P' = P''[r_t/x]$ et $\mathcal{H}' = \mathcal{H} \cup \{r_t \mapsto v\}$ avec $e \Downarrow v$. La première condition est bien vérifiée puisque, par hypothèse et en notant que $r_t \in Ref_t$, on a bien $acc(P') \subseteq Ref_t$. Pour vérifier la seconde condition, par hypothèse, il suffit de vérifier que $acc(v) \subseteq Ref_t$ puisque $\mathcal{H}$ et $\mathcal{H}'$ ne diffère que pour r_t. Or par hypothèse, on a $acc(v) \subseteq acc(e) \subseteq acc(P) \subseteq Ref_t$.
  – Si la réduction est une instance de la règle (ref_2), i.e $P = (x = ref_\ell\ e) \cdot P''$, $P' = P''[r_\ell/x]$ et $\mathcal{H}' = \mathcal{H} \cup \{r_\ell \mapsto v\}$ avec $e \Downarrow v$. Le résultat est immédiat puisque $r_\ell \notin acc(P)$ et $\mathcal{H}$ et $\mathcal{H}'$ ne diffère sur aucune référence associée à un thread.

- Si la réduction est une instance de (*set*), i.e. $P = set\ e_1\ e_2$ et $P' = 0$ avec $e_1 \Downarrow r_o$ et $e_2 \Downarrow v$. La première condition est évidement vérifiée puisque $acc(0) = \emptyset$. Pour la seconde condition, on distingue deux cas selon que $o = t$ ou $o = \ell$ pour $\ell \in Loc$. Si $o = t$ alors, par hypothèse, $acc(v) \subseteq acc(e) \subseteq acc(P) \subseteq Ref_t$; or $\mathcal{H}$ et $\mathcal{H}'$ ne diffèrent que pour r_o. Si $o = \ell$ alors le résultat est immédiat, par hypothèse, puisque $\mathcal{H}$ et $\mathcal{H}'$ ne diffèrent sur aucune référence associée à un thread.
- Si la réduction est une instance de la règle ($context_2$), le raisonnement est similaire au cas ($context_1$).
- Si la réduction est une instance de la règle ($thread_2$) (le cas $thread_1$ est similaire), i.e $R \equiv E[\texttt{run}\ A(\mathbf{e})]^t \mid R_1$ et $R' \equiv E[0]^t \mid P[\mathbf{v}/\mathbf{x}]^{t'} \mid R_1$ avec $A(\mathbf{x}) = P$ et $\mathbf{e} \Downarrow \mathbf{v}$. La seconde condition est évidemment vérifiée, par hypothèse, puisque $\mathcal{H}' = \mathcal{H}$. Pour la première condition, il suffit de vérifier que $acc(E[0]) \subseteq Ref_t$ et $acc(P[\mathbf{v}/\mathbf{x}]) \subseteq Ref_{t'}$. Pour $acc(E[0])$, le résultat est immédiat par hypothèse. On sait que $\Gamma \vdash \mathbf{e} : \overline{\tau}$ et $Public(\overline{\tau})$. En particulier, par hypothèse sur l'évaluation des expressions, $acc(P[\mathbf{v}/\mathbf{x}]) \subseteq acc(\mathbf{e}) = \emptyset$.
- Si la réduction est une instance de la règle (*eoi*). La seconde condition est évidemment vérifiée, par hypothèse, puisque $\mathcal{H}' = \mathcal{H}$. Pour la première condition, il suffit de vérifier que pour tout P, $\mathcal{E}_0$ et ℓ_0 on a $acc(\lVert P \rVert_{\mathcal{E}_0,\ell_0}) \subseteq acc(P)$. La preuve est par induction structurelle sur P.
- Si la réduction est une instance de l'une des règles (*cooperate*), (*in*),(*out*), et (*move*) alors il est immédiat que les deux conditions sont vérifiées par hypothèse.

10.4 Preuve du théorème 2

Soit P un programme bien typé et $R_0, \mathcal{H}_\emptyset, \mathcal{E}_\emptyset$, avec $R_0 = \langle \emptyset, \emptyset \rangle_{\ell_n} \mid \ldots \mid \langle \emptyset, \emptyset \rangle_{\ell_n} \mid P^{t_0}$, le système initial associé. Par définition d'un programme bien typé et en notant que $acc(P) = \emptyset$ et $Dom(\mathcal{H}_\emptyset) = \emptyset$, on a $\triangleright R_0, \mathcal{H}_\emptyset, \mathcal{E}_\emptyset$ et $R_0, \mathcal{H}_\emptyset, \mathcal{E}_\emptyset$ est bien formé. On montre que le programme est compatible avec les droits d'accés. Soit $S_0, S_1, \ldots$ une exécution du programme et soit $i \geq 0$. Supposons que $S_i \xrightarrow[(t,r_o)]{} S_{i+1}$. Par induction sur la longueur de la dérivation $S_0, S_1, \ldots, S_i$, d'après les lemmes 2 et 3, il est immédiat que S_i est bien formé et qu'il existe Γ tel que $\Gamma \triangleright S_i$. On doit montrer que soit $o = t$, soit $o = \ell$ pour $\ell \in Loc$ et t est actif en ℓ. La preuve est par cas sur la règle utilisée pour la réduction. En particulier, il est suffisant de montrer le résultat pour les réductions de la forme $S_i \xrightarrow[M]{} S_{i+1}$ telles que $M \neq \emptyset$. On peut donc supposer que la réduction est une instance de la règle ($context_1$) (le cas ($context_2$) est similaire). En particulier, $S_i = R_i, \mathcal{H}_i, \mathcal{E}_i$ et $S_{i+1} = R_{i+1}, \mathcal{H}_{i+1}, \mathcal{E}_{i+1}$ où $R_i \equiv \langle E[P]^t \cdot T, T' \rangle_\ell \mid R'_1$ et $R_{i+1} \equiv \langle E[P']^t \cdot T, T' \rangle_\ell \mid R'_1$ avec $\vdash_{t,\ell} P, \mathcal{H}, \mathcal{E} \xrightarrow[(t,r_o)]{} P', \mathcal{H}', \mathcal{E}'$, on procède par cas sur la règle utilisée pour la réduction.

- Supposons que la règle est une instance de la règle (*get*), i.e. $P = (x = \texttt{get}\ e) \cdot P''$ et $P' = P''[v/x]$ où $e \Downarrow r_o$ et $v = \mathcal{H}(r_o)$. Par définition de la sémantique, on a $E[P] \odot\!\!\rightarrow \ell$ et donc $\Gamma \triangleright P : E$ avec $E \leq \texttt{access}(\ell)$ par le lemme 7. En particulier, soit $o = \ell$ (règle (GET_2)) soit o est un nom de thread (règle (GET_2)). Si o est un nom de thread alors $o = t$ puisque, par hypothèse, $\{r_o\} \subseteq acc(e) \subseteq acc(E[P]) \subseteq Ref_t$.
- Si la règle utilisée est une instance d'une des règles (ref_1), (ref_2) et (*set*), le raisonnement est similaire.

Chapitre 11

Conclusion

Dans ce manuscript, nous nous sommes intéressé à deux propriétés des langages de programmation de la famille des langages réactifs. De manière générale, la vérification de ces deux propriétés, essentielles aux langages considérés, était jusqu'ici de la responsabilité du programmeur. A travers une analyse de la sémantique des modèles de programmation sous-jacents, nous avons proposé des définitions formelles de ces propriétés et des outils d'analyse statique qui permettent de garantir que celles-ci sont vérifiées par les programmes.

Dans la première partie de ce document, nous avons étudié la question de la réactivité des programmes pour un sous-ensemble du π-calcul synchrone. Plus précisément, nous avons proposé une propriété, la réactivité efficace, qui assure que, à chaque instant, la taille du programme et le nombre d'étapes de calcul sont polynômiales en la taille initiale du programme et en la taille des entrées du système. Nous avons également proposé une analyse statique qui garantit qu'un programme est réactif efficace. Cette analyse statique repose sur la génération de contraintes permettant, lorsque celles-ci sont interprétées dans une structure adaptée, de prouver la terminaison des instants et de contrôler la taille du système. En particulier, cette analyse repose sur des techniques empruntées aux systèmes de réécriture (preuves de terminaison) et sur une adaptation des quasi-interprétations à notre modèle (en particulier par la mise en oeuvre d'une orientation des flots de données).

Les résultats présentés ici concernent un sous-ensemble du π-calcul synchrone ; la création dynamique de threads est interdite et le nombre de lectures pouvant être réalisées par un thread au cours d'un même cycle est limité. Par la suite, nous espérons dépasser ces deux restrictions. D'une part, dans le cadre des contraintes portant sur la taille du système, nous pourrions envisager de considérer une interprétation additive de la composition parallèle. Par exemple, si un appel à A peut générer deux appels à B, A devrait contrôler la somme des deux appels et non pas contrôler individuellement chaque appel comme cela et le cas actuellement. D'autre part, il nous semble maintenant que l'intérêt de la condition read-once est purement technique ; celle-ci permet uniquement de simplifier les raisonnements. Nous essaierons donc par la suite d'adapter les résultats de manière à ce que nous puissions nous passer de cette restriction.

En partant de ces améliorations, nous espérons pouvoir réaliser une implantation expérimentale du modèle de calcul et de l'analyse statique correspondante. La synthèse des types, et en particulier des régions associées au signaux ne semble pas poser de problème. La difficulté principale réside dans la synthèse des quasi-interprétations. De ce point de vue, on peut noter que les contraintes générées par notre analyse statique sont similaires à celles considérés dans le cadre des langages fonctionnels du premier ordre. En particulier, il sera donc possible d'utiliser les outils développés dans ce cadre ; l'utilisation des quasi-interprétations multi-linéaires sur l'algèbre max-plus représenterait une première

étape intéressante permettant de considérer des systèmes non-size-increasing.

Une autre piste de recherche intéressante pour la suite de ce travail concerne le problème de la caractérisation de l'expressivité de l'analyse statique proposée. De ce point, les fonctions continues sur les flots calculées par les réseaux de Kahn[66] constituent un bon point de départ. Quelles fonctions continues sont calculables par les programmes réactifs efficaces du π-calcul synchrone. Par exemple, la fonction qui à chaque instant émet sur un signal la liste des valeurs émises sur un autre signal depuis le début du calcul est clairement continue mais n'est pas expressible par un programme réactif efficace (ce calcul suppose une mémoire non bornée). Pour cette étude, il sera intéressant de considérer un sous ensemble déterministe du π-calcul synchrone.

Dans la seconde partie de ce document, nous avons présenté le langage PACT qui nous a permis de mieux étudier l'utilisation du style de programmation introduit par les FAIR THREADS, i.e. le mélange des styles coopératif et préemptif, dans le cadre des architectures modernes à base de plusieurs unités de calcul concurrentes. Comme nous l'avons vu, ce modèle de programmation cherche à dépasser les limitations d'une approche purement coopérative en plongeant celle-ci dans un sur-ensemble permettant de bénéficier de certains des avantages d'une programmation basée sur un ordonnancement préemptif. Nous avons vu que la librairie des FAIR THREADS ne parvient pas totalement à atteindre ce but ; en particulier, l'atomicité de l'exécution du code placé entre deux points de coopération d'une thread n'est pas garantie. Par rapport aux travaux existants nous avons apporté : (1) la définition d'une sémantique formelle de ce style de programmation dans le cadre d'un langage du premier ordre, (2) la définition d'une discipline de programmation permettant d'atteindre le but mentionné ci-dessus et une analyse statique permettant de garantir le respect de cette dernière.

Par la suite, nous envisageons d'approfondir l'étude du paradigme de programmation proposé par PACT. Certaines expérimentations nous conduisent à penser qu'il serait intéressant de séparer l'aspect synchronisation (partage des instants et de la visibilité des signaux) de l'aspect partage de la mémoire. Nous envisageons de distinguer au sein de chaque zone synchrone plusieurs zones mémoires, partagées par des groupes distincts de threads. De cette manière, il serait possible pour des threads de se synchroniser, i.e. de partager les instants et de communiquer par signaux valués, tout en bénéficiant d'une exécution parallèle (actuellement, un thread ne peut bénéficier de ces deux possibilités simultanément). Un tel modèle pourrait être vu comme une construction sur trois niveaux correspondants à différentes utilisations de l'architecture : (1) les threads libres : utilisation de plusieurs cores, (2) les zones synchrones : utilisation de plusieurs cores synchronisés et (3) les threads partageant la mémoire : séquentialisation de l'exécution. On peut noter ici que le nouveau niveau envisagé (le niveau 2) correspond au modèle de programmation proposé par le $S\pi$-calcul. Il nous semble également qu'il serait intéressant de considérer ce modèle de programmation indépendamment de l'aspect contrôle des ressources. Pour cela nous envisageons de réaliser une implémentation du futur langage au dessus d'un langage tel qu'OBJECTIVE CAML afin de proposer une alternative aux threads utilisés dans ce dernier. Dans cette nouvelle version du langage, nous essaierons également de considérer la possibilité de créer dynamiquement de nouvelles zones synchrones.

Pour conclure, nous pouvons noter la réalisation, par F. Boussinot, d'un langage expérimental, appelé FUN LOFT [1], basé sur les développements de ce manuscript. Le choix d'un noyau fonctionnel du premier ordre pour le langage PACT découle de la volonté de combiner, au moins partiellement, les méthodes développées ici au sein de ce langage. Le langage FUN LOFT est très proche du langage PACT et permet, en particulier, la programmation d'architectures multi-cores. Cette implémentation comporte un algorithme d'inférence (non encore formalisé) pour un système de types proche de celui de PACT. Les résultats de la partie I, portant sur le π-calcul synchrone, sont également utilisés en partie dans FUN LOFT. L'analyse statique mise en oeuvre, qui ne repose pas sur les quasi-interprétations, assure une propriété plus faible que celle proposée pour le π-calcul synchrone. En particulier, les bornes

sur la taille du système et sur les temps de réaction existent mais ne sont pas mises en évidence par celle-ci. Cette restriction permet toutefois de garantir : (1) que les réactions terminent toujours et (2) l'absence de *fuite mémoire*. Malgrès ces restrictions, l'expressivité du langage reste raisonnable comme le montrent les exemples proposés[1].

Bibliographie

[1] The fun loft web page : http ://www-sop.inria.fr/mimosa/rp/funloft/index.html.

[2] Raul Acosta-Bermejo. Reactive operating system, reactive java objects. In *Proceedings of NOTERE'2000*, 2000.

[3] Roberto M. Amadio. Max-plus quasi-interpretations. In *Proceedings of Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[4] Roberto M. Amadio. The SL synchronous language, revisited. *Journal of Logic and Algebraic Programming*, 70 :121–150, 2005.

[5] Roberto M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2) :29 :60, 2005.

[6] Roberto M. Amadio. A synchronous pi-calculus. Technical Report hal-00078319, Université Paris 7, Laboratoire PPS, 2006.

[7] Roberto M. Amadio, Gérard Boudol, Frédéric Boussinot, and Illaria Castellani. Reactive programming, revisited. *Electronic Notes in TCS*, 162 :49–90, 2005.

[8] Roberto M. Amadio, Solange Coupet-Grimal, Silvano Dal-Zilio, and Line Jakubiec. A functional scenario for bytecode verification of resource bounds. In *Proceedings of CSL 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[9] Roberto M. Amadio and Frédéric Dabrowski. Feasible reactivity for synchronous cooperative threads. *Electronic Notes in TCS*, 154 :3 :33–43, 2005.

[10] Roberto M. Amadio and Silvano Dal Zilio. Resource control for synchronous cooperative threads. *Theoretical Computer Science*, 358 :229–254, 2004.

[11] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile resource guarantees for smart devices. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices : Proceedings of the International Workshop CASSIS 2004*, number 3362 in Lecture Notes in Computer Science, page 1–26. Springer-Verlag, 2005.

[12] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[13] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *STOC '92 : Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, page 283–293, New York, NY, USA, 1992. ACM Press.

[14] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, 79(9) :1270–1282, 1991.

[15] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[16] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[17] Andrew D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, Palo Alto, California, 1989.

[18] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *CSL*, page 372–384, 1998.

[19] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. On lexicographic termination ordering with space bound certifications. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Novosibirsk, Russia*, volume 2244 of *Lecture Notes in Computer Science*. Springer, Jul 2001.

[20] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*, (en révision).

[21] Gérard Boudol. ULM : A core programming model for global computing. In *Proceedings of the 13th European Symposium on Programming, ESOP'04*, volume 2986 of *LNCS*. Springer-Verlag, 2004.

[22] Frédéric Boussinot. Reactive c : an extension of c to program reactive systems. *Softw. Pract. Exper.*, 21(4) :401–428, 1991.

[23] Frédéric Boussinot. Icobj programming. Technical Report 3028, INRIA, 1996.

[24] Frédéric Boussinot. Java fair threads. Technical Report RR-4139, INRIA, 2002.

[25] Frédéric Boussinot. Operational semantics of cooperative fair threads, unpublished, 2002.

[26] Frédéric Boussinot. Fairthreads : mixing cooperative and preemptive threads in C. *Concurr. Comput. : Pract. Exper.*, 18(5) :445–469, 2006.

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

[28] Frédéric Boussinot, Laurent Hazard, and Jean-Ferdy Susini. The junior reactive kernel. Technical Report 3732, INRIA, 1999.

[29] Frédéric Boussinot and Jean-Ferdy Susini. The sugarcubes tool box. Technical Report 3247, INRIA, 1997.

[30] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming : preventing data races and deadlocks. In *OOPSLA '02 : Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, page 211–230, New York, NY, USA, 2002. ACM Press.

[31] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa Bay, FL, 2001.

[32] Christian Brunette. *Construction et simulation graphiques de comportements : le modèle des Icobjs*. PhD thesis, Université de Nice, 2004.

[33] Paul Caspi and Marc Pouzet. Synchronous kahn networks. In *International Conference on Functional Programming*, page 226–238, 1996.

[34] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications With Objective Caml*. O'Reilly France, 2000.

[35] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02 : Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, page 258–269, New York, NY, USA, 2002. ACM Press.

[36] Adam Cichon. Bounds on derivation lengths from termination proofs. Technical Report CSD-TR-622, Department of Computer Science, University of London, 1990.

[37] Alan Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Logic, Methodology, and Philosophy of Science*, page 24–30. North-Holland, 1964.

[38] The crocus web page : http ://libresource.inria.fr//projects/crocus.

[39] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3) :279–301, 1982.

[40] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, page 243–320. MIT Press, 1990.

[41] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8) :465–476, 1979.

[42] Edsger W. Dijkstra. Cooperating sequential processes. page 65–138, 2002.

[43] Stéphane Epardaud. Mobile reactive programming in ULM. In *2004 Scheme Workshop*, September 2004.

[44] Cormac Flanagan and Martin Abadi. Types for safe locking. In *ESOP '99 : Proceedings of the 8th European Symposium on Programming Languages and Systems*, page 91–108, London, UK, 1999. Springer-Verlag.

[45] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *PLDI '00 : Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, page 219–232, New York, NY, USA, 2000. ACM Press.

[46] Cormac Flanagan and Stephen N Freund. Atomizer : a dynamic atomicity checker for multithreaded programs. In *POPL '04 : Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 256–267, New York, NY, USA, 2004. ACM Press.

[47] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity, 2003.

[48] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI '03 : Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, page 1–12, New York, NY, USA, 2003. ACM Press.

[49] Georges Gonthier. *Sémantique et modèles d'exécution des langages réactifs synchrones : applications à Esterel*. PhD thesis, Université d'Orsay, 1988.

[50] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[51] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, 79(9) :1305–1320, 1991.

[52] Kevin Hammond. An abstract machine for resource-bounded computations in hume. In *Draft paper submitted to IFL '03˘2014 Implementation of Functional Languages, Edinburgh, Scotland*, 2003.

[53] Kevin Hammond and Greg Michaelson. The design of hume : a high-level language for the real-time embedded systems domain. In *TOCHECK*, 2003.

[54] Kevin Hammond and Greg Michaelson. Hume : a domain-specific language for real-time embedded systems. In *Proc. Conf. Generative Programming and Component Engineering*. Lecture Notes in Computer Science, Springer-Verlag, 2003.

[55] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (tcc). *SIGOPS Oper. Syst. Rev.*, 38(5) :1–13, 2004.

[56] Thérèse Hardin. Produire un logiciel de confiance : quelles hypothèses, quelles limites ? In *Journées Francophones des Langages Applicatifs*, 2002.

[57] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems, NATO Advanced Summer Institutes*, volume F-13, page 477–498. Springer-Verlag, 1985.

[58] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03 : Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, page 388–402, New York, NY, USA, 2003. ACM Press.

[59] C. A. R. Hoare. Monitors : an operating system structuring concept. *Commun. ACM*, 17(10) :549–557, October 1974.

[60] Dieter Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *Theor. Comput. Sci.*, 105(1) :129–140, 1992.

[61] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic J. of Computing*, 7(4) :258–289, 2000.

[62] Martin Hofmann. The strength of non-size increasing computation. In *POPL '02 : Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 260–269. ACM Press, 2002.

[63] John Ousterhout. Why threads are a bad idea (for most purposes) - http ://www.sunlabs.com/ ouster/, 1996.

[64] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, page 303–310. ACM Press, 1991.

[65] The MRG group web page : http ://groups.inf.ed.ac.uk/mrg/.

[66] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, volume 74. North-Holland, 1974.

[67] Sam Kamin and Jean-Jacques Lévy. Attempts for generalizing the recursive path orderings. Rapport technique, Université de l'Illinois, Urbana, 1980.

[68] Jan Willem Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Volumes 1 (Background : M athematical Structures) and 2 (Background : Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. Cambridge University Press, 1992.

[69] D.S. Landford. On proving term rewriting systems are noetheriens. Technical Report MTP-3, Louisiana Technical University, 1979.

[70] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real time applications with signal. *Proc. IEEE, Special Issue*, 79 :1321–1336, 1991.

[71] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3) :81–92, 2001.

[72] Daniel Leivant. A foundational delineation of computational feasibility. In *lics91*, page 2–11, 1991.

[73] Louis Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6, 2006.

[74] Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. In *Synchronous Languages, Applications and Programming (SLAP'05)*. ENTCS, 2005.

[75] Louis Mandel and Marc Pouzet. Reactiveml : a reactive extension to ml. In *PPDP '05 : Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, page 82–93, New York, NY, USA, 2005. ACM Press.

[76] Jean-Yves Marion. *Complexité implicite des calculs, de la théorie à la pratique*. Habilitation à diriger les recherches, Université Nancy 2, 2000.

[77] Jean-Yves Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183(1) :2–18, 2003.

[78] Jean-Yves Marion and Jean-Yves Moyen. Efficient first order functional program interpreter with time bound certifications. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France*, volume 1955 of *Lecture Notes in Computer Science*, page 25–42. Springer, Nov 2000.

[79] Greg Michaelson and Kevin Hammond. Hume : a functionally-inspired language for safety-critical systems. In *Draft Proc. 2nd Scottish Functional Programming Workshop, St Andrews, Scotland*, 2000.

[80] Greg Michaelson, Kevin Hammond, and Jocelyn Serot. The finite state-ness of FSM-hume. In *Draft paper submitted to TFP '03, Symposium on Trends in Functional Programming, Edinburgh, Scotland*, 2003.

[81] Greg Michaelson, Kevin Hammond, and Jocelyn Serot. FSM-hume : programming resource-limited systems using bounded automata. In *SAC '04 : Proceedings of the 2004 ACM symposium on Applied computing*, page 1455–1461. ACM Press, 2004.

[82] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[83] Robin Milner. The polyadic pi-calculus : a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, page 203–246. Springer-Verlag, 1993.

[84] Jean-Yves Moyen. System presentation : An analyser of rewriting systems complexity. Unpublished, 2001.

[85] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97 : The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 106–119, 1997.

[86] Robert H. B. Netzer and Barton P. Miller. What are race conditions ? : Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1) :74–88, 1992.

[87] Peter W. O'Hearn. Resources, concurrency, and local reasoning (abstract). In *ESOP*, pages 1–2, 2004.

[88] John Reynolds. Separation logic : a logic for shared mutable data structures. Master's thesis, , 2002.

[89] Michael F. Ringenburg and Dan Grossman. Atomcaml : first-class atomicity via rollback. In *ICFP '05 : Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, page 92–104, New York, NY, USA, 2005. ACM Press.

[90] The reactive programming web page : http ://www-sop.inria.fr/mimosa/rp.

[91] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser : a dynamic data race detector for multi-threaded programs. In *SOSP '97 : Proceedings of the sixteenth ACM symposium on Operating systems principles*, page 27–37, New York, NY, USA, 1997. ACM Press.

[92] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In *PPDP '04 : Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, page 203–214, New York, NY, USA, 2004. ACM Press.

[93] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95 : Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, page 204–213, New York, NY, USA, 1995. ACM Press.

[94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, page 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.

[95] Christoph von Praun and Thomas R. Gross. Object race detection. In *OOPSLA '01 : Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, page 70–82, New York, NY, USA, 2001. ACM Press.

[96] Andreas Weiermann. Termination proofs by lexicographic path orderings yield multiply recursive derivation lengths. *Theoretical Computer Science*, 139 :335–362, 1995.

[97] Jeannette M. Wing, Manuel Faehndrich, J. G Morrisett, and Scott Nettles. Extensions to standard ml to support transactions. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.