Vérification Statique de Programmes Répartis

Fabien Dagnat

N° d'ordre : 1784

THÈSE

soutenue en vue de l'obtention du titre de

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

spécialité : **informatique**

par

Fabien Dagnat

Vérification statique de programmes répartis

Soutenue le 28 Mai 2001, devant le jury composé de :

MM.	Louis FÉRAUD	Président
	Denis CAROMEL Marc GENGLER Didier RÉMY	Rapporteurs
	Élie NAJM Marc PANTEL	Examinateurs
	Patrick SALLÉ	Directeur de thèse

Mis en page avec la classe `thesisinpt` adaptation personnelle de la classe `thloria`.

Remerciements

Je remercie Louis Féraud, Professeur à l'université Paul Sabatier, pour l'honneur qu'il me fait de présider le jury de cette thèse.

Mes remerciements vont également à Denis Caromel, Professeur à l'université de Nice-Sophia Antipolis, Marc Gengler, Professeur à l'université d'Aix-Marseille II, et Didier Rémy, Directeur de Recherche à l'INRIA, pour avoir acceptés la lourde charge d'être rapporteur de ce travail. Les remarques et les parfois longues discussions, toujours pertinentes et passionnantes m'ont permis de faire progresser ce manuscrit et son contenu.

Je remercie Élie Najm, Professeur à l'École Nationale Supérieure de Télécommunication, de l'intérêt qu'il a porté à mon travail en acceptant de participer à ce jury.

J'adresse mes remerciements les plus sincères à Patrick Sallé et Marc Pantel, respectivement Professeur et Maître de Conférence à l'École Nationale Supérieure d'Électrotechnique, Électronique, Informatique et Hydraulique de Toulouse, d'avoir co-encadré ce travail. Toujours présents en cas de besoin, tout en me laissant une grande liberté dans mes choix, ils ont tout deux fortement contribués à la conclusion de ses travaux. Je les remercie également d'avoir lu et relu et ainsi contribué de façon importante à ce manuscrit.

J'exprime mes remerciements les plus vifs à tous les membres de l'équipe *sûreté de développement* pour m'avoir toujours chaleureusement reçu et épaulé que ce soit dans le cadre du laboratoire ou dans celui de l'enseignement. Je remercie plus particulièrement Matthias Colin pour ces quatre années de collaborations et de discussions toujours fructueuses.

Je tiens enfin à remercier largement l'ensemble des membres du laboratoire et du département de formation en Informatique de l'ENSEEIHT pour les conditions agréables et sympathiques de travail. Et plus particulièrement :

- Joseph Noailles et Gérard Padiou, Professeurs à l'ENSEEIHT et Directeurs successifs du Laboratoire d'Informatique et de Mathématiques Appliquées, pour l'accueil qu'ils m'ont réservé.
- Jean-Luc Basille et Alain Ayache, Professeurs à l'ENSEEIHT et Directeurs successifs du département de formation en Informatique et de Mathématiques Appliquées, de m'avoir permis de rejoindre l'équipe des enseignants en tant que Moniteur puis en tant qu'ATER.
- Annie Adelantado pour sa gentillesse toujours souriante et pour l'efficacité avec laquelle elle règle tout nos soucis administratifs.
- Bernard Coulette, pour la confiance et les collaborations qui m'ont permis de participer très activement à l'enseignement du Génie Logiciel et de la Programmation Objets aux élèves de l'ENSEEIHT.
- Ronan Guivarch qui m'a toujours soutenu et avec qui le travail a été un véritable plaisir.

Je dédie cette thèse
à Cécile, qui a dû en supporter tous les désagréments ;
à Lucas, dont la venue m'a comblé de joie même si elle a retardé ce manuscrit ;
à mes parents et surtout à ma mère qui n'a pu, hélas, voir la conclusion de ce travail...

Résumé

Dans les programmes acteurs ou objets concurrents, et plus généralement dans les logiciels clients/serveurs, certaines requêtes ne pourront pas être traitées par leur cible. Une telle requête est appelée message orphelin, elle peut être : soit un orphelin de sûreté (son destinataire ne pourra jamais la traiter), soit un orphelin de vivacité (son destinataire pourrait éventuellement la traiter, mais il n'atteindra pas l'état nécessaire à ce traitement).

Dans le cadre de l'équipe *Vestale* qui m'a accueilli, des systèmes de type ont été conçus pour détecter les orphelins de sûreté pour un calcul de processus modélisant les acteurs. Dans cette thèse, nous adaptons ces analyses statiques pour détecter certains problèmes de communication au sein de *vrais* langages de programmation. Le premier, ML-Act, une extension de ML avec des primitives du modèle d'acteurs conçue au sein de l'équipe *Vestale*. Le second, Erlang, est un langage fonctionnel concurrent et réparti conçu par ERICSSON pour construire des applications de télécommunication.

Pour détecter les orphelins, nos systèmes sont basés sur un processus d'inférence qui calcule le type des différentes entités du programme. Les types qui approximent les acteurs sont inspirés des types utilisés usuellement pour les enregistrements ou les objets. Les systèmes de type sont très sophistiqués, ils contiennent une notion de sous-typage et reposent sur des algorithmes d'inférence qui collectent des contraintes à partir du code source, puis les résolvent. Leur correction est démontrée en utilisant une sémantique opérationnelle du modèle d'acteurs reposant sur un entrelacement de deux réductions (une sur les expressions fonctionnelles et une sur les acteurs). Le formalisme modélisant les acteurs, appelé configuration, est général et commun aux deux langages (qui ne se distinguent donc que par le calcul fonctionnel).

En conclusion, nous faisons un bilan des évolutions des théories et techniques qui ont été nécessaires pour adapter des systèmes construits sur un calcul de processus à des langages de programmation complexes.

Mots-clés: Acteurs, Typage, Analyses statiques, Inférence, Erlang, Concurrence.

Abstract

In the context of actors, concurrent objects, and more generally in client/server software, some requests will not be treated by their target. Such a request is called an orphan message and can be either a safety orphan (its target will never be able to treat it) or a liveness orphan (its target could possibly treat it, but it will never reach the requested state for this processing).

Within our research team, some type systems have been built to detect safety orphans for a process calculus modeling actors. In this thesis, we adapt those static analyzes to the detection of some communication problems in *real* programming languages. Our work is conducted in the context of two real languages ; on one hand, ML-ACT, an extension of ML with primitives from the actor model designed by our research team, and on the other hand, ERLANG, a concurrent and distributed functional language designed by ERICSSON to program their telecommunication applications.

In order to detect orphans, our systems are based on an inference process which computes the type of all entities of the program. The types which approximate the actors are inspired by the types usually used for records or objects. The type systems are very sophisticated, they include a subtyping notion and rely on two inference algorithms which collect constraints from the source code, and then solve them. Their correction is demonstrated using an operational semantics of the actor model based on the interlacing of two reductions (one on functional expressions and one on actors). The formalism modeling actors, called *configuration* is general and common to both languages (which are distinguished only by the functional calculus).

To conclude, we make an analysis of the evolutions of theories and techniques which have been necessary to adapt systems built on process calculi to complex programming languages.

Keywords: Actors, Typing, Static Analysis, Type Inference, ERLANG, Concurrency.

Table des matières

Introduction

Face au développement fulgurant des réseaux dont le plus célèbre est *Internet*, les professionnels de l'informatique sont confrontés à une explosion de la demande de logiciels pour administrer et exploiter des groupes d'ordinateurs interconnectés. Or, ce type d'application, dite *répartie*, est bien plus difficile à mettre au point que les logiciels s'exécutant sur une machine isolée, dits *centralisés*.

Les deux principales causes de ces difficultés sont : d'une part, la *taille* atteinte par les logiciels actuels et d'autre part, l'*hétérogénéité* liée aux réseaux. Ces deux problèmes généraux ont un impact important sur la fiabilité des logiciels ainsi que sur leurs aspects coopératifs (problèmes de synchronisation, d'ordonnancement, ...).

La multiplication des utilisateurs potentiels et des contextes d'utilisations des machines informatiques provoque également une explosion de la complexité des logiciels. En effet, de nouvelles fonctionnalités, de nouveaux périphériques et de nouvelles interfaces homme machine sont régulièrement ajoutés. Ces extensions démocratisent l'ordinateur et permettent une généralisation de l'outil informatique qui a pour conséquence une croissance exponentielle de la quantité de code à développer. A titre d'exemple, Windows 2000, le système d'exploitation de Microsoft contiendrait dix millions de lignes de code [1], soit l'équivalent d'une encyclopédie comportant une centaine de volumes. De tels ordres de grandeur imposent un grand nombre d'intervenants pour réaliser le logiciel (5 000 programmeurs dans le cas de Windows 2000) et rendent donc illusoire, même avec un grand nombre de testeurs (750 000 versions β de Windows 2000 ont été distribuées), le fait d'obtenir une application exempte d'erreur logicielle grave.

Le second facteur d'explosion de la complexité des logiciels est l'hétérogénéité des contextes dans lesquels ils vont devoir s'intégrer. En effet, le contexte d'utilisation du logiciel n'est plus une machine spécifique figée (du point de vue matériel et logiciel) utilisée uniquement par des spécialistes pour des tâches bien précises. Il faut maintenant que le logiciel sache s'adapter à des supports informatiques très divers (Carte à puce JAVA, PDA, portable MAC, PC sous Windows, station de travail sous UNIX, ...). Il doit également prévoir que l'utilisateur peut être novice et donc qu'il ait besoin d'assistance. Ces utilisateurs non-spécialistes font souvent plus d'erreurs de manipulation qui ne doivent pas pour autant mettre en péril le logiciel. Enfin, les logiciels doivent être de plus en plus souples et polyvalents pour tenter de répondre aux attentes du plus grand nombre. Ces nouvelles exigences augmentent la taille des programmes mais surtout entraînent une croissance exponentielle du nombre de cas particuliers que les programmeurs doivent prévoir.

Cet environnement complexe pose d'importants problèmes de fiabilité et de sûreté de

1. Les chiffres cités dans ce paragraphe proviennent de communiqués publiés sur le site Internet de Microsoft : `http://www.microsoft.com`.

fonctionnement. Vu le rôle critique généralement joué actuellement par l'outil informatique, le comportement des anciens logiciels qui s'arrêtaient si un problème survenait, n'est plus acceptable. Un logiciel moderne doit être tolérant aux fautes et pouvoir fonctionner de manière dégradée (le moins possible) si certains services de son environnement ne sont plus disponibles. De plus, malgré une fiabilité physique des réseaux et ordinateurs sans cesse améliorée, les *pannes* sont des évènements courants qui ne peuvent donc pas être ignorées lors du développement d'une application.

D'autre part, l'évolution de la puissance des processeurs et la baisse de leur prix provoque une généralisation du parallélisme et de la répartition. Tous les logiciels se doivent donc d'utiliser au maximum la concurrence (que se soit sur un processeur ou sur plusieurs) et la répartition. Or, techniquement, la programmation concurrente est bien plus complexe que la programmation séquentielle. Elle nécessite plus de rigueur dans le traitement des échanges entre les différentes parties du programme s'exécutant simultanément. Cette synchronisation peut, par exemple, provoquer l'apparition de situations où deux portions de programme s'attendent mutuellement et bloquent donc l'exécution. Il devient également difficile pour le programmeur d'avoir une vision précise de l'enchaînement réel des opérations lors d'une exécution.

Le fonctionnement précis d'un programme ainsi que son environnement logiciel et matériel sont donc devenus difficilement prévisibles. En effet, l'ordre d'exécution des différentes tâches composant une application ne peut plus être prévu à cause d'un fort indéterminisme dû à la concurrence. Ainsi, certaines erreurs peuvent n'apparaître que durant certaines exécutions. Ceci augmente encore les difficultés de test (un même test peut fournir plusieurs résultats différents) et donc de garantie des produits informatiques.

Les erreurs

Les erreurs survenant lors de l'exécution d'un programme sont parfois peu pénalisantes, mais peuvent, également avoir des conséquences catastrophiques. En effet, les erreurs présentes au sein des logiciels ont un coût économique parfois très important, comme par exemple, la perte du premier lanceur Ariane V en 1996. Mais, au delà de ce coût, il convient de souligner surtout que les conséquences d'une erreur peuvent être dangereuses. Une erreur qui provoquerait l'arrêt ou bien le dysfonctionnement d'une application de gestion d'une centrale nucléaire ou d'une tour de contrôle aérien, pourrait avoir des conséquences dramatiques. Sans aller jusqu'à ces extrêmes, la récente panne logicielle d'une importante partie du réseau parisien de France Télécom le 3 mai 2000, montre qu'il est primordial de ne pas négliger ces erreurs cachées. Il faut également ajouter que les « *bugs* » sont souvent utilisés par les pirates afin de pénétrer au sein du réseau d'une entreprise pour y voler ou y détruire des données souvent vitales. Les erreurs posent donc également des problèmes de sécurité.

Les erreurs peuvent être séparées en deux familles :

– La première correspond aux erreurs populairement appelées *bugs*[2]. Le plus souvent, elles correspondent à l'arrêt de l'exécution du programme et à l'affichage d'un message plus ou moins cryptique pour le non-informaticien (et parfois même pour

2. Par un abus de langage qui nous espérons ne choquera personne, nous emploierons cet anglicisme.

l'informaticien) tel que « segmentation fault » ou « erreur fatale » par exemple.
- La seconde famille, beaucoup plus difficile à diagnostiquer (en général) peut être décrite par le slogan : « Mais, il ne fait pas ce que je veux ! ». On parle alors plutôt d'*erreurs de spécification*, le programme ne réalise pas tout à fait la tâche pour laquelle il était prévu. En fait, une fonctionnalité d'un logiciel peut être vue comme un *composant* réalisant un algorithme et communicant avec un contexte. Les erreurs de spécification sont souvent causées, soit par un algorithme erroné ou mal adapté, soit par une mésentente sur ce que le composant logiciel devrait échanger avec son contexte. Déceler et corriger cette forme d'erreur est une tâche ardue et, encore pour un long moment, le rôle du spécialiste. Malgré les nombreux progrès du domaine, l'automatisation de la certification de la concordance entre une spécification et un programme reste hors de portée du fait de la complexité des preuves nécessaires.

Notons que ces dernières années, les méthodes formelles rencontrent un certain succès dans ce domaine en proposant des outils d'aide interactifs à la construction et à la vérification des preuves de correction. Cependant, même une fois la concordance prouvée, le logiciel peut encore contenir des *bugs*. En effet, le *bug* est souvent beaucoup plus sournois car il est généralement causé par une erreur d'inattention du programmeur qui n'a pas pensé à traiter une valeur particulière d'une donnée du programme. Par exemple, la division est une opération simple à réaliser mais qui pose des problèmes puisqu'une division par zéro n'est pas définie. Ainsi, il est possible qu'un *bug* mette des années à être découvert (tant que personne ne demande de division par zéro par exemple). Les travaux de recherche que nous menons visent à détecter automatiquement le maximum d'erreurs potentielles de cette forme, que nous désignerons, dans le reste de cette thèse, par le terme d'erreur.

Les arguments précédemment évoqués montrent que la certification des logiciels est indispensable car elle ne peut plus reposer uniquement sur un développement avec des tests *a posteriori*. Toutes les phases des processus de construction d'une application devront être certifiées. De plus, la taille des applications impose une automatisation de cette certification, c'est-à-dire l'utilisation de logiciels qui vérifient que le résultat du développement en cours possède certaines propriétés [3]. Les métiers du logiciel doivent donc impérativement passer de l'ère de l'artisanat (traitement au cas par cas) à celle de l'industrie en faisant largement appel à l'automatisation des processus de production. Or, jusqu'à une époque récente, les nombreux outils développés dans ce but sont restés cantonnés au monde des logiciels centralisés et ont été peu utilisés dans les développements industriels. Cependant, certains domaines spécifiques tel que celui de la programmation synchrone intègrent de plus en plus d'outils automatiques de certification. Par exemple, les nombreux travaux réalisés sur LUSTRE (pour une description du langage voir [HCRP91] ou le tutoriel [Hal93]) et ESTEREL ([Ber00] présente le langage) ont permis de construire des ateliers de développement pour ces langages intégrant généralement des outils de vérification automatique :
- LESAR ([HLR92]) vérifie des propriétés logiques sur des programmes LUSTRE ;
- LURETTE ([RWNH98]) génère automatiquement des jeux de tests pour les pro-

3. Principalement des propriétés de correction du type : « aucun événement imprévu ne peut perturber tel composant de tel logiciel »

grammes LUSTRE ;
- XEVE[4] est un environnement graphique qui analyse et vérifie les programmes ESTEREL en les modélisant par des automates à états finis (« Finite State Machines » ou FSM) ;
- SCADE[5] est un atelier de développement de système temps réel critique commercialisée par TELELOGIC et exploité dans le domaine de l'aéronautique.

Plus généralement, de nombreux travaux ces dernières années tentent d'améliorer la puissance et la convivialité de ces outils et de les étendre au monde des réseaux et de l'*Internet*.

Dans le cadre séquentiel, différentes techniques ont été mises au point pour diminuer le nombre d'erreurs de programmation depuis quasiment les débuts de l'informatique. Le problème général de la recherche d'erreurs est *indécidable* mais le coût et les risques dus aux erreurs poussent à leur traque. L'idéal pour découvrir les erreurs d'un programme serait d'exécuter ce programme sur toutes ses entrées possibles. Or, ceci n'est clairement pas possible, la technique généralement employée (dans tous les domaines) est alors de construire une approximation du programme et de ses entrées. À partir de cette approximation, il est alors possible de simuler l'exécution du programme.

Cependant, dans la plupart des langages de programmation généraux, certains programmes (corrects) sont trop complexes (il s'agit ici de la complexité de la preuve de leur correction, pas de la complexité du programme lui-même) pour que l'approximation soit suffisante à certifier leur correction. Le compromis alors réalisé par les logiciels de traque d'erreur est de refuser un sur-ensemble de l'ensemble des programmes erronés. Ainsi, le système rejette les programmes qu'il ne parvient pas à certifier même si ceux-ci ne contiennent aucune erreur.

Il existe également une autre approche qui remporte de plus en plus de succès, elle consiste à limiter l'expressivité du langage utilisé pour diminuer le nombre de programmes complexes et d'automatiser certaines tâches que le programmeur devait réaliser avant. Par exemple, dans le cadre des langages généralistes, le langage JAVA est issu d'un tel processus de simplification :
- l'héritage multiple de C++ et d'autres langages à objets n'existe pas ;
- la redéfinition se fait sans modifier le type des arguments ;
- la mémoire est gérée automatiquement par un ramasse-miettes.

Ainsi, on diminue les sources d'erreurs possibles et on rend possible certaines analyses automatiques qui ne l'étaient pas. Notons que ces automatisations se font souvent au prix d'une diminution des performances, mais la contrepartie en terme de qualité du logiciel et de facilité de maintenance contrebalance largement ce défaut pour les applications pour lesquelles la rapidité n'est pas primordiale.

L'autre stratégie de simplification remportant un certain succès est la conception d'un petit langage très spécifique (« Domain Specific Language ») totalement dédié au domaine des applications que l'on veut programmer. Par exemple, les langages Python ou PERL sont utilisés pour la réalisation de script, TELESCRIPT est utilisé par Lucent pour la gestion de leurs équipements réseaux, PHP permet de rendre les pages web dynamiques ...

4. `http://www-sop.inria.fr/meije/verification/Xeve`
5. `http://www.telelogic.se/SCADE/default.asp`

Les calculs de processus

Les recherches dans le domaine de la détection d'erreurs sont généralement basées sur des modélisations mathématiques des programmes réels. Historiquement, le premier des formalismes utilisés a été le λ-calcul, il a permis une meilleure compréhension de la notion de fonction, ce qui a conduit à mieux les approximer dans le cadre des programmes séquentiels. Maintenant, les recherches portent plus sur les problèmes posés par la communication et la mobilité. Pour cela, la plupart des travaux sont menés dans le cadre de formalismes que l'on nommera génériquement *calculs de processus*. Ces calculs reposent sur une syntaxe algébrique relativement simple et concise. Leur expressivité et leur sémantique sont plus ou moins sophistiquées et dépendent souvent de l'objectif de l'étude (analyse de la communication, analyse des interblocages, étude des *plantages* de machines, vérification de propriétés de sécurité, ...).

Les premiers calculs de processus sont CCS (*Calculus of Communicating Systems*, voir [Mil80, Mil89]) et CSP (*Communicating Sequential Processes*, voir [Hoa85]). Ces deux calculs permettent une étude de la communication qui ne peut être que partielle puisqu'ils fixent statiquement la topologie de communication rendant ainsi impossible l'écriture de termes décrivant des systèmes dynamiques ou mobiles. Cette limitation a conduit à la définition par R. Milner du π-calcul ([MPW92, Mil99]) qui permet la communication de noms (qui combinée à l'extrusion rend donc la topologie de communication dynamique). Ce calcul repose sur la notion de canal de communication (synchrone) décrit par un nom et transportant des noms (représentant des canaux). Ce calcul minimal a servi de base a de nombreuses extensions, par exemple :

- les canaux de communication peuvent devenir polyadiques, ce qui introduit la possibilité d'erreurs de communication. En effet, si un processus envoie un message d'arité 1 sur un canal d'arité 2, l'exécution conduit à une erreur. R. Milner introduit une abstraction des canaux par leur arité dans le cadre d'une *discipline de sorte* pour détecter ces erreurs (on peut consulter [Mil99] pour plus de détails) ;
- dans le π-calcul d'ordre supérieur (HOπ), il est possible de communiquer des processus à travers les canaux. Cependant, dans sa thèse ([San92]), D. Sangiorgi montre que cette extension n'augmente pas le pouvoir expressif du π-calcul (même si certains termes s'écrivent plus simplement) ;
- dans [Bou92], G. Boudol, définit un π-calcul dans lequel la communication est asynchrone ;
- le spi calcul ([AG97a, AG97b]) ajoute au π-calcul des primitives cryptographiques pour permettre l'étude des protocoles de communications cryptées ;
- et plus récemment, une notion de site et de répartition est ajouté pour permettre l'étude de la mobilité (voir par exemple, le π_{1l} [Ama97], le dpi [Sew98a] ou le Dπ [RH99]).

Pour de plus amples analyses des différents π-calcul et de leur sémantique, nous renvoyons le lecteur à la bibliographie commentée de S. Dal Zilio [DZ00] et au tutoriel de P. Sewell [Sew00].

Une certaine unanimité s'est faite dans le monde de l'informatique pour la modélisation de la notion de fonction par le λ-calcul. Mais le modèle équivalent n'a pas encore été mis au point pour la concurrence et la répartition. La modélisation de la concurrence semble,

en effet, beaucoup plus complexe. Même si le π-calcul propose un modèle qui est devenu très populaire au sein de la communauté des chercheurs, il existe de nombreuses autres approches. Ainsi, divers calculs plus spécifiques sont également utilisés, ils sont basés sur des concepts proches du π-calcul mais sont légèrement différents aussi bien du point de vue de la syntaxe que de la sémantique :

- le ν-calcul (voir [HT91]), un calcul asynchrone créé à la même époque que le π-calcul ;
- Cap ([Col97]), un calcul de processus modélisant le modèle d'acteurs conçu au sein de l'équipe de recherche qui m'a accueilli ;
- le calcul bleu ([Bou97a]) qui peut être vu comme une intégration du π-calcul (pour décrire les interactions concurrentes) et du λ-calcul (pour décrire le comportement fonctionnel des processus) ;
- le Join calcul[6] ([FG96, FGL+96]), qui se distingue du π-calcul par une construction unique (les « *join-patterns* ») qui permet de décrire la restriction, la réception et la réplication et une hiérarchie de noms (les *locations*) ;
- Tyco[7] ([Vas99]) est une forme de π-calcul asynchrone s'inspirant des objets. Les messages y sont étiquetés et leur réception utilise cette étiquette pour déterminer la suite du calcul ;
- la CHAM ([BB92]), le fusion calcul ([PV98, Vic98]) et l'action calcul ([Mil93, Mil96]) sont trois calculs plus abstraits destinés à l'étude de la notion d'interaction ;
- seal ([VC99]) et le calcul des ambients (voir par exemple [CW85, Car99] ou le site web qui leur est consacré[8]) sont consacrés à l'étude de la sécurité, ils ajoutent à un calcul de processus de forme minimale une notion d'enveloppes (éventuellement hiérarchiques) qui contiennent des processus et peuvent se déplacer, s'ouvrir, ...

Dans le cadre de ces différents calculs, la recherche d'erreur peut être faite par diverses techniques que nous allons présenter maintenant. Nous allons débuter et insister par le typage puisque c'est l'approche que nous avons choisie.

L'approximation par les types

Le test exhaustif n'étant pas possible, il est nécessaire d'approximer l'exécution du programme pour obtenir des informations pertinentes sur sa correction. Pour cela, il faut déterminer des *collections* de valeurs ayant des comportements semblables et choisir dans chaque collection un représentant (qui n'est pas forcément élément de la collection) qui remplace alors toutes les occurrences des valeurs de sa collection. Ces collections sont appelées *types*, le sens commun employant plutôt le terme de type pour désigner le représentant d'une collection. Ainsi, par exemple, tous les réels ont des propriétés très proches et sont souvent regroupés dans le type *float*. La relation qui permet alors de passer d'une valeur à son type est appelée typage (c'est en fait, une approximation). Sa précision peut varier selon les besoins de son utilisateur. Par exemple, il est aussi possible d'approximer les réels strictement positifs par *posfloatnz*, ceux strictement négatifs par *negfloatnz* et 0 par *zéro*. Les entiers sont alors partitionnés dans trois types. Usuellement, le type d'une

6. `http://pauillac.inria.fr/join`
7. `http://www.di.fc.ul.pt/~vv/tyco.html`
8. `http://go.163.com/~mobileambient`

fonction est un couple, noté $t_a \to t_r$, composé du type de ses arguments possibles t_a et du type de ses résultats possibles t_r. La fonction $\lambda x.1/\sqrt{x}$ a dans la première approximation le type $\mathit{float} \to \mathit{float}$ et dans la seconde $\mathit{posfloatnz} \to \mathit{posfloatnz}$.

Le typage est ensuite étendu à toutes les constructions du langage de programmation. En général, le système de type est décrit par un ensemble de règles de typage dirigées par la syntaxe qui permettent de déduire le type d'une construction à partir du type de ses sous-constructions. Ainsi, typer un programme consiste à donner la preuve (sous forme de déduction) que certaines erreurs d'exécution ne peuvent se produire. Par exemple, la règle de typage généralement associée à l'application de fonction a la forme suivante :

$$\textsc{App:}\ \frac{\mathcal{E} \vdash_e f : t \to t' \quad \mathcal{E} \vdash_e e : t}{\mathcal{E} \vdash_e f\ e : t'}$$

Elle exprime le fait que si l'argument e d'une fonction f de type $t \to t'$ est du type attendu t alors l'application est *légale* et son résultat est de type t'. Dans cette explication, le sens du mot *légale* signifie qu'il ne conduira pas à certaines erreurs d'exécution et dépend fortement de la précision du système de type. Ainsi, par exemple, dans le cadre du premier système, l'application $(\lambda x.1/\sqrt{x})\ (-1.1)$ est déclarée correcte (puisque -1.1 est bien de type *float*). Mais, dans le cadre du second système, elle est déclarée incorrecte puisque -1.1 est de type *negfloatnz* qui n'est pas compatible avec le type du paramètre de la fonction.

Un système de type est caractérisé par ses types et ses règles de typage. Tous les systèmes de type possèdent une propriété commune : la *correction*. Ils garantissent ainsi que si un programme est bien typé alors il ne produira pas certaines erreurs lors de son exécution. Ces erreurs qui sont détectées sont donc une troisième caractéristique d'un système de type. Cette propriété doit être énoncée et démontrée à partir d'une modélisation de la notion d'exécution fournie par une sémantique du langage de programmation utilisé. Souvent, cette démonstration repose sur une sémantique dite opérationnelle qui décrit l'exécution du programme par une relation de réduction. Elle consiste, alors, à montrer que la relation de réduction est stable vis-à-vis du typage (*i.e.*, un programme typé se réduit forcément en un programme typé). Les erreurs que l'on cherche à éliminer n'étant pas typées, un programme typé ne peut se réduire en une erreur.

L'approche présentée ci-dessus se rapporte au typage dit *statique* et *fort*.

Le typage est dit statique par opposition au typage *dynamique*. Le premier, comme nous l'avons vu, vise à montrer la correction du programme sans l'exécuter. Alors, que le second, vérifie au cours de l'exécution avant chaque calcul que celui-ci ne conduira pas à une erreur. Pour cela, chaque entité du programme conserve son type. Le typage dynamique est plus flexible que le typage fort qui est souvent plus restrictif sur les possibilités de programmation. En contrepartie, la détection des erreurs repose sur l'exécution du programme avec tous les problèmes soulevés dans le début de cette introduction. Cette technique ne supprime donc pas les erreurs fatales mais permet de terminer un programme avant son « plantage » par un message clair permettant souvent de déterminer l'origine de l'erreur dans le programme source. Il est aussi également possible de capturer ces événements exceptionnels pour spécifier un comportement différent (reprise par exemple). Remarquons que la vérification de type dynamique augmente le temps de calcul du programme.

Le typage statique fort est plus restrictif que le typage dynamique du fait de l'approximation qu'il doit faire de l'exécution. En effet, il doit souvent travailler sur une version simplifiée de l'état d'exécution et sur l'historique du calcul pour des problèmes de décidabilité ou d'efficacité. Ainsi, il doit rejeter certains programmes corrects dont il n'a pas pu prouver la correction. Ces programmes corrects que le programmeur ne peut réaliser sont le prix à payer pour la garantie de correction. Le choix entre l'un ou l'autre des typages dépend donc de l'intérêt qu'a le programmeur pour ces programmes corrects rejetés. Notons, que le typage dynamique est plus adapté (ou tout du moins préféré) dans certains domaines de l'informatique, dans lesquels la flexibilité est primordiale. Par exemple, les domaines de programmation fortement répartie où les composants sont fortement hétérogènes sont difficiles à programmer dans les langages typés statiquement. Le langage ERLANG sur lequel nous avons travaillé est un exemple de langage typé dynamiquement actuel. Mais, on peut également citer la plupart des descendants de LISP ou de Prolog.

Le terme de typage *fort* vient de la propriété de correction exigée pour le système de type. Un système de type qui n'est pas fort ne garantit pas l'absence d'erreurs lors de l'exécution d'un programme bien typé. En général, cette approche est combinée avec une vérification dynamique qui sera chargée de combler les lacunes du système.

Des approches intermédiaires sont également possibles. Ainsi, R. CARTWRIGHT et M. FAGAN introduisent dans [CF91] la notion de typage *souple*. Cette forme de typage essaye de vérifier statiquement la correction de la plus grande partie du programme et détermine les portions qui nécessiteront des tests de type dynamiques. Le programme gagne en efficacité puisque certains tests dynamiques peuvent être supprimés. Par cette technique, aucun programme n'est rejeté par le compilateur. Pour chacune des constructions qu'il pense problématique, le compilateur génère des avertissements pour le programmeur. Celui-ci doit donc alors déterminer, si l'avertissement est dû à la faiblesse du système de type statique ou à une réelle erreur. Pour obtenir un système de type plus performant, M. WIDERA et C. BEIERLE ont introduit le typage *complet* dans [WB99]. Dans un tel système, un programme est statiquement typé, si le compilateur prouve qu'il conduit forcément à une erreur, il est rejeté et dans le cas contraire, il est accepté et un système de type dynamique prendra le relais à l'exécution.

Remarquons que dans la plupart des langages réels, le typage n'est jamais complètement fort. Par exemple, en OBJECTIVE CAML, le programme suivant :

```
let hd = function h::t -> h;;
```

est correctement typé (son type est `'a list -> 'a`) même si un avertissement nous signale que la fonction `hd` échouera si elle est appelée avec une liste vide comme paramètre. Ainsi, `hd []` conduit à une erreur d'exécution. Un système de type est donc fort ou correct par rapport à une classe d'erreurs fixée à l'avance.

Le typage d'un programme est une forme de preuve de correction suffisamment simple pour être automatisée. Deux stratégies sont alors possibles : la *vérification* et l'*inférence*.

Dans le cadre de la vérification, le programme contient suffisamment d'informations sur le type de ses différentes entités. Le compilateur se contente de vérifier que les contraintes d'utilisation des données sont vérifiées et donc que les types fournis par le programmeur sont corrects. Cette vérification est la plupart du temps décidable et automatisable de

manière relativement efficace. Cependant, il existe des systèmes de type pour lesquels la vérification n'est pas possible.

L'inférence consiste à augmenter le degré d'automatisation puisque le compilateur va lui-même calculer la plupart des informations de type. Le programmeur n'est plus alors obligé de donner ces informations, le confort de programmation est donc augmenté. En revanche, l'inférence est assez souvent indécidable ou nécessite des calculs qui sont très coûteux. Ce processus n'est donc pas toujours rentable par rapport à la vérification. Remarquons que l'inférence permet également de supprimer une source d'erreur dans les programmes. En effet, si les types sont complexes, le programmeur peut se tromper dans leur écriture. Dans le cadre du typage de la communication, les types des processus sont souvent si complexes dans le but d'être suffisamment expressifs qu'il ne nous semble pas raisonnable de demander au programmeur de les donner. Ainsi, dans cette thèse nous nous penchons uniquement sur des techniques d'inférence.

Remarquons que les compilateurs utilisant l'inférence offrent souvent la possibilité au programmeur d'indiquer certains types, s'il le souhaite. L'inférence se combine alors avec de la vérification.

Pour augmenter la précision des systèmes de type et donc diminuer le nombre de programmes corrects que le système rejette, il est possible de conférer plusieurs types à certaines entités du programme. En effet, dans un cadre conférant un unique type à chaque donnée, la fonction `hd` précédemment définie aurait son type contraint par sa première utilisation. Par exemple, le programme suivant est rejeté :

```
let hd = function h::t -> h in hd [1]; hd ['a'];;
```

car la première application confère à `hd` le type $\mathit{int\ list} \to \mathit{int}$ qui n'est pas compatible avec *char*. Or, il parait intuitif que la fonction `hd` pourrait être appliquée à d'autres listes que celles qui contiennent des entiers. Pour cela, on construit des systèmes de type dit *polymorphes*. Diverses catégories de polymorphismes existent et nous renvoyons le lecteur intéressé à [CW85] qui les analyse. Parmi ces formes de polymorphisme, deux ont eu un succès relativement important : le polymorphisme *paramétrique* et le polymorphisme *d'inclusion*.

Le polymorphisme paramétrique (qui prend souvent le nom de polymorphisme dans le cadre fonctionnel) exploite la quantification universelle des variables du type d'une entité pour permettre son utilisation dans des contextes différents. La fonction `hd` se voit associer le type $\forall\alpha.\alpha\ \mathit{list} \to \alpha$ et le programme ci-dessus devient correctement typé. Pour conserver la possibilité de reconstruire les types par un processus d'inférence, la position et l'introduction de cette quantification doit être limitée :

- les quantificateurs sont au plus haut niveau du terme représentant le type (par exemple, le type suivant n'est pas légal : $\forall\alpha.\alpha \to \forall\beta.\beta$) ;
- la généralisation a parfois lieu uniquement lors de la définition d'un nom (pour ML, au moment d'un `let`), on parle alors de *let-polymorphisme*.

Le polymorphisme d'inclusion (appelé polymorphisme dans le cadre des langages à objets) repose sur une relation d'ordre sur les types. Il est aussi appelé *sous-typage*. Il permet d'affaiblir le type d'une entité en ajoutant au système de type une règle généralement

appelée « subsumption » qui a la forme suivante :

$$\frac{\mathcal{E} \vdash_e e : t_1 \quad t_1 \sqsubseteq t_2}{\mathcal{E} \vdash_e e : t_2}$$

Il est alors possible d'utiliser une donnée simple dans un cadre où on attendait une donnée plus complexe sans devoir ajouter des conversions de type. Par exemple, l'application $f((\lambda x.1/\sqrt{x})\ 1.1)$ où f est de type $\mathit{float} \to \mathit{float}$ est légale si le type *posfloatnz* est déclarée sous-type de *float*.

Le sous-typage a été proposé indépendamment dans de nombreux contextes (voir par exemple L. Cardelli dans [Car85] qui l'introduit dans le contexte des objets et de l'héritage) mais il semble fondamental dans l'étude des enregistrements, des objets ou des acteurs.

Une relation de sous-typage définie sur les types de base induit une relation d'ordre sur les types construits. Cette extension peut suivre trois règles :

- la *covariance* si $C(t) \sqsubseteq C(t') \Leftrightarrow t \sqsubseteq t'$,
- la *contravariance* si $C(t) \sqsubseteq C(t') \Leftrightarrow t' \sqsubseteq t$ et
- l'*invariance* si $C(t) \sqsubseteq C(t') \Leftrightarrow t = t'$.

La plupart des constructeurs de type sont covariants. Par exemple, les constructeurs de données (tuples, listes, ...) sont covariants par rapport à toutes leurs composantes. Le constructeur de fonction, noté $\to$ est contravariant par rapport à son domaine. Par exemple, la fonction f peut déclarer qu'elle n'accepte que des réels positifs non-nuls en adoptant le type $\mathit{posfloatnz} \to \mathit{float}$ qui est sous-type de $\mathit{float} \to \mathit{float}$. Par contre, il est covariant par rapport à son codomaine, puisque si la fonction renvoie un réel positif non-nul, c'est un réel. On a donc :

$$t'_0 \sqsubseteq t_0 \wedge t_1 \sqsubseteq t'_1 \Longleftrightarrow t_0 \to t_1 \sqsubseteq t'_0 \to t'_1$$

Si le sous-typage reste cantonné à définir un ensemble de relations entre les types de base et à leur extension aux types construits, il est qualifié de *sous-typage atomique*. Cependant, cette forme de sous-typage assez limitée ne suffit pas dans le cadre des objets (ou des acteurs). Nous souhaitons pouvoir typer une fonction f de la forme : $\lambda o.o\#m$, qui prend en paramètre un objet o et utilise sa méthode m. Si nous disposons d'un objet o qui possède uniquement la méthode m, son type peut être décrit par $\{m : \mathit{int}\}$. Il est clair que cette fonction peut être appliquée à o sans risquer de provoquer une erreur à l'exécution. Nous pouvons donc attribuer à la fonction f le type $\{m : \mathit{int}\} \to \mathit{unit}$. Mais si nous disposons d'un objet o' qui possède les méthodes m et p, nous pouvons également appliquer f à o'. Il est alors nécessaire d'imposer $\{m : \mathit{int},\ p : \mathit{int}\} \sqsubseteq \{m : \mathit{int}\}$. Le sous-typage est ainsi utilisé pour exhiber uniquement les parties communes de deux entités, afin de les manipuler ensembles.

L'inférence de type en présence du sous-typage devient plus problématique. Diverses stratégies sont possibles mais elles reposent toutes sur une idée commune, chaque nœud du programme se voit attribuer une variable de type et chaque règle de typage génère des contraintes entre ces variables. Par exemple, la règle de typage de l'application devient :

$$\textsc{App:}\ \frac{\mathcal{E} \vdash_e f : t_f \quad \mathcal{E} \vdash_e e : t_e}{\mathcal{E} \vdash_e f\ e : t} \left(t_f \sqsubseteq t_e \to t \right)$$

Le typeur collecte un ensemble de contraintes qui, n'est soluble que si le programme est correctement typé. Dans le cadre du sous-typage, les contraintes sont en général des inclusions. Mais, l'algorithme d'unification du typeur de ML peut également être vu comme un algorithme de collecte et de résolution de contraintes d'égalité.

D'autres méthodes d'analyse

Nous avons restreint jusqu'à présent la notion d'analyse statique à la détection d'erreurs. Leur cadre est cependant plus général puisqu'il permet sans exécuter un programme, mais en en utilisant une approximation, de déterminer certaines propriétés intéressantes de ce programme. Par exemple, les résultats de ces analyses permettent souvent d'améliorer la compilation du programme. Les analyses de flots sont par exemple utilisées pour déterminer la forme de stockage d'une variable (registre, pile, ...).

D'autres méthodes d'approximation des programmes existent. Par exemple, le cadre très général de l'interprétation abstraite présenté dans [CC77] permet de construire de nombreuses analyses. Cette construction repose sur la définition d'un domaine abstrait de valeurs et d'une sémantique abstraite qui doivent être une abstraction du domaine réel et de la sémantique concrète. Si cette construction a respecté certaines règles, l'exécution du programme sera approchée par l'exécution de son abstraction dans le cadre de la sémantique abstraite. Cette exécution consiste, en général, en un calcul de point fixe. Ce calcul étant généralement très long ou même parfois divergeant, le cadre défini par P. Cousot permet d'utiliser des opérateurs qui permettent d'accélérer la convergence qui si elle ne fournit pas les points fixes exacts en donne une approximation sûre.

L'interprétation abstraite fournit un cadre très général dans lequel un certain nombre d'analyses statiques existantes peuvent être reformulées. Par exemple, B. Monsuez dans sa thèse fournit une reformulation du typage de ML. L'analyse statique par résolution de contraintes mise au point par A. Aiken et E. Wimmers ([AW93]) ainsi que d'autres approches sont ramenées à une interprétation abstraite dans [CC95]. Mais, elle a aussi été utilisée pour analyser des programmes concurrents, par exemple, A. Venet dans [Ven97, Ven98] réalise une interprétation abstraite du π-calcul qui lui permet de reconstruire des informations précises sur la topologie d'un système décrit par un terme du π-calcul. Cependant, la construction d'une interprétation abstraite est souvent très complexe. En effet, l'automatisation de la correction est obtenue par la construction d'une abstraction qui est bien plus difficile que celle effectuée lors du typage. Donc, la difficulté rencontrée au niveau de la preuve de correction dans le cadre du typage n'a pas disparu mais a été déplacée et il n'est pas évident de déterminer quelle méthode est la plus simple d'utilisation. Il est d'ailleurs probable que cela dépende de l'application que l'on souhaite réaliser.

Parmi les autres méthodes pour détecter des erreurs dans les programmes, le *model checking* remporte un certain succès du fait de son fort degré d'automatisation. En effet, le programme que l'on souhaite vérifier est modélisé par un graphe (en général un automate) puis les propriétés logiques voulues sont vérifiées par un test exhaustif sur ce graphe. Cette vérification n'est possible que dans le cas de systèmes à états finis et même dans ce cas le test peut être très long. Cette technique a fait ses preuves surtout dans le domaine

de la vérification de matériel électronique. Par exemple, l'outil SMV [9] a servi à valider le protocole du bus PCI.

Si le graphe du test est très grand ou infini, le *model checking* se combine bien avec une interprétation abstraite pour diminuer la taille de ce graphe. Cette approche est utilisée initialement par R. Cridlig dans [Cri95] et [Cri96] pour abstraire un système concurrent en un système de transitions fini sur lequel il peut ensuite utiliser le *model checking*. Cette combinaison est présentée de manière précise et pour de nombreuses applications dans [SS98]. Cette technique a été utilisée, dans le cadre d'Erlang, dans [Dam96] et [Huc99] pour prouver des propriétés de logique temporelle.

Pour la vérification des programmes concurrents, ces diverses techniques d'approximation ont été utilisés pour prouver des propriétés de logique temporelle, en général, dans le cadre du *model checking* (voir par exemple, [VM94, Dam96, DåF98, FGM+98, Huc99, Kön99]). Mais, c'est surtout le typage qui remporte un franc succès quelque soit le calcul de processus, de nombreux systèmes de type existent. Par exemple, pour les ambients [CGG99, Zim00, CG99], pour le π-calcul [PRT93, KPT96, PS96, Yos96, Sew98b, YH99], pour le calcul bleu [Bou97b, DZ99], pour le Join calcul [FLMR97], pour Tyco [Rav00], pour Cap [Col97] ou dans le cadre d'autres calculs [Kob97, RH98, NNS99b, Nim99, RH99]).

Nos travaux sont basés sur ces différentes études et surtout sur le typage de Cap dans [Col97]. Ces travaux, menés par l'équipe qui m'a accueilli, ont permis de construire plusieurs systèmes de type pour détecter les erreurs de communication dans le cadre des acteurs. Les types utilisés sont similaires aux types des objets et vérifient donc une relation de sous-typage. Ils sont calculés par l'inférence de contraintes d'inclusion sur un terme Cap, qui sont résolues par un algorithme de résolution de contraintes ensemblistes.

Dans cette thèse, nous cherchons à réaliser ces vérifications statiques dans le cadre de *vrais* langages de programmation. Les systèmes de type que nous proposons dans cette thèse doivent donc combiner des concepts issus de différents domaines du typage. Par exemple, dans le cadre d'Erlang, nous utilisons un calcul d'effets ([TJ94]) pour collecter les messages que comprend un processus.

Apport de ma thèse

Nous proposons une modélisation des acteurs relativement générale qui permet de construire facilement la sémantique des langages d'acteurs (ou des langages à objets concurrents ou des applications clients/serveurs). Nous utilisons ce formalisme (les *configurations*) pour exprimer les sémantiques de ML-Act et Erlang.

Le typage des langages séquentiels est bien maîtrisé. La concurrence est analysée par typage dans le cadre des calculs de processus. Par contre, la combinaison de l'analyse de ces deux aspects sur de vrais langages est difficile et peu abordée.

En partant d'un système de type construit dans le cadre très simple d'un calcul de processus (Cap), nous construisons et prouvons la correction de deux systèmes de type sophistiqués. Ces systèmes sont construits dans le but de détecter autant de messages dangereux (qui risque de provoquer des erreurs) que possible dans une application concur-

9. `http://www.cs.cmu.edu/~modelcheck`

rente. Nous définissons donc précisément la forme d'erreur que nous traquons dans les programmes ML-Act et Erlang. Les types ont une forme inspirée de celle des types enregistrements utilisés dans d'autres typages similaires ([FLMR97], [Rav00], [DZ99] ou [Sew98b]), leur originalité est la collecte simultanée des messages compris par un acteur et des messages qu'il reçoit. Ainsi, il est possible de vérifier leur compréhension, mais également, de permettre dans des extensions du système de compter les messages pour détecter des messages envoyés en trop.

Le passage à l'échelle de *vrais* langages a nécessité la construction d'un système de résolution plus complexe que celui utilisé pour Cap. Pour cela, nous nous sommes inspirés des travaux utilisant des graphes de contraintes (voir par exemple [Pot98]).

Tout au long de cette thèse, nous avons essayé de conserver une approche la plus générale possible pour rendre exploitable toutes nos techniques dans le cadre de langages reposant sur des principes similaires.

Plan

Ce manuscrit est divisé en trois parties. La première introduit les concepts fondamentaux utilisés dans nos travaux. La deuxième est consacrée à la présentation de ML-Act, de sa sémantique et de son système de type. Et, la troisième introduit Erlang, sa sémantique et son typage. L'ordre des parties est important puisque, les bases du système sont introduites dans le quatrième chapitre. L'instanciation dans le cadre de ML-Act et dans celui d'Erlang, n'insiste que sur leurs spécificités et les adaptations nécéssaires de ce cadre général.

Première partie

La première partie se subdivise en quatre chapitres : un premier sur les acteurs et leur modélisation, un deuxième sur les erreurs que nous souhaitons détecter et un troisième sur une introduction aux mécanismes d'inférence et de résolution des contraintes utilisés par nos systèmes de type.

Le chapitre 1 introduit le paradigme de programmation des acteurs en se basant sur le modèle défini par G. Agha dans [Agh86]. Un acteur est une entité réactive qui possède une boîte aux lettres et communique de manière asynchrone avec les autres acteurs de l'application. Ces acteurs sont comparés aux objets et nous présentons succinctement les langages de programmation qui sont inspirés des acteurs. Nous introduisons ensuite dans une perspective historique notre modélisation des acteurs : les *configurations*. Ce formalisme est à rapprocher des calculs de processus par son écriture algébrique. Il est en fait une simplification du calcul d'acteurs Cap conçu au sein de l'équipe de recherche qui m'a accueilli (voir par exemple [Col97]). Ensuite, nous poursuivons ce chapitre par la définition de la sémantique opérationnelle des configurations à l'aide d'une réduction dite *concurrente*. Cette définition laissera en suspens la définition précise de certaines notions (par exemple la gestion des boîtes aux lettres) jusqu'à son application à l'un des langages introduits par la suite. Enfin, ce chapitre se conclut sur une brève discussion des différentes formes de modélisation de la sémantique des acteurs et les avantages des configurations.

Le deuxième chapitre concerne l'analyse et la définition des erreurs de communication que nous souhaitons détecter statiquement au sein des programmes d'acteurs. En partant d'une définition informelle et intuitive de la notion de « *message not understood* » classique dans le monde des objets, nous raffinons le concept pour obtenir une classification formelle détaillée des messages d'un programme. Plusieurs classes de messages *dangereux* sont ainsi définies, dont la principale est celle des messages orphelins de sûreté qui sont l'objet de nos analyses statiques. À partir des définitions de ce chapitre, nous modifions légèrement la sémantique introduite dans le chapitre 1 pour prendre en compte la détection dynamique des messages orphelins par le biais d'une notion de potentiel approximant l'ensemble des messages compris par un acteur. Enfin, nous concluons ce chapitre en précisant le comportement que nous souhaitons avoir face aux différentes classes de messages.

Dans le troisième chapitre, nous introduisons la notion de typage. Pour cela, nous donnons une définition abstraite des types que nous utilisons dans la suite. Nous présentons alors le sous-typage sur cette forme de type ainsi que le mécanisme d'inférence sur lequel sont basés nos systèmes de type. Ensuite, nous évoquons rapidement le mécanisme de résolution des contraintes résultant de l'inférence suivant un modèle théorique issu des travaux de A. Aiken et E. Wimmers (voir par exemple [AW93]) mais techniquement réalisé en s'inspirant des travaux de F. Pottier ([Pot98]). Nous concluons ce chapitre en présentant rapidement l'historique de nos types et de nos solveurs.

Le quatrième et dernier chapitre de cette première partie décrit une première instanciation du système. Ainsi, en se limitant à un λ-calcul calcul tout simple ($\mathcal{F}unc_0$) pour la partie fonctionnelle, il est possible de présenter le principe de nos systèmes de type plus facilement.

Deuxième partie

La deuxième partie se subdivise en deux chapitres : un premier introduit le langage ML-Act ainsi que sa sémantique basée sur celle des configurations et un second définit précisément son système de type et contient la preuve de la correction du système de type vis-à-vis des messages orphelins.

Le chapitre 5 débute par une présentation de ML-Act et de sa syntaxe. Ensuite, nous définissons le micro langage fonctionnel $\mathcal{F}unc_1$ qui exprimera les calculs fonctionnels exécutés dans les configurations. Puis, nous précisons la traduction d'un programme ML-Act en un terme des configurations. Ceci permet en introduisant les points laissés en suspens dans le premier chapitre de définir la sémantique complète de ML-Act. Nous présentons le fonctionnement de cette sémantique sur un exemple. Pour conclure ce chapitre, nous discutons des implantations actuelles de ML-Act.

Le sixième chapitre contient la définition du système de type de ML-Act. Il commence par la construction des règles de calcul du potentiel d'un acteur nécessaire à la détection des messages orphelins. Puis, nous définissons précisément les types et la relation de sous-typage dans le cadre de ML-Act. Nous insistons surtout sur les extensions nécessaires par rapport au chapitre 4. Ainsi, la preuve de la correction de notre système de type est très proche de celle du chapitre 4. Puis, nous terminons ce chapitre par des travaux récents que nous réalisons pour améliorer l'approximation que réalise nos types.

Troisième partie

La dernière partie de cette thèse est également composée de deux chapitres, un pour introduire ERLANG et un pour définir et prouver son système de type. Les notions introduites précisément dans la deuxième partie et similaires dans le cadre d'ERLANG ne seront pas réexpliquées en détail.

Dans le septième chapitre, nous présentons le langage ERLANG conçu par ERICSSON dans un cadre industriel. Ce langage fonctionnel concurrent distribué plutôt complexe est alors simplifié (pas de notion de module, pas de nœuds pour la répartition, pas d'exception, ...). Le langage obtenu dénoté μErlang est encore très complexe du fait de son caractère fortement dynamique issu de ses origines historiques (PROLOG et LISP). Ensuite, nous donnons la définition du pendant de $\mathcal{F}unc_1$ qui porte le nom de $\mathcal{F}unc_2$. Puis, le processus de traduction de code μErlang en une configuration est décrit en détail. La sémantique de μErlang est alors définie dans le cadre des configurations. Enfin, nous comparons notre sémantique formelle d'ERLANG aux autres travaux sur le sujet.

Le chapitre 8 suit le cheminement des chapitres 4 et 6 après une introduction précisant les différences majeures de sémantique entre ML-ACT et μErlang, et leur conséquence sur l'abstraction que nous réalisons. Nous présentons le calcul du potentiel et le treillis des types, ainsi que les règles de déduction du système de type. Puis, nous présentons les adaptations nécessaires de la phase de résolution des contraintes. Nous terminons par la preuve de correction en suivant un cheminement similaire à celui du chapitre 4. Nous concluons, enfin, par une analyse de notre système par rapport au système de type déjà réalisé pour ERLANG et un bilan de l'apport à nos systèmes de cette réalisation.

Conclusion et annexes

Nous terminons ce manuscrit par un bilan des travaux réalisés et esquissons quelques pistes d'extension et d'amélioration de nos techniques de typage.

Ce document contient deux annexes, une qui contient les grammaires complètes des langages ML-ACT et ERLANG et une seconde qui reprend toutes les définitions de cette thèse pour résumer les sémantiques des deux langages.

Première partie

Généralités

Chapitre 1

Les Acteurs

La première partie de ce chapitre présente le modèle d'acteurs. Nous introduisons intuitivement les bases de ce modèle en partant d'un point de vue historique. Puis, nous comparons acteurs et objets afin de mieux comprendre l'essence des acteurs et le caractère très général de ce modèle. Enfin, nous remettons en perspective le modèle d'acteurs en présentant une grande partie des langages concurrents et/ou distribués qui utilisent plus ou moins explicitement la notion d'acteur. La seconde partie du chapitre est consacrée à la définition de la notion de configuration. Cette notion proche d'un calcul de processus tel le π-calcul fournit un outil de modélisation qui nous permet dans la suite de cette thèse de décrire les sémantiques de ML-Act et d'Erlang. Enfin, en conclusion de ce chapitre, nous exposons quelques travaux déjà réalisés sur la sémantique des acteurs et nous les confrontons à notre modélisation.

La mise au point de la notion de configuration qui est introduite dans ce chapitre est le fruit de deux influences majeures : Cap (un calcul d'acteurs conçu par notre équipe dans le cadre de la thèse de J-L. Colaço [Col97] et plus généralement les calculs de processus) et la modélisation des langages de programmation ML-Act et Erlang. La partie abstraite qui décrit la structure des acteurs et le milieu dans lequel ils évoluent (et où transitent les messages), est fortement inspirée par Cap. Cependant, des travaux préliminaires sur l'utilisation de Cap dans le cadre de l'analyse d'un langage de programmation (voir par exemple [Dag97]), nous ont convaincu que son modèle était trop faible. En effet, pour traduire un programme dans Cap deux approches sont possibles : soit la partie fonctionnelle est codée en Cap, soit elle est traitée séparément. Cependant, aucune de ces approches ne satisfait les objectifs de nos travaux : l'analyse statique. D'une part, si le calcul fonctionnel est codé en Cap, les programmes sont totalement déstructurés et il n'est alors plus possible de réaliser des analyses suffisamment précises. D'autre part, la séparation entre calcul fonctionnel et action concurrente entraîne une forte limitation des possibilités de programmation. Le mélange concurrent et fonctionnel étant alors interdit, il n'est pas possible, par exemple, de stocker des adresses d'acteurs dans une liste. La première solution réalisée a été d'ajouter une notion de fonction (via un λ) à Cap qui a conduit à un mélange trop homogène où les analyses statiques deviennent relativement complexes (voir [Col98]). C'est ici qu'est alors intervenue la seconde influence, en effet, les travaux menés pour exprimer la sémantique et le typage de ML-Act et d'Erlang nous ont convaincu de maintenir une certaine séparation entre la partie fonctionnelle et la

partie concurrente. Cette séparation ne peut pas être étanche (sinon elle contraint trop le programmeur) mais elle doit limiter l'interaction des deux mondes à quelques primitives bien maîtrisées. Ainsi, il est possible de réutiliser au mieux les différentes sémantiques et les différentes techniques de typage déjà développées dans le cadre des langages fonctionnels. De plus, notre travail peut alors se concentrer sur cette interface et sur la partie concurrente.

Les seules opérations qui provoquent une interaction entre le calcul fonctionnel et le calcul concurrent sont les *méthodes fonctionnelles* venues du monde des acteurs (envoi de message, création d'acteurs et changement de comportement). On peut alors modéliser les acteurs par un formalisme assez simple (les configurations) et l'étendre ensuite par quelques opérateurs concurrents et un noyau de calcul à un langage d'acteurs.

1.1 Généralités

Le modèle des Acteurs

Le modèle des acteurs proposé par C. Hewitt [HBS73] et [Hew77] est l'un des premiers modèles de la programmation concurrente. Initialement construit pour modéliser la notion d'agent de l'intelligence artificielle distribuée[10], il a été développé indépendemment de tout langage de programmation. Il a ensuite évolué pour servir d'outil permettant la spécification et l'implantation aisée d'applications distribuées et parallèles. Il a inspiré la plupart des modèles de concurrence développés par la suite mais reste l'un des plus simples à utiliser pour le programmeur. En effet, ce modèle abstrait les problèmes de bas niveau liés à la répartition, tels que les protocoles de communication entre entités ou leurs synchronisations. Parmi les nombreux développements qu'a connus ce modèle, notre référence est celui présenté par G. Agha dans son livre [Agh86].

L'acteur, abstraction de base du modèle, est un *agent* autonome qui encapsule des données et des procédures sur le même principe que celui des objets. Tous les acteurs d'une application s'exécutent en parallèle, mais à la différence d'un processus, un acteur n'est pas une entité qui effectue un calcul puis disparaît. C'est un objet réactif, il exécute des calculs lors de la réception d'une requête (sous la forme d'un message) et suspend son activité à la fin de ses calculs jusqu'à la prochaine requête. À l'image du *mail* d'Internet, chaque acteur possède une *adresse postale* qui est utilisée comme identité. Cette adresse postale correspond à une unique queue de messages appelée *boîte aux lettres* dans laquelle sont stockés les messages reçus par cet acteur. La boîte aux lettres est usuellement réalisée sous la forme d'une file (premier entré, premier sorti : FIFO) mais les implantations peuvent également être plus sophistiquées. Par exemple, le langage ABCL ou *Plasma II* (qui seront présentés plus loin) intègrent à chaque message une notion de priorité qui impose l'utilisation d'une structure de boîte aux lettres plus complexes. Les acteurs communiquent entre eux par envoi asynchrone de messages qui peuvent contenir des adresses. La topologie du médium de communication est donc dynamique. Un acteur est également composé d'un *comportement* : un programme qui indique sa réaction à la

10. Un agent intelligent est une entité autonome qui se déplace de machine en machine dans un but qui lui est propre. Il est de plus capable de décision autonome.

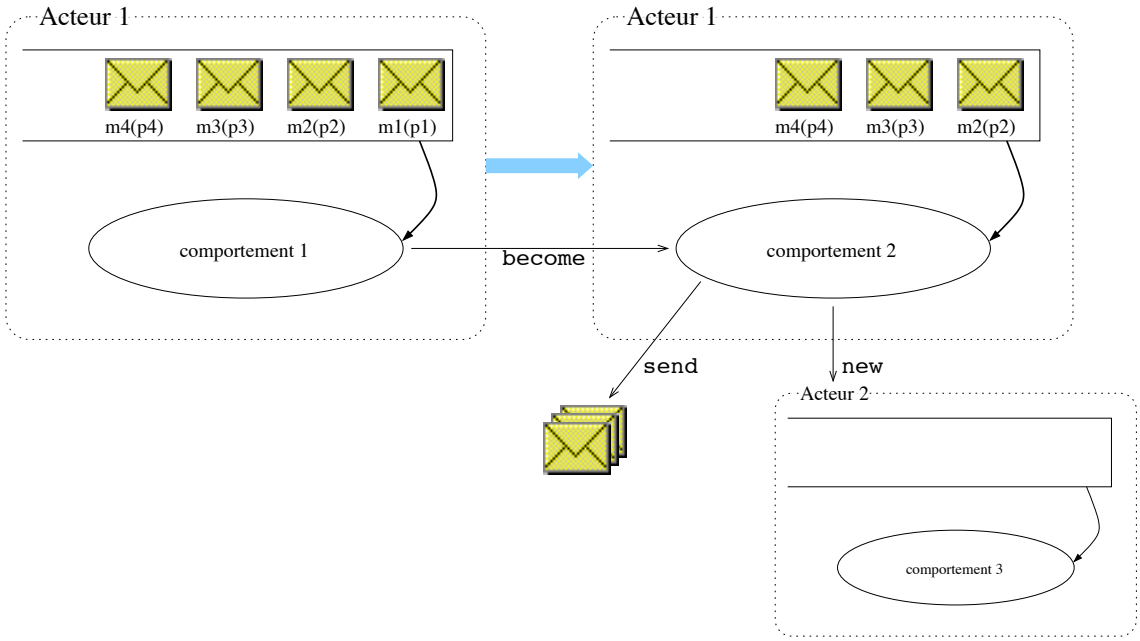


Figure 1.1 Un acteur avant et après traitement d'un message.

réception de tel ou tel message. Ce comportement peut changer de façon radicale. Pour illustrer ce fait, G. AGHA dans [AH92] présente l'exemple d'un acteur gérant un local commercial. Une agence bancaire utilise ce local commercial jusqu'à ce qu'elle déménage, à la suite de la réception d'un message particulier (par exemple, un courrier de résiliation de bail). Le local est alors loué à un livreur de pizza, lequel bien évidemment ne saura pas répondre aux courriers destinés à la banque. Ce changement d'état est généralement spécifié statiquement, *i.e.* dans le code d'une réaction, mais le nouveau comportement peut également être calculé dynamiquement (en étant par exemple reçu dans un message). Pour offrir un haut niveau d'abstraction, le modèle ne propose que trois opérations exécutables par un acteur à chaque réception de message. Celles-ci peuvent ensuite être implantées de manière spécifique lors de la conception d'un langage d'acteurs. Les comportements contiennent donc des calculs spécifiés dans un langage de programmation quelconque et des opérations directement issues du modèle :

- la *création* d'un nombre fini d'acteurs,
- l'*envoi* d'un nombre fini de messages vers d'autres acteurs,
- le *changement* du comportement de l'acteur (qui traitera alors le message suivant).

Dans le modèle des acteurs, le changement de comportement peut être absent de la réaction. Dans ce cas, l'acteur termine son exécution et peut donc être détruit.

Il est possible de schématiser l'exécution d'un programme contenant des acteurs par la figure 1.1. Lors du traitement d'un message, un acteur peut changer son comportement (`become`), envoyer des messages (`send`) et créer de nouveaux acteurs (`new`). Notons que pour simplifier le schéma, nous avons représenté la création d'un seul acteur.

La seconde abstraction du modèle d'acteur est le *message*, sa structure n'est pas précisée dans le modèle. Cependant, celui-ci suppose qu'il existe un moyen de déterminer la réaction d'un acteur recevant un message parmi toutes les réactions possibles de son

comportement [11]. Le message permet donc de décider de la réaction mais il contient également des données. Ces données peuvent être de forme classique comme des entiers, des chaînes de caractères ou des listes, mais elles peuvent également être des adresses ou des comportements d'acteurs. Cela signifie que dans le langage qui précise la notion de calcul interne des acteurs, les adresses et les comportements sont des valeurs du premier ordre.

La communication entre les acteurs est de type point à point unidirectionnelle, c'est-à-dire qu'il faut connaître l'adresse d'un acteur pour lui envoyer un message et celui-ci ne sait alors pas qui lui a écrit [12]. Les acteurs, dont un acteur connaît l'adresse, forment l'ensemble de ses *connaissances*. Remarquons que cet ensemble est dynamique car, au cours de son exécution, un acteur peut recevoir l'adresse d'un autre acteur qu'il ne connaissait pas encore, ou oublier l'adresse d'un acteur qu'il connaissait. De plus, l'envoi de message est une opération non-bloquante car asynchrone. L'accès à un autre acteur via un message mis dans une boîte aux lettres provoque donc une sérialisation automatique du traitement des messages. Ainsi, les acteurs peuvent devenir passifs, mais sont toujours accessibles aux autres acteurs. Ceci permet d'éviter les conflits d'accès et donc de dispenser le programmeur de la gestion des accès concurrents (par exemple par exclusion mutuelle). Les risques d'apparition d'interblocage, phénomène relativement courant lors de la programmation d'applications concurrentes, sont fortement réduits. Enfin, la seule hypothèse faite sur le médium de communication est qu'il est sûr, c'est-à-dire, qu'un message envoyé sera toujours reçu par son destinataire en un temps fini. En revanche, aucune hypothèse n'est faite sur l'ordre de réception des messages. Remarquons que le protocole de communication entre acteurs impose de nouvelles méthodes de programmation car il n'est pas possible de recevoir directement une réponse à un message envoyé. Si un acteur a besoin lors de son évaluation du résultat d'un calcul effectué par un autre acteur, il doit passer par un état (comportement) intermédiaire où il attend uniquement le message contenant le résultat de ce calcul.

Le modèle ainsi décrit permet la programmation d'applications concurrentes avec une certaine transparence au niveau des communications. De plus, par nature, les acteurs vérifient les *bonnes* propriétés que sont l'extensibilité et la réutilisabilité. Le lecteur intéressé par une analyse plus précise de l'expressivité du modèle d'acteur peut consulter le livre de G. Agha sur les Acteurs [Agh86].

Remarque :

En général, lors de la réalisation d'un langage complet, on ajoute d'autres primitives plus complexes ne faisant pas partie de cette base. On rencontre, par exemple, des primitives de communication synchrone ou des notions de groupe d'acteurs avec des fonctionnalités de type diffusion. Nous évoquerons quelques unes de ces extensions possibles lorsque nous présenterons quelques exemples de langages d'acteurs.

11. En général, dans les langages d'acteurs, cette opération de choix de la réaction est effectuée par filtrage du message, soit en utilisant sa structure (*i.e.* son type) comme en Erlang, soit le message contient une étiquette qui guide ce filtrage comme en ML-Act.

12. Remarquons qu'il est relativement aisé dans la pratique de joindre l'adresse de l'envoyeur à un message. D'ailleurs, certains langages d'acteurs le font automatiquement.

Acteurs et Objets

Le concept d'acteurs est proche de celui d'objets répartis (ou concurrents), mais les deux modèles ont une différence importante sur le but recherché. L'héritage des objets permet d'augmenter la réutilisabilité et de diminuer la quantité de programme à écrire. Dans le monde des acteurs, l'optique est sensiblement différente puisque l'on s'intéresse surtout à l'adaptabilité d'un acteur à son environnement : le changement de comportement – *i.e.* le changement de son interface. De plus, les acteurs utilisent plutôt la délégation qui est souvent plus souple et plus puissante que l'héritage.

La souplesse du changement de comportement semble être nécessaire dans bien des langages concurrents rendant nécessaire sa conservation dans l'étude d'un modèle de concurrence. Par exemple, sur le réseau Internet les serveurs offrent des services similaires mais souvent légèrement différents. Ainsi, un acteur modélisant un service changera de comportement selon le serveur qui répondra effectivement à la requête.

Un langage à objets sans héritage peut être modélisé aisément en utilisant des acteurs qui ne changeraient pas de comportement. Le modèle des acteurs est donc, dans un sens, plus général que celui des objets sans héritage. Afin de le rendre vraiment plus général que le modèle objet, il faudrait réaliser un modèle d'acteurs intégrant l'héritage [13], mais on sortirait alors du cadre de notre étude qui est essentiellement une étude de la communication. En fait, les acteurs suivent le modèle d'*encapsulation* des objets. C'est-à-dire que les *clients* d'un acteur a ne peuvent pas modifier directement son contenu et sont contraints d'utiliser les routines qu'il rend publiques. Dans le modèle des objets, cette interface publique est composée de méthodes que le client *appelle* (comme une fonction), alors que dans celui des acteurs, une interface est un ensemble de motifs (le comportement de l'acteur) qui filtrent les messages envoyés par le client.

Dans certains langages à objets concurrents, une méthode est *protégée* par une garde, c'est-à-dire qu'elle n'est accessible que sous certaines conditions. Cette notion de méthode gardée correspond exactement à un changement de comportement des acteurs, on pourra donc également modéliser ces langages dans ce cadre.

Enfin, nous préférons travailler sur le modèle d'acteur car nous trouvons que les acteurs permettent au programmeur de mieux mettre en avant la notion d'état (les comportements des acteurs) des systèmes qu'ils développent. Nous pensons que cette caractéristique augmente sensiblement la facilité de maintenance des logiciels, car elle clarifie les protocoles de communication entre entités. En effet, l'analyse des filtrages des comportements permet immédiatement de connaître la forme des messages que l'acteur attend.

Les langages d'Acteurs

Le modèle d'acteurs permet de construire un langage de programmation acteur en ajoutant à un langage séquentiel les primitives acteurs (`send`, `new`, `become` ainsi que la notion de comportement). Notons que lors de la réalisation d'un langage d'acteurs certaines hypothèses du modèle peuvent se retrouver adaptées (le protocole d'accès peut être changé, d'autres formes d'envoi de message sont autorisées ...). De nombreux langages d'acteurs ont été ainsi conçus parmi lesquels on peut citer :

13. Cela a été réalisé dans le cadre de certains langages d'acteurs, on peut citer par exemple ROSETTE.

- Le langage ROSETTE, développé par C. TOMLINSON et al. ([TS89, AKS+88]) au sein de Microelectronics and Computer Consortium (MCC), est un langage proche de SMALLTALK. En effet, dans ROSETTE, tout est acteur, y compris les données de base comme les entiers par exemple. Ce langage intègre également une notion d'héritage par *prototypage* et intègre la notion d'envoi de message asynchrone ainsi qu'une forme de communication synchrone (le résultat de la réception du message par le destinataire est renvoyé à l'expéditeur). C'est un des langages d'acteurs les plus aboutis. Il a été conçu dans le cadre d'un projet de MCC de construction d'un commutateur réseau *intelligent*. ROSETTE était alors le langage de commande du commutateur qui contenait un interpréteur de celui-ci.
- ACORE conçu par C. MANNING ([Man87]) au MIT. Ce langage principalement expérimental est issue d'une lignée de langages d'acteurs développés au MIT sous les noms de ACT1, ACT2 et ACT3 ([The82, The83, AH87]). Ces langages ont surtout servi à tester et mettre au point différentes stratégies pour l'exécution de programmes d'acteurs. Ils sont tous fortement inspirés de LISP et d'ailleurs implantés sur des machines disposant d'une plate-forme LISP répartie. Ces langages sont donc de type fonctionnel et ont été utilisés principalement dans le domaine de l'intelligence artificielle.
- La famille de langages HAL par G. AGHA et al. ([AH92, KA92, Kim97]). Ces langages sont des descendants de la famille ACTX du MIT. Ils ont été surtout développés envers un tout autre domaine d'application : le calcul scientifique haute performance. Ils ont donc servi à l'expérimentation de ce que les acteurs pouvaient apporter aux calculs numériques ainsi qu'à la mise au point d'implantation efficace sur des machines parallèles (une extension de HAL nommée THAL a par exemple été porté sur la CM5 de Thinking Machine Corporation). Les concepteurs de ces langages y ont systématisé la notion de groupe d'acteurs ainsi que la diffusion de message au sein d'un groupe.
- La famille de langages issus de ABCL développée par A. YONEZAWA et al. ([Yon90, YBS86, YSTH87]). Contrairement à la plupart des autres langages d'acteur ABCL est un langage impératif. Bien que basé sur le modèle d'acteur, il inclut également d'autres mécanismes, par exemple, il impose un ordre de causalité sur les messages. Ainsi, deux messages envoyés par un acteur à un autre acteur sont reçus dans l'ordre d'envoi par le destinataire. De plus, le langage contient deux sortes de messages, les messages *express* et les messages *normaux*[14] ainsi que trois formes d'envoi de message : l'envoi asynchrone classique, l'envoi synchrone bloquant (avant de poursuivre le calcul, on attend la réponse du destinataire comme pour un RPC) et l'envoi synchrone non-bloquant (après un envoi l'acteur poursuit son calcul et il n'est bloqué que lorsqu'il doit utiliser le résultat de la requête, c'est la notion de *futur*).
- *Plasma II*, conçu par l'équipe Plasma de l'IRIT ([MMS88]), est une extension distribuée du langage *Plasma* ([Hew77], [Pom80] et [Dur81]) dont la syntaxe et la sémantique sont proches de LISP. Le langage contient plusieurs mécanismes de communication assez sophistiqués : les messages ont des priorités, les acteurs sont

14. En fait, la boîte aux lettres d'un acteur est composée de deux queues qui n'ont pas la même priorité d'accès.

interruptibles par des évènements (messages de priorité maximale en quelque sorte) et enfin, il permet la capture de continuation et permet ainsi de réaliser des messages synchrones. Un interprète distribué a été réalisé pour un réseau de stations UNIX et sur une machine parallèle.

Le modèle d'acteur a également connu un certain succès comme modèle de concurrence dans le cadre du développement de certains langages concurrents. Ces langages, à la différence de ce qui ont déjà été décrits, ne sont pas résolument acteur. On peut citer :

- ERLANG conçu par J. ARMSTRONG et al. ([AVWW96]), qui est un langage fonctionnel concurrent, dont le modèle de concurrence est acteur. Il a été développé au sein d'ERICSSON et est utilisé pour la programmation logicielle des équipements réseaux vendus par la firme. Au sein d'ERLANG, la communication suit le modèle d'acteur mais il n'existe pas une véritable notion de comportement avec une alternance actif/passif selon la réception d'un message ou non. En fait, le langage contient un opérateur d'accès à la queue (`receive`) qui permet de récupérer un message y figurant.
- Le modèle d'acteur a également inspiré les travaux sur les objets actifs menés par D. CAROMEL dans le but d'étendre des langages existants tels C++, JAVA et EIFFEL (voir [CBR96, CV98, CR95]) afin de les utiliser dans le domaine de la programmation parallèle.

Pour finir la description de travaux ayant trait aux langages concurrents ou parallèles utilisant le modèle d'acteurs, il faut citer ACTALK ([Bri89, Bri88, LBB91]) et ACTRA ([LTP86]). Il s'agit de plates-formes d'expérimentation conçues pour étendre SMALLTALK au domaine de la programmation acteur. Ces deux extensions ne procèdent toutefois pas de manière similaire puisque la première, ACTALK, implante un noyau minimal de langage d'acteur en ajoutant à SMALLTALK l'envoi de message asynchrone et les files de messages, alors qu'ACTRA modifie la machine virtuelle de SMALLTALK afin de rendre les acteurs primitifs. Notons également qu'ACTRA utilise une notion de communication synchrone, alors qu'elle est asynchrone en ACTALK. Ces plates-formes ne sont pas conçues comme de véritables langages de programmation mais plutôt comme des outils permettant d'implanter rapidement des prototypes de langage et de les comparer.

Tous les langages présentés comportent des mécanismes de synchronisation plus ou moins évolués (et donc complexes) que le modèle d'acteur permet d'abstraire et de décrire simplement. Ainsi, une formalisation de ce modèle fournirait un outil puissant d'étude de tous ces langages. Elle est également nécessaire à une définition précise de leurs sémantiques qui est indispensable pour montrer la correction des analyses statiques que nous allons réaliser. La section suivante présente notre modélisation des acteurs.

1.2 Les configurations

Les deux langages que nous étudierons dans la suite de cette thèse sont des langages de programmation. Ils ont donc une syntaxe complète complexe qui alourdit la présentation de leur sémantique, de leur typage ainsi que la preuve de la correction des systèmes de type. Il est donc de tradition dans la communauté scientifique de travailler sur des noyaux très réduits.

Pour exprimer la sémantique des programmes ML-Act ou Erlang, il est possible d'utiliser le π-calcul. Mais, il faut alors coder les acteurs, ce qui a le défaut de déstructurer totalement les programmes. Si le sujet d'étude est la sémantique cette limitation n'est pas gênante, le π-calcul devient un *langage d'assemblage* par rapport à ML-Act et Erlang. Cependant, si l'on cherche à analyser les programmes une fois *traduits* en π-calcul, la structuration devient trop faible pour détecter les éventuelles erreurs contenues dans un programme. Même s'il est plus proche du modèle d'acteurs, Cap ne convient pas non plus. En effet, il n'intègre pas de notion de calcul (fonctionnel) qu'il faut alors coder. Nous avons mis au point un formalisme que nous appellerons les *configurations*. Ce formalisme proche d'un calcul de processus par sa syntaxe est issu de Cap. Cette section va présenter rapidement ce formalisme.

Une origine : Cap

Cap [15] est un calcul de processus conçu et développé au sein de notre équipe dans le cadre de la thèse de J-L. Colaço [Col97] (on peut également consulter [CPS96]). Il a été défini dans le but de décrire simplement le modèle d'acteur ainsi que ses mécanismes fondamentaux, comme par exemple la création d'acteurs ou le changement de comportement. Il est fortement inspiré du π-calcul de R. Milner et al. ([MPW92] ou [Mil99]) et plus particulièrement de ses versions asynchrones (voir [Bou92] et [HT91]). Son originalité par rapport au π-calcul est l'intégration de concepts et de mécanismes issus du monde des objets (plus précisément du Calcul d'Objets Primitifs de M. Abadi et L. Cardelli).

La volonté de définir un nouveau formalisme plutôt que d'utiliser directement le π-calcul résulte de la difficulté d'étudier un modèle et d'en analyser les propriétés en le traduisant dans un autre modèle. B. Pierce et D. Turner ont été confrontés au même problème lors de l'étude des objets fortement typés à travers leur traduction dans un calcul de processus. Dans [PT95], ils utilisent une extension du π-calcul afin de conserver suffisamment d'informations sur la structure des objets. Dans le cas de la modélisation des acteurs les deux principaux problèmes soulevés par une traduction en π-calcul sont :

- Les notions d'adresse d'acteurs et d'étiquette de messages dans le cas de ML-Act (voir le chapitre sur ML-Act) seraient toutes deux traduites par la notion de canal. Cette fusion rendrait impossible les analyses de type que nous souhaitons réaliser sur ML-Act.
- La structure complexe de comportement est fortement récursive dans le modèle d'acteur. En général, un acteur assume plusieurs comportements mutuellement récursifs. La traduction d'un comportement nécessite donc plusieurs processus (V. Vasconcelos dans [Vas94] fournit un codage de la récursivité dans un calcul avec réplication). Celui-ci a d'ailleurs introduit dans ses travaux avec A. Ravara une notion de récursivité pour cette raison. La structure d'acteur est alors perdue ce qui rend impossible toute analyse par typage.

Nous ne présenterons pas Cap en détail, le lecteur désireux d'en savoir plus pourra consulter la thèse de J-L. Colaço [Col97]. Cependant, afin de faciliter la présentation de la notion de configuration, nous allons évoquer Cap à partir d'un exemple.

15. Cap signifie Calcul d'Acteurs Primitifs.

$$\begin{array}{rl} \nu a, b, c(a \rhd [read(d) & = \zeta(e,s)(d \lhd rep(s.val) \parallel e \rhd s) \quad , \\ write(v) & = \zeta(e,s)e \rhd s.val \Leftarrow v \quad , \\ val & = b] \\ \parallel a \lhd write(c)) & \end{array}$$

Exemple 1.1 Une cellule en CAP.

CAP intègre de manière primitive les noms (ou adresses), les comportements, les acteurs et les messages. D'une manière similaire aux langages à objets dans lesquels une méthode prend implicitement en paramètre l'objet qui la définit (son *self* [16]) ; un acteur doit pouvoir accéder à son comportement (appelé également *self*) ainsi qu'à son nom (appelé *ego*, il correspond à l'identité des objets). Pour accéder à ces informations, le calcul intègre un mécanisme d'auto-application emprunté au calcul d'Objets Primitifs de M. ABADI et L. CARDELLI [AC94] dont le rôle est, au moment de la prise en compte d'un message, de capturer le *self* (comme dans les Objets Primitifs) et l'*ego*.

Une expression CAP est composée d'un nombre fini de messages et d'un nombre fini d'acteurs mis en parallèle. L'exemple 1.1 représente une cellule modifiable et permet d'illustrer les principaux traits de CAP. L'opérateur ν correspond à la déclaration (ou création) de trois noms (a, b, c), dont la portée est définie par les parenthèses les plus externes. Sous cette portée se trouve un acteur de nom a ($a \rhd [\cdots]$) dont le comportement définit deux méthodes $read(d)$ et $write(v)$; celles-ci sont accessibles par envoi de message. Il contient également un champ val que nous qualifions de *privé* car non accessible par communication (seuls les traitements faits à l'intérieur d'un comportement peuvent utiliser les champs privés de ce comportement). Cet acteur est mis en parallèle avec un message qui lui est destiné ($a \lhd write(c)$). Le corps de chaque méthode de l'acteur a est préfixé par l'opérateur ζ qui lie dans le corps de la méthode les variables e et s respectivement à l'*ego* (a) et au *self* (l'entité entre crochet). La méthode $read(d)$ envoie à son argument un message rep contenant la valeur mémorisée par la cellule et reprend son comportement courant ($e \rhd s$). La méthode $write(v)$ se contente de faire reprendre à l'acteur a son comportement courant en modifiant la valeur sauvegardée dans le champ privé val.

La syntaxe des configurations

Les sémantiques que nous allons donner aux programmes ML-ACT et ERLANG dans la suite de cette thèse vont se présenter sous la forme de règles de réduction. Celles-ci décriront toutes les étapes de l'évaluation et seront définies sur des *configurations*. Cette notion de configuration permet de modéliser le médium de communication – un groupe d'acteurs s'exécutant en parallèle ainsi qu'un certain nombre de messages en transit. Ces configurations expriment la partie concurrente d'un programme et leur syntaxe est fortement inspirée de CAP. Nous avons cependant décidé de ne pas utiliser CAP car celui-ci était à la fois trop riche pour un calcul de processus et trop pauvre en tant que langage

16. Dans les langages existants ce concept est généralement implicite, par exemple, en Eiffel, en Java ou en C++, l'objet se connaît lui même.

de calcul fonctionnel [17]. La notion de configuration est donc une simplification de CAP où tout ce qui concerne les comportements est supprimé.

Ce formalisme est paramétré par la donnée de différents ensembles vérifiant certaines propriétés (principalement pour la description du calcul fonctionnel). Ces différentes données abstraites, les contraintes qu'elles doivent respecter et une explication intuitive de leur signification seront présentées au fur et à mesure de l'introduction de la notion de configuration et de sa sémantique.

Définition 1.1 (Configuration) :
Une configuration est un terme qui représente un système d'acteurs à un instant donné. Sa définition est paramétrée par trois ensembles : l'ensemble des noms d'acteurs (notés $a \in \mathbb{A}$), celui des messages (notés $m \in \mathcal{M}ess$) et celui des expressions (notés $e \in \mathcal{E}xp$). Elles sont construites par la grammaire qui suit et leur ensemble est noté W :

$$\begin{aligned} w &::= \epsilon \mid \boldsymbol{Err} \mid \nu a.w \mid w \parallel w \mid a \lhd m \mid \alpha \rhd e \\ \alpha &::= \star \mid \langle a \,|\, q \rangle \\ q &::= \varnothing \mid m :: q \end{aligned}$$

Les ensembles $\mathbb{A}$, $\mathcal{M}ess$ et $\mathcal{E}xp$ ne peuvent pas être quelconques. En effet, les noms et les messages sont des expressions ($\mathbb{A} \subset \mathcal{E}xp$ et $\mathcal{M}ess \subset \mathcal{E}xp$). Les expressions intègrent la notion de fonction et son application mais également les primitives acteurs ainsi que la notion de comportement. Les deux langages que nous allons manipuler, même s'ils sont tout deux fonctionnels, ont des sémantiques fonctionnelles très différentes notamment quant à la liaison variable/valeur [18]. Ils utilisent donc chacun un λ-calcul avec constantes spécifique qui est présenté dans le chapitre qui leur est consacré.

Une configuration consiste en la mise en parallèle d'un certain nombre de sous-configurations. Celles-ci peuvent être vide (ϵ), correspondre à une erreur ($\mathbf{Err}$), un message m en transit dans le médium vers l'acteur d'adresse a ($a \lhd m$) ou un acteur d'adresse α exécutant son corps e ($\alpha \rhd e$). L'identité d'un acteur peut prendre deux formes, soit l'acteur est *anonyme* et son identité est $\star$, soit c'est un acteur classique et son identité $\langle a \,|\, q \rangle$ contient son nom a et sa boîte aux lettres q. Un acteur anonyme ne peut pas recevoir de message (car il n'a pas d'adresse) et donc une fois l'évaluation de son corps terminée, il devient définitivement inactif. Cette notion d'acteur anonyme permet de coder les expressions qui ne correspondent pas à un calcul se déroulant dans un acteur, comme par exemple les expressions globales. Il est de plus possible de restreindre la portée d'un nom d'acteur ($\nu a.w$). Cette restriction de portée correspond en fait à la création d'un nouveau nom (donc d'un nouvel acteur).

17. Dans sa thèse, J-L. COLAÇO donne un codage du λ-calcul en CAP, permettant ainsi d'effectuer tous les calculs possibles en CAP. Cependant, si on utilise ce codage, beaucoup d'informations sur la structure des calculs fonctionnels sont perdues et les analyses que nous avons mises au point ne peuvent plus être réalisées.

18. En ML-ACT, une variable est introduite avec une portée bien précise dont elle ne peut s'échapper. En ERLANG, une variable introduite a une portée qui peut échapper aux limites de l'instruction dans laquelle elle figure. Pour plus de détails sur ces différences, nous reportons le lecteur au chapitre 7 à la page 155.

$$
\begin{array}{llll}
\mathcal{FN}(\epsilon) & = \{\} & \mathcal{FN}(\nu a.w) & = \mathcal{FN}(w) \setminus \{a\} \\
\mathcal{FN}(\mathbf{Err}) & = \{\} & \mathcal{FN}(w_1 \parallel w_2) & = \mathcal{FN}(w_1) \cup \mathcal{FN}(w_2) \\
\mathcal{FN}(a \triangleleft m) & = \{a\} \cup \mathcal{FN}(m) & \mathcal{FN}(\alpha \triangleright e) & = \mathcal{FN}(\alpha) \cup \mathcal{FN}(e) \\
\mathcal{FN}(\star) & = \{\} & \mathcal{FN}(\langle a \,|\, q \rangle) & = \{a\} \cup \mathcal{FN}(q) \\
\mathcal{FN}(\varnothing) & = \{\} & \mathcal{FN}(m :: q) & = \mathcal{FN}(m) \cup \mathcal{FN}(q)
\end{array}
$$

Règles 1.1 Ensemble des noms libres d'une configuration.

Détaillons un peu plus la restriction de nom, l'opérateur ν est un lieur vis-à-vis des adresses, nous allons donc définir la notion de nom libre par rapport à ce lieur ainsi que la notion de substitution associée.

Définition 1.2 (Nom libre) :
*Un nom est **libre** dans une configuration, si et seulement si il ne figure pas dans un terme qui le lie par un opérateur ν. L'ensemble des noms libres d'une configuration w est noté $\mathcal{FN}(w)$ et est calculé inductivement sur la structure de w par l'ensemble des règles 1.1.*

Définition 1.3 (Substitution de Nom) :
*Une **substitution de nom** σ est un automorphisme de l'ensemble des configurations, son action sur une configuration w est notée $\sigma(w)$. Elle remplace toutes les occurrences libres de certains noms par d'autres noms. Une substitution simple est notée $[b/a]$ si elle remplace a par b, son domaine est alors $\{a\}$ et son codomaine est $\{b\}$. Ces substitutions se composent comme les fonctions et on définit un opérateur de restriction $|$:*

$$\sigma|_a(x) \triangleq \begin{cases} x & si\ x = a \\ \sigma(x) & sinon \end{cases}$$

La substitution d'un ensemble de noms est définie par les règles 1.2.

Remarque :
Remarquons que les règles définissant le calcul des noms libres (resp. l'action d'une substitution) utilisent les notions de nom libre et de substitution dans un message ou dans une expression. Ces notions seront introduites lors de la définition formelle de l'entité à laquelle elles correspondent.

L'évaluation de programmes d'acteurs sera définie sous la forme de règles de réduction entre configurations. Cette sémantique *à petit pas* est mieux adaptée à la définition de sémantiques concurrentes. En effet, du fait de son faible grain, elle permet de mieux exprimer tous les entrelacements qui peuvent se produire. Cette réduction exprimant l'exécution d'un programme d'acteur est en fait la combinaison de deux formes d'évaluations différentes : les calculs au sein d'un acteur et les activités concurrentes issues du

$$
\begin{array}{llll}
\sigma(\epsilon) & = \epsilon & \sigma(\nu a.w) & = \nu a.\sigma|_a(w) \\
\sigma(\mathbf{Err}) & = \mathbf{Err} & \sigma(w_1 \parallel w_2) & = \sigma(w_1) \parallel \sigma(w_2) \\
\sigma(a \triangleleft m) & = \sigma(a) \triangleleft \sigma(m) & \sigma(\alpha \triangleright e) & = \sigma(\alpha) \triangleright \sigma(e) \\
\sigma(\star) & = \star & \sigma(\langle a \,|\, q \rangle) & = \langle \sigma(a) \,|\, \sigma(q) \rangle \\
\sigma(\varnothing) & = \varnothing & \sigma(m :: q) & = \sigma(m) :: \sigma(q)
\end{array}
$$

Règles 1.2 Substitution de noms dans une configuration.

modèle d'acteur (échange de message, création d'acteur, ...). Ces deux évaluations vont être exprimées par des réductions différentes $\longrightarrow_e$ pour les expressions et $\longrightarrow$ pour les configurations. Ainsi, à chaque étape de réduction, il sera possible d'entrelacer totalement les calculs fonctionnels avec les calculs concurrents. Le deuxième avantage d'utiliser deux systèmes est de permettre leur présentation (presque) indépendamment l'un de l'autre. Ainsi, la partie concurrente commune à ML-ACT et ERLANG peut être introduite une seule fois. Cependant, pour abstraire la partie fonctionnelle, il est nécessaire d'introduire certains concepts qui ne seront définis précisément que dans les parties de cette thèse consacrées aux langages. Nous imposons ainsi des contraintes sur la forme du langage acceptable pour décrire la partie fonctionnelle et nous suivons une approche classique lors de la programmation d'application.

Définition 1.4 (Réduction fonctionnelle, Interface et Valeur Sémantique) :
Une relation de réduction (dite fonctionnelle) notée $\longrightarrow_e$ doit être définie sur l'ensemble abstrait des expressions.

*Les expressions doivent contenir une notion d'**interface**, une interface étant un ensemble de couple filtre de message / expression. L'ensemble des interfaces est noté $\mathbb{I}$ et ses éléments $\mathcal{R}$, il contient une interface vide qui correspond à un acteur autiste ne pouvant plus répondre à aucune requête (nous la notons $\emptyset$). Cette notion sera définie plus précisément pour chaque langage.*

*Une **valeur sémantique**, que nous notons v, est une expression sur laquelle aucune réduction fonctionnelle ne peut avoir lieu. L'ensemble des valeurs sémantiques est noté $\mathcal{V}$ et contient au moins trois entités : les adresses d'acteurs, l'expression inactive notée nop et la notion d'interface. Ainsi : $\mathbb{A} \cup \{nop\} \subset \mathcal{V}$ et $\emptyset \in \mathbb{I} \subset \mathcal{V}$ et $\mathcal{V} \subset \mathcal{E}xp$*

Une interface filtrera les messages reçus par un acteur et exécutera la réaction de l'acteur qui correspond à la forme du message.

Remarque :
Il convient de ne pas confondre la notion d'interface avec celle de comportement. Un comportement est une fonction qui prend en paramètre une adresse (l'ego de l'acteur qui l'exécutera), qui lie la valeur du self et qui renvoie une interface. Alors qu'une interface ne contient que la partie filtrage et le corps des réactions. En anticipant sur la forme des

(0)	$\nu a.w$	$\equiv \nu b.[b/a]w$ si $b \notin \mathcal{FN}(w)$	**(4)**	$w \parallel \mathbf{Err}$	$\equiv \mathbf{Err}$
(1)	$\nu a.w$	$\equiv w$ si $a \notin \mathcal{FN}(w)$	**(5)**	$w \parallel \epsilon$	$\equiv w$
(2)	$\nu a_1.\nu a_2.w$	$\equiv \nu a_2.\nu a_1.w$	**(6)**	$w_1 \parallel w_2$	$\equiv w_2 \parallel w_1$
(3)	$\nu a.w_1 \parallel w_2$	$\equiv \nu a.(w_1 \parallel w_2)$ si $a \notin \mathcal{FN}(w_2)$	**(7)**	$(w_1 \parallel w_2) \parallel w_3$	$\equiv w_1 \parallel (w_2 \parallel w_3)$
(8)	$\star \triangleright v$	$\equiv \epsilon$	**(9)**	$\nu a.(\langle a \,\vert\, \varnothing \rangle \triangleright \emptyset)$	$\equiv \epsilon$

Règles 1.3 Congruence sur les configurations.

λ-calcul que nous utiliserons :

$$\underbrace{\lambda ego.\,\texttt{letrec}\ self = \overbrace{\lambda[get(c).\ ...,\ set(v').\ ...]}^{interface}\ \texttt{in}\ self)}_{comportement}$$

De plus, nous identifierons certaines configurations qui ont la même signification au moyen d'une congruence. Celle-ci, notée $\equiv$ exprime les propriétés algébriques vérifiées par les configurations. Avant de la définir plus précisément, nous allons introduire la notion de valeur sémantique.

Définition 1.5 (Congruence) :
Nous définissons $\equiv$ comme la plus petite congruence définie sur l'ensemble des configurations vérifiant les règles 1.3.

Les propriétés algébriques vérifiées par les configurations peuvent être séparées en trois familles selon leur sujet (la restriction, l'opérateur $\parallel$ ou la construction d'acteurs). Ces propriétés expriment intuitivement :

- la restriction ν :
  - **(0)** Le renommage des noms liés ne modifie pas le sens d'une configuration.
  - **(1)** La restriction agit seulement sur les noms libres contenus dans sa portée.
  - **(2)** L'ordre de restriction des noms n'est pas important.
  - **(3)** La portée d'un nom a peut être étendue à une configuration dans laquelle il n'est pas libre. Cette règle porte usuellement le nom d'*extrusion des noms*. Ce mécanisme associé à l'α-conversion des noms permet de simuler l'allocation de nouveaux noms – *i.e.* des noms qui ne figurent pas déjà dans la configuration. En effet, si $a \in \mathcal{FN}(w_2)$ il sera toujours possible de trouver un nom a' non-libre dans w_2 et d'appliquer l'α-conversion de a en a' dans w_1. Ainsi, on pourra appliquer la règle d'extrusion à a' et ainsi étendre sa portée à w_2 .
- l'opérateur $\parallel$:
  - **(4)** **Err** est un élément absorbant de l'opérateur parallèle, cela signifie que les erreurs sont propagées jusqu'à l'arrêt de tous les calculs.

(5,6,7) $(W, \|, \epsilon)$ est un monoïde commutatif. Ainsi, les configurations vides peuvent être détruites par le ramasse-miettes.

– les acteurs $\triangleleft$:

(8,9) Les acteurs anonymes ayant terminé leurs calculs ou les acteurs d'interface vide n'étant plus connus et ayant une boîte aux lettres vide peuvent être détruits par le ramasse-miettes.

Dans la suite de ce manuscrit, nous ne travaillerons que sur les configurations modulo cette relation de congruence. Nos configurations seront donc des classes d'équivalence de configurations telles qu'elles ont été formellement définies. On peut définir une notion de représentant pour chaque classe d'équivalence, cette configuration particulière sera appelée *forme normale*. Leur forme est proche de la forme normale des termes du π-calcul (voir par exemple [Mil99]).

Définition 1.6 (Forme Normale d'une Configuration) :
Une configuration sera dite en ***forme normale*** *si elle a une des formes suivantes :*

– ϵ
– $\mathbf{Err}$
– $\nu a_1 \dots a_n.(W_1 \parallel \dots \parallel W_m)$ *avec* $\{a_1, \dots, a_n\} \subset \mathcal{FN}(W_1 \parallel \dots \parallel W_m)$ *et* W_i *de la forme* $a \triangleleft m$ *ou bien* $\alpha \triangleright e$.

Pour assurer l'unicité de la forme normale, il faut se donner un ordre sur l'ensemble des noms ($<_a$) *et un autre sur l'ensemble des configurations* ($<_w$) *qui ont la forme* $a \triangleleft m$ *ou bien* $\alpha \triangleright e$. *A partir de ces ordres, il est possible de fixer l'ordre des restrictions et celui des sous-configurations. Par exemple, on peut imposer* $i < j \Longrightarrow a_i <_a a_j$ *et* $i < j \Longrightarrow W_i <_w W_j$ *(à partir d'une indexation quelconque des noms et des configurations* $a \triangleleft m$ *et* $\alpha \triangleright e$*).*

On appelle ***forme normale*** *d'une configuration* w*, l'unique configuration* w' *en forme normale qui est équivalente à* w $(w \equiv w')$.

Il est aisé de construire un algorithme qui calcule la forme normale d'une configuration. L'algorithme suivant que nous donnons sous forme informelle et que nous ne démontrerons pas convient :

– Si w contient une erreur, sa forme normale est $\mathbf{Err}$.
– Sinon, on applique les règles suivantes jusqu'à ce que cela ne soit plus possible :
  – Éliminer les ϵ des configurations en parallèle.
  – Appliquer l'extrusion des noms pour remonter les définitions de noms le plus à l'extérieur possible.
  – Éliminer les restrictions de noms inutiles.

  À la fin de ce processus, soit la forme normale vaut ϵ, soit elle a la troisième forme de la définition et il ne reste plus qu'à ordonner les restrictions et les sous-configurations.

Notons que l'ordre choisi ici est totalement arbitraire, ainsi tout autre critère d'ordancement est valide. La notion de forme normale est donc définie modulo cet ordre.

Les configurations telles qu'elles ont été définies peuvent contenir plusieurs acteurs de même nom. Cependant dans le cadre de cette thèse, nous nous limitons aux configurations qui ne contiennent que des acteurs de noms différents. Cette propriété est appelée la *linéarité*. Signalons que dans sa thèse [Col97], J-L. Colaço présente un système de type qui permet de vérifier la linéarité d'un terme Cap. Il est aisé d'adapter ce système de

type aux configurations. Ce système n'a pas d'intérêt dans notre cadre puisque nous ne travaillons pas sur des configurations quelconques mais uniquement sur le sous-ensemble des configurations qui sont obtenues par traduction de programmes de haut niveau ou par la réduction d'une telle configuration (issue de la traduction). Or, les programmes ML-Act et Erlang utilisent un protocole de création d'acteurs qui garantit l'unicité de son adresse. Ainsi, le sous-ensemble des configurations que nous manipulons vérifie la propriété de linéarité.

Cette notion de configuration fournit donc un cadre très général pour l'expression de langages de haut niveau auxquels on souhaite intégrer la notion d'acteur. En effet, elle se focalise sur les notions de nom et de communication par envoi de messages.

Ainsi par exemple, le protocole d'accès à la boîte aux lettres d'un acteur ne fait pas partie de la sémantique des configurations. Les deux langages que nous étudierons utilisent des règles différentes pour déterminer le premier message à traiter. Il est ainsi possible de définir des protocoles d'accès relativement complexe et par exemple d'introduire des priorités de traitement. Par exemple, en Erlang, tous les messages de la queue sont filtrés par le premier motif avant de passer au second. Ainsi, si la première branche traite des messages étiquetés *Administration* par exemple, de tels messages seront traités en priorité par rapport aux autres messages.

Les configurations fournissent donc un formalisme simple permettant d'exprimer la sémantique concurrente d'un langage de manière relativement aisée et modulaire. Il devient alors possible de réutiliser des fragments de sémantique (et de système de type) mais aussi de comparer des langages de haut niveau relativement différents. Ce qui sera fait dans la suite de cette thèse avec ML-Act et Erlang.

1.3 La réduction concurrente des configurations

Le cœur de la sémantique de notre modélisation des acteurs est décrit par la réduction des configurations (que nous appelons *réduction concurrente*) notée $\longrightarrow$ et qui est définie par les règles 1.4, 1.5 et 1.6.

Avant d'expliquer précisément chaque règle de réduction, il convient d'exposer rapidement un certain nombre de conventions que nous supposons respectées par la partie fonctionnelle spécifique ($\mathcal{E}xp$). La réduction $\longrightarrow_e$ conduit au terme représentant les différentes erreurs **Err** si le calcul fonctionnel comporte une erreur. De plus, chaque langage fonctionnel verra sa réduction exprimée au moyen de la notion de *contexte d'évaluation*. Cette notion classique due à M. Felleisen et al. (voir [FFKD87, Fel87]) permet de spécifier la suite d'un calcul. La forme des contextes qui contraint l'ordre d'évaluation des sous-expressions sera définie précisément pour chaque langage fonctionnel. Il est cependant nécessaire pour expliquer le sens des différentes règles de la réduction concurrente d'en donner une présentation abstraite.

Définition 1.7 (Contexte d'évaluation) :
Un contexte d'évaluation est une fonction qui associe à une expression une autre expression. Il est noté C[] *et son application à une expression e est notée* C[e].

Intuitivement, un contexte sera une expression dans laquelle figure un (unique) *trou*.

$$\textsc{Cong}: \frac{w_1 \equiv w_1' \quad w_1' \longrightarrow w_2' \quad w_2' \equiv w_2}{w_1 \longrightarrow w_2} \qquad \textsc{Par}: \frac{w_1 \longrightarrow w_2}{w||w_1 \longrightarrow w||w_2}$$

$$\textsc{Res}: \frac{w_1 \longrightarrow w_2}{\nu a.w_1 \longrightarrow \nu a.w_2} \qquad \textsc{Exp}: \frac{e_1 \longrightarrow_e e_2}{\alpha \rhd e_1 \longrightarrow \alpha \rhd e_2} \qquad \textsc{ExpE}: \frac{e \longrightarrow_e \mathbf{Err}}{\alpha \rhd e \longrightarrow \mathbf{Err}}$$

$$\textsc{Rcv}: \frac{}{\langle a \,|\, q \rangle \rhd e \parallel a \lhd m \longrightarrow \langle a \,|\, q :: m \rangle \rhd e} \qquad \textsc{Rea}: \frac{\mathcal{R} \prec q \Longrightarrow q',\ e,\ \sigma}{\langle a \,|\, q \rangle \rhd \mathcal{R} \longrightarrow \langle a \,|\, q' \rangle \rhd \sigma(e)}$$

Règles 1.4 Les réductions concurrentes (I).

L'application d'un contexte C[] à une expression e exprimera le fait que le trou du contexte est rempli par e. Cette notion permet donc de mettre en valeur et suivre l'évolution au cours de la réduction d'une sous-expression particulière d'une expression. Ainsi, le contexte exprime la continuation du processus d'évaluation.

Enfin, nous imposons à la partie fonctionnelle de contenir les fonctions d'acteurs suivantes : *send* pour l'envoi de message, *new* pour créer un nom, *init* pour créer un acteur, *become* pour changer un comportement et *receive* pour accéder à la boîte aux lettres. Notons immédiatement que les deux dernières fonctions sont spécifiques à chacun des langages que nous étudierons : *become* pour ML-Act et *receive* pour Erlang. Le cadre reste général puisque ces deux méthodes d'accès à la boîte aux lettres couvrent les approches possibles : l'*accès asynchrone* et l'*accès synchrone*. En effet, le *become* ne provoque un nouvel accès qu'indépendamment du reste des calculs alors qu'un *receive* provoque un accès immédiat et renvoie le résultat de cet accès. Le *become* confère un caractère réactif à l'acteur puisqu'il suit un cycle précis : attente d'un message puis traitement de ce message et ce indéfiniment. Alors que le *receive* permet une programmation plus générale et plus proche de la notion de processus puisque l'entité peut calculer indépendamment des messages reçus.

Les sept premières règles 1.4 décrivent les réductions sur les configurations. Tout d'abord, la réduction s'applique sur l'ensemble des configurations quotienté par la congruence que nous avons définie précédemment (voir règles 1.3 page 31). Cette propriété est décrite par la règle (Cong). Les configurations concurrentes peuvent se réduire de manière indépendante (Par) et la réduction est possible sous la portée d'une introduction de nom (Res). Ces deux règles permettent de donner des règles plus concises pour toutes les autres réductions puisque nous n'aurons pas à faire intervenir la configuration englobante en cours d'évaluation. Le calcul fonctionnel est également indépendant de la configuration dans laquelle il a lieu. On peut donc à tout moment réduire un acteur qui exécute une expression e réductible. Si le calcul se déroule correctement (Exp), l'acteur qui l'exécute effectue un pas de réduction. Les réductions fonctionnelles sur certains sous-termes d'une configuration et les réductions concurrentes sur d'autres peuvent donc être entrelacées. Par contre, si une erreur fonctionnelle survient (ExpE), elle est propagée vers la configuration englobante. Lorsqu'un message destiné à un acteur arrive, il est ajouté dans la boîte aux lettres (Rcv). Notons que l'opération $q :: m$ ne préjuge en rien de l'ordre qui existera sur les messages de la boîte aux lettres. Enfin, lorsque l'acteur a fini d'exécuter sa

$$\textsc{Send}: \frac{}{\alpha \triangleright \mathrm{C}[send(m,a)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel a \triangleleft m} \qquad \textsc{SendE}: \frac{v_1 \notin \mathcal{M}ess \vee v_2 \notin \mathbb{A}}{\alpha \triangleright \mathrm{C}[send\ (v_1, v_2)] \longrightarrow \mathbf{Err}}$$

$$\textsc{New}: \frac{a \notin \mathcal{FN}(\alpha \triangleright \mathrm{C}[new])}{\alpha \triangleright \mathrm{C}[new] \longrightarrow \nu a.(\alpha \triangleright \mathrm{C}[a])}$$

$$\textsc{Init}: \frac{}{\alpha \triangleright \mathrm{C}[init(a,v)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel \langle a \,|\, \varnothing \rangle \triangleright v\ a} \qquad \textsc{InitE}: \frac{v_1 \notin \mathbb{A} \vee v_2 \notin \mathbb{A} \rightarrow \mathbb{I}}{\alpha \triangleright \mathrm{C}[init\ (v_1, v_2)] \longrightarrow \mathbf{Err}}$$

Règles 1.5 Les réductions concurrentes (II).

réaction précédente, il traite le *premier message acceptable* de sa boîte aux lettres (règle Rea). Rappelons que $\mathcal{R}$ est une interface et décrit le filtrage des messages et les réactions associées. Pour déterminer le message à traiter, nous utilisons l'opérateur (abstrait) $\prec$ de parcours de queue.

Définition 1.8 (Accès à la boite aux lettres) :
On suppose l'existence d'un opérateur abstrait $\prec$ qui fournit, à partir d'une queue de messages q et d'une interface $\mathcal{R}$, une queue de messages q', une expression e et une substitution σ (une fonction qui construit une expression à partir d'une expression). Son application est notée $\mathcal{R} \prec q \Longrightarrow q',\ e,\ \sigma$.

Intuitivement, e est la réaction au premier message de q *traitable*[19] par l'interface $\mathcal{R}$, q' la queue des messages restants (le message traité n'est plus dans q) et la substitution σ résultant du filtrage du message. Nous supposons donc que les sémantiques données aux calculs fonctionnels manipulent les variables par le biais de substitutions.

Notons que la présentation par règle d'induction suppose que l'on a construit l'arbre de preuve de toutes les hypothèses d'une règle pour pouvoir l'appliquer. Donc, si la queue d'un acteur est vide ou si elle ne contient aucun message acceptable, la règle (Rea) n'est pas applicable.

Les cinq règles 1.5 et les deux couples de règles 1.6 présentent la sémantique des opérations issues du modèle d'acteurs. Il s'agit de δ-règles exprimant le sens des fonctions : *send*, *new*, *init*, *become* et *receive*. Elles décrivent le comportement des seules opérations fonctionnelles qui interagissent avec la configuration qui les englobent. Dans ces règles, nous utilisons les notations de valeurs sémantiques (a et v) pour indiquer que l'appel de ces fonctions prédéfinies se fait par valeur. Ces règles ne s'appliquent donc que lorsque les paramètres ont tous été évalués.

Les deux opérations (*send* et *init*) ne peuvent être appliquées que sur certaines formes d'arguments. Chaque règle exprimant sa sémantique va donc de pair avec la règle décrivant son utilisation erronée. Examinons plus précisément les trois opérateurs classiques du modèle d'acteurs :

19. Erlang et ML-Act ont des politiques différentes vis-à-vis de la queue, cette notion de message *traitable* et donc celle d'opérateur de parcours sera définie formellement dans le cadre spécifique de chacun des langages étudiés.

$$\textsc{Bec}: \frac{}{\alpha \triangleright \mathrm{C}[become(a, \mathcal{R})] \longrightarrow \alpha \triangleright \mathcal{R} \parallel \star \triangleright \mathrm{C}[nop]} \qquad \textsc{BecE}: \frac{v_1 \notin \mathbb{A} \vee v_2 \notin \mathbb{I}}{\alpha \triangleright \mathrm{C}[become(v_1, v_2)] \longrightarrow \mathbf{Err}}$$

$$\textsc{Rec}: \frac{\mathcal{R} \prec q \Longrightarrow q',\ e,\ \sigma}{\langle a \,|\, q \rangle \triangleright \mathrm{C}[receive(\mathcal{R})] \longrightarrow \langle a \,|\, q' \rangle \triangleright \sigma(\mathrm{C}[e])}$$

Règles 1.6 Les réductions concurrentes (III).

- Tout d'abord l'envoi de message (SEND et SENDE) n'est correct que si l'on envoie bien un message à un acteur. Si c'est le cas, le message est ajouté dans le médium, simulant ainsi son transit jusqu'à son destinataire éventuellement via le réseau. Le résultat d'une telle opération est alors *nop*.
- La création d'un nom d'acteur par l'instruction *new* (NEW) consiste en la création d'un nouveau nom a (via ν). Le résultat de cette expression est la nouvelle adresse ainsi créée. Cette opération est donc comparable à une allocation, un nom est réserve pour une utilisation ultérieure.
- Ensuite, l'acteur doit être créé et recevoir son comportement initial, par la fonction *init*. Ses arguments doivent être une adresse a et un comportement v – *i.e.* une fonction prenant en argument une adresse (l'*ego*) et renvoyant une interface (sinon INITE). La fonction *init* provoque (INIT) l'ajout dans le médium d'un nouvel acteur de nom a, de queue vide et de corps l'application de v sur le nouvel ego (a) .

La création du nom d'un acteur et la création de l'acteur proprement dites sont séparées afin de permettre la création *récursive* d'acteurs se connaissant mutuellement. Il est ainsi possible d'allouer un groupe de noms d'acteurs et ensuite de créer des acteurs en utilisant tous les noms du groupe (ils se connaissent alors mutuellement).

Enfin, les trois dernières règles 1.6 concernent le changement de comportement et l'accès à la boîte aux lettres (opérateurs *become* et *receive*). Ces deux opérations différentes sont nécessaires car ML-ACT et ERLANG ont une stratégie très différente d'accès aux boîtes aux lettres.

En ML-ACT, le changement de comportement doit prendre en argument une adresse et une interface (sinon BECE). Si c'est le cas (BEC), les calculs en cours sont interrompus pour assumer la nouvelle interface, la poursuite des calculs est alors réalisée par un acteur anonyme (*i.e.*, de nom $\star$). Notons que cet acteur anonyme n'ayant pas d'adresse disparaîtra une fois son calcul terminé et qu'un changement de comportement n'aura ainsi aucun effet. Cette notion de changement de comportement manipule la boîte aux lettres de manière asynchrone (le calcul se poursuit) et est une implantation directe du modèle d'acteur. L'accès à l'adresse de l'acteur courant a été ajouté pour vérifier que le *become* n'est pas utilisé illégalement (*i.e.* hors d'un acteur) et il sert lors du typage à relier le type des futures interfaces à celui de l'acteur. Nous reviendrons plus en détail sur ce point lorsque nous introduirons ML-ACT.

En ERLANG, il est nécessaire d'effectuer une extension du modèle d'acteur pour exprimer la sémantique d'accès à la boîte aux lettres. En effet, la récupération d'un message dans la boite aux lettres peut influencer la suite du calcul et peux bloquer le calcul en

cours. On peut ainsi décrire l'opération *receive* comme un accès synchrone à la boîte aux lettres. La consultation de la queue due à un *receive*(REC) est similaire à celle de la règle REA et le résultat de cet accès fournit alors la continuation du calcul. Notons que la substitution résultant de la réception d'un message s'applique à tout le contexte courant car la liaison en ERLANG n'est pas lexicale mais dynamique (voir la section 7.3.0 page 166).

1.4 Discussion

Dans ce chapitre, nous avons introduit le modèle d'acteur en présentant ses diverses caractéristiques et nous avons évoqué l'influence qu'il a eue dans la conception de nombreux langages concurrents. Ensuite, les configurations, un formalisme similaire à un calcul de processus a été introduit. Enfin, nous avons utilisé les configurations pour donner une sémantique générale de haut niveau au modèle d'acteur indépendante des aspects fonctionnels du langage.

Comme nous l'avons présenté dans l'introduction du chapitre, la mise au point de la notion de configuration est un compromis entre un calcul de processus (qui est trop faible pour servir de base à nos systèmes de type) et un langage de programmation (qui est alors trop complexe à manipuler). Nous pensons que l'abstraction de la partie fonctionnelle [20] permet d'*adapter* simplement notre modèle à une sémantique différente (de celle de ML-ACT ou d'ERLANG) relativement simplement. En effet, la sémantique plutôt complète des configurations simplifie et clarifie les interactions entre les éléments concurrents d'un programme et ses éléments purement calculatoires. On obtient ainsi plus facilement des analyses (telles que celles que nous allons présenter dans la suite) combinant ces deux aspects.

Notons que le prix à payer, pour obtenir le niveau de généralité suffisant, est un léger éloignement du modèle des acteurs puisque le formalisme ne contient plus la notion de comportement. Cependant, comme nos travaux portent sur l'analyse de la communication, il suffira que les parties fonctionnelles ajoutées pour construire un langage contiennent une notion d'interface de communication.

Plusieurs approches ont été utilisées dans les différents travaux cherchant à formaliser la sémantiques des acteurs.

Historiquement les premières sémantiques du modèle d'acteur sont des sémantiques dénotationnelles. W. CLINGER et C. BOTELLA dans leurs thèses (respectivement [Cli81] et [Bot82]) définissent l'exécution des acteurs par des notions de diagrammes d'évènements. Toutes les actions que peuvent effectuer les acteurs sont des évènements et chaque acteur correspond alors à une suite de tels évènements. Cette formalisation permet une définition relativement intuitive de la sémantique d'un langage d'acteurs mais l'abstraction faite ne se prête pas du tout à la mise au point d'analyses statiques.

La seconde approche suivie consiste à modéliser l'exécution des acteurs par les dérivations d'une formule logique. Dans ce but, deux formes de logique ont été utilisées. D'une part, S. SCHACHT modélise le comportement des acteurs dans une logique temporelle et peut ainsi prouver des propriétés temporelles sur les programmes d'acteurs (voir

20. Elle pourrait également être impérative.

[Sch95] et [Sch96]). D'autre part, N. KOBAYASHI et A. YONEZAWA dans [KY94a, KY94b] et J. DARLINGTON et Y. GUO dans [DG94] utilisent la logique linéaire pour exprimer l'exécution des acteurs. Ces deux modélisations basées sur la logique linéaire reposent sur un principe similaire mais ne vise pas les mêmes objectifs. J. DARLINGTON et Y. GUO construisent une traduction dans un formalisme assez simple et ouvre ainsi une étude potentielle des programmes d'acteurs. Ils démontrent ainsi l'utilisation d'un fragment de logique linéaire (qu'ils ont conçu dans le cadre de la programmation logique contrainte) pour la modélisation de la concurrence. L'approche de N. KOBAYASHI et A. YONEZAWA est sensiblement différente puisque leur objectif est proche du nôtre. En effet, ils souhaitent typer un langage concurrent à objets, voisin du modèle d'acteurs. Ils construisent pour cela une logique (issue de la logique linéaire) ACL pour laquelle ils définissent un système de type *à la ML* et un algorithme de reconstruction des types. Ainsi, par traduction d'un programme en ACL, ils peuvent inférer des types et donc vérifier la correction des programmes. Cependant, leur approche est limitée aux acteurs ne changeant pas de comportement et la logique construite est plutôt complexe.

Enfin, à notre connaissance les travaux les plus approfondis sur la sémantique des acteurs sont ceux de G. AGHA et al. dans [Agh84, Agh90, AMST92, AMST97]. Ils définissent la notion de configuration [21] et introduisent une transition étiquetée pour décrire sa sémantique. Une configuration, dans leur sens, est proche de notre forme normale de configuration, c'est un quadruplet composé d'un ensemble d'acteurs (qui peuvent calculer [22], attendre un message ou bien être non-initialisé), d'un multi-ensemble de messages en transit et de deux ensembles de noms d'acteurs (celui des réceptionnistes et celui des acteurs extérieurs). Ils distinguent ainsi une notion de groupes (d'acteurs) qui ne peuvent communiquer entre eux qu'au travers d'un ensemble déclaré de *ports*. Ces ensembles d'adresses, permettant les communications extérieures, sont bien sûr dynamiques. Par exemple, si un acteur envoie son adresse à un acteur extérieur, il devient réceptionniste. Une fois cette sémantique opérationnelle construite, ils étudient alors systématiquement l'équivalence de programmes d'acteurs. Leur notion de configuration permet de construire une sémantique compositionnelle du modèle d'acteurs mais la présence des groupes et des ensembles d'adresses associées rend leur manipulation complexe. L'approche opérationnelle par réduction que nous avons suivie paraît plus intuitive pour exprimer les sémantiques concurrentes et cette méthode ne nécessite pas la maîtrise d'un formalisme logique complexe.

Notons que le but de cette thèse étant l'analyse statique, nous ne nous sommes pas intéressés à l'étude algébrique des configurations à la manière des études du π-calcul existantes ou bien des travaux de G. AGHA suscités.

Par contre, nous avons souhaité intégrer tous les aspects des langages que nous allons étudier pour bien en définir la sémantique et donc l'approximation que nous en ferons. Par exemple, nous avons conservé les boîtes aux lettres dans la syntaxe des configurations alors qu'elles ne figurent pas dans la syntaxe de CAP, afin de mieux expliciter leur fonctionnement. Notons que cela nous permet également de changer leur sémantique à l'intérieur du formalisme des configurations.

21. Nous leur avons emprunté le terme de configuration.
22. Dans un langage fonctionnel proche de LISP.

Par rapport aux différentes approches précédentes, nous pensons que le formalisme des configurations par sa simplicité et sa modularité fournit un outil plus simple pour la conception, l'étude et la preuve de systèmes de type.

Après cette présentation du modèle d'acteurs et de sa formalisation par les configurations, le chapitre suivant analyse et classe les messages d'une application pour définir de façon précise les erreurs de communication que nous traquons.

Chapitre 2

Erreurs

Dans ce chapitre, nous allons évoquer plus précisément la notion d'erreur dans le cadre du modèle d'acteur. Pour cela, nous introduirons les différentes catégories d'erreurs que nous souhaitons détecter. Nous conclurons ce chapitre par une brève discussion sur les différentes méthodes et solutions actuellement utilisées pour faire face aux erreurs.

Le coût important de la présence d'erreurs au sein d'un logiciel a déjà été évoqué dans l'introduction de cette thèse. Nous avons également signalé le fait qu'il est, à l'heure actuelle, impossible de garantir qu'un logiciel ne contienne pas d'erreur. Parmi les méthodes formelles utilisées pour essayer de valider les logiciels, la plus utilisée est le typage. Cette méthode souvent assez simple et efficace pour des erreurs simples est principalement exploitée dans le domaine des logiciels séquentiels. Le but de nos travaux est d'étendre cette détection par typage au monde concurrent.

Dans le cadre des langages et logiciels concurrents, les erreurs qui peuvent survenir sont de deux formes : les *erreurs fonctionnelles* et les *erreurs concurrentes*.

La première famille d'erreurs correspond aux erreurs usuelles des langages séquentiels (qu'ils soient fonctionnels ou impératifs), une valeur est manipulée dans un contexte erroné. Par exemple, le programmeur essaie d'additionner une chaîne de caractères et un entier ou essaie d'utiliser un entier comme une fonction. Ce sont des erreurs de calculs ou d'appels de fonctions qui sont généralement détectées par des systèmes de type. Nous ne présenterons pas plus en détail cette forme d'erreurs en raison de son caractère usuel.

La seconde famille d'erreurs est moins courante, il s'agit d'une mauvaise utilisation d'une interface de communication d'un acteur. C'est-à-dire, un message est émis vers un acteur qui ne saura jamais le traiter, ou bien son destinataire saura le traiter mais les arguments du message ne correspondent pas aux arguments attendus [23]. Ce sont des erreurs de non respect des protocoles de communication entre entités. Notons que cette famille d'erreurs englobe la notion de *message non compris* (« *message not understood* ») des langages à objets. La détection de cette forme d'erreurs est l'objectif principal de nos travaux nous allons donc la détailler dans ce chapitre.

23. Ce problème ne peut survenir que dans le cadre de ML-Act.

2.1 Les erreurs concurrentes

Dans le contexte de la programmation objet, une erreur classique de programmation consiste à vouloir appliquer sur un objet, une méthode que celui-ci ne possède pas. Par exemple, soit un objet p représentant un point et dont le code est $\{x : int;\ get = x;\ set = \lambda v.x \leftarrow v\}$ [24]. Si un objet client de p essaie d'appliquer à p une méthode d'affichage ($p.afficher$) cela doit conduire à une erreur puisque p n'a pas de méthode d'affichage. On doit alors obtenir une erreur d'exécution du type : *méthode non comprise*. Pour détecter statiquement cette forme d'erreur, le compilateur associe à chaque objet l'ensemble des méthodes qu'il sait exécuter et vérifie lors d'un appel de méthode qu'elle figure dans cet ensemble. Cette vérification peut être statique ou dynamique. Si nous nous plaçons dans le cadre d'une détection statique par typage, p a pour ensemble des méthodes $\{get;\ set\}$. En termes de typage, cela signifie que son type est $\{get : int;\ set : int \rightarrow unit\}$. Le compilateur vérifie alors immédiatement que $p.afficher$ est une erreur de type et rejette le programme.

Dans le contexte des acteurs, le problème est beaucoup plus complexe. En effet, les acteurs changent dynamiquement de comportements, l'ensemble des méthodes évolue donc dans le temps et ne peut pas être construit statiquement de manière exacte. En fait, notre système de type va approximer cette évolution. Plusieurs degrés d'approximation sont possibles et leur précision est déterminante dans la qualité du système de type.

L'aspect dynamique et non déterministe de cette forme d'erreur rend le problème ardu. En effet, lorsqu'un acteur reçoit un message qu'il ne sait pas traiter, il n'est pas possible de déterminer simplement s'il ne pourrait pas le traiter plus tard. Pour simplifier le problème certains systèmes ont adopté une politique plus drastique vis-à-vis de ces messages :

- Certains considèrent qu'il s'agit d'une erreur même si un comportement futur aurait pu réagir à ce message. Cette politique est suivi par différents systèmes d'acteurs ou à objets concurrents. G. Agha présente dans [Agh86] un modèle des acteurs dans lequel il considère que la réception d'un message, que son destinataire ne peut traiter au moment de son arrivée, est une erreur. Dans le calcul d'objets que V. Vasconcelos et M. Tokoro introduisent dans [VT93], ils considèrent également cet événement comme une erreur. Dans le cadre des langages d'acteurs, *Plasma II* considère également ces messages comme des erreurs. Ils sont envoyés à un acteur spécial appelé *Complaint* qui les affiche (c'est une sorte de trace des erreurs). Notons, que *Plasma II* offre la possibilité au programmeur de redéfinir le comportement de l'acteur *Complaint*. Il est ainsi possible de provoquer le renvoi du message reçu vers son expéditeur et simuler ainsi le fait que l'acteur le remet dans sa boîte aux lettres (au prix d'une forte augmentation du trafic de messages et de la perte de l'ordre de réception initial).
- L'autre solution extrême est de ne jamais les considérer comme des erreurs. C'est-à-dire que les messages restent dans l'environnement ou dans la boîte aux lettres de l'acteur jusqu'à ce que celui-ci assume un comportement capable de les traiter. Il est donc possible que ces messages restent indéfiniment dans le médium ou dans la boîte aux lettres d'un acteur. C'est l'approche choisie par N. Kobayashi et

24. Dans une syntaxe de type fonctionnelle.

A. Yonezawa pour leur langage d'objets concurrents HACL dans [KY94b], par G. Boudol pour le calcul bleu introduit dans [Bou97a] et par l'équipe du join-calcul [FGL+96]. Les langages de programmation ML-Act et Erlang que nous étudierons dans la troisième partie de cette thèse utilisent également cette stratégie vis-à-vis des messages non-compris.

Nous jugeons que la première approche est trop restrictive car elle impose un grand degré de synchronisation entre les acteurs. En effet, avant d'envoyer un message à un acteur, il faut être sûr qu'il aura atteint l'état adéquat. Par exemple, si nous considérons un *périphérique*[25] modélisé par un acteur d qui a deux comportements :

- un premier, *non-initialisé*, qui accepte uniquement le message *init* et
- un second, *en fonctionnement*, qui traite des messages *requête*.

Si le système considère que l'envoi d'une requête à d est une erreur si d n'est pas encore initialisé, alors, un acteur de l'application désirant utiliser d devra vérifier son état et éventuellement l'initialiser avant toute requête (par un mécanisme que nous ne préciserons pas ici). Nous pensons qu'il est plus pratique de pouvoir envoyer des requêtes à d quitte à ce qu'elles soient différées si d n'est pas encore en fonctionnement jusqu'à son éventuel démarrage (qui peut être dû à un tiers).

La seconde approche est par contre trop permissive. En effet, elle laisse, dans le médium ou dans les boîtes aux lettres, des messages inutiles (ils ne seront peut être jamais traités) sans signaler d'erreur ce qui risque d'engorger les boîtes aux lettres des acteurs du système. Par exemple, si quelqu'un cherche à envoyer des requêtes à un *périphérique* en maintenance (qui ne sera donc pas initialisé dans un futur proche), il est peut être pertinent d'en informer l'auteur.

Nous pensons que, si un message ne peut pas être traité par un acteur, c'est que le programmeur a probablement commis une erreur. Il faut alors l'en informer statiquement. Notre position est donc médiane entre ces deux approches : le système doit être suffisamment souple pour faciliter la tâche des programmeurs, mais une analyse statique doit indiquer les messages qui risquent de ne pas pouvoir être traités par leur cible. Notre objectif est donc de développer des théories et des techniques pour informer au mieux le programmeur sur ces *messages potentiellement problématiques*. Afin de mieux cerner cette notion de messages qui risquent de provoquer des erreurs, nous allons essayer de classer les messages.

2.2 Une taxinomie des messages

Cette classification des messages va être construite par raffinages successifs. Nous allons définir formellement les messages qui peuvent éventuellement provoquer des erreurs lors de l'exécution. Puis, nous diminuerons les contraintes pour arriver à construire un sous-ensemble de ces messages erronés que nous pourrons détecter statiquement.

Dans toute la suite de cette section, lorsque nous parlons d'un message m pour décrire tous les messages qui ont la même forme que m (il peut être instancié plusieurs fois dans un même terme). Un acteur peut donc recevoir un même message m de nombreuses fois.

25. Par exemple, une imprimante.

Nous emploierons le terme de *configuration* pour désigner un terme décrivant une configuration isolée, c'est-à-dire, ne recevant pas de messages de l'extérieur. Nous utiliserons *sous-configuration* pour les sous-termes d'une *configuration*.

Si l'on se place dans le cadre des configurations du chapitre précédent, la réduction d'un terme t peut conduire à deux situations :

- la réduction se termine en un terme t' sur lequel plus aucune réduction ne peut avoir lieu : $t \longrightarrow^* t' \not\longrightarrow$.
- la réduction boucle. Il est aisé d'écrire une telle configuration qui se réduit à l'infini. Par exemple, une configuration qui contient un acteur a et un message m en transit vers a. Le comportement de a est d'attendre un message m et à sa réception de réagir en s'envoyant un message m et en réassumant ce comportement [26]. En CAP, une telle configuration pourrait s'écrire : $a \triangleleft m \parallel a \triangleright [m = \zeta(e,s).(e \triangleleft m \parallel e \triangleright s)]$.

Remarques :

Notons que cette divergence est régulière, c'est-à-dire que le nombre de termes différents apparaissant au cours de la réduction d'un tel acteur est fini. Mais, ce n'est pas un caractère général puisqu'il est également assez simple de construire des acteurs divergents irrégulièrement. Par exemple, un acteur semblable au précédent qui s'envoie deux messages m à chaque réception fournit un acteur se réduisant indéfiniment et dont la réduction produit une infinité de termes différents. Cependant, le nombre de sous-termes différents entre deux étapes de la réduction dans la configuration reste fini (et la différence est toujours la même).

Les calculs fonctionnels peuvent éventuellement boucler indéfiniment, ce qui introduit un nouveau cas pathologique dans notre étude. Étant donné que notre centre d'intérêt est la communication, nous ignorons de tels termes.

La relation de réduction étant indéterministe, une configuration peut avoir de nombreuses réductions différentes. Formalisons la notion d'exécution d'une configuration :

Définition 2.1 (Dérivation) :
*Nous appelons **dérivation** d d'une configuration w toute suite $(w_n)_{n\in\mathbb{N}}$ de configurations vérifiant les deux propriétés suivantes :*

(1) $w_0 = w$ *et*

(2) $\forall n \in \mathbb{N} \quad (w_n \longrightarrow w_{n+1}) \vee (w_n \not\longrightarrow \wedge\, \forall k > n : w_n \equiv w_k)$

La n^e configuration de la suite d est notée de manière fonctionnelle : $d(n)$ et l'ensemble des dérivations de la configuration w est noté $\mathcal{D}(w)$.

26. Nous n'avons pas encore décrit la partie fonctionnelle des configurations qui permettrait de construire un tel comportement. Celle-ci sera présentée dans les chapitres spécifiques à chaque langage étudié.

Une première définition

Une dérivation *finie* est une suite *stationnaire*, *i.e.* constante à partir d'un certain rang. La valeur alors atteinte est appelée configuration *limite*. Il est possible de décrire plus précisément la forme de ces configurations limites : ce sont des termes sur lesquels plus aucune réduction ne peut avoir lieu. Soit, intuitivement, une telle configuration peut être vue comme une configuration qui ne contient plus de message en transit et dont tous les acteurs sont bloqués en attente de message. Nous les appelons des *configurations silencieuses* :

Définition 2.2 (Configuration Silencieuse) :
*Une **configuration silencieuse** est une configuration qui représente un système d'acteurs qui ne peut plus se réduire. Elles correspondent aux termes construits par la grammaire suivante :*

$$s ::= \epsilon \mid \textbf{\textit{Err}} \mid \nu a.s \mid s \parallel s \mid \langle a \,|\, q \rangle \triangleright \mathcal{R}$$
$$q ::= \varnothing \mid m :: q$$

*qui vérifient la condition d'**Arrêt** ci-dessous :*

$$\langle a \,|\, q \rangle \triangleright \mathcal{R} \in s \Longrightarrow \mathcal{R} \not\prec q \qquad (\textbf{\textit{Arrêt}})$$

Dans les boîtes aux lettres des acteurs d'une configuration silencieuse, il peut rester un certain nombre de messages non traités. Nous appelons ces messages des *messages orphelins*. On propose ainsi une première définition de la notion de message orphelin.

Définition 2.3 (Message Orphelin (version 0)) :
Dans une configuration silencieuse, un message orphelin est un message présent dans la boîte aux lettres d'un acteur.

Ce comportement peut être considéré comme une erreur en ajoutant alors à la sémantique des configurations une règle de la forme :

$$\textsc{Orph}: \frac{q \neq \varnothing \qquad \mathcal{R} \not\prec q}{s \parallel \langle a \,|\, q \rangle \triangleright \mathcal{R} \longrightarrow \textbf{Orph}}$$

La configuration sujet de cette règle est silencieuse puisqu'elle est composée d'une sous-configuration silencieuse et d'un acteur silencieux (car la condition d'arrêt est supposée vérifiée).

Cependant, cette règle a quatre défauts :

- Le fait qu'un message n'ait pas pu être traité lors d'une exécution ne signifie pas forcément qu'il y ait une erreur dans l'application. Il peut s'agir d'un problème de désynchronisation, par exemple, si le message était arrivé plus tôt, il aurait pu être traité. Un tel comportement peut être souhaitable dans certaines applications : si une information arrive trop tard, elle ne doit plus être traitée.
- Elle ne prend pas en compte l'indéterminisme de l'exécution d'un programme d'acteur. Ainsi, dans une certaine exécution, un message peut être déclaré orphelin, et dans une autre, il peut ne pas l'être.

– Elle n'est pas modulaire, il est nécessaire de connaître la totalité du terme décrivant l'exécution en cours pour pouvoir l'utiliser.
– En fait, cette règle n'offre que de peu d'intérêts pratiques puisqu'elle suppose et attend l'arrêt des calculs pour déterminer l'ensemble des messages orphelins. Or, c'est statiquement, avant tout calcul que nous souhaitons fournir au programmeur ces informations. Et ainsi lui permettre d'évaluer si les messages non-traités sont dus à des erreurs de programmation de sa part ou bien s'il s'agit d'un comportement qu'il considère comme normal pour son programme.

Une classification formelle des messages

Nous allons donc essayer de rendre dynamique la notion de message orphelin, c'est-à-dire, de permettre la détection de tels messages au cours du calcul. Il sera ainsi possible de déterminer à partir d'un terme quelconque les messages orphelins. Pour cela, nous allons étudier plus en détail les messages vis-à-vis de leur destinataire et essayer de les classer. Notons qu'un message peut avoir un statut différent selon la dérivation suivie. Nous allons donc réaliser une taxinomie des messages par rapport à un acteur et à toutes les dérivations possibles d'une configuration.

Remarque :
Nous supposons que les configurations que nous manipulons sont linéaires, i.e. *un acteur n'a qu'un seul comportement à un instant donné. Il est alors possible de définir une fonction $Q(a, w)$ qui permet de récupérer la boîte aux lettres de l'acteur a dans la configuration w. Le résultat est, en effet, unique du fait de la linéarité des configurations. Par exemple, si w a pour forme normale $\nu a_1 \ldots a_n.(W_1 \parallel \ldots \parallel \langle a \,|\, q\rangle \triangleright e \parallel \ldots \parallel W_m)$ alors $Q(a, w)$ vaut q.*

Commençons par analyser ce qui peut se passer au cours d'une dérivation d. La première classe de messages qui vient à l'esprit est celle des messages qui *ne posent pas de problèmes*. Informellement, ces messages sont ceux qui seront toujours traités par leur destinataire. Si d est finie, ces messages ne sont contenus dans aucune des boîtes aux lettres des acteurs dans la configuration limite. Nous appelons ces messages : les messages *sûrs* par rapport à la dérivation d.

Définition 2.4 (Message sûr par rapport à une dérivation) :
*Un message m est **sûr** par rapport à une dérivation d pour l'acteur a de la configuration w si et seulement si :*

$$\forall n \quad (m \in Q(a, d(n)) \Rightarrow \exists n' \;\; n' > n \;\wedge\; m \notin Q(a, d(n')))$$

Intuitivement, un message est donc sûr s'il ne reste pas indéfiniment dans la boîte aux lettres d'un acteur, soit, en d'autres termes, s'il est traité par l'acteur en un temps fini. On pourrait penser un peu rapidement qu'un message m est sûr s'il n'apparaît plus dans la boîte aux lettres de son destinataire en temps fini :

$$\exists n_0, \quad \forall n > n_0 \;\; m \notin Q(a, d(n)) \qquad (\star)$$

Cependant, comme l'acteur peut recevoir m infiniment souvent, cette propriété ($\star$) est fausse. Le terme CAP de la page 44 est un exemple de terme ne vérifiant pas ($\star$) mais dans lequel m est sûr.

La seconde classe est la classe duale de la précédente, c'est-à-dire la classe des messages qui *posent problème*. Ce sont les messages qui vont indéfiniment rester dans la boîte aux lettres de leur destinataire. Ce sont les messages de cette classe que nous nommons maintenant messages orphelins par rapport à la dérivation d :

Définition 2.5 (Orphelin par rapport à une dérivation) :
Dans une configuration w, un message m destiné à un acteur a est un message ***orphelin*** *par rapport à une dérivation d si et seulement si :*

$$\exists n_0, \quad \forall n > n_0 \;\; m \in Q(a, d(n))$$

Notons que la définition donnée des orphelins est très restrictive et qu'elle englobe les messages qui sont traités infiniment souvent mais dont au moins un exemplaire figure toujours dans la queue. Par exemple, le message m est orphelin dans le terme suivant :

$$a \triangleleft m \parallel a \triangleleft m \parallel \langle a \,|\, \varnothing \rangle \triangleright [m = \zeta(e, s).(e \triangleleft m \parallel e \triangleright s)]$$

Ces définitions s'étendent à toutes les définitions :

Définition 2.6 (Message sûr et Orphelin) :
Un message m est ***sûr*** *par rapport pour l'acteur a de la configuration w si et seulement si il est sûr par rapport à toutes les dérivations de $\mathcal{D}(w)$.*
Un message m est ***orphelin*** *par rapport pour l'acteur a de la configuration w si et seulement si il est orphelin par rapport à toutes les dérivations de $\mathcal{D}(w)$.*

Si l'on se place dans le cadre d'acteurs ne changeant pas de comportement (*i.e.* dans le cadre des objets concurrents), ces deux classes de messages permettent de classer tous les messages. En effet, si un message m figure dans la boîte aux lettres d'un acteur a qui ne change pas d'interface, soit m peut être pris en compte et le sera (la boîte aux lettres respecte une certaine équité) soit il ne peut pas et ne le sera jamais. De plus, le comportement que l'on vient de décrire ne dépend pas de la dérivation choisie puisque l'acteur ne change pas d'interface. Cependant, le changement de comportement des acteurs rend la classification un peu plus complexe et nous oblige à analyser le cas des messages dont le comportement n'est pas le même suivant la dérivation choisie. Nous arrivons donc à une nouvelle définition, celle de la notion de messages orphelins potentiels.

Définition 2.7 (Orphelin potentiel) :
Dans une configuration w, un message m destiné à un acteur a est un message ***orphelin potentiel*** *si et seulement si il vérifie les deux conditions suivantes :*
- *il existe une dérivation d_1 de $\mathcal{D}(w)$ telle que m soit sûr par rapport à d_1 ;*
- *il existe une dérivation d_2 de $\mathcal{D}(w)$ telle que m soit orphelin par rapport à d_2.*

Si un acteur peut changer de comportement, l'enchaînement de ses différentes interfaces dépend de la dérivation choisie, la classe des messages dépend donc de la dérivation.

Or, les deux classes (sûr et orphelin) forment une partition de l'ensemble des messages $\mathcal{M}ess$ selon une dérivation. Il est alors clair que les trois définitions précédentes permettent théoriquement de classer les différents messages. Dans la section précédente, nous avons esquissé notre stratégie vis-à-vis des erreurs et de leur élimination. Le but de nos travaux est donc de déterminer statiquement si un message est orphelin ou orphelin potentiel. Mais, notre attitude n'est pas la même pour les deux classes. En effet, un message orphelin sera considéré comme un facteur d'erreur inévitable et donc comme un échec du typage. Alors que la découverte d'un orphelin potentiel sera portée à la connaissance du programmeur (par exemple sous la forme d'un « *warning* ») mais ne sera pas considérée comme une erreur fatale.

Une seconde définition

Si, nous passons maintenant à la pratique, nous sommes confrontés à un problème. En effet, dans le cas général, la construction statique en un temps fini de ces trois ensembles est impossible. Il va donc falloir essayer de les approximer au mieux en utilisant alors une notion de *potentiel*. Pour calculer ce potentiel, on utilise une fonction $\mathcal{I}(a, w)$ qui calcule l'ensemble des messages que peut traiter l'interface de l'acteur a dans la configuration w (de la même manière que $Q(a, w)$ calcule la boîte aux lettres de a). Si w a pour forme normale $\nu a_1 \ldots a_n.(W_1 \parallel \ldots \parallel \langle a \mid q \rangle \triangleright e \parallel \ldots \parallel W_m)$ et que e est une interface, alors $\mathcal{I}(a, w)$ vaut l'ensemble des filtres de messages de e (la première partie du couple).

Définition 2.8 (Potentiel Complet) :
Le ***potentiel complet*** *d'un acteur a dans une configuration w ($\mathcal{P}^*(a, w)$) est la concaténation de toutes les interfaces que cet acteur va assumer dans toutes les dérivations de $\mathcal{D}(w)$:*

$$\mathcal{P}^*(a, w) = \bigcup_{d \in \mathcal{D}(w)} \bigcup_{w' \in d} \mathcal{I}(a, w')$$

Le calcul du potentiel complet d'un acteur consiste donc en l'union des messages que ses différents comportements peuvent traiter. Il s'agit de collecter tous les messages que l'acteur pourrait traiter s'il figurait dans sa boîte aux lettres au bon moment et ce pour toutes les dérivations possibles. Son calcul statique est indécidable, car les expressions fonctionnelles des configurations contiennent des opérateurs de choix. Nous allons donc nous contenter du calcul d'une approximation supérieure du potentiel complet :

Remarque :
Le calcul du potentiel d'un acteur a dans une configuration w ne dépend que de son comportement (actuel) et pas du reste de w. Le calcul effectif du potentiel ne sera défini précisément que lorsque la syntaxe fonctionnelle aura été introduite.

Définition 2.9 (Potentiel) :
Le potentiel *d'un acteur a dans une configuration w est l'ensemble des messages qu'il* ***pourrait*** *traiter durant tous ses cycles de vie. Il s'agit de collecter tous les messages traitables de toutes les interfaces que* ***pourrait*** *assumer l'acteur indépendemment des choix effectués dans les filtrages. On note ce potentiel $\mathcal{P}(a, w)$.*

Par exemple, les deux potentiels valent $\mathcal{P}^*(a, w) = \{m\}$ et $\mathcal{P}(a, w) = \{m, p\}$ si w est $a \triangleleft m \parallel \langle a \,|\, \varnothing \rangle \triangleright [m = \zeta(e, s).(\text{if } c \text{ then } e \triangleright s \text{ else } e \triangleright [p = \zeta(e, s).e \triangleright s])]$ où c est une condition toujours vraie. En effet, l'acteur ne pourra jamais atteindre l'état traitant les messages p.

Le fait de classer un message comme orphelin (détectable) s'il ne figure pas dans le potentiel, permet d'obtenir une caractérisation des messages par rapport à un acteur. Cependant, cette classification est dynamique puisqu'elle dépend de l'état de l'acteur. En effet, lorsqu'un acteur change de comportement, il se peut qu'il ne puisse plus jamais traiter un message qu'il pouvait traiter auparavant. Cette classification dépend également de la dérivation choisie car un message peut être orphelin dans une exécution et pas dans une autre. L'acteur modélisant le local commercial introduit dans le chapitre précédent page 21 est un exemple de destinataire pour lequel la classification dépend de l'exécution choisie. En effet, si la banque déménage tous les messages qui lui sont destinés deviennent orphelins et si elle ne déménage pas, ils ne le seront pas.

Nous n'avons pas défini formellement le calcul du potentiel d'un acteur, mais nous pouvons en donner une intuition. Celui-ci consiste en l'union des interfaces de tous les *become* dans le cas de ML-ACT et de tous les *receive* dans le cas d'ERLANG. Nous présenterons plus en détail cette collecte des interfaces dans les chapitres spécifiques à chaque langage. Un aperçu rapide peut cependant en être donné :

- Dans le cadre de ML-ACT, les interfaces sont installées par l'expression de changement de comportement *become*. Il faut donc parcourir le code de l'acteur et relever toutes les interfaces qui sont arguments d'un *become*. De plus, comme nous le verrons lors de la présentation du langage, en ML-ACT, les comportements sont des valeurs comme les autres, l'argument d'un *become* peut donc être une variable. Il conviendra donc, durant ce calcul, de garder une trace des interfaces qui seront stockées dans des variables. Une des difficultés du calcul du potentiel en ML-ACT vient du fait qu'un comportement peut également être reçu dans le corps d'un message. On considérera alors que l'acteur peut assumer n'importe quel comportement. Dans ce cas, il y a deux possibilités, soit la configuration ne comporte aucun nom libre (*configuration close*), soit elle contient des noms libres (*configuration ouverte*). Si la configuration est close, n'importe quel comportement signifie toutes les interfaces de la configuration. Et, si elle est ouverte, cela signifie tous les messages ($\mathcal{M}ess$). L'acteur a alors un potentiel *ouvert*.
- Dans le cadre d'ERLANG, ce calcul est plus simple puisque les interfaces ne sont pas des valeurs manipulables. Il suffit de parcourir le code à la recherche des expressions *receive*. Cependant, comme les fonctions peuvent contenir des réceptions et que les fonctions sont manipulables, les potentiels en ERLANG pourront également être ouverts.

Il découle de cette description informelle du calcul du potentiel d'un acteur, que le potentiel ne peut que diminuer lors d'une exécution. En effet, si un acteur rencontre de nouvelles interfaces lors d'un calcul, c'est qu'elles proviennent de la réception d'un message. Or, dans ce cas, le potentiel est ouvert et les contient donc. Un acteur ne peut donc qu'oublier des comportements qu'il n'assumera plus et pas en découvrir de nouveaux. Donc, la suite des potentiels d'un acteur pour une dérivation donnée $(\mathcal{P}(a, w_i))_I$ est une suite décroissante.

Le critère qui permet de déterminer si un message est un orphelin (détectable) devient alors : dans une configuration w, un message m destiné à un acteur a est orphelin si m ne figure pas dans le potentiel de l'acteur : $m \notin \mathcal{P}(a, w)$. Or, ce potentiel ne dépend que du corps e de l'acteur (unique) de nom a dans w, il est noté $\mathcal{P}(e)$. La règle d'inférence qui complète la sémantique des configurations serait donc :

$$\textsc{Orph}: \frac{m \notin \mathcal{P}(\mathcal{R})}{a \triangleleft m \parallel \langle a \,|\, q \rangle \triangleright \mathcal{R} \longrightarrow \textbf{Orph}}$$

On peut également pour diminuer le calcul des potentiels étiqueter chaque interface par son potentiel : $\mathcal{R}_{\mathcal{P}}$, la condition devient alors $m \notin \mathcal{P}$. Cette indication du potentiel est alors calculée à la demande. C'est une technique similaire qu'utilise J-L. Colaço, dans sa thèse [Col97], pour le calcul Cap : il définit un calcul décoré où un acteur à la forme :

$$a \overset{\mathcal{P}}{\triangleright} [\ldots]$$

La formalisation définitive

Dans le cadre de notre objectif (la détection statique des messages orphelins), le potentiel qui nous intéresse est le premier potentiel de l'acteur (avant qu'il ne commence tout calcul). Ce potentiel particulier est nommé *potentiel initial* et noté $\mathcal{P}_0(a, w)$ pour être particularisé.

Comme le potentiel ne peut que décroître, un message qui ne figurait pas dans le potentiel initial ne peut se retrouver dans un des potentiels suivants. Donc, un message déclaré orphelin à un instant donné le restera. On définit donc une nouvelle classe de messages orphelins :

Définition 2.10 (Orphelin Trivial) :
*Dans une configuration w, un message **orphelin trivial** est un message m destiné à un acteur a qui ne figure pas dans le potentiel initial de l'acteur : $m \notin \mathcal{P}_0(a, w)$.*

Remarque :
Si un acteur à un potentiel initial ouvert alors aucun des messages qui lui sont envoyés n'est orphelin trivial.

Nous allons donc, lors de la création d'un acteur, l'étiqueter avec son potentiel initial. Ainsi, à chaque réception d'un message, nous pourrons vérifier si le message est un orphelin trivial. Pour cela, il faut modifier deux règles de la sémantique des configurations et en ajouter une pour détecter ces messages orphelins triviaux :

$$\textsc{Rcv}: \frac{m \in \mathcal{P}}{\langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m \longrightarrow \langle a \,|\, q :: m \rangle_{\mathcal{P}} \triangleright e} \qquad \textsc{RcvE}: \frac{m \notin \mathcal{P}}{\langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m \longrightarrow \textbf{Err}}$$

$$\textsc{Init}: \frac{}{\alpha \triangleright \mathrm{C}[\mathit{init}(a, v)] \longrightarrow \alpha \triangleright \mathrm{C}[\mathit{nop}] \parallel \langle a \,|\, \varnothing \rangle_{\mathcal{P}(v\ a)} \triangleright v\ a}$$

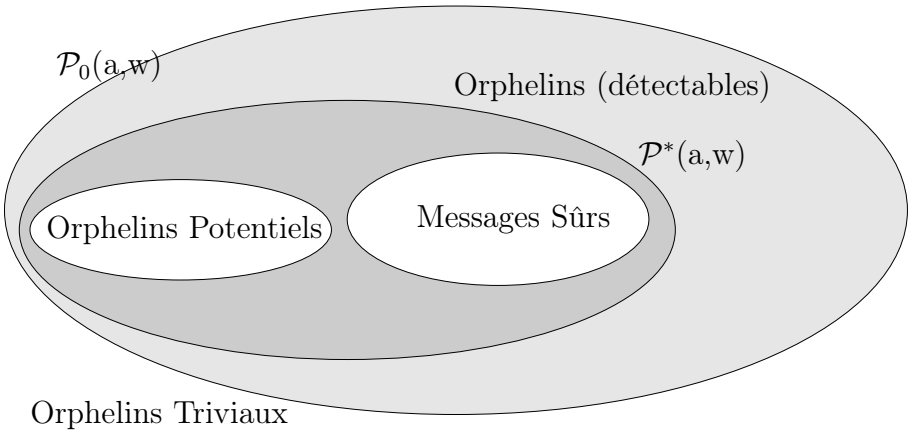


Figure 2.1 Les messages par rapport à un acteur a dans une configuration w.

De plus, le potentiel initial est une approximation supérieure du potentiel complet, il le contient donc : $\mathcal{P}^*(a, w) \subseteq \mathcal{P}_0(a, w)$. C'est-à-dire que nous intégrons au potentiel d'un acteur a plus de messages qu'il ne pourra effectivement en traiter, par concaténation de toutes les interfaces qu'il pourrait assumer. En effet, il est possible que l'acteur ne passe effectivement jamais par certains de ses états possibles (les interfaces potentiellement assumées).

La figure 2.1 schématise la caractérisation d'un message destiné à un acteur a d'une configuration w et ce par rapport à **toutes** les dérivations possibles de w. Dans ce cas, le statut d'un message est fixe et donc ne figure dans ce schéma que les notions indépendantes des dérivations. Nous avons vu que l'ensemble des messages peut alors être partitionné en trois parties :

- les messages sûrs,
- les messages orphelins et
- les messages orphelins potentiels.

Le plan est donc séparé en trois régions, deux figurent en blanc sur le schéma (les orphelins potentiels et les messages sûrs) et la troisième (les messages orphelins) recouvre le reste du plan. L'ensemble des orphelins est ensuite séparé en trois parties :

- les orphelins triviaux, ce sont les messages extérieurs au potentiel initial ($\mathcal{P}_0$(a,w)). Ils sont en blanc sur la figure et seront facilement éliminés par nos système de type ;
- les orphelins détectables, ce sont les messages extérieurs au potentiel complet de l'acteur ($\mathcal{P}^*$(a,w)). Sur la figure, ils correspondent à la zone gris clair, ils figurent dans le potentiel initial mais ne seront jamais traités. C'est l'ensemble des messages que nous allons essayer de déterminer le plus précisément possible en utilisant une approximation la plus fine possible de l'exécution d'un acteur ;
- les orphelins non détectables, situés dans la zone en gris foncé, sont éléments du potentiel complet mais ne seront jamais traités. Ces messages ne peuvent pas être détectés statiquement, *i.e.* leur détection est indécidable statiquement sans restreindre drastiquement l'ensemble des programmes acceptés. Cependant, il est possible de reporter la détection de tels messages (ainsi que celle de certains orphelins potentiels) à l'exécution.

Notons que nous sommes également confronté à l'indécidabilité dans la partie orphelins détectables. Considérons par exemple, un acteur a dont le comportement traite un message p et réagit à celui-ci en se renvoyant un message p et en choisissant entre réassumer son comportement courant et assumer un comportement qui saura traiter un message m. Soit, sous forme de terme CAP :

$$a \triangleleft m \parallel a \triangleright [p = \zeta(e,s).(e \triangleleft p \parallel \mathit{if\ pred\ then}\ e \triangleright s\ \mathit{else}\ e \triangleright [m = \ldots])]$$

où *pred* est un prédicat quelconque statiquement indécidable mais qui sera toujours vrai à l'exécution (en utilisant par exemple des comparaisons d'expressions arithmétiques). A tout moment de la dérivation, le potentiel de a contient p et m, donc lors de l'envoi du message m celui-ci ne sera pas déclaré orphelin. Or, a ne pourra jamais traiter m puisqu'il ne cesse de reboucler sur le comportement ne traitant que p. Donc, m est orphelin (au sens de la définition 2.5).

Cette analyse nous oblige à raffiner la notion de messages orphelins. Nous allons maintenant travailler dans le cadre d'**une** dérivation :

Définition 2.11 (Orphelin de sûreté) :
Dans une configuration w, un message ***orphelin de sûreté*** *est un message m qui n'appartient pas au potentiel de son destinataire a : $m \notin \mathcal{P}(a, w)$. Ce sont les messages détectés par la règle d'inférence* ORPH *présentée précédemment page 50.*

Définition 2.12 (Orphelin de vivacité) :
Dans une configuration w, un message ***orphelin de vivacité*** *est un message m contenu dans le potentiel de son destinataire a mais que a ne traitera pas.*

Soit, intuitivement, un message m destiné à l'acteur a est orphelin de sûreté si aucun des futurs comportements possibles de l'acteur a ne sait traiter le message m. Et, m est orphelin de vivacité si au moins un des futurs comportements possibles de l'acteur a sait traiter m, mais que a n'atteindra jamais l'un des comportements pouvant traiter m ou bien à chaque fois qu'il atteindra un tel comportement, il traitera un autre message plus *prioritaire*[27] de sa boîte aux lettres. Remarquons que cette situation ne peut se produire qu'un nombre fini de fois. Le terme CAP ci-dessous fournit un exemple de configuration dans lequel un message peut être orphelin de vivacité dans certaines exécutions :

$$a \triangleleft m_1 \parallel a \triangleleft p \parallel a \triangleleft m_2 \parallel a \triangleright [p = \ldots,\ \ m_1 = \zeta(e,s).e \triangleright s,\ \ m_2 = \zeta(e,s).e \triangleright [m_3 = \zeta(e,s).e \triangleright s]]$$

En effet, si l'acteur a reçoit les messages qui lui sont destinés dans l'ordre suivant : m_1, m_2 puis p, alors, p est orphelin de vivacité puisqu'il est dans le potentiel de l'acteur mais ne pourra pas être traité. Remarquons que ce classement peut être dynamique, par exemple, après le traitement des messages m_1 et m_2, le message p devient orphelin de sûreté puisqu'il n'est plus dans le potentiel de l'acteur. Pour fixer la classe d'un message orphelin, la classification entre orphelins de sûreté et de vivacité est faite au moment de la réception d'un message par son destinataire. Soit, dans l'exemple précédent, si a reçoit p avant de traiter m_2, p est orphelin de vivacité.

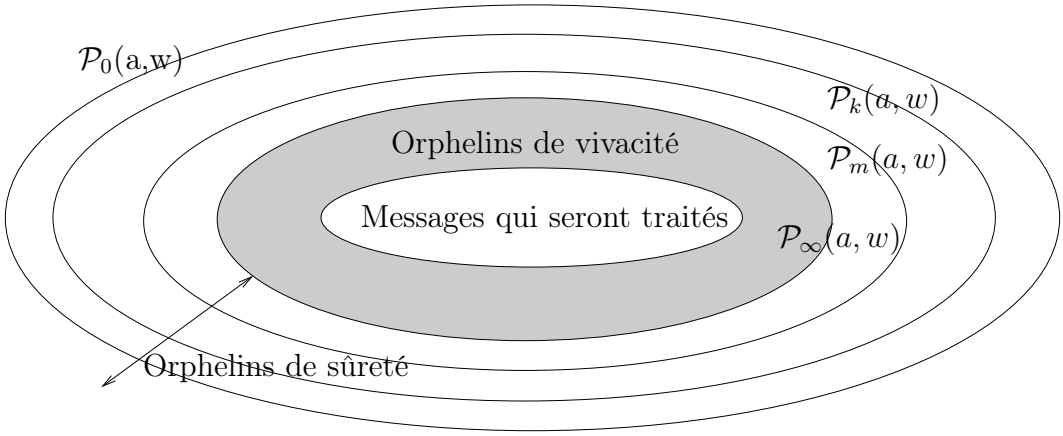


Figure 2.2 Les messages destinés à a dans une configuration w se réduisant indéfiniment.

Si l'exécution du programme se termine, tous les orphelins sont détectés par la règle ORPH de la page 45.

Si l'on se place dans le cas où l'exécution ne se termine pas (c'est le cas le plus courant dans le cadre des acteurs), le potentiel d'un acteur a converge vers une limite que nous noterons $\mathcal{P}_\infty$. En effet, le nombre d'interface que peut prendre un acteur étant fini (s'il ne reçoit pas un nombre infini d'interfaces de l'extérieur), l'automate décrivant les interfaces possibles de l'acteur finit par arriver dans un état puits ou alors dans un cycle d'états puits. Notons, que si a reçoit sans fin des interfaces venant du monde extérieur qu'il assume, son potentiel limite est ouvert et nos analyses ne pourront jamais détecter de messages orphelins.

Le schéma 2.2 décrit ce comportement dynamique d'un acteur a (qui figure dans une configuration qui se réduit indéfiniment). Celui-ci présente le potentiel initial de a, deux potentiels intermédiaires $\mathcal{P}_k$ et $\mathcal{P}_m$ ainsi que le potentiel limite (que l'on suppose clos). Les trois premiers potentiels sont calculés à des instants différents de la *vie* de l'acteur (à sa naissance puis aux instants k et m avec $m > k$). Les potentiels étant décroissants, l'ensemble des orphelins de sûreté (les messages extérieurs au potentiel courant) augmente jusqu'à ce que l'on atteigne le potentiel limite. L'ensemble des messages qui seront effectivement traités par a apparaît également sur la figure, c'est un ensemble qui diminue au fur et à mesure de l'avancement des calculs. Enfin, les messages qui sont dans le potentiel limite, mais qui ne seront pas traités, sont des orphelins de vivacité (zone grisée).

2.3 Discussion

Dans ce chapitre, nous avons examiné en détail la notion d'erreur concurrente sur laquelle nos travaux sont centrés : les messages orphelins. Il n'est pas aisée de définir précisément cette notion du fait de son caractère fortement dynamique et indéterministe. Cependant, les deux classifications : celle par rapport à toutes les dérivations et celle par

27. Par exemple, si on utilise une politique *fifo* pour les boîtes aux lettres, si un autre message traitable est avant lui dans la queue.

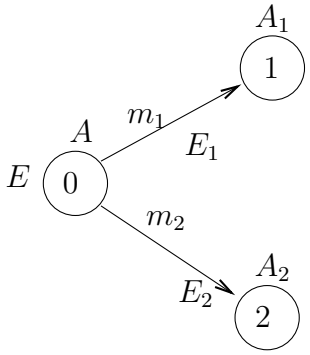


Figure 2.3 Extrait d'un graphe de comportement.

rapport à une exécution, nous semblent assez intuitives.

Les orphelins de sûreté sont clairement des erreurs alors que les orphelins de vivacité peuvent être considérés comme une mauvaise synchronisation entre acteurs. Notre politique vise à supprimer statiquement les premiers et donc *interdire* la compilation de programmes contenant des orphelins de sûreté. Nous chercherons également à détecter les orphelins potentiels qui peuvent être des erreurs. Cependant comme leur calcul statique exact n'est en général pas possible, il est nécessaire d'en faire une approximation. Une piste a déjà été étudiée par J-L. Colaço dans le cadre de sa thèse [Col97]. Ce système permet de détecter tous les messages orphelins de sûreté, le prix à payer étant la détection d'un certain nombre de messages qui semblent orphelins de sûreté ou orphelins potentiels sans l'être. Nous évoquerons ce système plus précisément lors de la présentation du système de type de ML-Act au chapitre 6. Supposons que l'automate de la figure 2.3 soit l'abstraction d'une partie du comportement d'un acteur a. Si a est dans l'état 0 dans lequel il peut traiter m_1 et m_2, avant cet état, les messages E lui ont été envoyés et il accepte les messages A. S'il accepte le message m_i, on lui envoie les messages E_i et il passe dans l'état i qui accepte les messages A_i. Alors, on a $A = \{m_1, m_2\} \cup (A_1 \cap A_2)$ et les émissions doivent vérifier $E \subseteq A$ et $E_i \cup (E \setminus m_i) \subseteq A_i$. Si un message m ne figure pas dans $A_1 \cap A_2$ (si, par exemple, il n'est accepté que dans l'état 1), alors m ne peut être envoyé avant que a n'est atteint l'état 1. Le coût de ce système est donc d'imposer aux programmes la discipline suivante :

> « Si un message n'est compris par un acteur a qu'à partir d'un état E, il faut attendre que a soit dans l'état E pour l'envoyer »

Le système de type impose donc une resynchronisation des acteurs. La configuration suivante n'est pas typable :

$$a \triangleright [m_1 = \xi(e,s).e \triangleright [m = \xi(e,s).e \triangleright s] \;\; m_2 = \xi(e,s).e \triangleright s] \parallel a \triangleleft m$$

Il faut la transformer en la configuration suivante pour qu'elle devienne typable :

$$a \triangleright [m_1 = \xi(e,s).(b \triangleleft c(e) \parallel e \triangleright [m = \xi(e,s).e \triangleright s]) \;\; m_2 = \xi(e,s).e \triangleright s] \parallel b \triangleright [c(a) = \xi(e,s).a \triangleleft m]$$

Lors de son envoi, un message doit figurer dans toutes les branches d'exécution sinon il est déclaré comme orphelin.

Enfin, la dernière catégorie de message orphelin que nous avons présentée, les messages orphelins de vivacité, ne sont pas l'objectif des travaux présentés dans cette thèse. Ils sont néanmoins l'objet de travaux menés au sein de l'équipe *Vestale*. Ils nécessitent cependant des approximations plus puissantes que celle mise en place jusqu'à présent puisqu'il faut intégrer aux systèmes de type des informations de causalité qu'il ne contiennent pas actuellement.

Le mécanisme que nous avons choisi pour essayer de détecter les erreurs dans les programmes ML-ACT et ERLANG est l'analyse statique par inférence de type. Nous allons introduire dans le chapitre suivant le vocabulaire et les concepts dont nous aurons besoin dans la suite de cette thèse lors de la présentation des systèmes de type de chaque langage.

Chapitre 3

Typage

Après avoir décrit la notion d'erreur que nous souhaitons détecter dans le chapitre précédent, nous allons introduire la méthode que nous avons choisie (le typage) de manière assez générale. Puis, nous définirons toutes les notions liées aux systèmes de type dans le cadre de notre approche. Ainsi, nous introduirons entre autres nos techniques d'inférence ainsi que les techniques nécessaires à la résolution des contraintes déduites du code source. Enfin, nous conclurons ce chapitre par une brève discussion sur les différents outils de résolutions existants et sur ce qu'ils pourraient apporter à nos outils d'analyses statiques.

3.1 Les types

Nous n'avons pas encore présenté en détail les langages que nous allons étudier. Cependant, nous avons déjà évoqué un certain nombre de valeurs sémantiques :

- les adresses d'acteurs $a \in \mathbb{A}$,
- les interfaces $\mathcal{R} \in \mathbb{I}$,
- les messages $m \in \mathcal{M}ess$ et
- nop

De plus, l'objectif de nos systèmes de type est la détection des messages orphelins. Ils doivent donc mettre en oeuvre les deux étapes suivantes :

- le calcul de toutes les interfaces que pourrait assumer un acteur ;
- la vérification que tous les messages qui lui sont envoyés sont compatibles avec le résultat du calcul précédent.

Ainsi, nos systèmes vont approximer le calcul du potentiel décrit dans le chapitre précédent. Pour cela, nous approchons les valeurs sémantiques par une nouvelle valeur qui est appelée type. Le principe du calcul de ces types est :

- le type d'une interface est l'ensemble des messages qu'elle accepte ;
- le type d'un acteur correspond à la concaténation de toutes ses interfaces possibles ainsi qu'à l'ensemble des messages qu'il reçoit ;
- un message est approximé par l'ensemble des valeurs qu'il peut prendre. Généralement cette valeur sera unique, mais si l'on abstrait une variable contenant un message, cette valeur peut être un ensemble de messages[28]. Chaque approxima-

28. Nous verrons plus en détail dans le chapitre 6 qu'elle est la raison de cette approximation.

tion d'un message unique ne peut pas être définie puisque nous n'avons pas encore donné la forme des messages. Nous utiliserons la notation M pour décrire le type d'un message (jusqu'à sa définition précise).

Cette stratégie mène à plusieurs formes de types auxquels on ajoute les types classiques :

- les types de base usuels noté T_B, ils contiendront au moins *unit* le type de *nop* ;
- les types construits $\mathcal{C}(T, \ldots, T)$ (par exemple, les types des listes ou des tuples) ;
- les types de messages $\{M, \ldots, M\}$;
- les types d'interfaces $\triangleright\{M, \ldots, M\}$;
- les types d'acteurs $@\{M, \ldots, M\}$.

Dans le cadre des acteurs comme dans celui des objets, une notion de *sous-typage* apparaît naturellement. En effet, la métaphore suivante est vraie :

« *Une personne au moins aussi compétente que moi peut faire mon travail.* »

Donc, un acteur a_1 qui sait traiter au moins tous les messages que l'acteur a_2 sait traiter, peut le remplacer. Cette notion de sous-typage a été proposée indépendamment dans de nombreux contextes (voir par exemple L. Cardelli dans [Car85] qui l'introduit dans le contexte des objets et de l'héritage). L'objectif du sous-typage est d'augmenter la finesse d'un système de type. Pour cela, il faut définir une relation d'ordre sur les types notée $\sqsubseteq$, T_1 est alors un *sous-type* de T_2 lorsque $T_1 \sqsubseteq T_2$. Le système de règles du typage est alors complété par une règle généralement appelée « subsumption » qui a la forme suivante :

$$\frac{\mathcal{E} \vdash_e e : T_1 \quad T_1 \sqsubseteq T_2}{\mathcal{E} \vdash_e e : T_2}$$

Cette règle permet intuitivement d'affaiblir le type d'une expression, c'est-à-dire de diminuer la précision de son type. Par exemple, il est possible de supposer que les entiers sont sous-types des réels, soit $int \sqsubseteq float$. Cela nous permet d'utiliser un entier partout où le système attendait un réel. Par exemple, la fonction de calcul de la racine carrée `sqrt` qui attend un réel peut être appelée sur un entier : `sqrt 2`.

Remarque :
La règle de « subsumption » est une règle non structurelle qui pose des problèmes de résolution.

La relation de sous-typage définie sur les types de base induit une relation d'ordre sur les types construits. Cette extension peut suivre trois règles : *la covariance*, *la contravariance* et *l'invariance*.

Définition 3.1 (Covariance, Contravariance et Invariance) :
Soit $\mathcal{C}$ un constructeur de type d'arité n, il est qualifié de ***covariant*** *(resp.* ***contravariant****) par rapport à sa i^e composante si et seulement si il vérifie pour tout type T, T', T_k l'équation 3.1 (resp. 3.2).*

$$T \sqsubseteq T' \iff \mathcal{C}(T_1, \ldots, T_{i-1}, T, T_{i+1}, \ldots, T_n) \sqsubseteq \mathcal{C}(T_1, \ldots, T_{i-1}, T', T_{i+1}, \ldots, T_n) \quad (3.1)$$

$$T' \sqsubseteq T \iff \mathcal{C}(T_1, \ldots, T_{i-1}, T, T_{i+1}, \ldots, T_n) \sqsubseteq \mathcal{C}(T_1, \ldots, T_{i-1}, T', T_{i+1}, \ldots, T_n) \quad (3.2)$$

Un constructeur est ***invariant*** *par rapport à une composante si et seulement si il est à la fois covariant et contravariant par rapport à cette composante.*

La plupart des constructeurs de type sont covariants. Par exemple, les constructeurs de données (tuples, listes, ...) sont covariants par rapport à toutes leurs composantes. Le plus courant des constructeurs contravariants est le constructeur de fonction, noté $\rightarrow$ (on le considère infixe). C'est un constructeur contravariant par rapport à son domaine. Par exemple, racine sachant calculer sur les réels peut prétendre ne savoir travailler que sur les entiers ($\mathit{int} \sqsubseteq \mathit{float} \iff \mathit{float} \rightarrow T \sqsubseteq \mathit{int} \rightarrow T$). Par contre, il est covariant par rapport à son codomaine, puisque si la fonction renvoie un entier, elle peut prétendre que c'est un réel. On a donc $\mathit{int} \sqsubseteq \mathit{float} \iff T \rightarrow \mathit{int} \sqsubseteq T \rightarrow \mathit{float}$, et plus généralement :

$$T'_0 \sqsubseteq T_0 \wedge T_1 \sqsubseteq T'_1 \iff T_0 \rightarrow T_1 \sqsubseteq T'_0 \rightarrow T'_1$$

Si le sous-typage reste cantonné à définir un ensemble de relations entre les types de base et à leur extension aux types construits, il est qualifié de *sous-typage atomique*. Cependant, cette forme de sous-typage assez limitée ne nous suffit pas. Nous souhaitons pouvoir typer une fonction f de la forme : $\lambda a.a \triangleleft m$ [29], qui prend en paramètre un acteur et lui envoie un message m. Si nous disposons d'un acteur a qui comprend (et comprendra) uniquement le message m, son type peut être décrit par $@\{m\}$. Il est clair que l'on peut appliquer notre fonction à a sans risquer de provoquer d'erreur à l'exécution. Nous pouvons donc attribuer à la fonction f le type $@\{m\} \rightarrow \mathit{unit}$. Mais si nous disposons d'un acteur a' qui comprend (et comprendra) m mais aussi p, nous pouvons également appliquer f à a'. Il est alors nécessaire d'imposer $@\{m,\ p\} \sqsubseteq @\{m\}$, ce qui nous conduit à la notion de sous-typage que nous avions introduite intuitivement entre deux acteurs. Le sous-typage permet ainsi d'exhiber uniquement les parties communes de deux entités, afin de les manipuler ensemble. Le sous-typage est donc une forme de *polymorphisme* (voir [Car85]).

Nous intégrons également les types extremums de cette relation d'ordre que l'on note $\bot$ et $\top$. Le plus petit élément des types $\bot$ est le type représentant l'indéfini, *i.e.* l'expression dont l'évaluation boucle ou échoue, alors que le plus grand des types $\top$ représente l'ensemble de toutes les valeurs sémantiques (sauf l'erreur).

L'ensemble des types peut alors être défini :

Définition 3.2 (Types) :
L'ensemble des types est noté $\mathbb{T}$*, ces éléments sont les termes construits par la grammaire suivante :*

$T ::=$	T_B	*types de base*
$\mid$	$\mathcal{C}(T, \ldots, T)$	*types construits*
$\mid$	$T \rightarrow T$	*type fonctionnel*
$\mid$	t	*variable de type*
$\mid$	I	*type de message*
$\mid$	$\triangleright I$	*type d'interface*
$\mid$	$@I$	*type d'acteur*
$\mid$	$\bot$	*zéro*
$\mid$	$\top$	*un*

29. Nous n'avons bien sur pas encore défini la syntaxe qui nous permettrait de le faire. On utilisera donc qu'une syntaxe simple que l'on pense accessible pour le lecteur avant sa définition précise dans les chapitres suivants.

$$\frac{}{\bot \sqsubseteq T} \qquad \frac{}{T \sqsubseteq \top} \qquad \frac{\forall i \in 1..n\ T_i \sqsubseteq T'_i}{\mathcal{C}(T_1, \ldots, T_n) \sqsubseteq \mathcal{C}(T'_1, \ldots, T'_n)} \qquad \frac{T'_1 \sqsubseteq T_1 \quad T_2 \sqsubseteq T'_2}{T_1 \to T_2 \sqsubseteq T'_1 \to T'_2}$$

$$\frac{I \sqsubseteq I'}{\triangleright I \sqsubseteq \triangleright I'} \qquad \frac{I' \sqsubseteq I}{@I \sqsubseteq @I'}$$

Règles 3.1 La relation de sous-typage.

Il est paramétré par l'ensemble des types de bases qui contient au moins unit, celui des constructeurs de types, celui des variables de types $\mathbb{V}_t$ et les types d'ensemble de messages (notés I). Nous considérerons que tous les autres constructeurs de types sont covariants par rapport à toutes leurs composantes.

Les types interfaces sont très différents dans les deux systèmes étudiés. Nous ne les définirons donc que dans les chapitres dédiés au typage de chacun des langages.

Sur l'ensemble des types, il est alors possible de définir une relation de sous-typage :

Définition 3.3 (Sous-typage) :
Un type T_1 est sous-type d'un type T_2 si et seulement si on peut dériver une preuve de $T_1 \sqsubseteq T_2$ en utilisant le système de la figure 3.1.

Parmi les règles de sous-typage, les deux seules règles qui ne sont pas usuelles sont celles concernant les interfaces et les acteurs. En effet, l'exemple précédent a montré que le sous-typage est contravariant sur les acteurs. Car, un acteur a_1 peut remplacer un acteur a_2 uniquement si a_1 accepte tous les messages que a_2 accepte. Ce qui s'exprime $@\{m,\ p\} \sqsubseteq @\{m\}$, soit si on souhaite conserver un ordre proche de celui sur les ensembles : $\{m\} \sqsubseteq \{m,\ p\}$. À l'opposé, la relation de sous-typage est covariante sur les interfaces. Cette relation qui peut a priori paraître contre-intuitive, mais elle provient du fait que dans le cadre des interfaces, la relation de sous-typage permet la collecte des différentes interfaces d'un comportement. Par exemple, une fonction $\lambda x.(\nu a.(a \triangleright x\,; a))$ a pour type $\triangleright i_x \to @t_a$ (avec la contrainte $i_x \sqsubseteq t_a$ due à la création) et donc son application à une interface I de type $\triangleright i$ conduit à la contrainte $\triangleright i \sqsubseteq \triangleright i_x$. Ce qui permet de conclure $i \sqsubseteq t_a$ et donc de collecter la capacité de I dans le type de l'acteur a.

Le typage des programmes va consister en un parcours de l'arbre du terme au cours duquel :

- une variable de type est attribuée à chaque nœud ;
- des informations sur les relations entre les différentes variables de types ainsi affectées sont récoltées.

3.2 Les environnements de typage

Au cours de la phase de typage, nous affectons des types aux variables rencontrées dans le programme. Les liaisons variables/types ainsi collectées sont stockées dans un environnement de typage.

Définition 3.4 (Environnement de typage) :
*Un **environnement de typage** est une fonction partielle dont le domaine est l'ensemble des variables du programme et le codomaine est l'ensemble des types. Un environnement de typage est noté $\mathcal{E}$, on a donc : $\mathcal{E} \in \mathbb{V} \to \mathbb{T}$.*

Un environnement est noté sous la forme d'un ensemble de liaisons, mais est utilisé sous la forme d'une fonction. C'est-à-dire :
- l'environnement vide est noté $\{\}$;
- l'environnement composé des éléments x_i de type T_i est noté $\{x_1 : T_1, \ldots, x_n : T_n\}$;
- la valeur associée à x dans l'environnement $\mathcal{E}$ est récupérée par $\mathcal{E}(x)$.

Deux opérateurs de sommation d'environnements sont utilisés pour composer les définitions de variables dans les différentes parties d'un programme. Le premier est la *somme simple* d'environnements qui ajoute les variables définies dans un environnement à un autre. Cette opération masque éventuellement une définition précédente si la variable appartient déjà à l'environnement, elle permet ainsi la redéfinition de variables. Le second opérateur est la *somme disjointe*, elle effectue le même calcul que la somme simple mais le résultat n'existe que si les deux environnements sont disjoints. Les deux opérateurs ont une utilité bien spécifique :
- l'introduction de variables dans un filtre ML-Act utilise la somme disjointe pour vérifier la linéarité du filtre ;
- l'introduction effective de variables dans l'environnement d'une expression en masquant éventuellement les variables précédemment définies est réalisée par la somme simple.

Définition 3.5 (Somme simple et Somme disjointe) :
Soit $\mathcal{E}_1$ et $\mathcal{E}_2$ deux environnements :
- *La somme simple $\mathcal{E}_1{+}\mathcal{E}_2$ est définie par :*

$$\mathcal{E}_1{+}\mathcal{E}_2(x) = \begin{cases} \mathcal{E}_2(x) & \text{si } x \in dom(\mathcal{E}_2) \\ \mathcal{E}_1(x) & \text{si } x \in dom(\mathcal{E}_1) \text{ et } x \notin dom(\mathcal{E}_2) \\ \textit{non définie sinon} & \end{cases}$$

- *La somme disjointe $\mathcal{E}_1 \oplus \mathcal{E}_2$ est définie par :*

$$\mathcal{E}_1 \oplus \mathcal{E}_2 = \begin{cases} \mathcal{E}_1{+}\mathcal{E}_2 & \text{si } dom(\mathcal{E}_1) \cap dom(\mathcal{E}_2) = \varnothing \\ \textit{non définie sinon} & \end{cases}$$

3.3 L'inférence

Les systèmes de types proposés dans cette thèse reposent sur un mécanisme d'inférence pour la reconstruction des types et la vérification de la correction d'un programme. Nous pensons en effet, qu'il n'est pas raisonnable d'imposer aux programmeurs de fournir des informations de type en plus de la réalisation du programme parce que nos types sont complexes. En effet, le principal intérêt de l'inférence n'est pas d'économiser le temps du programmeur (même s'il est précieux et que l'écriture des types est souvent fastidieuse),

mais bien de ne pas ajouter une nouvelle source d'erreurs de programmation. Dans un cadre comme celui de la concurrence où les types des programmes sont souvent très compliqués (parfois presque autant que le programme lui-même), écrire un type est une tâche ardue. La contrepartie de l'inférence est une diminution des possibilités d'expressivité du langage des types, car dans le cadre de l'inférence, les techniques actuelles ne permettent pas d'exprimer simplement certaines informations que le programmeur connaît sur son code. Par exemple, la relation de causalité entre divers composants est très difficile à approximer alors que souvent le programmeur la maîtrise bien. Il existe également des langages dans lequel la vérification de type est décidable alors que l'inférence ne l'est pas (par exemple, le système F, voir [Gir72]).

Une *contrainte* est une relation entre certains types du programme, elles sont collectées lors du processus d'inférence. Cette approche à base de contraintes semblent être universellement adoptée par les systèmes d'inférence. Les caractéristiques spécifiques des divers systèmes d'inférence sont la forme des contraintes adoptées et l'algorithme de résolution. Par exemple, l'algorithme d'inférence de ML est un cas particulier de cette méthode d'inférence avec contraintes. En effet, les contraintes sont des contraintes d'égalités entre types calculés et la résolution consiste en un processus d'unification incrémentale. Par exemple, dans un contexte où `e` est de type T, le résultat du typage de l'application de racine (`sqrt`) à `e` est une variable de type t avec la contrainte : $\mathit{float} \to \mathit{float} = T \to t$. La décomposition de cette contrainte conduit à la détermination de la valeur des deux variables de type : $t = T = \mathit{float}$. Ainsi, si ces types calculés sont compatibles avec le reste du programme, le programme est correct. Remarquons que l'algorithme d'unification de ML utilise un principe d'unification assez efficace qui permet de décrire l'ensemble des solutions via l'unificateur le plus général. Les présentations et les implantations de cet algorithme rendent donc les contraintes implicites.

La combinaison de l'inférence et du sous-typage conduit à des contraintes de forme plus complexe. En effet, il n'est plus nécessaire lors d'une application que le type de l'argument coïncide exactement avec le type de la fonction. La contrainte du paragraphe précédent devient $\mathit{float} \to \mathit{float} \sqsubseteq T \to t$. Cette contrainte se décompose alors en appliquant le principe du sous-typage en $T \sqsubseteq \mathit{float}$ et $\mathit{float} \sqsubseteq t$. Il est donc possible d'appliquer `sqrt` à `2` car $\mathit{int} \sqsubseteq \mathit{float}$. Les contraintes générées par nos systèmes seront donc des contraintes d'inégalité entre types.

Les règles de typage exploitées dans cette thèse sont des règles de déductions dirigées par la syntaxe. Chaque jugement présent dans les différentes déductions a la forme suivante : $H \vdash e\colon T,\ C$. Ce jugement exprime que sous les hypothèses H, l'expression e a pour type T si l'ensemble de contraintes C a une solution. Lorsque cela est nécessaire, l'environnement de typage fait partie des hypothèses. La forme générale d'une règle de déduction est alors :

$$\textsc{Nom de la règle:}\ \frac{H_1 \vdash e_1\colon T_1,\ C_1 \quad \cdots \quad H_n \vdash e_n\colon T_n,\ C_n}{H \vdash e\colon T,\ C_1 \cup \cdots \cup C_n \cup C}$$

Pour alléger l'écriture de telles règles, nous rendons l'ensemble des contraintes global. Chaque règle ayant un effet de bord sur cet ensemble global de contraintes exhibe uniquement la ou les contraintes qu'elle y ajoute (l'ensemble C de la règle précédente). La

forme des règles de déductions simplifiées devient alors :

$$\textsc{Nom de la règle:}\ \frac{H_1 \vdash e_1 : T_1 \quad \cdots \quad H_n \vdash e_n : T_n}{H \vdash e : T}\ \Big(C\Big)$$

A l'issue de l'examen du programme source, l'inférence produit un ensemble de liaisons variables/types et un ensemble de contraintes que doivent vérifier ces types. Afin, de conclure sur la validité du programme vis-à-vis du typage, il est alors nécessaire de résoudre les contraintes collectées. Cette résolution correspond au calcul de l'ensemble des solutions de cet ensemble de contraintes. Une solution est un ensemble de liaisons variables de type/types qui vérifie toutes les contraintes. Ce processus peut aboutir à deux situations :

- soit l'ensemble des solutions est vide, l'ensemble des contraintes n'est alors pas soluble ;
- soit le résultat est un ensemble de solutions.

La validité du programme découle alors de l'existence d'une solution pour l'ensemble des contraintes. Un programme est donc correctement typé si et seulement si l'ensemble des contraintes possède au moins une solution.

3.4 La résolution des contraintes

Il faut préciser la sémantique des types pour comprendre le sens des contraintes et ainsi pouvoir les résoudre. Dans le cadre de nos travaux, nos types suivent une sémantique à base d'idéal. Cette structure d'idéal est due à D. MacQueen, G. Plotkin et R. Sethi (on peut par exemple consulter [MPS86]). Elle est nécessaire pour représenter la notion de type récursif car la structure d'ensemble de valeurs est trop faible pour construire les points fixes associés.

Définition 3.6 (Idéal) :
Un sous-ensemble I d'un ensemble ordonné $(E, \sqsubseteq)$ est un idéal si et seulement si :

- *I n'est pas vide ;*
- *chaque suite croissante d'éléments de I possède un maximum dans I ;*
- *tous les éléments de E inférieurs à un élément de I sont dans I.*

Définition 3.7 (Type) :
*Un type est un idéal de l'ensemble des valeurs sémantiques ordonné par la relation d'**approximation**.*

Nous renvoyons le lecteur intéressé par ces notions à l'article de D. MacQueen et al. suscité, et plus particulièrement pour une définition précise de la notion d'ordre d'approximation. La structure d'idéal décrivant les types permet de démontrer un théorème sur l'existence et l'unicité de point fixe pour les types définis par des équations récursives. En effet, les idéaux forment un espace métrique complet dans lequel les opérateurs sur les types sont contractants. L'existence et l'unicité du point fixe provient d'un théorème général [30] sur la convergence de la suite des itérés d'un opérateur contractant.

Dans ce modèle à base d'idéal, les contraintes entre types s'expriment comme des contraintes d'inclusion ensembliste. Les contraintes collectées sur le programme sont donc

30. Il est en général appelé théorème du point fixe.

$$T^+ ::= \bot \mid \top \mid T_B \mid t \mid T^+ \cup T^+ \mid \mathcal{C}_T(T^{s(\mathcal{C})(1)}, \ldots, T^{s(\mathcal{C})(n)})$$
$$T^- ::= \bot \mid \top \mid T_B \mid t \mid T^- \cap T^- \mid \mathcal{C}_T(T^{-s(\mathcal{C})(1)}, \ldots, T^{-s(\mathcal{C})(n)})$$

$$\begin{array}{ll} V_s(\bot) = \{\} & V_s(t) = \{t\} \\ V_s(\top) = \{\} & V_s(T_1^+ \cup T_2^+) = V_s(T_1^+) \cup V_s(T_2^+) \\ V_s(T_B) = \{\} & V_s(T_1^- \cap T_2^-) = V_s(T_1^-) \cup V_s(T_2^-) \\ V_s(\mathcal{C}(\_)) = \{\} & \end{array}$$

Règles 3.2 Types et variables de sommet pour l'algorithme d'Aiken et Wimmers.

résolues en utilisant des techniques développées pour les contraintes ensemblistes (voir par exemple [AW93], [Hei93] ou [Hei92]).

L'algorithme proposé par A. Aiken et E. Wimmers, peut être considéré intuitivement comme une forme de pivot de Gauss qui se déroule en plusieurs étapes. La première étape de l'algorithme utilise des règles dites de *simplification* de contraintes pour réécrire chaque contrainte en un ensemble de contraintes plus simples et ayant une forme normalisée. Chaque système de type doit donc définir un ensemble de règles de simplifications. Cependant, ces règles ne sont pas quelconques, elles doivent, à partir d'un système S, produire un système équivalent à S. Par exemple :

$\bot \subseteq T$ ou $T \subseteq \top$ disparaissent
$T_1 \to T_2 \subseteq T_3 \to T_4$ devient $T_3 \subseteq T_1$ et $T_2 \subseteq T_4$

Cette étape de décomposition est répétée jusqu'à atteindre un point fixe, *i.e.* un ensemble de contraintes qui ne se simplifient plus. Il est alors possible de montrer que le système de contraintes obtenu est *inductif*, c'est-à-dire qu'il est de la forme :

$$\{T_i^+ \subseteq t_i \subseteq T_i^-\}_{i \in I} \text{ avec } V_s(T_i^+) \cup V_s(T_i^-) \subseteq \{t_1, ..., t_{i-1}\}$$

où les notions de types *positifs* T^+ ou *négatifs* T^- et la notion de *variables de sommet* V_s sont définies par les règles 3.2. Le signe associé aux types est appelée *polarité*. Dans le cas du constructeur $\mathcal{C}$, la polarité de la i$^{\text{e}}$ composante est calculée par $s(\mathcal{C})(i)$ dans le cas d'un type positif et l'opposé dans un type négatif. La fonction s renvoie alors $+$ si $\mathcal{C}$ est covariant par rapport à cette composante et $-$ s'il est contravariant.

La partition de l'ensemble des types en types positifs et types négatifs est nécessaire pour que le processus de résolution soit réalisable (du point de vue de la décidabilité). Cette restriction de la forme des contraintes conduit à un sous-ensemble stable (par rapport à la résolution) de contraintes qui portent généralement le nom de *contraintes convenables*. L'indexation des variables de type contenues dans les contraintes fournit un ordre sur ces variables qui est étendu aux contraintes par la notion de variables de sommet.

On montre, alors, qu'un tel système inductif est *équivalent* (*i.e.* il possède les mêmes solutions) au système en *cascade* (*i.e.* triangulaire inférieur) suivant :

$$\{t_i = T_i^+ \cup (T_i^- \cap t_i')\}_{i \in I}$$

où les variables de type t_i' ne sont pas déjà utilisées.

Dans une dernière étape, ce système est transformé en un système contractant (*i.e.* sans variable de sommet) pour conclure par le théorème dû à D. MacQueen et al. suscité qu'il possède une solution. Si ce processus n'aboutit pas le système est contradictoire et on signale une erreur.

Dans la pratique, le système de résolution mis au point pour Erlang est basé sur un autre point de vue de la résolution reposant sur un principe semblable. Nous avons exploité les travaux de F. Pottier (voir sa thèse [Pot98]) où un système de contraintes est vu non pas comme un ensemble mais comme un graphe :

Définition 3.8 (Graphe de contraintes) :
Un ***graphe de contraintes*** *de domaine V est un couple C composé :*
- *d'une relation réflexive sur V notée $\leq_C$ et*
- *de deux fonctions totales $C^{\uparrow}$ et $C^{\downarrow}$ de domaine V et de codomaine respectivement l'ensemble des types négatifs (noté $\mathbb{T}^-$) et celui des types positifs (noté $\mathbb{T}^+$) ; elles doivent, de plus, vérifier : $\forall t \in V \quad \mathcal{FV}(C^{\uparrow}(t)) \cup \mathcal{FV}(C^{\downarrow}(t)) \subseteq V$.*

Dans cette approche, les variables de type ont un rôle particulier, ce sont les nœuds du graphe et la relation d'ordre $\leq_C$ fournit les arêtes du graphe. Les deux fonctions peuvent être vues comme un étiquetage des nœuds du graphe. Chaque nœud est étiqueté par sa borne supérieure ($C^{\uparrow}$) et par sa borne inférieure ($C^{\downarrow}$). La condition sur les bornes impose que toutes les variables utilisées aient été introduites dans V et donc figurent dans le graphe.

Une solution du graphe de contraintes C de domaine V est une substitution σ (de $V \to T$) telle que :

$$\forall t_1,\ t_2 \in V^2 \quad t_1 \leq_C t_2 \Longrightarrow \sigma(t_1) \sqsubseteq \sigma(t_2) \quad \wedge \quad \forall t \in V \quad \sigma(C^{\downarrow}(t)) \sqsubseteq \sigma(t) \sqsubseteq \sigma(C^{\uparrow}(t))$$

Notons qu'un graphe de contraintes ne contient que des contraintes *simples* c'est-à-dire ayant une des trois formes suivantes : $t_1 \sqsubseteq t_2$, $t \sqsubseteq T^-$ ou bien $T^+ \sqsubseteq t$. L'algorithme doit donc, comme celui d'Aiken et Wimmers, comporter une phase de simplification des contraintes. Cette phase est effectuée avant chaque ajout au graphe de contraintes.

Définition 3.9 (Graphe de contraintes clos) :
Un graphe de contraintes C est dit ***clos*** *si et seulement si :*
- *$\leq_C$ est transitive*
- *$\forall t_1,\ t_2 \quad t_1 \leq_C t_2 \Longrightarrow C^{\downarrow}(t_1) \sqsubseteq C^{\downarrow}(t_2) \ \wedge \ C^{\uparrow}(t_2) \sqsubseteq C^{\uparrow}(t_1)$*
- *$\forall t \quad C^{\downarrow}(t) \sqsubseteq C^{\uparrow}(t)$*

On montre alors que tout graphe de contraintes clos possède une solution. En fait, le principal intérêt de cette approche provient du fait que le graphe reste clos à chaque ajout de contraintes. L'algorithme porte alors le nom de clôture incrémentale et gagne en efficacité. Soit C un graphe de contraintes et c une contrainte simple, l'algorithme 3.3 montre la construction de $C + c$ par :
- L'ajout d'une contrainte n'est effectif que si la contrainte n'est pas déjà contenue dans le graphe (cas 1.a, 2.a et 3.a).
- L'ajout d'une contrainte de la forme $t_1 \sqsubseteq t_2$ (qui ne figure pas dans C, cas 1.b) consiste en :

1. si $c = t_1 \sqsubseteq t_2$ alors $dom(C + c) = dom(C) \cup \{t_1;\ t_2\}$ et :
   (a) si $t_1 \leq_C t_2$ alors $C + c = C$
   (b) si $t_1 \not\leq_C t_2$ alors
$$\begin{cases} \leq_{C+c} = \leq_C \cup \{(t'_1, t'_2) \mid t'_1 \leq_C t_1 \wedge t_2 \leq_C t'_2\} \\ (C+c)^{\uparrow}(t'_1) = \begin{cases} C^{\uparrow}(t'_1) \sqcap C^{\uparrow}(t_2) & \text{si } t'_1 \leq_C t_1 \\ C^{\uparrow}(t'_1) & \text{sinon} \end{cases} \\ (C+c)^{\downarrow}(t'_2) = \begin{cases} C^{\downarrow}(t'_2) \sqcup C^{\downarrow}(t_1) & \text{si } t_2 \leq_C t'_2 \\ C^{\downarrow}(t'_2) & \text{sinon} \end{cases} \\ \text{il faut ajouter } C^{\downarrow}(t_1) \sqsubseteq C^{\uparrow}(t_2) \text{ à } C + c \end{cases}$$
2. si $c = t \sqsubseteq T$ alors :
   (a) si $T \in C^{\uparrow}(t)$ alors $C + c = C$
   (b) si $T \notin C^{\uparrow}(t)$ alors
$$\begin{cases} \leq_{C+c} = \leq_C \\ (C+c)^{\uparrow}(t') = \begin{cases} C^{\uparrow}(t') \sqcap T & \text{si } t' \leq_C t \\ C^{\uparrow}(t') & \text{sinon} \end{cases} \\ (C+c)^{\downarrow} = C^{\downarrow} \\ \text{il faut ajouter } C^{\downarrow}(t) \sqsubseteq T \text{ à } C + c \end{cases}$$
3. si $c = T \sqsubseteq t$ alors :
   (a) si $T \in C^{\downarrow}(t)$ alors $C + c = C$
   (b) si $T \notin C^{\downarrow}(t)$ alors
$$\begin{cases} \leq_{C+c} = \leq_C \\ (C+c)^{\uparrow} = C^{\uparrow} \\ (C+c)^{\downarrow}(t') = \begin{cases} C^{\downarrow}(t') \sqcup T & \text{si } t \leq_C t' \\ C^{\downarrow}(t') & \text{sinon} \end{cases} \\ \text{il faut ajouter } T \sqsubseteq C^{\uparrow}(t) \text{ à } C + c \end{cases}$$

Règles 3.3 Algorithme de construction de $C + c$.

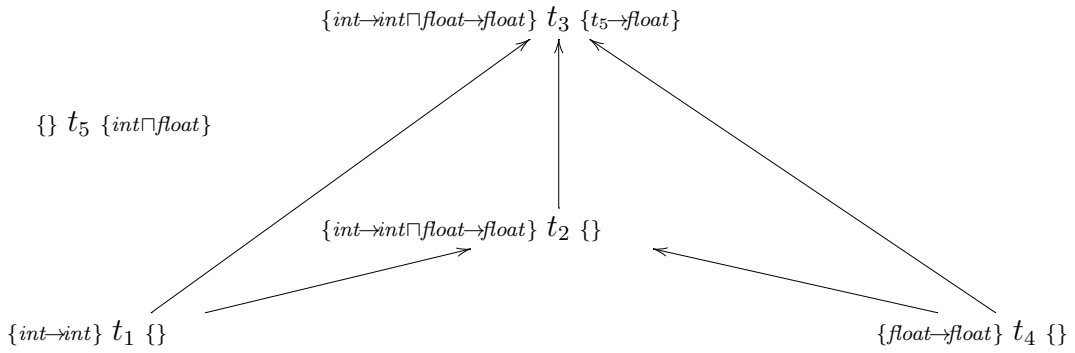


Figure 3.1 Un graphe de contrainte.

- l'ajout de la relation entre t_1 et t_2 à $\leq_C$ puis en la clôture par transitivité (et reflexivité) de cette nouvelle relation ;
- la mise à jour des bornes supérieures (resp. inférieures) des éléments inférieurs (resp. supérieurs) à t_1 (resp. t_2) par intersection (resp. union) avec la borne supérieure (resp. inférieure) de t_2 (resp. t_1) ;
- il faut ensuite ajouter des contraintes issues de la simplification de la contrainte entre la borne inférieure de t_1 et la borne supérieure de t_2 pour vérifier qu'aucune inconsistance n'a été ajoutée dans le graphe.

– L'ajout d'une contrainte des deux autres formes possibles (cas 2.b ou 3.b) est plus à l'ajout d'une contrainte entre variables. En effet, il suffit d'ajouter (par intersection resp. union) la borne T à la borne supérieure (resp. inférieure) des éléments plus petits (resp. grands) que t figurant dans le graphe de contrainte. Il faut ensuite s'assurer que le graphe n'est pas inconsistant en ajoutant les contraintes issues de la simplification de $C^{\downarrow}(t) \sqsubseteq T$ (resp. $T \sqsubseteq C^{\uparrow}(t)$).

Cet algorithme est donc récursif, puisque l'ajout d'une contrainte peut exiger l'ajout de plusieurs nouvelles contraintes. Cependant, il est possible de montrer que cet algorithme termine et que, soit il échoue ($C + c$ n'est donc pas soluble), soit il réussit et le graphe résultant est clos.

Remarque :
Notons que, dans l'algorithme, lorsqu'une contrainte est ajoutée à C, elle doit être décomposée en contraintes simples et que ce sont celles-ci qui sont ajoutées à C.

La figure 3.1 présente le graphe de contrainte obtenu par l'ajout successif des contraintes suivantes dans leur ordre d'apparition :

$$\begin{cases} t_1 \sqsubseteq t_2 \\ t_4 \sqsubseteq t_2 \\ t_2 \sqsubseteq t_3 \\ int \to int \sqsubseteq t_1 \\ float \to float \sqsubseteq t_4 \\ t_3 \sqsubseteq t_5 \to float \end{cases}$$

Les trois premières contraintes ajoutent au graphe les nœuds t_1 à t_4 ainsi que les arêtes présentes dans le diagramme final. La quatrième (resp. cinquième) contrainte ajoute la borne inférieure $int \to int$ (resp. $float \to float$) à t_1 (resp. t_4). Ces deux contraintes sont également ajoutées par transitivité à t_2 et t_3. Enfin, la dernière contrainte provoque la création du nœud t_5, ajoute la borne supérieure $t_5 \to float$ à t_3 et provoque l'ajout de la contrainte $int \to int \sqcap float \to float \sqsubseteq t_5 \to float$. Cette contrainte se décompose en $\{t_5 \sqsubseteq int,\ int \sqsubseteq float,\ t_5 \sqsubseteq float,\ float \sqsubseteq float\}$ et provoque donc l'ajout de deux bornes supérieures à t_5 (int et $float$).

Dans le cadre, de cette thèse, nous réutilisons directement les résultats sur les graphes de contraintes. Nos systèmes seront caractérisés par des règles de simplification et une phase terminale dite de minimisation spécifique (pour l'affichage des types). Nous renvoyons le lecteur intéressé par la description complète des graphes de contraintes, de leurs propriétés, de la démonstration de la propriété d'existence de solution et de la correction de l'algorithme de clôture incrémentale, à la thèse de F. Pottier [Pot98].

3.5 Discussion

L'équipe *Vestale* s'est intéressée, entre 1991 et 1995, à l'intégration du concept objet et de la programmation fonctionnelle typée dans le cadre du projet Fol. Le système de type de Fol conçu par M. Pantel, fonctionnant par inférence de contraintes, nécessitait l'utilisation d'un solveur de contraintes ensemblistes à la Aiken et Wimmers (voir [Pan94]). Le seul système de résolution de contraintes qui existait à l'époque était *Illyria*, mais ce prototype s'est révélé peu utilisable. X. Thirioux a donc implanté un algorithme équivalent en Camllight dans le cadre de son DEA [Thi94]. Cet algorithme et le solveur ont ensuite été adaptés à Cap par J-L. Colaço dans le cadre de sa thèse [Col97]. Les premiers travaux sur ML-Act ont donc naturellement réutilisé cette compétence et le code existant. Ainsi, dans le cadre de mon DEA (voir [Dag97]), j'ai réalisé un système de type fonctionnel avec une inférence d'effets. Les actions concurrentes de ML-Act sont considérées comme des effets dans ce système et sont approximées par un terme du calcul Cap. Cette technique, inspirée des travaux de H. Nielson et F. Nielson sur l'inférence d'un terme du π-calcul à partir d'un programme CML (voir [NN93] ou [NNA96]), a permis de réutiliser les outils d'analyse de Cap mais s'est révélée insuffisante pour analyser les programmes où le calcul fonctionnel et les opérations concurrentes sont très entremêlées. Par exemple, avec cette technique, il n'est pas possible de stocker dans des listes des adresses d'acteurs, ou bien, il n'est pas possible d'envoyer des messages dans le corps d'une fonction récursive. Pour passer outre cette limitation, nous avons, dans un premier temps, ajouté une abstraction fonctionnelle (un lambda) dans la syntaxe de Cap. Le système de type de Cap et son algorithme de résolution ont alors été étendu par M. COLIN dans le cadre de son DEA ([Col98]) pour prendre en compte cette ajout de fonction. Le système de type de ML-Act qui sera présenté dans la suite de cette thèse est une reconstruction du système de type de λCap dans le cadre du langage complet.

En revanche, le solveur de contraintes du système d'analyse d'Erlang a été conçu à partir de la notion de graphe de contraintes introduites dans la section précédente et a donc été repensé quasiment entièrement.

Cependant, depuis ces travaux deux prototypes de solveurs *génériques* ont été mis à disposition par leurs auteurs :

- BANE conçu par A. AIKEN et M. FÄNHDRICH et décrit dans [FA97] et [Fäh99]. Le logiciel est téléchargeable à l'adresse :
  `http://http.cs.berkeley.edu/Research/Aiken/bane.html`
- Wallace réalisé par F. POTTIER basé sur ses travaux dans le domaine des graphes de contraintes et de leur clôture incrémentale de manière efficace. Le solveur est disponible en téléchargement à l'adresse :
  `http://cristal.inria.fr/~fpottier/wallace/wallace.html.fr`

Pour des raisons de compatibilité avec nos codes et de compétences, nos essais se sont concentrés sur Wallace. Nous avons réalisé un petit prototype de typeur pour un sous-ensemble de ML-ACT. Un des objectifs futurs est donc de réussir à interfacer notre compilateur ML-ACT avec Wallace pour pouvoir utiliser sa puissance. Cependant, des travaux sont menés au sein de l'équipe pour construire des systèmes de type plus précis qui nécessitent un solveur plus puissant. Par exemple, actuellement, nous intégrons aux types des fonctions une notion de multiplicité pour compter le nombre de fois qu'elles sont appliquées et ainsi maintenir une comptabilité précise des messages émis. A notre connaissance, il ne semble pas facile d'intégrer cette notion de multiplicité ni dans BANE, ni dans Wallace. Car la résolution devra cohabiter avec un solveur d'équations arithmétiques comme Omega.

Chapitre 4

Une première instanciation

Afin de familiariser le lecteur avec les concepts généraux introduits dans les trois précédents chapitres et faciliter ainsi la compréhension des parties suivantes, nous allons réaliser une première instanciation de notre système sur un λ-calcul minimal. Ce langage sera nommé $\mathcal{F}unc_0$.

4.1 Le langage $\mathcal{F}unc_0$

Pour rendre plus accessible les systèmes de type proposés dans la suite de cette thèse, nous utilisons un noyau fonctionnel minimaliste appelée $\mathcal{F}unc_0$. Sa syntaxe est introduite dans cette section.

Définition 4.1 ($\mathcal{F}unc_0$) :
Une expression fonctionnelle est un terme qui représente un programme fonctionnel. Sa définition est paramétrée par quatre ensembles : l'ensemble des variables (notées $x \in \mathbb{V}$), l'ensemble des constantes (notées $c \in \mathbb{C}$), l'ensemble des fonctions constantes (notées $f \in \mathcal{F}_c$) et l'ensemble des adresses d'acteurs (notées $a \in \mathbb{A}$).

- *Les seules constantes sont nop (qui décrit un calcul vide) et new (qui crée une nouvelle adresse d'acteur) : $\mathbb{C} \triangleq \{nop,\ new\}$.*
- *Les fonctions constantes sont celles du modèle d'acteur : $\mathcal{F}_c \triangleq \{send,\ become,\ init\}$.*
- *L'ensemble des messages, noté $\mathcal{M}ess$, contient les données construites par un ensemble de constructeurs appelés étiquettes. Cet ensemble d'étiquettes est noté $\mathbb{M}$ et est distinct de tous les autres ensembles définis. On a donc : $\mathcal{M}ess \triangleq \mathbb{M} \times \mathcal{E}xp$*

Les expressions sont construites par la grammaire suivante :

$$e ::= x \mid a \mid c \mid f(e,\, e) \mid m(e) \mid e\ e \mid \lambda x.e \mid \chi[m(x).e, \ldots, m(x).e] \mid \texttt{letrec}\ x = e\ \texttt{in}\ e$$

Une expression peut être une variable x, une adresse d'acteur a, une constante c, l'appel d'une fonction prédéfinie f (elles sont toutes les trois binaires) ou un message d'étiquette m. À l'image du λ-calcul, la syntaxe contient l'application ainsi que l'abstraction. L'ensemble des abstractions (ou fonctions) est noté $\mathcal{F}$. Nous séparons syntaxiquement les interfaces $\chi[\ldots]$ (dont l'ensemble est noté $\mathbb{I}$) des abstractions usuelles $\lambda x.e$. Pour des raisons de commodité d'écriture, nous employons une notation indicée : $\chi[m_i(x_i).e_i]^{i \in 1..n}$. Enfin, nous autorisons les fonctions récursives.

$$
\begin{array}{ll}
\mathcal{FN}(a) & = \{a\} \\
\mathcal{FN}(c) & = \{\} \\
\mathcal{FN}(x) & = \{\} \\
\mathcal{FN}(m(e)) & = \mathcal{FN}(e) \\
\mathcal{FN}(\lambda x.e) & = \mathcal{FN}(e)
\end{array}
\qquad
\begin{array}{ll}
\mathcal{FN}(e_1\ e_2) & = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2) \\
\mathcal{FN}(\texttt{letrec}\ x = e_1\ \texttt{in}\ e_2) & = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2) \\
\mathcal{FN}(f(e_1,\, e_2)) & = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2) \\
\mathcal{FN}(\chi[m_i(x_i).e_i]^{i\in I}) & = \bigcup\{\mathcal{FN}(e_i) | i \in I\}
\end{array}
$$

Règles 4.1 Ensemble des noms libres d'une expression de $\mathcal{F}unc_0$.

Dans les différents exemples, nous exploitons également deux autres constructions syntaxiques qui ne sont pas incluses dans la syntaxe car elles sont immédiatement traduisibles en une composition de plusieurs expressions y figurant déjà. Ce sucre syntaxique est :

– $\texttt{let}\ x\ = e_1\ \texttt{in}\ e_2$ qui signifie $(\lambda x.e_2)\ e_1$,

– $e_1\,;e_2$ qui signifie également $(\lambda x.e_2)\ e_1$ avec x ne figurant pas dans e_1.

Car $\mathcal{F}unc_0$ suit la règle de l'appel par valeur.

La sémantique présentera de manière rigoureuse le sens de chaque expression, mais intuitivement, elles conservent leur sens usuel dans le contexte fonctionnel.

Avant de définir précisément la sémantique de $\mathcal{F}unc_0$, introduisons un exemple de programme qui décrit une cellule linéaire sur lequel nous définissons notre vocabulaire :

$$
\begin{array}{l}
\star \triangleright \texttt{letrec cell} = \lambda v.\lambda ego.\texttt{letrec}\ self = \chi[get(c).send(reply(v), c)\,;become(ego, self)\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad set(v').become(ego, (\texttt{cell}\ v')\ ego)]\ \texttt{in}\ self \\
\quad \texttt{in let a\_cell} = new\ \texttt{in}\ init(a\_cell, \lambda ego.\chi[start(v).become(ego, (\texttt{cell}\ v)\ ego)])\,; \\
\qquad send(start(a), \texttt{a\_cell})\,; send(get(c), \texttt{a\_cell})\,; send(set(b), \texttt{a\_cell})\,; send(get(c), \texttt{a\_cell})
\end{array}
$$

où a, b et c sont trois acteurs figurant dans le contexte de ce terme.

$$
\star \triangleright \texttt{letrec cell} = \lambda v.\ \underbrace{\lambda ego.\texttt{letrec}\ self = \overbrace{\chi[get(c).\ ...,\ set(v').\ ...]}^{interface}\ \texttt{in}\ self)}_{comportement}\ \texttt{in}\ ...
$$

On parle de *comportement* pour décrire une fonction qui prend en argument une adresse et renvoie une interface. Une *interface* étant la partie décrivant les messages acceptés et les réactions associées. Les messages qu'une interface peut traiter ($get(c)$ par exemple) sont dit *installés*. Dans l'envoi $send(start(a), \texttt{a\_cell})$, le message est dit *envoyé* ou *reçu* selon le point de vue.

Dans le premier chapitre, nous avons introduit la notion de configuration. Sur lesquelles nous avons défini une notion de nom libre. Pour compléter cette définition, il faut maintenant définir la notion de nom libre dans une expression de $\mathcal{F}unc_0$.

Un nom est **libre** dans une expression, si et seulement si ce nom n'est pas lié par un opérateur ν de configuration dans la portée duquel figure l'expression. Ainsi :

Définition 4.2 (Nom libre) :
Tout nom n figurant dans une expression e est libre. L'ensemble des noms libres d'une expression e est noté $\mathcal{FN}(e)$ et est calculé par les règles 4.1.

$$
\begin{array}{ll}
[v/x]a & = a \\
[v/x]c & = c \\
[v/x]f(e_1,\, e_2) & = f([v/x]e_1,\, [v/x]e_2) \\
[v/x]m(e) & = m([v/x]e) \\
[v/x]e_1\ e_2 & = [v/x]e_1\ [v/x]e_2
\end{array}
\qquad
[v/x]x' = \begin{cases} v & \text{si } x' = x \\ x' & \text{sinon} \end{cases}
$$

$$
[v/x]\lambda x'.e = \begin{cases} \lambda x'.e & \text{si } x' = x \\ \lambda x'.[v/x]e & \text{sinon} \end{cases}
$$

$$
[v/x](\texttt{letrec}\ x' = e_1\ \texttt{in}\ e_2) = \begin{cases} \texttt{letrec}\ x' = e_1\ \texttt{in}\ e_2 & \text{si } x' = x \\ \texttt{letrec}\ x' = [v/x]e_1\ \texttt{in}\ [v/x]e_2 & \text{sinon} \end{cases}
$$

$$
[v/x]\chi[m_i(x_i).e_i]^{i\in I} \;=\; \chi[m_i(x_i).e'_i]^{i\in I} \text{ avec } e'_i = \begin{cases} e_i & \text{si } x_i = x \\ [v/x]e_i & \text{sinon} \end{cases}
$$

Règles 4.2 Substitution d'une variable par sa valeur dans $\mathcal{F}unc_0$.

Remarque :
Cette définition porte sur les noms (ou adresses) et pas sur les variables.

4.2 La sémantique de $\mathcal{F}unc_0$

La sémantique d'une configuration combinée à $\mathcal{F}unc_0$ suit les règles de la sémantique des configurations introduites dans le premier chapitre. Cependant, cette introduction avait laissé en suspens la sémantique de la partie fonctionnelle que nous allons définir maintenant. Nous allons donc préciser les notions de valeur sémantique et de substitution, les contextes d'évaluation, la réduction fonctionnelle et enfin, la sémantique donnée à la boîte aux lettres d'un acteur.

Définition 4.3 (Valeur sémantique) :
Une valeur sémantique est soit une adresse d'acteur, soit nop, soit un message construit à partir d'une valeur sémantique, soit une abstraction soit une interface. L'ensemble des valeurs sémantiques est noté $\mathcal{V}$ et :

$$\mathcal{V} \triangleq \mathbb{A} \cup \{nop\} \cup \mathbb{M} \times \mathcal{V} \cup \mathcal{F} \cup \mathbb{I}$$

Lors de l'exécution d'un programme de $\mathcal{F}unc_0$, chaque variable est *remplacée*, au fur et à mesure de l'avancée du calcul, par sa valeur (une valeur sémantique) au moyen de substitutions.

Remarquons qu'en $\mathcal{F}unc_0$, une valeur sémantique ne peut contenir de variable libre car la liaison est lexicale. Ainsi, toute variable figurant dans une valeur sémantique ne peut se voir capturer par son contexte.

Définition 4.4 (Substitution) :
Une substitution $[v/x]$ est un automorphisme sur les expressions qui remplace toutes les occurrences libres de la variable x par la valeur v. Elle est définie par induction sur la structure des expressions par les règles 4.2. La substitution vide (l'identité) est notée $[]$, elle n'a aucun effet sur l'expression.

Pour présenter l'évaluation fonctionnelle, nous utilisons la notion de *contexte d'évaluation* introduite rapidement dans le premier chapitre. Elle permet de spécifier l'ordre des calculs dans le cadre d'une relation de réduction. En $\mathcal{F}unc_0$, l'application de fonctions utilise l'appel par valeur – *i.e.* les arguments sont tous évalués avant d'exécuter la fonction.

Définition 4.5 (Contexte d'évaluation) :
Un contexte d'évaluation C *est une expression dans laquelle figure un trou. La notation* C[e] *signifie que ce trou est rempli par une expression* e *en cours de réduction. Parmi les termes construits par la grammaire suivante, les contextes sont ceux qui ne contiennent qu'un seul trou :*

$$\mathrm{C} ::= [] \mid f(\mathrm{C}, e) \mid f(v, \mathrm{C}) \mid m(\mathrm{C}) \mid \mathrm{C}\ e \mid v\ \mathrm{C} \mid \texttt{letrec}\ x = \mathrm{C}\ \texttt{in}\ e$$

La notion de contexte d'évaluation permet de décrire simplement une sémantique *à petits pas*. Nous avons choisi cette forme de sémantique car elle nous semble la plus adaptée à la description de l'exécution d'un langage concurrent. En effet, l'évaluation peut à tout moment interagir avec l'environnement et donc provoquer d'autres exécutions. Par exemple, l'envoi d'un message lors d'une évaluation doit résulter immédiatement par son arrivée dans le médium de communication. Notre sémantique repose donc sur une notion d'entrelacement des différentes réductions possibles par rapport à laquelle chaque réduction est supposée atomique.

Le calcul fonctionnel

L'exécution fonctionnelle des programmes du langage $\mathcal{F}unc_0$ résulte de l'application des règles :

$$\textsc{Cont}: \frac{e \longrightarrow_e e'}{\mathrm{C}[e] \longrightarrow_e \mathrm{C}[e']} \qquad \textsc{ContE}: \frac{e \longrightarrow_e \mathbf{Err}}{\mathrm{C}[e] \longrightarrow_e \mathbf{Err}} \qquad \textsc{VarE}: \frac{}{x \longrightarrow_e \mathbf{Err}} \qquad \textsc{AppE}: \frac{v \notin \mathcal{F}}{v\ e \longrightarrow_e \mathbf{Err}}$$

$$\textsc{App}: \frac{}{\lambda x.e\ v \longrightarrow_e [v/x]e} \qquad \textsc{LRec}: \frac{}{\texttt{letrec}\ x = v\ \texttt{in}\ e \longrightarrow_e [\texttt{letrec}\ x = v\ \texttt{in}\ v/x]e}$$

L'évaluation des sous-expressions est indépendante du contexte. Le résultat de la réduction d'une sous-expression peut être erroné (ContE) auquel cas le calcul est interrompu et renvoie une erreur, ou bien fournit une autre expression (Cont) qui remplace alors la première dans le trou du contexte.

Comme toute variable introduite est substituée à l'exécution par sa valeur, aucune variable ne peut plus figurer dans une expression en cours d'évaluation sauf si cette variable est libre (VarE).

L'application d'une valeur non fonctionnelle (AppE) provoque également une erreur. L'application d'une abstraction est classique (App) l'argument est substitué dans le corps de l'abstraction par la valeur appliquée.

L'évaluation d'une définition récursive (LRec) consiste à substituer dans son corps e la variable x introduite par sa valeur v. Notons que, dans v, nous réintroduisons la définition récursive qui sera nécessaire à la prochaine utilisation de x. Il est clair qu'une telle règle

d'évaluation peut conduire à un bouclage sans fin des calculs. La valeur v introduite par une définition récursive doit avoir un comportement *paresseux*, en d'autre terme, il ne peut s'agir que d'une abstraction ou d'une interface qui ne s'évalue que lorsqu'elle est utilisée. Nous ne nous intéressons cependant pas ici à ce problème.

La sémantique des boîtes aux lettres

Dans le premier chapitre, lors de l'introduction des configurations et de leur sémantique, la sémantique des boîtes aux lettres a été abstraite. En $\mathcal{F}unc_0$, un acteur a inactif confronte chaque message de sa boîte aux lettres à toutes les branches de son interface courante. Ce comportement est décrit par les règles ci-dessous dans lesquelles, nous supposons que l'ensemble I indexant les cas est non vide. En effet, un acteur ayant un comportement vide ne consulte jamais sa boîte aux lettres.

$$\textsc{Bal}: \frac{\exists j \quad (\forall i < j \;\; m \neq m_i) \qquad\qquad m_j = m}{\chi[m_i(x_i).e_i]^{i\in I} \prec m(v) :: q \Longrightarrow q,\ e_j,\ [v/x_j]}$$

$$\textsc{BalO}: \frac{(\forall i \in I \;\; m \neq m_i) \qquad \chi[m_i(x_i).e_i]^{i\in I} \prec q \Longrightarrow q',\ e,\ \sigma}{\chi[m_i(x_i).e_i]^{i\in I} \prec m(v) :: q \Longrightarrow m(v) :: q',\ e,\ \sigma}$$

Si le filtrage du message en tête de file par les $j-1$ premiers motifs a échoué, et que, les étiquettes du j^{e} rang sont compatibles (BAL), le résultat est alors composé du reste de la file, du j^{e} corps et de la substitution $[v/x_j]$. Sinon, le processus continue avec le motif suivant. En fin de processus, si aucun des motifs n'a permis de filtrer $m(v)$ (BALO), on essaie de filtrer le message suivant de la queue.

Remarque :
Notons qu'après l'échec d'un traitement par tous les filtres, un message est remis dans la boîte aux lettres en préservant l'ordre initial pour garantir l'équité du traitement des messages.

4.3 Les erreurs

Les règles concernant le calcul du potentiel d'un acteur (qui a été défini dans le chapitre 2) doivent être présentées pour conclure l'instanciation de notre modèle. En effet, elles sont nécessaires à la sémantique pour la règle INIT où un acteur est marqué (en indice) par son potentiel initial :

$$\textsc{Init}: \frac{}{\alpha \triangleright \mathrm{C}[init(a,v)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel \langle a \,|\, \varnothing \rangle_{\mathcal{P}(v\ a)} \triangleright v\ a}$$

Introduisons le calcul de ce potentiel dans le cadre de $\mathcal{F}unc_0$. Il est effectué selon les règles 4.3 et utilise une fonction intermédiaire $\mathcal{P}$ qui approxime une expression par un couple composé d'une *valeur* V, qui approxime la valeur de l'expression et d'un *effet* E, qui contient toutes les étiquettes de toutes les interfaces effectivement assumées. Le

$$\textsc{Potentiel}: \frac{\mathcal{P}(e) = V,\ E}{\mathcal{P}_e(e) = setof(V) \cup E}$$

$$\textsc{Val1}: \frac{v \in \mathbb{A} \cup \mathbb{C}}{\mathcal{P}(v) = \{\},\ \{\}} \qquad \textsc{Val2}: \frac{V \in \mathcal{V}al}{\mathcal{P}(V) = V,\ \{\}} \qquad \textsc{Var}: \frac{}{\mathcal{P}(x) = \{\},\ \{\}}$$

$$\textsc{Mess}: \frac{\mathcal{P}(e) = V,\ E}{\mathcal{P}(m(e)) = \{\},\ E} \qquad \textsc{Int}: \frac{\mathcal{P}(\sigma_i(e_i)) = V_i,\ E_i \quad \text{avec } \sigma_i = [\mathbb{M}/x] \text{ si } p_i = x \text{ et } [] \text{ sinon}}{\mathcal{P}(\chi[m_i(p_i).e_i]^{i \in I}) = \bigcup_i \{m_i\},\ \bigcup_i E_i}$$

$$\textsc{Bec}: \frac{\mathcal{P}(e) = V,\ E}{\mathcal{P}(become(a,\ e)) = \{\},\ setof(V) \cup E}$$

$$\textsc{App}: \frac{\mathcal{P}(e) = \lambda x.e,\ E \quad \mathcal{P}(e') = V',\ E' \quad \mathcal{P}([V'/x]e) = V'',\ E''}{\mathcal{P}(e\ e') = V'',\ E \cup E' \cup E''}$$

$$\textsc{App}: \frac{\mathcal{P}(e) = V,\ E \quad \mathcal{P}(e') = V',\ E'}{\mathcal{P}(e\ e') = \{\},\ E \cup E'} \qquad \textsc{LRec}: \frac{\mathcal{P}(e) = V,\ E \quad \mathcal{P}([V/x]e') = V',\ E'}{\mathcal{P}(\texttt{letrec}\ x = e\ \texttt{in}\ e') = V',\ E \cup E'}$$

Règles 4.3 Potentiel d'un terme de $\mathcal{F}unc_0$.

calcul de cette approximation correspond à une exécution abstraite du programme sur un ensemble réduit de valeurs sémantiques. Ces *nouvelles* valeurs possibles sont un ensemble d'étiquettes (éventuellement vide ou valant $\mathbb{M}$) ou une abstraction. Elles peuvent être décrites par la grammaire suivante :

$$\mathcal{V}al ::= \{\} \mid \{m, \ldots, m\} \mid \mathbb{M} \mid \lambda x.e$$

Nous calculons donc les interfaces possibles d'un acteur en approximant une expression par une valeur. Ce calcul consiste intuitivement en la détermination de l'ensemble des interfaces que peut *valoir* une expression. Plus précisément, l'ensemble des étiquettes de messages acceptées par ces interfaces. Ainsi, si cette expression est argument d'un *become*, sa valeur est ajoutée à l'ensemble des interfaces possibles de l'acteur (l'effet). L'effet est donc un ensemble d'étiquettes qui approxime toutes les interfaces des futurs états de l'acteur courant. Les deux règles principales du calcul sont :

- La règle qui décrit l'approximation d'une interface (règle Int) car c'est la seule règle qui ajoute réellement des étiquettes.
- La règle de calcul du changement de comportement (règle Bec) qui relie le calcul de la valeur et celui de l'effet. Par l'intermédiaire de la fonction *setof*, elle ajoute dans l'ensemble des interfaces possibles la valeurs de l'expression décrivant le comportement. La fonction *setof*, définie sur les valeurs, renvoie un ensemble vide si la valeur est une abstraction ($setof(\lambda x.e) = \{\}$) et elle correspond à l'identité sur les ensembles d'étiquettes ($setof(V) = V$).

Notons, que si l'acteur assume un comportement reçu dans le corps d'un message, on ne peut plus calculer simplement le potentiel de l'acteur. Le système de type proposé

dans la suite de ce chapitre *calcule* cette approximation dans le cas d'un programme clos pour lequel il est alors possible d'évaluer l'ensemble des messages reçus par un acteur. Mais, si l'acteur peut recevoir des messages venant d'un contexte ouvert même cette approximation devient impossible. Dans le cadre du potentiel, nous sommes moins précis dans l'abstraction du programme et nous approximons alors le potentiel par $\mathbb{M}$ (il est ouvert). Pour cela, si le message contient un argument, celui-ci est approximé par $\mathbb{M}$ lors de la définition d'un comportement (règle INT) .

Enfin, comme le calcul du potentiel effectue un dépliage des entités récursives, sa terminaison requiert une forme de calcul d'occurrence qui vérifie qu'une fonction récursive qui a déjà été approximée ne l'est pas à nouveau. Pour cela, nous ne substituons la fonction récursive introduite qu'une seule fois par son corps (LREC). Alors, si la fonction se rappelle à nouveau, son nom est libre et est donc approximé en utilisant la règle VAR.

Notons que cette stratégie d'approximation est valide (vis-à-vis du potentiel) puisque chaque comportement est déplié au moins une fois dans toute expression dans laquelle il figure. Autrement dit, toutes les interfaces qui seront potentiellement assumées sont effectivement prises en compte.

Montrons un exemple de calcul du potentiel pour un acteur a de corps :

$$\texttt{letrec}\ x = \chi[m(y).become(a, \chi[p(z).become(a, x)])]\ \texttt{in}\ become(a, x)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P}(x) = \{\}, \{\}}{\mathcal{P}(become(a, x)) = \{\}, \{\}}}{\mathcal{P}(\chi[p(z).become(a, x)]) = \{p\}, \{\}}}{\mathcal{P}(become(a, \chi[p(z).become(a, x)])) = \{\}, \{p\}}}{\mathcal{P}(\chi[m(y).become(a, \chi[p(z).become(a, x)])]) = \{m\}, \{p\}} \quad \dfrac{\mathcal{P}(\{m\}) = \{m\}, \{\}}{\mathcal{P}([\{m\}/x]become(a, x)) = \{\}, \{m\}}}{\mathcal{P}(\texttt{letrec}\ x = \chi[m(y).become(a, \chi[p(z).become(a, x)])]\ \texttt{in}\ become(a, x)) = \{\}, \{m, p\}}$$

Une fois le potentiel défini, la sémantique est complète, et il est possible d'analyser les sources d'erreurs d'exécution. Tout d'abord, notons qu'il y a deux familles d'erreurs. La première forme est classique et correspond aux erreurs usuelles des langages fonctionnels : une valeur est manipulée dans un contexte erroné. Par exemple, le programme essaye d'utiliser *nop* comme une fonction. La seconde forme d'erreurs l'est moins, il s'agit d'une mauvaise utilisation d'une interface de communication : un message d'étiquette m est envoyé à un acteur qui ne saura jamais traiter ce message, ou bien qui sait le traiter mais son argument n'est pas correct.

Dans la réduction fonctionnelle, les erreurs possibles sont :

- une variable n'est pas définie (VARE) ;
- une valeur qui n'est pas une fonction est appliquée (APPE) ;

Dans la partie concurrente, les erreurs suivantes peuvent se produire :

- l'envoi d'une valeur qui n'est pas un message ou l'envoi d'un message à une valeur qui n'est pas une adresse (SENDE) ;
- la création d'un acteur en lui donnant une valeur qui n'est pas un comportement (*i.e.* une fonction qui prend en paramètre une adresse et renvoie une interface) (INITE) ;
- un changement de comportement avec une valeur qui n'est pas une interface ou dont le premier argument n'est pas une adresse (BECE) ;

– un message est reçu mais il ne figure pas dans le potentiel de son destinataire (c'est donc un orphelin trivial) (RcvE) ;

Parmi ces erreurs possibles, seules les deux dernières ne sont détectées par les systèmes de type usuels. Le but du système de type qui va être introduit est donc d'éliminer toutes ces erreurs potentielles en se basant sur les systèmes de types fonctionnels usuels. Cependant, la détection des deux dernières erreurs impose de procéder à une analyse plus fine des communications à travers des types suffisamment précis.

4.4 Les types

Dans cette section, nous définissons précisément les types et la relation de sous-typage utilisés pour ce premier système de type.

Types

Analysons en détail l'approximation des entités concurrentes : les messages, les interfaces et les acteurs. Nous avons vu dans le chapitre précédent qu'un message est une valeur construite à partir d'une étiquette. À l'image des *labels d'enregistrement* en Objective CAML, ces étiquettes ne sont pas des valeurs manipulables dans le langage. Nous exploitons donc une méthode similaire à celle utilisée dans Objective CAML pour approximer les valeurs de messages, les interfaces et les acteurs suivant le modèle des *rangées*[31]. Les rangées ont été initialement conçues par M. Wand pour typer des objets simples dans [Wan87]. Elles ont également été utilisées pour le typage des enregistrements extensibles (voir par exemple [Rém94]). Elles sont maintenant utilisées dans de nombreux systèmes de type, on peut par exemple citer les travaux de F. Pessaux sur la détection des exceptions qui ne sont pas rattrapées ([Pes99]), ceux de D. Rémy et J. Vouillon pour le typage de la partie objet d'Objective CAML (voir [RV98]) ou encore ceux de F. Pottier qui cherche à développer un solveur généraliste de contraintes de type dans lequel la notion de rangée est utilisée pour abstraire les ensembles (voir par exemple [Pot98]). Pour nous, une rangée est une fonction partielle dont le domaine est inclus dans $\mathbb{M}$. Elle associe à une étiquette m le type des paramètres d'un message d'étiquette m ainsi qu'un *drapeau* indiquant si le message est reçu ($\circ$) ou accepté ($\bullet$). La notation $@\{m_1 : \circ T_1,\ m_2 : \bullet T_2,\ i\}$ décrit ainsi le type d'un acteur recevant un message étiqueté m_1 transportant une donnée de type T_1 et acceptant les messages étiquetés m_2 transportant des données de type T_2. La variable i indique qu'une partie du type est inconnue pour l'instant.

Notons que dans le cadre de $\mathcal{F}unc_0$, les messages sont des valeurs du premier ordre. Il est ainsi possible d'écrire la fonction : `let` $send\_a = \lambda m.send(m, a)$ où l'adresse a est définie par le contexte. Si cette fonction est appelée plusieurs fois, plusieurs messages sont envoyés à a et il faut tous les collecter dans le type. Ainsi, si l'on abstrait le message émis m, il est possible qu'il soit instancié plusieurs fois (à chaque appel de $send\_a$) et que le type contienne alors plusieurs messages. Notons cependant, que dans la plupart des cas, cet ensemble est réduit à un élément : le message effectivement envoyé.

31. Le terme anglais *row* est plus familier.

Une interface est approximée par l'ensemble des messages qu'elle peut accepter. De plus, cette approximation tient compte du type du paramètre des messages éventuellement reçus, elle est ainsi plus précise que le calcul du potentiel introduit dans la section précédente. En fait, on montrera que le potentiel d'un acteur est une abstraction du type de son interface dans la preuve de correction du système de type (voir page 98).

Enfin, un acteur est approximé en collectant les messages qui lui sont destinés et en y ajoutant son interface.

Les rangées vérifient un grand nombre de propriétés, cependant nous ne présentons ici que celles qui sont nécessaires à notre système et nous renvoyons le lecteur intéressé à [Rém94].

- L'ordre d'apparition des étiquettes dans une rangée n'est pas important et peut être modifié : $\{m_1 : p_1,\ m_2 : p_2,\ i\} = \{m_2 : p_2,\ m_1 : p_1,\ i\}$.
- Une étiquette ne figure qu'une seule fois dans une rangée.
- On peut extraire d'une rangée se terminant par une variable une étiquette qui n'y figure pas initialement. Cette possibilité est appelée *l'expansion à la demande*, elle est assez intuitive mais délicate d'un point de vue théorique. Elle est utilisée pour comparer deux rangées ne contenant pas les mêmes étiquettes. Par exemple, si le système cherche à comparer $\{m_1 : p_1,\ m_2 : p_2,\ i_1\}$ et $\{m_2 : p'_2,\ i_2\}$, il doit *expanser* i_2 en $\{m_1 : p'_1,\ i'_2\}$ où p'_1 et i'_2 sont de nouvelles variables. La comparaison a alors lieu entre $\{m_1 : p_1,\ m_2 : p_2,\ i_1\}$ et $\{m_2 : p'_2,\ m_1 : p'_1,\ i'_2\}$ qui contiennent les mêmes étiquettes. Cependant, si le dernier élément de la rangée n'est pas une variable, l'expansion n'est pas possible.

Dans l'explication de la troisième propriété, nous avons utilisé un raccourci syntaxique que nous utiliserons désormais systématiquement, l'oubli des accolades des sous-rangées : $\{m_1 : p_1,\ \{m_2 : p_2,\ i\}\}$ est notée $\{m_1 : p_1,\ m_2 : p_2,\ i\}$ et $\{m_1 : p_1,\ \{\}\}$ est notée $\{m_1 : p_1\}$.

Nous pouvons donc maintenant introduire les types que nous utilisons sur $\mathcal{F}unc_0$. Ils suivent la définition générale 3.2 donnée page 59 et précisent l'ensemble des types de base, celui des constructeurs de type et la forme des types interfaces.

Définition 4.6 (Types) :
L'ensemble des ***types*** *est noté $\mathbb{T}$, sa définition est paramétrée par l'ensemble des variables de type (notées t et appartenant à $\mathbb{V}_t$), celui des variables de rangée (notées i et appartenant à $\mathbb{V}_i$) et enfin, celui des variables de présence (notée p et appartenant à $\mathbb{V}_p$). Un type est un terme construit par la grammaire suivante :*

$$\begin{array}{lll} T ::= \mathit{unit} \mid t \mid T \to T \mid I \mid \triangleright I \mid @I & & \textit{types} \\ I ::= \{\} \mid i \mid \{m : P,\ I\} & & \textit{rangées} \\ P ::= \dagger \mid p \mid \bullet T \mid \circ T & & \textit{présences} \end{array}$$

Un type est ***clos*** *si et seulement si il ne contient pas de variable (de $\mathbb{V}_t \cup \mathbb{V}_i \cup \mathbb{V}_p$).*

Pour que l'expansion d'une rangée soit toujours possible nous introduisons le terme de présence $\dagger$ qui signifie que l'étiquette à laquelle il est associé doit être absente. Ainsi, pour toute étiquette m, $\{\} = \{m : \dagger\}$. Ainsi, $\{m : P\}$ peut être expansé en $\{m : P,\ m' : \dagger\}$.

$$\frac{t'_1 \sqsubseteq t_1 \quad t_2 \sqsubseteq t'_2}{t_1 \to t_2 \sqsubseteq t'_1 \to t'_2} \qquad \frac{t' \sqsubseteq t}{@t \sqsubseteq @t'} \qquad \frac{t \sqsubseteq t'}{\triangleright t \sqsubseteq \triangleright t'}$$

$$\frac{}{\{\} \sqsubseteq i} \qquad \frac{p_1 \sqsubseteq p_2 \qquad i_1 \sqsubseteq i_2}{\{m : p_1, i_1\} \sqsubseteq \{m : p_2, i_2\}}$$

$$\frac{t_1 \sqsubseteq t_2}{\circ t_1 \sqsubseteq \circ t_2} \qquad \frac{t_1 \sqsubseteq t_2}{\circ t_1 \sqsubseteq \bullet t_2} \qquad \frac{t_2 \sqsubseteq t_1}{\bullet t_1 \sqsubseteq \bullet t_2}$$

Règles 4.4 La relation de sous-typage pour $\mathcal{F}unc_0$.

Relions plus précisément la notion de potentiel et la notion de type d'un acteur, en définissant la notion d'abstraction (Abs) d'un type clos :

$$Abs(@I) = Abs_I(I) \qquad Abs(\triangleright I) = Abs_I(I) \qquad Abs(T) = \{\} \text{ sinon}$$

en utilisant l'abstraction des rangées (Abs_I) :

$$Abs_I(\{\}) = \{\} \qquad Abs_I(\{m : \circ T,\ I\}) = Abs(I) \qquad Abs_I(\{m : \bullet T,\ I\}) = \{m\} \cup Abs(I)$$

L'abstraction est ensuite étendue aux types contenant des variables en collectant toutes les inclusions dues aux contraintes et à la règle suivante :

$$I_1 \sqsubseteq I_2 \Longrightarrow Abs_I(I_1) \subseteq Abs_I(I_2)$$

On montrera dans la preuve de correction de notre système de type que l'abstraction du type d'un acteur est une approximation inférieure de son potentiel initial.

Sous-typage

Le sous-typage suit également la définition du chapitre 2 en la spécialisant :

Définition 4.7 (Sous-typage) :
*Un type t_1 est **sous-type** d'un type t_2 si et seulement si il est possible de dériver une preuve de $t_1 \sqsubseteq t_2$ en utilisant le système de règle 4.4.*

Dans le cadre de nos travaux, le sous-typage est utilisé comme un moyen de collecter les différentes contraintes dues à l'utilisation des entités concurrentes (messages, acteurs et interfaces). Ainsi, une contrainte $@\varphi_1 \sqsubseteq @\varphi_2$ (qui est équivalent par sous-typage à $\varphi_2 \sqsubseteq \varphi_1$) signifie que l'acteur 1 (de type $@\varphi_1$) est utilisé dans un contexte lui donnant comme type $@\varphi_2$ et donc que son type final ($@\varphi_1$) doit contenir les éléments (messages reçus et messages installés) collectés par le type $@\varphi_2$.

Informellement, une variable de rangée peut être vue comme une forme d'ensemble. La rangée vide est la plus petite des rangées et le sous-typage de deux rangées s'effectue champ par champ de manière covariante. Le sous-typage des termes de présence est :

- covariant entre messages reçus ;
- covariant entre un message reçu et le message associé installé ;
- mais contravariant pour les messages installés.

En effet, il faut :

- cumuler les contraintes sur les messages reçus. Par exemple, si un acteur sait qu'il a reçu un réel, et reçoit un entier, il est inutile qu'il conserve l'information de réception de l'entier. Car, si il sait traiter les réels, il saura traiter les entiers. Ainsi, l'ensemble de contraintes $\{\circ int \sqsubseteq t,\ \circ float \sqsubseteq t\}$ se réduit à la seconde contrainte car $int \sqsubseteq float$ implique $\circ int \sqsubseteq \circ float$.
- cumuler les contraintes entre messages reçus et messages installés. Un message reçu est correct si un message plus grand est installé. Par exemple, si l'acteur comprend des messages contenant des réels, il peut recevoir des messages contenant des entiers. L'information de réception des entiers devient inutile. Ainsi, $int \sqsubseteq float$ implique $\circ int \sqsubseteq \bullet float$ et donc l'ensemble de contraintes $\{\circ int \sqsubseteq t,\ \bullet float \sqsubseteq t\}$ se réduit à la seconde contrainte
- faire l'intersection de tous les messages installés. Un acteur qui assume des comportements redéfinissant une réaction doit uniquement accepter les messages respectant la contrainte la plus sévère. En effet, si dans un état, un acteur accepte des entiers puis dans un autre des réels (avec la même étiquette), il ne peux recevoir de manière sûre que des messages contenant des entiers puisque l'état dans lequel il sera lors de la réception n'est pas prévisible dans la majorité des cas. Comme $int \sqsubseteq float$ implique $\bullet float \sqsubseteq \bullet int$, $\{\bullet int \sqsubseteq t,\ \bullet float \sqsubseteq t\} = \{\bullet int \sqsubseteq t\}$.

Notons que le sous-typage entre entiers et réels est utilisé uniquement pour décrire simplement le sous-typage des termes de présence. En effet, le système de type que nous utilisons ne contient pas cette forme de sous-typage mais travaille sur les ensembles de messages. Cependant, nous pensons que l'ajouter temporairement pour des exemples permet de simplifier les explications.

Un exemple

Présentons un autre exemple plus réaliste qui justifie et explique ces relations de sous-typage. Soit un acteur a qui reçoit deux messages m, un premier contenant un acteur sachant traiter des messages m_1 et un second contenant un acteur sachant traiter des messages m_1 et des messages m_2 ne contenant rien. Par exemple, dans le cadre de la configuration suivante :

$$a \triangleleft m(a_1) \parallel a \triangleleft m(a_2) \parallel a_1 \triangleright \chi[m_1(x) \dots] \parallel a_2 \triangleright \chi[m_1(x) \dots\ m_2(y) \dots]$$

Supposons, de plus, qu'après réception dem_1, a_1 envoie les messages $p(nop)$ et $q(nop)$ à l'acteur capturé par x et que a_2 envoie un message $p(nop)$ à l'acteur capturé par x.

Si une interface comprenant les messages $m(arg)$ est installée sur a, quels sont les types possibles pour arg ? En d'autres termes, quelles sont les opérations que l'on peut réaliser sur cet argument sans risquer de provoquer d'erreurs d'exécution ? Nous n'avons pas encore présenté le système de type ni la résolution des contraintes, mais on peut cependant en donner un petit aperçu. Les affirmations précédentes se concrétisent par les

contraintes suivantes, où a est de type t_a et arg de type t_{arg} :

$$\left\{ \begin{array}{c} t_a \sqsubseteq @\{m : \circ t_1\} \ , \ t_a \sqsubseteq @\{m : \circ t_2\} \ , \ t_a \sqsubseteq @\{m : \bullet t_{arg}\} \\ t_1 \sqsubseteq @\{m_1 : \bullet @\{p : \circ unit \ \ q : \circ unit\}\} \ , \ t_2 \sqsubseteq @\{m_1 : \bullet @\{p : \circ unit\}, \ m_2 : \bullet unit\} \end{array} \right\}$$

Dans la suite, nous noterons A_p pour $@\{p : \circ unit\}$ et A_{pq} pour $@\{p : \circ unit \ \ q : \circ unit\}$.

La variable dont le type est t_a correspond donc à un acteur qui contient au moins le message m, soit, après expansion de t_a : $t_a = @\{m : p_m, \ i_a\}$. Ce qui permet de réécrire le système de contraintes sous la forme :

$$\{\circ t_1 \sqsubseteq p_m, \ \ \circ t_2 \sqsubseteq p_m, \ \ \bullet t_{arg} \sqsubseteq p_m, \ \ t_1 \sqsubseteq @\{m_1 : \bullet A_{pq}\}, \ \ t_2 \sqsubseteq @\{m_1 : \bullet A_p, \ m_2 : \bullet unit\}\}$$

Compte tenu de la troisième inégalité et de la définition du sous-typage des termes de présence, on a $p_m = \bullet t_3$. Soit :

$$\{t_1 \sqsubseteq t_3 \ , \ \ t_2 \sqsubseteq t_3 \ , \ \ t_3 \sqsubseteq t_{arg} \ , \ \ t_1 \sqsubseteq @\{m_1 : \bullet A_{pq}\} \ , \ \ t_2 \sqsubseteq @\{m_1 : \bullet A_p, \ m_2 : \bullet unit\}\}$$

La variable intermédiaire t_3 peut donc être supprimée (car elle est inutile). De plus, les deux dernières inégalités contraignent la forme de t_1 et celle de t_2 et donc par transitivité celle de t_{arg}. Donc, en posant $t_i = @\phi_i$, le système devient :

$$\{\phi_{arg} \sqsubseteq \phi_1 \ , \ \ \phi_{arg} \sqsubseteq \phi_2 \ , \ \ \{m_1 : \bullet A_{pq}\} \sqsubseteq \phi_1 \ , \ \ \{m_1 : \bullet A_p, \ m_2 : \bullet unit\} \sqsubseteq \phi_2\}$$

Or, nous verrons que l'envoi d'un message $m_?(t_?)$ à un acteur de type $@\phi_?$ se traduit par l'inégalité $\{m_? : \circ t_?\} \sqsubseteq \phi_?$. Donc, si trois messages $m_k(\alpha_k)$ sont envoyés (où k vaut 1,2 et 3) à arg (dont le type est $@\phi_{arg}$), le système de contraintes devient :

$$\left\{ \begin{array}{c} \{m_1 : \circ\alpha_1\} \sqsubseteq \phi_1 \ , \ \ \{m_1 : \circ\alpha_1\} \sqsubseteq \phi_2 \ , \ \ \{m_2 : \circ\alpha_2\} \sqsubseteq \phi_1 \ , \\ \{m_2 : \circ\alpha_2\} \sqsubseteq \phi_2 \ , \ \ \{m_3 : \circ\alpha_3\} \sqsubseteq \phi_1 \ , \ \ \{m_3 : \circ\alpha_3\} \sqsubseteq \phi_2 \\ \{m_1 : \bullet A_{pq}\} \sqsubseteq \phi_1 \ , \ \ \{m_1 : \bullet A_p, \ m_2 : \bullet unit\} \sqsubseteq \phi_2 \end{array} \right\}$$

On en déduit par expansion la valeur de ϕ_1 et ϕ_2 : $\phi_j = \{m_1 : p_1^j, \ m_2 : p_2^j, \ m_3 : p_3^j, \ i_j\}$ avec j valant 1 ou 2 et le système devient :

$$\left\{ \begin{array}{c} \circ\alpha_1 \sqsubseteq p_1^1 \ , \ \ \circ\alpha_1 \sqsubseteq p_1^2 \ , \ \ \circ\alpha_2 \sqsubseteq p_2^1 \ , \ \ \circ\alpha_2 \sqsubseteq p_2^2 \ , \ \ \circ\alpha_3 \sqsubseteq p_3^1 \ , \ \ \circ\alpha_3 \sqsubseteq p_3^2 \\ \bullet A_{pq} \sqsubseteq p_1^1 \ , \ \ \bullet A_p \sqsubseteq p_1^2 \ , \ \ \bullet unit \sqsubseteq p_2^2 \end{array} \right\}$$

Finalement, le système se réduit en :

$$\{\alpha_1 \sqsubseteq A_{pq} \ , \ \ \alpha_1 \sqsubseteq A_p \ , \ \ \alpha_2 \sqsubseteq unit \ , \ \ \circ\alpha_2 \sqsubseteq p_2^1 \ , \ \ \circ\alpha_3 \sqsubseteq p_3^1 \ , \ \ \circ\alpha_3 \sqsubseteq p_3^2\}$$

Ce qui nous permet de conclure, que les messages envoyés à arg d'étiquette m_1 (resp. m_2) doivent contenir une donnée de type A_{pq} (resp. rien). De plus, l'envoi d'un message m_3 apparaîtra uniquement sous la forme $\circ$ dans le type de arg. Or, nous verrons que c'est la condition pour qu'un message soit un orphelin trivial.

La conclusion du typage est donc : la réaction associée à m dans une interface installée sur a ne peut envoyer sans erreur à son argument que des messages m_1 et m_2 dont les arguments respectifs doivent contenir un acteur de type A_{pq} et rien.

Nous pouvons donc maintenant présenter les règles de typage de $\mathcal{F}unc_0$.

4.5 Le système de type

Dans cette section, nous allons introduire les règles de typage de $\mathcal{F}unc_0$ ainsi que les règles (non nécessaires) de typage du sucre syntaxique ajouté.

Les expressions

Les règles de typage des expressions sont pour la plupart issues du λ-calcul avec appel par valeur et introduction de définitions récursives par les `letrec`. Leur forme est : $\mathcal{E} \vdash_e e : t$ où $\mathcal{E}$ est l'environnement local de typage de l'expression e et t est son type. Toute variable libre dans une règle est considérée comme nouvelle.

Le typage d'une utilisation de variable consiste à renvoyer le type associé à la variable dans l'environnement de typage (règle VAR). Le typage du *nop* (règle NOP) renvoie le type *unit*. L'application (règle APP) est typée de manière curryfiée, le type de la fonction doit être un sous-type d'un type fonctionnel dont l'argument est le type du paramètre réel de l'appel et dont le résultat est une variable de type fraîche. Le typage d'une abstraction (règle FUN) est classique. Lors d'une définition récursive (règle LREC), le corps e est typé dans l'environnement étendu par l'introduction de x. Enfin, la suite du programme est typée dans cet environnement étendu pour donner le type du `letrec`. Au cours de ce typage, une contrainte de sous-typage entre le type du corps et le type de la variable ainsi définie est générée pour relier le type calculé au type supposé. Notons que nous ne nous intéressons pas au problème de la détection des définitions récursives faisant boucler le programme.

Le typage des entités concurrentes est un plus complexe et inhabituel. Le typage du message est immédiat selon la définition donnée des types de messages, le type de l'expression argument donne le type associé à m dans la rangée correspondante (règle MESS). L'envoi d'un message résulte en rien (*unit*), il impose au second argument d'être un acteur acceptant les messages contenus dans le type du premier argument (règle SEND). Le typage d'une interface (règle INTER) produit un type interface contenant tous les messages installés qu'elle définie. La création d'une nouvelle adresse d'acteur (règle NEW) produit un nouveau type acteur sur lequel aucune contrainte n'existe encore. La création d'un acteur a (règle INIT) utilise son premier argument comme adresse de a et son second comme comportement initial. Son premier argument doit donc être un type acteur et son second une fonction prenant en argument un type acteur et produisant un type interface. Les messages envoyés à l'*ego* et ceux installés dans le corps du comportement doivent alors être collectés dans le type de l'adresse. Ainsi, l'application $init(a, \lambda ego.i)$, où a est de type $@t_a$, *ego* de type $@t_{ego}$ et i de type $\triangleright t_i$, génère les contraintes $@t_a \sqsubseteq @\varphi$ et $@t_{ego} \to \triangleright t_i \sqsubseteq @\varphi \to \triangleright\varphi$. Et ces contraintes se simplifient en : $\varphi \sqsubseteq t_a$, $t_{ego} \sqsubseteq \varphi$ et $t_i \sqsubseteq \varphi$, ce qui permet de cumuler dans t_a les contenus de t_{ego} et t_i. Enfin, le changement de comportement (règle BEC) est voisin de la création. La fonction *become* crée un acteur anonyme qui poursuit le calcul à partir de l'adresse de l'acteur courant et de sa future interface, puis change l'interface de l'acteur courant. Son typage est donc proche de celui de *init*, une fonction qui prend en argument un couple et ne renvoie aucun résultat. Ainsi, lors de l'appel de $become(a, i)$, où a est de type $@t_a$ et i de type $\triangleright t_i$, génère les contraintes $@t_a \sqsubseteq @\varphi$ et $\triangleright t_i \sqsubseteq \triangleright\varphi$. Ce qui conduit à $t_i \sqsubseteq \varphi \sqsubseteq t_a$, le contenu du type

$$\textsc{Var}: \frac{x \in dom(\mathcal{E})}{\mathcal{E} \vdash_e x : \mathcal{E}(x)} \qquad \textsc{Nop}: \frac{}{\mathcal{E} \vdash_e nop : unit} \qquad \textsc{App}: \frac{\mathcal{E} \vdash_e f : t_f \quad \mathcal{E} \vdash_e e : t_e}{\mathcal{E} \vdash_e f\ e : t} \left(t_f \sqsubseteq t_e \to t\right)$$

$$\textsc{Fun}: \frac{\mathcal{E}+\{x : t_x\} \vdash_e e : t}{\mathcal{E} \vdash_e \lambda x.e : t_x \to t} \qquad \textsc{Lrec}: \frac{\mathcal{E}+\{x : t_x\} \vdash_e e : t \quad \mathcal{E}+\{x : t_x\} \vdash_e e' : t'}{\mathcal{E} \vdash_e \texttt{letrec}\ x\ =\ e\ \texttt{in}\ e' : t'} \left(t \sqsubseteq t_x\right)$$

$$\textsc{Mess}: \frac{\mathcal{E} \vdash_e e : t}{\mathcal{E} \vdash_e m(e) : \{m : \circ t\}} \qquad \textsc{Send}: \frac{\mathcal{E} \vdash_e e_1 : t_1 \quad \mathcal{E} \vdash_e e_2 : t_2}{\mathcal{E} \vdash_e send(e_1, e_2) : unit} \left(t_2 \sqsubseteq @t_1\right)$$

$$\textsc{Inter}: \frac{\mathcal{E}+\{x_i : t_i\} \vdash_e e_i : t'_i}{\mathcal{E} \vdash_e \chi[m_i(x_i).e_i]^{i \in I} : \triangleright t} \left(\forall i \in I \ \ \{m_i : \bullet t_i\} \sqsubseteq t\right) \qquad \textsc{New}: \frac{}{\mathcal{E} \vdash_e new : @\varphi}$$

$$\textsc{Init}: \frac{\mathcal{E} \vdash_e e_1 : t_1 \quad \mathcal{E} \vdash_e e_2 : t_2}{\mathcal{E} \vdash_e init(e_1, e_2) : unit} \begin{pmatrix} t_1 \sqsubseteq @\varphi \\ t_2 \sqsubseteq @\varphi \to \triangleright\varphi \end{pmatrix}$$

$$\textsc{Bec}: \frac{\mathcal{E} \vdash_e e_1 : t_1 \quad \mathcal{E} \vdash_e e_2 : t_2}{\mathcal{E} \vdash_e become(e_1, e_2) : unit} \begin{pmatrix} t_2 \sqsubseteq \triangleright\varphi \\ t_1 \sqsubseteq @\varphi \end{pmatrix}$$

Règles 4.5 Le typage des expressions de $\mathcal{F}unc_0$.

du second argument est donc collecté dans celui du second. Une des raisons imposant au changement de comportement (*become*) de prendre en paramètre l'adresse courante de l'acteur provient de la nécessité de relier le type de la future interface de l'acteur au type de l'acteur lui-même. En effet, comme le type de l'acteur approxime toutes ces interfaces, il faut qu'il contienne toutes les futures interfaces. Remarquons que nous supposons que le premier argument du *become* est bien son *ego*.

Les exemples présentés par la suite utilisent le sucre syntaxique (le `let` et la séquence) introduit dans la première section et dont le typage suit les règles suivantes :

$$\textsc{Let}: \frac{\mathcal{E} \vdash_e e_1 : t_1 \quad \mathcal{E}+\{x : t_1\} \vdash_e e_2 : t_2}{\mathcal{E} \vdash_e \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 : t_2} \qquad \textsc{Seq}: \frac{\mathcal{E} \vdash_e e_1 : t_1 \quad \mathcal{E} \vdash_e e_2 : t_2}{\mathcal{E} \vdash_e e_1\,; e_2 : t_2}$$

Pour mieux appréhender ces différentes règles, analysons précisément la construction de la preuve du typage d'un exemple.

4.6 Un exemple de typage

Pour mieux illustrer le fonctionnement de notre système de type, examinons le typage du programme ci-dessous dans lequel nous supposons qu'un acteur de nom `act` (et de type $@\psi_{act}$) a été précédemment défini et figure donc dans l'environnement $\mathcal{E}$.

$$\texttt{let}\ f = \lambda a.\lambda b.(send(a, b))\ \texttt{in}\ f\ (m_1())\ \texttt{act}\,; f\ (m_2())\ \texttt{act}$$

La dérivation suivante (qui dépend de i) permet d'inférer les types du corps du `let` :

$$\mathrm{E(i)} : \frac{\dfrac{\mathcal{E}_f \vdash_e f : t_f \quad \mathcal{E}_f \vdash_e m_i() : \{m_i : \circ unit\}}{\mathcal{E}_f \vdash_e f\ (m_i()) : \alpha_1^i}\,(1) \qquad \mathcal{E}_f \vdash_e act : @\psi_{act}}{\mathcal{E}_f \vdash_e f\ (m_i())\ act : \alpha_2^i}\,(2)$$

On peut alors donner la dérivation de typage du programme :

$$\frac{\dfrac{\dfrac{\dfrac{\mathcal{E}_{ab} \vdash_e a : t_a \quad \mathcal{E}_{ab} \vdash_e b : t_b}{\mathcal{E}_{ab} \vdash_e send(a,b) : unit}\,(3)}{\mathcal{E}_a \vdash_e \lambda b.(send(a,b)) : t_b \to unit}}{\mathcal{E} \vdash_e \lambda a.\lambda b.(send(a,b)) : t_a \to t_b \to unit} \qquad \frac{\mathrm{E}(1) \qquad \mathrm{E}(2)}{\mathcal{E}_f \vdash_e f\ (m_1())\ act ; f\ (m_2())\ act : \alpha_2^2}}{\mathcal{E} \vdash_e \texttt{let}\ f\ = \lambda a.\lambda b.(send(a,b))\ \texttt{in}\ f\ (m_1())\ act ; f\ (m_2())\ act : \alpha_2^2}\,(4)$$

avec $\mathcal{E}_f = \mathcal{E}{+}\{f\colon t_f\}$, $\mathcal{E}_a = \mathcal{E}{+}\{a\colon t_a\}$ et $\mathcal{E}_{ab} = \mathcal{E}_a{+}\{b\colon t_b\}$ et les contraintes :

$$\begin{cases} 1 : & t_f \sqsubseteq \{m_i : \circ unit\} \to \alpha_1^i & \qquad 3 : & t_b \sqsubseteq @t_a \\ 2 : & \alpha_1^i \sqsubseteq @\psi_{act} \to \alpha_2^i & \qquad 4 : & t_a \to t_b \to unit \sqsubseteq t_f \end{cases}$$

La résolution du système de contraintes obtenu suit alors les étapes suivantes :

– on applique la transitivité du sous-typage et t_f qui est inutile est oubliée :

$$\begin{cases} t_b \sqsubseteq @t_a \\ t_a \to t_b \to unit \sqsubseteq \{m_1 : \circ unit\} \to \alpha_1^1 & \qquad \alpha_1^1 \sqsubseteq @\psi_{act} \to \alpha_2^1 \\ t_a \to t_b \to unit \sqsubseteq \{m_2 : \circ unit\} \to \alpha_1^2 & \qquad \alpha_1^2 \sqsubseteq @\psi_{act} \to \alpha_2^2 \end{cases}$$

– les types fonctionnels sont alors décomposés :

$$\begin{cases} t_b \sqsubseteq @t_a \\ \{m_1 : \circ unit\} \sqsubseteq t_a & \qquad t_b \to unit \sqsubseteq \alpha_1^1 \sqsubseteq @\psi_{act} \to \alpha_2^1 \\ \{m_2 : \circ unit\} \sqsubseteq t_a & \qquad t_b \to unit \sqsubseteq \alpha_1^2 \sqsubseteq @\psi_{act} \to \alpha_2^2 \end{cases}$$

– on réapplique la transitivité, certaines variables inutiles sont supprimées (α_1^1 et α_1^2) et les types fonctionnels sont décomposés :

$$\{m_1 : \circ unit\} \sqsubseteq t_a \quad \{m_2 : \circ unit\} \sqsubseteq t_a \quad @\psi_{act} \sqsubseteq t_b \sqsubseteq @t_a \;\; unit \sqsubseteq \alpha_2^1 \;\; unit \sqsubseteq \alpha_2^2$$

– la variable t_b devenue inutile est alors oubliée, comme nous le verrons dans la section suivante une variable comparable à *unit* vaut *unit*, soit $\alpha_2^1 = \alpha_2^2 = unit$ et on applique le sous-typage des acteurs :

$$\{m_1 : \circ unit\} \sqsubseteq t_a \sqsubseteq \psi_{act} \qquad \{m_2 : \circ unit\} \sqsubseteq t_a \sqsubseteq \psi_{act}$$

– l'ensemble de contraintes final est donc :

$$\{m_1 : \circ unit,\ m_2 : \circ unit\} \sqsubseteq \psi_{act}$$

On en déduit donc qu'il faut que *act* comprenne les messages m_1 et m_2 et que le type de l'expression est *unit*.

4.7 La résolution des contraintes

Dans le deuxième chapitre de cette thèse, nous avons introduit rapidement deux techniques pour résoudre les contraintes engendrées par notre système d'inférence de type.

Présentons cette phase de résolution pour $\mathcal{F}unc_0$ en utilisant une notion de graphe de contraintes (voir page 65 ou [Pot98]). En effet, nous combinons sous-typage et unification lors de la résolution des contraintes suivant ainsi une approche similaire à celle de M. Fähndrich et A. Aiken dans [FA97]. Cette stratégie permet de limiter le sous-typage aux entités concurrentes (acteurs, interfaces ou messages), on augmente alors nettement l'efficacité de la résolution et la forme des types *à la ML* est conservée. Ainsi, la résolution va manipuler un graphe de contraintes et une valuation. Plus précisément :

Définition 4.8 (Ensemble de contraintes) :
Un ensemble de contraintes (C, σ) *est composé de :*

- *un graphe de contraintes clos* C *;*
- *une substitution* σ *dont le domaine est inclus dans* $\mathbb{V}$ *et dont le codomaine est inclus dans l'union de l'ensemble des types*[32] $\mathbb{T}$ *avec l'ensemble des présences* $\mathbb{P}$ *;*

Ce couple doit vérifier $dom(C) \cap dom(\sigma) = \varnothing$ *et* $\forall V \in \mathbb{T} \cup \mathbb{P} \;\; Var(\sigma(V)) \cap dom(\sigma) = \varnothing$.

La seconde contrainte permet de conserver une valuation non récursive, une seule application de σ permet de construire un type qui ne contient aucune variable dont la valeur est connue (elle figure dans le domaine de la valuation). Pour simplifier les notations, l'extension de σ à l'ensemble des contraintes est également notée σ.

La résolution suit un processus incrémental, elle débute avec un ensemble de contraintes vide $(\varnothing, [])$ et à chaque ajout de contrainte dans cet ensemble, elle conserve cette structure. S'il n'est pas possible de la maintenir, cela signifie qu'une contrainte contradictoire avec les contraintes déjà inférées a été générée. Dans ce cas l'ajout échoue et signale une erreur de typage.

Chaque contrainte inférée est simplifiée avant d'être ajoutée au graphe de contraintes. Pour cela, définissons la fonction *subc* qui, à partir d'une contrainte, calcule un ensemble de contraintes simples. Rappelons tout d'abord qu'une contrainte est simple si elle a une des formes suivantes : $v_1 \sqsubseteq v_2$, $v \sqsubseteq T$, ou $T \sqsubseteq v$.

La figure 4.1 donne la définition de la simplification des contraintes. Elle décompose les contraintes portant sur des termes de même constructeurs de tête et provoque une erreur dans tous les autres cas.

Nous allons également utiliser :

- $C|_v$ qui restreint le graphe de contrainte en lui ôtant v ;
- $C|^v(V)$ qui produit les contraintes engendrées par la valuation de la variable v du graphe de contrainte C (V est la nouvelle valeur de v) :

$$C|^v(V) = \{v' \sqsubseteq V \mid v' \in dom(C) \wedge v' \leq_C v\} \cup \{V \sqsubseteq v' \mid v' \in dom(C) \wedge v \leq_C v'\} \cup \{V \sqsubseteq C^{\uparrow}(v);\; C^{\downarrow}(v) \sqsubseteq V\}$$

32. Une rangée est également un type.

$$subc(T \sqsubseteq T) = \{\} \quad (A) \qquad subc(v_1 \sqsubseteq v_2) = \{v_1 \sqsubseteq v_2\} \quad (B)$$

$$subc(v \sqsubseteq T) = \{v \sqsubseteq T\} \quad (C) \qquad subc(T \sqsubseteq v) = \{T \sqsubseteq v\} \quad (D)$$

$$subc(@T_1 \sqsubseteq @T_2) = \{T_2 \sqsubseteq T_1\} \quad (E) \qquad subc(\triangleright T_1 \sqsubseteq \triangleright T_2) = \{T_1 \sqsubseteq T_2\} \quad (F)$$

$$subc(T_1 \to T_2 \sqsubseteq T_1' \to T_2') = \{T_1' \sqsubseteq T_1;\ T_2 \sqsubseteq T_2'\} \quad (G) \qquad subc(\{\} \sqsubseteq I) = \{\} \quad (H)$$

$$subc(\{(m_j : P_j)_{j \in J}, I\} \sqsubseteq \{\}) = subc(I \sqsubseteq \{\}) \cup \bigcup_{j \in J} subc(P_j \sqsubseteq \dagger) \quad (I)$$

$$subc(\{(m_j : P_j)_{j \in J_1}, I_1\} \sqsubseteq \{(m_j : P_j')_{j \in J_2}, I_2)\} = subc(i_1' \sqsubseteq i_2') \cup \bigcup_{j \in J_1 \cup J_2} subc(P_j \sqsubseteq P_j') \quad (J)$$

$$\text{avec } \begin{cases} I_1 = \{(m_j : p_j)_{j \in J_2 \setminus J_1}, i_1'\} \\ I_2 = \{(m_j : p_j')_{j \in J_1 \setminus J_2}, i_2'\} \end{cases} \text{ et } \begin{cases} \forall j \in J_2 \setminus J_1 \quad P_j = p_j \\ \forall j \in J_1 \setminus J_2 \quad P_j' = p_j' \end{cases}$$

$$subc(\dagger \sqsubseteq \bullet T) = subc(\dagger \sqsubseteq \circ T) = \{\} \quad (K) \qquad subc(\bullet T_1 \sqsubseteq \bullet T_2) = \{T_2 \sqsubseteq T_1\} \quad (L)$$

$$subc(\circ T_1 \sqsubseteq \bullet T_2) = \{T_1 \sqsubseteq T_2\} \quad (M) \qquad subc(\circ T_1 \sqsubseteq \circ T_2) = \{T_1 \sqsubseteq T_2\} \quad (N)$$

Figure 4.1 Simplification de contraintes

– $hdc(V)$ qui calcule le constructeur de tête de V :

$$hdc(unit) = unit \qquad hdc(T_1 \to T_2) = \to \qquad hdc(\triangleright T) = \triangleright \qquad hdc(@T) = @$$

$$hdc(\{\}) = \{\} \qquad hdc(\{(m_J : P_J)_J, I\}) = \{\} \qquad hdc(\dagger) = \dagger \qquad hdc(\bullet T) = \bullet$$

$$hdc(\circ T) = \circ$$

– $mav(V)$ qui produit un type avec le même constructeur de tête que V et des variables fraîches :

$$mav(unit) = unit \qquad mav(T_1 \to T_2) = t_1 \to t_2 \qquad mav(\triangleright T) = \triangleright t$$

$$mav(@T) = @t \qquad mav(\{\}) = \{(), i\} \qquad mav(\{(m_J : P_J)_J, I\}) = \{(m_J : p_j)_J, i\}$$

$$mav(\dagger) = \dagger \qquad mav(\bullet T) = \bullet t \qquad mav(\circ T) = \circ t$$

L'ajout d'une contrainte lors de l'inférence va donc suivre les règles suivantes :

$$\begin{array}{lll}
(C, \sigma) \oplus \{c\} & \triangleq (C, \sigma) + \{subc(\sigma(c))\} & \\
(C, \sigma) + \{v_1 \sqsubseteq v_2\} & \triangleq (C + \{v_1 \sqsubseteq v_2\}, \sigma) & \\
(C, \sigma) + \{v \sqsubseteq \bullet T\} & \triangleq (C + \{v \sqsubseteq \bullet T\}, \sigma) & \\
(C, \sigma) + \{v \sqsubseteq V\} & \triangleq (C, \sigma \circ [mav(V)/v]) & \text{si } v \notin dom(C) \text{ et } hdc(V) \neq \bullet \\
(C, \sigma) + \{v \sqsubseteq V\} & \triangleq (C|_v, \sigma \circ [V'/v]) \oplus C|^v(V') \oplus \{V' \sqsubseteq V\} & \text{si } V' = mav(V) \text{ et } hdc(V) \neq \bullet \\
(C, \sigma) + \{V \sqsubseteq v\} & \triangleq (C + \{V \sqsubseteq v\}, \sigma) & \text{si } hdc(V) \in \{\dagger; \circ\} \\
(C, \sigma) + \{V \sqsubseteq v\} & \triangleq (C, \sigma \circ [mav(V)/v]) & \text{si } v \notin dom(C) \text{ et } hdc(V) \notin \{\dagger; \circ\} \\
(C, \sigma) + \{V \sqsubseteq v\} & \triangleq (C|_v, \sigma \circ [V'/v]) \oplus C|^v(V') \oplus \{V \sqsubseteq V'\} & \text{si } V' = mav(V) \text{ et } hdc(V) \notin \{\dagger; \circ\}
\end{array}$$

Théorème 4.1 (Correction et terminaison) :
L'algorithme consistant à partir d'un ensemble vide de contraintes à y ajouter les contraintes inférées par l'opérateur $\oplus$ est correct et se termine

PREUVE : Nous n'allons pas donner une démonstration complète de ce théorème mais seulement les grandes lignes. Il faut prouver trois choses : premièrement qu'après ajout la structure d'ensemble de contraintes est conservée, deuxièmement que l'ajout est correct, c'est à dire que le calcul donné produit bien un ensemble équivalent et enfin que ce calcul se termine.

Tout d'abord, l'opération d'ajout de contrainte $\oplus$ produit un ensemble de contraintes :

$$(C, \sigma) \oplus \{c\} = (C, \sigma) + \{subc(\sigma(c))\} = (C, \sigma) + \{c_1, \ldots, c_n\}$$

avec $Var = \bigcup_{i=1}^{n} Var(c_i)$ qui vérifie $Var \cap dom(\sigma) = \varnothing$

1. si $c_i = v_1 \sqsubseteq v_2$ ou $c_i = v \sqsubseteq \bullet T$ ou $c_i = V \sqsubseteq v$ avec $hdc(V) \in \{\dagger; \circ\}$:
   – $C + c_i$ est clos
   – $dom(C + c_i) = dom(C) \cup Var(c_i)$, or par hypothèse $Var(c_i) \cap dom(\sigma) \subseteq$

$Var \cap dom(\sigma) = \varnothing$
- σ est inchangé

2. si $c_i = v \sqsubseteq V$ avec $v \notin dom(C)$ et $hdc(V) \neq \bullet$ ou $c_i = V \sqsubseteq v$ avec $v \notin dom(C)$ et $hdc(V) \notin \{\dagger; \circ\}$ (on note V' pour $mav(V)$) :
   - C inchangé
   - $dom(\sigma \circ [V'/v]) = dom(\sigma) \cup \{v\}$, or comme v n'appartient pas au domaine de C : $dom(\sigma \circ [V'/v]) \cap dom(C) = \varnothing$
   - pour tout W, $Var(\sigma \circ [V'/v](W)) = (Var(\sigma(W)) \setminus \{v\}) \cup \{Var(V')\}$; il est trivial de montrer que $Var(V')$ ne contient que des variables fraîches et donc ne figurant pas dans $dom(\sigma)$; d'autre part, il est clair que $(Var(\sigma(W)) \setminus \{v\}) \cap dom(\sigma) = \varnothing$
3. si $c_i = v \sqsubseteq V$ avec $v \in dom(C)$ et $hdc(V) \neq \bullet$ ou $c_i = V \sqsubseteq v$ avec $v \in dom(C)$ et $hdc(V) \notin \{\dagger; \circ\}$: le raisonnement est similaire au cas précédent pour $(C|_v, \sigma \circ [V'/v])$ auquel, on ajoute des contraintes par $\oplus$ et donc hypothèse d'induction, la propriété est vérifiée

La correction de ces ajouts signifie qu'ils produisent un ensemble de contraintes qui possède les mêmes solutions que l'ensemble initial. La substitution de variables est trivialement correcte. La simplification de contraintes peut être montrée correcte par induction à partir des règles de sous-typage. Dans le cas 1 (voir ci-dessus), la correction repose sur un théorème de la thèse de F. Pottier. Dans le cas 2, on montre facilement par induction à partir des règles de sous-typage que v est bien égal à $mav(V)$. Enfin, dans le cas 3 :
- toutes les contraintes ne *concernant* pas v sont inchangées
- comme $v = V'$, il faut :
  - $V' \sqsubseteq^s V$
  - $V' \sqsubseteq C^{\uparrow}(v)$ et $C^{\downarrow}(v) \sqsubseteq V'$
  - $V' \sqsubseteq v'$ pour tout v' de $dom(C)$ qui vérifie $v' \leq_C v$ ou $v \leq_C v'$

  Or, ces trois composantes sont bien ajoutées au résultat.

La terminaison du processus d'ajout d'une contrainte repose sur la terminaison de l'ajout de contraintes simples dans le graphe de contraintes clos et sur la définition d'une notion de hauteur pour un type qui correspond au nombre de nœuds entre le constructeur de tête et la feuille la plus profonde. On peut montrer que l'ajout diminue la hauteur des types ou produit des contraintes de même profondeur mais qui diminue strictement la taille de C. Il n'est donc pas possible de boucler indéfiniment.

En fin d'ajout de contraintes dans le graphe, il convient alors de vérifier que les acteurs n'ont pas reçu de messages orphelins triviaux. Pour cela, une étape supplémentaire vérifie que les types des acteurs effectivement créés ne contiennent plus de messages reçus non installés, *i.e.* le type ne contient que des $\bullet$.

4.8 Correction du système de type

Prouvons maintenant la correction de notre système de type. Pour cela, nous allons procéder par étapes en démontrant de nombreux lemmes et théorèmes nécessaires à une

écriture simple de la preuve du théorème de correction.

La technique de preuve est une adaptation des preuves classiques de correction des systèmes de type utilisés dans le cadre de ML. Elle a été mise au point dans le cadre de λCap (où le calcul fonctionnel est plus simple) en collaboration avec M. Colin. Dans [DPCS00], elle est résumée dans le cadre d'une sémantique ne contenant pas les deux réductions.

Typage du contexte d'évaluation

Les deux premiers lemmes énoncent les propriétés des contextes d'évaluation montrant que les contextes se comportent *bien* vis-à-vis du typage et du sous-typage.

Lemme 4.1 (Typage d'une sous-expression) :
Si une expression est typable, toutes ses sous-expressions sont typables. Soit, formellement :

$$\mathcal{E} \vdash_e \mathrm{C}[e] : t \Longrightarrow \exists \mathcal{E}',\ \ \mathcal{E}' \vdash_e e : t'$$

De plus, l'environnement $\mathcal{E}'$ peut valoir $\mathcal{E}$ pour les quatre premiers contextes et vaut $\mathcal{E}+\{x : t_x\}$ dans le cas d'un contexte `letrec`.

Preuve : Le typage de toutes les sous-expressions e d'un contexte $\mathrm{C}[e]$ figurent en prémisse de la déduction du typage de $\mathrm{C}[e]$ et ce pour tous les contextes. Donc, si $\mathrm{C}[e]$ est typé, e l'est et la forme de son environnement de typage $\mathcal{E}'$ est également immédiate puisque seul le `letrec` étend l'environnement de ses sous-expressions.

Lemme 4.2 (Typage d'un contexte) :
Si une expression $\mathrm{C}[e]$ a pour type t_1, le remplacement de la sous-expression e de type t_2 par une autre sous-expression e' de type t_3 sous-type de t_2, produit une nouvelle expression $\mathrm{C}[e']$ de type t_4 sous-type de t_1. Soit, formellement :

$$\mathcal{E}_1 \vdash_e \mathrm{C}[e] : t_1 \wedge \mathcal{E}_2 \vdash_e e : t_2 \wedge \mathcal{E}_2 \vdash_e e' : t_3 \wedge t_3 \sqsubseteq t_2 \Longrightarrow \mathcal{E}_1 \vdash_e \mathrm{C}[e'] : t_4 \wedge t_4 \sqsubseteq t_1$$

Preuve : La preuve se fait en analysant le contexte. Notons que d'après le lemme précédant, les environnements $\mathcal{E}_1$ et $\mathcal{E}_2$ sont égaux sauf dans le cas du `letrec` où $\mathcal{E}_2 = \mathcal{E}_1+\{x : t_x\}$.

- $\mathrm{C}[e] = f(e, e_1)$: trois cas se présentent alors selon la valeur de f :
  - $f = \mathit{send}$, comme e et $\mathrm{C}[e]$ sont typées par hypothèse, la dérivation suivante est possible :

$$\frac{\mathcal{E}_1 \vdash_e e' : t_3 \quad \mathcal{E}_1 \vdash_e e_1 : \alpha}{\mathcal{E}_1 \vdash_e \mathit{send}(e', e_1) : \mathit{unit}} \left(\alpha \sqsubseteq @t_3\right)$$

    Or, la contrainte est trivialement vérifiée puisque $\alpha \sqsubseteq @t_2 \sqsubseteq @t_3$. Le type de retour valant dans les deux cas *unit* le lemme est vérifié.
  - $f = \mathit{init}$, le même raisonnement s'applique, il suffit de vérifier que le nouvel ensemble de contraintes est impliqué par l'ancien, soit que

$$\{t_2 \sqsubseteq @\varphi;\ \alpha \sqsubseteq @\varphi \rightarrow \triangleright\varphi\} \Rightarrow \{t_3 \sqsubseteq @\varphi;\ \alpha \sqsubseteq @\varphi \rightarrow \triangleright\varphi\}$$

Ce qui est immédiat car $t_3 \sqsubseteq t_2 \sqsubseteq @\varphi$.
  - $f = become$, à nouveau le même raisonnement s'applique, il suffit de vérifier que le nouvel ensemble de contraintes est impliqué par l'ancien, *i.e.* $\{t_2 \sqsubseteq @\varphi;\ \alpha \sqsubseteq \triangleright\varphi\} \Rightarrow \{t_3 \sqsubseteq @\varphi;\ \alpha \sqsubseteq \triangleright\varphi\}$. Ce qui est immédiat car $t_3 \sqsubseteq t_2 \sqsubseteq @\varphi$.
- C$[e] = f(v, e)$: trois cas se présentent alors selon la valeur de f, ils suivent le même raisonnement que ci-dessus :
  - $f = send$, $t_3 \sqsubseteq t_2 \sqsubseteq @\alpha$ et la contrainte est donc trivialement impliquée.
  - $f = init$, $t_3 \sqsubseteq t_2 \sqsubseteq @\varphi \rightarrow \triangleright\varphi$ et la contrainte est donc trivialement impliquée.
  - $f = become$, $t_3 \sqsubseteq t_2 \sqsubseteq \triangleright\varphi$ et la contrainte est donc trivialement impliquée.
- C$[e] = m(e)$: par un raisonnement similaire, il suffit cette fois de vérifier que $\{m : \circ t_3\} \sqsubseteq \{m : \circ t_2\}$. Or, ceci est trivial du fait de la définition du sous-typage.
- C$[e] = e\ e_1$: la dérivation suivante est valide (car $t_3 \sqsubseteq t_2 \sqsubseteq t_e \rightarrow t_1$) :

$$\frac{\mathcal{E}_1 \vdash_e e_1 : t_e \quad \mathcal{E}_1 \vdash_e e' : t_3}{\mathcal{E}_1 \vdash_e e'\ e_1 : t_1} \left(t_3 \sqsubseteq t_e \rightarrow t_1 \right)$$

- Si C$[e] = v\ e$, le raisonnement est identique au précédent cas avec cette fois : $t_3 \sqsubseteq t_2$ implique $t_2 \rightarrow t_1 \sqsubseteq t_3 \rightarrow t_1$. Donc, si la fonction v à pour type t_v, on a bien $t_v \sqsubseteq t_3 \rightarrow t_1$ puisque par hypothèse $t_v \sqsubseteq t_2 \rightarrow t_1$.
- Si C$[e] =$ `letrec` $x = e$ `in` e_1, alors, on peut alors écrire la dérivation suivante :

$$\frac{\mathcal{E}_1 + \{x : t_x\} \vdash_e e' : t_3 \quad \mathcal{E}_1 + \{x : t_x\} \vdash_e e_1 : t_1}{\mathcal{E}_1 \vdash_e \texttt{letrec}\ x = e\ \texttt{in}\ e_1 : t_1} \left(t_3 \sqsubseteq t_x \right)$$

puisque, t_3 est sous-type de t_2 lui-même sous-type de t_x (car C$[e]$ est bien typé), la contrainte est donc soluble et la dérivation valide.

Typage d'une substitution

Nous allons maintenant énoncer et démontrer que la substitution se comporte *bien* vis-à-vis du typage et du sous-typage.

Lemme 4.3 (Typage d'une substitution d'une variable) :
Dans une expression e, le remplacement d'une variable x par une sous-expression typable e′ sous-type de x produit une nouvelle expression typable sous-type de e. Soit, formellement :

$$\mathcal{E} + \{x : t_x\} \vdash_e e : t_1 \wedge \mathcal{E} \vdash_e e' : t' \wedge t' \sqsubseteq t_x \Longrightarrow \mathcal{E} \vdash_e [e'/x]e : t_2 \wedge t_2 \sqsubseteq t_1$$

Preuve : La preuve de ce lemme consiste en une induction sur la structure des expressions. Les cas initiaux de cette induction sont donc une variable, une adresse et une constante. La preuve du lemme dans les deux derniers cas est triviale puisqu'elles ne contiennent aucune variable. Dans le cas d'une variable y, on a :

$$\mathcal{E} \vdash_e [e'/x]y : t_2 = \begin{cases} \mathcal{E} \vdash_e e' : t_2 & \text{si } y = x, \text{ ce qui implique } t_2 = t' \sqsubseteq t_x = t_1 \\ \mathcal{E} \vdash_e y : t_2 & \text{sinon, ce qui implique } t_2 = t_1 \end{cases}$$

Il ne reste donc plus qu'à démontrer le lemme lorsque e est un appel de fonction prédéfinie, un message, une application, une abstraction, une interface ou une définition

récursive. Dans tous les cas, si une variable n'est pas redéfinie, sa substitution est propagée dans toutes les sous-expressions e_i de e. Donc, par application de l'hypothèse d'induction, $[e'/x]e_i$ est typable et son type t'_i est sous type du type t_i de e_i. De plus, l'hypothèse de typage de x peut être déchargée dans toutes les dérivation de typage des $[e'/x]e_i$.

- $e = f(e_1, e_2)$: trois cas se présentent selon la forme de f :
  - $f = send$:

$$\frac{\mathcal{E} \vdash_e [e'/x]e_1 : t'_1 \quad \mathcal{E} \vdash_e [e'/x]e_2 : t'_2}{\mathcal{E} \vdash_e [e'/x]send(e_1, e_2) : unit} \left(t'_2 \sqsubseteq @t'_1 \right)$$

    Or, $t'_2 \sqsubseteq t_2 \sqsubseteq @t_1 \sqsubseteq @t'_1$ et le type de retour est clairement décroissant puisqu'il ne change pas.
  - $f = init$: de même $t'_1 \sqsubseteq t_1 \sqsubseteq @\varphi$ et $t'_2 \sqsubseteq t_2 \sqsubseteq @\varphi \rightarrow \triangleright\varphi$.
  - $f = become$: de même $t'_1 \sqsubseteq t_1 \sqsubseteq @\varphi$ et $t'_2 \sqsubseteq t_2 \sqsubseteq \triangleright\varphi$.
- $e = m(e)$:

$$\frac{\mathcal{E} \vdash_e [e'/x]e : t'}{\mathcal{E} \vdash_e [e'/x]m(e) : \{m : \circ t'\}}$$

  Or, il est clair que $\{m : \circ t'\} \sqsubseteq \{m : \circ t\}$ puisque $t' \sqsubseteq t$.
- $e = f\ e$: la réduction suivante est valide :

$$\frac{\mathcal{E} \vdash_e [e'/x]f : t'_f \quad \mathcal{E} \vdash_e [e'/x]e : t'_e}{\mathcal{E} \vdash_e [e'/x](f\ e) : t} \left(t'_f \sqsubseteq t'_e \rightarrow t \right)$$

  puisque, $t'_f \sqsubseteq t_f \sqsubseteq t_e \rightarrow t \sqsubseteq t'_e \rightarrow t$ et la contrainte est donc soluble.
- $e = \lambda x'.e$:
  - si $x' = x$, e est inchangé et donc vérifie trivialement le lemme. Remarquons qu'il possible de décharger l'environnement car l'abstraction rajoute l'hypothèse sur le typage de x.
  - si $x' \neq x$, $[e'/x]\lambda x'.e = \lambda x'.[e'/x]e$ et donc :

$$\frac{\mathcal{E}+\{x' : t_a\} \vdash_e [e'/x]e : t'}{\mathcal{E} \vdash_e [e'/x]\lambda x'.e : t_a \rightarrow t'}$$

    Or, $t_a \rightarrow t' \sqsubseteq t_a \rightarrow t$ car $t' \sqsubseteq t$.
- $e = \chi[m_i(x_i).e_i]^{i \in I}$: soit $e'_i = \begin{cases} e_i & \text{si } x_i = x \\ [e'/x]e_i & \text{sinon} \end{cases}$, pour chaque indice i,par un raisonnement similaire au cas précédent, on montre que e'_i est typable et son type t'_i est sous-type de t_i le type de e_i et l'environnement peut donc être déchargé. La déduction suivante est donc valide :

$$\frac{\mathcal{E}+\{x_i : \alpha_i\} \vdash_e e'_i : t'_i}{\mathcal{E} \vdash_e [e'/x]\chi[m_i(x_i).e_i]^{i \in I} : \triangleright t} \left(\forall i \in I \ \ \{m_i : \bullet\alpha_i\} \sqsubseteq t \right)$$

  dont toutes les contraintes sont identiques à celle de l'hypothèse.
  Remarquons qu'aucune contrainte ne porte sur le type résultat des réactions

dans le cadre de $\mathcal{F}unc_0$, le système vérifie uniquement qu'elles sont correctement typées.

– $e = (\texttt{letrec}\ x' = e_1\ \texttt{in}\ e_2)$: suivant un raisonnement similaire aux deux cas précédents, $e'_i = \begin{cases} e_i & \text{si } x' = x \\ [e'/x]e_i & \text{sinon} \end{cases}$ (pour i valant 1 et 2) avec $t'_i \sqsubseteq t_i$. La déduction suivante est donc valide :

$$\frac{\mathcal{E}+\{x' : t_a\} \vdash_e e'_1 : t'_1 \quad \mathcal{E}+\{x' : t_a\} \vdash_e e'_2 : t'_2}{\mathcal{E} \vdash_e [e'/x](\texttt{letrec}\ x' = e_1\ \texttt{in}\ e_2) : t'_2} \left(t'_1 \sqsubseteq t_a\right)$$

or, $t'_1 \sqsubseteq t_1 \sqsubseteq t_a$ et $t'_2 \sqsubseteq t_2$.

Notons que ce lemme pourrait être limité à la substitution de valeur sémantique, car c'est la seule forme qui est indispensable à la preuve.

Typage des expressions

Nous pouvons maintenant énoncer et montrer la continuité[33] du typage fonctionnel.

Ce théorème va utiliser le typage d'une expression en cours de réduction, or, une telle expression risque de contenir des adresses. Nous verrons dans la sous-section suivante que le typage d'une configuration ajoutera aussi dans l'environnement les types des adresses induites par la configuration pour indiquer qu'elles existent. La règle de typage des variables qui vérifie leur présence dans l'environnement est donc étendue aux adresses.

Théorème 4.2 (Continuité du typage fonctionnel) :
Si une expression e de type t se réduit en une expression e' alors e' est bien typée et son type est inférieur à t. Soit, formellement :

$$\mathcal{E} \vdash_e e : t \wedge e \longrightarrow_e e' \Longrightarrow \mathcal{E} \vdash_e e' : t' \wedge t' \sqsubseteq t$$

Preuve : La preuve de ce théorème consiste en une induction sur la structure des expressions qui peuvent se réduire.

Le cas initial se réduit aux identificateurs (variables et adresses), qui s'ils sont bien typés, figurent dans l'environnement. Or, l'introduction dans l'environnement se fait uniquement pour des variables définies par abstraction, définition récursive ou filtrage d'interface, et toutes ces variables sont donc substituées et ainsi la règle VarE ne peut pas s'appliquer.

Avant de revenir à l'induction sur la structure de l'expression montrons que pour toutes les expressions, l'induction est valide si on applique la règle Cont et que la règle ContE ne peut s'appliquer :

– Supposons que C[e] ne soit pas une définition récursive, qu'elle soit typée $\mathcal{E} \vdash_e$ C[e] : t et qu'elle se réduise C[e] $\longrightarrow_e$ C[e']. Alors, par le lemme 4.1, on a $\mathcal{E} \vdash_e e : t_1$ et on peut donc appliquer l'hypothèse d'induction (car la règle Cont impose $e \longrightarrow_e e'$), ce qui conduit à : $\mathcal{E} \vdash_e e' : t_2$ et $t_2 \sqsubseteq t_1$. Le lemme 4.2 peut alors s'appliquer et montrer : $\mathcal{E} \vdash_e$ C[e'] : t' et $t' \sqsubseteq t$.

– Si C[e] est une définition récursive, le raisonnement est similaire avec un envi-

33. En anglais, on appelle généralement ce théorème *subject reduction*.

ronnement de typage étendu.
- Dans tous les cas, si C[e] est bien typée, alors le lemme 4.1 montre que e est bien typée et donc par application de l'hypothèse d'induction, elle ne peut se réduire en erreur. La règle ContE ne peut donc pas s'appliquer.

Revenons maintenant, à l'induction sur la structure de e :
- Si e est un appel de fonction prédéfinie, un message ou une interface, les seules réductions fonctionnelles qui pourraient s'appliquer sont les réductions de contexte que nous avons déjà analysées.
- Si e est une application :
  - Les règles de contextes ont déjà été analysées.
  - Comme l'application est bien typée, la contrainte impose un type fonctionnel à la première valeur et la règle AppE ne peut donc pas s'appliquer.
  - Si e vaut $\lambda x.e'\ v$, on a : $t_x \to t \sqsubseteq t_v \to t_r$ soit $t_v \sqsubseteq t_x$ et $t \sqsubseteq t_r$. L'application du lemme de substitution permet donc d'obtenir $\mathcal{E} \vdash_e [v/x]e' : t'$ avec $t' \sqsubseteq t \sqsubseteq t_r$. L'application de la règle App est donc correcte.
- Si e est une définition récursive :
  - Les règles de contextes ont déjà été analysées.
  - Si e vaut `letrec` $x = f$ `in` b (où b à le type t), on a $t_f \sqsubseteq t_x$, ce qui permet donc d'appliquer le lemme de substitution. Le résultat est alors : $\mathcal{E} \vdash_e [f/x]b : t'$ avec $t' \sqsubseteq t$ et la règle LRec s'applique donc correctement.

Les règles de typage des configurations

Avant de démontrer la continuité du typage des configuration, il faut présenter les règles de typage 4.6 qui assurent qu'une configuration ne contient pas d'erreur. Si $\mathcal{E}$ est l'environnement de typage de la configuration w, elles ont la forme $\mathcal{E} \vdash_w w$. Le typage parcourt la configuration et vérifie que toutes les expressions et toutes les valeurs qu'elles contiennent sont bien typées. On vérifie également que les acteurs reçoivent bien des messages compatibles avec leurs potentiels. Remarquons qu'un acteur doit avoir un type compatible avec son potentiel initial pour être bien typé.

Typage des configurations

Le théorème final de continuité du typage nécessite encore un lemme sur les configurations congrues.

Lemme 4.4 (Typage des configurations congrues) :
Si deux configurations w et w' sont congrues alors w est bien typée si et seulement si w' l'est. Soit, formellement :

$$w \equiv w' \Longrightarrow (\mathcal{E} \vdash_w w \Longleftrightarrow \mathcal{E} \vdash_w w')$$

Preuve : Nous allons prouver ce lemme en analysant chacune des équivalences possibles et en montrant comment, à partir de l'arbre de preuve du typage de w, on peut construire celui de w'.
- $\nu a.w \equiv \nu b.[b/a]w$ si $b \notin \mathcal{FN}(w)$: Ce résultat procède d'une induction sur la

$$\textsc{Empty}: \frac{}{\mathcal{E} \vdash_w \epsilon} \qquad \textsc{Par}: \frac{\mathcal{E} \vdash_w w_1 \quad \mathcal{E} \vdash_w w_2}{\mathcal{E} \vdash_w w_1 \parallel w_2} \qquad \textsc{Nu}: \frac{\mathcal{E}{+}\{a : t_a\} \vdash_w w}{\mathcal{E} \vdash_w \nu a.w}$$

$$\textsc{Mess}: \frac{\mathcal{E} \vdash_e v : t}{\mathcal{E} \vdash_w a \triangleleft m(v)} \Big(\mathcal{E}(a) \sqsubseteq @\{m : \circ t\} \Big) \qquad \textsc{Anon}: \frac{\mathcal{E} \vdash_e e : t}{\mathcal{E} \vdash_w \star \triangleright e}$$

$$\textsc{ActR}: \frac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e \mathcal{R} : t \quad \mathit{Abs}(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright \mathcal{R}} \begin{pmatrix} t \sqsubseteq \triangleright \phi \\ \mathcal{E}(a) \sqsubseteq @\phi \\ \mathcal{E}(a) \sqsubseteq @t_q \end{pmatrix}$$

$$\textsc{Act}: \frac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e e : t \quad \mathit{Abs}(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e} \Big(\mathcal{E}(a) \sqsubseteq @t_q \Big)$$

$$\frac{}{\mathcal{E} \vdash_q \varnothing : \{\}} \qquad \frac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e v : t_v}{\mathcal{E} \vdash_q m(v)::q : t} \begin{pmatrix} \{m : \circ t_v\} \sqsubseteq t \\ t_q \sqsubseteq t \end{pmatrix}$$

Règles 4.6 Le typage des configurations.

structure de w assez proche de la preuve du lemme de substitution des variables. Remarquons que a n'est pas connu de la configuration englobante (puisqu'il est restreint), que b (restreint également) n'est pas non plus connu et qu'enfin b ne figurant pas déjà dans w, il ne peut avoir que les mêmes contraintes que a.

– $\nu a.w \equiv w$ si $a \notin \mathcal{FN}(w)$: Le typage de $\nu a.w$ dans l'environnement $\mathcal{E}$ consiste en celui de w dans $\mathcal{E}{+}\{a : t_a\}$. Or, a n'est pas libre dans w, ce qui signifie que l'hypothèse sur le type de a n'est pas utilisée. Elle peut donc être supprimée (où ajoutée) à $\mathcal{E}$ sans changer la preuve, on a donc :

$$\mathcal{E} \vdash_w \nu a.w \iff \mathcal{E} \vdash_w w$$

– $\nu a_1.\nu a_2.w \equiv \nu a_2.\nu a_1.w$: Tout d'abord, notons que si a_1 et a_2 sont égales alors leurs deux configurations sont égales et ainsi il n'y a rien à prouver. On doit donc montrer la propriété uniquement dans le cas où a_1 et a_2 sont différentes. Dans ce cas, il est clair que $\{a_1 : t_1\}{+}\{a_2 : t_2\} = \{a_2 : t_2\}{+}\{a_1 : t_1\}$ et donc l'ordre d'abstraction ne change pas la preuve. En effet :

$$\mathcal{E} \vdash_w \nu a_1.\nu a_2.w \iff \mathcal{E}{+}\{a_1 : t_1\}{+}\{a_2 : t_2\} \vdash_w w \iff \mathcal{E} \vdash_w \nu a_2.\nu a_1.w$$

– $\nu a.w_1 \parallel w_2 \equiv \nu a.(w_1 \parallel w_2)$ si $a \notin \mathcal{FN}(w_2)$: Comme a ne figure pas dans w_2, on peut l'ajouter (ou le retirer) d'une preuve de typage de w_2 sans la modifier :

$$\mathcal{E} \vdash_w w_2 \iff \mathcal{E}{+}\{a : t\} \vdash_w w_2$$

Or,

$$\mathcal{E} \vdash_w \nu a.w_1 \parallel w_2 \Longleftrightarrow \mathcal{E}+\{a : t\} \vdash_w w_1 \wedge \mathcal{E} \vdash_w w_2$$
$$\mathcal{E} \vdash_w \nu a.(w_1 \parallel w_2) \Longleftrightarrow \mathcal{E}+\{a : t\} \vdash_w w_1 \wedge \mathcal{E}+\{a : t\} \vdash_w w_2$$

Par transitivité de l'équivalence, les deux preuves de typage sont donc équivalentes.

- $w \parallel \epsilon \equiv w$: Comme le typage de la configuration vide est un axiome et que le typage de deux sous-configurations en parallèle consiste à les typer séparément :

$$\mathcal{E} \vdash_w w \parallel \epsilon \Longleftrightarrow \mathcal{E} \vdash_w w$$

- $w_1 \parallel w_2 \equiv w_2 \parallel w_1$: Trivial puisque :

$$\mathcal{E} \vdash_w w_1 \parallel w_2 \Longleftrightarrow \mathcal{E} \vdash_w w_1 \wedge \mathcal{E} \vdash_w w_2$$

- $(w_1 \parallel w_2) \parallel w_3 \equiv w_1 \parallel (w_2 \parallel w_3)$: Également trivial puisque :

$$\mathcal{E} \vdash_w (w_1 \parallel w_2) \parallel w_3 \Longleftrightarrow \mathcal{E} \vdash_w w_1 \wedge \mathcal{E} \vdash_w w_2 \wedge \mathcal{E} \vdash_w w_3$$

- $\star \triangleright v \equiv \epsilon$: Les valeurs sémantiques sont typées dans tous les environnements et ainsi $\star \triangleright v$ est toujours bien typée. L'équivalence est donc triviale puisque les deux prédicats sont des tautologies.
- $\nu a.(\langle a \,|\, \varnothing \rangle_{\mathcal{P}} \triangleright \emptyset) \equiv \epsilon$: Une interface vide et une queue vide sont toujours bien typées par $@\{\}$, le raisonnement est donc identique à celui du cas précédent puisque les contraintes générées par le typage d'un acteur sont trivialement vérifiées. De plus, le nom a n'étant connu par aucune autre configuration, il n'existe aucune contrainte sur le type de a et son abstraction est donc vide (et donc incluse dans tous les $\mathcal{P}$).

Il est maintenant possible de montrer la continuité du typage sur les configurations.

Théorème 4.3 (Continuité du typage) :
Si une configuration w bien typée se réduit en une configuration w' alors w' est bien typée. Soit, formellement :

$$\mathcal{E} \vdash_w w \wedge w \longrightarrow w' \Longrightarrow \mathcal{E} \vdash_w w'$$

Preuve : La preuve de ce théorème consiste en une induction sur la structure des configurations. Les cas initiaux de cette preuve sont la configuration vide, le message en transit et l'acteur (anonyme ou non). Dans les deux premiers cas, il n'y a rien à prouver car la configuration vide et un message ne se réduisent pas. Soit un acteur d'identité α de corps e_1 ($w = \alpha \triangleright e_1$), de nombreuses réductions sont possibles selon les formes de α et de e_1. Plutôt que d'examiner tous les cas possibles pour α et e_1, nous allons examiner chaque réduction possible selon leurs formes.

- Exp : Par application du théorème de continuité du typage fonctionnel, on a :

$$\mathcal{E} \vdash_e e_1 : t_1 \wedge e_1 \longrightarrow_e e_2 \Longrightarrow \mathcal{E} \vdash_e e_2 : t_2 \wedge t_2 \sqsubseteq t_1$$

La réduction fonctionnelle n'affectant pas la queue de l'acteur ni son potentiel (si α vaut $\langle a \,|\, q \rangle_{\mathcal{P}}$), on a équivalence entre le typage de e_i et celui de $\alpha \triangleright e_i$ (pour

i valant 1 ou 2). Le typage de $\alpha \triangleright e_1$ implique donc celui de e_1 donc de e_2 et donc par transitivité de l'implication celui de $\alpha \triangleright e_2$. De plus, la réduction erronée ExpE ne peut pas avoir lieu puisque l'erreur n'est pas typée.

– Rea : Cette réduction n'est applicable que si $\alpha = \langle a \,|\, q \rangle$, $e_1 \in \mathbb{I}$ et un message de la boîte aux lettres peut être traité. On a alors :

$$\frac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e e_1 : t \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e_1} \begin{pmatrix} t \sqsubseteq \triangleright\phi \\ \mathcal{E}(a) \sqsubseteq @\phi \\ \mathcal{E}(a) \sqsubseteq @t_q \end{pmatrix}$$

Les types de tous les messages de la queue sont inclus dans le type de a car $\mathcal{E}(a) \sqsubseteq @t_q$. Soit aucun des messages n'est compréhensible par l'état actuel et aucune réduction du terme n'est possible. Soit, un message m est compris et alors le type du motif et la seule réduction pouvant s'appliquer est alors :

$$\frac{e_1 \prec q \Longrightarrow q',\ e,\ \sigma}{\langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e_1 \longrightarrow \langle a \,|\, q' \rangle_{\mathcal{P}} \triangleright \sigma(e)}$$

On montre aisément par induction sur la structure de la queue que $q' \subset q$ (au sens où tous les messages de q' sont dans q) et donc que q' est typable et que son type $t_{q'}$ vérifie $t_{q'} \sqsubseteq t_q$. Soit $m(v)$ le message accepté, $m(x)$ le motif qui l'accepte et e_2 le corps associé à ce motif. Supposons $\mathcal{E}(a) = @\phi_a$ alors $t_q \sqsubseteq \phi_a$ et $\phi \sqsubseteq \phi_a$. De plus, si t_v (resp. t_x) est le type de v (resp. de x), on a $\{m : \circ t_v\} \sqsubseteq t_q \sqsubseteq \phi_a$ et $\triangleright\{m : \bullet t_x\} \sqsubseteq t \sqsubseteq \triangleright\phi$ donc $\{m : \bullet t_x\} \sqsubseteq \phi \sqsubseteq \phi_a$. Ainsi, m figure dans ϕ_a sous forme installée avec le type α et $\bullet t_x \sqsubseteq \bullet\alpha$ et $\circ t_v \sqsubseteq \bullet\alpha$, soit $t_v \sqsubseteq \alpha \sqsubseteq t_x$.
Il est alors facile de démontrer à partir du lemme de substitution, que $\sigma(e)$ est typé. La conclusion est donc que le nouvel acteur est typable :

$$\frac{\mathcal{E} \vdash_q q' : t_{q'} \quad \mathcal{E} \vdash_e \sigma(e) : t'_e \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q' \rangle_{\mathcal{P}} \triangleright \sigma(e)} \left(\mathcal{E}(a) \sqsubseteq @t_{q'} \right)$$

– Send : $e = send(m(v), a)$ et deux possibilités se présentent pour α : $\alpha = \star$ ou $\alpha = \langle a \,|\, q \rangle$. Alors :

$$\frac{\mathcal{E} \vdash_e \mathrm{C}[e] : t}{\mathcal{E} \vdash_w \star \triangleright \mathrm{C}[e]} \quad \text{ou} \quad \frac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e \mathrm{C}[e] : t \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright \mathrm{C}[e]} \left(\mathcal{E}(a) \sqsubseteq @t_q \right)$$

et $\alpha \triangleright \mathrm{C}[send(m(v), a)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel a \triangleleft m(v)$, il nous faut donc montrer que le corps de l'acteur et le message sont typables après la réduction. Or, l'application de *send* est typée par :

$$\textsc{Send:}\ \frac{\mathcal{E} \vdash_e m(v) : \{m : \circ t_v\} \quad \dfrac{a \in dom(\mathcal{E})}{\mathcal{E} \vdash_e a : \mathcal{E}(a)}}{\mathcal{E} \vdash_e send(m(v), a) : unit} \begin{pmatrix} \{m : \circ t_v\} \sqsubseteq \psi \\ t_a \sqsubseteq @\psi \end{pmatrix}$$

Donc, $\mathcal{E}(a) \sqsubseteq @\{m : \circ t_v\}$ et le message est typable. Enfin, par application du lemme de typage d'un contexte avec *nop* (de type *unit*), le corps de l'acteur est

typable. La réduction SendE ne peut avoir lieu puisque comme l'envoi est bien typé, son premier argument est un message et son deuxième un acteur.

– New : $e = new$ et selon que $\alpha = \star$ ou que $\alpha = \langle a \,|\, q \rangle$, les deux règles de typage des acteurs rappelées dans le cas Send sont vérifiées. De plus, $\alpha \triangleright \mathrm{C}[new] \longrightarrow \nu a'.(\alpha \triangleright \mathrm{C}[a'])$ où a' ne figure pas dans $\mathrm{C}[new]$. Or, *new* a pour type $@\phi$ (où ϕ est une nouvelle variable de type). Il nous faut montrer que le corps de l'acteur est encore typable dans un environnement étendu par $\{a' : @\phi\}$. Ce qui est immédiat par application du lemme de typage d'un contexte.

– Init : $e = init(a, v)$ et $\alpha = \star$ ou $\alpha = \langle a \mid q \rangle$, les deux règles de typage des acteurs rappelées dans le cas Send sont vérifiées. Comme $\alpha \triangleright \mathrm{C}[init(a, v)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel \langle a \,|\, \varnothing \rangle_{\mathcal{P}(v\ a)} \triangleright v\ a$, il nous faut donc montrer que le corps de l'acteur et le nouvel acteur sont typables. Or, le typage de l'application de *init* conduit à :

$$\textsc{Init:}\ \frac{\textsc{Id:}\ \dfrac{a \in dom(\mathcal{E})}{\mathcal{E} \vdash_e a : \mathcal{E}(a)} \quad \mathcal{E} \vdash_e v : t}{\mathcal{E} \vdash_e init(a, v) : unit} \begin{pmatrix} t_a \sqsubseteq @\psi \\ t \sqsubseteq @\psi \to \triangleright\psi \end{pmatrix}$$

On a donc $t = @\alpha \to \triangleright\beta$ avec $t_a \sqsubseteq @\psi \sqsubseteq @\alpha$ et $\beta \sqsubseteq \psi$. Donc, $t_a = @\psi_a$, $\alpha \sqsubseteq \psi_a$ et $\beta \sqsubseteq \psi_a$. La dérivation suivante est donc valide :

$$\frac{\mathcal{E} \vdash_q \varnothing : @\{\} \qquad \dfrac{\mathcal{E} \vdash_e a : @\psi_a \quad \mathcal{E} \vdash_e v : @\alpha \to \triangleright\beta}{\mathcal{E} \vdash_e v\ a : t'} \begin{pmatrix} \alpha \sqsubseteq \psi_a \\ \triangleright\beta \sqsubseteq t' \end{pmatrix} \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}(v\ a)}{\mathcal{E} \vdash_w \langle a \,|\, \varnothing \rangle_{\mathcal{P}(v\ a)} \triangleright v\ a} \,(1)$$

avec $(1) = \{t' \sqsubseteq \triangleright\phi;\ \mathcal{E}(a) \sqsubseteq @\phi;\ \mathcal{E}(a) \sqsubseteq @\{\}\}$. La dernière contrainte est triviale et les deux premières deviennent $\{\beta \sqsubseteq \phi;\ @\psi_a \sqsubseteq @\phi\}$ et donc $\{\beta \sqsubseteq \psi_a\}$ qui est vérifiée par hypothèse.

Reste à montrer que l'abstraction de $@\psi_a$ est bien inférieure au potentiel. Or, les seules contraintes qui peuvent ajouter un message dans l'abstraction de ψ_a sont $\alpha \sqsubseteq \psi_a$ et $\beta \sqsubseteq \psi_a$ (où α est le type de l'*ego* et β le type du *self*).

Les autres contraintes qui pourront porter sur a ne peuvent pas contenir de messages installés. Or, si m est une étiquette figurant dans l'abstraction du type de a, par définition de l'abstraction, il est immédiat qu'il existe un type t tel que $\{m : \bullet t\} \sqsubseteq \phi_a$. Cette contrainte provient donc de l'interface initiale ou de l'*ego*.

Vu son type, v est de la forme $\lambda ego.e$ où e est de type $\triangleright\beta$. Donc, $\mathcal{P}(v\ a) = \lambda ego.e\ a = \mathcal{P}([\{\}/ego]e)$ car $\mathcal{P}(a) = \{\}$, $\{\}$ et $\mathcal{P}(\lambda ego.e) = \lambda ego.e, \{\}$. De plus, la seule règle d'inférence qui ajoute une contrainte contenant des messages installés est la définition d'interface (Inter), m figure donc dans une interface de la forme $\chi[m(x).e,\ \ldots]$ où x est de type t.

– Si $\{m : \bullet t\} \sqsubseteq \beta$:
  – Soit la contrainte a été inférée directement et comme la seule règle de typage introduisant des types interfaces est celle typant les interface, e est une interface de la forme $\chi[m(x).e',\ \ldots]$. Il est alors immédiat par la définition du potentiel que m figure dans le potentiel de e et donc dans $\mathcal{P}(v\ a)$.

- Soit cette contrainte est due à la transitivité et il existe e_1 de type $\triangleright\psi_1$, ..., e_n de type $\triangleright\psi_n$ telles que $\{m : \bullet t\} \sqsubseteq \psi_1 \sqsubseteq \cdots \sqsubseteq \psi_n \sqsubseteq \beta$. Or, les quatre seules façons de produire une contrainte $\triangleright\psi_1 \sqsubseteq \triangleright\psi_2$ sont décrites ci-dessous avec e_1 de type $\triangleright\psi_1$ et e de type t_e.
  - $e_2 = \lambda x.e_1\ e$ qui génère $t_x \to \triangleright\psi_1 \sqsubseteq t_e \to \triangleright\psi_2$. La simplification de cette contrainte produisant la contrainte $\triangleright\psi_1 \sqsubseteq \triangleright\psi_2$.
  - $\lambda x.e\ e_1$ qui génère $t_x \to t_e \sqsubseteq \triangleright\psi_1 \to t_r$. La réduction de cette contrainte produisant la contrainte $\triangleright\psi_1 \sqsubseteq t_x$. La valeur x prend le rôle de e_2 avec $t_x = \triangleright\psi_2$.
  - `letrec` $x = e_1$ `in` e qui produit la contrainte $\triangleright\psi_1 \sqsubseteq \triangleright\psi_2$ où $\triangleright\psi_2$ est le type de x dans e. La valeur x prend le rôle de e_2.
  - $\chi[m'(x).e,\ \ldots] \prec m'(e_1)$ qui correspond à la réception d'un message m' contenant l'interface. L'installation de cette interface sur l'acteur a provoqué (via un *init* ou un *become*) la génération d'une contrainte $\{m' : \bullet t_x\} \sqsubseteq \psi_{ego}$ et l'envoi du message a ajouté la contrainte $\{m' : \circ t_1\} \sqsubseteq \psi_{ego}$. Ces deux contraintes amènent à $t_1 \sqsubseteq t_x$. La valeur x prend le rôle de e_2 avec $t_x = \triangleright\psi_2$.

  Dans le premier cas, la valeur résultante sera évaluée en une interface. Le calcul du potentiel de $\lambda x.e_1\ e$ est : $\mathcal{P}(\lambda x.e_1\ e) = V,\ E \cup E_e$ avec $\mathcal{P}(e) = V_e,\ E_e$ et $\mathcal{P}([V_e/x]e_1) = V,\ E$. Donc, si m figure dans V ou dans E, il figure dans V ou dans $E \cup E_e$. Ainsi, si m figure dans le potentiel de e_1 il figure dans le potentiel de e_2.

  Dans les deux cas suivant, lors de l'évaluation de l'expression, e contient une variable reliée à l'interface $\chi[m(x).e,\ \ldots]$. Si m est dans le potentiel valeur de e_1, comme il remplacera toutes les occurrences de x, m est dans le potentiel valeur de e_2. De plus, s'il figure dans l'effet de e_1, il est également dans l'effet de e_2.

  Dans le dernier cas, assumant un comportement venant de l'extérieur l'acteur a un potentiel ouvert. Donc tout m y figure.

  Le raisonnement s'applique de proche en proche (récurrence simple) et nous permet donc de montrer que m appartient au potentiel de e et donc à celui de l'acteur a.
- Si $\{m : \bullet t\} \sqsubseteq \alpha$: le raisonnement est similaire excepté le fait que cette contrainte provient d'un changement de comportement. Il existe donc dans e une expression $become(ego, v')$ et la contrainte provient de $\{m : \bullet t\} \sqsubseteq \phi_{v'} \sqsubseteq \alpha$. Comme dans le cas précédent, on montre que m est dans le potentiel de v' et donc dans celui de a.

On a donc bien démontré que $Abs(\mathcal{E}(a)) \subseteq \mathcal{P}(v\ a)$.

De plus, la règle de typage de la fonction de création d'un acteur *init* impose aux valeurs qui lui sont passées en argument d'être un acteur et une fonction prenant en paramètre une adresse et renvoyant une interface. La règle de réduction InitE ne peut donc pas s'appliquer.

- Bec : Les règles de typage introduite dans le Send sont valides avec $\alpha = \star$ ou $\alpha = \langle a \mid q \rangle$ et $e = become(a, v)$. La réduction qui a lieu est $\alpha \triangleright \mathrm{C}[become(a, \mathcal{R})] \longrightarrow \alpha \triangleright \mathcal{R} \parallel \star \triangleright \mathrm{C}[nop]$, il faut donc montrer que les deux

acteurs sont typables.
Le premier est trivialement typé si $\alpha = \star$ puisque l'application du *become* suppose que le comportement est bien typé.
Si $\alpha = \langle a \,|\, q \rangle$, on ne touche ni à la queue ni au potentiel et donc les prémisses de la règle de typage sont immédiates. La contrainte sur la queue était vérifiée avant réduction et est donc encore vraie. Il reste à démontrer que les deux autres contraintes $t \sqsubseteq \triangleright\phi$ et $\mathcal{E}(a) \sqsubseteq @\phi$ sont solubles. Or, le typage du *become* a généré les contraintes $\mathcal{E}(a) \sqsubseteq @\psi$ et $t \sqsubseteq \triangleright\psi$ qui sont solubles par hypothèse. Donc $\phi = \psi$ est solution des deux contraintes précédentes.
La preuve du typage de l'acteur anonyme est également immédiate par application du lemme de contexte puisque *nop* est du même type que l'application du *become*.
Le typage de la fonction *become* fait que la règle de réduction BecE ne peut s'appliquer parce que les arguments du changement de comportement sont bien une adresse et une interface.

Il ne reste plus qu'à analyser les deux dernières formes possibles pour une configuration : une restriction ou une mise en parallèle.

- Si la configuration est une restriction $\nu a.w_1$, deux réductions sont possibles :
  - Res : Or, la restriction étant bien typée, w_1 est bien typée et se réduit, par hypothèse d'induction, en un terme bien typé w_2 dans le même environnement. Ce qui montre directement que $\nu a.w_2$ est bien typée.
  - Cong : Dans ce cas, la seule congruence possible impose $w_1 = \langle a \,|\, \varnothing \rangle_{\mathcal{P}} \triangleright \emptyset$ qui ne peut alors pas se réduire. La règle n'est donc pas applicable et il n'y a rien à prouver.
- Si la configuration est une mise en parallèle, deux réductions sont possibles, la réduction d'une sous-configuration (Par) ou bien la réception d'un message (Rcv). Le premier cas est immédiat par application de la règle de typage du parallèle et de l'hypothèse d'induction. La réception d'un message par son destinataire n'est possible que si la configuration est de la forme $\langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m(v)$, elle se réduit alors en $\langle a \,|\, q :: m(v) \rangle_{\mathcal{P}} \triangleright e$ avec $m \in \mathcal{P}$. La première configuration est typée (par hypothèse) par la déduction suivante :

$$\frac{\dfrac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e e : t \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e}\,(1) \qquad \dfrac{\mathcal{E} \vdash_e v : t}{\mathcal{E} \vdash_w a \triangleleft m(v)}\,(2)}{\mathcal{E} \vdash_w \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m(v)}$$

avec les contraintes : $\mathcal{E}(a) \sqsubseteq @t_q$ pour 1 et $\mathcal{E}(a) \sqsubseteq @\{m : \circ t\}$ pour 2, dans le cas où le corps de l'acteur n'est pas une interface. Il faut alors montrer que l'acteur avec la queue augmentée est bien typé. Le corps et le potentiel étant inchangé, les deux derniers prémisses sont immédiates. Le message étant bien typé, il est trivial de montrer que la queue étendue q' reste bien typée. Il reste

donc à montrer la validité de la contrainte $\mathcal{E}(a) \sqsubseteq @t_{q'}$. Or :

$$\frac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e v : t_v}{\mathcal{E} \vdash_q m(v)::q : t} \begin{pmatrix} \{m : \circ t_v\} \sqsubseteq t \\ t_q \sqsubseteq t \end{pmatrix}$$

ce qui implique donc immédiatement la contrainte puisque $@t_q \sqcap @\{m : \circ t\}$ qui contient $\mathcal{E}(a)$ vaut $@(t_q \sqcup \{m : \circ t\})$. Si le corps de l'acteur est une interface, alors la déduction du typage de la configuration est similaire à la précédente mais avec les contraintes suivantes :

$$\begin{cases} t \sqsubseteq \triangleright\phi \\ \mathcal{E}(a) \sqsubseteq @\phi \\ \mathcal{E}(a) \sqsubseteq @t_q \\ \mathcal{E}(a) \sqsubseteq @\{m : \circ t\} \end{cases}$$

La seule chose qui change dans la dérivation prouvant le typage de l'acteur après réduction est la boîte aux lettres. Le raisonnement pour montrer que la nouvelle contrainte sur la queue est valide est alors identique au cas où le corps n'est pas une interface.
Enfin, comme $\mathcal{E}(a) \sqsubseteq @\{m : \circ t\}$ et qu'en fin de la résolution des contraintes inférées sur le programme, on vérifie que tous les types d'acteurs créés dans le programmes ne contiennent pas de messages reçus non-installés, un message correspondant à m est installé dans le programme. Or, cette installation est prise en compte initialement (donc avant toute utilisation de a) et alors l'étiquette correspondant à m figure dans le potentiel. La règle RcvE ne peut donc pas s'appliquer.

Ceci conclut la démonstration du théorème par induction.

Correction du typage

Finalement, le théorème énonçant la correction de notre système de type est une conséquence directe de la continuité du typage des configurations.

Théorème 4.4 (Correction du typage) :
Une configuration w bien typée ne peut se réduire en une erreur. Soit, formellement :

$$\mathcal{E} \vdash_w w \Longrightarrow w \not\longrightarrow \mathbf{Err}$$

Preuve : L'erreur n'étant pas typable, si on suppose que w se réduit en **Err**, une application du théorème de continuité amène directement à une contradiction.

4.9 Conclusion

Dans ce chapitre, nous avons décrit le principe de nos systèmes de type dans le cadre des configurations utilisant un noyau fonctionnel restreint à un λ-calcul avec constante dont nous avons explicité la sémantique. Cette approche permet de simplifier grandement l'exposition à venir des systèmes de type de ML-Act et d'Erlang.

Deuxième partie

ML-Act

Chapitre 5

Le langage

Ce chapitre décrit l'essentiel du langage ML-Act. Pour cela, nous évoquons les concepts fonctionnels de la famille de langage ML, puis l'approche suivie pour combiner ces aspects fonctionnels avec le modèle d'acteur au sein de ML-Act. Ensuite, nous donnons la syntaxe du langage, et nous l'illustrons par un exemple qui nous sert de fil conducteur pour la suite de la présentation. Cette syntaxe est ensuite réduite à un sous-ensemble contenant les concepts essentiels de ML-Act, débarrassé d'une grande partie du sucre syntaxique. Le but de cette simplification est de diminuer le nombre et la complexité des règles de sémantique et de typage. Nous suivons alors le procédé d'instanciation décrit dans le chapitre précédent en utilisant un λ-calcul un peu plus étendu. Nous définissons ainsi formellement les spécificités de la sémantique de ML-Act. Enfin, nous concluons en évoquant les réalisations ainsi que les choix faits lors de la conception du langage.

5.1 Présentation rapide de ML-Act

ML-Act n'est pas un nouveau « *merveilleux* » langage, mais simplement un langage expérimental destiné à valider pratiquement des systèmes de type prouvés théoriquement. Ainsi, une fois le compilateur réalisé, il sera possible de tester et d'évaluer l'apport concret pour le programmeur des systèmes d'analyse de code. Comme nous l'avons déjà évoqué, l'équipe Plasma de l'IRIT avait conçu et réalisé une implantation d'un langage d'acteur : *Plasma II*. Cependant, celui-ci, basé sur LISP est fondamentalement un langage non-typé. Or, les analyses que nous avons mises au point reposent sur des mécanismes de typage. Nous avons donc choisi de proposer un langage dérivé de ML (qui est l'objet de nombreuses analyses statiques). Plus généralement, aucun des langages d'acteurs fonctionnels existants ne repose sur un noyau fonctionnel typé, ils sont généralement inspirés de LISP. D'autres langages concurrents fonctionnels existent, par exemple CML ([Rep91]) ou PICT ([PT97]), mais ils n'intègrent pas d'acteurs ou d'objets concurrents. Il a donc été nécessaire de construire un langage d'acteurs dérivé de la famille ML pour valider nos techniques.

ML-Act est un langage de programmation fonctionnel concurrent. Sa partie fonctionnelle est héritée d'Objective CAML dont il est une extension conservative du point de vue de la syntaxe et de la sémantique. Nous avons uniquement utilisé le cœur fonctionnel d'Objective CAML, ML-Act ne contient donc pas les notions de modules et d'objets,

ni pour l'instant, les traits impératifs, les types concrets et les exceptions. La syntaxe ainsi que la sémantique concurrente sont elles issues du modèle d'acteurs et de la notion de configurations tels qu'ils ont été présentés dans le premier chapitre.

OBJECTIVE CAML fait partie d'une famille de langages de programmation fonctionnels directement issus du λ-calcul. Historiquement, ML est le métalangage développé pour le système d'assistance à la construction de preuves LCF dû à R. MILNER et al. (voir par exemple [GMM+78] et [Plo77]). Même s'il s'agit d'un lointain descendant de LISP, son développement a su tirer les leçons apportées par celui-ci. Plusieurs dialectes en sont issus, ils se différencient aussi bien par la syntaxe que par la sémantique. La version qui nous a servi de modèle lors de la conception de ML-ACT est OBJECTIVE CAML. Ce langage est développé dans le cadre du projet CRISTAL à l'INRIA où il s'appuie sur une longue expérience de conception de langages de la famille ML. Il a été conçu dans le but de fournir un langage puissant et expressif pour la programmation d'algorithmes symboliques ou numériques. Il comporte donc les caractéristiques les plus intéressantes de nombreux autres langages, ce qui le rend généraliste. La meilleure preuve en est la variété des applications qui ont été développées en OBJECTIVE CAML. On peut, par exemple, citer : la programmation système (ensemble), la programmation Internet (MMM), la programmation graphique (GwML), la programmation numérique (IPLIB) ou la programmation symbolique (le système de preuve COQ). Il contient donc de nombreux concepts :

- c'est un langage *fonctionnel* ;
- contenant des traits impératifs ;
- dont la mémoire est gérée *automatiquement* par un ramasse-miettes ;
- il contient un système puissant de gestion des *exceptions* ;
- il intègre les notions d'*objet*, de *classe* et d'*héritage* ;
- il comporte un système sophistiqué de *modules* paramétrés ;
- il est muni d'un système de type *statique* autorisant le typage *polymorphe* paramétrique ;
- les types sont calculés automatiquement par *inférence* ;
- il permet d'exécuter des processus légers et de communiquer sur Internet ;
- il dispose enfin de nombreuses bibliothèques ainsi que de nombreux outils de programmation.

C'est un langage qui permet le développement d'applications complexes, éventuellement concurrentes ou distribuées, avec d'excellentes garanties quant à la sûreté de leur exécution. Enfin, il est performant tout en étant portable, ce qui explique son succès au sein de la recherche et de l'enseignement. Pour une présentation plus complète d'OBJECTIVE CAML, on peut consulter le livre de E. CHAILLOUX, P. MAMOURY et B. PAGANO [CMP00]. Un portail Internet [34] est consacré au langage et contient de nombreuses références sur celui-ci ainsi que les nombreuses applications pour lesquelles il a été utilisé (dont celles suscitées).

Un programme OBJECTIVE CAML peut être vu comme une suite de définitions et d'expressions. Les définitions introduisent des noms et les lient à des valeurs. Schématiquement, l'exécution d'un tel programme consiste à construire un environnement, puis à évaluer les expressions dans celui-ci. Cette description reste valable pour ML-ACT, au-

34. `http://caml.inria.fr`

quel deux déclarations spécifiques sont ajoutées : celle des étiquettes de message et celle des comportements. Les expressions ont également été étendues pour permettre l'envoi de message, la création d'acteur, le suicide et le changement du comportement d'un acteur. Enfin, un acteur peut référencer son adresse (*ego*) ainsi que son comportement courant (*self*). La sémantique de ces différentes extensions suit majoritairement la tradition de la communauté acteur. Toutefois, dans le modèle usuel, un acteur dont la réaction ne se termine pas par un changement de comportement meurt. Le suicide est donc implicite et si un acteur veut reprendre son comportement courant, il doit effectuer un `become self`. Or, nous pensons que d'une manière générale les objets réactifs ont une durée de vie *infinie* (du moins grande) et que leur suicide est un cas exceptionnel. Nous avons donc choisi un cycle de vie différent pour un acteur : en fin de réaction, celui-ci reprend automatiquement son comportement courant. Le programmeur doit donc spécifier clairement (via une instruction `suicide`) la fin de vie d'un acteur ou bien son nouveau comportement (via un `become`). En d'autres termes, si la réaction ne contient pas de changement explicite de comportement, l'acteur exécute implicitement un `become self`.

5.2 La syntaxe de ML-ACT

Dans cette section, nous définissons et expliquons la syntaxe du langage ML-ACT. Cette syntaxe est une extension du noyau fonctionnel de OBJECTIVE CAML par les primitives du modèle d'acteurs (envoi de message, changement de comportement et création de nouveaux acteurs) et par une définition globale de comportement.

Un exemple de programme

Avant de définir formellement la syntaxe de ML-ACT, étudions le petit exemple 5.1. Il commence par définir trois comportements par les instructions `behavior` :

- **empty**, un acteur assumant ce comportement attend un message étiqueté **start** et contenant une valeur `v`. Lorsqu'il reçoit un tel message, il adopte le comportement **cell** en lui passant la valeur `v` qu'il a reçue en paramètre.
- **cell**, conserve une valeur qui peut être modifiée par un message `set` et demandée par un message `get`. L'argument `c` du message `get` fournit l'adresse (la continuation) à laquelle doit être envoyée la valeur stockée (dans un message `prn`).
- **screen**, qui décrit le comportement d'un processus d'affichage. A chaque fois qu'il reçoit un message `prn` contenant une valeur `v`, il affiche cette valeur à l'écran.

La suite du programme est un `let` qui provoque la création de deux acteurs : d'une part, `a_scr`, l'écran, qui assume le comportement `screen` et d'autre part, `a_cell`, la cellule, qui assume initialement le comportement `empty`. Et finalement, le corps du `let` décrit l'envoi de quatre messages à la cellule.

La sémantique de ML-ACT n'a pas encore été présentée formellement, mais nous pouvons déjà décrire intuitivement une des exécutions possibles de ce programme. En effet, l'indéterminisme lié à la concurrence ne permet pas de préjuger de l'ordre dans lequel les messages vont être reçus par l'acteur `a_cell`. Cependant, toutes les exécutions possibles commencent par la création des deux *acteurs* et l'envoi des quatre messages. Pour

```
label start, get, set, prn;;

behavior empty =
  message start v = become (cell v)
and cell v =
  message get c  = send prn(v) to c
  message set v' = become (cell v')
end;;

behavior screen =
  message prn v = print_int v
end;;

let a_cell = new empty
and a_scr = new screen in
  send start(1) to a_cell;
  send get(a_scr) to a_cell;
  send set(2) to a_cell;
  send get(a_scr) to a_cell;;
```

Exemple 5.1 Une cellule linéaire.

mieux comprendre le déroulement de l'exécution, l'ensemble des comportements que va prendre un acteur donné est approximé par un automate. Les nœuds de cet automate correspondent aux comportements et les transitions sont étiquetées par le message accepté pour atteindre le nouveau comportement. L'état initial correspond au comportement initial de l'acteur. Il convient de noter que la présence de conditionnelles dans le langage rend, dans le cas général, cet automate non déterministe. L'automate de la figure 5.1 décrit ainsi tous les comportements possibles de l'acteur `a_cell`. Donc, selon l'entrelacement des messages `set` et `get`, à l'écran s'affiche soit 1 puis 1, soit 1 puis 2, soit 2 puis 2.

Remarque :
L'instruction `label` *n'est pas indispensable si le programme est clos. En effet, il suffit alors de parcourir tous les comportements pour collecter les différentes étiquettes (et ainsi construire automatiquement* $\mathbb{M}$*). Cependant, dans un cadre plus général, où le compilateur n'a pas toujours accès à l'ensemble du code, il est nécessaire de particulariser les variables qui sont des étiquettes. Une autre approche possible du problème, aurait été de différencier lexicalement les étiquettes des variables.*

La syntaxe

La syntaxe externe de ML-Act est une extension d'Objective CAML, sa grammaire complète figure dans l'annexe A, à la page 243. Présentons rapidement ses différentes composantes.

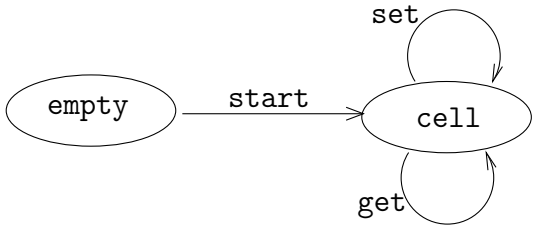


Figure 5.1 Les comportements de l'acteur `a_cell`

Les instructions : <*inst*>

Un programme sera composé d'une liste d'instructions.

<*expr*> :
Une expression peut être globale, elle est alors évaluée uniquement pour ses effets de bord.

'`label`' IDENT [',' IDENT]* :
Des identificateurs peuvent être déclarés *étiquettes de messages*, ils ne peuvent plus alors être utilisés comme des variables usuelles. Notons, que si le programme est clos, cette déclaration devient inutile, puisqu'il est trivial de calculer l'ensemble des étiquettes du programme.

'`let`' [<*motif*>]$^+$'=' <*expr*> ['`and`' [<*motif*>]$^+$'=' <*expr*>]* :
Il est possible de définir des variables en leur associant une valeur. En ML, la valeur associée à une variable ne peut être modifiée dans la suite du programme.

'`let`' '`rec`' IDENT [<*motif*>]*'=' <*expr*> ['`and`' IDENT [<*motif*>]*'=' <*expr*>]* :
L'instruction `let rec` permet de définir un ensemble de fonctions mutuellement récursives. Notons, qu'à la différence de OBJECTIVE CAML, nous imposons que la définition récursive se fasse sur des noms[35] et nous autorisons une définition sans argument (la phase de typage vérifiera qu'alors l'expression est une fonction). Pour simplifier le langage, nous limitons la forme des définitions récursives, soit elles définissent des fonctions, soit elles génèrent un groupe d'acteurs se connaissant mutuellement. Il est possible techniquement d'autoriser le mélange mais cela complique sensiblement la traduction de ML-ACT en $\mathcal{F}unc_1$. De plus, nous pensons qu'il est judicieux pour un programmeur de ne pas mélanger des définitions de fonctions et des créations d'acteurs. Nous avons cependant préféré garder la même syntaxe plutôt que de rajouter un éventuel `let actor` pour définir des groupes d'acteurs tel qu'on le trouve par exemple dans les travaux de G. AGHA et al. (voir par exemple [AH92] ou [Agh86]). Et ce, afin de respecter au maximum la même logique de la syntaxe des objets et des classes de OBJECTIVE CAML.

'`behavior`' IDENT [<*motif*>]*'=' [<*rea*>]$^+$['`and`' IDENT [<*motif*>]*'=' [<*rea*>]$^+$]*'`end`' :
La définition de comportement est récursive et suit donc les mêmes restrictions que le `let rec`. Notons qu'il est possible de définir un comportement avec des arguments, formellement, il s'agit d'une fonction renvoyant un comportement. Le corps

35. Une restriction similaire existe en OBJECTIVE CAML, mais la vérification est faite durant la phase sémantique de la compilation

du comportement a une forme spécifique qui décrit la liste des messages qu'il accepte et la réaction associée à chaque message. Le choix d'introduire dans le programme les comportements uniquement par des déclarations globales est inspiré de la déclaration des classes en OBJECTIVE CAML. De plus, cela rend les programmes plus clairs, puisque la description du fonctionnement des acteurs et leur initialisation sont ainsi clairement séparées. Par contre, techniquement, il est tout à fait possible de faire en sorte que la définition de comportement suive celle des `let` classiques (et puisse donc apparaître n'importe où dans le programme).

Les interfaces : *<rea>*

'`message`' IDENT [*<motif>*] '`=`' *<expr>* :
: La réaction à un message reçu est simplement une expression dans laquelle les arguments du message sont liés par le filtrage. Si le motif est absent, il est implicitement égal à `()`.

Les motifs : *<motif>*

IDENT :
: Un motif peut être une variable. En ML-ACT, toute occurrence de variable dans un motif est liante quitte à masquer une définition précédente de la variable.

<const> :
: En ML-ACT, les constantes possibles sont : les entiers (INT), les réels (FLOAT), les caractères (CHAR), les booléens (BOOL) et les chaînes de caractères (STRING).

'`_`' :
: Le joker permet de capturer tous les cas possibles dans un filtrage sans définir de nouvelles variables.

'`(`' [*<motif>* ['`,`' *<motif>*]⋆] '`)`' :
: Il est possible de décomposer un tuple. Un tuple vide correspond à la constante `()` – `unit` – et un tuple à un élément correspond à un simple parenthèsage.

'`[`' [*<motif>* ['`;`' *<motif>*]⋆] '`]`'

<motif> '`::`' *<motif>* :
: Une liste peut se décomposer, soit en la suite de ses éléments dans l'ordre, soit en tête et queue.

Les expressions : *<expr>*

IDENT :
: Un identificateur est introduit par un `let`, un `let rec`, un filtrage, une définition de comportement ou bien est déclaré étiquette de message.

<const>

'`(`' [*<expr>* ['`,`' *<expr>*]⋆] '`)`'

'`[`' [*<expr>* ['`;`' *<expr>*]⋆] '`]`' :
: Les constantes, les tuples et les listes sont similaires au cas des motifs.

<expr> <expr>

<expr> <bin> <expr>

<un> <expr> :
Il y a trois formes d'applications en ML-ACT : l'application classique d'une fonction sur un argument, l'application d'un opérateur binaire prédéfini (de manière infixe) et l'application d'un opérateur unaire prédéfini similaire au cas général. La liste des opérateurs prédéfinis figure dans la grammaire fournie en annexe page 243.

'`let`' [*<motif>*]$^+$'`=`' *<expr>* ['`and`' [*<motif>*]$^+$'`=`' *<expr>*]*'`in`' *<expr>*

'`let`' '`rec`' IDENT [*<motif>*]*'`=`' *<expr>* ['`and`' IDENT [*<motif>*]*'`=`' *<expr>*]*'`in`' *<expr>* :
Les définitions locales qu'elles soient récursives ou non, ont la même forme que les définitions globales. On ajoute simplement l'expression dans laquelle il sera possible d'utiliser les variables ainsi définies.

'`function`' ['`|`' *<motif>* '`->`' *<expr>*]$^+$:
Les fonctions se définissent par cas, chaque cas est alors un couple motif et expression. Lorsque la fonction est appliquée, l'argument reçu est comparé avec chaque motif dans l'ordre de définition de la fonction. Le premier filtrage de l'argument qui réussit, fournit l'expression poursuivant le calcul dans l'environnement résultant de la décomposition de l'argument par le motif.

'`match`' *<expr>* '`with`' ['`|`' *<motif>* '`->`' *<expr>*]$^+$:
L'application d'une fonction définie comme précédemment fournit un opérateur de choix décomposant le résultat de l'évaluation d'une expression donnée.

'`if`' *<expr>* '`then`' *<expr>* ['`else`' *<expr>*] :
La conditionnelle usuelle des langages de programmation. Notons que la partie « sinon » est facultative. Si elle est absente, elle est considérée valant `()`, ce qui impose donc à la partie « alors » d'avoir un type `unit`.

'`ego`'

'`self`' :
En ML-ACT, deux variables `ego` et `self` sont supposées prédéfinies. Elles contiennent respectivement l'adresse et le comportement de l'acteur qui exécute le comportement dans lequel elles figurent. Elles ne sont donc accessibles que dans le contexte d'un comportement. Pour simplifier le problème de la détermination statique de l'ensemble des portions de programmes où leur utilisation est légale, nous limitons leur accès aux corps des comportements. Ainsi, si le programmeur veut qu'une fonction appelée par un comportement ait accès à l'`ego` et au `self`, il doit explicitement les lui passer en argument lors de son appel.

'`new`' *<expr>* :
La création d'un acteur est une fonction unaire dont l'argument doit s'évaluer en une interface. Elle a pour effet de bord de créer un nouvel acteur en l'initialisant avec l'interface reçue en paramètre. Le résultat de son exécution est l'adresse de l'acteur ainsi créé. Le programmeur peut alors récupérer cette adresse dans une variable pour l'utiliser ultérieurement.

'`send`' <*expr*> '`to`' <*expr*> :
L'envoi de message correspond à une forme d'application binaire, le premier argument doit s'évaluer en un message et le second doit s'évaluer en une adresse d'acteur.

'`suicide`'

'`become`' <*expr*> :
Le changement de comportement d'un acteur peut prendre deux formes : soit, l'acteur se suicide (ce qui met fin à tous ses calculs), soit, il change de comportement pour assumer une interface qui lui est fournie en paramètre. Le `become` interrompt le calcul en cours, l'acteur assume alors son nouveau comportement et un acteur anonyme est créé temporairement pour achever les calculs. Cet acteur anonyme a accès à l'`ego` et au `self` de l'acteur qui l'a engendré.

5.3 Le langage $\mathcal{F}unc_1$

Les exemples que nous présentons dans cette thèse sont écrits en ML-Act. Cependant cette syntaxe n'est exploitée ni pour la présentation du système de type, ni pour la preuve de sa correction. Pour décrire le typage de ML-Act, nous utilisons un petit langage fonctionnel dont la syntaxe est plus concise. Ce noyau, appelée $\mathcal{F}unc_1$, est introduit dans cette section, il est une extension de $\mathcal{F}unc_0$ du chapitre 4. Cette traduction n'est pas indispensable, par exemple [DPCS00] définit la sémantique et le système de type pour tout le langage. Cependant, la restriction à $\mathcal{F}unc_1$ permet une présentation plus intuitive du système de type qui contient alors un nombre de règles beaucoup plus faible.

Définition 5.1 ($\mathcal{F}unc_1$) :
Une expression fonctionnelle est un terme qui représente un programme fonctionnel. Sa définition est paramétrée par cinq ensembles : l'ensemble des variables (notées $x \in \mathbb{V}$), l'ensemble des adresses d'acteurs (notées $a \in \mathbb{A}$), l'ensemble des constantes (notées $c \in \mathbb{C}$), celui des fonctions prédéfinies (notées $f \in \mathcal{F}_c$) et celui des constructeurs (notés $C \in \mathcal{C}$).

- *L'ensemble des constantes $\mathbb{C}$ contient les entiers, les flottants (décimaux en termes mathématiques), les deux booléens (que nous notons en anglais : true et false) ainsi que les caractères, les chaînes de caractères et l'erreur. Soit :*

$$\mathbb{C} \triangleq \mathbb{N} \cup \mathbb{D} \cup \{true,\ false\} \cup Char \cup Char^n \cup \{\textbf{\textit{Err}}\}$$

- *Celui des fonctions prédéfinies $\mathcal{F}_c$ contient toutes les fonctions usuelles de comparaison et d'arithmétique de base ainsi que les fonctions du modèle d'acteur : send, new, init et become. Soit :*

$$\mathcal{F}_c \triangleq \{send,\ new,\ init,\ become,\ \texttt{@},\ \texttt{\^{}},\ \texttt{=},\ \texttt{<},\ \ldots,\ \texttt{or},\ \texttt{\&},\ \texttt{+},\ \texttt{+.},\ \ldots,\ \texttt{not},\ \texttt{hd},\ \ldots\}$$

- *L'ensemble des messages, noté $\mathcal{M}ess$, contient des données construites par un ensemble de constructeurs particuliers appelés étiquettes. Ces étiquettes possibles sont construites (par la collecte des `label`) lors de la traduction de* ML-Act *(voir section suivante), leur ensemble est noté $\mathbb{M}$. Soit :*

$$\mathcal{M}ess \triangleq \mathbb{M} \times \mathcal{E}xp^n$$

– *L'ensemble des constructeurs $\mathcal{C}$ contient donc les usuels constructeurs de listes et de tuples ainsi que toutes les étiquettes de message. Soit*

$$\mathcal{C} \triangleq \{nop,\ nil,\ cons,\ tuple2,\ \ldots,\ tuplen\} \cup \mathbb{M}$$

– *L'ensemble des valeurs sémantiques contient : les adresses d'acteurs, les constantes, les termes construits à partir de valeurs sémantiques, les abstractions*[36] *et les fonctions constantes. On a donc*

$$\mathcal{V} \triangleq \mathbb{A} \cup \mathbb{C} \cup \mathcal{C} \times \mathcal{V}^n \cup \mathcal{F} \cup \mathcal{F}_c$$

Les expressions sont construites par la grammaire qui suit :

$$e ::= x \mid a \mid c \mid f \mid C(e, \ldots, e) \mid (e) \mid e\ e \mid \lambda[p.e, \ldots, p.e] \mid \texttt{letrec}\ p = e\ \texttt{in}\ e$$
$$p ::= \_ \mid x \mid c \mid C(p, \ldots, p)$$

Une expression peut être une variable x, une adresse d'acteur a, une constante c, une fonction prédéfinie f ou un terme construit par C. Il est également possible de créer des blocs d'expressions en mettant une expression entre parenthèses. À l'image du λ-calcul, la syntaxe contient l'application ainsi que l'abstraction. Celle-ci est plus complexe que l'abstraction usuelle car elle permet de construire un filtrage à plusieurs cas $[\lambda p_1.e_1 \ldots \lambda p_n.e_n]$. Chaque cas est composé d'un motif p_i et d'une expression e_i. Intuitivement, si l'argument reçu respecte p_i, l'expression e_i fournit la suite du calcul. L'ensemble des abstractions (ou fonctions) est noté $\mathcal{F}$. Enfin, il est possible d'introduire des fonctions récursives.

Par abus de notation, l'écriture de certaines fonctions est simplifiée . Les filtrages ne comportant qu'un seul cas suivent l'écriture usuelle du λ-calcul ($\lambda[p.e]$ est noté $\lambda p.e$) et les fonctions à plusieurs paramètres sont aplaties ($\lambda p_1.\ldots.\lambda p_n.e$ est noté $\lambda p_1 \ldots p_n.e$). Pour des raisons de commodité d'écriture, nous employons parfois une version indicée des fonctions : $\lambda[p_i.e_i]^{i \in 1..n}$.

Dans les différents exemples, nous exploitons également deux autres constructions syntaxiques qui ne sont pas incluses dans la syntaxe car elles sont immédiatement traduisibles en une composition de plusieurs expressions y figurant déjà. Ce sucre syntaxique est composé de :

– $\texttt{let}\ p\ = e_1\ \texttt{in}\ e_2$ qui signifie $(\lambda p.e_2)\ e_1$,

– $e_1;\ e_2$ qui signifie $(\lambda nop.e_2)\ e_1$.

La sémantique présentera de manière rigoureuse le sens de chaque expression, mais intuitivement, elles conservent leur sens usuel dans le contexte fonctionnel. Ajoutons également que $\mathcal{F}unc_1$ suit la règle de l'appel par valeur.

Dans le premier chapitre, nous avons introduit la notion de configuration sur lesquelles nous avons défini une notion de nom libre. Pour compléter cette définition, il faut maintenant définir la notion de nom libre dans une expression de ML-Act. Remarquons, qu'un nom est **libre** dans une expression, si et seulement si ce nom n'est pas lié par un opérateur ν de configuration dans la portée duquel figure l'expression.

36. Donc, également les interfaces (du modèle d'acteur) qui sont des fonctions particulières. En effet, elles prennent un message en paramètre, le filtrent et choisissent ainsi leur réaction.

$$
\begin{array}{llll}
\mathcal{FN}(a) & = \{a\} & \mathcal{FN}(C(e_1,\ldots,e_n)) & = \bigcup\{\mathcal{FN}(e_i)|i \in 1..n\} \\
\mathcal{FN}(c) & = \{\} & \mathcal{FN}(e_1\ e_2) & = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2) \\
\mathcal{FN}(x) & = \{\} & \mathcal{FN}(\texttt{letrec}\ p = e_1\ \texttt{in}\ e_2) & = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2) \\
\mathcal{FN}((e)) & = \mathcal{FN}(e) & \mathcal{FN}(\lambda[p_i.e_i]^{i\in I}) & = \bigcup\{\mathcal{FN}(e_i)|i \in I\}
\end{array}
$$

Règles 5.1 Ensemble des noms libres d'une expression de $\mathcal{F}unc_1$.

Définition 5.2 (Nom libre) :
Tout nom n figurant dans une expression e est libre. L'ensemble des noms libres d'une expression e est noté $\mathcal{FN}(e)$ et est calculé par les règles 5.1.

Remarque :
Cette définition porte sur les noms (ou adresses) et pas sur les variables. Les noms ne peuvent pas apparaître dans les motifs de filtrage. Il n'est donc pas nécessaire de les analyser.

5.4 Traduction de ML-Act vers $\mathcal{F}unc_1$

Dans cette section, nous allons présenter la traduction d'un programme ML-Act en une configuration dont la partie fonctionnelle est $\mathcal{F}unc_1$. En fait, un programme est traduit en un acteur anonyme dont le corps sera une expression fonctionnelle correspondant au programme.

Pour simplifier cette traduction, nous supposons qu'il existe un ensemble global d'étiquettes de message $\mathbb{M}$. Chaque instruction `label` déclare un ensemble d'étiquettes qui sont ajoutées à l'ensemble $\mathbb{M}$, ce qui permet de déterminer la catégorie d'une variable lors de la traduction.

La traduction des programmes ML-Act est notée $[\![]\!]$, elle est effectuée par l'intermédiaire des quatre fonctions :

- $[\![]\!]_I$ qui prend en argument le programme et renvoie l'expression qui sera le corps de l'acteur anonyme résultat de la traduction ;
- $[\![]\!]_F$ qui traduit un groupe de n motifs en une fonction à n paramètres ;
- $[\![]\!]_B$ qui construit la fonction correspondant à un comportement ;
- $[\![]\!]_E$ qui traduit une expression.

Nous allons examiner plus en détail ces différentes fonctions de traduction dans l'ordre donné ci-dessus. Notons que les motifs forment un sous-ensemble des expressions (en y ajoutant le joker). Pour simplifier la traduction, nous utilisons donc la fonction de traduction des expressions pour les motifs.

La traduction d'un programme P est donc un acteur anonyme dont le corps est la traduction fonctionnelle de P :

$$[\![P]\!] \triangleq \star \triangleright [\![P]\!]_I$$

Lors de la traduction de P par $[\![]\!]_I$:

– Un programme vide correspond à un calcul vide :

$$[\![]\!]_I \triangleq nop$$

– Les définitions d'étiquettes (`label`) sont oubliées (les étiquettes sont ajoutées à $\mathbb{M}$) :

$$[\![\texttt{label}\ \vec{m}\texttt{;;}P]\!]_I \triangleq [\![P]\!]_I$$

– Les expressions globales sont traduites en une séquence d'expressions de $\mathcal{F}unc_1$:

$$[\![e\texttt{;;}P]\!]_I \triangleq [\![e]\!]_E;\ [\![P]\!]_I$$

– Une définition simultanée (`let` et `let rec`) est traduite en une définition simple portant sur des tuples. Notons que les arguments d'un `let` peuvent tous être des motifs (contrairement au `let rec`) :

$$[\![\texttt{let}\ p_1\ \vec{p}_1\texttt{=}\ e_1 \ldots \texttt{and}\ p_n\ \vec{p}_n\texttt{=}\ e_n\texttt{;;}P]\!]_I \triangleq$$
$$\texttt{let}\ \mathit{tuplen}([\![p_1]\!]_E, \ldots, [\![p_n]\!]_E) = \mathit{tuplen}([\![\vec{p}_1]\!]_F\ [\![e_1]\!]_E, \ldots, [\![\vec{p}_n]\!]_F\ [\![e_n]\!]_E)\ \texttt{in}\ [\![P]\!]_I$$

$$[\![\texttt{let rec}\ x_1\ \vec{p}_1\texttt{=}\ e_1 \ldots \texttt{and}\ x_n\ \vec{p}_n\texttt{=}\ e_n\texttt{;;}P]\!]_I \triangleq$$
$$\texttt{letrec}\ \mathit{tuplen}(x_1, \ldots, x_n) = \mathit{tuplen}([\![\vec{p}_1]\!]_F\ [\![e_1]\!]_E, \ldots, [\![\vec{p}_n]\!]_F\ [\![e_n]\!]_E)\ \texttt{in}\ [\![P]\!]_I$$

Lorsqu'une définition introduit une fonction, *i.e.* elle contient plusieurs arguments $p_1 \ldots p_n$, le premier paramètre p_1 est utilisé comme motif dans la définition traduite et les paramètres suivants permettent de construire une fonction à laquelle sera unifié p_1. Pour cela, une suite de filtre $p_1 \ldots p_n$ est traduite ($[\![]\!]_F$) par une métafonction[37] prenant en paramètre un objet E (E est une expression ou un corps de comportement) et renvoyant une fonction à n arguments (les p_i) dont le corps est E :

$$[\![p_1 \ldots p_n]\!]_F \triangleq \Lambda\mathrm{E}.\lambda[\![p_1]\!]_E \ldots [\![p_n]\!]_E.\mathrm{E} \qquad [\![]\!]_F \triangleq \Lambda\mathrm{E}.\mathrm{E}$$

Ainsi, `let f x y =` e est traduit par :

$$\begin{aligned}[\![\texttt{let f x y =}\ e]\!]_I &= \texttt{let f} = [\![\texttt{x y}]\!]_F\ [\![e]\!]_E\ \texttt{in}\ \ldots \\ &= \texttt{let f} = (\Lambda E.\lambda\texttt{x y}.E)\ [\![e]\!]_E\ \texttt{in}\ \ldots \\ &= \texttt{let f} = \lambda\texttt{x y}.[\![e]\!]_E\ \texttt{in}\ \ldots\end{aligned}$$

– La création de plusieurs acteurs se traduit simplement en la création simultanée de toutes leurs adresses, puis, les acteurs sont effectivement créés par l'association d'un comportement à chaque adresse, un par un, en séquence. Ainsi, la fonction qui décrit le comportement d'un des acteurs du groupe peut accéder à l'adresse des autres acteurs du groupe. Enfin, l'évaluation se poursuit par le reste du programme. Soit :

$$[\![\texttt{let rec}\ x_1 \texttt{=}\ \texttt{new}\ e_1 \ldots \texttt{and}\ x_n \texttt{=}\ \texttt{new}\ e_n\texttt{;;}P]\!]_I \triangleq$$
$$\texttt{let}\ \mathit{tuplen}(x_1, \ldots, x_n) = \mathit{tuplen}(\mathit{new}, \ldots, \mathit{new})\ \texttt{in}$$
$$\mathit{init\ tuple2}(x_1, [\![e_1]\!]_E); \ldots;\ \mathit{init\ tuple2}(x_n, [\![e_n]\!]_E);\ [\![P]\!]_I$$

37. On utilise la notation ΛE. pour ne pas mélanger ces métafonctions avec les fonctions de $\mathcal{F}unc_1$.

– Enfin, la définition de comportement autorise la récursivité et est donc traduite en définition récursive :

$$[\![\texttt{behavior}\ x_1\ \vec{p}_1\texttt{=}\ b_1 \ldots \texttt{and}\ x_n\ \vec{p}_n\texttt{=}\ b_n\texttt{;;}P]\!]_I \triangleq$$
$$\texttt{letrec}\ tuplen(x_1,\ldots,x_n) = tuplen([\![\vec{p}_1]\!]_F\ [\![b_1]\!]_B,\ldots,[\![\vec{p}_n]\!]_F\ [\![b_n]\!]_B)\ \texttt{in}\ [\![P]\!]_I$$

De plus, lors du changement de comportement, il faut lier les variables *self* et *ego* à leur valeur et traduire l'ensemble des réactions en une abstraction (une interface). La liaison de *ego* est dynamique, c'est-à-dire qu'elle est effectuée lors de la création d'un acteur qui aura ce comportement, elle est alors liée à l'adresse passée en paramètre. La variable *ego* est donc abstraite et l'initialisation consistera en une application de cette fonction sur l'adresse de l'acteur courant. Par contre, la liaison de *self* est statique, puisqu'il est connu lors de la construction d'un comportement. Le reste de la traduction est ensuite quasi-immédiate, il ne reste qu'à ajouter le *rebouclage* sur le comportement courant – *i.e. self*. Remarquons que cet ajout ne modifie pas le sens des programmes puisque, si le comportement a déjà été modifié, le code qui suit sera évalué par un acteur anonyme sur lequel les changements de comportement n'ont aucun effet (voir la sémantique du *become* page 36). la traduction de la partie interface est donc :

$$[\![\texttt{message}\ m_1\ p_1\ \texttt{=}\ e_1\ \ldots\ \texttt{message}\ m_n\ p_n\ \texttt{=}\ e_n]\!]_B \triangleq$$
$$\lambda ego.\texttt{letrec}\ self = \lambda[m_1([\![p_1]\!]_E).([\![e_1]\!]_E;\ become(ego, self))\ ,\ldots$$
$$m_n([\![p_n]\!]_E).([\![e_n]\!]_E;\ become(ego, self))]\ \texttt{in}\ self$$

Remarque :
Il est clair que si tuple1 apparaît dans le source, nous l'omettons. Nous supprimons également toute parenthèse générée en trop[38]*. De plus, par abus de notation et lorsqu'aucune ambiguïté n'est possible, nous omettons les constructeurs de tuple dans l'écriture de termes de* $\mathcal{F}unc_1$.

Les règles 5.2 décrivent la plupart des cas de traduction des expressions (et donc des motifs). Elles consistent en une simple réécriture en terme de $\mathcal{F}unc_1$. Remarquons que dans la règle de traduction des opérateurs binaires, *op* prend ses valeurs dans $\mathcal{F}_c \setminus \{::\}$. Cet opérateur est traité par une règle spécifique. La définition des liaisons de noms réutilise celle donnée sur les programmes par souci de concision.

Cependant, cinq règles qui ne sont pas immédiates doivent être ajoutées :

– La traduction d'un message : un identificateur qui figure dans $\mathbb{M}$ est considéré comme une étiquette, ainsi l'application d'un identificateur m (de $\mathbb{M}$) à une expression e est traduite en un message $m(e)$:

$$[\![m\ e]\!]_E \triangleq m([\![e]\!]_E) \quad \text{si } m \in \mathbb{M}$$

– La traduction de la conditionnelle sans cas `else` en une conditionnelle complète dont le cas `else` ne fait rien (*nop*). Cette traduction est issue de Objective CAML et impose à la branche alors (`then`) de se terminer par une valeur vide (*nop*) :

$$[\![\texttt{if}\ e_1\texttt{then}\ e_2]\!]_E \triangleq \lambda[true.[\![e_2]\!]_E, false.nop]\ [\![e_1]\!]_E$$

38. Par exemple, si nous obtenons comme filtrage interface $m(())$, nous le remplaçons par $m()$.

$$
\begin{array}{llll}
[\![\_]\!]_E \triangleq \_ & [\![e_1\ e_2]\!]_E & \triangleq [\![e_1]\!]_E\ [\![e_2]\!]_E \\
[\![x]\!]_E \triangleq x & [\![e_1\,;e_2]\!]_E & \triangleq [\![e_1]\!]_E\,;[\![e_2]\!]_E \\
[\![c]\!]_E \triangleq c & [\![\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}]\!]_E & \triangleq \mathit{tuple}n([\![e_1]\!]_E,\ldots,[\![e_n]\!]_E) \\
[\![\texttt{()}]\!]_E \triangleq \mathit{nop} & [\![\texttt{[}e_1\texttt{;}\ldots\texttt{;}e_n\texttt{]}]\!]_E & \triangleq \mathit{cons}([\![e_1]\!]_E,\ldots \mathit{cons}([\![e_n]\!]_E, \mathit{nil})) \\
[\![\texttt{[]}]\!]_E \triangleq \mathit{nil} & [\![e_1\texttt{::}e_2]\!]_E & \triangleq \mathit{cons}([\![e_1]\!]_E, [\![e_2]\!]_E) \\
[\![\mathit{un}\ e]\!]_E \triangleq \mathit{un}\ [\![e]\!]_E & [\![e_1\ \mathit{op}\ e_2]\!]_E & \triangleq \mathit{op}\ \mathit{tuple2}([\![e_1]\!]_E, [\![e_2]\!]_E) \\
[\![\texttt{ego}]\!]_E \triangleq \mathit{ego} & [\![\texttt{if}\ e_1\texttt{then}\ e_2\ \texttt{else}\ e_3]\!]_E & \triangleq \lambda[\mathit{true}.[\![e_2]\!]_E, \mathit{false}.[\![e_3]\!]_E]\ [\![e_1]\!]_E \\
[\![\texttt{self}]\!]_E \triangleq \mathit{self} & [\![\texttt{send}\ e_1\texttt{to}\ e_2]\!]_E & \triangleq \mathit{send}\ \mathit{tuple2}([\![e_1]\!]_E, [\![e_2]\!]_E)
\end{array}
$$

$$[\![\texttt{function}\ \texttt{|}\ p_1\ \texttt{->}\ e_1 \ldots \texttt{|}\ p_n\ \texttt{->}\ e_n]\!]_E \triangleq \lambda[[\![p_1]\!]_P.[\![e_1]\!]_E, \ldots, [\![p_n]\!]_P.[\![e_n]\!]_E]$$

$$[\![\texttt{match}\ e\ \texttt{with}\ \texttt{|}\ p_1\ \texttt{->}\ e_1 \ldots \texttt{|}\ p_n\ \texttt{->}\ e_n]\!]_E \triangleq \lambda[[\![p_1]\!]_P.[\![e_1]\!]_E, \ldots, [\![p_n]\!]_P.[\![e_n]\!]_E]\ [\![e]\!]_E$$

$$[\![\texttt{let}\ \vec{p}_1\texttt{=}e_1 \ldots \texttt{and}\ \vec{p}_n\texttt{=}e_n\ \texttt{in}\ e]\!]_E \triangleq [\![\texttt{let}\ \vec{p}_1\texttt{=}e_1 \ldots \texttt{and}\ \vec{p}_n\texttt{=}e_n\texttt{;;}\ e\texttt{;;}]\!]_I$$

$$[\![\texttt{let rec}\ \vec{p}_1\texttt{=}e_1 \ldots \texttt{and}\ \vec{p}_n\texttt{=}e_n\ \texttt{in}\ e]\!]_E \triangleq [\![\texttt{let rec}\ \vec{p}_1\texttt{=}e_1 \ldots \texttt{and}\ \vec{p}_n\texttt{=}e_n\texttt{;;}\ e\texttt{;;}]\!]_I$$

Règles 5.2 Traduction des filtres et des expressions de ML-ACT.

– La traduction du suicide et du changement de comportement fait appel à la fonction *become*. Celle-ci prend en argument l'*ego* et la future interface. L'adresse n'est pas utilisée directement car la valeur de *ego* est déjà définie dans l'interface (par l'application). Mais, cela permet de vérifier que l'opération a bien lieu dans le corps d'un comportement. Dans le cas d'un suicide, la future interface est vide. Rappelons que lors de la présentation de la sémantique des configurations, nous avons vu qu'un acteur de comportement vide et de queue vide *disparaît* – *i.e.* il est détruit par le ramasse-miettes (règles 1.3 (8) page 31). Donc, si l'acteur possède une boîte aux lettres vide il est détruit, sinon, il subsiste dans le terme (tout en ne participant plus au calcul).

$$[\![\texttt{suicide}]\!]_E \triangleq \mathit{become}(\mathit{ego}, \lambda[])$$

$$[\![\texttt{become}\ e]\!]_E \triangleq \mathit{become}(\mathit{ego}, [\![e]\!]_E\ \mathit{ego})$$

– La traduction de la création d'un acteur unique enchaîne directement la création d'un acteur et son initialisation. Le résultat est alors l'adresse de l'acteur créé :

$$[\![\texttt{new}\ e]\!]_E \triangleq \texttt{let}\ a = \mathit{new}\ \texttt{in}\ \mathit{init}\ \mathit{tuple2}(a, [\![e]\!]_E);\ a$$

Remarque :
Actuellement, le langage ML-ACT *ne permet pas la manipulation des étiquettes. Pour cela, la traduction peut signaler une erreur, si elle rencontre une étiquette non appliquée. Cependant, une des extensions envisagées pour le langage est de permettre l'utilisation des étiquettes comme des valeurs du premier ordre.*

Afin d'illustrer cette traduction, l'exemple 5.2 contient la traduction du programme de l'exemple 5.1 de la page 108 que nous appelons P.

À l'issue de cette traduction, l'ensemble des étiquettes de messages de P est : $\mathbb{M} = \{start,\ get,\ set,\ prn\}$.

Maintenant que nous avons décrit la syntaxe de $\mathcal{F}unc_1$ ainsi que la traduction d'un programme ML-Act en une configuration, nous pouvons définir précisément la sémantique de notre langage.

5.5 La sémantique de ML-Act

La sémantique d'un programme ML-Act est décrite par celle de sa traduction dans le formalisme des configurations. Nous avons vu dans la section précédente qu'un programme est traduit en un acteur anonyme de la forme $\star \triangleright e$. Cet acteur est ensuite évalué en appliquant les règles de la sémantique des configurations introduites dans le premier chapitre. L'instanciation par $\mathcal{F}unc_1$ requiert la définition de la sémantique de la partie fonctionnelle. Nous allons donc préciser les notions de variable et de substitution, le filtrage, la notion de nom libre et enfin, la sémantique donnée à la boîte aux lettres d'un acteur.

Variables et filtrage

Lors de l'exécution d'un programme de $\mathcal{F}unc_1$, chaque variable est *remplacée*, au fur et à mesure de l'avancée du calcul, par sa valeur (une valeur sémantique) au moyen de substitutions.

Définition 5.3 (Substitution et composition) :
Une substitution $[v/x]$ est un automorphisme sur les expressions qui remplace toutes les occurrences libres de la variable x par la valeur v. Elle est définie par induction sur la structure des expressions par les règles 5.3. Trois substitutions particulières supplémentaires sont définies : failed, $\boldsymbol{Err}$ et $[]$. Ces trois substitutions n'ont aucun effet sur l'expression. La première donne un sens aux filtrages qui ont échoué, la seconde permet de signaler une erreur sémantique survenue pendant le filtrage, et la troisième correspond à l'identité. Enfin, les substitutions se composent comme les applications avec les règles supplémentaires suivantes :

$$\begin{cases} \sigma \circ \boldsymbol{Err} \triangleq \boldsymbol{Err} \circ \sigma \triangleq \boldsymbol{Err} \\ \sigma \circ failed \triangleq failed \circ \sigma \triangleq failed \quad si\ \sigma \neq \boldsymbol{Err} \end{cases}$$

Remarque :
La substitution ne doit pas remplacer les variables qui sont redéfinies par une introduction (que ce soit une définition récursive ou un motif d'abstraction), pour cela, dans les règles 5.3, nous utilisons la fonction $\mathcal{FV}$ qui calcule l'ensemble des variables contenues dans un motif.

Le langage ML-Act hérite de la notion de filtrage de Objective CAML. La sémantique du langage dépendant en partie de ces filtrages, nous allons en présenter une

$$
\begin{aligned}
[\![P]\!] &= \star \triangleright [\![\texttt{label start, get, set, prn;; ...}]\!]_I \\
&= \star \triangleright [\![\texttt{behavior ...;; ...}]\!]_I \\
&= \star \triangleright \texttt{letrec (empty, cell)} = ([\![]\!]_F\ [\![\texttt{message start ...}]\!]_B, \\
&\qquad\qquad [\![\texttt{v}]\!]_F\ [\![\texttt{message get ...}]\!]_B) \\
&\quad\ \texttt{in } [\![\texttt{behavior ...;;}]\!]_I \\
&= \star \triangleright \texttt{letrec (empty, cell)} = (\\
&\quad\ \lambda ego.\texttt{letrec } self = \lambda[start(v).[\![\texttt{become ...}]\!]_E;\ become(ego, self)] \texttt{ in } self, \\
&\quad\ \lambda v.\lambda ego.\texttt{letrec } self = \lambda[get(c).[\![\texttt{send ...}]\!]_E;\ become(ego, self)\ , \\
&\qquad\qquad set(v').[\![\texttt{become ...}]\!]_E;\ become(ego, self)] \texttt{ in } self) \\
&\quad\ \texttt{in letrec screen} = [\![]\!]_F\ [\![\texttt{message prn ...}]\!]_B \texttt{ in } [\![\texttt{let ...;;}]\!]_I \\
&= \star \triangleright \texttt{letrec (empty, cell)} = (\\
&\quad\ \lambda ego.\texttt{letrec } self = \lambda[start(v).become(ego, (\texttt{cell } v)\ ego);\ become(ego, self)] \\
&\qquad\ \texttt{in } self, \\
&\quad\ \lambda v.\lambda ego.\texttt{letrec } self = \lambda[get(c).send(prn(v), c);\ become(ego, self)\ , \\
&\qquad\qquad set(v').become(ego, (\texttt{cell } v')\ ego); become(ego, self)] \\
&\qquad\ \texttt{in } self) \\
&\quad\ \texttt{in letrec screen} = \ \lambda ego.\texttt{letrec } self = \lambda[prn(v).[\![\texttt{print\_int ...}]\!]_E; \\
&\qquad\qquad become(ego, self)] \texttt{ in } self) \\
&\quad\ \texttt{in let (a\_cell, a\_scr)} = ([\![\texttt{new ...}]\!]_E\ ,\ [\![\texttt{new ...}]\!]_E) \texttt{ in } [\![\texttt{send ...}]\!]_E \\
&= \star \triangleright \texttt{letrec (empty, cell)} = (\\
&\quad\ \lambda ego.\texttt{letrec } self = \lambda[start(v).become(ego, (\texttt{cell } v)\ ego);\ become(ego, self)] \\
&\qquad\ \texttt{in } self, \\
&\quad\ \lambda v.\lambda ego.\texttt{letrec } self = \lambda[get(c).send(prn(v), c);\ become(ego, self)\ , \\
&\qquad\qquad set(v').become(ego, (\texttt{cell } v')\ ego); become(ego, self)] \\
&\qquad\ \texttt{in } self) \\
&\quad\ \texttt{in letrec screen} = \ \lambda ego.\texttt{letrec } self = \lambda[prn(v).\texttt{print\_int } v; \\
&\qquad\qquad become(ego, self)] \texttt{ in } self) \\
&\quad\ \texttt{in let (a\_cell, a\_scr)} = (\texttt{let } a = new \texttt{ in } init(a, \texttt{empty});\ a, \\
&\qquad\qquad \texttt{let } a = new \texttt{ in } init(a, \texttt{screen});\ a) \\
&\qquad \texttt{in } send(start(1), \texttt{a\_cell});\ send(get(\texttt{a\_scr}), \texttt{a\_cell}); \\
&\qquad\ send(set(2), \texttt{a\_cell});\ send(get(\texttt{a\_scr}), \texttt{a\_cell})
\end{aligned}
$$

Exemple 5.2 La traduction du code de la cellule en $\mathcal{F}unc_1$

$$\mathcal{FV}(\_) = \{\} \qquad \mathcal{FV}(x) = \{x\}$$
$$\mathcal{FV}(c) = \{\} \qquad \mathcal{FV}(C(p_1, \ldots, p_n)) = \bigcup\{\mathcal{FV}(p_i) | i \in 1..n\}$$

$$[v/x]a = a \qquad [v/x]f = f$$
$$[v/x]c = c \qquad [v/x](e') = ([v/x]e')$$
$$[v/x]x' = \begin{cases} v & \text{si } x' = x \\ x' & \text{sinon} \end{cases} \qquad [v/x]C(e_1, \ldots, e_n) = C([v/x]e_1, \ldots, [v/x]e_n)$$
$$[v/x]e_1\ e_2 = [v/x]e_1\ [v/x]e_2$$

$$[v/x]\texttt{letrec}\ p = e_1\ \texttt{in}\ e_2 = \texttt{letrec}\ p = e'_1\ \texttt{in}\ e'_2 \text{ avec } e'_i = \begin{cases} e_i & \text{si } x \in \mathcal{FV}(p) \\ [v/x]e_i & \text{sinon} \end{cases}$$

$$[v/x]\lambda[p_i.e_i]^{i\in I} = \lambda[p_i.e'_i]^{i\in I} \text{ avec } e'_i = \begin{cases} e_i & \text{si } x \in \mathcal{FV}(p_i) \\ [v/x]e_i & \text{sinon} \end{cases}$$

Règles 5.3 Substitution d'une variable par sa valeur dans $\mathcal{F}unc_1$.

version volontairement simpliste. Pour des études plus poussées sur le filtrage dans les langages *à la* ML, nous renvoyons à la présentation de X. Leroy dans [Ler90] et plus particulièrement à la section 5.2.4 où il explique la compilation et donc la sémantique du filtrage de Objective CAML.

Le filtrage utilise un opérateur qui prend en paramètre une valeur sémantique v et un motif p et calcule une substitution permettant le filtrage de la valeur sémantique avec son motif. Cet opérateur, dit de *confrontation*, est noté $v/p \Rightarrow \sigma$. Dans le cadre de ML-Act et donc de $\mathcal{F}unc_1$, une erreur peut survenir lors d'une tentative de filtrage. Cette erreur est nécessaire pour assurer la détection des conflits d'arité lors de la réception de message. En effet, dans la sémantique que nous souhaitons donner à ML-Act, un acteur attendant le message $m(x)$ qui reçoit un message $m(v_1, v_2)$ interrompt le programme et signale une erreur.

Les règles de déduction 5.4 définissent inductivement le filtrage selon la forme du motif. Les filtrages par le motif joker ou par une variable réussissent et renvoient respectivement la substitution vide ou celle qui remplace la variable par la valeur filtrée. Le filtrage par une constante c ne réussit que si la valeur est identique à c et si c'est le cas, la substitution obtenue est vide. Le filtrage d'un terme composé repose sur trois conditions :

- le motif et la valeur ont le même constructeur de tête ;
- le constructeur du motif et celui de la valeur ont la même arité ;
- chaque sous-motif filtre la sous-valeur correspondante.

Les contextes d'évaluation

Afin de présenter l'évaluation fonctionnelle, nous utilisons à nouveau la notion de *contexte d'évaluation*. En ML-Act, l'application de fonctions utilise l'appel par valeur – *i.e.* les arguments sont tous évalués avant d'exécuter la fonction. Il est à noter que les

$$\frac{}{v/\_ \Rightarrow []} \qquad \frac{}{v/x \Rightarrow [v/x]} \qquad \frac{}{c/c \Rightarrow []} \qquad \frac{v \neq c}{v/c \Rightarrow \mathit{failed}}$$

$$\frac{\forall k\ \forall i\ \forall v_i \quad v \neq C(v_1, \ldots, v_k)}{v/C(p_1, \ldots, p_n) \Rightarrow \mathit{failed}} \qquad \frac{k \neq n}{C(v_1, \ldots, v_k)/C(p_1, \ldots, p_n) \Rightarrow \textbf{Err}}$$

$$\frac{\forall i \quad v_i/p_i \Rightarrow \sigma_i}{C(v_1, \ldots, v_n)/C(p_1, \ldots, p_n) \Rightarrow \sigma_1 \circ \cdots \circ \sigma_n}$$

Règles 5.4 Le filtrage de ML-ACT.

arguments d'une fonction sont évalués dans un ordre quelconque[39].

Définition 5.4 (Contexte d'évaluation) :
Parmi les termes construits par la grammaire suivante, les contextes sont ceux qui ne contiennent qu'un seul trou :

$$\begin{array}{l} \mathrm{C} ::= [] \mid (\mathrm{C}) \mid C(\mathrm{A}) \mid \mathrm{C}\ e \mid v\ \mathrm{C} \mid \texttt{letrec}\ p = \mathrm{C}\ \texttt{in}\ e \\ \mathrm{A} ::= [] \mid \mathrm{A}, e \mid e, \mathrm{A} \end{array}$$

Le calcul fonctionnel

L'exécution fonctionnelle des programmes du langage $\mathcal{F}unc_1$ résulte de l'application des règles 5.5.

L'évaluation des sous-expressions est indépendante du contexte. Le résultat de la réduction d'une sous-expression peut être erroné (CONTE) auquel cas le calcul est interrompu et renvoie une erreur, ou bien fournir une autre expression (CONT) qui remplace alors la première dans le trou du contexte.

Comme toute variable introduite est substituée à l'exécution par sa valeur, aucune variable ne peut plus figurer dans une expression en cours d'évaluation sauf si elle est libre (VARE).

L'application d'une valeur non fonctionnelle (APPE1) ou d'une fonctionnelle vide (APPE2) provoque également une erreur. Si aucun motif ne convient lors d'un filtrage (situation détectée par l'application d'une fonctionnelle vide), la sémantique produit une erreur notée **Fail** pour la différencier des autres erreurs. En effet, dans le cadre de nos systèmes de type, nous ne nous intéressons pas à cette forme d'erreur. Diverses analyses permettent de la détecter (l'incomplétude d'un filtrage), la section 5.2.4 de [Ler90] présente un exemple d'une telle analyse dans le cadre d'OBJECTIVE CAML. Cependant, ces systèmes sophistiqués sont totalement orthogonaux à notre objectif (le typage de la communication) et donc pourraient s'intégrer. Une autre approche possible pour supprimer cette forme d'erreur serait d'imposer à tous les filtrages de contenir un motif acceptant toute entrée (joker ou variable). Lors de l'application d'une fonction qui comporte plusieurs branches, les cas sont examinés dans l'ordre d'écriture. Le premier motif qui filtre

39. En fait, dans les implantations actuelles de OBJECTIVE CAML, il est de droite à gauche. Mais, c'est un détail d'implantation susceptible de changer dans le futur.

$$\textsc{Cont}: \frac{e \longrightarrow_e e'}{\mathrm{C}[e] \longrightarrow_e \mathrm{C}[e']} \qquad \textsc{ContE}: \frac{e \longrightarrow_e \mathbf{Err}}{\mathrm{C}[e] \longrightarrow_e \mathbf{Err}} \qquad \textsc{VarE}: \frac{}{x \longrightarrow_e \mathbf{Err}}$$

$$\textsc{AppE1}: \frac{v \notin \mathcal{F} \cup \mathcal{F}_c}{v\ e \longrightarrow_e \mathbf{Err}} \qquad \textsc{AppE2}: \frac{}{\lambda[]\ e \longrightarrow_e \mathbf{Fail}} \qquad \textsc{AppE3}: \frac{v/p_1 \Rightarrow \mathbf{Err}}{\lambda[p_i.e_i]^{i\in 1..n}\ v \longrightarrow_e \mathbf{Err}}$$

$$\textsc{AppF}: \frac{v/p_1 \Rightarrow \mathit{failed}}{\lambda[p_i.e_i]^{i\in 1..n}\ v \longrightarrow_e \lambda[p_i.e_i]^{i\in 2..n}\ v} \qquad \textsc{App}: \frac{v/p_1 \Rightarrow \sigma}{\lambda[p_i.e_i]^{i\in 1..n}\ v \longrightarrow_e \sigma(e_1)}$$

$$\textsc{LRec}: \frac{\sigma_i = [\texttt{letrec}\ \vec{x} = \vec{v}\ \texttt{in}\ v_i/x_i]}{\texttt{letrec}\ \vec{x} = \vec{v}\ \texttt{in}\ e \longrightarrow_e \sigma_1 \circ \cdots \circ \sigma_n(e)}$$

Règles 5.5 Les réductions fonctionnelles en ML-Act.

la valeur passée en paramètre fournit la branche qui est exécutée. Pour symboliser ce mécanisme, la valeur est confrontée au premier motif, trois cas peuvent alors se présenter :

- AppE3 : le filtrage se termine par une erreur sémantique qui est alors propagée ;
- AppF : le filtrage échoue et le calcul se poursuit alors en réappliquant la fonction privée de son premier motif à la valeur ;
- App : le filtrage réussit et résulte en une substitution, le résultat est alors le corps du cas sur lequel la substitution est appliquée.

La définition récursive (LRec) ne peut être que la construction d'un tuple de variables (éventuellement réduit à une seule variable). Son résultat est son corps e dans lequel chaque variable x_i introduite est substituée par sa valeur v_i. Notons que, dans chaque v_i, nous réintroduisons la définition récursive qui sera nécessaire à la prochaine utilisation d'une des variables x_i. Il est clair qu'une telle règle d'évaluation peut conduire à un bouclage sans fin des calculs. La valeur corps d'une telle définition récursive doit avoir un comportement *paresseux*, en d'autre terme, il peut s'agir d'une fonction ou d'une interface (qui ne s'évaluent que lorsqu'elles sont utilisées).

Les opérateurs binaires et unaires (opérations arithmétiques, ...) se comportent de manière classique en se réduisant selon les δ-règles usuelles que nous ne donnons pas ici.

La sémantique des boîtes aux lettres

En ML-Act, chaque message de la queue d'un acteur a est filtré par tous les cas de l'interface courante de a (s'il est inactif). Il s'agit donc d'étendre les règle du chapitre 4 en y intégrant les filtres. Cet extension est décrite par les règles 5.6 dans lesquelles, nous supposons que l'ensemble I indexant les cas est non vide (un acteur ayant une interface vide ne consulte jamais sa boîte aux lettres).

Si le filtrage du message en tête de file par les $j-1$ premiers motifs a échoué, trois cas se présentent lors du filtrage par le j^{e} motif :

- BalE : le filtrage provoque une erreur et l'accès à la queue renvoie également une erreur ;

$$\textsc{BalE}: \frac{\exists j \quad (\forall i < j \;\; m/p_i \Rightarrow \mathit{failed}) \qquad m/p_j \Rightarrow \mathbf{Err}}{\lambda[p_i.e_i]^{i\in I} \prec m :: q \Longrightarrow q,\ \mathbf{Err},\ []}$$

$$\textsc{Bal1}: \frac{\exists j \quad (\forall i < j \;\; m/p_i \Rightarrow \mathit{failed}) \qquad m/p_j \Rightarrow \sigma}{\lambda[p_i.e_i]^{i\in I} \prec m :: q \Longrightarrow q,\ e_j,\ \sigma}$$

$$\textsc{Bal2}: \frac{(\forall i \in I \;\; m/p_i \Rightarrow \mathit{failed}) \qquad \lambda[p_i.e_i]^{i\in I} \prec q \Longrightarrow q',\ e,\ \sigma}{\lambda[p_i.e_i]^{i\in I} \prec m :: q \Longrightarrow m :: q',\ e,\ \sigma}$$

Règles 5.6 Les boites aux lettres de ML-ACT.

- BAL1 : le filtrage réussit et le résultat est alors composé du reste de la file, du j^{e} corps et de la substitution issue du filtrage ;
- le filtrage échoue et le processus continue avec le motif suivant.

En fin de processus, si aucun des motifs ne convient pour filtrer m (BAL2), on essaye de filtrer le message suivant de la queue.

Remarque :
Notons qu'après traitement, les messages dont le filtrage a échoué, sont remis dans la boîte aux lettres en préservant l'ordre initial.

Un exemple

Pour donner un aperçu du fonctionnement du processus de réduction, étudions quelques étapes d'une des réductions possibles du programme P (exemple 5.1 de la page 108) dont nous avons donné une traduction page 119. Toutes les étapes intermédiaires de la réduction ne sont pas données et nous nous permettons de réduire simultanément plusieurs sous-termes d'une configuration. Chaque réduction fournie est étiquetée par le nom de la ou des règles appliquées. Afin d'améliorer la lisibilité, nous encadrons les étapes principales et nous utilisons des lettres pour remplacer certaines portions de programme :

- N pour désigner la création du nom a : `let` $a = \mathit{new}$
- L_1 et L_2 qui représentent respectivement la définition récursive des deux comportements de cellule et celui de l'écran :

$$\texttt{letrec}\ (\texttt{empty}, \texttt{cell}) = (\lambda \mathit{ego}.e_1, \lambda v.e_2) \quad \text{et} \quad \texttt{letrec}\ \texttt{screen} = \lambda \mathit{ego}.e_3$$

- S pour dénoter l'envoi des quatre messages :

$$\begin{array}{l} \mathit{send}(\mathit{start}(1), \texttt{a\_cell})\,;\, \mathit{send}(\mathit{get}(\texttt{a\_scr}), \texttt{a\_cell})\,; \\ \mathit{send}(\mathit{set}(2), \texttt{a\_cell})\,;\, \mathit{send}(\mathit{get}(\texttt{a\_scr}), \texttt{a\_cell}) \end{array}$$

Sachant que les expressions décrivant le corps des trois comportements sont :

$$e_1 \triangleq \texttt{letrec}\ \mathit{self} = \lambda[\mathit{start}(v).\mathit{become}(\mathit{ego}, (\texttt{cell}\ v)\ \mathit{ego})\,;\, \mathit{become}(\mathit{ego}, \mathit{self})]\ \texttt{in}\ \mathit{self}$$

$$e_2 \triangleq \lambda \mathit{ego}.\texttt{letrec}\ \mathit{self} = \lambda[\mathit{get}(c).\mathit{send}(\mathit{prn}(v), c)\,;\, \mathit{become}(\mathit{ego}, \mathit{self})\ , \quad \mathit{set}(v').\mathit{become}(\mathit{ego}, (\texttt{cell}\ v')\ \mathit{ego})\,;\, \mathit{become}(\mathit{ego}, \mathit{self})]\ \texttt{in}\ \mathit{self}$$

$$e_3 \triangleq \texttt{letrec}\ \mathit{self} = \lambda[\mathit{prn}(v).\texttt{print\_int}\ v\,;\, \mathit{become}(\mathit{ego}, \mathit{self})]\ \texttt{in}\ \mathit{self}$$

En utilisant ces diverses notations, la traduction du programme P s'écrit :

$$P \triangleq \star \triangleright \left(\begin{array}{l} L_1 \texttt{ in } L_2 \texttt{ in let } (\texttt{a\_cell}, \texttt{a\_scr}) = (N \texttt{ in } \mathit{init}(a, \texttt{empty})\,;a, \\ \qquad\qquad\qquad\qquad\qquad\qquad N \texttt{ in } \mathit{init}(a, \texttt{screen})\,;a) \texttt{ in } S \end{array} \right)$$

Son exécution commence par la définition récursive L_1 des deux comportements de la cellule `empty` et `cell` (LREC). Ensuite, la définition du comportement de l'écran est effectuée. Durant ces deux étapes, chaque occurrence du nom des trois comportements est substituée par sa valeur (L_1 `in` $\lambda ego.e_i$ pour $i \in \{1,\ 2\}$ et L_2 `in` $\lambda ego.e_3$).

$$P \xrightarrow{(\text{LREC})^2} \star \triangleright \left(\begin{array}{l} \texttt{let } (\texttt{a\_cell}, \texttt{a\_scr}) = (N \texttt{ in } \mathit{init}(a, L_1 \texttt{ in } \lambda ego.e_1)\,;a, \\ \qquad\qquad\qquad\qquad\qquad\qquad N \texttt{ in } \mathit{init}(a, L_2 \texttt{ in } \lambda ego.e_3)\,;a) \texttt{ in } S \end{array} \right)$$

Puis, les deux composantes de la paire de création sont évaluées. La réduction d'un des éléments de cette paire suit les étapes NEW puis LET [40], de la manière suivante :

$$\begin{array}{rl} N \texttt{ in } \mathit{init}(a, L_1 \texttt{ in } \lambda ego.e_1)\,;a & \xrightarrow{\text{NEW}}_e \texttt{let } a = a_1 \texttt{ in } \mathit{init}(a, L_1 \texttt{ in } \lambda ego.e_1)\,;a \\ & \xrightarrow{\text{LET}}_e \mathit{init}(a_1, L_1 \texttt{ in } \lambda ego.e_1)\,;a_1 \end{array}$$

De manière identique, on a :

$$\begin{array}{rl} N \texttt{ in } \mathit{init}(a, L_2 \texttt{ in } \lambda ego.e_3)\,;a & \xrightarrow{\text{NEW}}_e \texttt{let } a = a_2 \texttt{ in } \mathit{init}(a, L_2 \texttt{ in } \lambda ego.e_3)\,;a \\ & \xrightarrow{\text{LET}}_e \mathit{init}(a_2, L_2 \texttt{ in } \lambda ego.e_3)\,;a_2 \end{array}$$

Sur le terme obtenu, les deux acteurs a_1 et a_2 sont initialisés et les *nop* résultants sont éliminés par l'application des règles INIT, SEQ [41], INIT, SEQ :

$$\nu a_1. \left(\star \triangleright \left(\begin{array}{l} \texttt{let } (\texttt{a\_cell}, \texttt{a\_scr}) = (\mathit{init}(a_1, L_1 \texttt{ in } \lambda ego.e_1)\,;a_1, \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{init}(a_2, L_2 \texttt{ in } \lambda ego.e_3)\,;a_2) \texttt{ in } S \end{array} \right) \right) \xrightarrow{\text{INIT}}$$

$$\nu a_1. \left(\begin{array}{l} \star \triangleright \left(\begin{array}{l} \texttt{let } (\texttt{a\_cell}, \texttt{a\_scr}) = (\mathit{nop}\,;a_1, \\ \qquad\qquad\qquad\qquad\qquad\qquad N \texttt{ in } \mathit{init}(a, L_2 \texttt{ in } \lambda ego.e_3)\,;a) \texttt{ in } S \end{array} \right) \\ \| \ \langle a_1 \,|\, \varnothing \rangle \triangleright L_1 \texttt{ in } \lambda ego.e_1 \ a_1 \end{array} \right) \xrightarrow{\text{SEQ, INIT, SEQ}}$$

$$\boxed{\nu a_1. \left(\begin{array}{l} \nu a_2. (\star \triangleright \texttt{let } (\texttt{a\_cell}, \texttt{a\_scr}) = (a_1, a_2) \texttt{ in } S \ \| \ \langle a_2 \,|\, \varnothing \rangle \triangleright L_2 \texttt{ in } \lambda ego.e_3 \ a_2) \\ \| \ \langle a_1 \,|\, \varnothing \rangle \triangleright L_1 \texttt{ in } \lambda ego.e_1 \ a_1 \end{array} \right)}$$

Pour continuer, les trois sous-termes principaux de ce terme sont réduits en parallèle. Sur le premier, la définition (LET) est évaluée puis les quatre messages [42] sont envoyés ((SEND,SEQ)^4). Sur les deux autres, la première définition récursive (LREC) est évaluée

40. Cette règle qui n'existe pas dans la sémantique puisque le `let` est un sucre syntaxique correspond en fait à l'application de la règle APP.

41. De la même manière que pour le `let` cette règle correspond à la règle APP.

42. Notons qu'ici nous faisons un choix, car, par exemple la réception du premier message pourrait intervenir avant l'envoi des autres messages.

puis l'application (APP) et ensuite le deuxième niveau de définition récursive (LREC). Pour cela, une nouvelle définition est utilisée : $L_s \triangleq \texttt{letrec}\ \mathit{self}$ ainsi que le symbole $\hookleftarrow$. Celui-ci a l'intérieur d'un comportement indique le rebouclage, *i.e.* il référence le comportement le contenant.

$$
\begin{array}{l}
\star \triangleright \texttt{let}\ (\texttt{a\_cell}, \texttt{a\_scr}) = (a_1, a_2)\ \texttt{in}\ S \\
\quad \xrightarrow{\text{LET}} \star \triangleright (\mathit{send}(\mathit{start}(1), a_1)\,;\mathit{send}(\mathit{get}(a_2), a_1)\,;\mathit{send}(\mathit{set}(2), a_1)\,;\mathit{send}(\mathit{get}(a_2), a_1)) \\
\quad \xrightarrow{\text{SEND}} \star \triangleright (\mathit{nop}\,;\mathit{send}(\mathit{get}(a_2), a_1)\,;\mathit{send}(\mathit{set}(2), a_1)\,;\mathit{send}(\mathit{get}(a_2), a_1)) \parallel a_1 \triangleleft \mathit{start}(1) \\
\quad \xrightarrow{(\text{SEQ, SEND})^3} \star \triangleright \mathit{nop} \parallel a_1 \triangleleft \mathit{start}(1) \parallel a_1 \triangleleft \mathit{get}(a_2) \parallel a_1 \triangleleft \mathit{set}(2) \parallel a_1 \triangleleft \mathit{get}(a_2)
\end{array}
$$

$$
\begin{array}{l}
\langle a_1 \,|\, \varnothing \rangle \triangleright L_1\ \texttt{in}\ \lambda \mathit{ego}.e_1\ a_1 \\
\quad \xrightarrow{\text{LREC}} \langle a_1 \,|\, \varnothing \rangle \triangleright (\lambda \mathit{ego}.L_s = \lambda[\mathit{start}(v).\mathit{become}(\mathit{ego}, ((L_1\ \texttt{in}\ \lambda v.e_2)\ v)\ \mathit{ego}); \\
\qquad\qquad\qquad\qquad \mathit{become}(\mathit{ego}, \mathit{self})]\ \texttt{in}\ \mathit{self})\ a_1 \\
\quad \xrightarrow{\text{APP}} \langle a_1 \,|\, \varnothing \rangle \triangleright (L_s = \lambda[\mathit{start}(v).\mathit{become}(a_1, ((L_1\ \texttt{in}\ \lambda v.e_2)\ v)\ a_1); \mathit{become}(a_1, \mathit{self})] \\
\qquad\qquad \texttt{in}\ \mathit{self}) \\
\quad \xrightarrow{\text{LREC}} \langle a_1 \,|\, \varnothing \rangle \triangleright \lambda[\mathit{start}(v).\mathit{become}(a_1, ((L_1\ \texttt{in}\ \lambda v.e_2)\ v)\ a_1);\ \hookleftarrow]
\end{array}
$$

$$
\begin{array}{l}
\langle a_2 \,|\, \varnothing \rangle \triangleright L_2\ \texttt{in}\ \lambda \mathit{ego}.e_3\ a_2 \\
\quad \xrightarrow{\text{LREC}} \langle a_2 \,|\, \varnothing \rangle \triangleright (\lambda \mathit{ego}.L_s = \lambda[\texttt{print\_int}\ v\,;\mathit{become}(\mathit{ego}, \mathit{self})]\ \texttt{in}\ \mathit{self})\ a_2 \\
\quad \xrightarrow{\text{APP}} \langle a_2 \,|\, \varnothing \rangle \triangleright (L_s = \lambda[\texttt{print\_int}\ v\,;\mathit{become}(a_2, \mathit{self})]\ \texttt{in}\ \mathit{self}) \\
\quad \xrightarrow{\text{LREC}} \langle a_2 \,|\, \varnothing \rangle \triangleright \lambda[\mathit{prn}(v).\texttt{print\_int}\ v\,;\hookleftarrow]
\end{array}
$$

Nous allons à nouveau faire un choix en supposant que l'acteur a_1 reçoit dans un certain ordre les quatre messages ($(\text{RCV})^4$) qui lui ont été envoyés avant de réagir (REA). Nous commençons également à supprimer les sous-configurations vides de la configuration.

$$
\begin{array}{l}
\boxed{\begin{array}{l}
\nu a_1.\nu a_2.(a_1 \triangleleft \mathit{start}(1) \parallel a_1 \triangleleft \mathit{get}(a_2) \parallel a_1 \triangleleft \mathit{set}(2) \parallel a_1 \triangleleft \mathit{get}(a_2) \\
\quad \parallel \langle a_1 \,|\, \varnothing \rangle \triangleright \lambda[\mathit{start}(v).\mathit{become}(a_1, ((L_1\ \texttt{in}\ \lambda v.e_2)\ v)\ a_1);\ \hookleftarrow] \\
\quad \parallel \langle a_2 \,|\, \varnothing \rangle \triangleright \lambda[\mathit{prn}(v).\texttt{print\_int}\ v\,;\hookleftarrow])
\end{array}} \\
\quad \xrightarrow{(\text{RCV})^4} \nu a_1.\nu a_2.(\langle a_2 \,|\, \varnothing \rangle \triangleright \lambda[\mathit{prn}(v)\ldots] \\
\qquad\qquad \parallel \langle a_1 \,|\, \mathit{start}(1)::\mathit{get}(a_2)::\mathit{set}(2)::\mathit{get}(a_2) \rangle \triangleright \lambda[\mathit{start}(v)\ldots]) \\
\quad \xrightarrow{\text{REA}} \boxed{\begin{array}{l}
\nu a_1.\nu a_2.(\langle a_2 \,|\, \varnothing \rangle \triangleright \lambda[\mathit{prn}(v)\ldots] \\
\quad \parallel \langle a_1 \,|\, \mathit{get}(a_2)::\mathit{set}(2)::\mathit{get}(a_2) \rangle \triangleright (\mathit{become}(a_1, ((L_1\ \texttt{in}\ \lambda v.e_2)\ 1)\ a_1); \\
\qquad\qquad\qquad\qquad \mathit{become}(a_1, \lambda[\mathit{start}(v)\ldots])))
\end{array}}
\end{array}
$$

Le calcul se poursuit par la détermination du futur comportement de a_1 pour cela, la définition récursive L_1 est appliquée (LREC) puis deux applications sont réalisées ($(\text{APP})^2$)

et enfin, le *self* est introduit (LRec).

$$
\begin{aligned}
&((L_1 \texttt{ in } \lambda v.e_2)\ 1)\ a_1\\
&\quad \xrightarrow{\text{LRec}}_e \lambda v.(\lambda ego.L_s = \lambda[get(c).send(prn(v),c)\,;become(ego,self)\ ,\\
&\qquad\qquad set(v').become(ego,((L_1 \texttt{ in } \lambda v.e_2)\ v')\ ego);\\
&\qquad\qquad\qquad become(ego,self)]\ \texttt{in}\ self)\ 1\ a_1\\
&\quad \xrightarrow{\text{App}}_e \ \lambda ego.L_s = \lambda[get(c).send(prn(1),c)\,;become(ego,self)\ ,\\
&\qquad\qquad set(v').become(ego,((L_1 \texttt{ in } \lambda v.e_2)\ v')\ ego);\\
&\qquad\qquad\qquad become(ego,self)]\ \texttt{in}\ self)\ a_1\\
&\quad \xrightarrow{\text{App}}_e \ L_s = \lambda[get(c).send(prn(1),c)\,;become(a_1,self)\ ,\\
&\qquad\qquad set(v').become(a_1,((L_1 \texttt{ in } \lambda v.e_2)\ v')\ a_1)\,;become(a_1,self)]\ \texttt{in}\ self\\
&\quad \xrightarrow{\text{LRec}}_e \lambda[get(c).send(prn(1),c)\,;\hookleftarrow\ ,\\
&\qquad\qquad set(v').become(a_1,((L_1 \texttt{ in } \lambda v.e_2)\ v')\ a_1)\,;\hookleftarrow]
\end{aligned}
$$

L'acteur a_1 peut alors adopter le comportement calculé (Bec). L'acteur anonyme poursuivant le calcul change de comportement (Seq puis Bec) et ainsi crée un autre acteur anonyme. Finalement, ils deviennent tous les deux inactifs après avoir terminé leurs évaluations respectives. Nous notons q la boîte aux lettres courante de a_1 ($q \triangleq get(a_2)::set(2)::get(a_2)$).

$$
\begin{aligned}
&\langle a_1 \,|\, q\rangle \triangleright become(a_1,\lambda[get(c)\ldots,set(v')\ldots])\,;become(a_1,\lambda[start(v)\ldots])\\
&\quad \xrightarrow{\text{become}} \langle a_1 \,|\, q\rangle \triangleright \lambda[get(c)\ldots,set(v')\ldots] \parallel \star \triangleright nop\,;become(a_1,\lambda[start(v)\ldots])\\
&\quad \xrightarrow{\text{Seq}} \quad \langle a_1 \,|\, q\rangle \triangleright \lambda[get(c)\ldots,set(v')\ldots] \parallel \star \triangleright become(a_1,\lambda[start(v)\ldots])\\
&\quad \xrightarrow{\text{become}} \langle a_1 \,|\, q\rangle \triangleright \lambda[get(c)\ldots,set(v')\ldots] \parallel \star \triangleright nop \parallel \star \triangleright \lambda[start(v)\ldots]
\end{aligned}
$$

L'acteur a_1 peut alors traiter le message suivant de sa queue (Rea) avec ce nouveau comportement. Il envoie donc un message d'impression à a_2 (Send et Seq) puis reprend son comportement courant (Bec).

$$
\begin{aligned}
&\langle a_1 \,|\, get(a_2)::set(2)::get(a_2)\rangle \triangleright \lambda[get(c)\ldots,set(v')\ldots]\\
&\quad \xrightarrow{\text{Rea}} \langle a_1 \,|\, set(2)::get(a_2)\rangle \triangleright send(prn(1),a_2)\,;become(a_1,\lambda[get(c)\ldots,set(v')\ldots])\\
&\quad \xrightarrow{\text{Send}} \langle a_1 \,|\, set(2)::get(a_2)\rangle \triangleright nop\,;become(a_1,\lambda[get(c)\ldots,set(v')\ldots]) \parallel a_2 \triangleleft prn(1)\\
&\quad \xrightarrow{\text{Seq}} \ \langle a_1 \,|\, set(2)::get(a_2)\rangle \triangleright become(a_1,\lambda[get(c)\ldots,set(v')\ldots]) \parallel a_2 \triangleleft prn(1)\\
&\quad \xrightarrow{\text{Bec}} \ \boxed{\langle a_1 \,|\, set(2)::get(a_2)\rangle \triangleright \lambda[get(c)\ldots,set(v')\ldots] \parallel a_2 \triangleleft prn(1) \parallel \star \triangleright nop}
\end{aligned}
$$

L'acteur a_1 peut à nouveau traiter le prochain message de sa queue. Ce message *set* est traité de manière identique à la précédente mise à jour par le calcul du futur comportement (Rea, LRec, (App)^2 et LRec). Ce nouveau comportement est en fait quasiment identique au comportement courant excepté la valeur stockée qui vaut maintenant 2. Nous notons le comportement de cellule *cell* avec en indice la valeur qu'elle contient. Puis, a_1

assume son nouveau comportement (BEC) et les acteurs anonymes créés terminent leurs calculs (SEQ et BEC).

$$
\begin{array}{ll}
\langle a_1 \,|\, set(2) :: get(a_2)\rangle \triangleright cell_1 & \\
\xrightarrow{\text{REA}} & \langle a_1 \,|\, get(a_2)\rangle \triangleright become(a_1, ((L_1 \texttt{ in } \lambda v.e_2)\ 2)\ a_1)\,;\, become(a_1, cell_1) \\
\xrightarrow{\text{LREC, (APP)}^2\text{, LREC}} & \langle a_1 \,|\, get(a_2)\rangle \triangleright become(a_1, cell_2)\,;\, become(a_1, cell_1) \\
\xrightarrow{\text{BEC}\|\text{SEQ, BEC}} & \boxed{\langle a_1 \,|\, get(a_2)\rangle \triangleright cell_2 \parallel \star \triangleright cell_1 \parallel \star \triangleright nop}
\end{array}
$$

En parallèle, a_2 reçoit et traite la requête d'impression que lui a envoyé a_1 (RCV, REA, APP, SEQ et BEC). Cela provoque l'affichage de 1 à l'écran lors de l'application de la fonction prédéfinie `print_int`. Pour simplifier l'écriture, nous notons le comportement d'impression *printer*.

$$
\begin{array}{ll}
\langle a_2 \,|\, \varnothing\rangle \triangleright printer \parallel a_2 \triangleleft prn(1) & \xrightarrow{\text{RCV}} \langle a_2 \,|\, prn(1)\rangle \triangleright printer \\
& \xrightarrow{\text{REA}} \langle a_2 \,|\, \varnothing\rangle \triangleright \texttt{print\_int}\ 1\,;\, become(a_2, printer) \\
& \xrightarrow{\text{APP}} \langle a_2 \,|\, \varnothing\rangle \triangleright nop\,;\, become(a_2, printer) \\
& \xrightarrow{\text{SEQ}} \langle a_2 \,|\, \varnothing\rangle \triangleright become(a_2, printer) \\
& \xrightarrow{\text{BEC}} \boxed{\langle a_2 \,|\, \varnothing\rangle \triangleright printer \parallel \star \triangleright nop}
\end{array}
$$

Le terme atteint est donc :

$$\boxed{\nu a_1.\nu a_2.(\langle a_1 \,|\, get(a_2)\rangle \triangleright cell_2 \parallel \langle a_2 \,|\, \varnothing\rangle \triangleright printer)}$$

La dernière partie du calcul est la réaction de a_1 au dernier message de sa boite aux lettres (REA). Lors de cette réaction (SEND, SEQ et BEC), a_1 envoie une nouvelle requête d'impression à a_2 qui la traite. Ce traitement (RCV, REA, APP, SEQ et BEC) provoque l'affichage de 2 à l'écran.

$$
\begin{array}{ll}
\langle a_1 \,|\, get(a_2)\rangle \triangleright cell_2 & \xrightarrow{\text{REA}} \langle a_1 \,|\, \varnothing\rangle \triangleright send(prn(2), a_2)\,;\, become(a_1, cell_2) \\
& \xrightarrow{\text{SEND,SEQ}} \langle a_1 \,|\, \varnothing\rangle \triangleright become(a_1, cell_2) \parallel a_2 \triangleleft prn(2) \\
& \xrightarrow{\text{BEC}} \langle a_1 \,|\, \varnothing\rangle \triangleright cell_2 \parallel a_2 \triangleleft prn(2) \parallel \star \triangleright nop
\end{array}
$$

$$
\begin{array}{ll}
\langle a_2 \,|\, \varnothing\rangle \triangleright printer \parallel\ a_2 \triangleleft prn(2) & \xrightarrow{\text{RCV}} \langle a_2 \,|\, prn(1)\rangle \triangleright printer \\
& \xrightarrow{\text{REA}} \langle a_2 \,|\, \varnothing\rangle \triangleright \texttt{print\_int}\ 1\,;\, become(a_2, printer) \\
& \xrightarrow{\text{APP,SEQ}} \langle a_2 \,|\, \varnothing\rangle \triangleright become(a_2, printer) \\
& \xrightarrow{\text{BEC}} \langle a_2 \,|\, \varnothing\rangle \triangleright printer \parallel \star \triangleright nop
\end{array}
$$

Le terme final de cette exécution ne contient alors plus aucun sous-terme réductible :

$$\boxed{\nu a_1.\nu a_2.(\langle a_1 \,|\, \varnothing\rangle \triangleright cell_2 \parallel \langle a_2 \,|\, \varnothing\rangle \triangleright printer)}$$

5.6 Conclusion

Dans ce chapitre, nous avons décrit succinctement le langage ML-ACT et défini sa sémantique en donnant sa traduction dans le formalisme des configurations introduit au premier chapitre. Pour cela, nous avons restreint la partie fonctionnelle à un λ-calcul avec constante dont nous avons explicité la sémantique. Cette approche permet de simplifier grandement l'exposé de la sémantique de ML-ACT, mais surtout, elle permet de formaliser plus simplement le système de type que nous utilisons pour détecter les erreurs dans un programme écrit en ML-ACT. Le principal intérêt de cette traduction apparaît donc dans le chapitre suivant.

Deux prototypes de compilateur pour le langage ML-ACT ont été développés au sein de notre équipe de recherche.

Un premier, réalisé durant mon stage de DEA (voir [Dag97]) qui est relativement simple. À partir d'un programme ML-ACT, il effectue un typage simple de la partie fonctionnelle du programme et extrait un terme CAP qui exprime sa partie concurrente[43]. Si un programme est correctement typé, le compilateur traduit le programme en OBJECTIVE CAML en utilisant des bibliothèques spécifiques. Le compilateur OBJECTIVE CAML produit alors un exécutable correspondant au programme d'acteur source. Cet exécutable fonctionne de manière centralisée et simule les acteurs en utilisant des processus légers partageant une mémoire commune. Celle-ci contient la définition des fonctions et des données globales ainsi que les boîtes aux lettres de tous les acteurs. L'accès en mémoire commune permet de déposer simplement un message dans la boîte aux lettres de son destinataire. Ce dépôt s'effectue après un délai non nul tiré au hasard. Ce système permet de représenter le transit des messages dans un réseau, et ainsi de simuler l'indéterminisme de la durée de ce transit. La concurrence d'accès aux boîtes aux lettres entre leurs possesseurs et les expéditeurs est résolue par un couple variable *condition* et *verrou d'exclusion mutuelle*. Ce premier compilateur très simpliste, nous a permis de mieux cerner les problèmes à résoudre pour la réalisation d'une version répartie. Les deux points cruciaux d'une architecture d'acteurs répartis sont la migration de données et d'acteurs entre les différents sites et la transmission sûre des messages.

Le second compilateur en cours de finalisation a principalement été développé par D. CHEMOUIL dans le cadre de son stage de DEA (voir [Che00]). Le cœur de ce travail a été la réalisation d'une implantation réellement répartie du langage. Une première solution a été la conception en OBJECTIVE CAML d'un *middleware* distribué que le programme généré utilise pour toutes les opérations concurrentes (création d'acteur, envoi de message, migration d'acteur ...). Cependant, ce prototype n'a pu être finalisé, car des problèmes avec les bibliothèques de transmission de données (Marshalling en anglais) d'OBJECTIVE CAML, n'ont pas permis une implantation complète. Par exemple, la transmission de comportement n'est pas possible dans ce prototype. En fait, le problème provient du choix fait par OBJECTIVE CAML sur la transmission de fonction qui prend la forme de l'envoi d'un pointeur de code et qui donc impose a tous les sites d'avoir accès au même binaire. Une seconde stratégie a donc été choisie, la traduction d'un programme ML-ACT en JOCAML. Ce langage est une extension d'OBJECTIVE CAML dans le domaine

43. Nous présentons plus précisément cette partie typage dans le cadre du chapitre suivant.

distribué basé sur le *calcul join* (le lecteur intéressé pourra consulter [FG96], [FGL+96] et le portail internet qui lui est consacré `http://pauillac.inria.fr/join`). Il est développé à l'INRIA par M. Le Fessant dans le cadre de sa thèse, [CLF99] permet de découvrir les principes de JoCAML. L'équipe développant JoCAML a été confrontée au même problème de transmission. Ils l'ont résolu en remplaçant la transmission de pointeur de code par un chargement dynamique à distance. Cette deuxième version du prototype a donné satisfaction et est en cours de finalisation par l'intégration du système de type présenté dans le chapitre suivant.

Chapitre 6

Le typage de ML-Act

Le but des travaux réalisés dans le cadre de cette thèse est d'essayer de garantir statiquement, c'est-à-dire sans l'exécuter, que l'évaluation d'un programme ne produira pas certaines erreurs. Pour cela, nous avons choisi de développer des techniques d'inférence de type. Nous commençons par préciser le calcul du potentiel d'un acteur en ML-Act pour donner une forme définitive à la sémantique. Ensuite, nous précisons les différentes erreurs qui peuvent se produire. Puis, en nous basant sur les chapitres précédents, nous définissons le système de type de ML-Act dans le cadre de $\mathcal{F}unc_1$ et présentons son fonctionnement sur un exemple. En revanche, nous ne parlerons pas de la résolution des contraintes qui est une extension immédiate de celle introduite dans le chapitre 4. Nous prouvons alors la correction de ce système de type, c'est-à-dire le fait qu'un programme typé ne peut conduire à une erreur (**Err**). Enfin, nous concluons en discutant des différentes extensions possibles de ce système de type ainsi que de son implantation.

6.1 Les erreurs

La sémantique du langage a été présentée au fur et à mesure de l'introduction des différents concepts contenus dans le langage, l'annexe B page 247 contient toutes les règles concernant la sémantique des configurations et de $\mathcal{F}unc_1$ pour faciliter leur analyse. Les règles concernant le calcul du potentiel d'un acteur (qui a été définit dans le chapitre 2) doivent encore être présentées. En effet, elles sont nécessaires à la sémantique pour la règle Init où un acteur est marqué (en indice) par son potentiel initial :

$$\textsc{Init:}\ \frac{}{\alpha \triangleright \mathrm{C}[\mathit{init}(a, v)] \longrightarrow \alpha \triangleright \mathrm{C}[\mathit{nop}] \parallel \langle a \,|\, \varnothing \rangle_{\mathcal{P}(v\ a)} \triangleright v\ a}$$

Introduisons le calcul de ce potentiel dans le cadre de $\mathcal{F}unc_1$. Il s'agit d'une extension de celui de $\mathcal{F}unc_0$ pour tenir compte du filtrage. Maintenant une valeur peut donc également être une union de valeur :

$$V ::= \{\} \mid \{m, \ldots, m\} \mid \mathbb{M} \mid \lambda[p_i.e_i]^{i \in I} \mid \bigcup\{V, \ldots, V\}$$

$$\textsc{Potentiel}: \frac{\mathcal{P}(e) = V,\ E}{\mathcal{P}_e(e) = setof(V) \cup E}$$

$$\textsc{Val1}: \frac{v \in \mathbb{A} \cup \mathbb{C} \cup \mathcal{F}_c \cup \{nop,\ nil\}}{\mathcal{P}(v) = \{\},\ \{\}} \qquad \textsc{Val2}: \frac{V \in \mathcal{V}al}{\mathcal{P}(V) = V,\ \{\}} \qquad \textsc{Var}: \frac{}{\mathcal{P}(x) = \{\},\ \{\}}$$

$$\textsc{Mess}: \frac{\mathcal{P}(e_i) = V_i,\ E_i}{\mathcal{P}(m(\vec{e})) = \{\},\ \bigcup_i E_i} \qquad \textsc{Const}: \frac{C \in \{cons,\ tuple2, \ldots,\ tuplen\} \quad \mathcal{P}(e_i) = V_i,\ E_i}{\mathcal{P}(C(\vec{e})) = \bigcup_i V_i,\ \bigcup_i E_i}$$

$$\textsc{Int}: \frac{\vec{x}_i = \mathcal{FV}(p_i) \quad \mathcal{P}([\mathbb{M}/\vec{x}_i]e_i) = V_i,\ E_i}{\mathcal{P}(\lambda[m_i(p_i).e_i]^{i \in I}) = \bigcup_i \{m_i\},\ \bigcup_i E_i} \qquad \textsc{Bec}: \frac{\mathcal{P}(e) = V,\ E}{\mathcal{P}(become(a,\ e)) = \{\},\ setof(V) \cup E}$$

$$\textsc{App}: \frac{\mathcal{P}(e) = \bigcup_j V_j,\ E \quad \mathcal{P}(e') = V',\ E' \quad App(V_j, V') = V'_j,\ E_j}{\mathcal{P}(e\ e') = \bigcup_j V'_j,\ \bigcup_j E_j \cup E \cup E'}$$

$$\textsc{LRec1}: \frac{\mathcal{P}(e) = V,\ E \quad \mathcal{P}([V/x]e') = V',\ E'}{\mathcal{P}(\texttt{letrec}\ x = e\ \texttt{in}\ e') = V',\ E \cup E'}$$

$$\textsc{LRecn}: \frac{\mathcal{P}(\texttt{letrec}\ \vec{x}|_i = \vec{e}|_i\ \texttt{in}\ e_i) = V_i,\ E_i \quad \mathcal{P}([V_1/x_1] \ldots [V_n/x_n]e) = V,\ E}{\mathcal{P}(\texttt{letrec}\ \vec{x} = \vec{e}\ \texttt{in}\ e) = V,\ \bigcup_i E_i \cup E}$$

$$\textsc{App1}: \frac{\vec{x}_i = \mathcal{FV}(p_i) \quad \mathcal{P}([V/\vec{x}_i]e_i) = V_i,\ E_i}{App(\lambda[p_i.e_i]^{i \in I}, V) = \bigcup_i V_i,\ \bigcup_i E_i} \qquad \textsc{App2}: \frac{}{App(V, V') = \{\},\ \{\}}$$

Règles 6.1 Potentiel d'un terme de $\mathcal{F}unc_1$.

Le calcul suit les règles 6.1. Dans lesquelles la fonction *setof* est étendue et vaut donc :

$$setof(\{\}) = setof(\lambda[p_i.e_i]^{i\in I}) = \{\} \qquad setof(\mathbb{M}) = \mathbb{M}$$

$$setof(\{m_1,\ldots,m_n\}) = \{m_1,\ldots,m_n\} \qquad setof(\bigcup_i V_i) = \bigcup_i setof(V_i)$$

Les règles contiennent également une notion de substitution étendue aux séquences : $[V/\vec{x_i}]$ dans les règles INT et APP1. Elle signifie que chaque x_i est subsitué par V.

La valeur et l'effet des constructeurs et des fonctions comportant plusieurs cas sont approximés par l'union des approximations des sous-termes. Ainsi, par exemple la paire de message (m_1, m_2) est approximée par $\{m_1, m_2\}$ et l'application de $\lambda(a, b).a$ sur (m_1, m_2) est également approximée par $\{m_1, m_2\}$. La précision n'est donc pas très grande, mais cela n'a pas d'influence sur notre analyse puisqu'elle effectue une approximation beaucoup plus précise lors du typage.

Enfin, comme dans le chapitre 4, notre calcul effectue un dépliage de ces entités récursives lors de leur approximation. Il faut, par exemple, éviter de déplier indéfiniment les comportements ou fonctions qui bouclent. Nous avons donc adopté la stratégie suivante :

- lors de la définition récursive d'une seule variable, nous substituons la fonction introduite une seule fois par son corps (LREC1). Si la fonction se rappelle, son nom est maintenant libre et est donc approximé en utilisant la règle VAR ;
- lors de la définition récursive d'un groupe de fonctions mutuellement récursives, nous déplions toutes les fonctions une seule fois. Pour cela, nous remplaçons le nom d'une fonction par son corps dans lequel nous conservons la définition de toutes les autres fonctions.

Notons que cette stratégie d'approximation est valide (vis-à-vis du potentiel) puisque chaque comportement est déplié au moins une fois dans toute expression dans laquelle il figure. Autrement dit, toutes les interfaces qui seront potentiellement assumées sont effectivement prises en compte.

Une fois le potentiel défini, la sémantique de ML-ACT est complète, et il est possible d'analyser les sources d'erreurs d'exécution. Tout d'abord, notons qu'il y a deux familles d'erreurs. La première forme est classique et correspond aux erreurs classiques des langages fonctionnels : une valeur est manipulée dans un contexte erroné. Par exemple, le programme additionne une chaîne de caractères et un entier. La seconde forme d'erreurs l'est moins, il s'agit d'une mauvaise utilisation d'une interface de communication : un message d'étiquette m est envoyé à un acteur qui ne saura jamais traiter ce message, ou bien qui sait le traiter mais les arguments que le message contient ne sont pas corrects.

Remarquons immédiatement qu'avec la sémantique fournie, le fait d'envoyer un message avec un type erroné est détecté par une application fonctionnelle erronée et sera donc considéré comme une erreur fonctionnelle au niveau détection. Cependant, cette détection nécessite une extension des types fonctionnels classiques et impose des contraintes au niveau de la construction des types concurrents associés aux acteurs. Ainsi, une liaison est maintenu entre le type d'un acteur et le type de son interface. Par exemple, si un acteur a a pour type t_a, si son interface actuelle comprend les messages de type $m(\top)$ et s'il reçoit un message de type $m(int, int)$. Alors, nous verrons que $m(\top)$ est inclus (au

sens de la construction d'interface) dans t_a et $m(int, int)$ est aussi inclus (au sens de l'envoi de message) dans t_a. La seconde inclusion générée lors de l'envoi du message impose par transitivité la contrainte $m(int, int) \sqsubseteq m(\top)$ qui correspond à l'erreur du filtrage du message envoyé par celui du filtre de l'interface de a.

Dans le filtrage, la seule erreur possible est une différence d'arité entre deux constructeurs. En fait, cette erreur ne peut survenir que lors du filtrage d'un message par un motif de même étiquette mais de type différent. Dans la réduction fonctionnelle, les erreurs possibles sont :

- une variable n'est pas définie, cela permet également de détecter les utilisations illégales de *ego*, *self*, *suicide* et *become* ;
- une valeur qui n'est pas une fonction est appliquée ;
- une application de fonction définie par cas se termine par l'échec de tous les filtrages possibles ;
- dans une définition récursive, des valeurs non fonctionnelles sont définies

Dans la partie concurrente, les erreurs suivantes peuvent se produire :

- l'envoi d'une valeur qui n'est pas un message ;
- l'envoi d'un message à une valeur qui n'est pas une adresse ;
- la création d'un acteur en lui donnant une valeur qui n'est pas un comportement (*i.e.* une fonction qui prend en paramètre une adresse et renvoie une interface) ;
- un changement de comportement sur une valeur qui n'est pas une interface ;
- un message est reçu mais il ne figure pas dans le potentiel de son destinataire (c'est donc un orphelin trivial)

Parmi ces erreurs possibles, seule la dernière n'est pas usuellement détectée par les systèmes de type. Le but du système de type qui va maintenant être introduit est donc d'éliminer toutes ces erreurs potentielles en se basant sur les systèmes de types fonctionnels usuels. Cependant, la détection de la dernière erreur impose de procéder à une analyse fine des communications à travers des types suffisamment précis.

6.2 Les types

Dans cette section, nous définissons précisément les types et la relation de sous-typage utilisés par le système de type de ML-ACT. Ces types vont consistés en une extension de ceux du chapitre 4. Nous introduisons également les types des fonctions prédéfinies qui sont contenues dans l'environnement initial de typage.

Types

Avant de donner la définition précise des types, présentons les informellement. Les valeurs fonctionnelles usuelles sont approximées de manière classique par les types de base (T_B) pour les valeurs constantes, le produit cartésien de types ($T \times \cdots \times T$) pour les tuples, les types listes ($T\ list$) pour les listes et les types fonctionnels ($T \to T$) pour les fonctions. L'approximation des entités concurrentes (messages, interfaces et acteurs) suit les définitions du chapitre 4.

Nous pouvons donc maintenant introduire les types que nous utilisons sur $\mathcal{F}unc_1$. Ils suivent la définition générale 3.2 donnée page 59 et précisent l'ensemble des types de base, celui des constructeurs de type et la forme des types interfaces.

Définition 6.1 (Types) :
L'ensemble des ***types*** *est noté* $\mathbb{T}$*, sa définition est paramétrée par l'ensemble des variables de type (notées* t *et appartenant à* $\mathbb{V}_t$*), celui des variables de rangée (notées* i *et appartenant à* $\mathbb{V}_i$*) et enfin, celui des variables de présence (notée* p *et appartenant à* $\mathbb{V}_p$*). Un type est un terme construit par la grammaire suivante :*

$$
\begin{array}{lll}
T & ::= T_B \mid t \mid T \times \cdots \times T \mid T\ list \mid T \to T & \\
& \mid\ I & \textit{type de message} \\
& \mid\ \triangleright I & \textit{type d'interface} \\
& \mid\ @I & \textit{type d'acteur} \\
T_B & ::= \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{float} \mid \mathit{string} \mid \mathit{char} & \\
I & ::= \{\} & \textit{rangée vide} \\
& \mid\ i & \textit{variable de rangée} \\
& \mid\ \{m : P,\ I\} & \textit{rangée contenant } m \\
P & ::= p & \textit{variable de présence} \\
& \mid\ \dagger & \textit{message absent} \\
& \mid\ \bullet T & \textit{message installé} \\
& \mid\ \circ T & \textit{message reçu}
\end{array}
$$

Un type est ***clos*** *si et seulement si il ne contient pas de variable (de type).*

Relions plus précisément la notion de potentiel et la notion de type d'un acteur, en définissant la notion d'abstraction (Abs) d'un type clos :

$$Abs(@I) = Abs_I(I) \qquad Abs(\triangleright I) = Abs_I(I) \qquad Abs(T) = \{\} \text{ sinon}$$

en utilisant l'abstraction des rangées (Abs_I) :

$$Abs_I(\{\}) = \{\} \qquad Abs_I(\{m : \circ T,\ I\}) = Abs(I) \qquad Abs_I(\{m : \bullet T,\ I\}) = \{m\} \cup Abs(I)$$

L'abstraction est ensuite étendue au type contenant des variables en collectant toutes les inclusions dues aux contraintes et à la règle suivante :

$$I_1 \sqsubseteq I_2 \Longrightarrow Abs_I(I_1) \subseteq Abs_I(I_2)$$

On montrera dans le chapitre suivant que l'abstraction d'un type d'acteur est une approximation inférieure du potentiel initial d'un acteur.

Sous-typage

Le sous-typage suit également la définition du chapitre 2 en la spécialisant à ML-Act :

$$\frac{}{\bot \sqsubseteq t} \qquad \frac{}{t \sqsubseteq \top} \qquad \frac{t'_1 \sqsubseteq t_1 \quad t_2 \sqsubseteq t'_2}{t_1 \to t_2 \sqsubseteq t'_1 \to t'_2} \qquad \frac{\forall i \in 1..n\ t_i \sqsubseteq t'_i}{t_1 \times \cdots \times t_n \sqsubseteq t'_1 \times \cdots \times t'_n}$$

$$\frac{t \sqsubseteq t'}{t\ list \sqsubseteq t'\ list} \qquad \frac{t' \sqsubseteq t}{@t \sqsubseteq @t'} \qquad \frac{t \sqsubseteq t'}{\rhd t \sqsubseteq \rhd t'}$$

$$\frac{}{\{\} \sqsubseteq i} \qquad \frac{p_1 \sqsubseteq p_2 \qquad i_1 \sqsubseteq i_2}{\{m : p_1, i_1\} \sqsubseteq \{m : p_2, i_2\}}$$

$$\frac{t_1 \sqsubseteq t_2}{\circ t_1 \sqsubseteq \circ t_2} \qquad \frac{t_1 \sqsubseteq t_2}{\circ t_1 \sqsubseteq \bullet t_2} \qquad \frac{t_2 \sqsubseteq t_1}{\bullet t_1 \sqsubseteq \bullet t_2}$$

Règles 6.2 La relation de sous-typage pour $\mathcal{F}unc_1$.

Définition 6.2 (Sous-typage) :
*Un type t_1 est **sous-type** d'un type t_2 si et seulement si il est possible de dériver une preuve de $t_1 \sqsubseteq t_2$ en utilisant le système de règle 6.2.*

Dans le cadre de nos travaux sur ML-ACT, le sous-typage étend simplement celui de $\mathcal{F}unc_0$ en tenant compte des constructeurs et constantes.

Avant de présenter les règles de typage, introduisons l'environnement initial de typage.

Environnement initial

Il est aisé d'utiliser des fonctions définies dans OBJECTIVE CAML dans ML-ACT, il suffit que le type de la dite fonction figure dans l'environnement initial. Dans cette thèse, nous n'approfondirons pas le sujet de la construction d'un tel environnement initial qui est orthogonal à nos travaux. Nous supposons seulement qu'il contient un certain nombre de fonctions utiles. L'environnement initial ainsi crée est manipulé par le nom $\mathcal{E}_0$ dans les règles de typage et est accessible en tout point d'un programme.

L'environnement initial de typage contient certaines fonctions usuelles prédéfinies :

- des fonctions d'affichage : `print_int` de type $int \to unit$, `print_string` de type $string \to unit$, ...,
- des fonctions arithmétiques sur les entiers : `+`, `-`, `*`, `/`, `mod` de type $int \times int \to int$ et `-` de type $int \to int$,
- des fonctions arithmétiques sur les réels : `+.`, `-.`, `*.`, `/.` de type $float \times float \to float$ et `-.` de type $float \to float$,
- des fonctions de comparaisons : `=`, `<>`, `<`, `>`, `<=`, `>=` de type $\alpha \times \alpha \to bool$,
- des fonctions de calcul sur les booléens : `or`, `&` de type $bool \times bool \to bool$ et `not` de type $bool \to bool$,
- des fonctions de manipulation de données construites : `^` sur les chaînes de caractères de type $string \times string \to string$, `fst` et `snd` sur les paires de type respectif $\alpha \times \beta \to \alpha$ et $\alpha \times \beta \to \beta$ et enfin, sur les listes `::` de type $\alpha \times \alpha\ list \to \alpha\ list$, `@` de type $\alpha\ list \times \alpha\ list \to \alpha\ list$, `hd` de type $\alpha\ list \to \alpha$ et `tl` de type $\alpha\ list \to \alpha\ list$

Il contient également la définition des fonctions prédéfinies modélisant la partie acteur du programme :

- La création d'une adresse d'acteur (*new*) est une instruction qui renvoie une nouvelle adresse sur laquelle aucune contrainte n'existe encore. Soit :

$$\boxed{new : @\psi}$$

- La création d'un acteur (*init*) crée l'acteur à partir du couple d'arguments qu'elle reçoit : l'adresse de l'acteur à créer et son comportement initial. Rappelons qu'un comportement est une fonction qui prend en paramètre l'adresse de l'acteur qui l'exécutera et renvoie une interface (dans lequel l'*ego* est lié) et qu'une interface est abstraite par un type interface qui comprend un ensemble de messages. Son type est donc un type fonctionnel dont l'argument est un couple et le résultat est *unit*. Le second argument le comportement doit être une fonction sous-type de $@\psi \to \triangleright\psi$ car son *ego* doit correspondre au type de l'adresse reçue en premier argument (sous-type de type $@\psi$) et qu'il faut collecter dans ce type l'apport de l'interface de ce comportement initial. Le type de *init* est donc :

$$\boxed{init : @\psi \times (@\psi \to \triangleright\psi) \to unit}$$

  Ainsi, l'application $init(a, \lambda ego.i)$, où a est de type $@t_a$, *ego* de type $@t_{ego}$ et i de type $\triangleright t_i$, génère les contraintes $@t_a \sqsubseteq @\psi$ et $@t_{ego} \to \triangleright t_i \sqsubseteq @\psi \to \triangleright\psi$. Et ces contraintes se simplifient en : $\psi \sqsubseteq t_a$, $t_{ego} \sqsubseteq \psi$ et $t_i \sqsubseteq \psi$, ce qui permet de cumuler dans t_a les contenus de t_{ego} et t_i.
- Le changement de comportement (*become*) est voisin de la création. La fonction *become* crée un acteur anonyme qui poursuit le calcul à partir de l'adresse de l'acteur courant et de sa future interface, puis change l'interface de l'acteur courant. Son type est donc proche de celui de *init*, une fonction qui prend en argument un couple et ne renvoie aucun résultat :

$$\boxed{become : @\psi \times \triangleright\psi \to unit}$$

  Ainsi, lors de l'appel de $become(a, i)$, où a est de type $@t_a$ et i de type $\triangleright t_i$, génère les contraintes $@t_a \sqsubseteq @\psi$ et $\triangleright t_i \sqsubseteq \triangleright\psi$. Ce qui conduit à $t_i \sqsubseteq \psi \sqsubseteq t_a$, le contenu du type du second argument est donc collecté dans celui du second. Une des raisons imposant au changement de comportement (*become*) de prendre en paramètre l'adresse courante de l'acteur provient de la nécessité de relier le type de la future interface de l'acteur au type de l'acteur lui-même. En effet, comme le type de l'acteur approxime toutes ses interfaces, il faut que le type de l'acteur contienne les futures interfaces. Il aurait également été possible de dédier une règle de typage à *become* dans laquelle le type associé à *ego* aurait été récupéré dans l'environnement de typage afin de générer la contrainte voulue. C'est cette seconde approche qui est suivie dans [DPCS00]. Nous avons souhaité dans le cadre de ce manuscrit garder une forme de fonction prédéfinie pour les primitives acteurs et de les typer ainsi de manière uniforme.

– La procédure d'envoi de message prend en argument le couple du message et de l'acteur destination. Soit :

$$\boxed{\mathit{send} : \psi \times @\psi \to \mathit{unit}}$$

Ainsi, l'appel de $\mathit{send}(m(), a)$, où a est de type $@t_a$, génère les contraintes $\{m : \circ \mathit{unit}\} \sqsubseteq \psi$ et $@t_a \sqsubseteq @\psi$. Ce qui conduit à $\{m : \circ \mathit{unit}\} \sqsubseteq \psi \sqsubseteq t_a$. Le message m est donc collecté dans le type de a.

Et finalement, trois fonctions sont utilisées pour calculer le type des valeurs :
– *Type* pour calculer le type des constantes,
– TypeC_p qui calcule le type d'un constructeur apparaissant dans un motif et
– TypeC_e qui calcule le type d'un constructeur apparaissant dans une expression.

En plus, des constantes usuelles déjà introduites, la fonction *Type* est utilisée pour typer la valeur d'échec du filtrage : **Fail**. En effet, comme nous l'avons signalé, nous ne nous intéressons pas à la complétude du filtrage et donc comme la preuve de correction de notre système de type reposera sur le fait que l'erreur n'est pas typable, il nous faut donner un type à cet échec : le type indéfini ($\bot$).

Le type d'un constructeur dépend de son mode d'utilisation dans le cas des constructeurs de messages. Ceux-ci vont, en effet, avoir des types différents puisqu'ils vont indiquer, dans un cas, un message qui est potentiellement envoyé (si c'est une expression) et dans l'autre cas, l'installation (si c'est un motif). Dans la définition des fonctions TypeC_p et TypeC_e, nous utilisons une notation sans indice pour indiquer une définition commune aux deux fonctions. De plus, notons que ces fonctions renvoient également un ensemble de contraintes car le constructeur de liste impose une contrainte entre les types de ses deux arguments.

$$\begin{array}{lcl}
\mathit{TypeC}(\mathit{nop}) & = & \Lambda().(\mathit{unit},\ \varnothing) \\
\mathit{TypeC}(\mathit{nil}) & = & \Lambda().(\alpha\ \mathit{list},\ \varnothing) \\
\mathit{TypeC}_p(\mathit{cons}) & = & \Lambda(t_1, t_2).(t_1\ \mathit{list},\ \{t_2 = t_1\ \mathit{list}\}) \\
\mathit{TypeC}_e(\mathit{cons}) & = & \Lambda(t_1, t_2).(\alpha\ \mathit{list},\ \{t_1 \sqsubseteq \alpha,\ t_2 \sqsubseteq \alpha\ \mathit{list}\}) \\
\mathit{TypeC}(\mathit{tuple2}) & = & \Lambda(t_1, t_2).(t_1 \times t_2,\ \varnothing) \\
\vdots & = & \vdots \\
\mathit{TypeC}(\mathit{tuplen}) & = & \Lambda(t_1, \ldots, t_n).(t_1 \times \cdots \times t_n,\ \varnothing) \\
\mathit{TypeC}_p(m) & = & \Lambda(t).(\{m\colon\ \bullet\, t\},\ \varnothing) \\
\mathit{TypeC}_e(m) & = & \Lambda(t).(\{m\colon\ \circ\, t\},\ \varnothing)
\end{array}$$

Un motif correspond à une définition, ses variables sont donc de nouvelles variables introduites pour une utilisation ultérieure. Le typage motif d'un constructeur consiste alors en l'introduction de ses variables dans l'environnement de typage en leur associant une variable de type (non encore utilisée). Dans une expression, les constructeurs servent effectivement à construire des objets. Ils vont donc introduire des contraintes (en générale des bornes supérieures) sur les objets qu'ils utilisent. Cette différence sémantique explique que la construction de liste ne soit pas typée de la même manière dans un motif et dans une expression. Nous avons choisi dans ML-ACT de donner aux listes un type *à la ML*, ceci crée une perte d'information sur la structure de la liste et sur ses éléments. Cependant,

les types gagnent en simplicité et en lisibilité par rapport à un type liste comportant deux constructeurs de type, un pour la liste vide (*nil*) et un pour une liste composée d'une tête et d'une queue (*cons*). La perte d'information provient du fait que le type d'une liste correspond à la borne supérieure de tous ses éléments. Ainsi, dans une expression le type d'une liste est α *list* et sa construction génère les contraintes $\{t_1 \sqsubseteq \alpha,\ t_2 \sqsubseteq \alpha\ list\}$. Dans le cas d'une construction de motif liste, on cherche à déstructurer une liste existante, il faut donc que chaque variable du filtre référence le type commun de la liste. Une contrainte d'égalité est donc générée. Notons que dans un motif, seuls des types construits à partir des types de base et des variables peuvent apparaître. Ainsi, imposer une égalité entre ces types consiste à leur imposer d'avoir une structure commune (ce qu'aurait également imposé une inégalité) mais également contraint toutes les variables feuilles de cette structure à être identiques. Toutes les contraintes qui sont dues à l'utilisation des éléments extraits de la liste portent donc sur le type commun des éléments de la liste. Par exemple, le filtre $cons(x_1,\ cons(x_2,\ x_3))$ a pour type t *list* et les trois variables figurant dans le filtre sont ajoutées dans l'environnement avec le type t pour x_1 et x_2 et le type t *list* pour x_3. Donc, si dans la portée de ce filtre se trouve les deux envois de messages suivants : $send(m_1(), x_1);\ send(m_2(), x_2)$, on obtient un système de contraintes $(\{t \sqsubseteq @\{m_1 : \circ unit\},\ t \sqsubseteq @\{m_2 : \circ unit\}\})$ qui impose à tous les éléments de la liste d'être capable de recevoir les messages m_1 et m_2.

Montrons maintenant deux lemmes sur la variance des constructeurs de types qui sont nécessaires pour la preuve de la correction du système de type.

Lemme 6.1 (Covariance des constructeurs de types) :
Quelque soit le constructeur C, le constructeur de type (d'expression) associé $TypeC_e(C)$ est covariant vis-à-vis de tous ses paramètres. Soit, si on note $TypeC_e(C)(t_1, \ldots, t_n) = \alpha_i,\ \mathcal{C}$ et $TypeC_e(C)(t_1, \ldots, t_{i-1}, t'_i, t_{i+1}, \ldots, t_n) = \alpha'_i,\ \mathcal{C}'$:

$$\forall i \in 1..n \quad t_i \sqsubseteq t'_i \Longrightarrow \alpha_i \sqsubseteq \alpha'_i \wedge \mathcal{C}' \Rightarrow \mathcal{C}$$

où $\mathcal{C}' \Rightarrow \mathcal{C}$ signifie que si $\mathcal{C}'$ possède une solution alors $\mathcal{C}$ en possède une également.

PREUVE : La preuve est immédiate pour les constructeurs sans arguments ainsi que pour les constructeurs de tuples (ils sont covariants). Il ne reste donc plus qu'à montrer que le lemme est aussi vrai pour les listes et la construction de messages.

- Soit t_1, t_2 et t'_1 trois types tels que t_1 soit un sous-type de t'_1 (*i.e.* $t_1 \sqsubseteq t'_1$). Alors, on a $TypeC_e(cons)(t'_1, t_2) = (\alpha\ list,\ \{t'_1 \sqsubseteq \alpha,\ t_2 \sqsubseteq \alpha\ list\})$, ce qui implique que $t_1 \sqsubseteq t'_1 \sqsubseteq \alpha$. On peut donc attribuer à $TypeC_e(cons)(t_1, t_2)$ le type α *list* et les contraintes associées à $TypeC_e(cons)(t'_1, t_2)$ implique alors celle de $TypeC_e(cons)(t_1, t_2)$. La démonstration est identique sur le deuxième argument.
- Dans le cas d'un message, i ne peut valoir que 1, α_1 vaut $\{m\colon \circ\, t\}$ et α'_1 vaut $\{m\colon \circ\, t'\}$. Comme t est sous-type de t', on a $\alpha_1 \sqsubseteq \alpha'_1$ et les contraintes étant vides, elles s'impliquent $(\{\} \Rightarrow \{\})$.

Le deuxième lemme est une forme de réciproque du précédent, il est indispensable dans la preuve d'un lemme sur le filtrage dans le cadre de la preuve de correction.

Lemme 6.2 (Sous-typage des constructeurs de types) :

Quelque soit le constructeur C, si un type construit (d'expression) $TypeC_e(C)(t_1, \ldots, t_n)$ est sous-type d'un type construit (de motif) $TypeC_p(C)(t'_1, \ldots, t'_n)$, tous les sous-types sont également en relation de sous-typage. Soit, si on a $TypeC_e(C)(t_1, \ldots, t_n) = \alpha$, $\mathcal{C}$ et $TypeC_p(C)(t'_1, \ldots, t'_n) = \beta$, $\mathcal{C}'$ alors :

$$\alpha \sqsubseteq \beta \Longrightarrow \forall i \in 1..n \quad t_i \sqsubseteq t'_i$$

Preuve : Sur le même principe que pour la preuve précédente, le lemme est trivial pour les constructeurs sans arguments et pour les tuples. Il ne reste donc qu'à démontrer sa validité pour les listes et les construction de messages.

- Si le constructeur construit une liste, on a :

$$\begin{cases} TypeC_e(cons)(t_1, t_2) = (\alpha\ list,\ \{t_1 \sqsubseteq \alpha,\ t_2 \sqsubseteq \alpha\ list\}) \\ TypeC_p(cons)(t'_1, t'_2) = (t'_1\ list,\ \{t'_2 = t'_1\ list\}) \end{cases}$$

  Ce qui conduit à $t_1 \sqsubseteq \alpha \sqsubseteq t'_1$ et $t_2 \sqsubseteq \alpha\ list \sqsubseteq t'_1\ list = t'_2$.
- Dans le cas d'un message, on a :

$$\begin{cases} TypeC_e(m)(t) = (\{m\colon \circ\, t\},\ \varnothing) \\ TypeC_p(m)(t') = (\{m\colon \bullet\, t'\},\ \varnothing) \end{cases}$$

  Et, donc par définition du sous-typage $\{m\colon \circ\, t\} \sqsubseteq \{m\colon \bullet\, t'\} \Longleftrightarrow t \sqsubseteq t'$.

Remarques :

- *Il n'y a aucun lien particulier entre $\mathcal{C}$ et $\mathcal{C}'$ si ce n'est qu'ils possèdent tous les deux des solutions.*
- *Ce lemme ne concerne que le typage que les constructeurs, α et β ne peuvent donc être que de l'une des formes suivantes : unit, T list, $T_1 \times \cdots \times T_n$, $\{m\colon \circ\, T\}$ ou $\{m\colon \bullet\, T\}$.*

Après avoir défini tous les outils nécessaires, nous pouvons enfin présenter le système de type de ML-Act.

6.3 Le système de type

Dans cette section, nous allons introduire les règles de typage ainsi que des règles non nécessaires mais qui facilitent la construction d'arbre de dérivation du typage. Ces règles appelées métarègles sont en fait des compositions de règles de typage et permettent ainsi de diminuer la taille de l'arbre de typage et d'améliorer le système.

Les motifs

Tout d'abord, introduisons les règles 6.3 qui décrivent le typage des motifs. Ce typage est usuel et immédiat, une prémisse ou une conclusion ne contient aucune hypothèse et calcule le type du motif et un environnement. Celui-ci contient l'ensemble des variables introduites par le motif ainsi que leur type. Notons que ces règles peuvent ajouter des

$$\textsc{Constr}:\ \frac{\forall i \in 1..n \quad \vdash_p p_i\colon t_i,\ \mathcal{E}_i \quad \mathit{TypeC}_p(C)(t_1,\ldots,t_n) = t,\ \mathcal{C}}{\vdash_p C(p_1,\ldots,p_n)\colon t,\ \mathcal{E}_1 \oplus \cdots \oplus \mathcal{E}_n}\ (\mathcal{C})$$

$$\textsc{Any}:\ \frac{}{\vdash_p \_\colon t,\ \{\}} \qquad \textsc{Var}:\ \frac{}{\vdash_p x\colon t,\ \{x : t\}} \qquad \textsc{Cst}:\ \frac{}{\vdash_p c\colon \mathit{Type}(c),\ \{\}}$$

Règles 6.3 Le typage des motifs de $\mathcal{F}unc_1$.

contraintes dans l'ensemble global des contraintes. La forme d'une règle de typage d'un motif p est donc : $\vdash_p p\colon t,\ \mathcal{E}$ où t est le type de p et $\mathcal{E}$ est l'environnement résultat.

Remarquons que comme la sommation d'environnements $\oplus$ apparaît en prémisse de la règle Constr, celle-ci n'est applicable que si cette somme existe (voir la définition 3.5 page 61). On vérifie ainsi que chaque variable n'apparaît qu'une seule fois dans un motif. C'est également la seule règle qui ajoute des contraintes (lors du filtrage d'une liste pour relier le type de la tête au type de la queue). Le type renvoyé est alors celui qui est calculé par la fonction *TypeC*. Le typage du motif jocker (règle Any) et celui des variables (règle Var) sont presque identiques et renvoient une nouvelle variable de type t, la différence est l'environnement résultat qui dans le cas du jocker est vide et dans celui d'une variable x contient uniquement la liaison de x au type t. Enfin, les constantes (règle Cst) sont typées grâce à la fonction *Type* et produisent un environnement vide.

Les expressions

Les règles de typage des expressions sont pour la plupart issues du λ-calcul avec appel par valeur et introduction de définitions récursives par les `letrec`. Leur forme est : $\mathcal{E} \vdash_e e\colon t$ où $\mathcal{E}$ est l'environnement local de typage de l'expression e et t est son type. Elles consistent en l'extension du système de type du chapitre 4 en ajoutant des constantes, des constructeurs et le filtrage.

Pour typer un constructeur (règle Constr), il faut d'abord typer ses arguments, puis calculer le type résultant ainsi que les éventuelles contraintes qui relient les types des paramètres par le biais de la fonction TypeC_e. Le typage des constantes (règle Cst) utilise simplement la fonction *Type*. L'application (règle App) est typée de manière curryfiée, le type de la fonction doit être sous-type d'un type fonctionnel dont l'argument est le type du paramètre réel de l'appel et dont le résultat est une variable de type fraiche. Le typage d'une abstraction suit deux règles selon qu'il s'agisse d'une interface (règle Inter) ou d'une fonction (règle Fun). Une interface est approximée par un type interface contenant tous les messages installés définis dans l'abstraction. Remarquons que l'ajout systématique d'un *become* en fin de code des réactions impose aux e_i d'avoir toujours *unit* comme type. Le type d'une fonction est supérieur au type de chaque branche de l'abstraction. Il peut donc être considéré comme l'union des types de chaque branche. Cette abstraction est très grossière puisqu'il est possible de conserver des informations reliant plus précisément un résultat au type des paramètres permettant ainsi de typer de manière bien plus fine le filtrage. On obtient alors une forme restreinte de type dépendant. Ce typage s'effectue alors à l'aide d'un type dit *conditionnel* qui simule le filtrage dans le langage

$$\textsc{Constr}: \frac{\forall i \in 1..n \quad \mathcal{E} \vdash_e e_i : t_i \quad \mathit{TypeC}_e(C)(t_1, \ldots, t_n) = t, \boldsymbol{\mathcal{C}}}{\mathcal{E} \vdash_e C(e_1, \ldots, e_n) : t} \Big(\boldsymbol{\mathcal{C}}\Big)$$

$$\textsc{Cst}: \frac{}{\mathcal{E} \vdash_e c : \mathit{Type}(c)} \qquad \textsc{App}: \frac{\mathcal{E} \vdash_e f : t_f \quad \mathcal{E} \vdash_e e : t_e}{\mathcal{E} \vdash_e f\ e : t} \Big(t_f \sqsubseteq t_e \to t\Big)$$

$$\textsc{Inter}: \frac{\forall i \in I \quad \vdash_p m_i(p_i) : t_i,\ \mathcal{E}_i \quad \mathcal{E}{+}\mathcal{E}_i \vdash_e e_i : t'_i}{\mathcal{E} \vdash_e \lambda[m_i(p_i).e_i]^{i \in I} : \rhd t} \Big(\forall i \in I \ \ t_i \sqsubseteq t\Big)$$

$$\textsc{Fun}: \frac{\forall i \in I \quad \vdash_p p_i : t_i,\ \mathcal{E}_i \quad \mathcal{E}{+}\mathcal{E}_i \vdash_e e_i : t'_i}{\mathcal{E} \vdash_e \lambda[p_i.e_i]^{i \in I} : t} \Big(\forall i \in I \ \ t_i \to t'_i \sqsubseteq t\Big) \qquad \textsc{Id}: \frac{x \in \mathit{dom}(\mathcal{E}_0{+}\mathcal{E})}{\mathcal{E} \vdash_e x : (\mathcal{E}_0{+}\mathcal{E})(x)}$$

$$\textsc{Lrec}: \frac{\vdash_p \vec{x} : t_1 \times \cdots \times t_n,\ \mathcal{E}_p \quad \mathcal{E}{+}\mathcal{E}_p \vdash_e e_i : t'_i \quad \mathcal{E}{+}\mathcal{E}_p \vdash_e e' : t}{\mathcal{E} \vdash_e \texttt{letrec}\ \vec{x} = \vec{e}\ \texttt{in}\ e' : t} \Big(t'_i \sqsubseteq t_i\Big)$$

Règles 6.4 Le typage des expressions de $\mathcal{F}unc_1$.

des types. Ce type conditionnel a été introduit par A. AIKEN et al. dans [AWL94]. Il est surtout intéressant dans le cadre des langages qui autorisent les filtres hétérogènes et présente donc peu d'intérêt pour ML-ACT. En revanche sa puissance sera indispensable au typage de ERLANG. Dans le cadre du système de type de $\mathcal{F}unc_1$, nous utilisons la notion d'identificateur pour regrouper les variables, les adresses d'acteurs et les fonctions prédéfinies. Pour typer les identificateurs (règle ID), nous utilisons le type figurant dans l'environnement ou bien celui figurant dans l'environnement initial si le programmeur n'a pas défini la variable en question. Lors d'une définition récursive (règle LREC), les n variables du motif sont typées puis les n corps e_i sont typés dans un environnement étendu par le résultat des définitions. Enfin, la suite du programme est typée dans cet environnement étendu pour donner le type du `letrec`. Au cours de ce typage, une contrainte de sous-typage entre le type du corps et le type du filtre est générée. Notons que nous ne nous intéressons pas au problème de la détection des définitions récursives faisant boucler le programme. Nous faisons le choix d'ignorer ce problème car il est orthogonal à notre système de type et facile à résoudre[44]. Dans la pratique du compilateur, chaque corps d'une définition récursive doit être une fonction ou bien une interface.

Quelques métarègles

Les exemples présentés par la suite utilisent les métarègles 6.5 pour simplifier les arbres de déduction. Ces règles peuvent être déduites du système de type précédent et ne sont donc pas indispensables. Cependant, en simplifiant l'arbre de typage, elles facilitent sa construction et sa présentation. La démonstration de ces règles est une application immédiate des règles du système de type, elle est laissée au lecteur. Nous pensons que celles-ci

44. Il ne s'agit pas, ici, de détecter les fonctions récursives qui bouclent peut-être mais de détecter des constructions qui bouclent toujours (par exemple, `letrec` $x = x + 1$ `in x`).

$$\textsc{Let}: \frac{\vdash_p p\colon t_p,\ \mathcal{E}_1 \quad \mathcal{E} \vdash_e e_1\colon t_1 \quad \mathcal{E}{+}\mathcal{E}_1 \vdash_e e_2\colon t_2}{\mathcal{E} \vdash_e \texttt{let}\ p = e_1\ \texttt{in}\ e_2\colon t_2} \left(t_1 \sqsubseteq t_p\right)$$

$$\textsc{Seq}: \frac{\mathcal{E} \vdash_e e_1\colon t_1 \quad \mathcal{E} \vdash_e e_2\colon t_2}{\mathcal{E} \vdash_e e_1;\ e_2\colon t_2} \left(t_1 \sqsubseteq \mathit{unit}\right)$$

$$\textsc{Appn}: \frac{\mathcal{E} \vdash_e f\colon t_1 \to \cdots \to t_n \to t \quad \forall i \in 1..n \quad \mathcal{E} \vdash_e e_i\colon \alpha_i}{\mathcal{E} \vdash_e f\ e_1\ \ldots\ e_n\colon t} \left(\forall i \in 1..n\ \ \alpha_i \sqsubseteq t_i\right)$$

$$\textsc{Funn}: \frac{\forall i \in 1..n \quad \vdash_p p_i\colon t_i, \mathcal{E}_i \quad \mathcal{E}{+}\mathcal{E}_1{+}\cdots{+}\mathcal{E}_n \vdash_e e\colon t}{\mathcal{E} \vdash_e \lambda p_1 \ldots p_n.e\colon t_1 \to \cdots \to t_n \to t}$$

$$\textsc{Send}: \frac{\mathcal{E} \vdash_e e_1\colon t_1 \quad \mathcal{E} \vdash_e e_2\colon @t_2}{\mathcal{E} \vdash_e \mathit{send}(e_1, e_2)\colon \mathit{unit}} \left(t_1 \sqsubseteq t_2\right) \qquad \textsc{Bec}: \frac{\mathcal{E} \vdash_e e_1\colon @t_1 \quad \mathcal{E} \vdash_e e_2\colon \rhd t_2}{\mathcal{E} \vdash_e \mathit{become}(e_1, e_2)\colon \mathit{unit}} \left(t_2 \sqsubseteq t_1\right)$$

$$\textsc{New}: \frac{}{\mathcal{E} \vdash_e \mathit{new}\colon @t} \qquad \textsc{Init}: \frac{\mathcal{E} \vdash_e e_1\colon @t_1 \quad \mathcal{E} \vdash_e e_2\colon @t_2 \to \rhd t_3}{\mathcal{E} \vdash_e \mathit{init}(e_1, e_2)\colon \mathit{unit}} \begin{pmatrix} t_2 \sqsubseteq t_1 \\ t_3 \sqsubseteq t_1 \end{pmatrix}$$

Règles 6.5 Quelques métarègles de typage pour $\mathcal{F}unc_1$.

permettent également d'avoir une meilleure intuition du fonctionnment du système de type.

Les deux premières règles décrivent le typage du sucre syntaxique ajouté à $\mathcal{F}unc_1$: la définition (règle LET) et la séquence (règle SEQ). Dans une définition, le type de la fonction résultant d'une abstraction à plusieurs arguments est construit directement (règle FUNN). Le typage de fonctions non curryfiées est possible lorsque le type de la fonction est connu lors de l'application (règle APPN). De plus :

- si la fonction est prédéfinie (c'est-à-dire dans $\mathcal{E}_0$), nous ne faisons pas apparaître le prémisse du typage de la fonction ;
- si son résultat est de type constant (*i.e.* *bool*, *unit*, *int*, *float* ou *string*) nous renvoyons directement ce type comme type résultat

Par exemple, les quatre dernières métarègles typent directement l'envoi de message par la règle SEND, le changement de comportement par la règle BEC, la création d'une nouvelle adresse par la règle NEW et enfin la création d'un acteur par la règle INIT. Remarquons que ces règles sont très proches de celle du chapitre 4.

Pour mieux appréhender ces différentes règles, analysons précisément la construction de la preuve du typage d'un exemple.

6.4 Un exemple de typage

Le premier exemple que nous allons typer est le programme P' dont le code est :

```
let f a b = send a to b;; f (m1 1) act;; f (m2 1) act;;
```

Nous supposons qu'un acteur de nom `act` (et de type $@\psi_{act}$) a été précédemment défini et figure donc dans l'environnement $\mathcal{E}$. Ce programme est traduit dans la syntaxe interne par :

$$
\begin{aligned}
[\![P]\!] &= \texttt{let } [\![\texttt{f a b = send a to b}]\!]_D \texttt{ in } [\![\texttt{f } (m_1\ 1)\ \texttt{act};;\texttt{f } (m_2\ 1)\ \texttt{act};;]\!] \\
&= \texttt{let f} = [\![\texttt{a b}]\!]_F\ [\![\texttt{send a to b}]\!]_E \texttt{ in } [\![\texttt{f } (m_1\ 1)\ \texttt{act}]\!]_E ; [\![\texttt{f } (m_2\ 1)\ \texttt{act};;]\!] \\
&= \texttt{let f} = \lambda [\![\texttt{a}]\!]_P [\![\texttt{b}]\!]_P.(\mathit{send}([\![\texttt{a}]\!]_E, [\![\texttt{b}]\!]_E)) \texttt{ in } [\![\texttt{f } (m_1\ 1)]\!]_E\ [\![\texttt{act}]\!]_E ; [\![\texttt{f } (m_2\ 1)\ \texttt{act}]\!]_E \\
&= \texttt{let f} = \lambda\texttt{ab}.(\mathit{send}(\texttt{a},\texttt{b})) \texttt{ in } [\![\texttt{f}]\!]_E\ [\![(m_1\ 1)]\!]_E\ \texttt{act} ; [\![\texttt{f } (m_2\ 1)]\!]_E\ [\![\texttt{act}]\!]_E \\
&= \texttt{let f} = \lambda\texttt{ab}.(\mathit{send}(\texttt{a},\texttt{b})) \texttt{ in f } [\![m_1]\!]_E([\![1]\!]_E)\ \texttt{act} ; [\![\texttt{f}]\!]_E\ [\![(m_2\ 1)]\!]_E\ [\![\texttt{act}]\!]_E \\
&= \texttt{let f} = \lambda\texttt{ab}.(\mathit{send}(\texttt{a},\texttt{b})) \texttt{ in f } m_1(1)\ \texttt{act} ; \texttt{f } [\![m_2]\!]_E([\![1]\!]_E)\ \texttt{act} \\
&= \texttt{let f} = \lambda\texttt{ab}.(\mathit{send}(\texttt{a},\texttt{b})) \texttt{ in f } m_1(1)\ \texttt{act} ; \texttt{f } m_2(1)\ \texttt{act}
\end{aligned}
$$

La dérivation suivante permet d'inférer les types des entités que contient P' :

$$
\mathrm{F} : \frac{\dfrac{}{\vdash_p \texttt{a}\colon t_a, \mathcal{E}_a} \quad \dfrac{}{\vdash_p \texttt{b}\colon t_b, \mathcal{E}_b} \quad \dfrac{\mathcal{E}{+}\mathcal{E}_a{+}\mathcal{E}_b \vdash_e \texttt{a}\colon t_a \quad \mathcal{E}{+}\mathcal{E}_a{+}\mathcal{E}_b \vdash_e \texttt{b}\colon t_b}{\mathcal{E}{+}\mathcal{E}_a{+}\mathcal{E}_b \vdash_e \mathit{send}(\texttt{a},\texttt{b})\colon \mathit{unit}}\,(1)}{\mathcal{E} \vdash_e \lambda\texttt{ab}.(\mathit{send}(\texttt{a},\texttt{b}))\colon \alpha_1 \to \alpha_2 \to \beta}\,(2)
$$

$$
\mathrm{E1} : \frac{\dfrac{\mathcal{E}_t \vdash_e \texttt{f}\colon t_f \quad \mathcal{E}_t \vdash_e (m_1\ 1)\colon \{m_1\colon \circ \mathit{int}\}}{\mathcal{E}_t \vdash_e \texttt{f } (m_1\ 1)\colon \beta_1}\,(3) \quad \mathcal{E}_t \vdash_e \texttt{act}\colon @\psi_{act}}{\mathcal{E}_t \vdash_e \texttt{f } (m_1\ 1)\ \texttt{act}\colon \beta_2}\,(4)
$$

$$
\mathrm{E2} : \frac{\dfrac{\mathcal{E}_t \vdash_e \texttt{f}\colon t_f \quad \mathcal{E}_t \vdash_e (m_2\ 1)\colon \{m_1\colon \circ \mathit{int}\}}{\mathcal{E}_t \vdash_e \texttt{f } (m_2\ 1)\colon \gamma_1}\,(5) \quad \mathcal{E}_t \vdash_e \texttt{act}\colon @\psi_{act}}{\mathcal{E}_t \vdash_e \texttt{f } (m_2\ 1)\ \texttt{act}\colon \gamma_2}\,(6)
$$

$$
\frac{\dfrac{}{\vdash_p \texttt{f}\colon t_f, \mathcal{E}_f} \quad \mathrm{F} \quad \dfrac{\mathrm{E1} \qquad \mathrm{E2}}{\mathcal{E}_t \vdash_e \texttt{f } (m_1\ 1)\ \texttt{act};\texttt{f } (m_2\ 1)\ \texttt{act}\colon t'}\,(7)}{\mathcal{E} \vdash_e \texttt{let f} = \lambda\texttt{ab}.(\mathit{send}(\texttt{a},\texttt{b})) \texttt{ in f } (m_1\ 1)\ \texttt{act};\texttt{f } (m_2\ 1)\ \texttt{act}\colon t}\,(8)
$$

avec $\mathcal{E}_f = \{f\colon t_f\}$, $\mathcal{E}_a = \{a\colon t_a\}$, $\mathcal{E}_b = \{b\colon t_b\}$ et $\mathcal{E}_t = \mathcal{E}{+}\mathcal{E}_f$ et les contraintes :

$$
\begin{cases}
1 : \quad t_b \sqsubseteq @t_a & & 5 : \quad t_f \sqsubseteq \{m_2\colon \circ \mathit{int}\} \to \gamma_1 \\
2 : \quad \alpha_1 \sqsubseteq t_a \qquad \alpha_2 \sqsubseteq t_b \qquad \mathit{unit} \sqsubseteq \beta & & 6 : \quad \gamma_1 \sqsubseteq @\psi_{act} \to \gamma_2 \\
3 : \quad t_f \sqsubseteq \{m_1\colon \circ \mathit{int}\} \to \beta_1 & & 7 : \quad \beta_2 \sqsubseteq \mathit{unit} \qquad \gamma_2 \sqsubseteq t' \\
4 : \quad \beta_1 \sqsubseteq @\psi_{act} \to \beta_2 & & 8 : \quad \alpha_1 \to \alpha_2 \to \beta \sqsubseteq t_f \qquad t' \sqsubseteq t
\end{cases}
$$

La résolution du système de contraintes obtenu suit alors les étapes suivantes :

– on applique la transitivité du sous-typage, les valeurs comparables à *unit* sont remplacées par *unit* et certaines variables inutiles sont oubliées (t_b et t_f) :

$$
\begin{cases}
\alpha_1 \sqsubseteq t_a \qquad \alpha_2 \sqsubseteq @t_a & \\
\alpha_1 \to \alpha_2 \to \mathit{unit} \sqsubseteq \{m_1\colon \circ \mathit{int}\} \to \beta_1 & \beta_1 \sqsubseteq @\psi_{act} \to \mathit{unit} \\
\alpha_1 \to \alpha_2 \to \mathit{unit} \sqsubseteq \{m_2\colon \circ \mathit{int}\} \to \gamma_1 & \gamma_1 \sqsubseteq @\psi_{act} \to \gamma_2 \qquad \gamma_2 \sqsubseteq t
\end{cases}
$$

– les types fonctionnels sont alors décomposés :

$$\begin{cases} \alpha_2 \sqsubseteq @t_a \\ \{m_1 : \circ int\} \sqsubseteq \alpha_1 \sqsubseteq t_a \qquad \alpha_2 \to unit \sqsubseteq \beta_1 \sqsubseteq @\psi_{act} \to unit \\ \{m_2 : \circ int\} \sqsubseteq \alpha_1 \sqsubseteq t_a \qquad \alpha_2 \to unit \sqsubseteq \gamma_1 \sqsubseteq @\psi_{act} \to \gamma_2 \qquad \gamma_2 \sqsubseteq t \end{cases}$$

– on réapplique la transitivité, certaines variables inutiles sont supprimées (α_1, β_1 et γ_1) et les types fonctionnels sont décomposés :

$$\{m_1 : \circ int\} \sqsubseteq t_a \quad \{m_2 : \circ int\} \sqsubseteq t_a \quad @\psi_{act} \sqsubseteq \alpha_2 \sqsubseteq @t_a \quad unit \sqsubseteq unit \quad unit \sqsubseteq \gamma_2 \sqsubseteq t$$

– les variables α_2 et γ_2 devenues inutiles sont alors oubliées et on applique le sous-typage des acteurs :

$$\{m_1 : \circ int\} \sqsubseteq t_a \sqsubseteq \psi_{act} \qquad \{m_2 : \circ int\} \sqsubseteq t_a \sqsubseteq \psi_{act} \qquad t = unit$$

– l'ensemble de contraintes final est donc :

$$\{m_1 : \circ int,\ m_2 : \circ int\} \sqsubseteq \psi_{act} \qquad t = unit$$

On en déduit donc qu'il faut que *act* comprenne les messages m_1 et m_2 et que le type de l'expression est *unit*.

6.5 Correction du typage

Dans cette section, nous allons démontrer la correction du système de type précédemment construit. Pour réaliser cette preuve, il faut introduire un certain nombre de lemmes nécessaires. Cette preuve se base sur celle du chapitre 4, nous ne détaillerons donc que les cas différents.

Typage du contexte d'évaluation

Les deux premiers lemmes énoncent les propriétés des contextes d'évaluation montrant que les contextes se comportent *bien* vis-à-vis du typage et du sous-typage. Le seul contexte ajouter dans le cadre de ML-Act est le constructeur, les preuves se limiterons donc à ce cas.

Lemme 6.3 (Typage d'une sous-expression) :
Si une expression est typable, toutes ses sous-expressions sont typables. Soit, formellement :

$$\mathcal{E} \vdash_e \mathrm{C}[e] : t \Longrightarrow \exists \mathcal{E}',\ \ \mathcal{E}' \vdash_e e : t'$$

De plus, l'environnement $\mathcal{E}'$ vaut $\mathcal{E}$ pour les quatre premiers contextes et vaut $\mathcal{E}+\mathcal{E}_p$ dans le cas d'un contexte `letrec` *p et où $\mathcal{E}_p$ est l'environnement obtenu par typage de p.*

Preuve : La preuve est identique.

Lemme 6.4 (Typage d'un contexte) :
Si une expression C[e] *a pour type* t_1*, le remplacement de la sous-expression* e *de type* t_2 *par une autre sous-expression* e' *de type* t_3 *sous-type de* t_2*, produit une nouvelle expression* C[e'] *de type* t_4 *sous-type de* t_1*. Soit, formellement :*

$$\mathcal{E}_1 \vdash_e \mathrm{C}[e] : t_1 \wedge \mathcal{E}_2 \vdash_e e : t_2 \wedge \mathcal{E}_2 \vdash_e e' : t_3 \wedge t_3 \sqsubseteq t_2 \Longrightarrow \mathcal{E}_1 \vdash_e \mathrm{C}[e'] : t_4 \wedge t_4 \sqsubseteq t_1$$

PREUVE : Supposons que $\mathrm{C}[e] = C(X_1, \ldots, X_{i-1}, e, X_{i+1}, \ldots, X_n)$, où X_i est une valeur ou une expression. Cette expression et e' étant typées par hypothèse, la dérivation suivante est donc possible :

$$\frac{\forall j \;\; \mathcal{E}_1 \vdash_e X_j : \alpha_j \quad \mathcal{E}_1 \vdash_e e' : t_3 \quad T(t_3) = t_4,\ \mathcal{C}'}{\mathcal{E}_1 \vdash_e C(X_1, \ldots, X_{i-1}, e', X_{i+1}, \ldots, X_n) : t_4}\ \Big(\mathcal{C}'\Big)$$

où $T(t) = \mathit{TypeC}_e(C)(\alpha_1, \ldots, \alpha_{i-1}, t, \alpha_{i+1}, \ldots, \alpha_n)$. Or, $T(t_2) = t_1,\ \mathcal{C}$ et donc par application du lemme de covariance des constructeurs de type, on a $t_4 \sqsubseteq t_1$ et $\mathcal{C} \Longrightarrow \mathcal{C}'$. Comme $\mathcal{C}$ est soluble par hypothèse alors $\mathcal{C}'$ l'est et donc la dérivation est valide.

Typage du filtrage

Nous allons maintenant énoncer et démontrer que la substitution et plus généralement le filtrage se comportent *bien* vis-à-vis du typage et du sous-typage.

Lemme 6.5 (Typage d'une substitution de variable) :
Dans une expression e*, le remplacement d'une variable* x *par une sous-expression typable* e' *sous-type de* x *produit une nouvelle expression typable sous-type de* e*. Soit, formellement :*

$$\mathcal{E}{+}\{x : t_x\} \vdash_e e : t_1 \wedge \mathcal{E} \vdash_e e' : t' \wedge t' \sqsubseteq t_x \Longrightarrow \mathcal{E} \vdash_e [e'/x]e : t_2 \wedge t_2 \sqsubseteq t_1$$

PREUVE : Si e est un constructeur, par hypothèse :

$$\frac{\mathcal{E}{+}\{x : t_x\} \vdash_e e_i : \alpha_i \quad \mathit{TypeC}_e(C)(\alpha_1, \ldots, \alpha_n) = t_1,\ \mathcal{C}_1}{\mathcal{E}{+}\{x : t_x\} \vdash_e C(e_1, \ldots, e_n) : t_1}\ \Big(\mathcal{C}_1\Big)$$

Or, $[e'/x]C(e_1, \ldots, e_n) = C([e'/x]e_1, \ldots, [e'/x]e_n)$ ce qui implique par n applications de l'hypothèse d'induction $\mathcal{E} \vdash_e [e'/x]e_i : \alpha'_i \wedge \alpha'_i \sqsubseteq \alpha_i$. On a donc :

$$\frac{\mathcal{E} \vdash_e [e'/x]e_i : \alpha'_i \quad \mathit{TypeC}_e(C)(\alpha'_1, \ldots, \alpha'_n) = t_2,\ \mathcal{C}_2}{\mathcal{E} \vdash_e [e'/x]C(e_1, \ldots, e_n) : t_2}\ \Big(\mathcal{C}_2\Big)$$

avec $t_2 \sqsubseteq t_1$ et $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ par le lemme de covariance des constructeurs de types. Donc, $\mathcal{C}_2$ est soluble.

Notons que ce lemme pourrait être limité à la substitution de valeur sémantique, car c'est la seule forme qui est indispensable à la preuve.

Cette propriété, nous permet de déduire une propriété plus générale sur le résultat d'un filtrage qui permet de substituer simultanément toutes les variables issues d'un filtrage.

Lemme 6.6 (Typage du filtrage) :
L'application d'une substitution σ, issue d'un filtrage réussi, à une expression e typable produit une nouvelle expression $\sigma(e)$ typable sous-type de e. Soit, formellement :

$$\left.\begin{array}{l} \vdash_p p\colon t_p,\ \mathcal{E}_p \\ \mathcal{E} \vdash_e v\colon t_v \wedge t_v \sqsubseteq t_p \\ v/p \Rightarrow \sigma \wedge \sigma \neq \mathit{failed} \\ \mathcal{E}{+}\mathcal{E}_p \vdash_e e\colon t_1 \end{array}\right\} \Longrightarrow \mathcal{E} \vdash_e \sigma(e)\colon t'_1 \wedge t'_1 \sqsubseteq t_1$$

PREUVE : Nous allons prouver ce lemme par induction sur la structure du filtre. Les cas initiaux de l'induction sont le joker, une variable ou une constante.

- Dans le cas d'un joker, la substitution et l'environnement sont vides et donc le lemme est trivial.
- Si p est une variable x, on se retrouve dans le cas du lemme précédent qui s'applique et fournit immédiatement le résultat.
- Si p est une constante c, l'hypothèse faite sur le type de la valeur filtrée v et celle sur la réussite du filtrage font que v vaut c. La substitution et l'environnement sont alors vides et donc le lemme est trivial.

Si le filtre est construit par le constructeur C d'arité n alors par hypothèse v vaut $C(v_1, \ldots, v_n)$. L'hypothèse $t_v \sqsubseteq t_p$ impose une même forme à v et p, le filtrage ne peut produire une erreur. De plus, comme le filtrage ne peut échouer, pour tout i, $v_i/p_i \Rightarrow \sigma_i$ avec $\sigma = \sigma_1 \circ \cdots \circ \sigma_n$.

$$\frac{\mathcal{E} \vdash_e v_i\colon \alpha_i \quad \mathit{TypeC}_e(C)(\alpha_1, \ldots, \alpha_n) = t_v,\ \mathcal{C}_v}{\mathcal{E} \vdash_e C(v_1, \ldots, v_n)\colon t_v}\left(\mathcal{C}_v\right)$$

$$\frac{\vdash_p p_i\colon \beta_i,\ \mathcal{E}_i \quad \mathit{TypeC}_p(C)(\beta_1, \ldots, \beta_n) = t_p,\ \mathcal{C}_p}{\vdash_p C(p_1, \ldots, p_n)\colon t_p,\ \mathcal{E}_p}\left(\mathcal{C}_p\right)$$

Par application du lemme de sous-typage des constructeurs de types, on a $\alpha_i \sqsubseteq \beta_i$ et ainsi par application de l'hypothèse d'induction $\mathcal{E} \vdash_e \sigma_n(e)\colon t'_n$ et $t'_n \sqsubseteq t_1$. On réapplique alors $n-1$ fois l'hypothèse d'induction et on obtient $\mathcal{E} \vdash_e \sigma_1(\cdots\sigma_n(e)\cdots)\colon t'_1$ et $t'_1 \sqsubseteq t'_2$, soit par transitivité $t'_1 \sqsubseteq t'_2 \sqsubseteq \cdots \sqsubseteq t'_n \sqsubseteq t_1$.

Typage des expressions

Nous pouvons maintenant énoncer et montrer la continuité[45] du typage fonctionnel.

Théorème 6.1 (Continuité du typage fonctionnel) :
Si une expression e de type t se réduit en une expression e' alors e' est bien typée et son type est inférieur à t. Soit, formellement :

$$\mathcal{E} \vdash_e e\colon t \wedge e \longrightarrow_e e' \Longrightarrow \mathcal{E} \vdash_e e'\colon t' \wedge t' \sqsubseteq t$$

45. En anglais, on appelle généralement ce théorème *subject reduction*.

PREUVE : La preuve de ce théorème consiste en une induction sur la structure des expressions. Les seuls qui changent par rapport au chapitre 4 sont :

- Si e est une application :
  - Les règles de contextes ont déjà été analysées.
  - Comme l'application est bien typée, la contrainte impose un type fonctionnel à la première valeur et la règle APPE1 ne peut donc pas s'appliquer.
  - Si e vaut $\lambda[p_i.e_i]^{i\in I}\ v$, par hypothèse, elle bien typée et donc chacune de ses sous-expressions est bien typée avec la contrainte : $t_{p_i} \to t_{e_i} \sqsubseteq t_v \to t$ pour tout i. On en déduit donc que $t_v \sqsubseteq t_{p_i}$ et donc le filtrage ne peut pas résulter en une erreur. La règle APPE3 ne s'applique donc pas.
  - Si l'abstraction appliquée est vide (le filtrage à échoué), la règle APPF1 la réduit en **Fail** qui est typé par $\bot$.
  - Si e vaut $\lambda[p_i.e_i]^{i\in I}\ v$ et que le premier filtrage réussit ($v/p_1 \Rightarrow \sigma$), on a : $t_{p_1} \to t_{e_1} \sqsubseteq t_v \to t$ soit $t_v \sqsubseteq t_{p_1}$ et $t_{e_1} \sqsubseteq t$. L'application du lemme de filtrage permet d'obtenir $\mathcal{E} \vdash_e \sigma(e_1) : t'$ avec $t' \sqsubseteq t_{e_1} \sqsubseteq t$. L'application de la règle APP est donc correcte.
  - Enfin, si e vaut $\lambda[p_i.e_i]^{i\in I}\ v$ et que le premier filtrage échoue ($v/p_1 \Rightarrow \mathit{failed}$) le résultat est trivial puisque si l'on retire un cas à une fonction elle reste typée.
- Si e est une définition récursive :
  - Les règles de contextes ont déjà été analysées.
  - Si e vaut `letrec` $\vec{x} = \vec{f}$ `in` e (où e à le type t), le filtrage ne peut pas échouer ou comporter une erreur puisque le motif et le corps ont la même structure et le motif est composé de variables. De plus, $t_{f_i} \sqsubseteq t_{x_i}$, ce qui permet donc d'appliquer le lemme de filtrage n fois. Le résultat est alors : $\mathcal{E} \vdash_e \sigma_1 \circ \cdots \circ \sigma_n(e) : t'_n$ avec $t'_n \sqsubseteq \cdots \sqsubseteq t'_1 \sqsubseteq t$ et la règle LREC s'applique donc correctement.

Remarque :
La décroissance du type au cours de la réduction fonctionnelle n'est pas indispensable à la correction du système de type.

Typage des configurations

La preuve de correction du typage concurrent est exactement identique à celle du chapitre 4, nous l'énoncerons seulement :

Théorème 6.2 (Continuité du typage) :
Si une configuration w bien typée se réduit en une configuration w' alors w' est bien typée. Soit, formellement :

$$\mathcal{E} \vdash_w w \wedge w \longrightarrow w' \Longrightarrow \mathcal{E} \vdash_w w'$$

Correction du typage

Finalement, le théorème énonçant la correction de notre système de type est une conséquence directe de la continuité du typage des configurations.

Théorème 6.3 (Correction du typage) :
Une configuration w bien typée ne peut se réduit en une erreur. Soit, formellement :

$$\mathcal{E} \vdash_w w \Longrightarrow w \not\longrightarrow \textbf{\textit{Err}}$$

PREUVE : L'erreur n'étant pas typable, si on suppose que w se réduit en **Err**, une application du théorème de continuité amène directement à une contradiction.

6.6 Conclusion

Dans ce chapitre, nous avons décrit précisément la forme des erreurs que nous souhaitons détecter. Puis, nous avons défini les types et le système de type construit dans ce but. Enfin, nous avons exposé les grandes lignes du processus de résolution des contraintes inférées. La preuve de la correction de ce système de type est présentée dans le chapitre suivant.

L'équipe *Vestale*, dont je suis membre, a développé deux prototypes de compilateur pour le langage ML-ACT. Le premier, que j'ai réalisé durant mon DEA (voir [Dag97]) est relativement simple. Celui-ci, à partir d'un programme ML-ACT, effectue un typage simple *à la* ML de la partie fonctionnelle et extrait un terme CAP qui exprime la partie concurrente. Le typage fonctionnel est un typage classique à la ML fortement inspiré du typage de OBJECTIVE CAML. Cette stratégie avait été choisie pour pouvoir ensuite analyser les termes CAP obtenus avec les analyseurs développés au sein de l'équipe principalement par J-L. COLAÇO dans le cadre de sa thèse [Col97]. Ces analyseurs permettaient de mettre en évidence les messages orphelins au sein des programmes ML-ACT. Cependant, cette approche de séparation des aspects fonctionnels et concurrents imposait des limites aux programmes que le système pouvait traiter. En effet, il n'était pas possible de stocker les adresses d'acteurs dans des structures fonctionnelles[46] et on ne pouvait pas effectuer d'opérations concurrentes dans une fonction récursive[47]. Pour en savoir plus sur ces limitations, on pourra consulter mon DEA [Dag97] ainsi que [DPS98]. On peut résumer ces difficultés par un slogan :

> « Il n'est pas possible de séparer les parties fonctionnelles et concurrentes d'un programme sans restreindre drastiquement les possibilités du programmeur. Le réduisant ainsi à réaliser son programme comme s'il utilisait deux langages différents et indépendants. »

Cette stratégie s'inspirait des travaux de F. NIELSON et al. qui dans [NN93] et [NNA96] calculent un effet (sous la forme d'un terme du π-calcul) à partir d'un programme CML. Les difficultés qu'ils ont rencontrées les ont amenés à simplifier à l'extrême la forme du terme concurrent extrait. Cette approche ne nous convenait pas puisqu'elle ne permettait plus beaucoup de vérifications intéressantes sur l'effet. Nous avons donc choisi de restreindre le langage.

46. Par exemple, un acteur ne pouvait pas stocker ses connaissances dans une liste.
47. Il était par exemple impossible d'écrire une fonction recevant une liste d'adresse d'acteur et un message et envoyant ce message à tous les acteurs de la liste.

Devant les difficultés (qui nous semblaient importantes) posées par cette forme de calcul d'effets, nous nous sommes lancés dans la réalisation d'un système de type couvrant les aspects fonctionnels et concurrents de ML-ACT. Dans un premier temps, nous avons étendu CAP en lui ajoutant une abstraction fonctionnelle (voir [Col98]). Avant de construire le système de type présenté dans ce chapitre, qui permet de découvrir une bonne partie des problèmes de messages orphelins dans un programme. Cependant, dans le cadre de CAP, J-L. COLAÇO a défini un système de type plus précis qui capture tous les orphelins potentiels du programme. Ce système repose sur un décompte précis au moyen de multiplicités des messages envoyés et des messages installés (voir [Col97] ou [CPDS99]).

Le système de type de ML-ACT a été conçu de façon à permettre simplement une augmentation de la précision de l'approximation des interfaces. Ainsi, pour rapprocher le système de type de l'analyse à base de multiplicités suscitée, il faut modifier :

- les termes de présence qui doivent alors contenir les multiplicités ;
- les règles de typage de l'envoi de message et celle de la construction de comportements pour « additionner » les messages ;
- le processus de résolution qui doit alors intégrer une résolution de contraintes en arithmétique entière pour calculer ces multiplicités.

La première adaptation est facilement intégrée dans le système. Un terme de présence devient un couple qui mémorise le nombre de messages envoyés et installés. Par exemple, le type de l'acteur `a_cell` (la cellule linéaire de l'exemple 5.1 page 108) est :

$$@\{start : (1,1)int,\ get : (2,\omega)@\{prn : (1,\omega)int\},\ set : (1,\omega)int\}$$

avec une contrainte qui permet de déterminer que le deuxième acteur (`a_scr`) reçoit autant de message que le premier.

Le troisième point pose des difficultés techniques pour arriver à faire collaborer le solveur de contraintes ensemblistes avec un solveur de contraintes arithmétiques. M. COLIN effectue actuellement des travaux dans ce but en s'inspirant du système de type mis au point par J. HUGHES et al. dans [HPS96, Par01]. Notons que pour des problèmes de décidabilité des inéquations arithmétiques, il faut limiter la forme des programmes typables.

Enfin, le second point, qui consiste à « compter » les messages installés, s'intègre assez facilement mais celui des messages envoyés est nettement plus complexe. À chaque nouvel interface, tous les motifs sont ajoutés dans le type en incrémentant le drapeau de la bonne présence. Il convient de remarquer que le type associé à un comportement peut éventuellement contenir une variable s'il effectue un changement de comportement sur un paramètre reçu dans un message (venant de l'extérieur). Dans le cas des messages envoyés, il faut modifier le type fonctionnel. En effet, une fonction qui contient l'envoi d'un message m, provoque l'émission de m autant de fois qu'elle est appelée. Par exemple, la fonction `send_l` suivante :

```
let rec send_l a = function
  | []   -> a
  | h::t -> send m(h) to a; send_l a t
```

peut être approximée par un type de la forme :

$$@\{m : (x,y)t\} \to t\ list_n \to @\{m : (x+n,y)t\}$$

Ce type permet de conserver le nombre de messages m envoyés à chaque appel de `send_l`. Ce nombre est égal à la taille des listes passées en second argument. Il faut donc qu'une liste conserve une approximation du nombre d'éléments qu'elle contient. Cependant, cette forme de type s'avère insuffisante puisque l'envoi de message peut être un effet de bord de la fonction. Par exemple, si la fonction `send_l` ne prend pas l'acteur `a` en paramètre ou ne le renvoie pas, le type de la fonction ne permet plus de mémoriser les messages envoyés à `a`. M. Colin explore actuellement deux pistes pour résoudre ce problème :

- Les contraintes générées par une fonction sont stockées dans son type. Le type fonctionnel devient alors : $\alpha \xrightarrow{\mathcal{C}} \beta$ et il faut modifier les règles de typage des fonctions (Fun) et d'application (App) :

$$\textsc{Fun'}: \frac{\forall i \in I \quad \vdash_p p_i : t_i,\ \mathcal{E}_i,\ \mathcal{C}_i \quad \mathcal{E}+\mathcal{E}_i \vdash_e e_i : t'_i,\ \mathcal{C}'_i}{\mathcal{E} \vdash_e \lambda[p_i.e_i]^{i\in I} : t,\ \{\}} \left(\forall i \in I \ \ t_i \xrightarrow{\mathcal{C}_i \cup \mathcal{C}'_i} t'_i \sqsubseteq t\right)$$

$$\textsc{App'}: \frac{\mathcal{E} \vdash_e f : t_f,\ \mathcal{C}_f \quad \mathcal{E} \vdash_e e : t_e,\ \mathcal{C}_e}{\mathcal{E} \vdash_e f\ e : t,\ \mathcal{C}_f \cup \mathcal{C}_e \cup \mathcal{C}} \left(t_f \sqsubseteq t_e \xrightarrow{\mathcal{C}} t\right)$$

  Cette stratégie nécessite une remise en question de la notion de sous-typage en y intégrant une notion d'implication de contraintes qui n'est pas encore aboutie.
- Le système estime le nombre de fois qu'une fonction sera appelée et paramètre les contraintes par ce nombre. Cette stratégie impose une refonte majeure du système car il faut être capable de déterminer non seulement le nombre d'appels mais surtout distinguer les appels effectués sur des paramètres identiques.
  Remarquons que cette stratégie peut être dégradée pour déterminer les fonctions qui sont réellement appelées pour cela il suffit de compter uniquement 0 ou 1. Cette extension devient relativement aisée à intégrer (puisqu'elle ne tient plus compte des arguments). Elle permet ainsi, d'augmenter la précision du système présenté dans ce chapitre. En effet, celui-ci ajoute toujours les contraintes dues aux effets de bord d'une fonction même si celle-ci n'est pas appliquée. Par exemple, si le programme contient une fonction qui envoie un message m à un acteur a, le message m est collecté dans le type de a lors de l'analyse du corps de la fonction. Cette contrainte est donc toujours générée mais si la fonction n'est pas appelée.

Troisième partie

Erlang

Chapitre 7

Le langage

Le langage ERLANG est un langage de programmation fonctionnel, concurrent et réparti conçu par ERICSSON dans le but de faciliter la programmation de leurs applications de télécommunication. Son développement découle d'une étude approfondie de l'apport des nouveaux paradigmes de la programmation moderne pour l'élaboration d'applications destinées au monde des télécommunications. L'expérience acquise par ses concepteurs lors du développement du langage les a confortés dans l'idée qu'il permettait d'améliorer sensiblement l'écriture de logiciels temps réel. Nous avons choisi d'adapter nos techniques de typage à ce langage en raison de son caractère à la fois industriel et suffisamment abstrait. Il nous semble qu'alors l'importante quantité de code opérationnel existant fournira une base intéressante d'évaluation de nos systèmes.

Dans cette thèse, nous allons présenter le système de type d'ERLANG sur un sous-ensemble du langage. En effet, certaines parties du langage se typent de façon orthogonale à l'approximation que nous utilisons. Ainsi, par exemple nous ne nous intéressons pas aux enregistrements ni aux exceptions. Un système de type pour une plus large partie du langage a été réalisé dans le cadre du prototype et ne pose pas de problèmes théoriques supplémentaires. Le sous-ensemble étudié sera désigné sous le nom de μErlang.

Dans la première section de ce chapitre, nous présentons les principales caractéristiques d'ERLANG en le replaçant dans le contexte dans lequel il a vu le jour. Puis, nous définissons plus précisément la syntaxe de μErlang et donnons quelques exemples de programme. Ensuite, nous introduisons la syntaxe de $\mathcal{F}unc_2$, le λ-calcul dans lequel nous traduisons la partie fonctionnelle de μErlang. Celui-ci diffère de $\mathcal{F}unc_1$ principalement pour la portée des définitions de variables. Nous définissons ainsi de manière formelle une sémantique de μErlang à travers la sémantique des configurations et de $\mathcal{F}unc_2$. Enfin, ce chapitre se conclut par une discussion sur les sémantiques existantes d'ERLANG. L'annexe A contient une grammaire quasiment complète du langage ERLANG. Nous renvoyons le lecteur souhaitant de plus amples informations sur ERLANG au livre de J. ARMSTRONG et al. [AVWW96], au manuel de référence du langage [BV99] et enfin au site Internet de la version libre d'ERLANG [48].

48. `http://www.erlang.org`

7.1 Présentation rapide d'Erlang

Un langage descendant de Prolog

En 1987, le premier interprète du langage a été écrit en Prolog. La version actuelle n'utilise plus Prolog mais cet héritage transparaît encore dans la syntaxe d'Erlang. Par exemple, le langage contient les notions d'atomes (identificateur commençant par une minuscule) et la notion de variable (identificateur commençant par une majuscule). Erlang a également hérité de la notion de typage dynamique et souple, se distinguant ainsi du courant fonctionnel moderne ML soutenant la notion de typage statique fort. Il est ainsi à rapprocher de LISP. Il peut également être rapproché de *Plasma* et de ses extensions (voir [Hew77] et [MMS88]).

Un langage fonctionnel moderne

Cet héritage n'empêche pas Erlang d'être un langage fonctionnel moderne qui a su intégrer les avancées récentes dans le domaine des langages de programmation. La famille ML a ainsi inspiré Erlang dans le domaine des variables, de la notion de filtrage, de la gestion automatique de la mémoire ou dans celui des fonctions d'ordre supérieur.

Erlang est ainsi un langage déclaratif où les variables sont dites *à assignation unique*. En effet, une fois définie, une variable ne pourra pas voir son contenu modifié. Cette forme de variable permet de faciliter le raisonnement sur le contenu d'une variable et se généralise actuellement dans les langages fonctionnels. Elle va de pair, en général, avec une gestion automatique de la mémoire. Simultanément à l'exécution d'un programme un *ramasse-miettes* libère la mémoire occupée par des variables dont il détermine par calcul qu'elles ne serviront plus. L'introduction d'une variable se fait par l'une des trois formes de filtrage d'une valeur : le filtrage des arguments d'une fonction, le filtrage dans une instruction de branchement multiple (`case`), le filtrage d'introduction (de variable) utilisant l'opérateur `=`[49]. Ce filtrage même si il est fortement inspiré de celui de ML conserve de son origine Prolog trois caractéristiques :

- Le filtrage est hétérogène, c'est-à-dire que lorsqu'une instruction de filtrage comporte plusieurs cas ceux-ci peuvent être de formes totalement différentes. On retrouve ainsi la notion de typage souple et dynamique. Il est, par exemple, possible de filtrer dans un premier cas sur une liste et dans un deuxième cas sur un entier, comme démontré par la première expression de l'exemple 7.1.
- Le filtrage est non-linéaire, ainsi, un motif peut contenir plusieurs fois la même variable, chacune de ces occurrences devant être liée à la même valeur après le filtrage. Par exemple, le motif `{X,X}` de la deuxième expression filtre les paires dont les deux composantes sont identiques.
- Le filtrage peut être dynamique, si une variable figurant dans le motif est déjà définie dans le contexte courant, alors sa valeur est utilisée comme motif. Par exemple, la fonction `f` définie dans l'exemple 7.1 filtre la boîte aux lettres de l'acteur courant et accepte les messages qui sont des paires dont le premier élément doit coïncider avec le paramètre reçu par la fonction lors de son appel. Ainsi, `f(get)` accepte les paires

49. Cet opérateur correspond au `let` de la famille ML.

```
% exemple de filtrage heterogene
case V of
    [X|_] -> X;
    I when integer(I) -> I
end

% exemple de filtrage non lineaire
case V of
    {X,X} -> X
end

% exemple de filtrage dynamique
f(V) ->
    receive
        {V,P} -> P
    end.
```

Exemple 7.1 Les différentes particularités du filtrage de ERLANG.

dont le premier membre est `get` et `f(set)` accepte les paires dont le premier membre est `set`. Les occurrences de variables dans les motifs sont non liantes, ce qui permet de réaliser des filtres dynamiques (par exemple dans le corps d'une fonction).

Un langage industriel

Le contexte industriel dans lequel ERLANG a été conçu a eu grande influence sur le langage. En effet, la maintenance des applications devant être aisée, le langage est resté relativement simple : seules ont été conservées les primitives utilisées régulièrement par les programmeurs. De plus, ce langage est exploité pour programmer des équipements de l'industrie des télécommunications, la robustesse a donc été un objectif primordial dans sa conception. Pour cela, ERLANG possède un puissant *système de détection des erreurs* composé d'un *runtime* et d'une notion d'exceptions tous deux distribués qui permettent de bien contrôler les erreurs et leurs propagations. Les centraux téléphoniques devant fonctionner sans arrêt, il intègre la possibilité de *changer des portions d'un programme sans arrêter son exécution*. Enfin, le contexte d'utilisation des programmes ERLANG nécessite une gestion du temps rigoureuse. C'est un langage temps réel qui contient une notion interne de temps que nous n'aborderons pas.

ERLANG est utilisé pour développer des applications complexes dont le code source est de taille importante. Le langage contient donc un *système de module* simple et il est relativement facile de faire communiquer un programme ERLANG avec des programmes écrits dans d'autres langages (C, Java...). En fait, une *grosse* application est généralement composée de trois parties :

- le noyau proche de la machine et généralement temps réel qui est codé en C,
- la partie logique et algorithmique de l'application écrite en ERLANG,
- l'interface graphique réalisée en Java.

ERLANG est utilisé dans le développement de nombreux projets au sein d'ERICSSON.

Parmi les produits aboutis et commercialisés dont la partie logicielle est écrite en ERLANG, on peut citer :

- Un serveur de mobilité [50] : une application pour PABX dont le code totalise 486 modules ERLANG consistant au total en environ 230 000 lignes de codes. Ce système permet d'utiliser le même numéro de téléphone pour plusieurs équipements (par exemple un téléphone de bureau, un téléphone portable, un fax ou même un téléphone chez soi), le serveur se chargeant de transmettre à l'équipement adéquat tout appel vers ce numéro. Ce produit ainsi que deux autres de taille nettement moindre sont décrits dans [Arm96].
- Le commutateur ATM AXD 301 dont on peut lire une présentation dans [BR98].

Au delà d'ERICSSON, de plus en plus de projets sont développés autour d'ERLANG. On peut par exemple citer le travail de BLUETAIL [51] qui commercialise des logiciels réalisés en ERLANG dans le domaine des produits Internet sécurisés.

Un langage concurrent réparti

Enfin, ERLANG est un langage concurrent réparti dans lequel la concurrence n'est pas intégrée pour augmenter ses capacités de calcul, mais dans le but de décrire explicitement le comportement concurrent naturel des applications de télécommunication. De plus, ces applications nécessitent en général la coopération de nombreuses entités ne s'exécutant pas sur le même site, ERLANG doit donc intégrer une notion de répartition.

La sémantique qui est associée aux processus Erlang est très proche de celle des acteurs. En effet, un processus possède une adresse et une boîte aux lettres dans laquelle il stocke les messages qu'il reçoit dans l'ordre de leur réception. Dès qu'il le peut un processus traite le premier message acceptable en attente dans sa boîte aux lettres. Le résultat de ce traitement peut modifier la façon dont il traitera les messages suivants, ce qui correspond à la notion de changement de comportement du modèle d'acteur. La transmission est asynchrone de type point à point et est garantie. De plus, deux messages envoyés en séquence par un processus à un même processus arrivent dans cet ordre.

Remarque :
La gestion de la boîte aux lettres en ERLANG *est assez différente de celle de* ML-ACT *et plus généralement des langages d'acteurs. En effet, pour chaque motif possible de réception, l'acteur va essayer tous les messages de sa boîte aux lettres et réagira au premier qui s'unifie avec le motif. Par exemple, soir un acteur a dont l'interface courante filtre les messages m_1 et m_2 (dans cet ordre). Si la boîte aux lettres de a est $m_2 :: m_1$, alors a traite d'abord m_1.*

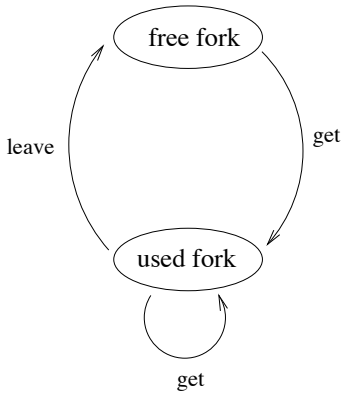
7.2 μErlang

Avant d'introduire la syntaxe de μErlang, nous allons présenter un exemple de programme afin de faciliter la compréhension du langage.

50. Son nom original est : *mobility server.*
51. `http://www.bluetail.com`

```
%Le comportement d'une fourchette libre
free_fork() ->
  receive
    {get,Source} -> Source!{ok,self()}, used_fork()
  end.

%Le comportement d'une fourchette qui a ete saisie
used_fork() ->
  receive
    {get,Source} -> Source!no, used_fork();
    leave        -> free_fork()
  end.
```



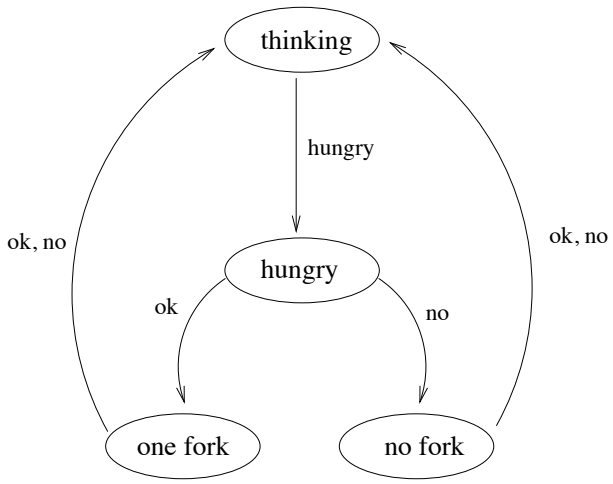
Exemple 7.2 Le comportement d'une fourchette.

Un exemple de programme Erlang

Celui-ci propose une solution du problème classique des philosophes et de leurs fourchettes. Il s'agit de réaliser un programme qui modélise n philosophes autour d'une table ronde. Chacun d'eux suit un cycle où il réfléchit, puis a faim et enfin, mange lorsqu'il dispose de deux fourchettes, ou attend ses fourchettes. Ce cycle se répète sans fin. Une fourchette est placée entre deux philosophes, il y a donc également n fourchettes sur la table. Ce problème est souvent utilisé comme exemple pédagogique lors de l'étude de la programmation concurrente car il y a de grandes chances, en cas de problème de synchronisation, de produire des états d'interblocage où par exemple chaque philosophe dispose d'une seule fourchette.

Le code du comportement des fourchettes ainsi que l'automate le décrivant figure dans l'exemple 7.2. Lorsqu'elle est libre (`free_fork`) une fourchette attend de recevoir un message (`receive`) émis par un philosophe qui cherche à la saisir. Ce message contiendra un atome indiquant qu'il s'agit bien d'attraper la fourchette (`get`), et l'adresse de celui qui cherche à l'attraper (`Source`). À la réception d'un tel message, la fourchette répond à celui qui cherche à s'en saisir qu'elle est libre en renvoyant un message (`!`) contenant également son adresse (accédée par la fonction prédéfinie `self`), puis elle devient une fourchette utilisée. Une fourchette utilisée (`used_fork`) attend un message de relâchement (`leave`) pour redevenir libre. De plus, lorsqu'elle est occupée, elle répond à toute personne cherchant à la saisir qu'elle est occupée et ne peut donc pas être attrapée (message `no`).

Les philosophes suivent le comportement de l'exemple 7.3. Nous supposons qu'ils mangent instantanément et donc l'état dans lequel ils mangent disparaît. Les philosophes effectuent donc un cycle entre leurs trois états possibles, à savoir : je réfléchis (`thinking_philo`), j'ai faim (`hungry_philo`) et j'attend les fourchettes (`ok_f_philo` et `no_f_philo`). Chaque philosophe connaît l'identité de ses fourchettes droite et gauche pour essayer de les saisir lorsqu'il a faim (dans le programme, elles sont respectivement désignées par `R` et `L`). Lorsqu'un philosophe réfléchit, il n'accepte que les messages lui indiquant qu'il a faim. S'il reçoit un tel message, il essaye de se saisir de ses fourchettes droite et gauche en leur envoyant à toutes les deux un message `get` contenant son adresse, puis devient un philosophe affamé. Lorsqu'un philosophe a faim, il attend la réponse de ses requêtes auprès des fourchettes. S'il reçoit une réponse favorable, il passe dans un second




```
%Le philosophe pense
think_philo(L,R) ->
  receive
    hungry -> L!{get,self()}, R!{get,self()}, hungry_philo(L,R)
  end.

%Il devient affame
hungry_philo(L,R) ->
  receive
    {ok,Fork} -> ok_f_philo(L,R,Fork);
    no        -> no_f_philo(L,R)
  end.

%Le philosophe a une fourchette et attend la deuxieme
ok_f_philo(L,R,F) ->
  receive
    {ok,_} -> L!leave, R!leave;
    no     -> F!leave
  end,
  self()!hungry, think_philo(L,R).

%Le philosophe sait qu'il n'aura pas les deux fourchettes
no_f_philo(L,R) ->
  receive
    {ok,Fork} -> Fork!leave, self()!hungry;
    no        -> true
  end,
  think_philo(L,R).
```

Exemple 7.3 Le comportement d'un philosophe.

```
%La fonction principale qui intialise les fourchettes et les philosophes
main() ->
    Fork1  = spawn(free_fork,[]),
    Fork2  = spawn(free_fork,[]),
    Philo1 = spawn(think_philo,[Fork1,Fork2]),
    Philo2 = spawn(think_philo,[Fork2,Fork1]),
    Philo1!(Philo2!hungry).
```

Exemple 7.4 L'initialisation des fourchettes et des philosophes.

état d'attente (`ok_f_philo`) en mémorisant qu'elle est la fourchette qu'il réussi à saisir. Si la réponse est négative, il change pour un autre état d'attente (`no_f_philo`). Lorsqu'il attend la réponse de la deuxième fourchette, un philosophe libère la fourchette qu'il a pu saisir, s'il en a qu'une (cas `no` du comportement `ok_f_philo` ou bien `ok` du comportement `no_f_philo`). De plus, comme il n'a pas pu manger, il se remet à penser mais se renvoie immédiatement un message lui signalant sa faim. Ce cas s'applique également si le philosophe a reçu deux réponses négatives. Enfin, s'il parvient à saisir simultanément ses deux fourchettes, le philosophe mange (instantanément), puis libère ses fourchettes et se remet à penser. On suppose qu'un mécanisme extérieur supplémentaire lui dira s'il a faim plus tard.

Enfin, nous présentons dans l'exemple 7.4 l'initialisation de ces acteurs dans le cas où n vaut 2. Ce code contient une fonction prédéfinie (elles sont nommées *bif* en ERLANG pour *built in function*) : `spawn`, fonction d'arité 2 prenant en paramètre une fonction `f` et la liste de ces arguments `[a1,...,an]` et créant un acteur exécutant l'application `f(a1,...,an)`. L'initialisation consiste donc à créer quatre acteurs, deux pour les fourchettes et deux pour les philosophes. Puis, deux messages sont envoyés aux philosophes pour les affamer.

La syntaxe de μErlang

La syntaxe complète d'ERLANG figure dans l'annexe A, à la page 243. Nous allons uniquement présenter le sous-ensemble de cette syntaxe qui nous intéresse. Pour simplifier, nous avons, en effet, supprimé les notions d'enregistrement, d'exception, de module et de port. De plus, l'opération de réception ne contient plus de clause *after* permettant de donner une limite temporelle à la durée de blocage du processus. La syntaxe de μErlang est alors composé de quatre sortes d'entités : les fonctions, les motifs, les expressions de garde et les expressions.

Dans les règles de la grammaire de μErlang, nous utilisons une version plurielle d'un non-terminal pour signifier une liste de tels non-terminaux. Soit, sous forme de grammaire :

$$
\begin{array}{ll}
\langle motifs \rangle & ::= \langle motif \rangle \; [\text{','} \; \langle motif \rangle]^\star \\
\langle gardes \rangle & ::= \langle garde \rangle \; [\text{','} \; \langle garde \rangle]^\star \\
\langle garde\_exprs \rangle & ::= \langle garde\_expr \rangle \; [\text{','} \; \langle garde\_expr \rangle]^\star \\
\langle exprs \rangle & ::= \langle expr \rangle \; [\text{','} \; \langle expr \rangle]^\star
\end{array}
$$

Un programme : <*prog*> ::=

[<*definition*> [';' <*definition*>]* '.']$^{+}$:
Un programme μErlang est composé d'un ensemble de définitions de fonctions dont une particulière est nommée `main`. Cette fonction principale sera exécutée automatiquement lors du lancement du programme. Chaque fonction peut comporter plusieurs cas de filtrage.

Les cas de fonction : <*definition*> ::=

ATOM '(' [<*motifs*>] ')' ['when' <*gardes*>] '->' <*exprs*> :
Tous les filtrages d'une fonction doivent introduire le même atome qui est alors le nom de la fonction. Chaque cas contient un certain nombre de motifs qui filtrent les paramètres reçus par la fonction. De plus, il est possible d'ajouter des gardes pour contraindre le choix d'une branche. Enfin, le corps de la branche est composé de plusieurs expressions qui sont exécutées en séquence.

Les motifs : <*motif*> ::=

VAR :
Un motif peut être une variable. En Erlang, les occurrences de variables dans les motifs ne sont pas toujours liantes, il est ainsi possible de référencer une variable de l'environnement dans certains cas. Nous reviendrons sur cette caractéristique page 166.

<*const*> :
En Erlang, les constantes possibles sont : les entiers (INT), les réels (FLOAT), les caractères (CHAR), les chaînes de caractères (STRING) et les atomes (ATOM). Erlang ne possède pas de vrais booléens cependant les deux atomes `true` et `false` ont presque ce statut.

'_' :
Le joker permet de capturer tous les cas possibles dans un filtrage sans définir de nouvelles variables.

'{' [<*motifs*>] '}' :
Il est possible de décomposer un tuple.

'[' [<*motifs*> ['|' <*motif*>]] ']' :
Une liste se décompose en la suite de ses premiers éléments dans l'ordre suivi de sa queue.

Les gardes : <*garde*> ::=

'true' :
Une garde peut être triviale, ce qui permet d'avoir une branche qui capture tout dans un choix `if`.

<*bif_rec*> '(' <*garde_expr*> ')' :
Il est possible de tester dynamiquement le type d'un objet, pour cela, on dispose des fonctions prédéfinies[52] (<*bif_rec*>) suivantes : '`atom`', '`float`', '`integer`', '`tuple`', '`list`', '`number`', '`pid`', '`constant`'. Les cinq premières fonctions ont un sens clair, '`number`' reconnaît les entiers et les réels, '`pid`' reconnaît les adresses d'acteurs, '`constant`' reconnaît les atomes, les nombres et les adresses d'acteurs.

<*garde_expr*> <*comp_op*> <*garde_expr*> :
Une garde peut être une comparaison de deux expressions. Les opérateurs de comparaisons (<*comp_op*>) sont : '`<`', '`=<`', '`>`', '`>=`', '`==`', '`/=`', ils sont hétérogènes et suivent l'ordre suivant sur les données : nombre < atome < pid < tuple < liste. De plus, pour ordonner deux tuples, leurs tailles sont comparées puis leurs éléments et pour deux listes leurs têtes sont ordonnées puis leurs queues comparées.

Les expressions de garde : <*garde_expr*> ::=

Les expressions de garde forment un sous-ensemble des expressions. Lors de la présentation des expressions, nous ne réexpliquons pas celles déjà introduites dans cette sous-section.

VAR

<*const*>

'`{`' [<*garde_exprs*>] '`}`'

'`[`' [<*garde_exprs*> ['`|`' <*garde_expr*>]] '`]`' :
Les variables, constantes, tuples et listes suivent la même syntaxe que celle des motifs. Notons que la variable lorsqu'elle est utilisée doit avoir été introduite dans l'environnement par un motif de filtre.

'`(`' <*garde_expr*> '`)`' :
On peut regrouper les expressions de garde en bloc par des parenthèses.

<*bif_gar*> '`(`' <*garde_exprs*> '`)`'

<*garde_expr*> <*bin_op*> <*garde_expr*>

<*un_op*> <*garde_expr*> :
Dans une expression de garde, il n'est pas possible d'appliquer une fonction définie par le programmeur. En effet, l'application est limitée à un nombre restreint de fonctions prédéfinies : les fonctions de test dynamique du type déjà introduites, les fonctions arithmétiques unaires ou binaires ('`+`', '`-`', ...), les fonctions logiques ('`or`', ...), les opérateurs sur les listes ('`++`', '`hd`', ...), les opérateurs sur les tuples ('`size`', '`element`'), les opérateurs de comparaisons et enfin l'auto-référence d'un processus sur lui-même ('`self`'). Notons que, parmi les opérateurs présentés '`+`' et '`-`' ont un sens qui dépend du contexte, en effet, ils sont soit binaires soit unaires.

52. D'autres fonctions de test de type sont incluses dans Erlang ('`record`' par exemple) mais ne seront pas définies pour μErlang, car les données correspondantes ne sont pas traitées par notre système de type.

Les expressions : <*expr*> ::=

VAR

<*const*>

'`{`' [<*exprs*>] '`}`'

'`[`' [<*exprs*> ['`|`' <*expr*>]] '`]`'

'`(`' <*expr*> '`)`'

<*expr*> <*bin_op*> <*expr*>

<*un_op*> <*expr*> :
Les expressions peuvent avoir la même forme que les expressions de garde.

<*expr*> '`(`' [<*exprs*>] '`)`' :
Dans une expression, les fonctions applicables ne sont plus restreintes.

'`begin`' <*exprs*> '`end`' :
On peut regrouper une séquence d'expressions pour former un bloc dont le résultat lors de l'évaluation est celui de l'exécution de la dernière expression.

<*motif*> '`=`' <*expr*> :
Il est possible d'introduire par filtrage de nouvelles variables, cette expression est similaire au `let` de ML. La variable, une fois introduite est figée et son contenu ne peut être modifié. L'évaluation d'une liaison renvoie le résultat de l'évaluation de l'expression.

<*expr*> '`!`' <*expr*> :
Pour envoyer un message à un acteur, il faut fournir deux expressions, la première permet le calcul de l'adresse du destinataire et le résultat du calcul de la seconde est le contenu du message. Le résultat d'un envoi est le message.

'`if`' <*gardes*> '`->`' <*exprs*> ['`;`' <*gardes*> '`->`' <*exprs*>]*'`end`' :
Il est possible de construire des choix en μErlang. Notons que la forme des expressions de condition est limité puisqu'il s'agit de gardes.

'`case`' <*expr*> '`of`' <*filtrage*> ['`;`' <*filtrage*>]*'`end`' :
En Erlang, on préférera souvent au choix ('`if`') le filtrage d'une expression avec branchement.

'`receive`' <*filtrage*> ['`;`' <*filtrage*>]*'`end`' :
Les comportements concurrents se construisent également par filtrage/branchement. La différence par rapport au '`case`' est que l'expression filtrée n'est pas donnée explicitement, mais il s'agit d'un message extrait de la queue de l'acteur courant. Contrairement au modèle classique d'acteur, Erlang contient une instruction explicite d'accès à la boite aux lettres. Cet accès est de plus synchrone puisque cette fonction renvoie comme résultat celui de l'évaluation de l'expression ayant été choisie. Notons, qu'en Erlang, il est de plus possible de fournir une limite de temps d'attente si l'expression se bloque en attente d'un message acceptable par un '`after`' <*expr*> '`->`' <*expr*> facultatif. Ce mécanisme s'intègre relativement facilement au système de type sans y introduire de notion de temps. Par contre, il complique notablement la sémantique du langage, car il faut alors y ajouter une notion de temps avec des compteurs pour déterminer si un processus doit être débloqué ou non.

'`fun`''('[< *motifs* >]')' ['`when`' < *gardes* >] '`->`' < *exprs* > [';' '('[< *motifs* >]')' ['`when`' <*gardes*>] '`->`' <*exprs*>]⋆:
Il est possible de construire des fonctions locales anonymes. Elles sont surtout utiles pour fournir une fonction simple en argument d'un appel de fonction.

Les filtrages : <*filtrage*> ::=

<*motif*> ['`when`' <*gardes*>] '`->`' <*exprs*> :
Un filtrage est similaire à la définition de fonction excepté le fait que dans le cas du filtrage un seul motif est possible.

Un certain nombre de fonctions, les *bif* sont prédéfinies en μErlang. Certaines ont été introduites lors de la présentation de sa syntaxe. Une de ces *bif* nécessite un traitement particulier, il s'agit de celle permettant de créer les acteurs : `spawn`. C'est une fonction d'arité 2 qui prend en paramètre une fonction et la liste de ses arguments et qui crée un processus exécutant l'application de la fonction. Cette fonction est traitée de façon particulière dans la section qui introduit la traduction en $\mathcal{F}unc_2$, le λ-calcul utilisé pour définir la sémantique de μErlang.

7.3 Une sémantique particulière

Avant de présenter plus en détail le cœur fonctionnel $\mathcal{F}unc_2$ et la traduction de μErlang dans celui-ci, nous allons nous pencher sur certaines particularités sémantiques du langage Erlang. Nous les avons déjà évoquées, et nous allons donner plus de détails. Il s'agit :
- des fonctions (leurs noms sont des atomes et pas des variables);
- de l'ordre d'évaluation des expressions;
- des variables et de leur portée particulière.

Les fonctions

En Erlang, le nom d'une fonction est un atome, c'est donc une entité qui peut être manipulée et en particulier dynamiquement. Cette possibilité peut rendre quasiment impossible la détermination statique des noms des fonctions qui seront réellement appliquées à l'exécution des programmes. En effet, le programmeur peut calculer le nom de la fonction qu'il souhaite appliquer. Ceci conduit immédiatement à l'indécidabilité de la détermination de l'identité de la fonction qui est appliquée. Cette caractéristique qui ancre fortement Erlang dans le camp des langages dynamiques nous a conduit à étudier finement la base de programmes Erlang *industriels*. D'une part, les programmes (et donc les programmeurs) utilisant cette possibilité sont extrêmement rares, nos outils pourront donc traiter l'immense majorité des programmes Erlang. D'autre part, notre approche du typage pour un langage à forte composante dynamique se rapproche de celle du typage souple, à savoir :

> « Calculons ce qu'il est possible de déterminer statiquement et le reste sera traité à l'exécution. »

Les fonctions sont, par contre, définies globalement et ne peuvent pas être redéfinies. La détermination de l'environnement des fonctions éventuellement accessibles est donc relativement aisé. Ce calcul est effectué en même temps que le parcours du code pour la traduction. Notons, qu'en ERLANG, il est possible de surcharger une fonction au niveau de l'arité, il est ainsi possible d'écrire une fonction `f(X)` et une fonction `f(X,Y)` dans le même module. Cet environnement est utilisé lors de l'exécution pour déterminer la fonction à appliquer lors d'un appel de fonction. L'identité d'une fonction est donc composée de son nom et de son arité. Dans un appel `f(a1,...,an)`, l'arité est connue. En revanche, lors d'une création d'acteur par `spawn(f,l)`, la liste `l` peut être calculée et donc l'arité difficile à déterminer statiquement car ce choix est fait à l'exécution en utilisant la taille de `l`.

L'ordre d'évaluation

En ERLANG, l'ordre d'évaluation de la plupart des entités d'un programme n'est pas défini. Il est supposé quelconque, donc, par exemple le calcul des éléments d'une paire `{e1,e2}` n'est pas prévisible. Il est possible que `e2` soit évalué en premier suivi de `e1` ou bien `e1` suivi de `e2`. Une variable introduite dans `e1` ne peut donc pas être utilisée dans `e2` car elle n'existe pas nécessairement. Or, ERLANG est un langage qui a été développé avec un fort souci de sécurité. Le compilateur a donc été doté d'outils permettant de vérifier statiquement qu'aucune des exécutions possibles d'un programme ERLANG ne se termine par une erreur de ce type. Les deux premières lignes de l'exemple 7.5 montrent la détection de cette sorte de problème par le compilateur. La notion de contexte que nous définissons pour exprimer la sémantique de $\mathcal{F}unc_2$ tiendra compte de cet indéterminisme sur l'ordre d'évaluation. Remarquons que cet indéterminisme permet d'évaluer plusieurs expressions en parallèle puisque le calcul de l'une ne peut dépendre de celui de l'autre.

Les variables

Après son introduction, une variable en ERLANG conserve toujours la même valeur. De plus, la définition d'une variable peut être réutilisée dans une autre introduction, ainsi, il est possible d'écrire `X=Y=8` pour `Y=8, X=8`. Cependant pour des raisons de sûreté d'exécution la portée des variables peut être limitée par certaines instructions. Par exemple, dans une expression qui comporte plusieurs branches d'exécution possibles, une définition figurant dans une branche n'est exportée à l'extérieur que si elle est définie dans toutes les autres branches. Ainsi, quelque soit la branche d'exécution effectivement choisie cette variable est introduite dans l'environnement. Dans la fonction `f/1` de l'exemple 7.5, à l'issue du `case` seule la variable `YY` est disponible, et ce même si la première branche est exécutée (où `ZZ` est définie). Chaque variable locale est indéfinie en sortie de l'expression de branchement (dans le cas de la première branche `ZZ`). Notons que le compilateur signale le fait que la variable `YY` est exportée hors de son expression de définition pour `f/1` et que la variable `ZZ` n'est pas sûre dans le cas de `g/1`. Dans le deuxième cas, la compilation se termine alors par une erreur.

Les deux règles du calcul de la portée d'une variable sont :

– Aucune variable ne peut être exportée d'une fonction, qu'elle soit globale ou locale.

```
% Definition et ordre d'evaluation
f() ->
    {(Z=1,2),Z}.
%% variable 'Z' is unbound

% Exportation de variable
f(X) ->
    case X of
        1 -> YY=2,ZZ=3,1;
        _ -> YY=3,X
    end,
    YY.
%% Warning: variable 'YY' exported from 'case' (line 6)
%% {ok,essai}
{f(1),f(2)}.
%% {2,3}

% Limite de l'exportation de variable
g(X) ->
    case X of
        1 -> YY=2,ZZ=3,1;
        _ -> YY=3,X
    end,
    ZZ.
%% Warning: variable 'ZZ' unsafe in 'case' (line 12)
%% Warning: variable 'YY' exported from 'case' (line 6)
%% error

% Le filtrage est dynamique dans le case
f(X,Y) ->
    case Y of
        X -> egaux;
        _ -> differents
    end.
{f(1,2),f(2,2)}.
%% {differents,egaux}

% mais pas dans le fun
f(X,Y) ->
    apply(fun(X) -> X end,Y).
%% Warning: variable 'X' shadowed in 'fun'
%% {ok,essai}
f(8,1).
%% 1
```

Exemple 7.5 Quelques exemples de code ERLANG.

– Pour qu'une variable définie dans une expression de *choix* (`if`, `case` ou `receive`) soit visible, elle doit être définie dans toutes les branches de cette expression.
– Aucune variable ne peut être libre dans le corps d'une fonction globale.

Dans la traduction nous effectuons ce calcul de portée afin de simplifier la sémantique de $\mathcal{F}unc_2$.

La deuxième particularité des introductions de variable est illustrée par les deux versions de la fonction `f/2` de l'exemple 7.5. Il s'agit du fait que les filtrages peuvent être dynamiques et donc qu'une occurrence de variable dans un motif n'est pas toujours liante. La première version de `f/2` permet de montrer un cas où le filtrage est dynamique. On utilise cette possibilité pour comparer les deux arguments reçus par une fonction en utilisant un `case`. Le second corps de la fonction `f/2` met en évidence une autre instruction où, par contre, une occurrence de variable dans un motif est liante et ne dépend pas du contexte. C'est le cas de la définition de fonction locale `fun`, notons qu'alors le compilateur signale que, dans le corps de cette fonction locale, la variable `X` de l'environnement n'est pas accessible puisqu'elle est cachée par le paramètre de la fonction.

Les deux règles de détermination du type d'occurrence d'une variable dans un motif sont :

– Si le motif figure dans le filtrage de définition d'une fonction, il est liant, et ce, que la fonction soit globale (`f(X)-> ...`) ou locale (`fun(X)-> ...`).
– Une variable figurant dans un motif d'expression de choix (`if`, `case` ou `receive`) n'est pas liante. Donc, si la variable figure dans l'environnement d'évaluation du motif, on la remplace par sa valeur, sinon elle correspond à une introduction usuelle.

Enfin, les deux règles qui régissent les variables libres sont :

– Aucune variable ne peut être libre dans le corps d'une fonction globale.
– Si une variable est libre dans une expression de *choix* (`if`, `case` ou `receive`) ou dans une fonction locale (`fun`) elle est liée à la valeur de cette variable lors de son évaluation.

Après avoir présenté la syntaxe de μErlang et ses particularités sémantiques, nous pouvons définir le cœur fonctionnel dans lequel μErlang sera traduit.

7.4 Le langage $\mathcal{F}unc_2$

Suivant un procédé similaire à celui utilisé pour ML-Act, nous utilisons un petit noyau fonctionnel pour décrire la sémantique de μErlang et son système de type. Ce λ-calcul avec constantes, que nous appelons $\mathcal{F}unc_2$, est présenté succinctement dans cette section. Sa syntaxe est proche de celle de $\mathcal{F}unc_1$ (sauf pour les définitions) mais leurs sémantiques sont radicalement différentes.

Définition 7.1 ($\mathcal{F}unc_2$) :
Une expression fonctionnelle est un terme qui représente un programme fonctionnel. Sa définition est paramétrée par cinq ensembles : l'ensemble des variables (notées $x \in \mathbb{V}$), l'ensemble des adresses d'acteur (notées $a \in \mathbb{A}$), l'ensemble des constantes (notées $c \in \mathbb{C}$), celui des fonctions prédéfinies (notées $f \in \mathcal{F}_c$) et celui des constructeurs (notés $C \in \mathcal{C}$).

- *L'ensemble des constantes* $\mathbb{C}$ *contient les entiers, les flottants (décimaux en terme mathématique), les atomes ainsi que les caractères, les chaînes de caractères et l'erreur. Soit :*

$$\mathbb{C} \triangleq \mathbb{N} \cup \mathbb{D} \cup \mathbb{A}_t \cup \mathit{Char} \cup \mathit{Char}^n \cup \{\boldsymbol{\mathit{Err}}\}$$

- *Celui des fonctions prédéfinies* $\mathcal{F}_c$ *contient toutes les fonctions usuelles de comparaison et d'arithmétique élémentaire ainsi que les fonctions du modèle d'acteur : send, new, init et receive*[53]*. Soit :*

$$\mathcal{F}_c \triangleq \{\mathit{send},\ \mathit{new},\ \mathit{init},\ \mathit{receive},\ \texttt{++},\ \texttt{==},\ \texttt{=<},\ldots,\ \texttt{or},\ldots,\ \texttt{+},\ \texttt{-},\ldots,\ \texttt{atom},\ \texttt{constant},\ \texttt{float},\ \texttt{integer},\ \texttt{list},\ \texttt{number},\ \texttt{pid},\ \texttt{tuple},\ \texttt{hd},\ \texttt{element},\ldots\}$$

- *Toutes les expressions peuvent être des messages. Soit :*

$$\mathcal{M}\mathit{ess} \triangleq \mathcal{E}\mathit{xp}$$

- *L'ensemble des constructeurs* $\mathcal{C}$ *est identique à celui de* $\mathcal{F}unc_1$ *donc :*

$$\mathcal{C} \triangleq \{\mathit{nop},\ \mathit{nil},\ \mathit{cons},\ \mathit{tuple2},\ldots,\ \mathit{tuplen}\}$$

- *L'ensemble des valeurs sémantiques contient : les adresses d'acteurs, les constantes, les termes construits à partir de valeurs sémantiques, les abstractions et les fonctions constantes. On a donc*

$$\mathcal{V} \triangleq \mathbb{A} \cup \mathbb{C} \cup \mathcal{C} \times \mathcal{V}^n \cup \mathcal{F} \cup \mathcal{F}_c$$

Les expressions sont construites par la grammaire qui suit :

$$e ::= x \mid a \mid c \mid f \mid C(e,\ldots,e) \mid (e) \mid e\,;e \mid e\ e \mid \lambda[p,e.e,\ldots,p,e.e] \mid \texttt{let}\ p = e$$
$$p ::= \_ \mid x \mid c \mid C(p,\ldots,p)$$

Remarques :

- *Les variables introduites par abstraction (*$\lambda[p,e.e,\ldots,p,e.e]$*) ou par définition (*$\texttt{let}\ p = e$*) sont accessible dans tout le reste du programme.*
- *Remarquons immédiatement que* $\mathcal{F}unc_2$ *a une syntaxe un peu plus expressive que celle de* μ*Erlang notamment au niveau des gardes. Ce choix a été fait uniquement pour simplifier sa syntaxe. Comme, nous n'écrivons pas directement de programmes en* $\mathcal{F}unc_2$*, mais que nous ne manipulons uniquement des termes résultant de la traduction, ils respectent les limitations de* ERLANG *(certaines gardes théoriquement possibles n'apparaissent jamais). De plus, pour faciliter la lecture des expressions, nous notons les gardes g même si ce sont des expressions.*

53. Il est également possible ajouter l'opérateur *become*, mais comme il n'est pas généré par la traduction d'un programme μErlang présentée ici, nous ne l'avons pas intégré à $\mathcal{F}unc_2$. Notons qu'avec une analyse plus précise du programme source μErlang, il est possible de traduire un certain nombre de *receive* en *become*. Nous ignorons cette possibilité car elle n'apporte aucun intérêt au système de type que nous allons développer.

– *Dans cette syntaxe, le programmeur ne peut réaliser de définitions récursives (du type* `letrec`*). Cela provient du fait que la définition de fonction globale est récursive par défaut en* ERLANG *et que c'est la seule utilisation possible de la récursivité. Cette récursivité n'est pas codée comme en* ML-ACT *par une substitution paresseuse mais par un environnement global que nous pouvons calculer statiquement : l'environnement de toutes les fonctions globales définies dans un programme.*

Une expression peut donc être une variable x, une adresse d'acteur a, une constante c, une fonction prédéfinie f ou un terme construit par C. Il est également possible de créer des blocs d'expressions en mettant une expression entre parenthèses. Les expressions peuvent être mises en séquence et cette séquence ne peut être considérée comme un banal sucre syntaxique contrairement à ce qui se passe pour $\mathcal{F}unc_1$. Cette différence provient de la portée des variables plutôt inhabituelle dans le langage ERLANG. À l'image du λ-calcul, elle contient l'application ainsi que l'abstraction. Celle-ci, plus complexe que l'abstraction usuelle, permet de construire un filtrage à plusieurs cas. Chaque cas est composé d'un motif, d'une garde et d'une expression, et intuitivement, si l'argument reçu respecte le motif et que le prédicat de garde est vérifié, l'expression fournit la suite du calcul. L'ensemble des abstractions (ou fonctions) est noté $\mathcal{F}$. Enfin, il est possible d'introduire des variables. La sémantique donnera un sens rigoureux à chaque expression, mais intuitivement, elles conservent leur sens usuel dans le contexte fonctionnel. La seule exception est la définition de variable qui contrairement à l'usage ne définit pas la portée de cette introduction (elle est calculée statiquement).

Par abus de notation, l'écriture de certaines fonctions est simplifiée. Les filtrages ne comportant qu'un seul cas suivent l'écriture usuelle du λ-calcul ($\lambda[p, g.e]$ est notée $\lambda p, g.e$) et les fonctions à plusieurs paramètres sont aplaties ($\lambda p_1, g_1. \ldots . \lambda p_n, g_n.e$ est notée $\lambda p_1, g_1 \ldots p_n, g_n.e$). Pour des raisons de commodité d'écriture, une version indicée est employée pour les fonctions : $\lambda[p_i, g_i.e_i]^{i \in 1..n}$. Enfin, les gardes sont omises si elles sont triviales (toujours vraies).

Dans le premier chapitre, nous avons introduit la notion de configuration. Sur ces configurations, nous avons défini une notion de nom libre qui dépendait de la partie fonctionnelle. Maintenant, que celle-ci a été introduite précisément dans le cadre de μErlang, nous pouvons définir ce qu'est un nom libre dans une expression de μErlang. Le principe est identique au calcul des noms libres sur les termes de $\mathcal{F}unc_1$, seules les règles changent légèrement. Un nom est **libre** dans une expression, si et seulement si il n'est pas lié par un opérateur ν de configuration dans la portée duquel figure l'expression. Autrement dit, tout nom figurant dans une expression est libre et la configuration le contenant le restreint si nécessaire.

Définition 7.2 (Nom libre) :
Tout nom figurant dans une expression est ***libre****. L'ensemble des noms libres d'une expression e est noté $\mathcal{FN}(e)$ et est calculé par l'ensemble des règles 7.1.*

Remarque :
Cette définition porte sur les noms (ou adresses) et pas sur les variables. Même si le principe de calcul est similaire à celui de $\mathcal{F}unc_1$, il est légèrement modifié pour parcourir les gardes et les motifs. En effet, le filtrage dynamique fait qu'un motif peut contenir une adresse.

$$
\begin{array}{llll}
\mathcal{FN}(a) & = \{a\} & \mathcal{FN}(C(e_1,\ldots,e_n)) & = \bigcup\{\mathcal{FN}(e_i)|i \in 1..n\} \\
\mathcal{FN}(c) & = \{\} & \mathcal{FN}(e_1\ e_2) & = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2) \\
\mathcal{FN}(x) & = \{\} & \mathcal{FN}(e_1\,;e_2) & = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2) \\
\mathcal{FN}(f) & = \{\} & \mathcal{FN}(\texttt{let}\ p = e) & = \mathcal{FN}(p) \cup \mathcal{FN}(e) \\
\mathcal{FN}((e)) & = \mathcal{FN}(e) & \mathcal{FN}(\lambda[p_i, g_i.e_i]^{i\in I}) & = \bigcup\{\mathcal{FN}(p_i) \cup \mathcal{FN}(g_i) \cup \mathcal{FN}(e_i)|i \in I\}
\end{array}
$$

Règles 7.1 Ensemble des noms libres d'une expression de $\mathcal{F}unc_2$.

7.5 Traduction de μErlang vers $\mathcal{F}unc_2$

Pour que la sémantique de $\mathcal{F}unc_2$ respecte les particularités introduites à la section 7.3, nous utilisons la stratégie suivante :

> « Renommons, par des variables *fraîches*, les variables risquant d'être capturées par l'environnement alors qu'elles ne devraient pas l'être. »

Dans le premier programme de l'exemple 7.6, la sémantique d'ERLANG impose aux occurences de la variable `X` de la fonction locale de ne pas référencer la variable définie (`X=1`). Or, nous souhaitons traduire ce programme en `let X` $= 1\,;(\lambda x.x)\ 2$ dans lequel, les `X` seront capturés. Nous renommons donc toutes les occurrences de `X` dans le corps de la fonction locale pour éviter la capture. Dans le deuxième petit programme, à l'inverse, on ne renommera pas la variable `Y` dans le corps du `case` pour provoquer cette liaison. Le second problème dû aux portées apparaît également dans cet exemple. A l'issue du choix, il faut interdire l'accès à la variable `W` qui est définie seulement dans la première branche, mais à l'opposé, la définition de `Z` qui figure dans les deux branches doit être exportée. Donc, toutes les occurrences de `W` dans le corps du choix sont renommées mais pas celle de `Z`. L'exemple 7.7 montre ce que le programme devient après l'application de cette stratégie.

La traduction d'un programme μErlang se déroule en deux étapes. Au cours d'une première passe, nous allons marquer dans chaque expression les variables introduites qui risquent de poser problème. Ces variables sont ensuite renommées grâce à une fonction qui construit une variable *fraîche* (à chaque fois qu'elle est appelée). Cette première étape n'est pas décrite en détail, on suppose simplement disposer d'un programme dans lequel aucun problème de conflit de variable ne peut se produire. Le résultat de cette phase est composé de la suite des fonctions du programme ainsi que du corps de la fonction principale (le corps de la fonction de nom `main` ou *nop* s'il n'y a pas de fonction principale).

La deuxième phase de traduction des programmes μErlang en programmes $\mathcal{F}unc_2$ construit, à partir du résultat de la première passe, l'environnement de définition des fonctions globales et l'acteur anonyme qui traduit la fonction principale. Cette phase suit un procédé similaire à celui de la traduction de ML-ACT en $\mathcal{F}unc_1$. Cette traduction est notée $[\![]\!]$ et son calcul est effectué par l'intermédiaire de cinq fonctions de traduction :

- $[\![]\!]_M$ qui prend en argument le programme et renvoie l'expression qui sera le corps de l'acteur anonyme résultat de la traduction ;

```
% exemple de capture possible non voulue
X = 1,
fun(X) -> X end 2,
...

% exemple de capture voulue et d'extension
Y = 1,
case (X) of
    Y -> Z = 2, W = 2, egaux;
    _ -> Z = 3, differents
end,
Z,
W = 12,
...
```

Exemple 7.6 Les risques de capture en ERLANG.

```
% exemple de capture possible non voulue
X = 1,
fun(XX) -> XX end 2,
...

% exemple de capture voulue et d'extension
Y = 1,
case (X) of
    Y -> Z = 2, WW = 2, egaux;
    _ -> Z = 3, differents
end,
Z,
W = 12,
...
```

Exemple 7.7 La résolution des risques de capture en ERLANG.

- $[\![]\!]_F$ qui traduit toutes les fonctions et les range dans un environnement qui est utilisé dans la suite ;
- $[\![]\!]_D$ qui traduit une fonction en abstraction ;
- $[\![]\!]_{cons}$, $[\![]\!]_{and}$ et $[\![]\!]_{seq}$ qui traduisent des listes d'expressions ;
- $[\![]\!]_P$ qui traduit un motif ;
- $[\![]\!]_E$ qui traduit une expression.

La première de règle est en partie proche de celle de ML-ACT, elle exprime le fait que les expressions principales du programme sont traduites en un acteur anonyme. En général, ces expressions démarrent les acteurs nécessaires à l'application. La traduction d'un programme produit donc un couple :

$$[\![P]\!] \triangleq [\![P]\!]_F,\ \star \triangleright [\![P]\!]_M$$

Pour traduire la fonction d'initialisation, la fonction $[\![]\!]_M$ parcourt le programme pour rechercher la fonction `main`. Si elle ne la trouve pas le corps sera vide et sinon il est composé de la traduction des expressions du corps de la fonction `main` par la fonction $[\![]\!]_{seq}$. Soit :

$$[\![]\!]_M \triangleq nop \qquad [\![\texttt{main() -> } \vec{e}.\ \ P]\!]_M \triangleq [\![\vec{e}]\!]_{seq}$$

$$[\![\texttt{f(}\overrightarrow{p_1}\texttt{) when } \overrightarrow{g_1} \texttt{ -> } \overrightarrow{e_1}\texttt{; ... ;f(}\overrightarrow{p_n}\texttt{) when } \overrightarrow{g_n} \texttt{ -> } \overrightarrow{e_n}.\ \ P]\!]_M \triangleq [\![P]\!]_M$$

Les noms de fonctions en ERLANG sont des atomes et donc, lors de l'application d'une fonction ou de la création d'un acteur par l'opération `spawn`, il est nécessaire de retrouver le corps de la fonction associée à l'atome utilisé. Pour cela, le processus de traduction construit un environnement dans lequel chaque atome définissant une fonction est relié à son corps. Cet environnement est construit par la fonction de traduction $[\![]\!]_F$. Celle-ci parcourt le programme et traduit toutes les fonctions (excepté la fonction principale) en une abstraction de $\mathcal{F}unc_2$. Cette traduction est alors reliée à l'atome nom de la fonction. Cet environnement $\mathcal{E}$ résultat est utilisé dans la sémantique lors d'un appel de fonction. Ainsi, l'atome nom de la fonction à appliquer est calculé et permet via $\mathcal{E}$ de retrouver son corps.

$$[\![]\!]_F \triangleq \{\} \qquad [\![\texttt{main() -> } \vec{e}.\ \ P]\!]_F \triangleq [\![P]\!]_F$$

$$[\![\begin{array}{l}\texttt{f(}\overrightarrow{p_1}\texttt{) when } \overrightarrow{g_1} \texttt{ -> } \overrightarrow{e_1}\texttt{;}\\ \quad\ldots\texttt{;}\\ \texttt{f(}\overrightarrow{p_n}\texttt{) when } \overrightarrow{g_n} \texttt{ -> } \overrightarrow{e_n}.\ \ P\end{array}]\!]_F \triangleq \begin{array}{l}\{\texttt{f}, \lambda[[\![\overrightarrow{p_1}]\!]_{cons}, [\![\overrightarrow{g_1}]\!]_{and}.[\![\overrightarrow{e_1}]\!]_{seq},\\ \qquad\ldots,\\ \qquad [\![\overrightarrow{p_n}]\!]_{cons}, [\![\overrightarrow{g_n}]\!]_{and}.[\![\overrightarrow{e_n}]\!]_{seq}]\} + [\![P]\!]_F\end{array}$$

Remarquons que les gardes sont facultatives en μErlang, durant la traduction, le vecteur de garde peut alors être vide.

Les trois derniers couples de règles traduisent des séquences d'expressions respectivement en une liste (par *cons*), en une conjonction (par *and*) ou en une séquence (par ;). La première des règles du couple donne la valeur d'une séquence vide, soit respectivement *nil*, *true* et *nop*.

$$\begin{array}{lcllcl}[\![]\!]_{cons} & \triangleq & nil & [\![p_1, \ldots, p_n]\!]_{cons} & \triangleq & cons([\![p_1]\!]_E, \ldots cons([\![p_n]\!]_E, nil))\\ [\![]\!]_{and} & \triangleq & true & [\![g_1, \ldots, g_n]\!]_{and} & \triangleq & [\![g_1]\!]_E, \ldots, [\![g_n]\!]_E\\ [\![]\!]_{seq} & \triangleq & nop & [\![e_1, \ldots, e_n]\!]_{seq} & \triangleq & [\![e_1]\!]_E\,; \ldots ; [\![e_n]\!]_E\end{array}$$

$$
\begin{array}{ll}
[\![\texttt{\_}]\!]_E & \triangleq \_ \\
[\![x]\!]_E & \triangleq x \\
[\![c]\!]_E & \triangleq c \\
[\![\texttt{\{\}}]\!]_E & \triangleq nop \\
[\![\texttt{[]}]\!]_E & \triangleq nil \\
[\![un\ e]\!]_E & \triangleq un\ cons([\![e]\!]_E, nil) \\
[\![\texttt{(}e\texttt{)}]\!]_E & \triangleq ([\![e]\!]_E)
\end{array}
\qquad
\begin{array}{ll}
[\![\texttt{\{}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{\}}]\!]_E & \triangleq tuplen([\![e_1]\!]_E, \ldots, [\![e_n]\!]_E) \\
[\![\texttt{[}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{|}e\texttt{]}]\!]_E & \triangleq cons([\![e_1]\!]_E, \ldots cons([\![e_n]\!]_E, [\![e]\!]_E)) \\
[\![e_1\texttt{,}\ e_2]\!]_E & \triangleq [\![e_1]\!]_E\,;[\![e_2]\!]_E \\
[\![e_1\ op\ e_2]\!]_E & \triangleq op\ cons([\![e_1]\!]_E, cons([\![e_2]\!]_E, nil)) \\
[\![p\ \texttt{=}\ e]\!]_E & \triangleq \texttt{let}\ [\![p]\!]_P = [\![e]\!]_E \\
[\![\texttt{self()}]\!]_E & \triangleq ego
\end{array}
$$

Règles 7.2 Traduction des motifs et des expressions de μErlang.

Comme nous l'avons déjà évoqué, les gardes sont considérées être des expressions quelconques, elles sont donc traduites par la fonction de traduction des expressions. Notons que pour diminuer le nombre de règles à présenter, nous considérons que les motifs sont également des expressions, car leur traduction est identique.

Les règles 7.2 décrivent la traduction des motifs et des expressions. Ces règles de traduction sont pour la plupart identiques à celles utilisées pour la traduction de ML-Act en $\mathcal{F}unc_1$. Notons que les paramètres des fonctions ne sont plus des tuples mais des listes et que l'on perd donc l'arité. Cette traduction suppose que le programme ne contient pas de variable appelée *ego* (sinon, elle est renommée) afin de pouvoir utiliser le même nom que dans le cadre de $\mathcal{F}unc_1$. Nous allons décrire en détail uniquement celles qui sont un peu plus complexes :

- L'application d'une fonction doit, à l'exécution, retrouver son corps en utilisant son nom (l'atome calculé). Pour cela, nous définissons une fonction *Fun* de $\mathcal{F}unc_2$ qui prend en argument un atome et qui utilise l'environnement (issu de la traduction $[\![]\!]_F$) pour récupérer le corps de la fonction :

$$[\![e(\vec{e})]\!]_E \triangleq Fun([\![e]\!]_E)\ [\![\vec{e}]\!]_{cons}$$

- L'envoi de message diffère légèrement de celui de ML-Act puisqu'il renvoie comme résultat la valeur émise :

$$[\![e_1\texttt{!}\ e_2]\!]_E \triangleq send\ tuple2(\texttt{let}\ \_x = [\![e_2]\!]_E, [\![e_1]\!]_E)\,;\_x$$

  Nous utilisons la notation $\_x$ pour dénoter une variable qui ne peut pas figurer dans le programme.
- La création d'acteur est voisine de celle de ML-Act:

$$[\![\texttt{spawn(}e_1\texttt{,}e_2\texttt{)}]\!]_E \triangleq init\ tuple2(\texttt{let}\ \_a = new, \lambda ego.([\![e_1]\!]_E\ [\![e_2]\!]_E))\,;\_a$$

- Une fonction locale (`fun`) est traduite exactement comme une fonction globale.

$$
\begin{array}{l}
[\![\texttt{fun (}\overrightarrow{p_1}\texttt{) when }\overrightarrow{g_1}\texttt{ -> }\overrightarrow{e_1}\texttt{;} \\
\quad \ldots\texttt{;} \\
\quad \texttt{(}\overrightarrow{p_n}\texttt{) when }\overrightarrow{g_n}\texttt{ -> }\overrightarrow{e_n}\ \texttt{end}]\!]_E
\end{array}
\triangleq
\begin{array}{l}
\lambda[[\![\overrightarrow{p_1}]\!]_{cons}, [\![\overrightarrow{g_1}]\!]_{and}.[\![\overrightarrow{e_1}]\!]_{seq}, \\
\quad \ldots, \\
\quad [\![\overrightarrow{p_n}]\!]_{cons}, [\![\overrightarrow{g_n}]\!]_{and}.[\![\overrightarrow{e_n}]\!]_{seq}]
\end{array}
$$

– Les deux choix (`if` et `case`) sont traduits de manière identique sur le principe du choix de ML-ACT (`match`). Le filtrage est traduit sur le même principe que celui des fonctions globales, la seule différence étant que le motif n'est pas transformé en liste. Cette fonction est alors immédiatement appliquée sur la valeur test du choix. Dans le cas d'un `if` tous les motifs valent 1, les gardes correspondent aux divers choix possibles et la valeur testée vaut 1.

$$[\![\texttt{if } \overrightarrow{g_1} \texttt{ -> } \overrightarrow{e_1}\texttt{; ... ; } \overrightarrow{g_n} \texttt{ -> } \overrightarrow{e_n} \texttt{ end}]\!]_E \triangleq \begin{array}{l}\lambda[1, [\![\overrightarrow{g_1}]\!]_{and}.[\![\overrightarrow{e_1}]\!]_{seq},\\ \quad \ldots,\\ \quad 1, [\![\overrightarrow{g_n}]\!]_{and}.[\![\overrightarrow{e_n}]\!]_{seq}]\ 1\end{array}$$

$$[\![\texttt{case } e \texttt{ of } p_1 \texttt{ when } \overrightarrow{g_1} \texttt{ -> } \overrightarrow{e_1}\texttt{;} \quad \ldots\texttt{;} \quad p_n \texttt{ when } \overrightarrow{g_n} \texttt{ -> } \overrightarrow{e_n} \texttt{ end}]\!]_E \triangleq \begin{array}{l}\lambda[[\![p_1]\!]_E, [\![\overrightarrow{g_1}]\!]_{and}.[\![\overrightarrow{e_1}]\!]_{seq},\\ \quad \ldots,\\ \quad [\![p_n]\!]_E, [\![\overrightarrow{g_n}]\!]_{and}.[\![\overrightarrow{e_n}]\!]_{seq}]\ [\![e]\!]_E\end{array}$$

– En μErlang, un acteur accède à sa boîte aux lettres explicitement de manière synchrone, en lui fournissant alors son interface. Cette opération est traduite en un appel de la fonction *receive* sur la traduction de son filtrage.

$$[\![\texttt{receive } p_1 \texttt{ when } \overrightarrow{g_1} \texttt{ -> } \overrightarrow{e_1}\texttt{; ... end}]\!]_E \triangleq \mathit{receive}(\lambda[[\![p_1]\!]_E, [\![\overrightarrow{g_1}]\!]_{and}.[\![\overrightarrow{e_1}]\!]_{seq}, \ldots])$$

Afin d'illustrer cette traduction, traitons le programme des philosophes et des fourchettes composé des exemples 7.2 de la page 159, 7.3 de la page 160 et 7.4 de la page 161. Ce programme, que nous nommons P, contient six fonctions (`free_fork`, `used_fork`, `think_philo`, `hungry_philo`, `ok_f_philo` et `no_f_philo`), que nous traduisons par $[\![]\!]_F$ et stockons dans un environnement $\mathcal{E}$, et une fonction principale qui est traduite par $[\![]\!]_M$. Notons que toutes les gardes de ce programme sont vides et sont traduites en gardes vraies (*true*) et *oubliées*. De plus, une liste à deux éléments $cons(x, cons(y, nil))$ est notée $[x, y]$ et les constructeurs de tuples sont omis.

$$\begin{aligned}[\![\texttt{free\_fork() ->...}]\!]_F &= \{\texttt{free\_fork},\ \lambda[[\![]\!]_{cons}, [\![]\!]_{and}.[\![\texttt{receive...}]\!]_{seq}]\}\\ &= \{\texttt{free\_fork},\ \lambda[nil, true.receive(FF)]\} \quad \text{avec}\end{aligned}$$

$$\begin{aligned}FF &= \lambda[[\![\texttt{\{get,Source\}}]\!]_E, [\![]\!]_{and}.[\![\texttt{Source...}]\!]_{seq}]\\ &= \lambda[\mathit{tuple2}([\![\texttt{get}]\!]_E, [\![\texttt{Source}]\!]_E), true.[\![\texttt{Source!\{ok,self()\}}]\!]_E\,;[\![\texttt{used\_fork()}]\!]_{seq}]\\ &= \lambda[(\texttt{get}, \texttt{Source}).\mathit{send}(\texttt{let } x = [\![\texttt{\{ok,self()\}}]\!]_E, [\![\texttt{Source}]\!]_E)\,;x\,;\\ &\quad \mathit{Fun}(\texttt{used\_fork})\ [\![]\!]_{seq}]\\ &= \lambda[(\texttt{get}, \texttt{Source}).\mathit{send}(\texttt{let } x = ([\![\texttt{ok}]\!]_E, [\![\texttt{self()}]\!]_E), \texttt{Source})\,;x\,;\\ &\quad \mathit{Fun}(\texttt{used\_fork})\ nil]\\ &= \lambda[(\texttt{get}, \texttt{Source}).\mathit{send}(\texttt{let } x = (\texttt{ok}, ego), \texttt{Source})\,;x\,;\mathit{Fun}(\texttt{used\_fork})\ nil]\end{aligned}$$

La fonction `used_fork` est traduite presque identiquement :

$$\begin{aligned}[\![\texttt{used\_fork() ->...}]\!]_F &= \{\texttt{used\_fork},\ \lambda[[\![]\!]_{cons}, [\![]\!]_{and}.[\![\texttt{receive...}]\!]_{seq}]\}\\ &= \{\texttt{used\_fork},\ \lambda[nil, true.receive(UF)]\} \quad \text{avec}\end{aligned}$$

$$
\begin{aligned}
UF &= \lambda[(\texttt{get}, \texttt{Source}).[\![\texttt{Source!no}]\!]_E\,;[\![\texttt{used\_fork()}]\!]_E,\\
&\qquad \texttt{leave}.[\![\texttt{free\_fork()}]\!]_E]\\
&= \lambda[(\texttt{get}, \texttt{Source}).\mathit{send}(\texttt{let}\ x = \texttt{no},\ \texttt{Source})\,;x\,;\mathit{Fun}(\texttt{used\_fork})\ \mathit{nil},\\
&\qquad \texttt{leave}.\mathit{Fun}(\texttt{free\_fork})\ \mathit{nil}]
\end{aligned}
$$

La traduction des quatre fonctions décrivant le comportement des philosophes sont dénotées HP, TP, OP et NP. Nous ne détaillons pas toutes les étapes de leur traduction.

$$
\begin{aligned}
HP &= \lambda[[\![\texttt{L},\texttt{R}]\!]_{cons}, [\![]\!]_{and}.[\![\texttt{receive...}]\!]_{seq}]\\
&= \lambda[[\texttt{L},\texttt{R}].\mathit{receive}(\lambda[(\texttt{ok},\texttt{Fork}).[\![\texttt{ok\_f\_philo(L,R,Fork)}]\!]_E,\\
&\qquad\qquad \texttt{no}.[\![\texttt{no\_f\_philo(L,R)}]\!]_E])]\\
&= \lambda[[\texttt{L},\texttt{R}].\mathit{receive}(\lambda[(\texttt{ok},\texttt{Fork}).\mathit{Fun}(\texttt{ok\_f\_philo})\ [\texttt{L},\texttt{R},\texttt{Fork}],\\
&\qquad\qquad \texttt{no}.\mathit{Fun}(\texttt{no\_f\_philo})\ [\texttt{L},\texttt{R}]])]
\end{aligned}
$$

$$
\begin{aligned}
TP &= \lambda[[\![\texttt{L},\texttt{R}]\!]_{cons}, [\![]\!]_{and}.[\![\texttt{receive...}]\!]_{seq}]\\
&= \lambda[[\texttt{L},\texttt{R}].\mathit{receive}(\lambda[\texttt{hungry}.[\![\texttt{L!(R!\{get,self()\})}]\!]_E\,;[\![\texttt{hungry\_philo(L,R)}]\!]_E])]\\
&= \lambda[[\texttt{L},\texttt{R}].\mathit{receive}(\lambda[\texttt{hungry}.\mathit{send}(\texttt{let}\ x = [\![\texttt{R!\{get,self()\}}]\!]_E,\ \texttt{L})\,;x\,;\\
&\quad \mathit{Fun}(\texttt{hungry\_philo})\ [\texttt{L},\texttt{R}]])]\\
&= \lambda[[\texttt{L},\texttt{R}].\mathit{receive}(\lambda[\texttt{hungry}.\mathit{send}(\texttt{let}\ x = \mathit{send}(\texttt{let}\ y = (\texttt{get}, \mathit{ego}),\ \texttt{R})\,;y,\texttt{L})\,;x\,;\\
&\quad \mathit{Fun}(\texttt{hungry\_philo})\ [\texttt{L},\texttt{R}]])]
\end{aligned}
$$

$$
\begin{aligned}
OP &= \lambda[[\![\texttt{L},\texttt{R},\texttt{F}]\!]_{cons}, [\![]\!]_{and}.[\![\texttt{receive...}]\!]_{seq}]\\
&= \lambda[[\texttt{L},\texttt{R},\texttt{F}].\mathit{receive}(\lambda[(\texttt{ok},\ \_).[\![\texttt{L!(R!leave)}]\!]_E, \texttt{no}.[\![\texttt{F!leave}]\!]_E]\,;\\
&\quad [\![\texttt{self()!hungry}]\!]_E\,;[\![\texttt{think\_philo(L,R)}]\!]_E)]\\
&= \lambda[[\texttt{L},\texttt{R},\texttt{F}].\mathit{receive}(\lambda[(\texttt{ok},\ \_).\mathit{send}(\texttt{let}\,x = \mathit{send}(\texttt{let}\,y = \texttt{leave},\texttt{R});y,\texttt{L});x,\\
&\qquad\qquad \texttt{no}.\mathit{send}(\texttt{let}\,z = \texttt{leave},\texttt{F})\,;z]\,;\mathit{send}(\texttt{let}\,w = \texttt{hungry}, \mathit{ego})\,;\\
&\quad w\,;\mathit{Fun}(\texttt{think\_philo})\ [\texttt{L},\texttt{R}])]
\end{aligned}
$$

$$
\begin{aligned}
NP &= \lambda[[\![\texttt{L},\texttt{R}]\!]_{cons}, [\![]\!]_{and}.[\![\texttt{receive...}]\!]_{seq}]\\
&= \lambda[[\texttt{L},\texttt{R}].\mathit{receive}(\lambda[(\texttt{ok},\texttt{Fork}).[\![\texttt{Fork!leave}]\!]_E\,;[\![\texttt{self()!hungry}]\!]_E, \texttt{no}.[\![\texttt{true}]\!]_E]\,;\\
&\quad [\![\texttt{think\_philo(L,R)}]\!]_E)]\\
&= \lambda[[\texttt{L},\texttt{R}].\mathit{receive}(\lambda[(\texttt{ok},\texttt{Fork}).\mathit{send}(\texttt{let}\ x = \texttt{leave},\ \texttt{Fork})\,;x\,;\\
&\qquad\qquad\qquad \mathit{send}(\texttt{let}\ y = \texttt{hungry},\ \mathit{ego})\,;y,\\
&\qquad\qquad \texttt{no}.\mathit{true}]\,;\mathit{Fun}(\texttt{think\_philo})\ [\texttt{L},\texttt{R}])]
\end{aligned}
$$

L'environnement est donc :

$$
\begin{aligned}
\mathcal{E} \triangleq \{&\texttt{free\_fork}, \mathit{FF};\ \texttt{used\_ fork}, \mathit{UF};\ \texttt{thinking\_philo}, \mathit{TP};\ \texttt{hungry\_philo}, \mathit{HP}\\
&\texttt{ok\_f\_philo}, \mathit{OP};\ \texttt{no\_f\_philo}, \mathit{NP}\}
\end{aligned}
$$

Finalement, le programme est traduit en :

$$
\begin{aligned}
[\![P]\!] &= \mathcal{E},\ \star \triangleright [\![\texttt{Fork1 = ...}]\!]_{seq} \\
&= \mathcal{E},\ \star \triangleright ([\![\texttt{Fork1 = ...}]\!]_E ; [\![\texttt{Fork2 = ...}]\!]_E ; [\![\texttt{Philo1 = ...}]\!]_E ; [\![\texttt{Philo2 = ...}]\!]_E ; \\
&\quad [\![\texttt{Philo1!(Philo2!hungry)}]\!]_E) \\
&= \mathcal{E},\ \star \triangleright (\texttt{let Fork1} = [\![\texttt{spawn...}]\!]_E ; \texttt{let Fork2} = [\![\texttt{spawn...}]\!]_E ; \\
&\quad \texttt{let Philo1} = [\![\texttt{spawn...}]\!]_E ; \texttt{let Philo2} = [\![\texttt{spawn...}]\!]_E ; \\
&\quad \mathit{send}(\texttt{let } x = \mathit{send}(\texttt{let } y = \texttt{hungry},\ \texttt{Philo2}) ; y,\ \texttt{Philo1}) ; x) \\
&= \mathcal{E},\ \star \triangleright (\texttt{let Fork1} = \mathit{init}(\texttt{let } a = \mathit{new}, \lambda \mathit{ego}.(\mathit{Fun}(\texttt{free\_fork})\ \mathit{nil})) ; a; \\
&\quad \texttt{let Fork2} = \mathit{init}(\texttt{let } b = \mathit{new}, \lambda \mathit{ego}.(\mathit{Fun}(\texttt{free\_fork})\ \mathit{nil})) ; b ; \\
&\quad \texttt{let Philo1} = \mathit{init}(\texttt{let } c = \mathit{new}, \lambda \mathit{ego}.(\mathit{Fun}(\texttt{think\_philo})[\texttt{Fork1},\ \texttt{Fork2}])) ; c ; \\
&\quad \texttt{let Philo2} = \mathit{init}(\texttt{let } d = \mathit{new}, \lambda \mathit{ego}.(\mathit{Fun}(\texttt{think\_philo})[\texttt{Fork2},\ \texttt{Fork1}])) ; d ; \\
&\quad \mathit{send}(\texttt{let } x = \mathit{send}(\texttt{let } y = \texttt{hungry},\ \texttt{Philo2}) ; y,\ \texttt{Philo1}) ; x)
\end{aligned}
$$

Après la description de la syntaxe de la partie fonctionnelle associée à ERLANG ainsi que la traduction d'un programme μErlang en une configuration, nous pouvons définir la sémantique de μErlang en nous basant sur la sémantique des configurations.

7.6 La sémantique de μErlang

La sémantique d'un programme μErlang est décrite par celle de sa traduction dans le formalisme des configurations. Dans la section précédente, nous avons vu qu'un programme est traduit en un acteur anonyme $\star \triangleright e$ qui est ensuite évalué en appliquant les règles de la sémantique des configurations. Cependant, certaines notions avaient été laissées en suspens lors de l'introduction de la sémantique des configurations. Afin de compléter la sémantique de μErlang, nous devons définir la sémantique de $\mathcal{F}unc_2$. Nous précisons dans cette section la notion de variable et de substitution, le filtrage, la notion de nom libre et enfin, la sémantique donnée à la boîte aux lettres d'un acteur.

Variables et filtrage

Lors de l'exécution d'un programme de $\mathcal{F}unc_2$, les variables sont *remplacées* au fur et à mesure de l'avancée du calcul par des valeurs sémantiques au moyen de substitutions.

Définition 7.3 (Substitution et composition) :
Une ***substitution*** $[v/x]$ *est un automorphisme sur les expressions. Elle remplace toutes les occurrences libres de la variable x par la valeur v. Elle est définie par induction sur la forme des expressions par les règles 7.3. Nous définissons de plus, deux substitutions particulières failed et [] qui n'ont aucun effet. La première donne un sens aux filtrages qui ont échoué. Et la seconde correspond à la substitution identité. Enfin, l'opérateur de composition de substitutions est défini comme l'usuelle composition d'applications avec la règle supplémentaire suivante : $\sigma \circ \mathit{failed} \triangleq \mathit{failed} \circ \sigma \triangleq \mathit{failed}$.*

La variable *ego* a un statut un peu particulier puisque contrairement aux autres variables, elle a une portée totalement dynamique. Ainsi, par exemple, elle peut être libre

$$
\begin{array}{lll}
[v/x]\_ & = \_ & [v/x]C(e_1,\ldots,e_n) = C([v/x]e_1,\ldots,[v/x]e_n) \\
[v/x]a & = a & [v/x]e_1\ e_2 \quad\ = [v/x]e_1\ [v/x]e_2 \\
[v/x]c & = c & [v/x]e_1\,;e_2 \quad\ = [v/x]e_1\,;[v/x]e_2 \\
[v/x]f & = f & [v/x]\texttt{let}\ p\ \texttt{=}\ e \quad\ = \texttt{let}\ [v/x]p\ \texttt{=}\ [v/x]e \\
[v/x](e') & = ([v/x]e') & [v/x]\lambda[p_i, g_i.e_i]^{i\in I} = \lambda[[v/x]p_i, [v/x]g_i.[v/x]e_i]^{i\in I} \\
[v/x]x' & = \begin{cases} v & \text{si } x' = x \\ x' & \text{sinon} \end{cases} & [v/x]Fun(e) \quad\ = \begin{cases} Fun_v([v/x]e) & \text{si } x = ego \\ Fun([v/x]e) & \text{sinon} \end{cases}
\end{array}
$$

Règles 7.3 Substitution d'une variable par sa valeur dans $\mathcal{F}unc_2$.

dans une fonction globale et donc être instanciée par l'adresse du processus courant lors de l'appel de cette fonction. Pour cela, nous mémorisons dans chaque *Fun* la valeur de l'*ego* lors de sa définition. Cette valeur sera substituée à *ego* dans le corps de la fonction lors de l'exécution.

Remarques :

- *Le mécanisme de traduction supprime tous les conflits de captures de variables possibles, ainsi, toutes les apparitions de variables dans les motifs sont non-liantes (une variable n'est définie que si elle n'existe pas déjà). Donc, lors de la substitution, les variables figurant dans les motifs sont systématiquement remplacées.*
- *La substitution étant identique sur les motifs et sur les expressions de même forme, nous ne donnons que les règles sur les expressions ainsi que celle sur le joker.*

Le langage μErlang utilise un filtrage plus complexe que celui de ML-Act puisqu'il n'est pas linéaire. Cette propriété impose l'utilisation d'un opérateur de composition stricte des substitutions qui vérifie que les deux substitutions coïncident sur l'intersection de leur domaine.

Définition 7.4 (Composition stricte) :
L'opérateur de composition stricte de substitutions est noté $\oplus$. C'est un opérateur commutatif et associatif dont l'élément neutre est l'identité $[]$. Il est défini par les règles suivantes :

$$
\begin{array}{ll}
\sigma \oplus \mathit{failed} \triangleq \mathit{failed} & \\
\sigma_1 \oplus \sigma_2 & \triangleq \begin{cases} \sigma_1 \circ \sigma_2 & \mathit{si}\ \sigma_1|_{dom(\sigma_1)\cap dom(\sigma_2)} = \sigma_2|_{dom(\sigma_1)\cap dom(\sigma_2)} \\ \mathit{failed} & \mathit{sinon} \end{cases}
\end{array}
$$

Le filtrage utilise l'opérateur de *confrontation*, noté $v/p \Rightarrow \sigma$, il prend en paramètre une valeur sémantique v et un motif p et calcule une substitution permettant l'unification de v avec p. Les règles de déduction 7.4 définissent inductivement le filtrage selon la forme du motif. Le filtrage par le motif joker ou par une variable réussit et renvoie respectivement la substitution vide ou celle qui remplace la variable par la valeur filtrée. Le filtrage d'une valeur constante (adresse, constante, fonction ou *bif*) ne réussit que si le filtre lui est égal, auquel cas la substitution obtenue est vide. Le filtrage d'un terme composé exige

$$\frac{}{v/\_ \Rightarrow []} \qquad \frac{}{v/x \Rightarrow [v/x]} \qquad \frac{v \in \mathbb{A} \cup \mathbb{C} \cup \mathcal{F} \cup \mathcal{F}_c}{v/v \Rightarrow []} \qquad \frac{v_1 \in \mathbb{A} \cup \mathbb{C} \cup \mathcal{F} \cup \mathcal{F}_c \quad v_1 \neq v_2}{v_1/v_2 \Rightarrow \mathit{failed}}$$

$$\frac{\forall k\ \forall i\ \forall v_i \quad v \neq C(v_1, \ldots, v_k)}{v/C(p_1, \ldots, p_n) \Rightarrow \mathit{failed}} \qquad \frac{\forall i \quad v_i/p_i \Rightarrow \sigma_i}{C(v_1, \ldots, v_n)/C(p_1, \ldots, p_n) \Rightarrow \sigma_1 \oplus \cdots \oplus \sigma_n}$$

Règles 7.4 Le filtrage de μErlang.

que le constructeur de tête du motif soit identique au constructeur de tête de la valeur et que chaque sous-motif filtre la sous-valeur correspondante. La composition stricte des substitutions en résultant permet d'imposer à deux variables identiques figurant dans le motif d'avoir la même valeur.

Les contextes d'évaluation

Afin de présenter l'évaluation fonctionnelle, nous utilisons la notion de contexte d'évaluation déjà introduite dans le cadre de ML-Act. En μErlang, l'application de fonction utilise l'appel par valeur – *i.e.* les arguments sont tous évalués avant d'exécuter le corps de la fonction. Cependant, comme cela a été évoqué dans la section 7.3, les paramètres d'une fonction et plus généralement les séquences d'expressions sont évalués dans un ordre inconnu.

Définition 7.5 (Contexte de μErlang) :
Parmi les termes construits par la grammaire suivante, les contextes de μErlang sont ceux qui ne contiennent qu'un seul trou :

$$\begin{aligned} \mathrm{C} &::= [] \mid (\mathrm{C}) \mid C(\mathrm{A}) \mid \mathrm{C}\,; e \mid \mathrm{C}\ e \mid v\ \mathrm{C} \mid \mathit{Fun}(\mathrm{C}) \mid \texttt{let}\ p = \mathrm{C} \\ \mathrm{A} &::= [] \mid \mathrm{A}, e \mid e, \mathrm{A} \end{aligned}$$

Remarque :
La forme du contexte d'évaluation sur les séquences d'expressions permet d'obtenir une réduction indéterministe puisque tous les ordres sont possibles.

Le calcul fonctionnel

L'exécution fonctionnelle des programmes $\mathcal{F}unc_2$ se déduit par application des règles 7.5. Les trois premières règles généralisent la notion de filtrage pour prendre en compte les gardes. En effet, pour qu'un filtrage réussisse, il faut que le motif filtre la donnée mais également que la garde soit vraie dans l'environnement résultant. Contrairement à la réduction des termes de $\mathcal{F}unc_1$, l'évaluation des expressions dépend du contexte. En effet, lors de la définition d'une variable, on doit appliquer la substitution issue du filtrage à l'intégralité du terme. Comme toute variable introduite est substituée à l'exécution par la valeur à laquelle elle est liée, aucune variable ne peut plus figurer dans une expression en cours d'évaluation (VarE). L'application d'une valeur non fonctionnelle (AppE1) ou

$$\frac{v/p \Rightarrow \mathit{failed}}{v/p, g \Rightarrow \mathit{failed}} \qquad \frac{v/p \Rightarrow \sigma \quad \sigma(g) \longrightarrow_e \mathit{false}}{v/p, g \Rightarrow \mathit{failed}} \qquad \frac{v/p \Rightarrow \sigma \quad \sigma(g) \longrightarrow_e \mathit{true}}{v/p, g \Rightarrow \sigma}$$

$$\textsc{VarE}: \frac{}{\mathrm{C}[x] \longrightarrow_e \mathbf{Err}} \qquad \textsc{AppE1}: \frac{v \notin \mathcal{F} \cup \mathcal{F}_c}{\mathrm{C}[v\ e] \longrightarrow_e \mathbf{Err}} \qquad \textsc{App}: \frac{v/p_1, g_1 \Rightarrow \sigma}{\mathrm{C}[\lambda[p_i, g_i.e_i]^{i\in 1..n}\ v] \longrightarrow_e \sigma(\mathrm{C}[e_1])}$$

$$\textsc{AppE2}: \frac{}{\mathrm{C}[\lambda[]\ v] \longrightarrow_e \mathbf{Err}} \qquad \textsc{AppF}: \frac{v/p_1, g_1 \Rightarrow \mathit{failed}}{\mathrm{C}[\lambda[p_i, g_1.e_i]^{i\in 1..n}\ v] \longrightarrow_e \mathrm{C}[\lambda[p_i, g_i.e_i]^{i\in 2..n}\ v]}$$

$$\textsc{Seq}: \frac{}{\mathrm{C}[v\,; e] \longrightarrow_e \mathrm{C}[e]} \qquad \textsc{Let}: \frac{v/p \Rightarrow \sigma}{\mathrm{C}[\texttt{let}\ p = v] \longrightarrow_e \sigma(\mathrm{C}[v])} \qquad \textsc{LetE}: \frac{v/p \Rightarrow \mathit{failed}}{\mathrm{C}[\texttt{let}\ p = v] \longrightarrow_e \mathbf{Err}}$$

$$\textsc{FunEgo}: \frac{f \in \mathit{dom}(\mathcal{E})}{\mathrm{C}[\mathit{Fun}_v(f)] \longrightarrow_e \mathrm{C}[[v/\mathit{ego}]\mathcal{E}(f)]} \qquad \textsc{Fun}: \frac{f \in \mathit{dom}(\mathcal{E})}{\mathrm{C}[\mathit{Fun}(f)] \longrightarrow_e \mathrm{C}[\mathcal{E}(f)]}$$

$$\textsc{FunE1}: \frac{v \notin \mathit{dom}(\mathcal{E})}{\mathrm{C}[\mathit{Fun}_{v'}(v)] \longrightarrow_e \mathbf{Err}} \qquad \textsc{FunE2}: \frac{v \notin \mathit{dom}(\mathcal{E})}{\mathrm{C}[\mathit{Fun}(v)] \longrightarrow_e \mathbf{Err}}$$

Règles 7.5 Les réductions fonctionnelles en μErlang.

d'une fonctionnelle vide[54] (AppE2) provoque également une erreur. Lors de l'application d'une fonction qui comporte plusieurs branches, celles-ci sont examinées dans l'ordre de définition, le premier motif qui filtre la valeur passée en paramètre fournit la branche qui est exécutée. Pour représenter ce mécanisme, on tente de filtrer la valeur par le premier motif, deux cas se présentent alors :

- le filtrage échoue (AppF), le filtrage est alors poursuivi en appliquant la fonction privée de son premier cas de filtrage à la valeur ;
- il réussit (App) et le résultat est alors le corps du cas correspondant que l'on replace dans le contexte.

Ensuite, la substitution résultant du filtrage est appliquée à tout le contexte. Ainsi, la valeur est exportée hors de sa portée statique. Par exemple :

$$\lambda[1.\texttt{let}\ x = 2]\ 1\,; x \longrightarrow_e [](\texttt{let}\ x = 2\,; x) \longrightarrow_e [2/x](2\,; x) \longrightarrow_e 2$$

La définition est similaire à l'application d'une fonction comportant un seul cas. Soit, le filtrage réussit (Let) et la substitution résultante est appliquée au contexte après y avoir replacé la valeur définie. Soit, il échoue (LetE) et le résultat du calcul est une erreur.

Enfin, les quatre dernières règles concernent la réduction des appels de fonctions globales par *Fun*. Ces quatre règles référencent l'environnement des fonctions globales (noté $\mathcal{E}$) qui a été obtenu par la traduction. Durant la traduction, chaque *Fun* aurait pu être étiquetée par cet environnement, mais cela compliquant inutilement les termes obtenus, nous supposons que cette reférence est globale. Si l'argument de cet appel est un atome et qu'il figure dans $\mathcal{E}$, la valeur de la fonction à laquelle il est rattaché remplace l'appel

54. Cette possibilité correspond à la fin d'un filtrage dans le cas où aucun filtre ne convient.

$$\textsc{Bal1:}\ \frac{\exists j \quad (\forall i < j \ \ m_i/p_1, g_1 \Rightarrow \mathit{failed}) \quad m_j/p_1, g_1 \Rightarrow \sigma}{\lambda[p_i, g_i.e_i]^{i\in 1..n} \prec (m_i)^{i\in J} \Longrightarrow (m_i)^{i\in J\setminus\{j\}},\ e_1,\ \sigma}$$

$$\textsc{Bal2:}\ \frac{(\forall i \in J \ \ m_i/p_1, g_1 \Rightarrow \mathit{failed}) \quad \lambda[p_i, g_i.e_i]^{i\in 2..n} \prec (m_i)^{i\in J} \Longrightarrow q,\ e,\ \sigma}{\lambda[p_i, g_i.e_i]^{i\in 1..n} \prec (m_i)^{i\in J} \Longrightarrow q,\ e,\ \sigma}$$

Règles 7.6 Les boîtes aux lettres de μErlang.

(règles FunEgo et Fun). Si l'appel est dans la portée d'un processus, *ego* a été défini, nous propageons alors sa définition dans le corps de la fonction (FunEgo). Enfin, si la valeur calculée n'est pas un atome ou si c'est un atome ne représentant pas une fonction globale, le programme conduit à une erreur (FunE1 et FunE2).

Les règles concernant les constantes fonctionnelles (par exemple les opérations arithmétiques) ne sont pas données ici. En effet, les opérateurs binaires et unaires se comportent de manière classique en se réduisant selon les δ-règles usuelles.

La sémantique des boîtes aux lettres

Comme nous l'avons précisé dans le chapitre qui a introduit les configurations et leur sémantique, les boîtes aux lettres ont une sémantique très différente en ML-Act et en Erlang. En μErlang, chaque message de la queue $((m_i)^{i\in J})$ d'un acteur a est filtré par le premier motif de l'interface courante de a lors d'un *receive*. Le premier message m_j qui convient est utilisé (règle Bal1). Puis, si aucun message de la queue ne convient, le filtrage se poursuit avec le motif suivant (règle Bal2). Ce comportement est décrit par les règles 7.6 dans lesquelles, nous supposons que l'ensemble J indexant la boîte aux lettres est non vide. Remarquons que, si aucun message n'est acceptable, aucune réduction de la boîte aux lettres n'est possible, ce qui bloque l'acteur en attente.

7.7 Conclusion

Dans ce chapitre, nous avons présenté succinctement le langage Erlang en insistant sur le noyau μErlang que nous avons étudié plus en détail. Nous avons défini la sémantique de μErlang en donnant sa traduction dans le formalisme des configurations introduit au premier chapitre. Pour cela, la partie fonctionnelle d'Erlang a été restreinte à un λ-calcul avec constante ($\mathcal{F}unc_2$) dont nous avons explicité la sémantique. Cette optique permet de simplifier grandement l'exposé de la sémantique de μErlang, mais surtout, elle nous permet de formaliser plus simplement le système de type que nous utilisons pour détecter les erreurs dans un programme écrit en μErlang. Le principal intérêt de cette traduction apparaît au chapitre suivant.

Dans le cadre de cette thèse, j'ai réalisé un prototype d'analyseur de programme Erlang dont les bases théoriques sont introduites dans le chapitre suivant. Mais, pour introduire notre système de type et surtout le prouver, nous avions besoin d'une sémantique d'Erlang. Or, peu de sémantiques formelles ont été définies pour Erlang. À notre

connaissance, seuls deux travaux s'y sont attaqués. Remarquons que ces deux études ont également pour but la vérification statique des programmes ERLANG.

- La première sémantique est due M. DAM et al. (voir [DåF98]). Elle est conçue dans le but de démontrer la validité d'une analyse par *model checking* au moyen d'une logique temporelle. Les auteurs construisent donc une sémantique opérationnelle d'ERLANG définie par un système de transition étiqueté. La sémantique obtenue (fortement inspirée de celle de CCS et du π-calcul) est complexe et ne se prête pas bien à la preuve de correction de notre système. Notons également que dans cette sémantique, la communication se fait par synchronisation directe entre l'envoyeur et la queue du destinataire. De plus, cette sémantique ne suit pas du tout les règles de portée des identificateurs d'ERLANG.
- La seconde sémantique est due à F. HUCH dans [Huc99]. L'auteur présente une interprétation abstraite issue de cette sémantique qui permet de la rendre finie et donc de pouvoir vérifier un programme sur la base d'une logique LTL par *model checking*. La sémantique opérationnelle introduite consiste également en un système de transition étiqueté. Elle est nettement plus simple et intuitive que la précédente mais suit la même contrainte de synchronisation lors de l'envoi de message et ne respecte pas tout a fait la règle de portée des identificateurs d'ERLANG.

Nos travaux pour construire une sémantique opérationnelle formelle d'ERLANG sont principalement inspirés des travaux que nous avons réalisés dans le cadre de ML-ACT. La sémantique introduite dans ce chapitre est donc dérivée de celle des configurations. La confrontation aux autres sémantiques d'ERLANG a permis de la valider. Dans le cadre de nos travaux, la sémantique du langage n'est utilisée que pour démontrer la correction de notre système de type, il est donc suffisant de construire une sémantique opérationnelle basé sur un système de transition.

Remarquons, qu'écrire une sémantique formelle complète du langage ERLANG est une tâche assez complexe. Il faudrait, en effet :

- ajouter la notion de site. Pour cela, les configurations doivent être augmentées par une construction de la forme $s\{w\}$ pour exprimer le fait que la configuration w s'exécute sur le site s.
- gérer le chargement dynamique de code. Chaque site doit contenir l'environnement des fonctions globales accessibles et les valeurs de ces fonctions peuvent changer. Le site devient $s\{\mathcal{E} \mid w\}$.
- permettre l'envoi de messages entre sites. Le destinataire du message peut être local en gardant la même syntaxe ou situé sur le site s par $a@s \triangleleft m$.
- intégrer la notion de temps. En ERLANG, l'instruction de réception des messages (`receive`) contient une clause `after` qui permet d'interrompre le cours du calcul au bout d'un certain temps. Il faudrait donc probablement intégrer une notion de compteur à chaque exécution de processus.
- ajouter une notion de nom symbolique et de dictionnaire. Un service peut être abstrait du processus l'exécutant en lui associant un nom. Ce nom est déclaré représentant un certain processus (cette valeur peut changer). Chaque site doit donc maintenir un tel dictionnaire (en fait un environnement) et donc un site devient $s\{\mathcal{E}_f \mid \mathcal{E}_n \mid w\}$.
- rajouter la gestion des signaux. ERLANG intègre une notion de signal pour gérer les

exceptions et leur propagation au sein des processus. Pour cela, il suffit d'étendre l'envoi classique de message en ajoutant un drapeau pour qu'un processus puisse distinguer si c'est un message ou un signal qu'il reçoit.

Les points ci-dessus sont les principaux problème à résoudre mais ne sont pas les seuls. Le lecteur intéressé par la sémantique précise des caractéristiques concernant la communication et la répartition en ERLANG peut consulter les chapitres 10, 11 et 12 de [BV99] qui, s'ils ne contiennent pas une description formelle, essaient de procéder de manière la plus systématique possible.

Notre sémantique et les deux introduites ci-dessus pourraient servir de point de départ à d'éventuels travaux dans ce sens. De plus, cette sémantique n'a pas été implantée car nous ne réalisons pas un compilateur complet (comme dans le cadre de ML-ACT) mais uniquement un analyseur de programmes.

Chapitre 8

Le typage d'Erlang

Après avoir confronté les méthodes de typage développées au sein de notre équipe à la pratique de ML-Act qui est un langage expérimental, nous avons décidé de tenter de les adapter à un langage industriel. Cette confrontation sert deux objectifs : en premier lieu, il s'agit de tester les limites de faisabilité de ces méthodes ainsi que leur utilité; en second lieu, l'application de ces systèmes de type à des exemples réels doit permettre de mieux cibler les futurs travaux nécessaires à leurs extensions. Cette thèse a principalement répondu au premier objectif, celui d'une faisabilité pratique d'un système de type pour des langages industriels assez complexes et sur lesquels nous n'avons aucune possibilité de modification du langage lui-même. Ce chapitre introduit un système de type pragmatique qui étend le système de type conçu pour ML-Act. Le plan de ce chapitre est proche de celui présentant le typage de ML-Act. Une première section discute de certains choix effectués (par exemple sur la forme des messages) pour simplifier le système de type. Puis, dans une deuxième section, nous présentons les erreurs que nous souhaitons détecter dans les programmes Erlang. Ces erreurs sont sensiblement différentes des erreurs détectées dans le cadre de ML-Act car l'approche adoptée ici est celle du typage souple. Puis, nous définissons précisément les types utilisés, ils diffèrent sensiblement de ceux utilisés pour ML-Act car ils sont plus précis et intègrent un calcul d'effet pour la partie interface des acteurs. Ensuite, nous présentons le système de type et démontrons sa validité. Enfin, nous concluons par une présentation rapide du prototype réalisé ainsi qu'une discussion sur les extensions possibles de notre système.

8.1 Une première analyse de l'approximation

La sémantique de μErlang contient un certain nombre de particularités par rapport à celle de ML-Act. Certaines ne compliquent que très peu le système de type. Par exemple, les filtrages contiennent une notion de garde dont l'approximation est proche de celle des expressions. D'autres changent beaucoup plus fondamentalement les types nécessaires : l'accès aux boîtes aux lettres est synchrone, le filtrage est hétérogène, le filtrage est dynamique et la forme des messages est quelconque. Présentons plus précisément l'impact sur le système de type de ces trois différences.

Interface

Le langage ERLANG ne possède pas de réelle notion de comportement et de changement de comportement. En effet, l'accès d'un acteur à sa boîte aux lettres est synchrone, l'expression *receive* récupère un message traitable effectue éventuellement un calcul sur ce message et renvoie le résultat de ce calcul. Ainsi, chaque expression peut contenir un accès à la boîte aux lettres. Nous utilisons un calcul d'effet de la même forme que [LG88], [Wri92], [TJ94] ou [Dag97] pour collecter les différentes interfaces d'accès à la boîte aux lettres. Le typage des expressions calcule donc un effet en plus du type et cet effet est stocké dans les types fonctionnels lors de l'abstraction. Par exemple, une fonction qui a le type $t_1 \xrightarrow{I} t_2$ prend en argument un objet de type t_1 renvoie un objet de type t_2 et a pour effet d'accéder au moins une fois à la boîte aux lettres de l'acteur qui l'exécute avec l'interface I. Remarquons que des *règles de bonne programmation* conseillent d'expliciter les comportements des processus en utilisant des fonctions dont le corps est uniquement un *receive* (voir par exemple [WA96] et [AVWW96] pages 73 et 74). L'exemple des philosophes et des fourchettes du chapitre 7 (exemples 7.2 et 7.3 page 159 et 160) suit cette règle.

Filtrage hétérogène

Les filtrages peuvent être hétérogènes et le type d'une abstraction possède *a priori* plusieurs bornes inférieures dont les types sont radicalement différents. La politique de filtrage adoptée sur ML-ACT doit donc être abandonnée ici. En effet, l'unification entre les types des différents filtres n'a plus de sens puisque ceux-ci peuvent avoir des structures très différentes. De plus, pour garder suffisamment de précision et donner des types concurrents pertinents, l'approximation du type fonctionnel $(\alpha_1 \to \alpha_2) \sqcup (\beta_1 \to \beta_2)$ en $(\alpha_1 \sqcap \beta_1) \to (\alpha_2 \sqcup \beta_2)$ n'est plus acceptable. Dans le contexte des langages dynamiques comme ERLANG, il convient de conserver une liaison précise entre le type de l'argument et celui du résultat obtenu. Pour cela, A. AIKEN et al. utilisent un type dit *conditionnel* qui approxime le filtrage dans le langage des types. Ainsi, ils réintroduisent dans les types une forme d'analyse de flot de contrôle qui est nécessaire à l'analyse statique des langages typés dynamiquement (on pourra consulter leurs travaux initiaux [AM91] pour une discussion plus précise du problème). Ce type conditionnel a été introduit dans [AWL94] sous l'écriture $t_1?t_2$. Informellement, la sémantique de ce type est : si le type t_2 n'est pas *nul* (égal à $\bot$) alors le type vaut t_1 sinon il est nul ($\bot$), soit :

$$t_1?t_2 \triangleq \begin{cases} t_1 & \text{si } t_2 \neq \bot \\ \bot & \text{sinon} \end{cases}$$

Ce type est utilisé pour approximer les filtrages hétérogènes des langages fonctionnels *à la* SCHEME, par exemple le choix :

```
case e of true -> 1; false -> autre                    (case)
```

est approximé par $(int?(t_e \sqcap true)) \sqcup (autre?(t_e \sqcap false))$ avec t_e type de e. Dans le cadre de cette thèse, nous n'utilisons pas un type conditionnel mais une notion de contrainte

conditionnelle $c_1 \Rightarrow c_2$ dont le sens est : si c_1 est vérifiée alors c_2 doit l'être. Cette contrainte est générée lors de l'appel d'une fonction f et permet de relier précisément les contraintes sur le type de retour et l'effet de f en fonction du type de l'entrée sur le modèle du type conditionnel. Une contrainte entre types fonctionnels $t_1 \to t_2 \sqsubseteq t'_1 \to t'_2$ est alors décomposée en $t'_1 \sqsubseteq t_1 \Rightarrow t_2 \sqsubseteq t'_2$. Ce qui signifie que si t'_1 est sous-type de t_1 alors la branche du filtrage est éventuellement empruntée. Une contrainte globale (sur les types de tous les motifs) permet de vérifier que l'application est correcte. En effet, dans le cadre de μErlang, l'exemple (`case`) précédent (et, en général, tous les choix) est traduit en fonction filtrante à plusieurs cas :

$$\lambda[\mathit{true} \to 1;\ \mathit{false} \to \mathit{autre}]\ e$$

Le typage de l'abstraction conduit au type $(\mathit{true} \to \mathit{int}) \sqcup (\mathit{false} \to \mathit{autre}) \sqsubseteq t_e \to t_r$ et celui de l'application à $(\mathit{true} \to \mathit{int}) \sqcup (\mathit{false} \to \mathit{autre}) \sqsubseteq t_e \to t_r$ et $t_e \sqsubseteq \mathit{dom}((\mathit{true} \to \mathit{int}) \sqcup (\mathit{false} \to \mathit{autre}))$. Une fonction sur les types *dom* est définie et renvoie $\sqcup_i t_i$ si son argument vaut $\sqcup_i (t_i \to t'_i)$ et sinon provoque une erreur (sur les types clos non fonctionnels). Les contraintes se réduisent alors par décomposition des types unions en :

$$\begin{cases} \mathit{true} \to \mathit{int} \sqsubseteq t_e \to t_r \\ \mathit{false} \to \mathit{autre} \sqsubseteq t_e \to t_r \\ t_e \sqsubseteq \mathit{true} \sqcup \mathit{false} \end{cases} \quad \text{soit} \quad \begin{cases} t_e \sqsubseteq \mathit{true} \Rightarrow \mathit{int} \sqsubseteq t_r \\ t_e \sqsubseteq \mathit{false} \Rightarrow \mathit{autre} \sqsubseteq t_r \\ t_e \sqsubseteq \mathit{true} \sqcup \mathit{false} \end{cases}$$

Si la forme de t_e est connue, le système peut être simplifié. Mais, si aucune information supplémentaire n'est connue sur le type t_e alors, le système de contraintes se décompose en deux systèmes (car t_e est une conjonction de deux types) :

- Si $t_e \sqsubseteq \mathit{true}$ alors la deuxième contrainte disparaît et la troisième est trivialement vérifiée, le système se réduit alors en $t_e \sqsubseteq \mathit{true}$ et $\mathit{int} \sqsubseteq t_r$.
- Si $t_e \sqsubseteq \mathit{false}$ alors la première contrainte disparaît et la troisième est trivialement vérifiée, le système se réduit alors en $t_e \sqsubseteq \mathit{false}$ et $\mathit{autre} \sqsubseteq t_r$.

L'appel d'une fonction avec un argument de type inconnu doit être bien typé pour tous les types possibles d'arguments. Ainsi, les deux systèmes obtenus doivent être solubles pour que le programme soit typable. Notons que les types fonctionnels ne peuvent pas être quelconques, lors de la construction du type d'une abstraction, nous rendons les types des différents filtres disjoints et nous fusionnons les cas traitant des données de même forme. Ainsi, chaque branche de l'union ne peut être vraie que quand les autres ne le sont pas et au moins une des branches est vraie du fait de la contrainte sur le domaine. Pour effectuer cette opération, nous définissons une notion de différence de type (voir page 209). De plus, afin d'attribuer des types suffisamment précis aux filtrages, nous considérons que chaque constante possède son propre type. Ainsi, par exemple, 1 a pour type 1 qui est sous-type des entiers $1 \sqsubseteq \mathit{int}$.

Filtrage dynamique

Le filtrage dynamique apporte au système une difficulté supplémentaire. En effet, une variable n'a pas du tout le même sens selon qu'elle est l'occurrence de liaison, ou non. Analysons l'exemple suivant :

$$(a) \qquad \texttt{let G} = \lambda\texttt{F}.(\lambda[\texttt{F.ok}\ ;\ \texttt{\_.no}]\ 1)\ ;(\texttt{G}\ 1,\ \texttt{G}\ 2)$$

Il se réduit en :

$$(\lambda[1.\texttt{ok}\ ;\ \_.\texttt{no}]\ 1,\ \lambda[2.\texttt{ok}\ ;\ \_.\texttt{no}]\ 1)$$

Et donc, le calcul se termine par :

$$(\texttt{ok},\ \texttt{no})$$

Or, usuellement le typage d'un appel de fonction impose aux types des paramètres réels d'être compatibles avec toutes les contraintes dues à leurs utilisations dans le corps de la fonction. Pour cela, dans le corps de la fonction, on collecte des contraintes de la forme $\alpha \sqsubseteq t$ où α est le type du paramètre. Et à chaque appel, des contraintes de la forme $t' \sqsubseteq \alpha$ apparaissent. Par transitivité, la compatibilité est vérifiée par $t' \sqsubseteq t$. En effet, les appels fournissent les valeurs possibles des paramètres pour toutes les utilisations dans le corps de la fonction.

Si un paramètre est utilisé dans un motif, un filtre différent sera construit à chaque appel. Dans l'exemple précédent, le typage du corps de la fonction `G` produit le type $\alpha \to t$ et les contraintes $\{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \setminus \alpha) \Rightarrow \texttt{no} \sqsubseteq t\}$. On collecte donc d'autres contraintes de « définition » (de la forme $\beta \sqsubseteq \alpha$) sur le paramètre `F`. Dans le cas présent, il n'est pas possible de conclure sur un type précis avec les règles usuelles qui conduisent au type $(\texttt{ok} \sqcup \texttt{no}) \times (\texttt{ok} \sqcup \texttt{no})$. Le problème devient plus critique si la partie qui filtre tout (avec le joker) ne figure pas dans le programme :

$$(b) \qquad \texttt{let G} = \lambda\texttt{F}.((\lambda\texttt{F.ok})\ 1)\ ;(\texttt{G}\ 1,\ \texttt{G}\ 2) \longrightarrow_e ((\lambda 1.\texttt{ok})\ 1,\ (\lambda 2.\texttt{ok})\ 1) \longrightarrow_e \textbf{Err}$$

Le typage conduit à l'ensemble de contraintes $\{1 \sqsubseteq \alpha;\ \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq \alpha;\ 2 \sqsubseteq \alpha\}$ qui possède des solutions. On ne peut donc pas détecter cette erreur d'exécution par un système usuel.

Pour résoudre ce problème, lors de la phase de renommage des variables décrites dans le chapitre précédent, nous marquons les occurrences de liaison des variables qui apparaissent dans d'autres filtres (`F` devient $\texttt{F}^*$ dans le motif). Sur les deux exemples précédents, le $\lambda\texttt{F}$ le plus externe devient $\lambda\texttt{F}^*$. Une variable ainsi étiquetée se voit attribuer un type α qui sera également marqué (α^*).

Intuitivement le procédé adopté construit, pour chaque appel d'une telle fonction, une nouvelle instance du type de cette fonction (par une fonction notée **inst** qui sera définie précisément plus tard). En effet, le corps de la fonction dépend des arguments et sera donc différent à chaque appel. Ainsi, l'appel d'une fonction dont le type t_f contient des variables étiquetées α_i^*, provoque la création d'une instanciation β_i des α_i. Le type utilisé pour l'application devient alors $[\beta_i/\alpha_i]t_f$ et les contraintes qui contiennent une variable α_i sont recopiées en y substituant α_i par β_i . Cette approche est voisine du polymorphisme par nom introduit dans le sixième chapitre de la thèse de X. LEROY ([Ler92]). Nous limitons cependant cette instanciation aux variables étiquetées.

Sur cette instance, chaque variable de type étiquetée doit être égale à la portion de type qui lui correspond dans le type des arguments. En effet, lors de l'appel, une variable prend la valeur qui lui correspond dans le paramètre. Cette nouvelle valeur construit alors de nouveaux filtres. Par exemple, dans le programme ci-dessus, l'appel `G` 1 instancie le corps de `G` en $(\lambda 1.\texttt{ok})$ 1 et celui de `G` 2 en $(\lambda 2.\texttt{ok})$ 1. Donc le type de la fonctionnelle qui utilise la variable étoilée devient $t \to \texttt{ok}$ si t est le type de l'argument de `G` lors de son appel (ici $t = 1$ et $t = 2$).

De plus, le résultat d'une fonction dont un des arguments est étoilé l'est lui-même. En effet, chaque appel introduit des contraintes sur le résultat qui ne sont valables que pour cet appel. Par exemple, la première version de la fonction `G` conduit à $(\lambda[1.\texttt{ok};\ \_.\texttt{no}])\ 1$ pour `G` 1 et à $(\lambda[2.\texttt{ok};\ \_.\texttt{no}])\ 1$ pour `G` 2. Or, le typage de ces deux fonctions conduit respectivement à `ok` et à `no`. Donc, si l'on souhaite conserver un type suffisamment précis, il faut que chaque appel génère des types résultats différents. Il faut donc aussi instancier le type résultat. Notons que dans ce cadre, l'égalité n'est pas indispensable, nous avons cependant préféré utilisé la notion d'étoile des arguments plutôt que de créer une classe de variables différente.

Par exemple, le typage du programme (b) attribue le type t_G à `G` avec les contraintes $C = \{1 \sqsubseteq \alpha;\ \texttt{ok} \sqsubseteq t;\ \alpha^* \to t^* \sqsubseteq t_G\}$. Ainsi, la première application instancie α et t par α_1 et t_1 et donc ajoute à C les contraintes $\{1 \sqsubseteq dom(\mathbf{inst}(\mathbf{t_G})_\mathbf{1});\ \mathbf{inst}(\mathbf{t_G})_\mathbf{1} \sqsubseteq 1 \to \beta_1\}$ où β_1 est le type du résultat. Puis, si β_2 est le type du résultat de la seconde application, elle conduit à l'ajout de $\{2 \sqsubseteq dom(\mathbf{inst}(\mathbf{t_G})_\mathbf{2});\ \mathbf{inst}(\mathbf{t_G})_\mathbf{2} \sqsubseteq 2 \to \beta_2\}$.

Le sous-typage sur les variables de type étoilées est transformé en égalité. L'ensemble des contraintes devient donc :

$$\begin{aligned} C = &\{1 \sqsubseteq \alpha;\ \texttt{ok} \sqsubseteq t;\ \alpha^* \to t^* \sqsubseteq t_G;\ 1 \sqsubseteq \alpha_1;\ \texttt{ok} \sqsubseteq t_1;\ 1 = \alpha_1;\ \alpha_1^* \to t_1^* \sqsubseteq 1 \to \beta_1;\ 1 \sqsubseteq \alpha_2; \\ &\texttt{ok} \sqsubseteq t_2;\ 2 = \alpha_2;\ \alpha_2^* \to t_2^* \sqsubseteq 2 \to \beta_2\} \\ = &\{1 \sqsubseteq \alpha;\ \texttt{ok} \sqsubseteq t;\ \texttt{ok} \sqsubseteq t_1;\ t_1^* \sqsubseteq \beta_1;\ \boxed{1 \sqsubseteq 2};\ \texttt{ok} \sqsubseteq t_2;\ t_2^* \sqsubseteq \beta_2\} \end{aligned}$$

L'ensemble des contraintes contient alors clairement une contradiction (la contrainte entourée) et l'erreur est donc détectée.

Si l'on revient à la première version du programme (a), C devient :

$$C = \{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \backslash \alpha) \Rightarrow \texttt{no} \sqsubseteq t\}$$

Ainsi, la première application conduit à $\{1 \sqsubseteq dom(\mathbf{inst}(\mathbf{t_G})_\mathbf{1});\ \mathbf{inst}(\mathbf{t_G})_\mathbf{1} \sqsubseteq 1 \to \beta_1\}$. Donc :

$$\begin{aligned} C = &\{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \backslash \alpha) \Rightarrow \texttt{no} \sqsubseteq t;\ 1 \sqsubseteq \alpha_1^*;\ \alpha_1^* \to t_1^* \sqsubseteq 1 \to \beta_1; \\ &1 \sqsubseteq \alpha_1 \Rightarrow \texttt{ok} \sqsubseteq t_1;\ 1 \sqsubseteq (\top \backslash \alpha_1) \Rightarrow \texttt{no} \sqsubseteq t_1\} \\ = &\{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \backslash \alpha) \Rightarrow \texttt{no} \sqsubseteq t;\ 1 \sqsubseteq 1 \Rightarrow \texttt{ok} \sqsubseteq \beta_1;\ 1 \sqsubseteq \top \setminus 1 \Rightarrow \texttt{no} \sqsubseteq \beta_1\} \\ = &\{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \backslash \alpha) \Rightarrow \texttt{no} \sqsubseteq t;\ \texttt{ok} \sqsubseteq t_1;\ t_1 \sqsubseteq \beta_1\} \end{aligned}$$

Puis la seconde application conduit à $\{2 \sqsubseteq dom(\mathbf{inst}(\mathbf{t_G})_\mathbf{2});\ \mathbf{inst}(\mathbf{t_G})_\mathbf{2} \sqsubseteq 2 \to \beta_2\}$. Soit :

$$\begin{aligned} C = &\{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \backslash \alpha) \Rightarrow \texttt{no} \sqsubseteq t;\ 2 \sqsubseteq \alpha_2^*;\ \alpha_2^* \to t_2^* \sqsubseteq 2 \to \beta_2; \\ &1 \sqsubseteq \alpha_2 \Rightarrow \texttt{ok} \sqsubseteq t_2;\ 1 \sqsubseteq (\top \backslash \alpha_2) \Rightarrow \texttt{no} \sqsubseteq t_2\} \\ = &\{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \backslash \alpha) \Rightarrow \texttt{no} \sqsubseteq t;\ \texttt{ok} \sqsubseteq \beta_1;\ 1 \sqsubseteq 2 \Rightarrow \texttt{ok} \sqsubseteq \beta_2; \\ &1 \sqsubseteq \top \setminus 2 \Rightarrow \texttt{no} \sqsubseteq \beta_2\} \\ = &\{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \backslash \alpha) \Rightarrow \texttt{no} \sqsubseteq t;\ \texttt{ok} \sqsubseteq \beta_1;\ \texttt{no} \sqsubseteq \beta_2\} \end{aligned}$$

Cet ensemble est alors soluble et si les types β_1 et β_2 sont minimisés pour être affichés, le type du résultat du programme est $\texttt{ok} \times \texttt{no}$.

Une deuxième difficulté est posée par la comparaison de fonction. En effet, une fonction peut apparaître dans un motif après une substitution. Par exemple :

```
F = fun(X) -> X end,
case E of F -> ok end
```

est traduit en `let F` $= \lambda$`X.X` , (λ`F.ok`) `E` qui se réduit en ($\lambda(\lambda$`X.X`).`ok`) `E`. La comparaison se fait alors par « nom », chaque fonction globale utilise son nom et les fonctions locales se voient attribuer des noms uniques lors de leur création. La définition par `=` recopie alors ce champ nom. Ainsi, si on ajoute au code précédent :

```
G = fun(X) -> X end,
H = F,
```

alors, si `E` vaut `G` le filtrage échoue et renvoie `ok` si `E` vaut `H`. Notons que la comparaison générique de OBJECTIVE CAML fonctionne de cette manière également.

Pour tenir compte de cette sémantique au niveau du système de type, le type d'une fonction doit intégrer son nom et l'égalité sur de tels types se réduit à celle des noms. Sur l'exemple précédent, le `case` conduit à typer l'application d'une fonction de type $(\alpha^* \to \alpha^*)_{n_1} \to$ `ok` à `E` de type t_E et produit la contrainte $t_E = (\alpha_1^* \to \alpha_1^*)_{n_1}$. Or, `G` est de type $(\beta \to \beta)_{n_2}$ et `H` est de type $(\alpha \to \alpha)_{n_1}$. Si `E` vaut `G`, l'erreur est détectée et si `E` vaut `H` la contrainte est trivialement vérifiée.

Les acteurs doivent également être étiquetés par leur adresse. Cependant, nous considérons que cette possibilité de substituer une variable d'un motif par une fonction ou un acteur n'a que peu d'intérêt pour la programmation. Nous choisissons de les ignorer dans le système présenté dans ce chapitre. Nous supposerons donc que les programmes que nous analysons n'utilisent pas cette possibilité.

Messages

Enfin, la dernière particularité de μErlang est l'absence d'étiquette de message, toutes les valeurs sémantiques peuvent être des messages. Cependant, une analyse automatique du code du compilateur, des librairies standards d'ERLANG et de programmes écrits en ERLANG disponibles sur Internet [55], montre que les messages envoyés et les filtrages utilisés dans le cas d'accès à la boîte aux lettres sont en quasi totalité des tuples (éventuellement réduit à un seul élément) dont le premier élément est un atome. Ces atomes jouent en fait le rôle des étiquettes de ML-ACT pour le programmeur. De plus, la règle 5.7 issues des *règles de bonne programmation* ([WA96]) conseille dans le cas des messages d'ajouter des étiquettes :

> « Tous les messages devraient être étiquetés. »

Nous avons donc fait le choix de développer notre système dans un cadre où tout message est un tuple dont la première composante est un atome. Contrairement à S. MARLOW et P. WADLER dans [MW97a], nous ne généralisons pas cette politique à toutes les données et tous les filtrages, mais uniquement aux données et aux filtrages concernant les messages. La grammaire des motifs (des filtrages de *receive*) devient donc :

$$p ::= \_ \mid x \mid C_{tuple}(s, p', \ldots, p')$$
$$p' ::= \_ \mid x \mid c \mid C(p', \ldots, p')$$

55. La quantité de code analysée représente environ 200 000 lignes de code.

où C_{tuple} est un constructeur de tuple (de longueur quelconque) et s un atome. La seule forme de filtrage de réception qui ne contient pas d'étiquette et qui est employé relativement (peu) régulièrement est l'accès avec un motif joker ou une variable. Celui-ci récupère alors le premier message présent dans la queue, pour, par exemple, en déléguer le traitement à une instruction de choix suivante ou bien à un autre processus en le lui envoyant. Les filtrages utilisant cette politique contiennent alors une partie variable qui correspond à ces messages délégués. Trois situations se présentent alors : aucune contrainte supplémentaire n'apparaît sur le message (par exemple, avec un motif joker), le message est délégué à un autre acteur ou le message est traité par un choix (`case`) dans la suite du programme. Le premier cas conduit à un acteur dont le type est ouvert, on peut donner une partie de la forme des messages qu'il reçoit mais il accepte toutes les formes. Le second cas consiste juste à ajouter au type de l'acteur initial le type de l'acteur à qui est délégué le traitement. Enfin, dans le troisième cas, les contraintes générées par le choix vont donner un type à ce message qui est alors converti par la stratégie précédente (le premier élément du tuple devient une étiquette). Cette stratégie rejette certains programmes corrects mais ils peuvent être sensiblement modifiés pour ne plus être rejetés. Par exemple, le programme suivant pose problème :

```
receive V -> case V of _ when integer(V) -> exp
             end
end
```

En effet, le type de `V` dans le corps `exp` est *int* et ne contient donc pas d'étiquette. Ce programme peut cependant être réécrit par exemple en :

```
receive V -> case V of {int,VV} when integer(VV) -> exp
             end
end
```

Notons qu'il serait possible d'étiqueter automatiquement de tels messages. Mais, il faut alors légèrement changer la politique de typage puisque de nombreux messages de types différents vont avoir la même étiquette (ce qui provoque actuellement une erreur).

Ces particularités du système de type de μErlang étant présentées, nous pouvons analyser plus précisément les erreurs dynamiques pouvant survenir lors de l'exécution d'un programme ainsi que notre stratégie vis-à-vis de ces erreurs.

8.2 Les erreurs

La sémantique du langage μErlang a été présentée au fur et à mesure de l'introduction des différents concepts figurant dans le langage, l'annexe B page 251 contient toutes les règles concernant la sémantique des configurations et de $\mathcal{F}unc_2$ pour faciliter leur analyse. Cependant, comme dans le cadre du système de type pour ML-Act, il nous reste à introduire le calcul du potentiel nécessaire à la règle Init des configurations dans laquelle un acteur est marqué (en indice) par son potentiel initial :

$$\textsc{Init:}\ \frac{}{\alpha \triangleright \mathrm{C}[init(a,v)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel \langle a \,|\, \varnothing \rangle_{\mathcal{P}(v\ a)} \triangleright v\ a}$$

$$\textsc{Potentiel:}\ \frac{\mathcal{P}(\bot, \{\}, e) = V,\ E,\ \mathcal{E}}{\mathcal{P}_e(e) = E}$$

$$\textsc{Val:}\ \frac{v \in \mathbb{A} \cup \mathbb{C} \cup \mathcal{F}_c \cup \{nop,\ nil\} \cup \{\bot\}}{\mathcal{P}(\mathcal{E}, \mathcal{F}, v) = \bot,\ \{\},\ \mathcal{E}} \qquad \textsc{Var1:}\ \frac{x \in dom(\mathcal{E})}{\mathcal{P}(\mathcal{E}, \mathcal{F}, x) = \mathcal{P}(\mathcal{E}, \mathcal{F}, \mathcal{E}(x))}$$

$$\textsc{Var2:}\ \frac{x \notin dom(\mathcal{E})}{\mathcal{P}(\mathcal{E}, \mathcal{F}, x) = \bot,\ \{\},\ \mathcal{E}} \qquad \textsc{Const:}\ \frac{C \in \{cons,\ tuple2, \ldots\} \quad \mathcal{P}(\mathcal{E}, \mathcal{F}, e_i) = V_i,\ E_i,\ \mathcal{E}_i}{\mathcal{P}(\mathcal{E}, \mathcal{F}, C(\vec{e})) = \bigcup_i V_i,\ \bigcup_i E_i,\ \bigcup_i \mathcal{E}_i}$$

$$\textsc{Seq:}\ \frac{\mathcal{P}(\mathcal{E}, \mathcal{F}, e_1) = V_1,\ E_1,\ \mathcal{E}_1 \quad \mathcal{P}(\mathcal{E}_1, \mathcal{F}, e_2) = V_2,\ E_2,\ \mathcal{E}_2}{\mathcal{P}(\mathcal{E}, \mathcal{F}, e_1; e_2) = V_2,\ E_1 \cup E_2,\ \mathcal{E}_2}$$

$$\textsc{Fun1:}\ \frac{s \in dom(\mathcal{F})}{\mathcal{P}(\mathcal{E}, \mathcal{F}, Fun(s)) = \bot,\ \{\},\ \mathcal{E}} \qquad \textsc{Fun2:}\ \frac{s \notin dom(\mathcal{F})}{\mathcal{P}(\mathcal{E}, \mathcal{F}, Fun(s)) = Fun(s),\ \{\},\ \mathcal{E}}$$

$$\textsc{Abs:}\ \frac{}{\mathcal{P}(\mathcal{E}, \mathcal{F}, \lambda[p_i, g_i.e_i]^{i \in I}) = \lambda[p_i, g_i.e_i]^{i \in I},\ \{\},\ \mathcal{E}}$$

$$\textsc{Rcv:}\ \frac{\vec{x}_i = \mathcal{FV}(p_i) \qquad \mathcal{P}(\mathcal{E}{+}\{\vec{x}_i \mapsto \vec{\top}\}, \mathcal{F}, e_i) = V_i,\ E_i,\ \mathcal{E}_i}{\mathcal{P}(\mathcal{E}, \mathcal{F}, receive(\lambda[p_i, g_i.e_i]^{i \in I})) = \bigcup_i V_i,\ \bigcup_i (label(p_i) \cup E_i),\ \bigcup_i \mathcal{E}_i}$$

$$\textsc{App:}\ \frac{\mathcal{P}(\mathcal{E}, \mathcal{F}, e) = \bigcup_j V_j, E, \mathcal{E}_1 \quad \mathcal{P}(\mathcal{E}, \mathcal{F}, e') = \bigcup_k V'_k, E', \mathcal{E}_2 \quad App(\mathcal{E}_1 \cup \mathcal{E}_2, \mathcal{F}, V_k, V'_j) = V^j_k, E^j_k, \mathcal{E}^j_k}{\mathcal{P}(\mathcal{E}, \mathcal{F}, e\ e') = \bigcup_{j,k} V^j_k,\ \bigcup_{j,k} E^j_k \cup E \cup E',\ \bigcup_{k,j} \mathcal{E}^j_k}$$

$$\textsc{Let:}\ \frac{\vec{x} = \mathcal{FV}(p) \qquad \mathcal{P}(\mathcal{E}, \mathcal{F}, e) = V,\ E,\ \mathcal{E}_1}{\mathcal{P}(\mathcal{E}, \mathcal{F}, \texttt{let}\ p = e) = V,\ E,\ \mathcal{E}_1{+}\{\vec{x} \mapsto \vec{V}\}}$$

$$\frac{\mathcal{P}(\mathcal{E}, \mathcal{F} \cup \{s\}, \mathcal{E}_f(s)\ V) = V_1,\ E,\ \mathcal{E}_1}{App(\mathcal{E}, \mathcal{F}, Fun(s), V) = V_1,\ E,\ \mathcal{E}} \qquad \frac{}{App(\mathcal{E}, \mathcal{F}, \bot, V') = \bot,\ \{\}}$$

$$\frac{\vec{x}_i = \mathcal{FV}(p_i) \quad \mathcal{P}(\mathcal{E}{+}\{\vec{x}_i \mapsto \vec{V}\}, \mathcal{F}, e_i) = V_i,\ E_i,\ \mathcal{E}_i}{App(\mathcal{E}, \mathcal{F}, \lambda[p_i, g_i.e_i]^{i \in I}, V) = \bigcup_i V_i,\ \bigcup_i E_i,\ \bigcup_i \mathcal{E}_i} \qquad \frac{}{App(\mathcal{E}, \mathcal{F}, \top, V') = \top,\ \mathbb{M}}$$

Règles 8.1 Potentiel d'un terme de $\mathcal{F}unc_2$.

Ce calcul, défini par les règles 8.1, est très voisin de celui propre à ML-ACT car les structures des interfaces et des messages sont semblables. La principale différence provient de la gestion des variables par un environnement au lieu d'une substitution. Le calcul utilise une fonction intermédiaire $\mathcal{P}$ qui approxime une expression par un couple composé d'une *valeur*, d'un *effet* et d'un *environnement*. Le calcul de cette approximation correspond à une exécution abstraite du programme sur un ensemble de valeurs sémantiques réduit. Ces *nouvelles* valeurs possibles sont :

- *rien* dénoté $\bot$ qui approxime les expressions n'apportant aucune information,
- *tout* dénoté $\top$ qui approxime les expressions dont l'apport est inconnu,
- une abstraction,
- une référence à une fonction globale ($Fun(s)$)

Notons, que l'approximation faite sur les choix peut également conduire à une union de telles valeurs, elles peuvent donc être décrites par la grammaire suivante :

$$V ::= \bot \mid \top \mid Fun(s) \mid \lambda[p_i, g_i.e_i]^{i\in I} \mid \bigcup\{V, \ldots, V\}$$

Ces valeurs permettent de déterminer toutes les fonctions qui seront appliquées et donc de cumuler leurs effets (qui décrivent les interfaces possibles d'un acteur). L'effet est un ensemble d'étiquettes qui approxime toutes les interfaces des états futurs de l'acteur courant. La seule instruction ajoutant réellement des étiquettes dans l'effet d'une expression est le *receive*. Pour cela, on utilise une fonction *label* qui calcule l'étiquette d'un motif de filtre de *receive* :

$$label(\_) = label(x) = \mathbb{M} \qquad label(C_{tuple}(s, p', \ldots, p')) = \{s\}$$

On approxime ainsi de façon extrême les variables et les motifs jokers en rendant le potentiel de l'acteur ouvert. Cette politique diminue le nombre de messages orphelins triviaux que nous pouvons déterminer par le biais du potentiel. Cependant, cela ne diminue pas la qualité de nos analyses puisque le potentiel n'est utilisé que dans la sémantique. En effet, le système de type fournira une approximation plus précise des interfaces que le potentiel. Notons que si l'acteur appelle une fonction reçue de l'extérieur dans le corps d'un message, le potentiel de l'acteur ne peut plus être calculé simplement. Le système de type proposé dans la suite de ce chapitre *calcule* cette approximation pour un programme clos dans lequel il est alors possible d'évaluer l'ensemble des messages reçus par un acteur. Mais, si l'acteur peut recevoir des messages venant d'un contexte ouvert cette approximation devient impossible. Dans le cadre du potentiel, nous sommes moins précis dans l'abstraction du programme et nous approximons alors le potentiel par $\mathbb{M}$ (il est ouvert). Pour cela, chaque variable argument d'un message lors de la définition d'un comportement (règle RCV) est approximée par $\top$.

Les quatre dernières règles (qui ne sont pas nommées) décrivent l'approximation de l'appel de fonction effectuée par *App* qui prend en paramètre l'environnement des variables ($\mathcal{E}$), la liste des noms de fonctions déjà dépliées ($\mathcal{F}$), la fonction et son argument :

- l'application d'une fonction nommée consiste en l'application de son abstraction (stockée dans $\mathcal{E}_f$) ;

- l'application d'une fonction prédéfinie (de $\mathcal{F}_c$) ou d'une fonction nommée déjà dépliée n'apporte aucune information ;
- le potentiel de l'application d'une abstraction est l'union des potentiels des différents corps de cette abstraction. Notons que nous supposons dans chaque corps que les variables sont toutes liées à l'argument. En effet, nous approximons les structures construites (listes et tuples) en les *écrasant* (règle CONST), leur déconstruction suit la même logique ;
- enfin, l'appel d'une fonction reçue dans un message conduit à un potentiel ouvert.

Le calcul du potentiel doit comporter pour se terminer un test d'occurrence pour vérifier qu'une fonction récursive déjà approximée ne l'est pas à nouveau. En effet, notre calcul s'effectuant par le remplacement d'un atome nom de fonction par le corps de cette fonction, il faut éviter de déplier une fonction récursive plusieurs fois pour ne pas boucler. Les seules fonctions récursives en μErlang étant les fonctions définies par leur nom (elles sont alors dans $\mathcal{E}_f$), la détermination du non-dépliage consiste en la collecte du nom des fonctions déjà dépliées dans un ensemble noté $\mathcal{F}$. Ainsi, nous éliminons le dépliage en boucle de ces fonctions en approximant une fonction dont le nom figure dans $\mathcal{F}$ par $\perp$ (FUN1).

Remarque :
Comme dans le calcul du potentiel des acteurs en ML-ACT*, les programmes erronés ne nous intéressent pas, l'approximation de ces programmes est donc quelconque. Par exemple, une variable libre est approximée par* $\perp$*.*

Une fois le potentiel défini, la sémantique de μErlang est complète, on peut donc analyser les sources d'erreurs d'exécution. Tout d'abord, comme dans le cadre de ML-ACT, il y a deux familles d'erreurs : les erreurs fonctionnelles et les erreurs concurrentes. La première forme d'erreurs correspond aux erreurs classiques des langages fonctionnels, une valeur est manipulée dans un contexte erroné (le programme additionne une chaîne de caractère et un entier). Le second type d'erreurs est moins courant, il s'agit d'une mauvaise utilisation d'une interface. C'est-à-dire, l'envoi d'un message à un acteur qui ne saura jamais le traiter, soit parce qu'il ne sait pas traiter les messages avec cette étiquette soit parce que les arguments de ce message ne coïncident pas avec le motif de filtrage associé à cette étiquette.

Dans le filtrage, aucune erreur n'est possible contrairement à ML-ACT. Dans la réduction fonctionnelle, les erreurs possibles sont :

- une variable n'est pas définie, cela permet également de détecter les utilisations illégales de *ego* (`self` dans la syntaxe ERLANG) et *receive* ;
- une valeur appliquée n'est pas une fonction ;
- une application de fonction définie par cas s'est terminée par l'échec de tous les filtrages possibles ou bien un `let` échoue.

Dans la partie concurrente, les erreurs suivantes peuvent se produire :

- l'envoi d'un message à une valeur qui n'est pas une adresse ;
- un message est reçu mais il ne figure pas dans le potentiel de son destinataire (c'est donc un orphelin trivial).

Comme en ML-ACT, une erreur est traitée dans la sémantique mais ne peut pas apparaître. En effet, les programmes étant obtenus par traduction, nous savons que l'erreur de

création d'acteur (InitE) ne peut pas avoir lieu. Chacun de ces paramètres étant généré automatiquement, ils sont toujours corrects.

Remarquons que la grande précision du typage du filtrage nous permet (contrairement au système de type de ML-Act) de détecter les filtrages qui échouent.

Parmi les erreurs possibles, seule la dernière n'est pas usuellement détectée par les systèmes de type des langages typés statiquement. Cependant, notre approche, sans être totalement originale, n'est pas courante : nous essayons de typer statiquement un langage dynamique. Le but du système de type que nous allons maintenant introduire est donc d'éliminer toutes ces erreurs potentielles en nous basant sur les systèmes de types fonctionnels usuels. Cependant, la détection de la dernière erreur impose de procéder à une analyse fine des communications à travers des types suffisamment précis.

8.3 Les types

Dans cette section, nous définissons précisément les types utilisés par le système de type de Erlang et l'ordre de sous-typage qu'ils respectent. Nous introduisons également les types des fonctions prédéfinies qui sont contenues dans l'environnement initial de typage.

Types

Avant de donner la définition des types, présentons les informellement. Leur forme est, en général, sensiblement différente de celle des types de ML-Act. Un type est associé à chaque constante et les types de base (T_B) approximent l'ensemble des constantes (comme pour ML-Act). Une constante a donc deux types : elle-même et son type de base, le premier étant sous-type du second. En revanche, les tuples suivent la définition de ML-Act, un tuple est un produit cartésien de types ($T \times \cdots \times T$).

Les types des listes et les types fonctionnels sont également sensiblement différents du fait des particularités sémantiques de μErlang (voir la première section de ce chapitre) :

- Les types de listes (*nil* et *cons*(T, T)) doivent être suffisamment précis pour deux raisons : les listes peuvent être hétérogènes (*i.e.* contenir des données de formes différentes) et elles sont utilisées pour passer les paramètres à une fonction lors de son appel. Si l'on veut être capable de donner des types intéressants aux fonctions, il faut être capable de conserver le type de chaque élément de la liste en préservant l'ordre (lorsque cela est possible).
- La forme des types des fonctions a déjà été introduite et justifiée en partie dans la première section de ce chapitre. Sa forme est :

$$T_1 \xrightarrow[r,@\phi]{E} T_2$$

  C'est donc un constructeur d'arité cinq dont les arguments sont :
  - le type du domaine T_1 ;
  - le type de l'*ego* $@\phi$ (voir page 204) ;
  - une information sur la possibilité d'echec d'une garde r, celle-ci peut prendre comme valeur ? (voir page 204) ;

– l'effet E ;
– le type du codomaine T_2.

Ce constructeur est contravariant par rapport aux deux premiers arguments et covariant par rapport aux trois autres.

Enfin, dans le cadre de μErlang, la seule entité concurrente que nous approximons est l'acteur. En effet, les interfaces n'ont plus d'équivalent manipulables, chaque interface est donnée explicitement à chaque accès à la boîte aux lettres. Le type interface correspond, en fait, à l'effet calculé par le système de type et mémorisé dans le type fonctionnel. Les messages pouvant avoir n'importe quelle forme, ils n'ont pas de type particulier.

Nous pouvons maintenant introduire les types que nous utilisons pour $\mathcal{F}unc_2$. Ils suivent la définition générale 3.2 donnée page 59 et précisent l'ensemble des types de base, celui des constructeurs de type et la forme des types interfaces. La forme des interfaces et celle des présences est identique en ML-ACT et en μErlang.

Définition 8.1 (Types) :
L'ensemble des types est noté $\mathbb{T}$, sa définition est paramétrée par l'ensemble des atomes (notés s et appartenant à $\mathbb{A}\mathrm{t}$) et par trois ensembles de variables : celui des variables de type (notées t et appartenant à $\mathbb{V}_t$), celui des variables de rangée (notées i et appartenant à $\mathbb{V}_i$) et enfin, celui des variables de présence (notées p et appartenant à $\mathbb{V}_p$). Un type est un terme construit par la grammaire suivante :

$$
\begin{array}{lll}
T & ::= \bot \mid \top \mid c \mid T_B \mid t \mid \bigsqcup\{T,\ldots,T\} \mid \bigsqcap\{T,\ldots,T\} & \\
& \mid\ T \underset{R,@I}{\overset{I}{\rightarrow}} T & \textit{type de fonction} \\
& \mid\ \mathit{unit} \mid T \times \cdots \times T \mid \mathit{tuple} & \textit{type de tuples} \\
& \mid\ \mathit{nil} \mid \mathit{cons}(T,\ T) & \textit{type de listes} \\
& \mid\ @I & \textit{type d'acteurs} \\
T_B & ::= \mathit{int} \mid \mathit{float} \mid \mathit{char} \mid \mathit{atom} & \\
I & ::= \{\} \mid \top_I \mid i \mid \{m : P,\ I\} & \\
P & ::= \dagger \mid p \mid \bullet T \mid \circ T & \\
R & ::= \varnothing \mid ? &
\end{array}
$$

Un type est clos si et seulement si il ne contient pas de variable.

Dans la suite, l'écriture du type des fonctions est adaptée à la situation. Ainsi, si elles n'ont pas d'effet (l'effet vaut $\{\}$), il est oublié ou si elles n'utilisent pas *ego* dans leur corps $@\{\}$ est omis. Les listes de la forme $\mathit{cons}(t_1, \mathit{cons}(t_2, \mathit{nil}))$ sont notées $[t_1, t_2]$. Enfin, nous utilisons quelques abréviations éventuellement récursives :

– $\mathit{bool} \triangleq \mathit{true} \sqcup \mathit{false}$ pour dénoter l'ensemble des booléens ;
– $\mathit{num} \triangleq \mathit{int} \sqcup \mathit{float}$ pour dénoter l'ensemble des constantes arithmétiques ;
– $\mathit{string} \triangleq \mathit{nil} \sqcup \mathit{cons}(\mathit{char}, \mathit{string})$ pour dénoter l'ensemble des chaînes de caractères ;
– $\mathit{list} \triangleq \mathit{nil} \sqcup \mathit{cons}(\top, \mathit{list})$ pour dénoter l'ensemble de toutes les listes ;
– $\mathit{fun} \triangleq \bot \xrightarrow{\top_I} \top$ est la plus grande des fonctions, elle dénote donc l'ensemble des fonctions ;
– $\mathit{pid} \triangleq @\{\}$ est le plus grand des acteurs et dénote l'ensemble des acteurs ;

Remarquons que certaines de ces abréviations sont récursives entre elles. Le système de type ne contient pas de types récursifs car ceux-ci n'apparaissent pas pendant l'inférence ou la résolution. En fait, cette récursivité des types est codée par des contraintes récursives. Cependant, en phase finale du typage, les types doivent être simplifiés dans un but de lisibilité, or, cette simplification fait apparaître des types récursifs. Nous reviendrons sur cette simplification et sur les types récursifs plus loin dans ce chapitre.

Le calcul de l'abstraction d'un type (*Abs*) est très proche de celui de ML-Act. Elle permet de relier plus précisément les notions de potentiel et de type d'un acteur. Elle est définie sur les types clos par :

$$Abs(@I) = Abs_I(I) \qquad Abs(T \xrightarrow{I} T) = Abs_I(I)$$

$$Abs(\bigsqcup_i T_i) = \bigcup_i Abs(T_i) \qquad Abs(T) = \{\} \text{ dans les autres cas}$$

en utilisant l'abstraction des rangées (Abs_I) :

$$Abs_I(\{\}) = \{\} \qquad Abs_I(\{m : \circ T,\ I\}) = Abs(I) \qquad Abs_I(\{m : \bullet T,\ I\}) = \{m\} \cup Abs(I)$$

L'abstraction est également étendue aux types contenant des variables en collectant toutes les inclusions dues aux contraintes par la règle suivante :

$$I_1 \sqsubseteq I_2 \Longrightarrow Abs_I(I_1) \subseteq Abs_I(I_2)$$

On montrera lors de la preuve de correction du système de type que l'abstraction d'un type d'acteur sera une approximation inférieure du potentiel initial d'un acteur.

Sous-typage

Le sous-typage s'inspire des définitions des chapitres 3 (cas des configurations) et 6 (cas de ML-Act) en les adaptant à μErlang. Les parties interfaces et présences sont quasiment identiques à celles présentées dans le cadre de ML-Act (on ajoute uniquement une plus grande interface).

Définition 8.2 (Sous-typage) :
*Un type T_1 est **sous-type** d'un type T_2 si et seulement si il est possible de dériver une preuve de $T_1 \sqsubseteq T_2$ en utilisant le système de règles 8.2.*

De manière similaire à ML-Act, le sous-typage est utilisé pour collecter les différentes contraintes dues à l'utilisation des entités concurrentes (acteurs et interfaces). Ainsi, une contrainte $@\varphi_1 \sqsubseteq @\varphi_2$ (qui est équivalente par sous-typage à $\varphi_2 \sqsubseteq \varphi_1$) signifie que l'acteur 1 (de type $@\varphi_1$) est utilisé dans un contexte lui donnant comme type $@\varphi_2$ et donc que son type final ($@\varphi_1$) doit contenir les éléments (messages reçus et installés) collectés par le type $@\varphi_2$. Le sous-typage n'est guère différent de celui de ML-Act, seule la forme des règles est adaptée à la nouvelle forme des types. Par exemple, le sous-typage fonctionnel de μErlang condense le typage fonctionnel et celui des interfaces de ML-Act. Nous faisons

$$\frac{}{\bot \sqsubseteq T} \qquad \frac{}{T \sqsubseteq \top} \qquad \frac{T \sqsubseteq T_1 \quad T \sqsubseteq T_2}{T \sqsubseteq \bigsqcap\{T_1, T_2\}} \qquad \frac{T \sqsubseteq T_1}{T \sqsubseteq \bigsqcup\{T_1, T_2\}} \qquad \frac{T \sqsubseteq T_2}{T \sqsubseteq \bigsqcup\{T_1, T_2\}}$$

$$\frac{c \in \mathit{Char}}{c \sqsubseteq \mathit{char}} \qquad \frac{i \in \mathbb{N}}{i \sqsubseteq \mathit{int}} \qquad \frac{r \in \mathbb{D}}{r \sqsubseteq \mathit{float}} \qquad \frac{}{\mathit{char} \sqsubseteq \mathit{int}} \qquad \frac{s \in \mathbb{A}\mathrm{t}}{s \sqsubseteq \mathit{atom}}$$

$$\frac{}{T_1 \times \cdots \times T_n \sqsubseteq \mathit{tuple}} \qquad \frac{\forall i \in 1..n \;\; T_i \sqsubseteq T'_i}{T_1 \times \cdots \times T_n \sqsubseteq T'_1 \times \cdots \times T'_n} \qquad \frac{T_1 \sqsubseteq T'_1 \qquad T_2 \sqsubseteq T'_2}{\mathit{cons}(T_1, T_2) \sqsubseteq \mathit{cons}(T'_1, T'_2)}$$

$$\frac{T' \sqsubseteq T}{@T \sqsubseteq @T'} \qquad \frac{T'_1 \sqsubseteq T_1 \quad @\phi' \sqsubseteq @\phi \quad R \sqsubseteq R' \quad I \sqsubseteq I' \quad T_2 \sqsubseteq T'_2}{T_1 \underset{R,@\phi}{\overset{I}{\to}} T_2 \sqsubseteq T'_1 \underset{R',@\phi'}{\overset{I'}{\to}} T'_2} \qquad \frac{}{\varnothing \sqsubseteq ?}$$

$$\frac{}{\{\} \sqsubseteq I} \qquad \frac{}{i \sqsubseteq \top_I} \qquad \frac{P_1 \sqsubseteq P_2 \qquad I_1 \sqsubseteq I_2}{\{m : P_1, I_1\} \sqsubseteq \{m : P_2, I_2\}}$$

$$\frac{T_1 \sqsubseteq T_2}{\circ T_1 \sqsubseteq \circ T_2} \qquad \frac{T_1 \sqsubseteq T_2}{\circ T_1 \sqsubseteq \bullet T_2} \qquad \frac{T_2 \sqsubseteq T_1}{\bullet T_1 \sqsubseteq \bullet T_2}$$

Rèles 8.2 La relation de sous-typage pour $\mathcal{F}unc_2$.

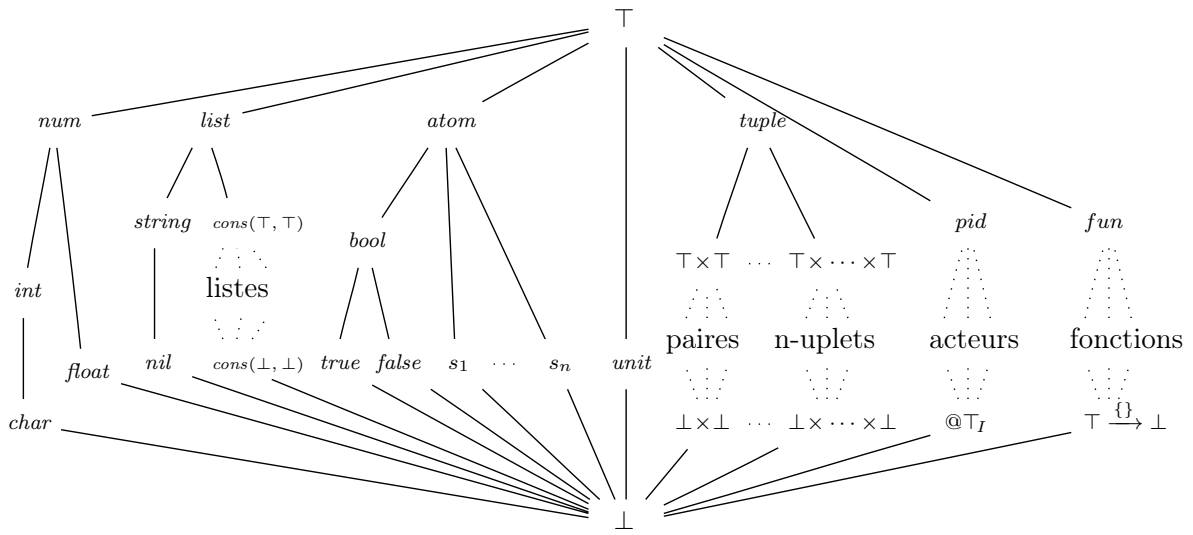


Figure 8.1 Le treillis des types.

également apparaître une relation de sous-typage entre entité de base ($char \sqsubseteq int$) ainsi que certaines bornes supérieures (*atom* et *tuple*).

Il est également possible de démontrer d'autres relations de sous-typage en plus de celles définies explicitement. Pour résumer la forme de la relation d'ordre sur les types, la figure 8.1 présente une version simplifiée du diagramme de Hasse de $\sqsubseteq$. Pour éviter qu'il ne soit surchargé, nous n'avons pas fait figurer toutes les unions et intersections ainsi que les caractères, les entiers et les réels.

Avant de présenter les règles de typage, il ne reste plus qu'à introduire l'environnement initial de typage.

Environnement initial

Dans la version du système de type présenté dans cette thèse, il n'existe pas de notion de module avec la possibilité d'importer des fonctions. En effet, la construction d'un tel environnement de typage initial est orthogonal à nos travaux. Nous supposons seulement que l'environnement initial contient toutes les fonctions nécessaires au typage du programme en cours d'analyse. L'environnement initial ainsi crée est manipulé par le nom $\mathcal{E}_0$ dans les règles de typage et il est accessible en tout point d'un programme.

L'environnement initial de typage contient les fonctions usuelles prédéfinies. Celles-ci se voient attribuer des types plus ou moins précis selon leur capacité à apporter une information intéressante :

- des fonctions de comparaisons : `<`, `=<`, `>`, `>=`, `=:=`, `=/=`, `==` et `/=` de type $[\top, \top] \rightarrow bool$, toutes les données étant comparables ;
- des fonctions arithmétiques sur les entiers ou les flottants :

`+`, `-`	:	$([int] \rightarrow int) \sqcup ([float] \rightarrow float) \sqcup$
		$([int, int] \rightarrow int) \sqcup (([float, num] \sqcup [int, float]) \rightarrow float)$
`*`	:	$([int, int] \rightarrow int) \sqcup (([float, num] \sqcup [int, float]) \rightarrow float)$
`/`	:	$[float, float] \rightarrow float$
`div`, `rem`	:	$[int, int] \rightarrow int$
`abs`	:	$([int] \rightarrow int) \sqcup ([float] \rightarrow float)$
`round`, `trunc`	:	$num \rightarrow int$
`float1`	:	$num \rightarrow float$

- des fonctions arithmétiques binaires sur les entiers : `bor`, `bxor`, `bsl`, `bsr` et `band` de type $[int, int] \rightarrow int$ et `bnot` de type $[int] \rightarrow int$;
- des fonctions de calcul sur les booléens : `or`, `xor` et `and` de type $[bool, bool] \rightarrow bool$ et `not` de type $[bool] \rightarrow bool$;
- des fonctions de manipulation de listes : `++` et `-` de type $[list, list] \rightarrow list$; `hd` de type $[cons(\alpha, \beta)] \rightarrow \alpha$; `tl` de type $[cons(\alpha, \beta)] \rightarrow \beta$ et `length` de type $[list] \rightarrow int$;
- des fonctions de manipulation de tuples : `element` de type $[int, tuple] \rightarrow \bot$ et `size` de type $[tuple] \rightarrow int$.

Remarque :
Notons que les types de `element` *et de* `size` *sont plus précis dans le prototype réalisé. En effet, une analyse du code* ERLANG *existant permet de déterminer que la plupart des*

utilisations de ces fonctions se fait sur des tuples de dimensions inférieures à six. Le type de `element` *devient alors* $[1,t]\to\pi_1(t)\cup\cdots\cup[6,t]\to\pi_6(t)\cup[int,tuple]\to\bot$ *(où* π_i *est la* i^e *projection) et celui de* `size` *calcul la vraie taille s'il le peut. Comme ces types ne rendent pas plus complexes le système tout en ayant des formes plus lourdes, nous ne les utilisons pas dans la présentation formelle.*

L'envoi contient également la définition des fonctions prédéfinies modélisant la partie acteur du programme :

- La création d'une adresse d'acteur (*new*) renvoie une nouvelle adresse sur laquelle aucune contrainte n'existe encore. Soit :

$$\boxed{new\colon @\psi}$$

- La création d'un acteur (*init*) reçoit en argument un couple composé de l'adresse de l'acteur à créer et du comportement initial de cet acteur et elle crée l'acteur. Rappelons que dans le cadre de μErlang, un comportement est une fonction qui prend en paramètre l'adresse de l'acteur qui l'exécutera et dont l'effet correspond à son interface (son résultat peut donc être quelconque). Son type est donc :

$$\boxed{init\colon @\psi\times(@\psi\xrightarrow{\psi}\top)\to unit}$$

- Le changement de comportement (*receive*), à partir de l'adresse de l'acteur courant (*ego*) et de l'interface I (d'accès à la boîte aux lettres), récupère et traite un message de sa queue avant de poursuivre son calcul. L'interface I est une fonction dont la forme générale est $\lambda[p_i, g_i.e_i]$ (car en μErlang, le *receive* reçoit explicitement la fonction de filtrage). Son type est donc $\bigsqcup_i(t_i^p\xrightarrow{E_i}t_i^e)$. Cette interface traitera les messages t_i^p et ceux décrits par les E_i. L'application du *receive* a pour effet $\bigsqcup_i(E_i\sqcup t_i^p)$. Comme les t_i^p n'ont pas tous la forme voulue, ils sont transformés par la fonction *Inter*. Le type d'un motif d'une interface doit donc être s, $s\times t_1\times\cdots\times t_n$, $\top$ ou une variable de type vue la limitation introduite dans l'introduction sur la forme des messages et des interfaces. Il est alors transformé par :

$$Inter(s)=\{s:\bullet unit\}\qquad Inter(s\times t_1\times\cdots\times t_n)=\{s:\bullet(t_1\times\cdots\times t_n)\}\qquad Inter(\top)=\top_I$$

Si le filtre est une variable x de type t_x, l'évaluation de $Inter(t_x)$ est repoussée en fin de résolution, lorsque l'on connaît T la borne supérieure de t_x. Alors, $Inter(t_x)=Inter(T)$ qui utilise les règles de calculs supplémentaires suivantes :

$$Inter(\bigsqcup_i T_i)=\bigsqcup_i Inter(T_i)\qquad Inter(\bigsqcap_i T_i)=\bigsqcap_i Inter(T_i)\qquad Inter(T)=\textbf{Err}\ \text{sinon}$$

Par exemple, si les appels $\lambda[$`set`$\ldots.;$ `get`$\ldots.]\ x$ et $\lambda[$`set`$\ldots.;$ `put`$\ldots.]\ x$ sont contenus dans la suite du code, alors la borne supérieure de t_x sera `set` qui est bien le seul message que pourra filtrer x sans conduire à une erreur.
Pour noter le calcul précédent, nous utilisons **effect**, dont la définition est :

$$\textbf{effect}(\bigsqcup_i(t_i^p\xrightarrow{E_i}t_i^e))=\bigsqcup_i(E_i\sqcup Inter(t_i^p))$$

Cette définition ne contient qu'un seul cas car le *receive* fournit toujours l'interface immédiatement.
Le résultat de l'application d'un *receive* est l'un des résultats possibles de réaction et son type est l'union des types de résultat. Pour cela, nous définissons la fonction *codom* par :

$$codom(\bigsqcup_i (t_i^p \xrightarrow{E_i} t_i^e)) = \bigsqcup_i t_i^e$$

C'est également le seul cas possible par construction du *receive*.
Finalement, la fonction *receive* va avoir le type :

$$\boxed{receive\colon @\mathbf{effect}(\alpha)\times\alpha \xrightarrow{\mathbf{effect}(\alpha)} codom(\alpha)}$$

De manière similaire à ML-ACT, nous passons en paramètre du *receive* l'adresse de l'acteur courant pour relier le type de la future interface de l'acteur au type de l'acteur lui-même ;

– La procédure d'envoi de message prend en argument un couple composé du message et de l'acteur destinataire. Soit :

$$\boxed{send\colon \alpha\times@Mess(\alpha)\to unit}$$

Pour les mêmes raisons que dans le cas du *receive*, le type des messages est transformé par la fonction *Mess* qui reprend le principe de *Inter* :

$$Mess(s) = \{s : \circ unit\} \qquad Mess(s\times t_1\times\cdots\times t_n) = \{s : \circ(t_1\times\cdots\times t_n)\}$$

$$Mess(\bigsqcup_i T_i) = \bigsqcup_i Mess(T_i) \qquad t_1 \sqsubseteq t_2 \Longrightarrow Mess(t_1) \sqsubseteq Mess(t_2)$$

Contrairement à *Inter* qui est toujours correctement appliquée (si l'hypothèse de la première section est vérifiée), *Mess* peut être appliquée à un type de mauvaise forme. Par exemple, si le programmeur envoie un entier, $Mess(int)$ peut apparaître, on rejette alors le programme en indiquant que l'analyseur n'accepte que des messages avec étiquette.

Pour le typage des gardes dans lesquelles figurent des fonctions de test de type dynamique, on définit le domaine de chacun de ces tests de type :

$dom(\texttt{atom})$	$= atom$	$dom(\texttt{pid})$	$= pid$
$dom(\texttt{float})$	$= float$	$dom(\texttt{constant})$	$= num \sqcup atom \sqcup pid \sqcup nil$
$dom(\texttt{integer})$	$= int$	$dom(\texttt{list})$	$= list$
$dom(\texttt{number})$	$= num$	$dom(\texttt{tuple})$	$= tuple$

Les constructeurs se typent de manière triviale en μErlang puisque *TypeC* associe un constructeur de type à chaque constructeur :

$$\begin{array}{lcl} \mathit{TypeC}(\mathit{nop}) & = & \mathit{unit} \\ \mathit{TypeC}(\mathit{nil}) & = & \mathit{nil} \\ \mathit{TypeC}(\mathit{cons}) & = & \Lambda(t_1, t_2).\mathit{cons}(t_1, t_2) \\ \mathit{TypeC}(\mathit{tuple2}) & = & \Lambda(t_1, t_2).t_1 \times t_2 \\ \ldots & = & \ldots \\ \mathit{TypeC}(\mathit{tuplen}) & = & \Lambda(t_1, \ldots, t_n).t_1 \times \cdots \times t_n \end{array}$$

Les deux lemmes montrés sur les constructeurs de types de ML-ACT restent vrais :

Lemme 8.1 (Covariance des constructeurs de types) :
Pour tout constructeur C, le constructeur de type associé $\mathit{TypeC}(C)$ est covariant vis-à-vis de tous ces paramètres. Soit :

$$\forall i \in 1..n \quad t_i \sqsubseteq t'_i \Longrightarrow \mathit{TypeC}(C)(t_1, \ldots, t_n) \sqsubseteq \mathit{TypeC}(C)(t_1, \ldots, t_{i-1}, t'_i, t_{i+1}, \ldots, t_n)$$

PREUVE : La preuve est immédiate en appliquant la définition de la relation de sous-typage à chaque constructeur.

Le deuxième lemme est la réciproque du précédent, il est utilisé dans la preuve d'un lemme sur le filtrage.

Lemme 8.2 (Sous-typage des constructeurs de types) :
Pour tout constructeur C, si un type construit $\mathit{TypeC}(C)(t_1, \ldots, t_n)$ est sous-type d'un type construit $\mathit{TypeC}(C)(t'_1, \ldots, t'_n)$, tous les sous-types sont également en relation de sous-typage. Soit :

$$\mathit{TypeC}(C)(t_1, \ldots, t_n) \sqsubseteq \mathit{TypeC}(C)(t'_1, \ldots, t'_n) \Longrightarrow \forall i \in 1..n \quad t_i \sqsubseteq t'_i$$

PREUVE : La preuve est également triviale par application directe de la définition du sous-typage.

Après cette définition des types et environnements de typage, présentons le système de type.

8.4 Le système de type

Dans cette section, nous introduisons les règles de typage des configurations (et donc des termes de $\mathcal{F}\mathit{unc}_2$) issues de la traduction de programmes μErlang. Rappelons qu'à l'issue de l'étape de traduction de μErlang en configuration, nous disposons, d'une part, d'un environnement associant à chaque atome une (ou plusieurs) définition et, d'autre part, d'un acteur anonyme dont le corps est le programme principal.

Le typage est constitué de deux phases distinctes, la première qui construit l'environnement initial de typage (et vérifie la validité des définitions) et la seconde qui type effectivement le programme. En effet, toutes les fonctions d'un module pouvant être récursives, il faut introduire une variable décrivant leur type dans l'environnement de typage.

$$\frac{}{\{\} \Rightarrow \{\},\ \{\},\ \{\}} \qquad \frac{q \Rightarrow \mathcal{E}_t,\ V,\ A \quad s \in dom(\mathcal{E}_t) \quad lg(p_1) \in A(s)}{\{s \mapsto \lambda[p_i, g_i.e_i]^{i \in I}\} :: q \Rightarrow \textbf{Err}}$$

$$\frac{q \Rightarrow \mathcal{E}_t,\ V,\ A \qquad s \in dom(\mathcal{E}_t) \qquad lg(p_1) \notin A(s) \qquad V(s) = \lambda[p_i, g_i.e_i]^{i \in K}}{\{s \mapsto \lambda[p_i, g_i.e_i]^{i \in I}\} :: q \Rightarrow \mathcal{E}_t,\ \{s \mapsto \lambda[p_i, g_i.e_i]^{i \in I \cup K}\} :: V|_s,\ \{s \mapsto lg(p_1) :: A(s)\} :: A|_s}$$

$$\frac{q \Rightarrow \mathcal{E}_t,\ V,\ A \qquad s \notin dom(\mathcal{E}_t)}{\{s \mapsto \lambda[p_i, g_i.e_i]^{i \in I}\} :: q \Rightarrow \{s \mapsto t\} :: \mathcal{E}_t,\ \{s \mapsto \lambda[p_i, g_i.e_i]^{i \in I}\} :: V,\ \{s \mapsto \{lg(p_1)\}\} :: A}$$

Règles 8.3 Une première passe de typage.

Une première phase

La première phase du typage réalise trois tâches : elle définit toutes les fonctions en leur attribuant un type variable, elle vérifie qu'il n'y a pas deux fonctions ayant le même nom et la même arité et elle fusionne les abstractions (d'arités différentes) attachées au même atome. En effet, un module ERLANG ne doit pas contenir deux fonctions identiques. Elle construit donc un environnement de typage ($\mathcal{E}_t$) associant à chaque atome (nom de fonction) une variable de type et un nouvel environnement (V) contenant les résultats de la fusion des différentes abstractions associées à un atome. Notons que nous supposons qu'il ne puisse pas y avoir de chevauchement des branches lors de cette fusion, dans la troisième règle $I \cap K = \varnothing$. Pour faciliter la vérification de l'absence de collision d'arité pour un même atome, on construit également un environnement (A) contenant pour chaque atome toutes les arités déjà utilisées. Les règles de déductions 8.3 sont alors appliquées sur l'environnement issu de la traduction. À partir de l'environnement $\mathcal{E}$, les trois environnements déjà décrits sont produits, ce qui s'exprime par la prémisse $\mathcal{E} \Rightarrow \mathcal{E}_t,\ V,\ A$. Dans ces règles, $lg(p_1)$ est utilisé pour déterminer la longueur de la liste formée par les paramètres formels et $E|_s$ restreint l'ensemble E en en ôtant s.

Une fois cette première phase terminée, il faut parcourir l'environnement V obtenu pour typer effectivement toutes les fonctions.

Le typage proprement dit

Dans toutes les règles introduites par la suite, nous respectons la forme des règles de typage de ML-ACT en rendant l'ensemble des contraintes global. Ainsi, l'application d'une règle peut ajouter une ou plusieurs contraintes sur les types (celles-ci apparaissent à la droite de la règle).

Cette seconde phase du typage exploite l'environnement de typage initial $\mathcal{E}_t$ issu de la phase précédente. Nous typons alors l'ensemble des fonctions du module (l'ensemble V de la phase 1) et le corps e de l'acteur anonyme résultant de la traduction de la fonction principale du programme (celle de nom `main`). On applique donc la déduction suivante :

$$\frac{\mathcal{E}_t \vdash_F V \quad \mathcal{E}_t \vdash_e e : t,\ \_,\ \_}{\mathcal{E}_t \vdash_M V,\ e}$$

où nous anticipons la forme des règles de typage d'une expression : $\mathcal{E}_t \vdash_e e : t,\ \mathcal{E}'_t,\ E$. Cette prémisse signifie que l'expression e est typée dans l'environnement de typage $\mathcal{E}_t$ et qu'il en résulte : un type t, un environnement de typage $\mathcal{E}'_t$ dans lequel la *suite* du programme est typée et un effet E (une interface de l'acteur qui exécute la fonction). Un environnement résultat est nécessaire car chaque expression peut introduire des variables (via un `let`) et ces variables doivent être transmises à la suite du calcul. Lorsque certains résultats du typage ne nous intéressent pas[56], nous ne les nommons pas et nous utilisons un joker. Le typage des fonctions ($\vdash_F$) suit alors les deux règles ci-dessous :

$$\frac{}{\mathcal{E}_t \vdash_F \{\}} \qquad \frac{\begin{array}{c}\{\} \vdash_p p_i : t_i,\ \mathcal{E}_i,\ V_i \qquad \mathcal{E}_i{+}\{ego : @\phi_i\} \vdash_g g_i : r_i \\ \mathcal{E}_i{+}\{ego : @\phi_i\} \vdash_e e_i : t'_i,\ \mathcal{E}'_i,\ E_i \quad \alpha_i = t_i \backslash \bigcup\{t_j \mid j{<}i \wedge r_j{\neq}?\} \\ t{=}\bigsqcup\{\mathbf{star}(V_i \cup \mathcal{V}(t_i),\ \alpha_i \xrightarrow[r_i,\ @\phi_i]{E_i} t_i) \mid i \in I \wedge \alpha_i \neq \bot\} \quad \mathcal{E}_t \vdash_F q\end{array}}{\mathcal{E}_t \vdash_F \{s \mapsto \lambda[p_i, g_i.e_i]^{i\in I}\} :: q} \begin{pmatrix} t \sqsubseteq \mathcal{E}_t(s) \\ E_i \sqsubseteq \phi_i \end{pmatrix}$$

Le typage du corps d'une fonction globale est très proche de celui d'une fonction locale qui sera décrit précisément un peu plus loin (voir page 209). La seule différence provient de la gestion de l'*ego*. En effet, *ego* est une variable dynamique en μErlang, il est donc possible qu'elle soit libre dans le corps d'une fonction globale. L'environnement de typage de la garde et de l'expression d'une branche est légèrement modifié pour intégrer éventuellement le type de l'*ego*. Donc, chaque atome se voit associer un type fonctionnel dont la construction est expliquée page 209 et toutes les branches de fonctions globales conservent le type de l'$ego(@\phi_i)$. Notons que toutes les branches d'une fonction sont indépendantes.

Les définitions de fonctions étant récursives par défaut en ERLANG, la première contrainte générée est identique à celle ajoutée lors de l'application de la règle LREC du typage de ML-ACT. La seconde contrainte cumule l'effet de la fonction dans le type de l'*ego*.

Nous allons maintenant introduire le typage des filtres (motifs et gardes) puis ensuite celui des expressions.

Les filtrages

Le typage des filtrages par les règles 8.4 se décompose en typage des gardes et typage des motifs.

L'approximation des gardes apporte des informations sur le domaine de certaines variables et une indication permettant de savoir si la garde peut échouer, sa forme est donc $\mathcal{E} \vdash_g g : r$. En effet, si la garde ne contient que des *true* et des fonctions de test de type, elle ne peut pas échouer (sur des arguments de son domaine) et elle renvoie $\emptyset$. Par exemple, le filtre `I when integer(I) -> ...` ne peut pas échouer sur les entiers et donc tous les motifs des filtres qui le suivent ne recevrons jamais d'entiers. En revanche, les comparaisons peuvent échouer et la détermination de cet échec est statiquement indécidable, nous supposons donc qu'elles peuvent toujours échouer (le résultat est ?). Notons que dans le cas de gardes multiples (GUARDS), nous utilisons un ou ($\vee$) pour indiquer que si un ? apparaît dans une des sous-gardes, le résultat doit être également ?. Le typage des gardes

56. Dans le cas présent, ces informations ne nous intéressent pas, car les données de l'expression principale et sa valeur ne peuvent pas être utilisées par une autre portion de programme.

$$\textsc{Guards}: \frac{\mathcal{E} \vdash_g g_i : r_i}{\mathcal{E} \vdash_g g_1, \ldots, g_n : \vee_i r_i} \qquad \textsc{True}: \frac{}{\mathcal{E} \vdash_g true : \emptyset} \qquad \textsc{Comp}: \frac{\mathcal{E} \vdash_e e : t,\ \mathcal{E},\ \{\}}{\mathcal{E} \vdash_g op\ e : ?}$$

$$\textsc{Type}: \frac{\mathcal{E} \vdash_e e : t,\ \mathcal{E},\ \{\}}{\mathcal{E} \vdash_g Fun(s)\ e : \emptyset}\ \Big(t \sqsubseteq dom(s)\Big)$$

$$\textsc{Constr}: \frac{1 \leq i \leq n \quad \mathcal{E}_{i-1} \vdash_p p_i : t_i,\ \mathcal{E}_i,\ V_i}{\mathcal{E}_0 \vdash_p C(p_1, \ldots, p_n) : TypeC(C)(t_1, \ldots, t_n),\ \mathcal{E}_n,\ \bigcup_i V_i} \qquad \textsc{Cst}: \frac{}{\mathcal{E} \vdash_p c : c,\ \mathcal{E},\ \{\}}$$

$$\textsc{Any}: \frac{}{\mathcal{E} \vdash_p \_ : \top,\ \mathcal{E},\ \{\}} \qquad \textsc{Var1}: \frac{x \in dom(\mathcal{E})}{\mathcal{E} \vdash_p x : \mathcal{E}(x),\ \mathcal{E},\ \{\}} \qquad \textsc{Var2}: \frac{x \notin dom(\mathcal{E})}{\mathcal{E} \vdash_p x : t,\ \mathcal{E}{+}\{x : t\},\ \{\}}$$

$$\textsc{Var3}: \frac{}{\mathcal{E} \vdash_p x^* : t,\ \mathcal{E}{+}\{x : t\},\ \{t\}}$$

Règles 8.4 Le typage des motifs de $\mathcal{F}unc_2$.

utilisant des comparaisons consiste à vérifier que l'expression argument est bien typée et qu'aucune contrainte n'existe sur le type de l'argument car, en μErlang, toutes les valeurs peuvent être comparées. Nous aurions pu ajouter la vérification du fait que l'argument est un couple, mais cette contrainte est inutile puisque la traduction ne génère que des programmes la respectant. En fait, seuls les opérateurs de test de type contraignent les types de manière utile en ajoutant des contraintes sur les domaines des variables.

Le typage d'un motif fournit trois résultats : son type, l'environnement initial auquel les variables qu'il introduit ont été ajoutées et l'ensemble des variables de type étoilées que le motif contient. Sa forme est donc $\mathcal{E}_i \vdash_p p : t,\ \mathcal{E},\ V$ et le traitement des noms effectué lors de la traduction en $\mathcal{F}unc_2$ permet d'uniformiser tous les filtrages. Ainsi, le fonctionnement du filtrage devient similaire à celui de ML-Act excepté le fait que si une variable figure dans l'environnement, elle est remplacée par sa valeur dans le motif. Notons également qu'à la place de l'opérateur d'union disjointe utilisé dans le cadre de $\mathcal{F}unc_1$ pour *fusionner* les environnements issus des sous-motifs, nous transmettons l'environnement résultat du typage d'un sous-motif au sous-motif suivant.

Les expressions

Le calcul des types introduit par les règles 8.5 est plus complexe que celui de ML-Act. Ainsi, la règle $\mathcal{E}_i \vdash_e e : t,\ \mathcal{E},\ E$ type l'expression e dans l'environnement $\mathcal{E}_i$. Elle produit alors un type, un environnement et un effet (qui est un type interface).

Environnements Commençons par examiner le calcul de la partie environnement dans toutes les règles. Chaque expression pouvant introduire de nouvelles variables dans l'environnement, l'environnement résultat du typage d'une sous-expression sert d'environnement pour le typage de la suite du calcul. L'ordre dans lequel les sous-expressions sont

$$\textsc{Constr:}\ \frac{\mathcal{E} \vdash_e e_i : t_i,\ \mathcal{E}_i,\ E_i}{\mathcal{E} \vdash_e C(e_1, \ldots, e_n) : \mathit{TypeC}(C)(t_1, \ldots, t_n),\ \bigvee_{i=1}^{n} \mathcal{E}_i,\ \bigsqcup_{i=1}^{n} E'_i}$$

$$\textsc{Cst:}\ \frac{}{\mathcal{E} \vdash_e c : c,\ \mathcal{E},\ \{\}} \qquad \textsc{Id:}\ \frac{x \in \mathit{dom}(\mathcal{E}_0{+}\mathcal{E})}{\mathcal{E} \vdash_e x : (\mathcal{E}_0{+}\mathcal{E})(x),\ \mathcal{E},\ \{\}}$$

$$\textsc{Seq:}\ \frac{\mathcal{E} \vdash_e e_1 : t_1,\ \mathcal{E}_1,\ E_1 \quad \mathcal{E}_1 \vdash_e e_2 : t_2,\ \mathcal{E}_2,\ E_2}{\mathcal{E} \vdash_e e_1\,; e_2 : t_2,\ \mathcal{E}_2,\ E_1 \sqcup E_2}$$

$$\textsc{Let:}\ \frac{\mathcal{E} \vdash_e e : t_e,\ \mathcal{E}_e,\ E_e \quad \mathcal{E}_e \vdash_p p : t_p,\ \mathcal{E}_p,\ V_p}{\mathcal{E} \vdash_e \texttt{let}\ p\ =\ e : t_e,\ \mathcal{E}_p,\ E_e} \left(t_e \sqsubseteq \mathbf{star}(V_p,\ t_p) \right)$$

$$\textsc{App:}\ \frac{\mathcal{E} \vdash_e f : t_f,\ \mathcal{E}_f,\ E_f \quad \mathcal{E} \vdash_e e : t_e,\ \mathcal{E}_e,\ E_e \quad t'_f = \mathbf{inst}(t_f)}{\mathcal{E} \vdash_e f\ e : t,\ \mathcal{E}_e \vee \mathcal{E}_f,\ E \sqcup E_f \sqcup E_e} \begin{pmatrix} t_e \sqsubseteq \mathit{dom}(t'_f) \\ t'_f \sqsubseteq t_e \xrightarrow{E} t \end{pmatrix}$$

$$\textsc{Fun:}\ \frac{\mathcal{E} \vdash_e f : t_f,\ \mathcal{E}_f,\ E_f \quad \mathcal{E} \vdash_e e : t_e,\ \mathcal{E}_e,\ E_e \quad \alpha = \mathbf{inst}(\mathit{Fun}(t_f))}{\mathcal{E} \vdash_e \mathit{Fun}(f)\ e : t,\ \mathcal{E}_e \vee \mathcal{E}_f,\ E \sqcup E_f \sqcup E_e} \begin{pmatrix} t_f \sqsubseteq \mathit{dom}(\mathcal{E}_t) \\ t_e \sqsubseteq \mathit{dom}(\alpha) \\ \alpha \sqsubseteq t_e \xrightarrow[\mathcal{E}(ego)]{E} t \end{pmatrix}$$

$$\textsc{Abs:}\ \frac{\mathcal{E} \vdash_p p_i : t_i,\ \mathcal{E}_i,\ V_i \quad \mathcal{E}_i \vdash_g g_i : r_i \quad \mathcal{E}_i \vdash_e e_i : t'_i,\ \mathcal{E}'_i,\ E_i \quad \alpha_i = t_i \setminus \bigcup\{t_j \mid j{<}i \wedge r_j{\neq}?\}}{\mathcal{E} \vdash_e \lambda[p_i, g_i.e_i]^{i \in I} : \bigsqcup\{\mathbf{star}(V_i \cup \mathcal{V}(t_i),\ \alpha_i \xrightarrow[r_i]{E_i} t_i) \mid i \in I \wedge \alpha_i \neq \bot\},\ \bigwedge_{i=1}^{n} \mathcal{E}'_i,\ \{\}}$$

Règles 8.5 Le typage des expressions de $\mathcal{F}unc_2$.

évaluées n'est pas connu sauf dans les règles LET et SEQ, ce sont donc les seules règles dans lesquelles l'environnement est transmis. Dans les autres règles contenant des sous-expressions (CONSTR, FUN , APP et ABS), l'ordre d'évaluation de ses sous-expressions n'étant pas défini, nous les typons uniquement dans l'environnement englobant (voir section 7.3 page 165). Ainsi, les variables non accessibles ne figurent pas dans l'environnement de typage. Puis, il faut cumuler les contraintes portant sur toutes les définitions possibles d'une variable. Pour cela, nous utilisons deux opérateurs :

– la disjonction de deux environnements, notée $\mathcal{E}_1 \vee \mathcal{E}_2$, est définie par :

$$\begin{cases} dom(\mathcal{E}_1 \vee \mathcal{E}_2) & = & dom(\mathcal{E}_1) \cup dom(\mathcal{E}_2) \\ (\mathcal{E}_1 \vee \mathcal{E}_2)(x) & = & \mathcal{E}_1(x) \sqcap \mathcal{E}_2(x) \text{ si } x \in dom(\mathcal{E}_1) \cap dom(\mathcal{E}_2) \\ (\mathcal{E}_1 \vee \mathcal{E}_2)(x) & = & \mathcal{E}_1(x) \text{ si } x \in dom(\mathcal{E}_1) \setminus dom(\mathcal{E}_2) \\ (\mathcal{E}_1 \vee \mathcal{E}_2)(x) & = & \mathcal{E}_2(x) \text{ si } x \in dom(\mathcal{E}_2) \setminus dom(\mathcal{E}_1) \end{cases}$$

La disjonction permet de collecter toutes les variables définies dans les sous-expressions d'un constructeur ou d'une application. Remarquons que les valeurs des variables (définies dans plusieurs sous-expressions) doivent alors être égales puisqu'à l'exécution toutes les sous-expressions seront exécutées.

– la fusion de deux environnements qui est notée $\mathcal{E}_1 \wedge \mathcal{E}_2$ et est définie par :

$$\begin{cases} dom(\mathcal{E}_1 \wedge \mathcal{E}_2) & = & dom(\mathcal{E}_1) \cap dom(\mathcal{E}_2) \\ (\mathcal{E}_1 \wedge \mathcal{E}_2)(x) & = & \mathcal{E}_1(x) \sqcup \mathcal{E}_2(x) \text{ si } x \in dom(\mathcal{E}_1 \wedge \mathcal{E}_2) \end{cases}$$

Lors d'un choix, une seule branche est exécutée, donc une variable en sortie du choix peut avoir l'une des valeurs possibles. Ainsi, tous les environnements issus du typage des sous-expressions sont donc fusionnés.

Effets La collecte des effets est cumulative, c'est-à-dire que l'effet d'une expression est la composition (par $\sqcup$) des effets de ses sous-expressions.

Les effets et types se *rencontrent* dans les règles d'application (APP et FUN) et dans celle d'abstraction (ABS). Dans les deux premières règles (APP et FUN), un effet E stocké dans le type d'une fonction se réalise chaque fois que cette fonction est appliquée. Il est alors ajouté à l'effet de l'expression contenant l'appel. La seconde règle ABS décrit la mémorisation d'un effet dans un type lors de la définition d'une fonction. L'effet E_i de chaque corps e_i est stocké dans le type de la fonction.

Types Enfin, décrivons le calcul des types. Les trois premières règles (CONSTR, CST et ID) sont similaires à celle de ML-ACT. Le typage de la séquence SEQ est similaire mais n'ajoute pas de contraintes sur le type résultat du premier élément de la séquence.

Le typage du `let` est voisin de celui de ML-ACT mais la contrainte entre le type de l'expression et celui du motif ne correspond pas au sous-typage usuel. En effet, le filtrage dynamique impose une discipline plus stricte sur la relation entre le type d'une valeur et celui de la variable étoilée qu'elle remplace. En fait, nous allons imposer une égalité entre leurs types. Le problème rencontré est l'éventuelle non monotonie de la substitution dans le cas contraire. Par exemple, examinons le programme suivant : `let` $x = 1,\ \lambda[x.e]\ y$. Les

contraintes de type générées[57] par ce programme (avec l'usuel sous-typage $t_e \sqsubseteq t_p$) sont : $1 \sqsubseteq t_x$, $t_y \sqsubseteq t_x$ et $t_x \to t_e \sqsubseteq t_y \to t$. Ce système se réduit en $\{1 \sqsubseteq t_x,\ t_y \sqsubseteq t_x,\ t_e \sqsubseteq t\}$ et donc l'analyseur ne détecte pas d'erreur si y vaut 2. Or, l'évaluation conduit à la substitution de x par 1, et donc au terme suivant : $\lambda[1.e]\ y$. Si y vaut 2 dans ce programme, une erreur de typage survient puisque $2 \not\sqsubseteq 1$. L'égalité peut être justifiée *a posteriori*, car la définition d'une variable lui confère exactement son type et le sous-typage n'est nécessaire que lors d'une utilisation de cette variable. En fait, le problème est identique à celui des variables abstraites utilisées dans des motifs qui a été présenté dans l'introduction de ce chapitre.

La fonction **star** est utilisée pour étoiler les variables déclarées comme telles par le typage du motif et le type de retour. Les limitations supposées dans l'introduction de ce chapitre sur la forme des valeurs sémantiques substituables (pas de fonction ni d'acteur) font que les motifs ne peuvent contenir que des termes construits dont les feuilles sont des variables, des jokers ou des constantes. La définition exacte de **star** est donc :

$$\mathbf{star}(\{\},\ T) = T \qquad \mathbf{star}(V,\ \alpha) = \begin{cases} \alpha* & \text{si } \alpha \in V \\ \alpha & \text{sinon} \end{cases} \qquad \mathbf{star}(V,\ c) = c$$

$$\mathbf{star}(V,\ C(T_1, \ldots, T_n)) = C(\mathbf{star}(V,\ T_1), \ldots, \mathbf{star}(V,\ T_n))$$

Remarque :
Remarquons que le sous-typage sur les variables (non-étoilées) du motif n'a que peu d'intérêt puisque le `let` *est le seul endroit où elles sont définies. Elles n'auront donc qu'une borne inférieure et donc l'égalité réapparaît lors de la minimisation. Dans les exemples, nous utiliserons donc directement cette borne inférieure comme valeur pour simplifier la présentation.*

L'application de fonction APP est typée presque usuellement, la seule différence est le fait que le type de fonction retenu pour générer les contraintes est instanciation du type calculé par la fonction **inst**. En effet, un type peut contenir des variables étoilées (voir l'introduction de ce chapitre) qu'il faut renommer. Supposons l'existence d'une fonction $\mathcal{VE}(T)$ qui calcul l'ensemble des variables étoilées contenu dans T. La définition précise de la fonction **inst**, si T est un type fonctionnel, est :

$$\mathbf{inst}(T) = [t'_1/t_1] \circ \cdots \circ [t'_n/t_n]\, T \quad \text{avec } \mathcal{VE}(T) = \{t_1, \ldots, t_n\} \text{ et } t'_1, \ldots, t'_n \text{ fraîches}$$

$$\text{et cela provoque } \mathcal{C} = \mathcal{C} \cup [t'_1/t_1] \circ \cdots \circ [t'_n/t_n]\, \mathcal{C} \text{ si } \mathcal{C} \text{ est l'ensemble des contraintes}$$

Si T est une variable de type, l'instanciation est mémorisée, jusqu'à ce que l'on connaisse la forme de la fonction pour la réaliser. Donc, $\mathbf{inst}(t) = \mathbf{inst}(t)_n$ avec l'indice n qui permet de reconnaître les types issus de la même instanciation dans l'ensemble des contraintes.

L'application génère deux contraintes : la première vérifie que l'appel est légal en s'assurant que l'argument est dans le domaine de la fonction ; la seconde permet par décomposition, quand le type du paramètre est connu, de calculer le type résultat et

57. Nous ignorons les effets car il n'interviennent pas.

l'effet de la fonction. Le domaine d'un type *dom* est défini par :

$$dom(T \xrightarrow[?]{E} T') = \bot \qquad dom(T \xrightarrow{E} T') = T \qquad dom(\bigsqcup_i T_i) = \bigsqcup_i dom(T_i)$$

$$dom(\bigsqcap_i T_i) = \bigsqcap_i dom(T_i) \qquad dom(t) = dom(t) \qquad dom(T) = \mathbf{Err}$$

De plus, $T \sqsubseteq t$ implique $dom(T) \sqsubseteq dom(t)$ pour un type T quelconque.

Le typage de l'appel d'une fonction globale (*Fun*) consiste à typer son argument et à vérifier que celui-ci est un atome qui figure dans l'environnement des fonctions globales. Nous détectons ainsi les appels erronés. Remarquons que nous considérons que les *bif* sont traitées par la règle ID. Le type renvoyé est calculé par une fonction auquel nous donnons le même nom et définie par :

$$Fun(s) = \mathcal{E}_t(s) \qquad Fun(\bigsqcup_i T_i) = \bigsqcup_i Fun(T_i) \qquad Fun(\bigsqcap_i T_i) = \bigsqcap_i Fun(T_i)$$

$$Fun(t) = Fun(t) \qquad Fun(T) = \mathbf{Err}$$

De plus, $s \sqsubseteq t$ implique $\mathcal{E}_t(s) \sqsubseteq Fun(t)$ pour tout atome s. Le reste de la règle est similaire à l'application. Le sous-typage des types fonctionnels est étendu aux types de fonctions globales. Il est contravariant par rapport au type de l'*ego* (indiqué sous la flèche). En effet, les *ego*de toutes les éventuelles branches qui vont être exécutées doivent être cumulées dans le type de l'*ego*. Ainsi, par exemple, si $(t_1 \xrightarrow[@\phi_1]{E_1} t'_1 \sqcup t_2 \xrightarrow[@\phi_2]{E_2} t_2) \sqsubseteq t_e \xrightarrow[\mathcal{E}(ego)]{E} t$ alors $\{t_e \sqsubseteq t_1 \sqcap t_2;\ \mathcal{E}(ego) \sqsubseteq @\phi_1 \sqcap @\phi_2;\ E_1 \sqcup E_2 \sqsubseteq E;\ t'_1 \sqcup t'_2 \sqsubseteq t\}$. Ainsi, si $\mathcal{E}(ego) = @\phi$ alors $\phi_1 \sqcap \phi_2 \sqsubseteq \phi$. Donc, si *ego* figurait libre dans une de ces fonctions, il est capturé par l'*ego* courant. Si *ego* n'est pas défini localement mais libre dans la fonction, une erreur se produit. Et, si ni l'un ni l'autre ne sont définis, la contrainte n'est pas générée.

Enfin, analysons le typage d'une abstraction ABS. Pour chaque branche de la fonction, le typage du motif construit l'environnement dans lequel sont typées la garde et l'expression. Les types des branches du filtrage sont réécrits de façon à les rendre disjoints en utilisant l'opération de différence de type : $\setminus$. Sa définition est $t_1 \setminus t_2 \triangleq t_1 \sqcap \overline{t_2}$ où la négation d'un type t est le plus petit type t' tel que $t \sqcup t' = \top$. Les règles 8.6 précisent le calcul de la négation d'un type. Remarquons qu'au cours de la résolution, certaines négations ne sont pas calculées pour augmenter la précision. Ainsi, la négation d'une constante ou d'un caractère n'est pas transformée en $\top$ tant que le processus de résolution n'est pas bloqué.

Ainsi, les données déjà filtrées par les motifs des filtres précédents sont retirées sauf si le typage de la garde renvoie ?. Après cette opération, certains cas de filtrage peuvent disparaître, si le motif devient vide (le compilateur émet alors un avertissement signalant un cas de filtrage non-utilisé). Le type est alors l'union de toutes les branches dont le type de motif n'est pas vide. Notons que pour le calcul du domaine d'une fonction, chaque branche est étiquetée par ? si sa garde peut échouer.

Enfin, la fonction **star** agit sur des types de forme un peu plus générale mais elle agit de la même manière, elle parcourt le terme du type et ajoute une étoile aux variables de type figurant dans l'ensemble qui lui est fourni en premier argument.

$$\overline{t} = \top \text{ si } t \in \mathbb{C} \cup \{\bot,\ char\} \qquad \overline{unit} = num \cup atom \cup tuple \cup list \cup fun \cup pid$$

$$\overline{int} = unit \cup float \cup atom \cup tuple \cup list \cup fun \cup pid$$

$$\overline{float} = unit \cup int \cup atom \cup tuple \cup list \cup fun \cup pid$$

$$\overline{atom} = unit \cup num \cup tuple \cup list \cup fun \cup pid \qquad \overline{tuple} = cst \cup list \cup fun \cup pid$$

$$\overline{nil} = cst \cup tuple \cup cons(\top,\ list) \cup fun \cup pid \qquad \overline{\bigsqcup_i t_i} = \bigsqcap_i \overline{t_i} \qquad \overline{\bigsqcap_i t_i} = \bigsqcup_i \overline{t_i}$$

$$\overline{t_1 \times \cdots \times t_n} = cst \cup list \cup fun \cup pid \cup \bigcup_{i \neq n} tuplei \cup \overline{t_1} \times \underbrace{\top \times \cdots \times \top}_{n-1} \cup \cdots \cup \underbrace{\top \times \cdots \times \top}_{n-1} \times \overline{t_n}$$

$$\overline{cons(t_1, t_2)} = cst \cup tuple \cup fun \cup pid \cup cons(\overline{t_1}, \top) \cup cons(\top, \overline{t_2})$$

Règles 8.6 La négation de type.

8.5 Exemples de typage

Dans cette section, nous allons réaliser le typage de deux programmes pour illustrer le fonctionnement du système de type de μErlang. Un premier exemple présente le typage des filtres dynamiques. Le second, les philosophes, introduit dans le chapitre précédent, illustre la manipulation des acteurs.

Premier exemple

Le premier exemple que nous allons typer est :

$$\begin{array}{l}
\texttt{let}\ h_1 = \lambda x_1^*.((\lambda x_1.\texttt{ok})\ 1)\,; \\
\texttt{let}\ h_2 = \lambda x_2^*.((\lambda[x_2.\texttt{ok},\ _.\texttt{no}])\ 1)\,; \\
\texttt{let}\ g_1 = \lambda(f_1^*, e_1, i_1).(((\lambda f_1.\texttt{yes})\ 0),\ e_1\ i_1)\,; \\
\texttt{let}\ g_2 = \lambda(f_2^*, e_2, i_2).(((\lambda[f_2.\texttt{yes},\ _.\texttt{nop}])\ 0),\ e\ i)\,; \\
\textbf{Suite}
\end{array}$$

Avec **Suite** valant successivement $g_1(1, h_1, 1)$, $g_1(0, h_1, 0)$, $g_1(0, h_1, 1)$ et $g_2(1, h_2, 0)$.

Remarque :
Comme la partie effet de ce programme est vide, nous oublions systématiquement les effets.

La première fonction h_1

$$\dfrac{\dfrac{\dfrac{\mathcal{E} \vdash_p x_1 : t_x^1,\ \mathcal{E},\ \{\}\quad \mathcal{E} \vdash_e \texttt{ok} : \texttt{ok},\ \mathcal{E}}{\mathcal{E} \vdash_e \lambda x_1.\texttt{ok} : t_x^1 \to \texttt{ok},\ \mathcal{E}} \qquad \mathcal{E} \vdash_e 1 : 1,\ \mathcal{E}}{\mathcal{E} \vdash_e (\lambda x_1.\texttt{ok})\ 1 : \alpha_1,\ \mathcal{E}}\ (1)}{\{\} \vdash_p x_1^* : t_x^1,\ \mathcal{E},\ \{t_x^1\}} \qquad \mathcal{E} \vdash_p h_1 : t_h^1,\ \mathcal{E}_1}{\{\} \vdash_e \lambda x_1^*.((\lambda x_1.\texttt{ok})\ 1) : t_x^{1*} \to \alpha_1^*,\ \mathcal{E}}\ (2)$$

$$\{\} \vdash_e \texttt{let}\ h_1\ =\ \lambda x_1^*.((\lambda x_1.\texttt{ok})\ 1) : t_x^{1*} \to \alpha_1^*,\ \mathcal{E}_1$$

Avec

$$\begin{array}{ll}(1) = \{1 \sqsubseteq \mathit{dom}(t_x^1 \to \mathtt{ok});\ t_x^1 \to \mathtt{ok} \sqsubseteq 1 \to \alpha_1\} & \mathcal{E} = \{x_1 : t_x^1\} \\ (2) = \{t_x^{1*} \to \alpha_1^* \sqsubseteq t_h^1\} & \mathcal{E}_1 = \{x_1 : t_x^1;\ h_1 : t_h^1\}\end{array}$$

La deuxième fonction $\mathbf{h_2}$ La dérivation de typage de h_2 est presque identique, elle résulte en $t_x^{2*} \to \alpha_2^*,\ \mathcal{E}_3$ avec :

$$\begin{array}{l}(1) = \{1 \sqsubseteq \mathit{dom}((t_x^2 \to \mathtt{ok}) \sqcup (\top \setminus t_x^2 \to \mathtt{no}));\ (t_x^2 \to \mathtt{ok}) \sqcup (\top \setminus t_x^2 \to \mathtt{no}) \sqsubseteq 1 \to \alpha_2\} \\ (2) = \{t_x^{2*} \to \alpha_2^* \sqsubseteq t_h^2\} \\ \mathcal{E}_2 = \mathcal{E}_1 \cup \{x_2 : t_x^2\} \\ \mathcal{E}_3 = \mathcal{E}_2 \cup \{h_2 : t_h^2\}\end{array}$$

La troisième fonction $\mathbf{g_1}$

$$A_1 = \dfrac{\dfrac{\dfrac{\mathcal{E}_4 \vdash_p f_1 : t_f^1,\ \mathcal{E}_4,\ \{\}}{\mathcal{E}_4 \vdash_e \mathtt{yes} : \mathtt{yes},\ \mathcal{E}_4}}{\mathcal{E}_4 \vdash_e \lambda f_1.\mathtt{yes} : t_f^1 \to \mathtt{yes},\ \mathcal{E}_4} \quad \mathcal{E}_4 \vdash_e 0 : 0,\ \mathcal{E}_4}{\mathcal{E}_4 \vdash_e (\lambda f_1.\mathtt{yes})\ 0 : \alpha_2^1,\ \mathcal{E}_4}\ (1) \quad \dfrac{\dfrac{\mathcal{E}_4 \vdash_e e_1 : t_e^1,\ \mathcal{E}_4}{\mathcal{E}_4 \vdash_e i_1 : t_i^1,\ \mathcal{E}_4}}{\mathcal{E}_4 \vdash_e e_1\ i_1 : \alpha_3^1,\ \mathcal{E}_4}\ (2)}{\mathcal{E}_4 \vdash_e ((\lambda f_1.\mathtt{yes})\ 0),\ e_1\ i_1 : \alpha_2^1 \times \alpha_3^1,\ \mathcal{E}_4}$$

$$\dfrac{\dfrac{\mathcal{E}_3 \vdash_p (f_1^*, e_1, i_1) : t_f^1 \times t_e^1 \times t_i^1,\ \mathcal{E}_4,\ \{t_f^1\} \qquad A_1}{\mathcal{E}_3 \vdash_e \lambda(f_1^*, e_1, i_1).(((\lambda f_1.\mathtt{yes})\ 0),\ e_1\ i_1) : t_f^{1*} \times t_e^1 \times t_i^1 \to \alpha_2^{1*} \times \alpha_3^{1*},\ \mathcal{E}_4} \quad \mathcal{E}_4 \vdash_p g_1 : t_g^1,\ \mathcal{E}_5}{\mathcal{E}_3 \vdash_e \mathtt{let}\ g_1 = \lambda(f_1^*, e_1, i_1).(((\lambda f_1.\mathtt{yes})\ 0),\ e_1\ i_1) : t_f^{1*} \times t_e^1 \times t_i^1 \to \alpha_2^{1*} \times \alpha_3^{1*},\ \mathcal{E}_5}\ (3)$$

Avec

$$\begin{array}{ll}(1) = \{0 \sqsubseteq \mathit{dom}(t_f^1 \to \mathtt{yes});\ t_f^1 \to \mathtt{yes} \sqsubseteq 0 \to \alpha_2^1\} & \mathcal{E}_4 = \mathcal{E}_3 \cup \{f_1^* : t_f^1;\ e_1 : t_e^1;\ i_1 : t_i^1\} \\ (2) = \{t_i^1 \sqsubseteq \mathit{dom}(\mathbf{inst}(t_e^1)_1);\ \mathbf{inst}(t_e^1)_1 \sqsubseteq t_i^1 \to \alpha_3^1\} & \mathcal{E}_5 = \mathcal{E}_4 \cup \{g_1 : t_g^1\} \\ (3) = \{t_f^{1*} \times t_e^1 \times t_i^1 \to \alpha_2^{1*} \times \alpha_3^{1*} \sqsubseteq t_g^1\} & \end{array}$$

La quatrième fonction $\mathbf{g_2}$ La dérivation de typage de g_2 est presque identique, elle résulte en $t_f^{2*} \times t_e^2 \times t_i^2 \to \alpha_2^{2*} \times \alpha_3^{2*},\ \mathcal{E}_7$ avec :

$$\begin{array}{ll}(1) = \{0 \sqsubseteq \mathit{dom}((t_f^2 \to \mathtt{yes}) \sqcup (\top \setminus t_f^2 \to \mathtt{nop}));\ (t_f^2 \to \mathtt{yes}) \sqcup (\top \setminus t_f^2 \to \mathtt{nop}) \sqsubseteq 0 \to \alpha_2^2\} & \\ (2) = \{t_i^2 \sqsubseteq \mathit{dom}(\mathbf{inst}(t_e^2)_2);\ \mathbf{inst}(t_e^2)_2 \sqsubseteq t_i^2 \to \alpha_3^2\} & \mathcal{E}_6 = \mathcal{E}_5 \cup \{f_2^* : t_f^2;\ e_2 : t_e^2;\ i_2 : t_i^2\} \\ (3) = \{t_f^{2*} \times t_e^2 \times t_i^2 \to \alpha_2^{2*} \times \alpha_3^{2*} \sqsubseteq t_g^2\} & \mathcal{E}_7 = \mathcal{E}_6 \cup \{g_2 : t_g^2\}\end{array}$$

Les contraintes À l'issue du typage des quatre fonctions, l'ensemble des contraintes vaut donc :

$$\begin{array}{l}C = \{1 \sqsubseteq t_x^1;\ \mathtt{ok} \sqsubseteq \alpha_1;\ 1 \sqsubseteq \top;\ (t_x^2 \to \mathtt{ok}) \sqcup (\top \setminus t_x^2 \to \mathtt{no}) \sqsubseteq 1 \to \alpha_2;\ 0 \sqsubseteq t_f^1;\ \mathtt{yes} \sqsubseteq \alpha_2^1; \\ \quad t_i^1 \sqsubseteq \mathit{dom}(\mathbf{inst}(t_e^1)_1);\ \mathbf{inst}(t_e^1)_1 \sqsubseteq t_i^1 \to \alpha_3^1;\ 0 \sqsubseteq \top; \\ \quad (t_f^2 \to \mathtt{yes}) \sqcup (\top \setminus t_f^2 \to \mathtt{nop}) \sqsubseteq 0 \to \alpha_2^2;\ t_i^2 \sqsubseteq \mathit{dom}(\mathbf{inst}(t_e^2)_2);\ \mathbf{inst}(t_e^2)_2 \sqsubseteq t_i^2 \to \alpha_3^2\}\end{array}$$

Les appels Examinons maintenant chaque appel et le résultat du typage dans chaque cas. Notons que les deux premiers appels conduisent à une erreur et les deux suivants sont bien typés.

- $\mathbf{g_1}(\mathbf{1}, \mathbf{h_1}, \mathbf{1})$: ajoute les contraintes $1 \times (t_x^{1*} \to \alpha_1^*) \times 1 \sqsubseteq dom(\beta_1^* \times t_e^1 \times t_i^1 \to \beta_2^* \times \beta_3^*)$, $\beta_1^* \times t_e^1 \times t_i^1 \to \beta_2^* \times \beta_3^* \sqsubseteq 1 \times (t_x^{1*} \to \alpha_1^*) \times 1 \to \chi_1 \times \chi_2$ et par instanciation $0 \sqsubseteq \beta_1$, $\texttt{yes} \sqsubseteq \beta_2$ et $\mathbf{inst}(t_e^1)_1 \sqsubseteq t_i^1 \to \beta_3$

$$\begin{aligned} C' &= C \cup \{1 \sqsubseteq \beta_1^*;\ t_x^{1*} \to \alpha_1^* \sqsubseteq t_e^1;\ 1 \sqsubseteq t_i^1;\ \beta_2^* \times \beta_3^* \sqsubseteq \chi_1 \times \chi_2;\ 0 \sqsubseteq \beta_1;\ \texttt{yes} \sqsubseteq \beta_2; \\ &\qquad \mathbf{inst}(t_e^1)_1 \sqsubseteq t_i^1 \to \beta_3\} \\ &= C \cup \{t_x^{1*} \to \alpha_1^* \sqsubseteq t_e^1;\ 1 \sqsubseteq t_i^1;\ \beta_2^* \times \beta_3^* \sqsubseteq \chi_1 \times \chi_2;\ \underline{0 \sqsubseteq 1};\ \texttt{yes} \sqsubseteq \beta_2; \\ &\qquad \mathbf{inst}(t_e^1)_1 \sqsubseteq t_i^1 \to \beta_3\} \end{aligned}$$

  C' contient donc une contrainte contradictoire (soulignée).
- $\mathbf{g_1}(\mathbf{0}, \mathbf{h_1}, \mathbf{0})$: ajoute les contraintes $0 \times (t_x^{1*} \to \alpha_1^*) \times 0 \sqsubseteq dom(\beta_1^* \times t_e^1 \times t_i^1 \to \beta_2^* \times \beta_3^*)$, $\beta_1^* \times t_e^1 \times t_i^1 \to \beta_2^* \times \beta_3^* \sqsubseteq 0 \times (t_x^{1*} \to \alpha_1^*) \times 0 \to \chi_1 \times \chi_2$ et par instanciation $0 \sqsubseteq \beta_1$, $\texttt{yes} \sqsubseteq \beta_2$ et $\mathbf{inst}(t_e^1)_1 \sqsubseteq t_i^1 \to \beta_3$

$$\begin{aligned} C' &= C \cup \{0 \sqsubseteq \beta_1^*;\ t_x^{1*} \to \alpha_1^* \sqsubseteq t_e^1;\ 0 \sqsubseteq t_i^1;\ \beta_2^* \times \beta_3^* \sqsubseteq \chi_1 \times \chi_2;\ 0 \sqsubseteq \beta_1;\ \texttt{yes} \sqsubseteq \beta_2; \\ &\qquad \mathbf{inst}(t_e^1)_1 \sqsubseteq t_i^1 \to \beta_3\} \\ &= C \cup \{\beta_4^* \to \beta_5^* \sqsubseteq t_i^1 \to \beta_3;\ 1 \sqsubseteq \beta_4;\ \texttt{ok} \sqsubseteq \beta_5;\ 0 \sqsubseteq t_i^1;\ \beta_2^* \times \beta_3^* \sqsubseteq \chi_1 \times \chi_2; \\ &\qquad \texttt{yes} \sqsubseteq \beta_2\} \\ &= C \cup \{0 \sqsubseteq t_i^1 \sqsubseteq \beta_4^*;\ \beta_5^* \sqsubseteq \beta_3;\ 1 \sqsubseteq \beta_4;\ \texttt{ok} \sqsubseteq \beta_5;\ \beta_2^* \times \beta_3^* \sqsubseteq \chi_1 \times \chi_2;\ \texttt{yes} \sqsubseteq \beta_2\} \\ &= C \cup \{0 \sqsubseteq t_i^1;\ \beta_5^* \sqsubseteq \beta_3;\ \underline{1 \sqsubseteq 0};\ \texttt{ok} \sqsubseteq \beta_5;\ \beta_2^* \times \beta_3^* \sqsubseteq \chi_1 \times \chi_2;\ \texttt{yes} \sqsubseteq \beta_2\} \end{aligned}$$

  C' contient donc une contrainte contradictoire.
- $\mathbf{g_1}(\mathbf{0}, \mathbf{h_1}, \mathbf{1})$: ajoute les contraintes $0 \times (t_x^{1*} \to \alpha_1^*) \times 1 \sqsubseteq dom(\beta_1^* \times t_e^1 \times t_i^1 \to \beta_2^* \times \beta_3^*)$, $\beta_1^* \times t_e^1 \times t_i^1 \to \beta_2^* \times \beta_3^* \sqsubseteq 0 \times (t_x^{1*} \to \alpha_1^*) \times 1 \to \chi_1 \times \chi_2$ et par instanciation $0 \sqsubseteq \beta_1$, $\texttt{yes} \sqsubseteq \beta_2$ et $\mathbf{inst}(t_e^1)_1 \sqsubseteq t_i^1 \to \beta_3$

$$\begin{aligned} C' &= C \cup \{0 \sqsubseteq \beta_1^*;\ t_x^{1*} \to \alpha_1^* \sqsubseteq t_e^1;\ 1 \sqsubseteq t_i^1;\ \beta_2^* \times \beta_3^* \sqsubseteq \chi_1 \times \chi_2;\ 0 \sqsubseteq \beta_1;\ \texttt{yes} \sqsubseteq \beta_2; \\ &\qquad \mathbf{inst}(t_e^1)_1 \sqsubseteq t_i^1 \to \beta_3\} \\ &= C \cup \{\beta_6^* \to \beta_7^* \sqsubseteq t_i^1 \to \beta_3;\ 1 \sqsubseteq \beta_6;\ \texttt{ok} \sqsubseteq \beta_7;\ 1 \sqsubseteq t_i^1;\ \beta_2^* \times \beta_3^* \sqsubseteq \chi_1 \times \chi_2; \\ &\qquad \texttt{yes} \sqsubseteq \beta_2\} \\ &= C \cup \{1 \sqsubseteq t_i^1 \sqsubseteq \beta_6^*;\ \beta_7^* \sqsubseteq \beta_3;\ 1 \sqsubseteq \beta_6;\ \texttt{ok} \sqsubseteq \beta_7;\ \beta_2^* \sqsubseteq \chi_1;\ \beta_3^* \sqsubseteq \chi_2;\ \texttt{yes} \sqsubseteq \beta_2\} \\ &= C \cup \{1 \sqsubseteq t_i^1;\ \beta_7^* \sqsubseteq \beta_3;\ \texttt{ok} \sqsubseteq \beta_7;\ \beta_2^* \sqsubseteq \chi_1;\ \beta_3^* \sqsubseteq \chi_2;\ \texttt{yes} \sqsubseteq \beta_2\} \end{aligned}$$

  Ce qui conduit par minimisation à l'affichage du type $\texttt{ok} \times \texttt{yes}$.
- $\mathbf{g_2}(\mathbf{1}, \mathbf{h_2}, \mathbf{0})$: ajoute les contraintes $1 \times (t_x^{2*} \to \alpha_2^*) \times 0 \sqsubseteq dom(\beta_8^* \times t_e^2 \times t_i^2 \to \beta_9^* \times \beta_{10}^*)$, $\beta_8^* \times t_e^2 \times t_i^2 \to \beta_9^* \times \beta_{10}^* \sqsubseteq 1 \times (t_x^{2*} \to \alpha_2^*) \times 0 \to \chi_3 \times \chi_4$ et par instanciation $(\beta_8 \to$

$\texttt{yes}) \sqcup (\top \setminus \beta_8 \to \texttt{nop}) \sqsubseteq 1 \to \beta_9$ et $\mathbf{inst}(t_e^2)_2 \sqsubseteq t_i^2 \to \beta_{10}$

$$\begin{aligned}
C' &= C \cup \{1 \sqsubseteq \beta_8^*;\ t_x^{2*} \to \alpha_2^* \sqsubseteq t_e^2;\ 0 \sqsubseteq t_i^2;\ \beta_9^* \times \beta_{10}^* \sqsubseteq \chi_3 \times \chi_4;\\
&\qquad (\beta_8 \to \texttt{yes}) \sqcup (\top \setminus \beta_8 \to \texttt{nop}) \sqsubseteq 0 \to \beta_9;\ \mathbf{inst}(t_e^2)_2 \sqsubseteq t_i^2 \to \beta_{10}\}\\
&= C \cup \{\beta_{11}^* \to \beta_{12}^* \sqsubseteq t_e^2 \sqsubseteq t_i^2 \to \beta_{10};\ (\beta_{11} \to \texttt{ok}) \sqcup (\top \setminus \beta_{11} \to \texttt{no}) \sqsubseteq 1 \to \beta_{12};\ 0 \sqsubseteq t_i^2;\\
&\qquad \beta_9^* \times \beta_{10}^* \sqsubseteq \chi_3 \times \chi_4;\ (1 \to \texttt{yes}) \sqcup (\top \setminus 1 \to \texttt{nop}) \sqsubseteq 0 \to \beta_9\}\\
&= C \cup \{0 \sqsubseteq t_i^2 \sqsubseteq \beta_{11}^*;\ \beta_{12}^* \sqsubseteq \beta_{10};\ (\beta_{11} \to \texttt{ok}) \sqcup (\top \setminus \beta_{11} \to \texttt{no}) \sqsubseteq 1 \to \beta_{12};\\
&\qquad \beta_9^* \sqsubseteq \chi_3;\ \beta_{10}^* \sqsubseteq \chi_4;\ \texttt{nop} \sqsubseteq \beta_9\}\\
&= C \cup \{\beta_{12}^* \sqsubseteq \beta_{10};\ (0 \to \texttt{ok}) \sqcup (\top \setminus 0 \to \texttt{no}) \sqsubseteq 1 \to \beta_{12};\ \beta_9^* \sqsubseteq \chi_3;\ \beta_{10}^* \sqsubseteq \chi_4;\\
&\qquad \texttt{nop} \sqsubseteq \beta_9\}\\
&= C \cup \{\beta_{12}^* \sqsubseteq \beta_{10};\ \texttt{no} \sqsubseteq \beta_{12};\ \beta_9^* \sqsubseteq \chi_3;\ \beta_{10}^* \sqsubseteq \chi_4;\ \texttt{nop} \sqsubseteq \beta_9\}
\end{aligned}$$

Ce qui conduit par minimisation à l'affichage du type $\texttt{no} \times \texttt{nop}$.

Deuxième exemple

Cet exemple consiste à typer le programme des philosophes (exemples 7.2 , 7.3 et 7.4 des pages 159, 160 et 161). Nous allons en détailler uniquement certaines parties qui illustrent la gestion de la communication lors de l'approximation.

Remarque :
Dans cette section, aucun filtre dynamique ne sera utilisé, nous ignorons donc tout ce qui les concerne (instanciation et « étoilage »).

Fourchettes Commençons par typer les comportement des fourchettes ($\texttt{ff}$ et $\texttt{uf}$)[58]. L'environnement $\mathcal{E}_t$ vaut :

$$\begin{aligned}
&\{\texttt{ff}, \lambda[\mathit{nil}.\mathit{receive}(\lambda[(\texttt{get},\ \texttt{S}).\mathit{send}(\texttt{let}\ x = (\texttt{ok},\ \mathit{ego}),\ \texttt{S})\,;x\,;\mathit{Fun}(\texttt{uf})\ \mathit{nil}])]\\
&\ \texttt{uf}, \lambda[\mathit{nil}.\mathit{receive}(\lambda[(\texttt{get},\ \texttt{S}).\mathit{send}(\texttt{let}\ x = \texttt{no},\ \texttt{S});x\,;\mathit{Fun}(\texttt{uf})\ \mathit{nil}, \texttt{leave}.\mathit{Fun}(\texttt{ff})\ \mathit{nil}])]\}
\end{aligned}$$

Le typage du comportement de fourchette libre est le suivant :

$$A_1 = \frac{\dfrac{\mathcal{E}_1 \vdash_e \texttt{let}\ x = (\texttt{ok}, \mathit{ego}) : \texttt{ok} \times @\phi,\ \mathcal{E}_2,\ \{\} \quad \mathcal{E}_1 \vdash_e \texttt{S} : t_S,\ \mathcal{E}_1,\ \{\}}{\mathcal{E}_1 \vdash_e \texttt{let}\ x = (\texttt{ok}, \mathit{ego}),\ \texttt{S} : (\texttt{ok} \times @\phi) \times t_S,\ \mathcal{E}_2,\ \{\}}\,(1)}{\mathcal{E}_1 \vdash_e \mathit{send}(\dots) : \alpha_1,\ \mathcal{E}_2,\ \{\}}$$

$$A_2 = \frac{\mathcal{E} \vdash_p \texttt{get},\ \texttt{S} : \texttt{get} \times t_S,\ \mathcal{E}_1 \qquad \dfrac{A_1 \quad \mathcal{E}_2 \vdash_e x : \texttt{ok} \times @\phi,\ \mathcal{E}_2,\ \{\} \quad \dfrac{\mathcal{E}_2 \vdash_e \texttt{uf} : \texttt{uf},\ \mathcal{E}_2,\ \{\}}{\mathcal{E}_2 \vdash_e \mathit{Fun}(\texttt{uf})\ \mathit{nil} : \alpha_2,\ \mathcal{E}_2,\ E}\,(2)}{\mathcal{E}_1 \vdash_e \mathit{send}(\dots)\,;x\,;\mathit{Fun}(\texttt{uf})\ \mathit{nil} : \alpha_2,\ \mathcal{E}_2,\ E}}{\mathcal{E} \vdash_e \lambda[(\texttt{get},\ \texttt{S}).\mathit{send}(\texttt{let}\ x = (\texttt{ok}, \mathit{ego}),\ \texttt{S})\,;x\,;\mathit{Fun}(\texttt{uf})\ \mathit{nil}] : \texttt{get} \times t_S \xrightarrow{E} \alpha_2,\ \mathcal{E}_2,\ \{\}}$$

$$\frac{\{\} \vdash_p \mathit{nil} : \mathit{nil},\ \{\} \quad \dfrac{\mathcal{E} \vdash_e \mathit{ego} : @\phi,\ \mathcal{E},\ \{\} \qquad A_2}{\mathcal{E} \vdash_e \mathit{receive}(\lambda[(\texttt{get},\ \texttt{S}).\dots.]) : \alpha_2,\ \mathcal{E}_2,\ E_1}}{\mathcal{E}_t \vdash_F \{\texttt{ff} \longmapsto \lambda[\mathit{nil}.\mathit{receive}(\dots)]\}}\,(3)$$

58. Nous utilisons une forme simplifiée de certains noms pour ne pas trop alourdir le typage.

Avec

$$
\begin{aligned}
(1) &= \{\texttt{ok}\times@\phi \sqsubseteq \beta_1;\ t_S \sqsubseteq @Mess(\beta_1);\ unit \sqsubseteq \alpha_1\}\\
(2) &= \{\texttt{uf} \sqsubseteq dom(\mathcal{E}_t);\ nil \sqsubseteq dom(t_{UF});\ t_{UF} \sqsubseteq nil \xrightarrow[@\phi]{E} \alpha_2\}\\
(3) &= \{nil \xrightarrow[@\phi]{E_1} \alpha_2 \sqsubseteq t_{FF};\ E_1 \sqsubseteq \phi\}\\
\mathcal{E} &= \{ego : @\phi\}\\
\mathcal{E}_1 &= \mathcal{E} \cup \{\texttt{S} : t_S\}\\
\mathcal{E}_2 &= \mathcal{E}_1 \cup \{x : \texttt{ok}\times@\phi\}\\
E_1 &= E \sqcup Inter(\texttt{get}\times t_S)
\end{aligned}
$$

Le typage des fourchettes occupées est en grande partie très proche du précédent, nous ne détaillons que la partie de l'arbre qui est fortement différente.

$$
A_3 = \dfrac{\dfrac{\vdots}{\mathcal{E}_4 \vdash_e e_1 : \alpha_4,\ \mathcal{E}_5,\ E_2}(1) \qquad \dfrac{\mathcal{E}_3 \vdash_e \texttt{ff} : \texttt{ff},\ \mathcal{E}_3,\ \{\}}{\mathcal{E}_3 \vdash_e Fun(\texttt{ff})\ nil : \alpha_5,\ \mathcal{E}_3,\ E_3}(2)}{\mathcal{E}_3 \vdash_e \lambda[(\texttt{get}, \texttt{S}).e_1, \texttt{leave}.Fun(\texttt{ff})\ nil] : (\texttt{get}\times t'_S \xrightarrow{E_2} \alpha_4) \sqcup (\texttt{leave} \xrightarrow{E_3} \alpha_5),\ \mathcal{E}_5,\ \{\}}
$$

$$
\dfrac{\mathcal{E}_3 \vdash_p nil : nil,\ \mathcal{E}_3 \quad \dfrac{\mathcal{E}_3 \vdash_e ego : @\phi_1,\ \mathcal{E}_3,\ \{\} \qquad A_3}{\mathcal{E}_3 \vdash_e receive(\lambda[(\texttt{get}, \texttt{S}).\ldots, \texttt{leave}.\ldots]) : \alpha_4 \sqcup \alpha_5,\ \mathcal{E}_5,\ E_4}}{\mathcal{E}_t \vdash_F \{\texttt{uf} \longmapsto \lambda[nil.receive(\ldots)]\}}(3)
$$

Avec

$$
\begin{aligned}
(1) &= \{\texttt{no} \sqsubseteq \beta_2;\ t'_S \sqsubseteq @Mess(\beta_2);\ unit \sqsubseteq \alpha_3;\ \texttt{uf} \sqsubseteq dom(\mathcal{E}_t);\ nil \sqsubseteq dom(t_{UF});\\
&\qquad t_{UF} \sqsubseteq nil \xrightarrow[@\phi_1]{E_2} \alpha_4\}\\
(2) &= \{\texttt{ff} \sqsubseteq dom(\mathcal{E}_t);\ nil \sqsubseteq dom(t_{FF});\ t_{FF} \sqsubseteq nil \xrightarrow[@\phi_1]{E_3} \alpha_5\}\\
(3) &= \{nil \xrightarrow[@\phi_1]{E_4} \alpha_4 \sqcup \alpha_5 \sqsubseteq t_{UF};\ E_4 \sqsubseteq \phi_1\}\\
\mathcal{E}_3 &= \{ego : @\phi_1\}\\
\mathcal{E}_4 &= \mathcal{E}_3 \cup \{\texttt{S} : t'_S\}\\
\mathcal{E}_5 &= \mathcal{E}_4 \cup \{x : \texttt{no}\}\\
E_4 &= E_2 \sqcup Inter(\texttt{get}\times t'_S) \sqcup E_3 \sqcup Inter(\texttt{leave})
\end{aligned}
$$

L'ensemble de contraintes associé est :

$$
\begin{aligned}
C = \{&\texttt{ok}\times@\phi \sqsubseteq \beta_1;\ Mess(\beta_1) \sqsubseteq \phi_S;\ unit \sqsubseteq \alpha_1;\ t_{UF} \sqsubseteq nil \xrightarrow[@\phi]{E} \alpha_2;\ E_1 \sqsubseteq \phi;\\
&nil \xrightarrow[@\phi]{E_1} \alpha_2 \sqsubseteq t_{FF};\ \texttt{no} \sqsubseteq \beta_2;\ Mess(\beta_2) \sqsubseteq \phi'_S;\ unit \sqsubseteq \alpha_3;\ t_{UF} \sqsubseteq nil \xrightarrow[@\phi_1]{E_2} \alpha_4;\\
&t_{FF} \sqsubseteq nil \xrightarrow[@\phi_1]{E_3} \alpha_5;\ E_4 \sqsubseteq \phi_1;\ nil \xrightarrow[@\phi_1]{E_4} \alpha_4 \sqcup \alpha_5 \sqsubseteq t_{UF}\}
\end{aligned}
$$

avec $E_1 = E \sqcup \{\texttt{get} : \bullet t_S\}$, $E_4 = E_2 \sqcup E_3 \sqcup \{\texttt{get} : \bullet t'_S;\ \texttt{leave} : \bullet unit\}$, $@\phi = @\phi_1$, $t_S = @\phi_S$ et $t'_S = @\phi'_S$. La résolution conduit à :

$$
\begin{aligned}
C = \{&\{\texttt{ok} : \circ@\phi\} \sqsubseteq \phi_S;\ unit \sqsubseteq \alpha_1;\ E_4 \sqsubseteq E;\ E_4 \sqsubseteq E_2;\ \alpha_4 \sqcup \alpha_5 \sqsubseteq \alpha_2;\ \alpha_4 \sqcup \alpha_5 \sqsubseteq \alpha_4;\\
&E_1 \sqsubseteq \phi;\ E_1 \sqsubseteq E_3;\ \alpha_2 \sqsubseteq \alpha_5;\ \{\texttt{no} : \circ unit\} \sqsubseteq \phi'_S;\ unit \sqsubseteq \alpha_3;\ E_4 \sqsubseteq \phi_1\}
\end{aligned}
$$

On peut alors conclure que $E = E_1 = E_2 = E_3 = E_4 = \{\texttt{get} : \bullet@\phi_S;\ \texttt{leave} : \bullet \mathit{unit}\}$, $@\phi'_S = @\phi_S$ et $\alpha_2 = \alpha_4 = \alpha_5$. Ce qui donne :

$$C = \{\{\texttt{ok} : \circ@\phi\} \sqsubseteq \phi_S;\ \mathit{unit} \sqsubseteq \alpha_1;\ E \sqsubseteq \phi;\ \{\texttt{no} : \circ \mathit{unit}\} \sqsubseteq \phi_S;\ \mathit{unit} \sqsubseteq \alpha_3\}$$

Il est possible de minimiser/maximiser les variables de type pour les afficher et ainsi de faire apparaître des variables de type récursives (ϕ ci-dessous). Alors, $\mathcal{E}_t$ devient :

$$\{\texttt{ff},\ \texttt{uf} : \mathit{nil} \xrightarrow[@\phi]{\phi} \alpha\} \text{ avec } \phi = \{\texttt{get} : \bullet@\{\texttt{ok} : \circ@\phi;\ \texttt{no} : \circ \mathit{unit}\};\ \texttt{leave} : \bullet \mathit{unit}\}$$

Philosophes Le typage des philosophes que nous ne décrivons pas en détail conduit à :

$$\{\texttt{hp},\ \texttt{tp},\ \texttt{np} : @\psi,\ [@\varphi; @\varphi] \xrightarrow{\psi} \beta;\ \ \texttt{op} : @\psi,\ [@\varphi; @\varphi; @\{\texttt{leave} : \circ \mathit{unit}\}] \xrightarrow{\psi} \beta\} \text{ avec}$$
$$\varphi = \{\texttt{get} : \circ@\psi;\ \texttt{leave} : \circ \mathit{unit}\},\ \psi = \{\texttt{hungry} : \bullet \mathit{unit};\ \texttt{no} : \bullet \mathit{unit};\ \texttt{ok} : \bullet@\{\texttt{leave} : \circ \mathit{unit}\}\}$$

Programme principal Nous allons montrer le typage d'une création d'un acteur avant de donner les contraintes générées par le programme principal.

$$\dfrac{\dfrac{\dfrac{\{\} \vdash_p a : t_a,\ \mathcal{E}_1,\ \{\}}{\{\} \vdash_e \mathit{new} : @\psi_1,\ \{\},\ \{\}}}{\{\} \vdash_e \texttt{let}\ a = \mathit{new} : @\psi_1,\ \mathcal{E}_1,\ \{\}}\ (1) \qquad \dfrac{\dfrac{\vdots}{\mathcal{E}_2 \vdash_e \mathit{Fun}(\texttt{ff})\ \mathit{nil} : \alpha,\ \mathcal{E}_2,\ \phi}\ (2)}{\mathcal{E}_1 \vdash_p \mathit{ego} : t_{\mathit{ego}},\ \mathcal{E}_2,\ \{\}}}{\{\} \vdash_e \mathit{init}(\texttt{let}\ a = \mathit{new}, \lambda \mathit{ego}.(\mathit{Fun}(\texttt{ff})\ \mathit{nil})) : r,\ \mathcal{E}_2,\ \{\}}\ (3)$$
$$\dfrac{\mathcal{E}_2 \vdash_e a : t_a,\ \mathcal{E}_2,\ \{\}}{\{\} \vdash_e \mathit{init}(\texttt{let}\ a = \mathit{new}, \lambda \mathit{ego}.(\mathit{Fun}(\texttt{ff})\ \mathit{nil}))\,; a : t_a,\ \mathcal{E}_2,\ \{\}}$$

avec $(1) = \{@\psi_1 \sqsubseteq t_a\}$, $(2) = \{t_{\mathit{ego}} \sqsubseteq @\phi\}$ et $(3) = \{\mathit{unit} \sqsubseteq r;\ @\psi_1 \times (t_{\mathit{ego}} \xrightarrow{\phi} \alpha) \sqsubseteq @\psi^1 \times (@\psi^1 \xrightarrow{\psi^1} \top)\}$. Ce qui impose $t_{\mathit{ego}} = @\phi_{\mathit{ego}}$ et $t_a = @\phi_a$ ainsi que les contraintes $\{\phi \sqsubseteq \phi_{\mathit{ego}} \sqsubseteq \psi^1 \sqsubseteq \psi_1;\ \phi_a \sqsubseteq \psi_1;\ \mathit{unit} \sqsubseteq r\}$.
Le programme principal amènera donc les contraintes suivantes :

$$\begin{aligned} C = \{&\phi \sqsubseteq \phi_{\mathit{ego}} \sqsubseteq \psi^1 \sqsubseteq \psi_1;\ \phi_a \sqsubseteq \psi_1;\ \mathit{unit} \sqsubseteq r;\ \phi \sqsubseteq \phi^2_{\mathit{ego}} \sqsubseteq \psi^2 \sqsubseteq \psi_2;\ \phi_b \sqsubseteq \psi_2;\ \mathit{unit} \sqsubseteq r^2;\\ &\psi \sqsubseteq \phi^3_{\mathit{ego}} \sqsubseteq \psi^3 \sqsubseteq \psi_3;\ \phi_c \sqsubseteq \psi_3;\ \mathit{unit} \sqsubseteq r^3;\ \psi \sqsubseteq \phi^4_{\mathit{ego}} \sqsubseteq \psi^4 \sqsubseteq \psi_4;\ \phi_d \sqsubseteq \psi_4;\ \mathit{unit} \sqsubseteq r^4;\\ &@\phi_a \sqsubseteq @\varphi;\ @\phi_b \sqsubseteq @\varphi;\ \{\texttt{hungry} : \circ \mathit{unit}\} \sqsubseteq \phi_c;\ \{\texttt{hungry} : \circ \mathit{unit}\} \sqsubseteq \phi_d\} \end{aligned}$$

Ce qui conduit donc pour prendre en compte et vérifier les réceptions de respectivement `hungry`, `leave` et `get` à :

$$\begin{aligned} C' = \{&\circ \mathit{unit} \sqsubseteq \bullet t_1;\ \bullet \mathit{unit} \sqsubseteq \bullet t_1;\ \circ \mathit{unit} \sqsubseteq \bullet t_2;\ \bullet \mathit{unit} \sqsubseteq \bullet t_2;\ \circ@\psi \sqsubseteq \bullet t_3;\\ &\bullet@\{\texttt{ok} : \circ@\phi;\ \texttt{no} : \circ \mathit{unit}\} \sqsubseteq \bullet t_3\} \end{aligned}$$

qui se réduit en :

$$\begin{aligned}
C' &= \{unit \sqsubseteq t_1 \sqsubseteq unit;\ unit \sqsubseteq t_2 \sqsubseteq unit;\ @\psi \sqsubseteq t_3 \sqsubseteq @\{\texttt{ok} : \circ@\phi;\ \texttt{no} : \circ unit\}\} \\
&= \{\{\texttt{ok} : \circ@\phi;\ \texttt{no} : \circ unit\} \sqsubseteq \{\texttt{hungry}: \bullet unit;\ \texttt{no}: \bullet unit;\ \texttt{ok}: \bullet@\{\texttt{leave}: \circ unit\}\} \\
&= \{\circ@\phi \sqsubseteq \bullet@\{\texttt{leave}: \circ unit\};\ \circ unit \sqsubseteq \bullet unit\} \\
&= \{\{\texttt{leave}: \circ unit\} \sqsubseteq \{\texttt{get} : \bullet@\{\texttt{ok} : \circ@\phi;\ \texttt{no} : \circ unit\};\ \texttt{leave} : \bullet unit\}\} \\
&= \{\circ unit \sqsubseteq \bullet unit\} \\
&= \{\}
\end{aligned}$$

Le programme est donc correctement typé et ne peut donc pas conduire à une erreur lors de son exécution.

Notons que si le philosophe dans l'état `nf` envoyait un message `end` (au lieu de `leave`) à l'argument reçu dans le message `ok` alors ψ vaudrait :

$$\{\texttt{hungry}:\bullet unit;\ \texttt{no}:\bullet unit;\ \texttt{ok}:\bullet@\{\texttt{end}:\circ unit\}\}$$

La résolution ci-dessus devient alors :

$$\begin{aligned}
C' &= \{\{\texttt{ok} : \circ@\phi;\ \texttt{no} : \circ unit\} \sqsubseteq \{\texttt{hungry}: \bullet unit;\ \texttt{no}: \bullet unit;\ \texttt{ok}: \bullet@\{\texttt{end}: \circ unit\}\} \\
&= \{\circ@\phi \sqsubseteq \bullet@\{\texttt{end}: \circ unit\};\ \circ unit \sqsubseteq \bullet unit\} \\
&= \{\{\texttt{end}: \circ unit\} \sqsubseteq \{\texttt{get} : \bullet@\{\texttt{ok} : \circ@\phi;\ \texttt{no} : \circ unit\};\ \texttt{leave} : \bullet unit\}\}
\end{aligned}$$

Et l'erreur de programmation est alors détectée.

8.6 La résolution des contraintes

Le processus de résolution des contraintes issues de l'inférence repose sur le même principe que celui introduit dans le chapitre 4. La résolution est donc incrémentale, nous partons d'un graphe de contrainte vide (donc clos) et à chaque ajout de contrainte, nous maintenons le graphe clos. L'impossibilité de maintenir clos le graphe des contraintes signifie qu'une contrainte contradictoire a été ajoutée, une erreur de typage est alors signalée. En fin d'analyse, une étape supplémentaire vérifie que les types des acteurs effectivement créés ne contiennent plus de messages reçus non installés, *i.e.* le type ne contient que des $\bullet$. Remarquons que, dans le cadre de μErlang, l'unification réintroduite pour ML-ACT est abandonnée sur les types (mais conservée sur les interfaces et présences).

Durant la résolution, nous utilisons le terme de présence † (déjà défini dans ML-ACT et qui indique l'absence d'une étiquette m dans une interface I). Enfin, nous définissons à nouveau deux formes de types : les positifs et les négatifs. Contrairement à ce qui se passe pour ML-ACT, une union peut apparaître dans une borne supérieure.

$$\begin{aligned}
T^+ &::= t \mid \mathcal{C}(A_1^{s(1)}, \ldots, A_n^{s(n)}) \mid \bigsqcup\{T^+, \ldots, T^+\} \\
T^- &::= t \mid \mathcal{C}(A_1^{-s(1)}, \ldots, A_n^{-s(n)}) \mid \bigsqcap\{T^-, \ldots, T^-\} \mid \bigsqcup\{T^-, \ldots, T^-\} \\
A^+ &::= T^+ \mid I^+ \\
A^- &::= T^- \mid I^- \\
I^+ &::= i \mid \{\} \mid \{m : P^+,\ I^+\} \mid \bigsqcup\{I^+, \ldots, I^+\} \\
I^- &::= i \mid \{\} \mid \{m : P^-,\ I^-\} \mid \bigsqcap\{I^-, \ldots, I^-\} \mid \bigsqcup\{I^-, \ldots, I^-\} \\
P^+ &::= p \mid \dagger \mid \bullet T^- \mid \circ T^+ \mid \bigsqcup\{P^+, \ldots, P^+\} \\
P^- &::= p \mid \dagger \mid \bigsqcap\{P^-, \ldots, P^-\} \mid \bigsqcup\{P^-, \ldots, P^-\}
\end{aligned}$$

où chaque constructeur de type $\mathcal{C}$ d'arité n dispose d'une fonction $s(\mathcal{C})$ qui à un indice i associe la variance ($+$ ou $-$) du constructeur par rapport à sa i^{e} composante. Si l'identité du constructeur est évidente d'après le contexte, la variance est notée $s(i)$ au lieu de $s(\mathcal{C})(i)$. Dans la suite, les signes des types ne sont pas détaillés lorsqu'ils sont clairs d'après le contexte.

Remarquons que les présences négatives ne contiennent pas de messages, cette limitation vient du fait que la résolution des présences ne s'effectue pas incrémentalement mais après la collecte de toutes les contraintes. Donc, durant la phase de résolution effectuée *à la volée*, ces formes de présence ne figurent qu'en partie gauche d'une contrainte (elles sont issues des règles de typage des constructeurs). Ce report en fin de résolution est dû à l'applatissement effectué entre les messages reçus et les messages installés. En effet, une contrainte $\circ T_1^+ \sqsubseteq \bullet T_2^+$ se décompose en $T_1^+ \sqsubseteq T_2^+$ et peut donc provoquer l'apparition d'union dans les bornes supérieures. Un tel choix complique considérablement la forme des types et la résolution (voir [Thi94]). Or, les informations obtenues par cette décomposition ne sont pas très intéressantes lors des résolutions intermédiaires. Notre stratégie peut être rapprochée d'une variante basée sur des contraintes ensemblistes présentée dans [DPCS00]. Dans cette variante, le type d'un acteur est séparé en deux parties (ϕ, ψ) où ϕ collecte les messages reçus et ψ les messages installés. En fin de résolution, le système vérifie alors que $\phi \sqsubseteq \psi$ quitte à relancer une étape de résolution du nouveau système de contraintes.

union et intersection

Dans le graphe des contraintes, chaque borne supérieure (resp. inférieure) est recalculée lors de l'ajout d'une nouvelle contrainte par *intersection* $\sqcap$ (resp. *union* $\sqcup$) de l'éventuelle nouvelle borne avec l'ancienne valeur. Pour cela, les opérateurs $\sqcap$ et $\sqcup$ sont définis sur les types, les interfaces et les présences (respectivement négatifs et positifs). Notons que ces opérateurs doivent respecter une discipline de *sorte* pour que le résultat soit correct. En effet, trois sortes sont définies : les types ($\mathcal{T}$), les interfaces ($\mathcal{I}$) et les présences ($\mathcal{P}$). Et, il n'est pas possible de faire l'union de deux entités de sortes différentes. Si c'est le cas, le processus de résolution du graphe des contraintes échoue (le symbole de l'erreur d'exécution **Err** est alors utilisé). De plus, notre approche utilise l'unification sur les constructeurs de tête d'un type. Faire l'union, l'intersection ou même comparer deux types dont le constructeur de tête est différent provoque donc une erreur. Ainsi, pour chaque constructeur $\mathcal{C}$, l'ensemble de sorte $\mathcal{T}(\mathcal{C})$ est défini (l'union des $\mathcal{T}(\mathcal{C})$ forme la sorte des types). Une variable pouvant être de toutes les sortes possibles, le processus de résolution utilise une notion de variable de sorte Σ et un environnement de sorte $\mathcal{S}$. Cet environnement est une fonction de l'ensemble des variables (de type, d'interface et de présence : $\mathbb{V}_t \cup \mathbb{V}_i \cup \mathbb{V}_p$) dans l'ensemble des sortes. Pour vérifier que les différentes sortes éventuellement attribuées à une variable coïncident, une sommation ($+$) d'environnement est définie par :

$$\{t_1 : \mathcal{S}_1\} + \{t_2 : \mathcal{S}_2\} = \begin{cases} \{t_1 : \mathcal{S}_1;\ t_2 : \mathcal{S}_2\} & \text{si } t_1 \neq t_2 \\ \{t_1 : \mathcal{S}_1\} & \text{si } t_1 = t_2 \text{ et } \mathcal{S}_1 = \mathcal{S}_2 \\ \textbf{Err} & \text{si } t_1 = t_2 \text{ et } \mathcal{S}_1 \neq \mathcal{S}_2 \end{cases}$$

La sorte d'un type est déterminée par l'ensemble des règles de déductions ci-dessous.

Une variable peut être de toutes les sortes possibles et la valeur de sa sorte est mémorisée. Chaque constructeur possède une signature de sorte que ses arguments doivent respecter. La sorte de la i^{e} composante d'un constructeur $\mathcal{C}$ est déterminée par la notation $\mathcal{S}_i(\mathcal{C})$.

$$\frac{}{\{t : \mathcal{T}(\mathcal{C})\} \vdash t : \mathcal{T}(\mathcal{C})} \qquad \frac{}{\{t : \mathcal{I}\} \vdash t : \mathcal{I}} \qquad \frac{}{\{t : \mathcal{P}\} \vdash t : \mathcal{P}}$$

$$\frac{\forall i \in 1..n \;\; \mathcal{S}_i \vdash T_i : \mathcal{S}_i(\mathcal{C})}{\overset{n}{\underset{i=1}{+}} \mathcal{S}_i \vdash \mathcal{C}(T_1, \ldots, T_n) : \mathcal{T}(\mathcal{C})}$$

$$\frac{\mathcal{S}_1 \vdash P : \mathcal{P} \qquad \mathcal{S}_2 \vdash I : \mathcal{I}}{\mathcal{S}_1 + \mathcal{S}_2 \vdash \{m : P,\ I\} : \mathcal{I}} \qquad \frac{}{\{\} \vdash \{\} : \mathcal{I}}$$

$$\frac{}{\{\} \vdash \dagger : \mathcal{P}} \qquad \frac{\mathcal{S} \vdash T : \mathcal{T}}{\mathcal{S} \vdash \bullet T : \mathcal{P}} \qquad \frac{\mathcal{S} \vdash T : \mathcal{T}}{\mathcal{S} \vdash \circ T : \mathcal{P}}$$

Pour simplifier l'écriture, tous les opérateurs sont surchargés pour qu'ils aient la même écriture sur les trois sortes. Ces opérateurs sont idempotents (1), commutatifs (2), associatifs (3), distributif l'un envers l'autre (4) et enfin, sont l'inverse l'un de l'autre (5) :

$$\begin{array}{llll}
T \sqcup T = T & & T \sqcap T = T & (1)\\
T_1 \sqcup T_2 = T_2 \sqcup T_1 & & T_1 \sqcap T_2 = T_2 \sqcap T_1 & (2)\\
T_1 \sqcup (T_2 \sqcup T_3) = (T_1 \sqcup T_2) \sqcup T_3 & & T_1 \sqcap (T_2 \sqcap T_3) = (T_1 \sqcap T_2) \sqcap T_3 & (3)\\
T_1 \sqcup (T_2 \sqcap T_3) = (T_1 \sqcup T_2) \sqcap (T_1 \sqcup T_3) & & T_1 \sqcap (T_2 \sqcup T_3) = (T_1 \sqcap T_2) \sqcup (T_1 \sqcap T_3) & (4)\\
\sqcup^+ = \sqcup & \sqcup^- = \sqcap & \sqcap^+ = \sqcap & \sqcap^- = \sqcup \quad (5)
\end{array}$$

Les constructeurs $\bigsqcup$ et $\bigsqcap$ ajoutés à la syntaxe des types sont inverses l'un de l'autre, et ne sont pas binaires mais prennent en paramètre une liste de type. Nous écrivons les interfaces un peu différemment, sous la forme d'une liste de couple étiquette/présence et éventuellement d'une variable dénotant une éventuelle extension : $\{(m_j : P_j)_{j \in J}\}$ ou $\{(m_j : P_j)_{j \in J},\ i\}$. Pour faciliter l'écriture des différentes manipulations des interfaces, une indexation des éléments de $\mathbb{M}$ est définie. Ainsi dans toutes les interfaces l'étiquette m_j (pour un i fixé) est identique.

Nous utilisons les propriétés introduites précédemment et ne donnons que la partie nécessaire au calcul des unions et intersections. Nous employons le terme d'union, mais les formules contenant un exposant de signe sont valables dans le cas d'intersection également. De plus, le comportement des trois sortes de variables étant identique, nous les manipulons de la même manière sous le nom v.

Le graphe des contraintes mémorise la sorte des variables (quitte à utiliser des variables de sorte). L'union suit les neuf règles suivantes dans lesquelles l'ordre d'application est important :

$$\begin{array}{lll}
(1)\ \bot \sqcup T = T & (4) \quad \mathit{char} \sqcup \mathit{int} = \mathit{int} & (7)\ \bigsqcup S_1 \sqcup \bigsqcup S_2 = \bigsqcup(S_1 \cup S_2)\\
(2)\ \top \sqcup T = \top & (5) \quad s \sqcup \mathit{atom} = \mathit{atom} & (8)\ \ T \sqcup \bigsqcup S = \bigsqcup(\{T\} \cup S)\\
(3)\ T \sqcup T = T & (6) \times(T_1, \ldots, T_n) \sqcup \mathit{tuple} = \mathit{tuple} & (9)\ \ T_1 \sqcup T_2 = \bigsqcup\{T_1,\ T_2\}
\end{array}$$

Remarque :
Notons que la règle (9) capture toutes les autres formes d'union, par exemple $(T_1 \to T_2) \sqcup (T'_1 \to T'_2) = \bigsqcup\{(T_1 \to T_2),\ (T'_1 \to T'_2)\}$.

L'intersection suit des règles un peu plus complexes. En effet, les intersections sont distribuées sur les sous-termes car elles permettent d'obtenir des informations par « unification » (contrairement aux unions). On peut montrer qu'une intersection, une fois calculée, est composée de l'intersection d'un type (éventuellement absent) et d'une liste de variables. Nous allons donc noter une intersection $\bigsqcap(T, S)$, si T est le type et S la liste de variables. Pour le calcul d'une intersection, les règles ci-dessous doivent être appliquées dans l'ordre. Pour simplifier l'écriture des règles, nous considérons que si le type est absent, il vaut $\top$ (ce qui est correct d'après la règle (1)).

$$
\begin{array}{lll}
(1)\ \bot \sqcap T = \bot & (4) & \mathit{char} \sqcap \mathit{int} = \mathit{char} \\
(2)\ \top \sqcap T = T & (5) & s \sqcap \mathit{atom} = s \\
(3)\ T \sqcap T = T & (6) & \times(T_1, \ldots, T_n) \sqcap \mathit{tuple} = \times(T_1, \ldots, T_n)
\end{array}
$$

$$
\begin{array}{lll}
(7)\ \mathfrak{C}(T_1, \ldots, T_n) \sqcap \mathfrak{C}'(T'_1, \ldots, T'_m) = \bot & & \\
(8)\ \mathfrak{C}(T_1, \ldots, T_n) \sqcap \mathfrak{C}(T'_1, \ldots, T'_n) & = & \mathfrak{C}(T_1 \sqcap^{s(1)} T'_1, \ldots, T_n \sqcap^{s(n)} T'_n) \\
(9)\ \bigsqcap(T_1, S_1) \sqcap \bigsqcap(T_2, S_2) & = & (T_1 \sqcap T_2) \sqcap \bigsqcap(\top, S_1 \cup S_2) \\
(10)\ T \sqcap \bigsqcup_i T_i & = & \sqcup_i (T_i \sqcap T) \\
(11)\ t \sqcap \bigsqcap(T, S) & = & \bigsqcap(T, \{t\} \cup S) \\
(12)\ T_1 \sqcap \bigsqcap(T_2, S) & = & (T_1 \sqcap T_2) \sqcap \bigsqcap(\top, S) \\
(13)\ T_1 \sqcap T_2 & = & \bigsqcap\{T_1,\ T_2\}
\end{array}
$$

Les règles 8.7 de calcul des unions, des intersections et de simplification des contraintes sur les interfaces et les présences.

Sur les interfaces, les deux opérateurs d'union et d'intersection ont un comportement spécifique sur une interface vide. Notons que dans le cas d'une union avec une interface vide, une information sur la sorte de v est également mémorisée. L'union et l'intersection ont un comportement similaire sur les autres formes d'interface : Remarquons que si la *queue* d'une interface est vide dans la deuxième formule (7), l'interface, à laquelle elle est associée, est vide. Ainsi, toutes les étiquettes (p_j et p'_j) et la queue (i'_1 ou i'_2) sont vides.

somme d'ensemble de contraintes

L'ajout d'une union dans les bornes supérieures rend la résolution plus complexe. Pour résoudre un système C auquel une contrainte de la forme $t \sqsubseteq t_1 \sqcup t_2$ est ajoutée, nous divisons la contrainte en deux ($t \sqsubseteq t_1$ et $t \sqsubseteq t_2$) et générons deux systèmes à résoudre $C \cup \{t \sqsubseteq t_1\}$ et $C \cup \{t \sqsubseteq t_2\}$. Trois situations peuvent alors se présenter :

- aucun des deux systèmes n'a de solution et alors, $C \cup \{t \sqsubseteq t_1 \sqcup t_2\}$ n'en a pas ;
- un seul des deux systèmes à une solution et alors, $C \cup \{t \sqsubseteq t_1 \sqcup t_2\}$ lui est équivalent ;
- les deux systèmes ont une solution et le processus de résolution doit se poursuivre avec les deux systèmes en parallèle.

$$\{\} \sqcup I = I \quad (4) \qquad\qquad \{\} \sqcap I = \{\} \quad (5)$$

$$\{(m_j : P_j)_{j\in J}, I\} \sqcup^s v = \{(m_j : P_j \sqcup^s P'_j)_{j\in J}, I \sqcup^s i\} \text{ et } v = \{(m_j : P'_j)_{j\in J}, i\} \quad (6)$$

$$\{(m_j : P_j)_{j\in J_1}, I_1\} \sqcup^s \{(m_j : P'_j)_{j\in J_2}, I_2\} = \{(m_j : P_j \sqcup^s P'_j)_{j\in J_1\cup J_2}, i'_1 \sqcup^s i'_2\} \quad (7)$$

$$\text{et } \begin{cases} I_1 = \{(m_j : p_j)_{j\in J_2\setminus J_1}, i'_1\} \\ I_2 = \{(m_j : p'_j)_{j\in J_1\setminus J_2}, i'_2\} \end{cases} \text{ où } \begin{cases} \forall j \in J_2 \setminus J_1 & P_j = p_j \\ \forall j \in J_1 \setminus J_2 & P'_j = p'_j \end{cases}$$

$$\dagger \sqcap P = \dagger \quad (a) \qquad \dagger \sqcup P = P \quad (b) \qquad \circ T_1 \sqcup \bullet T_2 = \bullet T_2 \text{ et } T_1 \sqsubseteq T_2 \quad (c)$$

$$\begin{array}{lcll} \bullet T \sqcup v & = & \bigsqcup\{\bullet T,\ v\} & (d) \\ \bullet T_1 \sqcup \bullet T_2 & = & \bullet(T_1 \sqcap T_2) & (e) \end{array} \qquad \begin{array}{lcll} \circ T \sqcup v & = & \bigsqcup\{\circ T,\ v\} & (f) \\ \circ T_1 \sqcup \circ T_2 & = & \circ(T_1 \sqcup T_2) & (g) \end{array}$$

$$\{(m_j : P_j)_{j\in J}, I\} \sqcup^s \bigsqcup_j^s I_j = \sqcup_j^s(\{(m_j : P_j)_{j\in J}, I\} \sqcup^s I_j)$$

$$P_1 \sqcup \bigsqcup(\{P_2\} \cup S) = \bigsqcup((P_1 \sqcup P_2) \cup S) \quad (h)$$

Règles 8.7 Union et intersection d'interface et de présence.

L'idée est donc de remplacer la résolution d'un système de contraintes par la résolution d'un ensemble de systèmes de contraintes (voir [AWL94]).

Dans la pratique, il est assez souvent possible de déterminer rapidement qu'un des systèmes générés n'a pas de solution. Par exemple, si la contrainte est $t \sqsubseteq \mathit{int} \sqcup \mathit{float}$ et que t est un entier, la deuxième contrainte est trivialement fausse. Un système sans solution est noté de la même manière que l'erreur d'exécution **Err** et nous définissons un opérateur de *somme de systèmes de contraintes*, noté $+$, pour exprimer ce processus de division de systèmes.

Définition 8.3 (Somme de systèmes de contraintes) :
*La somme de deux systèmes de contraintes C_1 et C_2 est un multi-système de contraintes. C'est un opérateur commutatif dont **Err** est l'élément neutre qui est distributif par rapport à l'union de systèmes de contraintes. Soit :*

$$\begin{cases} C_1 + C_2 & = & C_2 + C_1 \\ \mathbf{Err} + C & = & C \\ C \cup (C_1 + C_2) & = & (C \cup C_1) + (C \cup C_2) \end{cases}$$

simplification

Chaque contrainte inférée est simplifiée avant d'être ajoutée au graphe de contrainte. Pour cela, la résolution suit le même principe que celle de ML-ACT, la fonction *subc* est définie. Elle calcule à partir d'une contrainte un multi-système de contraintes simples.

Tout d'abord, les quatre règles de simplifications générales de ML-Act (A, B, C et D) s'appliquent également :

$$subc(T \sqsubseteq \bigsqcap_{j\in J} T_j) = \bigcup_{j\in J} subc(T \sqsubseteq T_j) \quad (A) \qquad subc(\bigsqcup_{j\in J} T_j \sqsubseteq T) = \bigcup_{j\in J} subc(T_j \sqsubseteq T) \quad (B)$$

$$subc(T \sqsubseteq T) = \{\} \quad (C) \qquad subc(v_1 \sqsubseteq v_2) = \{v_1 \sqsubseteq v_2\} \quad (D)$$

Il faut alors ajouter trois autres règles générales :

$$subc(T \sqsubseteq \bigsqcup_{j\in J} T_j) = \underset{j\in J}{+} subc(T \sqsubseteq T_j) \quad (E) \qquad subc(v \sqsubseteq \bot) = \textbf{Err} \quad (F)$$

$$subc(\top \sqsubseteq v) = \{\} \text{ et } v = \top \quad (G)$$

Notons que dans la règle (F), au lieu de donner $\bot$ comme valeur à v, nous signalons une erreur. En effet, nous ne souhaitons pas avoir de programmes contenant des données de type indéfini et donc si une valeur devient indéfinie le typage échoue.

Puis, la relation d'ordre sur les types rend certaines contraintes triviales. Chaque constante (entier, réel ou caractère) est contenue dans son type de base et :

$$subc(char \sqsubseteq int) = \{\} \quad (\alpha) \qquad subc(s \sqsubseteq atom) = \{\} \quad (\beta)$$

$$subc(\times(T_1, \ldots, T_n) \sqsubseteq tuple) = \{\} \quad (\chi)$$

La comparaison d'une variable avec un constructeur devient :

$$subc(v \sqsubseteq^s \mathcal{C}(T_1, \ldots, T_n)) = \{v \sqsubseteq^s \mathcal{C}(T_1, \ldots, T_n)\} \quad (1)$$

puisque nous n'utilisons plus d'unification sur les types.

Lors de la comparaison de termes construits, soit les constructeurs de tête sont identiques et la contrainte est propagée sur les sous-termes, soit ils sont différents et le processus de résolution échoue :

$$subc(\mathcal{C}(T_1, \ldots, T_n) \sqsubseteq \mathcal{C}(T'_1, \ldots, T'_n)) = \bigcup_{j\in 1..n} subc(T_j \sqsubseteq^{s(j)} T'_j) \quad (4)$$

$$subc(\mathcal{C}_1(T_1, \ldots, T_n) \sqsubseteq \mathcal{C}_2(T'_1, \ldots, T'_m)) = \textbf{Err} \text{ si } \quad \mathcal{C}_1 \neq \mathcal{C}_2 (5)$$

Dans le cas des interfaces, soit le membre gauche de la contrainte est vide et cette contrainte disparaît, soit il n'est pas vide et la contrainte est décomposée selon la forme de l'interface :

$$subc(\{\} \sqsubseteq I) = \{\} \quad (6)$$

$$subc(\{(m_j : P_j)_{j\in J}, I\} \sqsubseteq \{\}) = subc(I \sqsubseteq \{\}) \cup \bigcup_{j\in J} subc(P_j \sqsubseteq \dagger) \quad (7)$$

$$subc(\{(m_j : P_j)_{j\in J_1}, I_1\} \sqsubseteq \{(m_j : P'_j)_{j\in J_2}, I_2)\} = subc(i'_1 \sqsubseteq i'_2) \cup \bigcup_{j\in J_1\cup J_2} subc(P_j \sqsubseteq P'_j) \quad (8)$$

$$\text{avec } \begin{cases} I_1 = \{(m_j : p_j)_{j\in J_2\setminus J_1}, i'_1\} \\ I_2 = \{(m_j : p'_j)_{j\in J_1\setminus J_2}, i'_2\} \end{cases} \text{ et } \begin{cases} \forall j \in J_2 \setminus J_1 & P_j = p_j \\ \forall j \in J_1 \setminus J_2 & P'_j = p'_j \end{cases}$$

Enfin, les règles décrivant la simplification des contraintes de présences sont triviales :

$$\begin{array}{llll} subc(v \sqsubseteq \dagger) & = \{\} \text{ et } v = \dagger & (a) & \qquad subc(\dagger \sqsubseteq P) = \{\} \quad (b) \\ subc(\bullet T \sqsubseteq v) & = \{\bullet T \sqsubseteq v\} & (c) & \qquad subc(\bullet T \sqsubseteq \dagger) = \textbf{Err} \quad (d) \\ subc(\circ T \sqsubseteq v) & = \{\circ T \sqsubseteq v\} & (e) & \qquad subc(\circ T \sqsubseteq \dagger) = \textbf{Err} \quad (f) \end{array}$$

La gestion des variables de type étoilée est un peu particulière. En effet, l'opération d'instanciation vue dans la présentation de l'inférence provoque la recopie (avec renommage) de certaines contraintes. De plus, le processus de résolution peut également calculer la valeur exacte d'une variable étoilée x^* dès lors qu'elle possède une borne inférieure b (qui n'est pas une variable). Alors, x devient égal à b.

Le processus final est similaire à la dernière phase du système de type de ML-ACT. Les deux seules différences sont, d'une part, le fait de travailler sur un ensemble de systèmes de contraintes, et, d'autre part, le fait que les variables de type peuvent avoir des bornes (supérieures et inférieures).

En fin d'ajout de contraintes dans le graphe, le système vérifie que les acteurs n'ont pas reçu de messages orphelins triviaux. Pour cela, il minimise toutes les variables de présences contenues dans le graphe de contraintes. En effet, dans la dérivation de typage du programme, le type $@t_a$ d'un acteur a ne reçoit que des bornes supérieures, donc t_a ne possède que des bornes inférieures. Le type de l'acteur peut donc être maximisé et donc celui de t_a minimisé. Or, lors de la résolution les présences des étiquettes contenues dans t_a ne vont recevoir que des bornes inférieures : les messages effectivement envoyés et reçus.

Comme dans le cadre de ML-ACT, l'analyseur contient également un phase de minimisation des types pour les afficher que nous ne présenterons pas ici.

8.7 La preuve de correction du système de type

Dans cette section, nous allons démontrer la validité du système de type proposé. C'est-à-dire, nous allons établir sa correction en montrant qu'une réduction d'un programme typé ne peut conduire à une erreur (**Err**). Avant de réaliser cette preuve, il faut introduire un certain nombre de lemmes nécessaires.

Cette preuve suit le même schéma que celle de la correction du système de type de ML-ACT (chapitre 6).

Typage du contexte d'évaluation

Un premier lemme décrivant le comportement des contextes d'évaluation vis-à-vis du typage et du sous-typage est nécessaire.

Lemme 8.3 (Typage d'un contexte) :
Si une expression C[e] *a pour type* t_1*, le remplacement de la sous-expression* e *de type* t_2 *par une autre sous-expression* e' *de type* t_3 *sous-type de* t_2*, produit une nouvelle expression* C[e'] *de type* t_4 *sous-type de* t_1*. De plus, si* e *et* e' *introduisent les mêmes variables et ont*

les mêmes effets, alors, C[e'] *a le même environnement résultat et le même effet que* C[e]. *Soit, formellement :*

$$\begin{cases} \mathcal{E} \vdash_e \mathrm{C}[e] : t_1,\ \mathcal{E}_1,\ E_1 \\ \mathcal{E}' \vdash_e e : t_2,\ \mathcal{E}_2,\ E_2 \\ \mathcal{E}' \vdash_e e' : t_3,\ \mathcal{E}_2,\ E_2 \\ t_3 \sqsubseteq t_2 \end{cases} \Longrightarrow \mathcal{E} \vdash_e \mathrm{C}[e'] : t_4,\ \mathcal{E}_1,\ E_1 \ \wedge\ t_4 \sqsubseteq t_1$$

PREUVE : La preuve de ce lemme est une combinaison de celle des lemmes 6.3 et 6.4 de typage du contexte dans le cadre de ML-ACT. Elle se fait par induction sur la structure des contextes d'évaluation. Le seul cas d'induction relativement différent que nous allons détailler est celui de l'application d'une fonction :

Supposons que C[e] = $e\ e_1$ soit bien typée dans l'environnement $\mathcal{E}$, la racine de son arbre de typage est donc :

$$\textsc{App:}\ \frac{\mathcal{E} \vdash_e e : t_2,\ \mathcal{E}_2,\ E_2 \quad \mathcal{E} \vdash_e e_1 : t_e,\ \mathcal{E}_e,\ E_e \quad \alpha = \mathbf{inst}(t_2)}{\mathcal{E} \vdash_e e\ e_1 : t_1,\ \mathcal{E}_2 \vee \mathcal{E}_e,\ E \sqcup E_2 \sqcup E_e} \begin{pmatrix} t_e \sqsubseteq dom(\alpha) \\ \alpha \sqsubseteq t_e \xrightarrow{E} t_1 \end{pmatrix}$$

Vérifions que les contraintes de la dérivation suivante sont valides :

$$\textsc{App:}\ \frac{\mathcal{E} \vdash_e e' : t_3,\ \mathcal{E}_2,\ E_2 \quad \mathcal{E} \vdash_e e_1 : t_e,\ \mathcal{E}_e,\ E_e \quad \beta = \mathbf{inst}(t_3)}{\mathcal{E} \vdash_e e'\ e_1 : t_1,\ \mathcal{E}_2 \wedge \mathcal{E}_e,\ E \sqcup E_2 \sqcup E_e} \begin{pmatrix} t_e \sqsubseteq dom(\beta) \\ \beta \sqsubseteq t_e \xrightarrow{E} t_1 \end{pmatrix}$$

Or, par hypothèse $t_3 \sqsubseteq t_2$ qui est instanciée et donc conduit à $\beta \sqsubseteq \alpha$. En effet, une instance $\mathbf{inst}(t)$ d'un type t est une recopie de t avec renommage des variables étoilées, donc t et $\mathbf{inst}(t)$ ont la même structure. Si C est l'ensemble des contraintes élémentaires dans lequel se décompose $t_1 \sqsubseteq t_2$, alors, $\mathbf{inst}(t_1) \sqsubseteq \mathbf{inst}(t_2)$ conduit un ensemble de contraintes élémentaires similaire à C car le sous-typage des variables étoilées sera transformé en égalité (et que les autres parties ne sont pas touchées).

La décroissance des domaines montre alors la validité de la première contrainte et une simple transitivité montre celle de la seconde.

Remarque :
Les égalités de l'environnement résultat et de l'effet sont dues au fait que l'évaluation des opérations qui ajoutent des variables ou modifient les effets ne sont pas codées par les contextes. En effet, les modifications des environnements et effets ont lieu dans d'autres règles que celles de la réduction fonctionnelle. Nous nous sommes donc limité à cette condition.

Typage du filtrage

Nous allons maintenant démontrer que la substitution et plus généralement le filtrage se comportent *bien* vis-à-vis du typage et du sous-typage. Le lemme de substitution est moins puissant en μErlang qu'en ML-ACT. L'expression qui remplace la variable doit être limitée à une valeur sémantique (autre qu'une fonction ou un acteur). Ainsi, son

environnement de sortie est égal à son environnement de typage et son effet est vide. Soit :

$$v \in \mathcal{V} \wedge \mathcal{E} \vdash_e v : t,\ \mathcal{E}',\ E' \Longrightarrow \mathcal{E}' = \mathcal{E} \wedge E' = \{\}$$

En effet, une expression contenant une définition de variable peut rendre le programme non typable. Par exemple, la substitution de x par `let` $y = 2$ dans l'expression x ; `let` $y = 1$, construit un programme qui va échouer puisqu'à l'exécution, il filtre 1 par 2, alors que toutes les sous-expressions sont typables.

De plus, si une variable x de type t_x est définie lors du typage d'une expression e, alors la liaison $\{x : t_x\}$ figure encore dans l'environnement résultat du typage. En effet, la seule règle de typage qui ajoute une variable dans l'environnement est VAR2 qui n'est pas applicable ici puisque $x \in dom(\mathcal{E}+\{x : t_x\})$. Donc, il existe $\mathcal{E}'_1$ tel que $\mathcal{E}_1 \triangleq \mathcal{E}'_1 + \{x : t_x\}$.

Enfin, μErlang contient une particularité qui modifie légèrement la forme du lemme par rapport à ce qui se passe en ML-ACT. En effet, la substitution affecte les filtrages (nous avons employé le terme de filtrage dynamique). Il faut donc être plus strict sur la relation entre le type de certaines valeurs et celui des variables qu'elles remplacent. En fait, nous allons imposer une égalité entre leurs types si la variable est étoilée. Le problème rencontré est l'éventuelle non monotonie de la substitution dans le cas contraire.

Nous allons montrer deux lemmes selon que x figure dans un filtre autre que celui de sa définition (elle est « étoilée ») ou pas.

Lemme 8.4 (Typage d'une substitution de variable non-étoilée) :
Dans une expression e de type t_1, le remplacement d'une variable x de type t_x, qui ne figure dans aucun motif, par une valeur sémantique v de type t_v sous-type de t_x produit une nouvelle expression de type t'_1 sous-type de t_1. De plus, l'hypothèse sur le type de x peut être déchargée *des différents environnements. Soit, formellement :*

$$\left.\begin{array}{c} \mathcal{E}+\{x : t_x\} \vdash_e e : t_1,\ \mathcal{E}_1+\{x : t_x\},\ E_1 \\ \mathcal{E} \vdash_e v : t_v,\ \mathcal{E},\ \{\} \\ t_v \sqsubseteq t_x \end{array}\right\} \Longrightarrow \mathcal{E} \vdash_e [v/x]e : t'_1,\ \mathcal{E}_1,\ E_1 \wedge t'_1 \sqsubseteq t_1$$

PREUVE : La preuve de ce lemme est une induction sur la structure des expressions qui est proche du lemme correspondant sur ML-ACT.

Les cas initiaux sont les variables, les adresses, les constantes et les fonctions prédéfinies. La preuve du lemme dans les trois derniers cas est triviale puisqu'ils ne contiennent aucune variable. Dans le cas d'une variable y, $\mathcal{E}_1 = \mathcal{E}$ et $E_1 = \{\}$. Donc :

$$[v/x]y = \begin{cases} v & \text{si } y = x \\ y & \text{sinon} \end{cases} \Longrightarrow \begin{cases} \mathcal{E} \vdash_e v : t_v,\ \mathcal{E},\ \{\} \text{ et } t_v \sqsubseteq t_x = t_1 \\ \mathcal{E} \vdash_e y : t_1,\ \mathcal{E},\ \{\} \end{cases}$$

La propriété est triviale dans le cas d'une expression entre parenthèses. Il reste donc à démontrer le lemme lorsque e est un constructeur, une séquence, une référence à une fonction globale, une application, une abstraction ou une définition.

– **Constructeur**, par hypothèse :

$$\frac{\mathcal{E}+\{x : t_x\} \vdash_e e_i : t_i,\ \mathcal{E}_i+\{x : t_x\},\ E_i}{\mathcal{E}+\{x : t_x\} \vdash_e C(\vec{e}) : TypeC(C)(\vec{t}),\ (\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n) + \{x : t_x\},\ E_1 \sqcup \cdots \sqcup E_n}$$

Car $\mathcal{E}_1{+}\{x : t_x\}\vee\cdots\vee\mathcal{E}_n{+}\{x : t_x\} = (\mathcal{E}_1\vee\cdots\vee\mathcal{E}_n){+}\{x : t_x\}$. Par n applications de l'hypothèse d'induction, on obtient $\mathcal{E} \vdash_e [v/x]e_i : t'_i,\ \mathcal{E}_i,\ E_i$ avec $t'_i \sqsubseteq t_i$. Comme $[v/x]C(e_1, \ldots, e_n) = C([v/x]e_1, \ldots, [v/x]e_n)$:

$$\frac{\forall i\ \ \mathcal{E} \vdash_e [v/x]e_i : t'_i,\ \mathcal{E}_i,\ E_i}{\mathcal{E} \vdash_e [v/x]C(\vec{e}) : \mathit{TypeC}(C)(\vec{t'}),\ \mathcal{E}_1 \vee \cdots \vee \mathcal{E}_n,\ E_1 \sqcup \cdots \sqcup E_n}$$

Le lemme de covariance des constructeurs permet alors de montrer que $\mathit{TypeC}(C)(\vec{t'}) \sqsubseteq \mathit{TypeC}(C)(\vec{t})$.

– **Séquence**, par hypothèse :

$$\frac{\mathcal{E}{+}\{x : t_x\} \vdash_e e_1 : t_1,\ \mathcal{E}_1{+}\{x : t_x\},\ E_1 \quad \mathcal{E}_1{+}\{x : t_x\} \vdash_e e_2 : t_2,\ \mathcal{E}_2{+}\{x : t_x\},\ E_2}{\mathcal{E}{+}\{x : t_x\} \vdash_e e_1\,; e_2 : t_2,\ \mathcal{E}_2{+}\{x : t_x\},\ E_1 \sqcup E_2}$$

Par deux applications de l'hypothèse d'induction, on obtient :

$$\begin{cases} \mathcal{E} \vdash_e [v/x]e_1 : t'_1,\ \mathcal{E}_1,\ E_1 \wedge t'_1 \sqsubseteq t_1 \\ \mathcal{E}_1 \vdash_e [v/x]e_2 : t'_2,\ \mathcal{E}_2,\ E_2 \wedge t'_2 \sqsubseteq t_2 \end{cases}$$

Or, $[v/x]e_1\,; e_2 = [v/x]e_1\,;[v/x]e_2$, ce qui implique :

$$\frac{\mathcal{E} \vdash_e [v/x]e_1 : t'_1,\ \mathcal{E}_1,\ E_1 \quad \mathcal{E} \vdash_e [v/x]e_2 : t'_2,\ \mathcal{E}_2,\ E_2}{\mathcal{E} \vdash_e [v/x]e_1\,; e_2 : t'_2,\ \mathcal{E}_2,\ E_1 \sqcup E_2}$$

Et trivialement, $t'_2 \sqsubseteq t_2$.

– **Application**, par hypothèse :

$$\frac{\mathcal{E}{+}\{x : t_x\} \vdash_e e_1 : t_1,\ \mathcal{E}_1{+}\{x : t_x\},\ E_1 \quad \mathcal{E}{+}\{x : t_x\} \vdash_e e_2 : t_2,\ \mathcal{E}_2{+}\{x : t_x\},\ E_2}{\mathcal{E}{+}\{x : t_x\} \vdash_e e_1\ e_2 : t,\ (\mathcal{E}_1 \vee \mathcal{E}_2) + \{x : t_x\},\ E_1 \sqcup E_2 \sqcup E}\ (\mathcal{C})$$

Où les contraintes sont : $t_2 \sqsubseteq \mathit{dom}(\alpha)$ et $\alpha \sqsubseteq t_2 \xrightarrow{E} t$ avec $\alpha = \mathbf{inst}(t_1)$. Deux applications de l'hypothèse d'induction conduisent à : $\mathcal{E} \vdash_e [v/x]e_i : t'_i,\ \mathcal{E}_i,\ E_i$ avec $t'_i \sqsubseteq t_i$. Or, $[v/x]e_1\ e_2 = [v/x]e_1\ [v/x]e_2$, ce qui implique donc :

$$\frac{\mathcal{E} \vdash_e [v/x]e_1 : t'_1,\ \mathcal{E}_1,\ E_1 \quad \mathcal{E} \vdash_e [v/x]e_2 : t'_2,\ \mathcal{E}_2,\ E_2 \quad \beta = \mathbf{inst}(t'_1)}{\mathcal{E} \vdash_e [v/x]e_1\ e_2 : t,\ \mathcal{E}_1 \vee \mathcal{E}_2,\ E_1 \sqcup E_2 \sqcup E}\begin{pmatrix} t'_2 \sqsubseteq \mathit{dom}(\beta) \\ \beta \sqsubseteq t'_2 \xrightarrow{E} t \end{pmatrix}$$

L'instanciation remplace des variables étoilées par de nouvelles variables étoilées, donc si $t'_1 \sqsubseteq t_1$ alors $\mathbf{inst}(t'_1) \sqsubseteq \mathbf{inst}(t_1)$. En effet, les variables étoilées seront égales à leur borne inférieure et la relation entre les types de t_1 et t'_1 fait que ces bornes seront identiques.
En effet, il est immédiat que les contraintes sont vérifiées puisque $t'_2 \sqsubseteq t_2 \sqsubseteq \mathit{dom}(\alpha) \sqsubseteq \mathit{dom}(\beta)$ et $\beta \sqsubseteq \alpha \sqsubseteq t_2 \xrightarrow{E} t \sqsubseteq t'_2 \xrightarrow{E} t$.

– **Appel d'une fonction globale**, supposons que x soit différent de *ego*, alors,

par hypothèse :

$$\frac{\begin{array}{c}\mathcal{E}{+}\{x:t_x\}\vdash_e f:t_f,\ \mathcal{E}_f{+}\{x:t_x\},\ E_f\\ \mathcal{E}{+}\{x:t_x\}\vdash_e e:t_e,\ \mathcal{E}_e{+}\{x:t_x\},\ E_e\\ \alpha=\mathbf{inst}(Fun(t_f))\end{array}}{\mathcal{E}{+}\{x:t_x\}\vdash_e Fun(f)\ e:t,\ (\mathcal{E}_e\vee\mathcal{E}_f)+\{x:t_x\},\ E\sqcup E_f\sqcup E_e}\begin{pmatrix}t_f\sqsubseteq dom(\mathcal{E}_t)\\ t_e\sqsubseteq dom(\alpha)\\ \alpha\sqsubseteq t_e\xrightarrow[\mathcal{E}(ego)]{E}t\end{pmatrix}$$

Par application de l'hypothèse d'induction, on obtient $\mathcal{E}\vdash_e [v/x]f:t'_f,\ \mathcal{E}_f,\ E$ et $\mathcal{E}\vdash_e [v/x]e:t'_e,\ \mathcal{E}_e,\ E$. Comme $[v/x]Fun(f)\ e = Fun([v/x]f)\ [v/x]e$:

$$\frac{\mathcal{E}\vdash_e f:t'_f,\ \mathcal{E}_f,\ E_f\quad \mathcal{E}\vdash_e e:t'_e,\ \mathcal{E}_e,\ E_e\quad \alpha'=\mathbf{inst}(Fun(t'_f))}{\mathcal{E}\vdash_e Fun(f)\ e:t,\ (\mathcal{E}_e\vee\mathcal{E}_f),\ E\sqcup E_f\sqcup E_e}\begin{pmatrix}t'_f\sqsubseteq dom(\mathcal{E}_t)\\ t'_e\sqsubseteq dom(\alpha')\\ \alpha'\sqsubseteq t'_e\xrightarrow[\mathcal{E}(ego)]{E}t\end{pmatrix}$$

La première contrainte est triviale ($t'_f \sqsubseteq t_f \sqsubseteq dom(\mathcal{E}_t)$). Si $t'_f \sqsubseteq t_f$, cela signifie que t'_f est une union d'atomes qui figurent tous dans l'union d'atomes t_f. Donc $Fun(t'_f)$ est une portion du type $Fun(t_f)$ et donc $Fun(t'_f) \sqsubseteq Fun(t_f)$. On a donc $\alpha' \sqsubseteq \alpha$ et alors $t'_e \sqsubseteq t_e \sqsubseteq dom(\alpha) \sqsubseteq dom(\alpha')$ et $\alpha' \sqsubseteq \alpha \sqsubseteq t_e \xrightarrow[\mathcal{E}(ego)]{E} t \sqsubseteq t'_e \xrightarrow[\mathcal{E}(ego)]{E} t$. Si x vaut *ego*, le raisonnement est voisin avec le changement de la contrainte sur *ego* qui ne figure plus dans l'environnement de typage puisqu'elle est déchargée. En fait, le typage d'un terme Fun_v est légèrement différent, il suit la règle :

$$\frac{\begin{array}{cc}\mathcal{E}\vdash_e v:t_v,\ \mathcal{E},\ \{\} & \mathcal{E}\vdash_e f:t'_f,\ \mathcal{E}_f,\ E_f\\ \mathcal{E}\vdash_e e:t'_e,\ \mathcal{E}_e,\ E_e & \alpha'=\mathbf{inst}(Fun(t'_f))\end{array}}{\mathcal{E}\vdash_e Fun_v(f)\ e:t,\ (\mathcal{E}_e\vee\mathcal{E}_f),\ E\sqcup E_f\sqcup E_e}\begin{pmatrix}t'_f\sqsubseteq dom(\mathcal{E}_t)\\ t'_e\sqsubseteq dom(\alpha')\\ \alpha'\sqsubseteq t'_e\xrightarrow[t_v]{E}t\end{pmatrix}$$

Or, par hypothèse $t_v \sqsubseteq t_x = \mathcal{E}{+}\{ego:t_x\}(ego)$ donc la troisième contrainte est une conséquence immédiate de celle existant avant substitution.

– **Abstraction**, par hypothèse :

$$\frac{\begin{array}{cc}\mathcal{E}{+}\{x:t_x\}\vdash_p p_i:\alpha_i,\ \mathcal{E}_i{+}\{x:t_x\},\ V_i & \mathcal{E}_i{+}\{x:t_x\}\vdash_g g_i:r_i\\ \mathcal{E}_i{+}\{x:t_x\}\vdash_e e_i:\beta_i,\ \mathcal{E}'_i+\{x:t_x\},\ E_i & t_i=\alpha_i\setminus\bigcup\{\alpha_j\mid j<i\wedge r_j\neq?\}\end{array}}{\mathcal{E}{+}\{x:t_x\}\vdash_e \lambda[p_i,g_i.e_i]^{i\in I}:\bigsqcup\{\mathbf{star}(V_i\cup\mathcal{V}(\beta_i),\ t_i\xrightarrow[r_i]{E_i}\beta_i)\mid t_i\neq\bot\},\ \bigwedge_{i=1}^{n}\mathcal{E}'_i,\ \{\}}$$

Or, $[v/x]\lambda[p_i,g_i.e_i]^{i\in I} = \lambda[[v/x]p_i,[v/x]g_i.[v/x]e_i]^{i\in I}$ mais $[v/x]p_i = p_i$ par hypothèse puisque x ne figure dans aucun filtrage car elle n'est pas étoilée. Donc, on a : $[v/x]\lambda[p_i,g_i.e_i]^{i\in I} = \lambda[p_i,[v/x]g_i.[v/x]e_i]^{i\in I}$. Alors par hypothèse d'induction, on montre $\mathcal{E}_i\vdash_e [v/x]e_i:\beta'_i,\ \mathcal{E}'_i,\ E_i$ et $\beta'_i\sqsubseteq\beta_i$. De plus, on peut montrer facilement par induction sur la structure des gardes qu'une substitution ne change

pas r_i (si une comparaison apparait dans la garde, elle y figure encore).

$$\frac{\begin{array}{cc}\mathcal{E} \vdash_p p_i : \alpha_i,\ \mathcal{E}_i,\ V_i & \mathcal{E}_i \vdash_g [v/x]g_i : r_i \\ \mathcal{E}_i \vdash_e [v/x]e_i : \beta'_i,\ \mathcal{E}'_i,\ E_i & t_i = \alpha_i \setminus \bigcup\{\alpha_j \mid j<i \wedge r_j \neq ?\}\end{array}}{\mathcal{E} \vdash_e [v/x]\lambda[p_i, g_i.e_i]^{i\in I} : \bigsqcup\{\mathbf{star}(V_i \cup \mathcal{V}(\beta'_i),\ t_i \xrightarrow[r_i]{E_i} \beta'_i) \mid t_i \neq \bot\},\ \bigwedge_i \mathcal{E}'_i,\ \{\}}$$

Pour chaque i, $t_i \xrightarrow[r_i]{E_i} \beta'_i \sqsubseteq t_i \xrightarrow[r_i]{E_i} \beta_i$, de plus, l'étoilage ne modifie pas la forme des types, l'inégalité reste donc vraie après l'application de **star**.

– **Définition**, par hypothèse :

$$\frac{\begin{array}{c}\mathcal{E}{+}\{x : t_x\} \vdash_e e : t_e,\ \mathcal{E}_e{+}\{x : t_x\},\ E \\ \mathcal{E}_e{+}\{x : t_x\} \vdash_p p : t_p,\ \mathcal{E}_p{+}\{x : t_x\},\ V\end{array}}{\mathcal{E}{+}\{x : t_x\} \vdash_e \texttt{let}\ p = e : t_e,\ \mathcal{E}_p{+}\{x : t_x\},\ E}\ \Big(t_e \sqsubseteq t_p\Big)$$

Or, p ne contient pas x et une application de l'hypothèse d'induction conduit à $\mathcal{E} \vdash_e [v/x]e : t'_e,\ \mathcal{E}_e,\ E$ avec $t'_e \sqsubseteq t_e$. Comme $[v/x](\texttt{let}\ p = e) = (\texttt{let}\ [v/x]p = [v/x]e) = (\texttt{let}\ p = [v/x]e)$, on a :

$$\frac{\begin{array}{c}\mathcal{E} \vdash_e [v/x]e : t'_e,\ \mathcal{E}_e,\ E \\ \mathcal{E}_e \vdash_p p : t_p,\ \mathcal{E}_p,\ V\end{array}}{\mathcal{E} \vdash_e [v/x]\texttt{let}\ p = e : t'_e,\ \mathcal{E}_p,\ E}\ \Big(t'_e \sqsubseteq t_p\Big)$$

La contrainte étant trivialement vraie puisque $t'_e \sqsubseteq t_e \sqsubseteq t_p$.

Le cas des variables étoilées est légèrement différents puisqu'il faut imposer l'égalité entre valeurs et filtres pour obtenir un comportement correct

Lemme 8.5 (Typage d'une substitution de variable étoilée) :
Dans une expression e (resp. un motif p), le remplacement d'une variable étoilée x^* par une valeur sémantique v du même type produit une nouvelle expression (resp. un nouveau motif) du même type que e (resp. que p). De plus, l'hypothèse sur le type de x peut être déchargée *des différents environnements. Soit, formellement :*

$$\left.\begin{array}{c}\mathcal{E}{+}\{x : t_x\} \vdash_e e : t_1,\ \mathcal{E}_1{+}\{x : t_x\},\ E_1 \\ \mathcal{E} \vdash_e v : t',\ \mathcal{E},\ \{\} \\ t' = t_x\end{array}\right\} \Longrightarrow \mathcal{E} \vdash_e [v/x]e : t_1,\ \mathcal{E}_1,\ E_1$$

et

$$\left.\begin{array}{c}\mathcal{E}{+}\{x : t_x\} \vdash_p p : t_1,\ \mathcal{E}_1{+}\{x : t_x\},\ V \\ \mathcal{E} \vdash_e v : t',\ \mathcal{E},\ \{\} \\ t' = t_x\end{array}\right\} \Longrightarrow \mathcal{E} \vdash_p [v/x]p : t_1,\ \mathcal{E}_1,\ V$$

Preuve : La preuve de ce lemme est une induction sur la structure des expressions (resp. des motifs) qui est proche du lemme correspondant sur ML-Act. La preuve du lemme sur les motifs est une simplification de celle sur les expressions. En effet, les motifs ont une structure plus simple et n'ajoutent que le motif joker pour lequel il n'y a rien à prouver puisqu'il ne contient aucune variable et donc que la substitution n'agit

pas dessus. Nous ne donnons donc pas la preuve du lemme dans le cas des motifs.

De plus, la preuve est assez proche de celle du lemme précédent, nous n'aborderons que les cas qui diffèrent, à savoir lorsque e est une abstraction ou une définition.

– **Abstraction**, par hypothèse :

$$\frac{\begin{array}{cc}\mathcal{E}{+}\{x:t_x\}\vdash_p p_i:\alpha_i,\ \mathcal{E}_i{+}\{x:t_x\},\ V_i & \mathcal{E}_i{+}\{x:t_x\}\vdash_g g_i:r_i \\ \mathcal{E}_i{+}\{x:t_x\}\vdash_e e_i:\beta_i,\ \mathcal{E}'_i+\{x:t_x\},\ E_i & t_i=\alpha_i\setminus\bigcup\{\alpha_j\mid j<i\wedge r_j\neq ?\}\end{array}}{\mathcal{E}{+}\{x:t_x\}\vdash_e \lambda[p_i,g_i.e_i]^{i\in I}:\bigsqcup\{\mathbf{star}(V_i\cup\mathcal{V}(\beta_i),\ t_i\xrightarrow[r_i]{E_i}\beta_i)\mid t_i\neq\bot\},\ \bigwedge_{i=1}^n\mathcal{E}'_i,\ \{\}}$$

Or, $[v/x]\lambda[p_i,g_i.e_i]^{i\in I}=\lambda[[v/x]p_i,[v/x]g_i.[v/x]e_i]^{i\in I}$ alors par hypothèse d'induction, on montre $\mathcal{E}_i\vdash_e[v/x]e_i:\beta_i,\ \mathcal{E}'_i,\ E_i$ et $\mathcal{E}\vdash_p[v/x]p_i:\alpha_i,\ \mathcal{E}_i,\ V_i$ (car x ne peut être étoilée dans p_i). De plus, on peut montrer facilement par induction sur la structure des gardes qu'une substitution ne change pas r_i (si une comparaison figurait dans la garde, elle est encore là).

$$\frac{\begin{array}{cc}\mathcal{E}\vdash_p[v/x]p_i:\alpha_i,\ \mathcal{E}_i,\ V_i & \mathcal{E}_i\vdash_g[v/x]g_i:r_i \\ \mathcal{E}_i\vdash_e[v/x]e_i:\beta_i,\ \mathcal{E}'_i,\ E_i & t_i=\alpha_i\setminus\bigcup\{\alpha_j\mid j<i\wedge r_j\neq ?\}\end{array}}{\mathcal{E}\vdash_e\lambda[[v/x]p_i,[v/x]g_i.[v/x]e_i]^{i\in I}:\bigsqcup\{\mathbf{star}(V_i\cup\mathcal{V}(\beta_i),\ t_i\xrightarrow[r_i]{E_i}\beta_i)\mid t_i\neq\bot\},\ \bigwedge_i\mathcal{E}'_i,\ \{\}}$$

– **Définition**, par hypothèse :

$$\frac{\begin{array}{c}\mathcal{E}{+}\{x:t_x\}\vdash_e e:t_e,\ \mathcal{E}_e{+}\{x:t_x\},\ E \\ \mathcal{E}_e{+}\{x:t_x\}\vdash_p p:t_p,\ \mathcal{E}_p{+}\{x:t_x\},\ V\end{array}}{\mathcal{E}{+}\{x:t_x\}\vdash_e \texttt{let}\ p=e:t_e,\ \mathcal{E}_p{+}\{x:t_x\},\ E}\ \Big(t_e\sqsubseteq\mathbf{star}(V,\ t_p)\Big)$$

On applique deux fois l'hypothèse d'induction. La règle précédente est alors inchangée.

De cette propriété, nous déduisons une propriété plus générale sur le résultat d'un filtrage qui permet de substituer simultanément toutes les variables issues d'un filtrage.

Lemme 8.6 (Typage du filtrage) :
L'application d'une substitution issue d'un filtrage réussi à une expression typable produit une nouvelle expression du même type que la précédente. Soit, formellement :

$$\left.\begin{array}{l}\mathcal{E}\vdash_p p:t_p,\ \mathcal{E}_p,\ V\\ \mathcal{E}\vdash_e v:t_v,\ \mathcal{E},\ \{\}\\ t_v\sqsubseteq t_p\wedge v/p\Rightarrow\sigma\\ \mathcal{E}{+}\mathcal{E}_p\vdash_e e:t_1,\ \mathcal{E}_1{+}\mathcal{E}_p,\ E\end{array}\right\}\Longrightarrow\mathcal{E}\vdash_e\sigma(e):t'_1,\ \mathcal{E}_1,\ E\ \ \wedge\ \ t'_1\sqsubseteq t_1$$

PREUVE : Nous allons prouver ce lemme, de la même manière que celui de ML-ACT: par induction sur la structure du filtre. Les cas initiaux de l'induction sont le joker, une variable ou une constante.

– Dans le cas d'un joker, la substitution et l'environnement sont vides et donc le lemme est trivial.
– Si p est une variable x, un des deux lemmes précédents s'applique selon que x soit étoilée ou non et fournit immédiatement le résultat.

– Si p est une valeur constante, l'hypothèse faite sur le type de la valeur filtrée v implique la réussite du filtrage et $v = p$. La substitution et l'environnement sont alors vides et donc le lemme est trivial.

Si le filtre est construit par le constructeur C d'arité n alors par hypothèse v vaut $C(v_1, \ldots, v_n)$ et donc on a, pour tout i $v_i/p_i \Rightarrow \sigma_i$ et $\sigma = \sigma_1 \circ \cdots \circ \sigma_n$. L'application du lemme de sous-typage des constructeurs de types ainsi que n applications de l'hypothèse d'induction conduisent alors au résultat sur le même raisonnement que dans le cas de ML-ACT.

Typage des expressions

Nous pouvons maintenant énoncer et prouver la continuité du typage fonctionnel.

Théorème 8.1 (Continuité du typage fonctionnel) :
Si une expression e de type t se réduit en une expression e′ alors e′ est bien typée et son type est inférieur à t. De plus, Soit, formellement :

$$\mathcal{E} \vdash_e e : t,\ \mathcal{E}',\ E' \wedge e \longrightarrow_e e' \Longrightarrow \mathcal{E} \vdash_e e' : t',\ \mathcal{E}',\ E' \wedge t' \sqsubseteq t$$

PREUVE : La preuve de ce théorème consiste en une induction sur la structure des expressions qui peuvent se réduire. Le cas initial unique est celui des variables car ce sont les expressions ne contenant pas de sous-expressions qui peuvent se réduire. Or, une variable bien typée figure dans l'environnement, elle a été définie dans un filtre et a été substituée. Ainsi, la règle VARE ne peut pas s'appliquer.

Puis :

– Si e est de la forme $C[x]$, la réduction VARE ne peut pas s'appliquer grâce au même raisonnement que sur une variable seule.
– Si e est une application :
  – Comme l'application est bien typée, la seconde contrainte ($t_f \sqsubseteq t_e \to t$) impose un type fonctionnel à la première valeur et la règle APPE1 ne peut donc pas s'appliquer.
  – Si l'abstraction appliquée est vide (le filtrage à échoué), cela signifie qu'une réduction précédente était de la forme $C[\lambda[p, g.e]\ v] \longrightarrow_e C[\lambda[]\ v]$ et le terme avant réduction était bien typé. Donc, $t_v \sqsubseteq dom(t_f)$ où t_f est le type de l'abstraction. Deux cas se présentent alors :
    – la garde contient une comparaison et donc $dom(t_f) = \bot$; la contrainte d'inclusion ne pouvait être vérifiée ce qui contredit l'hypothèse faite sur la typabilité du terme ;
    – la garde ne contient pas de comparaison et donc $dom(t_f) = t_p$; la contrainte d'inclusion imposait $t_v \sqsubseteq t_p$ et le filtrage ne pouvait échouer et la réduction ne pouvait donc pas produire $\lambda[]\ v$.

  La règle APPE2 ne peut donc pas apparaître.
  – Si e vaut $C[\lambda[p_i, g_i.e_i]^{i \in I}\ v]$, deux cas se présentent :
    – le premier filtrage réussit ($v/p_1, g_1 \Rightarrow \sigma$), alors l'abstraction réduite à ce

premier cas réussit également et on a :

$$t_v \sqsubseteq \mathbf{inst}(t_{p_1})_n \text{ et } \mathbf{inst}(t_{p_1} \xrightarrow{E_1} t_{e_1})_n \sqsubseteq t_v \xrightarrow{E} t$$

Le lemme de filtrage permet alors de typer $\sigma(\mathrm{C}[e_1])$ par $\mathcal{E} \vdash_e \sigma(\mathrm{C}[e_1]) : t',\ \mathcal{E}',\ E'$ avec $t' \sqsubseteq \mathbf{inst}(t_{e_1})_n \sqsubseteq t$. La réduction APP est donc correcte.

- le premier filtrage échoue ($v/p_1, g_1 \Rightarrow \mathit{failed}$), deux causes peuvent provoquer cet échec :
  - le motif, cela signifie que leurs types ne sont pas compatibles et donc $t_v \sqsubseteq t_{p_1}$, or le domaine de l'abstraction avant et après suppression du premier filtre ne diffère que de t_{p_1} et donc t_v est encore inclus dans le domaine de l'abstraction restreinte ;
  - la garde, elle contient donc une comparaison ou bien un test de type qui échoue ; si c'est une comparaison, le type de la garde est ? et il ne figure pas dans le domaine ; si c'est un test de type qui est faux, on retrouve la non-compatibilité des types t_{p_1} et t_v.

  Dans tous les cas, t_v est encore inclus dans le domaine de l'abstraction restreinte. L'application de v à l'abstraction restreinte est donc correcte.

- Si e est une séquence $\mathrm{C}[v\,; e]$, le typage de la séquence impose celui de e avec le même type et donc le résultat est immédiat.
- Si e est une définition de la forme $\mathrm{C}[\texttt{let}\ p = v]$:
  - la définition est bien typée, donc $t_v \sqsubseteq t_p$; ce qui implique que v et p ont la même structure et le filtrage ne peut pas échouer. La réduction LETE ne peut donc pas se produire.
  - Il est donc possible d'appliquer le lemme de filtrage qui montre alors $\mathcal{E} \vdash_e \sigma(\mathrm{C}[e]) : t',\ \mathcal{E}',\ E'$ avec $t' \sqsubseteq t_v$.
- Si e est une référence à une fonction globale, la contrainte du typage de *Fun* impose que le type de son argument soit dans le domaine de $\mathcal{E}_t$, les règles FUNE1 et FUNE2 ne peuvent donc pas s'appliquer. Dans le cas d'une réduction FUN, la définition de la fonction *Fun* permet de dire que le type de la fonction dans l'environnement de typage est sous-type de $\mathit{Fun}(t_f)$, ce qui montre la décroissance du type. Le raisonnement est similaire dans le cas de FUNEGO avec la contrainte supplémentaire sur l'*ego* qui permet de montrer par le lemme de substitution que $\mathrm{C}[[v/\mathit{ego}]\mathcal{E}(f)]$ est bien typé est que son type est sous-type de $\mathit{Fun}(t_f)$.

Les règles de typage des configurations

Avant de passer à la preuve de continuité du typage, il faut présenter les règles 8.8 qui assurent qu'une configuration ne contient pas d'erreur. Elles sont proches de celles de ML-ACT en oubliant la règle qui concerne l'interface (ACTR) puisque cette notion n'existe pas en μErlang. Si $\mathcal{E}$ est l'environnement de typage de la configuration w, elles ont la forme $\mathcal{E} \vdash_w w$. Le typage parcourt la configuration et vérifie que toutes les expressions et toutes les valeurs qu'elle contient sont bien typées. On vérifie également que les acteurs reçoivent bien des messages compatibles avec leur potentiel. Remarquons qu'un acteur

$$\textsc{Empty}: \frac{}{\mathcal{E} \vdash_w \epsilon} \qquad \textsc{Par}: \frac{\mathcal{E} \vdash_w w_1 \quad \mathcal{E} \vdash_w w_2}{\mathcal{E} \vdash_w w_1 \parallel w_2} \qquad \textsc{Nu}: \frac{\mathcal{E}+\{a : t_a\} \vdash_w w}{\mathcal{E} \vdash_w \nu a.w}$$

$$\textsc{Mess}: \frac{\mathcal{E} \vdash_e m : t,\ \mathcal{E},\ \{\}}{\mathcal{E} \vdash_w a \triangleleft m} \left(\mathcal{E}(a) \sqsubseteq @Mess(t)\right) \qquad \textsc{Anon}: \frac{\mathcal{E} \vdash_e e : t,\ \_,\ \_}{\mathcal{E} \vdash_w \star \triangleright e}$$

$$\textsc{Act}: \frac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e e : t,\ \_,\ E \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e} \begin{pmatrix} \mathcal{E}(a) \sqsubseteq @t_q \\ \mathcal{E}(a) \sqsubseteq @E \end{pmatrix}$$

$$\frac{}{\mathcal{E} \vdash_q \varnothing : \{\}} \qquad \frac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e m : t,\ \mathcal{E},\ \{\}}{\mathcal{E} \vdash_q m::q : Mess(t) \sqcup t_q}$$

Règles 8.8 Le typage des configurations dans le cadre de μErlang.

doit avoir un type compatible avec son potentiel initial pour être bien typé.

Typage des configurations

Le théorème final de continuité du typage nécessite encore un lemme sur les configurations congrues.

Lemme 8.7 (Typage des configurations congrues) :
Si deux configurations w et w' sont congrues alors w est bien typée si et seulement si w' l'est. Soit, formellement :

$$w \equiv w' \Longrightarrow (\mathcal{E} \vdash_w w \Longleftrightarrow \mathcal{E} \vdash_w w')$$

Preuve : Nous ne démontrerons pas ce lemme puisque sa preuve est identique à celle réalisée dans le cadre de ML-Act.

Il est maintenant possible de démontrer la continuité du typage sur les configurations.

Théorème 8.2 (Continuité du typage) :
Si une configuration w bien typée se réduit en une configuration w' alors w' est bien typée. Soit, formellement :

$$\mathcal{E} \vdash_w w \wedge w \longrightarrow w' \Longrightarrow \mathcal{E} \vdash_w w'$$

Preuve : La preuve de ce théorème consiste en une induction sur la structure des configurations. De nombreux cas sont identiques à la preuve du chapitre 6 et ne seront pas détaillés.

Les cas initiaux de cette preuve sont la configuration vide, le message en transit et l'acteur (anonyme ou non). La configuration vide et le message en transit ne se réduisant pas, il n'y a donc rien à prouver.

Soit un acteur d'identité α de corps e, de nombreuses réduction sont possibles selon les formes de α et de e :

– Cong : *idem* ML-Act.

– EXP : *idem* ML-ACT.
– SEND : $e \triangleq send(m,a)$ et α peut avoir deux formes : $\star$ ou $\langle a \,|\, q\rangle$. Alors :

$$\frac{\mathcal{E} \vdash_e \mathrm{C}[e] : t,\, \mathcal{E}',\, E}{\mathcal{E} \vdash_w \star \triangleright \mathrm{C}[e]} \text{ ou } \frac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e \mathrm{C}[e] : t,\, \mathcal{E}',\, E \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q\rangle_{\mathcal{P}} \triangleright \mathrm{C}[e]} \begin{pmatrix} \mathcal{E}(a) \sqsubseteq @t_q \\ \mathcal{E}(a) \sqsubseteq @E \end{pmatrix}$$

et $\alpha \triangleright \mathrm{C}[send(m,a)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel a \triangleleft m$, il nous faut donc montrer que le corps de l'acteur est encore typable et que le message est typable. Or, si on type l'application de *send* :

$$\frac{\mathcal{E} \vdash_e m : t_m,\, \mathcal{E},\, \{\} \quad \dfrac{(\mathcal{E}_0{+}\mathcal{E})(a) = t_a}{\mathcal{E} \vdash_e a : t_a,\, \mathcal{E},\, \{\}}}{\mathcal{E} \vdash_e send(m,a) : unit,\, \mathcal{E},\, \{\}} \begin{pmatrix} t_m \sqsubseteq \alpha \\ t_a \sqsubseteq @Mess(\alpha) \end{pmatrix}$$

Donc, $\mathcal{E}(a) \sqsubseteq @Mess(t_m)$ et le message est typable. Enfin, l'application du lemme de typage d'un contexte avec *nop* qui est aussi de type *unit* montre immédiatement que le corps de l'acteur est typable. La réduction SENDE ne peut avoir lieu puisque comme il est bien typé son premier argument est bien un message et son deuxième un acteur.
– NEW : *idem* ML-ACT.
– INIT : $e \triangleq init(a,v)$ et $\alpha = \star$ ou $\alpha = \langle a \mid q\rangle$, les deux règles de typage des acteurs rappelées dans le cas SEND sont vérifiées. Comme $\alpha \triangleright \mathrm{C}[init(a,v)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel \langle a \mid \varnothing\rangle_{\mathcal{P}(v\ a)} \triangleright v\ a$, il nous faut donc montrer que le corps de l'acteur est encore typable et que le nouvel acteur est typable. Dans le cadre des programmes traduits, le deuxième argument du *init* vaut toujours $\lambda ego.v$. Or, si on type l'application de *init* :

$$\frac{\dfrac{(\mathcal{E}_0{+}\mathcal{E})(a) = t_a}{\mathcal{E} \vdash_e a : t_a,\, \mathcal{E},\, \{\}} \quad \dfrac{\mathcal{E} \vdash_p ego : t_{ego},\, \mathcal{E}{+}\{ego{:}t_{ego}\},\, \{\} \quad \mathcal{E}{+}\{ego{:}t_{ego}\} \vdash_e v : t_v,\, \mathcal{E}_v,\, E}{\mathcal{E} \vdash_e \lambda ego.v : t_{ego} \xrightarrow{E} t_v,\, \mathcal{E},\, \{\}}}{\mathcal{E} \vdash_e init(a, \lambda ego.v) : unit} (1)$$

Avec $(1) = \{t_a \sqsubseteq @\psi;\ t_{ego} \xrightarrow{E} t_v \sqsubseteq @\psi \xrightarrow{\psi} \top\}$. La dérivation suivante :

$$\frac{\mathcal{E} \vdash_q \varnothing : @\{\} \quad \dfrac{\mathcal{E} \vdash_e a : t_a,\, \mathcal{E},\, \{\} \quad \mathcal{E} \vdash_e \lambda ego.v : t_{ego} \xrightarrow{E} t_v,\, \mathcal{E},\, \{\}}{\mathcal{E} \vdash_e v\ a : t_v,\, \mathcal{E},\, E}\,(2)}{\mathcal{E} \vdash_w \langle a \,|\, \varnothing\rangle_{\mathcal{P}(v\ a)} \triangleright v\ a}\,(3)$$

avec $(2) = \{t_{ego} \xrightarrow{E} t_v \sqsubseteq t_a \xrightarrow{E} t_v\}$ et $(3) = \{t_a \sqsubseteq @\{\};\ t_a \sqsubseteq @E\}$. Or, (1) se réduit en $\{t_a \sqsubseteq @\psi \sqsubseteq t_{ego};\ E \sqsubseteq \psi\}$ et $(2+3)$ en $\{t_a \sqsubseteq t_{ego};\ t_a \sqsubseteq @E\}$. Donc les contraintes $(2+3)$ sont vérifiées puisque les contraintes (1) le sont.
Reste à montrer que l'abstraction de $@\psi_a$ est bien inférieure au potentiel. La preuve est voisine de celle sur ML-ACT tout en étant plus simple. La simplification vient du fait qu'une interface est immédiatement paramètre d'un *receive* et qu'elle ne peut donc pas être transmise à travers des variables intermédiaires. Les seules contraintes qui peuvent ajouter un message dans l'abstraction de ψ_a

sont $\alpha \sqsubseteq \psi_a$ et $\beta \sqsubseteq \psi_a$ (où α est le type de l'*ego* et β le type du *self*).
Les autres contraintes qui pourront porter sur a ne peuvent pas contenir de messages installés. Or, si m est une étiquette figurant dans l'abstraction du type de a, par définition de l'abstraction, il est immédiat qu'il existe un type t tel que $\{m : \bullet t\} \sqsubseteq \phi_a$. Cette contrainte provient donc du comportement initial ou de l'*ego*. Or, l'*ego* ne contient que les messages qui lui sont envoyés et les interfaces obtenues comme effet de l'interface initiale v. Ainsi, la contrainte provient nécessairement d'un *receive* figurant dans v.
De plus, la seule règle d'inférence qui ajoute une contrainte contenant des messages installés est l'appel de la fonction *receive* qui exploite **effect** et donc utilise *Inter*. Celle-ci suppose donc que m figure dans une interface I de la forme $\lambda[m(p).e,\ \ldots]$ où p est de type t. Donc, dans v, I est le paramètre d'un *receive*, et la règle de calcul du potentiel du *receive* (RCV) montre que m figure dans le potentiel effet. Or, les effets sont accumulés sauf lors de l'abstraction. Mais, si le *receive* est contenu dans une abstraction non appliquée, son type effet n'est pas cumulé et l'inclusion $\{m : \bullet t\} \sqsubseteq \phi_v$ n'est pas possible. De plus, l'application d'une abstraction relâche son effet et donc celui-ci figure dans le potentiel.
Ainsi, dans tous les cas, m figure dans le potentiel de a et donc que $Abs(\mathcal{E}(a)) \subseteq \mathcal{P}(v\ a)$.
La traduction impose à la valeur passée en deuxième argument d'être une fonction (dans le cadre d'ERLANG, l'ensemble des interfaces $\mathbb{I}$ est égale à l'ensemble des valeurs sémantiques) et le typage impose au premier d'être une adresse. La règle de réduction INITE ne peut donc pas s'appliquer.

– REC : Les règles de typage introduites dans le SEND sont valides avec $\alpha = \langle a \,|\, q \rangle$ et $e \triangleq \mathit{receive}(v)$. Supposons qu'un message m de la boîte aux lettres puisse être traité. Alors, la réduction qui a lieu est $\langle a \mid q \rangle_{\mathcal{P}} \triangleright \mathrm{C}[\mathit{receive}(v)] \longrightarrow \langle a \mid q' \rangle_{\mathcal{P}} \triangleright \sigma(\mathrm{C}[e])$ avec $v \prec q \Longrightarrow q',\ e,\ \sigma$. Il nous faut donc montrer que l'acteur après réduction est typable.

$$\frac{\mathcal{E} \vdash_q q' : t'_q \quad \mathcal{E} \vdash_e \sigma(\mathrm{C}[e]) : t,\, \mathcal{E}',\, E \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q' \rangle_{\mathcal{P}} \triangleright \sigma(\mathrm{C}[e])} \begin{pmatrix} \mathcal{E}(a) \sqsubseteq @t'_q \\ \mathcal{E}(a) \sqsubseteq @E \end{pmatrix}$$

La queue décroît (on lui retire un message), il est aisé alors de montrer que son type décroît également (c'est l'union des types des messages). La première contrainte $\mathcal{E}(a) \sqsubseteq @t_q \sqsubseteq @t'_q$ est donc trivialement vérifiée. Comme, on ne touche pas au potentiel la prémisse sur l'abstraction du potentiel est vraie. Reste à monter que la seconde contrainte $\mathcal{E}(a) \sqsubseteq @E$ est vérifiée. Or, la définition de $\prec$ montre que l'on peut appliquer le lemme de filtrage et donc montrer que $\sigma(\mathrm{C}[e])$ est bien typé. De plus, la valeur passée à un *receive* en μErlang est une interface qui a mémorisé l'effet de l'un quelconque de ses corps dans le type fonctionnel de cette branche et donc E figure dans l'effet E' du *receive*. Or, avant réduction $\mathcal{E}(a) \sqsubseteq @E'$ ce qui permet de conclure.

Il ne reste plus qu'à analyser les deux dernières formes possibles pour une configuration : une restriction ou une mise en parallèle.

– RES : *idem* ML-ACT.

– Si la configuration est une mise en parallèle, deux réductions sont possibles : la réduction d'une sous-configuration (PAR) ou bien la réception d'un message (RCV).
  – PAR : *idem* ML-ACT.
  – RCV : La réception d'un message par son destinataire n'est possible que si la configuration a la forme suivante : $\langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m$ et elle se réduit alors en $\langle a \,|\, q :: m \rangle_{\mathcal{P}} \triangleright e$ si $m \in \mathcal{P}$. La première configuration est typée (par hypothèse) par la déduction suivante :

$$\frac{\dfrac{\mathcal{E} \vdash_q q : t_q \quad \mathcal{E} \vdash_e e : t_e,\ \mathcal{E}_e,\ E \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e}\,(1) \quad \dfrac{\mathcal{E} \vdash_e m : t_m,\ \mathcal{E},\ \{\}}{\mathcal{E} \vdash_w a \triangleleft m}\,(2)}{\mathcal{E} \vdash_w \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m}$$

avec $(1) = \{\mathcal{E}(a) \sqsubseteq @t_q;\ \mathcal{E}(a) \sqsubseteq @E\}$ et $(2) = \{\mathcal{E}(a) \sqsubseteq @Mess(t_m)\}$.

$$\frac{\mathcal{E} \vdash_q q :: m : t'_q \quad \mathcal{E} \vdash_e e : t_e,\ \mathcal{E}_e,\ E \quad Abs(\mathcal{E}(a)) \subseteq \mathcal{P}}{\mathcal{E} \vdash_w \langle a \,|\, q :: m \rangle_{\mathcal{P}} \triangleright e} \begin{pmatrix} \mathcal{E}(a) \sqsubseteq @t'_q \\ \mathcal{E}(a) \sqsubseteq @E \end{pmatrix}$$

Il suffit donc de montrer que la queue augmentée est bien typée et vérifie la première contrainte.
Or, il est facile de montrer par induction sur la forme de la queue que $q :: m$ est bien typée et que son type t'_q vaut $t_q \sqcup Mess(t_m)$. Alors, $@t'_q = @(t_q \sqcup Mess(t_m)) = @t_q \sqcap @Mess(t_m)$ et $\mathcal{E}(a)$ est dans les deux membres de cette intersection. Ce qui permet de conclure que $\mathcal{E}(a) \sqsubseteq @t'_q$ est vraie et donc que la configuration après réduction est bien typée.
Enfin, comme $\mathcal{E}(a) \sqsubseteq @Mess(t_m)$ et qu'en fin de la résolution des contraintes inférées sur le programme, on vérifie que tous les types d'acteurs créés dans le programme ne contiennent pas de messages reçus non-installés, un message correspondant à m est installé dans le programme. Or, cette installation est prise en compte :
  – soit initialement (donc avant toute utilisation de a) et alors l'étiquette correspondant à m figure dans le potentiel ;
  – soit lors de la réception d'une interface que l'acteur assume et alors son potentiel est ouvert et donc contient également m.

  La règle RCVE ne peut donc pas s'appliquer.

Ceci conclut la démonstration du théorème.

Nous pouvons donc énoncer le théorème de correction de notre système de type qui est une conséquence directe de la continuité du typage des configurations.

Théorème 8.3 (Correction du typage) :
Une configuration w bien typée ne peut se réduire en une erreur. Soit, formellement :

$$\mathcal{E} \vdash_w w \Longrightarrow w \not\longrightarrow \textbf{\textit{Err}}$$

PREUVE : L'erreur n'étant pas typable, si on suppose que w se réduit en **Err**, une application du théorème de continuité amène directement à une contradiction.

8.8 Conclusion

Ce chapitre débute par une analyse des particularités du système de type destiné à μErlang par rapport à celui de ML-Act. Ensuite, nous avons décrit précisément la forme des erreurs que nous souhaitons détecter dans les programmes. Puis, nous avons défini les types et le système de type construit dans ce but. Le nouveau processus de résolution des contraintes inférées a été exposé dans ses grandes lignes. Enfin, nous avons détaillé la preuve de la correction de ce système de type.

Dans le cadre de mes travaux de thèse, j'ai réalisé un prototype d'analyseur de programme Erlang. Celui-ci est, en grande partie, basé sur le système présenté dans ce chapitre. Il en est un peu différent dans le sens où il ne contient pas de traduction dans un langage plus simple. Il manipule directement l'arbre du programme Erlang. Cet outil bien que rudimentaire permet de réaliser des analyses intéressantes. Dans sa phase de conception et d'implantation, aucun outil générique de résolution de contraintes réellement utilisable n'existait. Il a donc fallu mettre au point un algorithme de résolution ainsi qu'un processus de simplification pour l'affichage des types obtenus. Ces travaux basés sur les résultats de F. Pottier ont demandé des adaptations du fait de la forme de nos indicateurs de présence. Le constructeur de messages installés $\bullet$ est, en effet, assez complexe puisque comparé au constructeur de messages reçus $\circ$, il est covariant, et comparé à un autre message installé, il est contravariant. Cette possibilité semble poser des problèmes lors de l'utilisation de l'outil de résolution générique Wallace conçu par F. Pottier. Il semble pourtant qu'il serait intéressant d'étudier cette éventuelle collaboration.

Notre travail étant centré sur la forme des types et sur la détection des erreurs, nous ne nous sommes intéressés que sommairement à l'aspect affichage et utilisation des résultats de l'analyse. Ce problème paraît cependant relativement complexe et mériterait une étude approfondie.

Comme le système de type de ML-Act, celui d'Erlang a été conçu de façon a permettre simplement une augmentation de la précision de l'approximation des interfaces. Ainsi, il s'agit dans un avenir proche de lui intégrer l'analyse à base de multiplicités présentée dans la conclusion du chapitre 5 page 149. Un terme de présence deviendra, alors, un couple qui mémorisera le nombre de messages envoyés et installés. Une comparaison plus précise des envois et réceptions sera ainsi possible.

Le système de type nécessite également quelques extensions pour devenir un outil d'analyse précis de programmes Erlang complets.

Premièrement, les travaux de M. Shields et al. dans [SM01] ou [Shi01] qui construit un système de type pour un langage fonctionnel XMλ destiné à manipuler des documents XML. Pour typer correctement les choix de XML et les tuples non ordonnés de SGML, ils doivent construire un λ-calcul qui manipule des enregistrements et des variants sans étiquette. Par exemple, ils veulent typer le choix + par une rangée des types des différents éléments de la somme. Ainsi, $(1) + (\texttt{"test"}) + (\lambda x.\texttt{if } x \texttt{ then } 1 \texttt{ else } 0)$ est typé par $\{int;\ string;\ bool \to int\}$.

Les messages d'Erlang ne contenant pas d'étiquettes de messages, les types donnés aux processus et les effets de fonctions devraient s'inspirer de cette technique pour lever l'hypothèse imposée par le système actuel. Ainsi, les fonctions *Mess* et *Inter* ne seraient plus d'aucune utilité. La sémantique d'Erlang devrait alors être modifiée pour remplacer

le potentiel d'un acteur par son type.

Cette adaptation ne semble pas évidente car le système de type de XMλ repose sur des contraintes d'égalité entre types et sur une notion d'implication de ces contraintes. Ainsi, son intégration avec le sous-typage nécessaire dans le cadre d'ERLANG nécessite l'étude de l'implication de contraintes de sous-typage. Or, à notre connaissance, aucun des travaux menés dans ce domaine n'a véritablement abouti.

Deuxièmement, dans le cadre des applications de télécommunication et donc dans ERLANG, la notion d'exception est primordiale pour obtenir une certaine qualité de programme. La fiabilité nécessaire pour de telles applications passe par un traitement précis de toutes les exceptions pouvant éventuellement surgir. Le système de type pourrait alors devenir une aide précieuse pour le programmeur en calculant une approximation des exceptions qui peuvent être levées par une fonction et qui ne sont pas traitées. L'extension du système de type avec une analyse inspirée des travaux de F. PESSAUX sur la détection des exceptions non rattrapées en OBJECTIVE CAML (voir [Pes99] serait donc importante pour obtenir un outil d'analyse statique permettant de faciliter la programmation en ERLANG.

Enfin, un système de type pour ERLANG devra réaliser une approximation compatible avec le chargement de code dynamique durant l'exécution (« hot code swapping » en anglais). En effet, dans le cadre des développements d'applications de télécommunication réalisées en ERLANG, cet aspect est primordial. Un module est utilisé par des centaines de milliers de nœuds qui ne peuvent être stoppés et redémarrés. Une évolution de ce module passe donc par un changement au cours de l'exécution du code et surtout le nouveau code du module doit pouvoir cohabiter pendant une période transitoire avec l'ancienne version s'exécutant sur certains nœuds. Un premier pas vers de tels systèmes pourrait être l'adaptation du système de type conçu par P. SEWELL et présenté dans [Sew01].

Conclusions et perspectives

Nous avons présenté en détail dans cette thèse deux systèmes de type utilisant un mécanisme d'inférence reposant sur le sous-typage pour détecter les erreurs au sein de programmes concurrents et répartis.

Le but initial de nos travaux était de valider l'application, dans le cadre de véritables langages de programmation, de techniques de typage, mises au point pour un calcul de processus modélisant la notion d'acteurs. Cette étude comportait deux aspects : d'une part, l'analyse des adaptations nécessaires lors de l'extension à un langage complexe des systèmes et des techniques de résolution précédemment developpés au sein de l'équipe *vestale* ; d'autre part, la réalisation d'une analyse critique de la qualité et de l'utilité des informations apportées par nos analyses.

Mes travaux ont débuté par l'extension des systèmes de type construits sur Cap. La première cible a été ML-Act, l'ajout au cœur fonctionnel d'Objective CAML de primitives permettant la programmation par acteurs (`send to`, `become`, `new`, `ego`, `self` et `behavior message end`) conçue durant mon DEA.

La première approche proposée, dans le cadre de mon DEA, était un système de type inspiré des travaux de F. Nielson et al. ([NN93] et [NNA96]). Une analyse d'effet du programme ML-Act construisait un terme Cap qui approximait les aspects concurrents du programme. Un système de type classique *à la* ML (avec quelques types de base supplémentaires) vérifiait que l'aspect fonctionnel ne comportait pas d'erreur. Dans une deuxième phase, le terme Cap était analysé par les prototypes développés dans [Col97]. Cependant, il s'est avéré que cette approche n'était pas pertinente car elle limitait trop l'expressivité du langage pour rester décidable. Par exemple, une fonction récursive ne pouvait pas avoir d'effet concurrent, ainsi, il n'était pas possible d'écrire une fonction qui recevait une liste de messages et une liste d'acteurs et envoyait tous les messages à tous les acteurs. En effet, l'unification sur des algèbres de termes non libres comme le π-calcul est indécidable. Donc, l'unification nécessaire lors de l'appel d'une fonction récursive était indécidable. F. Nielson et al. ont identifié ce problème et l'ont résolu en réduisant l'expressivité du calcul. Cependant, cette stratégie ne nous permettait plus d'analyse pertinente de la communication et de la concurrence.

Le premier système de type de Cap ([Col97]) repose sur une approximation originale de l'arbre décrivant la suite des interfaces assumées par l'acteur ainsi que les messages reçus. Les *noms* (*i.e.* les acteurs) sont approximés par un ensemble de couples étiquette de message / type des arguments du message, par exemple $\langle m_1(\langle\rangle)\ \ m_2(\langle p(\langle\rangle)\rangle)\rangle$. Ensuite, deux relations de sous-typage sur les *noms* sont définies, un sous-typage usuel $\subseteq$ qui prend en compte les messages reçus ($\langle m(\alpha)\rangle \subseteq t$) et un sous-typage $\subseteq_\beta$ qui cumule les interfaces

$(\langle m(\alpha)\rangle \subseteq_\beta t)$.

$$\begin{cases} \langle m_i(\overrightarrow{\alpha_i})^{i\in I}\rangle \subseteq \langle m_j(\overrightarrow{\alpha'_j})^{j\in J}\rangle & \iff & I \subseteq J \wedge (\forall k \in I)\overrightarrow{\alpha'_k} \subseteq \overrightarrow{\alpha_k} \\ \langle m_i(\overrightarrow{\alpha_i})^{i\in I}\rangle \subseteq_\beta \langle m_j(\overrightarrow{\alpha'_j})^{j\in J}\rangle & \iff & I \subseteq J \wedge (\forall k \in I)\overrightarrow{\alpha_k} \subseteq \overrightarrow{\alpha'_k} \end{cases}$$

La seconde approche étudiée au sein de notre équipe, a été la construction de λCAP une extension de CAP par ajout d'une notion de fonction (voir [Col98]). Cette extension très générale pose cependant des problèmes importants dus au mélange fin entre le calcul fonctionnel et le calcul concurrent. Il a fallu limiter l'expressivité d'un tel langage en restreignant les aspects fonctionnels aux comportements des acteurs. Le système de type de λCAP a alors servi de base à mes travaux sur ML-ACT, il repose sur les formes de type suivantes :

- $(\chi, \psi)\triangleright$ pour les comportements, χ collectant les informations sur les interfaces et ψ collectant les messages envoyés à l'*ego* ;
- $@\psi$ pour les noms qui collectent les messages reçus.

Les ψ et χ ont la même forme que pour le système de CAP. Il est ainsi possible de rendre le système plus intuitif en séparant les deux parties d'un type comportement.

$$\begin{cases} @\psi_1 \subseteq @\psi_2 & \iff & \psi_2 \subseteq \psi_1 \\ (\chi_1, \psi_1)\triangleright \subseteq (\chi_2, \psi_2)\triangleright & \iff & \chi_1 \subseteq_\beta \chi_2 \wedge \psi_1 \subseteq \psi_2 \end{cases}$$

Soit a un acteur de type $@\psi$ et $(\chi_1, \psi_1)\triangleright$ le type de son comportement initial (fourni lors de sa création). La détection des messages orphelins est réalisée en fin de résolution par comparaison entre tous les messages reçus par a (ils figurent dans ψ et ψ_1) et tous les messages que a sait traiter (ils figurent dans χ_1). Ainsi, un acteur correct a vérifie $\psi \cup \psi_1 \subseteq \chi_1$. Cette version du système de type a été réalisée, en collaboration avec M. COLIN, dans le cadre du compilateur ML-ACT et présentée en détail dans [DPCS00]. La version présentée dans cette thèse est relativement différente du point de vue de la formalisation. Ces changements sont dus aux travaux réalisés dans le cadre d'ERLANG qui nous a permis de construire un système jugé plus intuitif reposant sur des techniques plus usuelles.

Dans la deuxième partie de mes travaux, je suis passé d'un langage universitaire à un langage industriel. Nous avons choisi d'adapter nos systèmes à ERLANG car c'était un langage à caractère industriel ayant un modèle de concurrence proche des acteurs. Il est exploité par ERICSSON pour la programmation d'applications fortement concurrentes et réparties destinées à leur équipement de télécommunication. Le système de type conçu pour l'étude des communications au sein d'ERLANG s'inspirait initialement des travaux réalisés sur ML-ACT. Cependant, la forte différence entre les deux langages au niveau de la sémantique de la partie fonctionnelle a amené une évolution importante du typage fonctionnel et l'ajout d'un calcul d'effet. Ces deux modifications ont demandé des changements assez importants dans le code des analyseurs.

ERLANG est un langage non typé et nous avons conçu un système de type qui permet de capturer les erreurs fonctionelles d'un programme ERLANG et de détecter tous les messages orphelins triviaux tels qu'ils ont été présentés dans le second chapitre. Ce travail sur le système de type d'ERLANG a été réalisé en parallèle avec l'implantation d'un prototype

d'analyseur. Celle-ci a posé des problèmes lors de la présentation des résultats de la phase de typage qui ne sont qu'évoqués dans cette thèse. En effet, l'algorithme de résolution des contraintes est inspiré des travaux de thèse de F. POTTIER [Pot98] et la gestion de la minimisation/maximisation des types pour l'affichage a soulevé des difficultés que nous n'avons qu'en partie résolues et uniquement d'un point de vue pratique. De plus, la précision du système ayant un coût en calcul très important, il a fallu optimiser la collecte et la résolution des contraintes. Ces optimisations sont relativement difficiles à mettre au point et à justifier. En effet, les relations entre le nombre de contraintes inférées, le nombre de variables de type, l'efficacité de la résolution et la qualité du résultat semblent complexes. Il semble parfois que générer des contraintes plus complexes et en plus grand nombre simplifie la résolution de manière globale. D'ailleurs, dans sa thèse, F. POTTIER présente une amélioration du système de type qui provoque une augmentation importante du nombre de variables de type. Ces aspects des systèmes de type nécessitent encore bien des travaux aussi bien théoriques que pratiques pour obtenir des analyseurs réellement efficaces et produisant une information pertinente. L'affichage de toutes les contraintes fournit l'information la plus précise mais, alors, même pour la personne qui a conçu le système, la lecture du résultat du typage devient extrêmement difficile et pénible.

Le système de type d'ERLANG est inspiré des systèmes de type basés sur les rangées. Nous avons modifié l'approximation en refusionnant les deux moitiés du type et en utilisant une seule relation de sous-typage. Les indicateurs de présences permettent alors de différencier les types sur lequel le sous-typage est covariant (les messages reçus) et ceux sur lequel il est contravariant (les messages installés).

Ce manuscrit a essayé de donner une forme cohérente à toutes ces influences. Nous avons factorisé au maximum les notions communes aux systèmes de type de ML-ACT et d'ERLANG. Ainsi, il nous a semblé intéressant de mettre au point la sémantique concurrente de manière indépendante de la partie fonctionnelle puisque les aspects concurrents des deux langages sont relativement proches. Ce travail de généralisation et d'abstraction complique la compréhension des systèmes de type puisqu'il faut maîtriser le formalisme des configurations, un noyau fonctionnel et la traduction du langage dans ces deux écritures. Par contre, il simplifie énormément le travail de preuve et permet la construction de parties communes qui diminuent la quantité de travail.

Nous proposons donc un formalisme qui permet une adaptation relativement aisée de nos systèmes (ou d'autres systèmes) à de nouveaux langages de programmation qui ont un modèle de concurrence proche (notion de processus et de communication asynchrone par message).

Actuellement, les deux systèmes de type présentés détectent tous les messages orphelins triviaux (de sûreté). Ainsi, tout envoi d'un message qu'un acteur ne saura jamais traiter est signalé. Notons que le message peut ne pas être traité parce qu'il contient une étiquette qui ne figure pas dans le potentiel, mais également parce qu'il transporte des données qui ne seront pas compatibles avec les motifs de réception.

L'extension du système de type par comptage des messages présenté dans la conclusion du chapitre 5 page 149 doit permettre d'augmenter sa précision. Il sera alors possible de détecter une partie des orphelins de vivacité. Par exemple, soit un acteur a comprenant un message `init` et qui à sa réception adopte un comportement à partir duquel il ne saura plus jamais traiter de message `init`. Si le programme contient deux envois du message

`init` à a, le second ne sera jamais traité. Le comptage détectera aisément cette forme de problème.

Pour la détection de messages orphelins de vivacité plus complexes, il faudra réintégrer une notion de causalité au sein des types. Cependant, cette extension demande une réflexion en profondeur car elle remet en cause la faisabilité de nombreux choix réalisés jusqu'à présent. Ainsi, l'inférence de types d'acteurs plus complexes (sous forme d'arbre par exemple) est probablement difficile à obtenir si elle n'est pas indécidable. Il est ainsi possible qu'il faille abandonner l'inférence complète au profit d'un système intégrant la vérification de type comme par exemple les systèmes présenté dans [NNS99b, NN99, NNS99a] ou dans [Pun00]. La forme des types devient alors plus expressive et le programmeur déclare le type de ses processus en exprimant la relation de causalité entre leurs différents états. La détection des messages orphelins de vivacité consiste alors à vérifier la compatibilité des envois avec le type de chaque processus.

Un certain nombre de systèmes de type pour l'analyse de la communication au sein des calculs de processus ont été réalisés ([PRT93, KPT96, PS96, Yos96, Bou97b, Kob97, Sew98b, YH99, DZ99, Nim99, RH99]). Mais aucun de ces systèmes ne s'intéresse à l'intégration de ces analyses dans le système de type d'un langage de programmation. Deux autres études sont en ce sens plus proches de nos travaux : d'une part, celle qui porte sur la construction de JoCaml et de son système de type ([FLMR97]) et d'autre part, la mise au point du langage Tyco ([Rav00]). Ces deux approches suivent le cheminement de ML-Act, c'est-à-dire, qu'ils construisent un nouveau langage de programmation concurrent fortement typé à partir d'un calcul de processus. La principale différence entre ML-Act et ces deux approches sont le choix du modèle de programmation des acteurs qui de par son caractère fortement dynamique (au niveau des interfaces) est plus complexe à typer.

En revanche, l'approche que nous avons suivie dans le cadre de l'analyse d'Erlang semble originale. En effet, très peu de travaux étudient la construction de système d'analyse statique pour des langages industriels fortement dynamiques. Les travaux qui ont lieu dans ce cadre se limitent au typage de la partie fonctionnelle [Lin96, MW97a] ou alors utilisent des techniques d'interprétation abstraite combinées avec du *model checking* pour analyser les communications [Huc99, DåFG98]. L'approche à base de typage nous semble plus habituelle pour le programmeur et est complémentaire de ses outils d'analyses plus complexes basés sur la logique temporelle qui seront plutôt utilisés par les concepteurs de l'architecture d'une application.

En Erlang il est possible de construire dynamiquement des atomes, le calcul statique des fonctions appelées n'est donc, en général, pas possible. Le typage d'Erlang ne peut donc pas être complet, nous pensons cependant l'utilisation de cette caractéristique reste marginale et qu'une approche pragmatique qui consiste à ajouter un certain degré de vérification ne peut qu'être profitable aux programmeurs. Ainsi, notre objectif n'est pas de convertir le langage Erlang en un langage fortement typé, mais de fournir un outil capable de détecter de nombreuses erreurs potentielles dans un programme Erlang.

Les prototypes réalisés au cours de cette thèse permettront l'étude critique des analyses proposées et ainsi il sera possible de mesurer leur qualité vis-à-vis de l'utilisateur. En effet, il faut déterminer plus précisément l'apport des informations de type que notre système fournit actuellement et ainsi, éventuellement déterminer les axes de sophistication nécessaires des analyses.

A plus long terme, il serait également intéressant d'étudier en détail certaines autres options choisies par nos outils :

- l'utilisation du *tout inférence*. En effet, souvent le programmeur a une idée relativement précise de ce que son code doit réaliser. S'il spécifiait explicitement certaines informations, cela simplifierait le travail du système de type mais également augmenterait sa précision. Cependant, l'ajout d'informations par le développeur ne doit pas introduire une nouvelle source d'erreurs. Il faut donc obtenir un compromis satisfaisant du point de vue de l'utilisateur et du système, sans trop alourdir la tâche du programmeur.
- l'intégration avec de l'analyse dynamique. Les applications actuelles utilisent de plus en plus la possibilité de télécharger automatiquement du code qu'elles exécutent ensuite (applet par exemple). Comment un processus de vérification statique peut se combiner avec des aspects totalement dynamiques ? Des études sont nécessaires pour mieux comprendre l'éventuelle coopération de ces deux aspects a priori contradictoires.
- le but de ces outils. En effet, ces outils sont actuellement destinés particulièrement à la détection des messages orphelins qui sont des erreurs de synchronisation ou d'exploitation de protocole de communication. Or, les analyses statiques peuvent également apporter une aide pour garantir le respect :
  - de comportements prédéfinis et ainsi diminuer les erreurs d'exécution au sein des systèmes répartis ;
  - de contraintes particulières d'utilisation comme, par exemple, la fiabilité, la disponibilité ou la qualité d'un service ;
  - de politiques de sécurité ;
  - de règles de collaboration et ainsi permettre une composition sûre de services.
- leur domaine d'application. Même si le champs d'application de telles méthodes est bien plus large, il nous semble que les domaines des applications de télécommunication, de services internet (commerce électronique) ou de logiciels de commande embarqués (dans les avions, voitures ...) sont à privilégier du fait de leur caractère critique nécessitant une garantie de leur fonctionnement. Ces domaines spécifiques permettront d'adapter les systèmes à leur objectif et ainsi les optimiser.

Annexe A

Grammaires

A.1 ML-Act

<*prog*> ::= [<*inst*> ';;']$^+$
<*inst*> ::= <*expr*>
| 'label' IDENT [',' IDENT]*
| 'let' <*def*> ['and' <*def*>]*
| 'let' 'rec' <*defrec*> ['and' <*defrec*>]*
| 'behavior' <*comp*> ['and' <*comp*>]*'end'
<*def*> ::= [<*motif*>]$^+$'=' <*expr*>
<*defrec*> ::= IDENT [<*motif*>]*'=' <*expr*>
<*comp*> ::= IDENT [<*motif*>]*'=' [<*rea*>]$^+$
<*rea*> ::= 'message' IDENT [<*motif*>] '=' <*expr*>
<*expr*> ::= IDENT
| <*const*>
| '(' [<*expr*> [',' <*expr*>]*] ')'
| '[' [<*expr*> [';' <*expr*>]*] ']'
| <*expr*> <*expr*>
| <*expr*> <*bin*> <*expr*>
| <*un*> <*expr*>
| 'let' <*def*> ['and' <*def*>]*'in' <*expr*>
| 'let' 'rec' <*defrec*> ['and' <*defrec*>]*'in' <*expr*>
| 'function' [<*filtre*>]$^+$
| 'match' <*expr*> 'with' [<*filtre*>]$^+$
| 'if' <*expr*> 'then' <*expr*> ['else' <*expr*>]
| 'ego'
| 'self'
| 'send' IDENT <*expr*> 'to' <*expr*>
| 'new' <*expr*>
| 'become' <*expr*>
| 'suicide'
<*filtre*> ::= '|' <*motif*> '->' <*expr*>

```
<motif>  ::= IDENT
           | <const>
           | '_'
           | '(' [<motif> [',' <motif>]*] ')'
           | '[' [<motif> [';' <motif>]*] ']'
           | <motif> '::' <motif>
<const>  ::= INT | FLOAT | CHAR | BOOL
           | STRING
<bin>    ::= ';' | '::' | '@' | '=' | '<>' | '<' | '>' | '<='
           | '>=' | '&' | 'or' | '^' | '+' | '+.' | '-' | '-.'
           | '*' | '*.' | '/' | '/.' | 'mod'
<un>     ::= '-' | '-.' | 'not'
```

A.2 Erlang

```
<programme>    ::= [<directive> '.' | <fonction> '.']+
<directive>    ::= '-' 'module' '(' ATOM ')'
                 | '-' 'export' '(' '[' <arites> ']' ')'
                 | '-' 'import' '(' ATOM ',' '[' <arites> ']' ')'
                 | '-' 'record' '(' ATOM ',' '{' <attribut> '}' ')'
                 | '-' ATOM '(' ATOM ')'
<attribut>     ::= ATOM ['=' <expr>] [',' ATOM ['=' <expr>]]*
<arites>       ::= <arite> [',' <arite>]*
<arite>        ::= ATOM '/' NUMBER
<fonction>     ::= ATOM <definition> [';' ATOM <definition>]*
<definition>   ::= '(' [<filtres>] ')' ['when' <gardes>] '->' <exprs>
<filtres>      ::= <filtre> [',' <filtre>]*
<filtre>       ::= VAR | '_' | <constante>
                 | '[' [<filtres> ['|' <filtre>]] ']'
                 | '{' [<filtres>] '}'
                 | '#' ATOM '{' [<champs>] '}'
<champs>       ::= ATOM '=' <filtre> [',' ATOM '=' <filtre>]
<gardes>       ::= <garde> [',' <garde>]*
<garde>        ::= 'true' | '(' <garde> ')'
                 | 'record' '(' <garde_expr> ',' ATOM ')'
                 | <bif_rec> '(' <garde_expr> ')'
                 | <garde_expr> <comp_op> <garde_expr>
<garde_exprs>  ::= <garde_expr> [',' <garde_expr>]*
<garde_expr>   ::= VAR | <constante>
                 | '(' <garde_expr> ')'
                 | '[' [<garde_exprs> ['|' <garde_expr>]] ']'
                 | '{' <garde_exprs> '}'
```

	| *<bif_gar>* '(' [*<garde_exprs>*] ')'
	| [*<garde_expr>*] '#' ATOM '.' ATOM
	| *<garde_expr>* *<bin_op>* *<garde_expr>*
	| *<un_op>* *<garde_expr>*
<exprs>	::= *<expr>* [',' *<expr>*]*
<expr>	::= VAR | *<constante>*
	| '(' *<expr>* ')'
	| 'begin' *<exprs>* 'end'
	| '[' [*<exprs>* ['|' *<expr>*] | *<expr>* '||' *<conds>*] ']'
	| '{' [*<exprs>*] '}'
	| [*<expr>* ':'] *<expr>* '(' [*<exprs>*] ')'
	| *<expr>* *<bin_op>* *<expr>*
	| *<un_op>* *<expr>*
	| 'catch' *<expr>*
	| 'if' *<clauses>* 'end'
	| *<filtre>* ['=' *<filtre>*]*'=' *<expr>*
	| *<expr>* '!' *<expr>*
	| 'case' *<expr>* 'of' *<filtrages>* 'end'
	| 'receive' *<filtrages>* ['after' *<expr>* '->' *<expr>*] 'end'
	| 'receive' 'after' *<expr>* '->' *<expr>* 'end'
	| 'fun' *<arite>*
	| 'fun' *<definition>* [';' *<definition>*]*'end'
	| *<expr>* '#' ATOM ['.' ATOM | '{' [*<inits>*] '}']
<inits>	::= ATOM '=' *<expr>* [',' ATOM '=' *<expr>*]*
<conds>	::= [*<filtre>* '<-'] *<expr>* [',' [*<filtre>* '<-'] *<expr>*]*
<filtrages>	::= *<filtrage>* [';' *<filtrage>*]*
<filtrage>	::= *<filtre>* ['when' *<gardes>*] '->' *<exprs>*
<clauses>	::= *<gardes>* '->' *<exprs>* [';' *<gardes>* '->' *<exprs>*]*
<constante>	::= ATOM | NUM | STRING | 'true'
<bin_op>	::= *<log_add_op>* | *<log_mul_op>* | *<comp_op>* | *<list_op>*
	| *<add_op>* | *<mul_op>*
<log_add_op>	::= 'or' | 'xor'
<log_mul_op>	::= 'and'
<comp_op>	::= '<' | '=<' | '>' | '>=' | '=:=' | '=/=' | '==' | '/='
<list_op>	::= '++' | '--'
<add_op>	::= '+' | '-' | 'bor' | 'bxor' | 'bsl' | 'bsr'
<mul_op>	::= '*' | '/' | 'div' | 'rem' | 'band'
<un_op>	::= '+' | '-' | 'not' | 'bnot'
<bif_rec>	::= 'atom' | 'binary' | 'constant' | 'float' | 'integer' | 'list'
	| 'number' | 'pid' | 'port' | 'reference' | 'tuple'
<bif_gar>	::= *<bif_rec>* | 'abs' | 'element' | 'hd' | 'length' | 'node'
	| 'nodes' | 'round' | 'self' | 'size' | 'tl' | 'trunc'

Annexe B

Sémantiques

Cette annexe contient un rappel de toutes les règles de la sémantique des deux langages étudiés dans cette thèse : ML-Act et μErlang. La première section résume ainsi la sémantique de ML-Act, c'est-à-dire celle des configurations et celle de $\mathcal{F}unc_1$. La seconde section rappelle quant à elle, la sémantique des configuration et $\mathcal{F}unc_2$ pour exprimer celle de μErlang. Notons que la partie sémantique des configurations figurent donc dans les deux sections.

B.1 ML-Act

La sémantique de ML-Act se compose de quatre parties : la définition des configurations et de leur congruence, celle du filtrage dans $\mathcal{F}unc_1$, la notion de potentiel en ML-Act et enfin toutes les règles de réductions (sur les configurations aussi bien que sur $\mathcal{F}unc_1$). Avant de l'introduire rappellons la syntaxe des configurations et celle de $\mathcal{F}unc_1$:

$$
\begin{array}{rcl}
w & ::= & \epsilon \mid \mathbf{Err} \mid \nu a.w \mid w \parallel w \mid a \triangleleft m(v,\ldots,v) \mid \star \triangleright e \mid \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e \\
q & ::= & \varnothing \mid m(v,\ldots,v) :: q \\
e & ::= & x \mid a \mid c \mid f \mid C(e,\ldots,e) \mid (e) \mid e\ e \mid \lambda[p.e,\ldots,p.e] \mid \texttt{letrec}\ p = e\ \texttt{in}\ e \\
p & ::= & \_ \mid x \mid c \mid C(p,\ldots,p) \\
v & ::= & a \mid c \mid f \mid C(v,\ldots,v) \mid \lambda[p.e,\ldots,p.e]
\end{array}
$$

Avec :

- $a \in \mathbb{A}$: ensemble des adresses ;
- $x \in \mathbb{V}$: ensemble des variables ;
- $m \in \mathbb{M}$: ensemble des étiquettes de message ;
- $c \in \mathbb{C} \triangleq \mathbb{N} \cup \mathbb{D} \cup \{\mathit{true},\ \mathit{false}\} \cup \mathit{Char} \cup \mathit{Char}^n \cup \{\mathbf{Err}\}$: ensemble des constantes
- $f \in \mathcal{F}_c \triangleq \{\mathit{send},\ \mathit{new},\ \mathit{init},\ \mathit{become},\ \texttt{@},\ \texttt{\^{}},\ \texttt{=},\ \texttt{<},\ \ldots,\ \texttt{or},\ \texttt{\&},\ \texttt{+},\ \texttt{+.},\ \ldots,\ \texttt{not},\ \texttt{hd},\ \ldots\}$: ensemble des fonctions prédéfinies ;
- $C \in \mathcal{C}\{\mathit{nop},\ \mathit{nil},\ \mathit{cons},\ \mathit{tuple2},\ \ldots,\ \mathit{tuplen}\} \cup \mathbb{M}$: ensemble des constructeurs.

Noms libres, substitution de noms et congruence

Il est alors possible de définir la notion de nom libre dans une configuration de ML-ACT:

$$\mathcal{FN}(\epsilon) = \mathcal{FN}(\mathbf{Err}) = \{\} \qquad \mathcal{FN}(a \triangleleft m(v_1, \ldots, v_n)) = \{a\} \cup \bigcup_{i=1}^{n} \mathcal{FN}(v_i)$$

$$\mathcal{FN}(\nu a.w) = \mathcal{FN}(w) \backslash \{a\} \qquad \mathcal{FN}(w_1 \parallel w_2) = \mathcal{FN}(w_1) \cup \mathcal{FN}(w_2)$$

$$\mathcal{FN}(\star \triangleright e) = \mathcal{FN}(e) \qquad \mathcal{FN}(\langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e) = \{a\} \cup \mathcal{FN}(q) \cup \mathcal{FN}(e)$$

$$\mathcal{FN}(\varnothing) = \{\} \qquad \mathcal{FN}(m(v_1, \ldots, v_n) :: q) = \bigcup_{i=1}^{n} \mathcal{FN}(v_i) \cup \mathcal{FN}(q)$$

$$\mathcal{FN}(a) = \{a\} \qquad \mathcal{FN}(c) = \mathcal{FN}(x) = \{\} \qquad \mathcal{FN}((e)) = \mathcal{FN}(e)$$

$$\mathcal{FN}(C(e_1, \ldots, e_n)) = \bigcup_{i=1}^{n} \mathcal{FN}(e_i) \qquad \mathcal{FN}(e_1\ e_2) = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2)$$

$$\mathcal{FN}(\texttt{letrec}\ p = e_1\ \texttt{in}\ e_2) = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2) \qquad \mathcal{FN}(\lambda[p_i.e_i]^{i \in I}) = \bigcup_{i=1}^{n} FN(e_i)$$

La notion de substitution de noms suit alors :

$$[a'/a]\epsilon = \epsilon \qquad [a'/a]\mathbf{Err} = \mathbf{Err}$$

$$[a'/a](b \triangleleft m(v_1, \ldots, v_n)) = [a'/a]b \triangleleft m([a'/a]v_1, \ldots, [a'/a]v_n) \qquad [a'/a]\nu a.w = \nu a.w$$

$$[a'/a]\nu b.w = \nu b.[a'/a]w \qquad [a'/a](w_1 \parallel w_2) = [a'/a]w_1 \parallel [a'/a]w_2$$

$$[a'/a] \star \triangleright e = \star \triangleright [a'/a]e \qquad [a'/a]\langle b \,|\, q \rangle_{\mathcal{P}} \triangleright e = \langle [a'/a]b \,|\, [a'/a]q \rangle_{\mathcal{P}} \triangleright [a'/a]e$$

$$[a'/a]\varnothing = \varnothing \qquad [a'/a](m(v_1, \ldots, v_n) :: q) = m([a'/a]v_1, \ldots, [a'/a]v_n) :: [a'/a]q$$

$$[a'/a]a = a' \qquad [a'/a]b = b \qquad [a'/a]c = c \qquad [a'/a]x = x \qquad [a'/a](e) = ([a'/a]e)$$

$$[a'/a]C(e_1, \ldots, e_n) = C([a'/a]e_1, \ldots, [a'/a]e_n) \qquad [a'/a]e_1\ e_2 = [a'/a]e_1\ [a'/a]e_2$$

$$[a'/a](\texttt{letrec}\ p = e_1\ \texttt{in}\ e_2) = \texttt{letrec}\ p = [a'/a]e_1\ \texttt{in}\ [a'/a]e_2$$

$$[a'/a]\lambda[p_i.e_i]^{i \in I} = \lambda[p_i.[a'/a]e_i]^{i \in I}$$

La relation de congruence $\equiv$ sur les configurations est définie par :

$$\nu a.w \equiv \nu b.[b/a]w \text{ si } b \notin \mathcal{FN}(w) \qquad \nu a.w \equiv w \text{ si } a \notin \mathcal{FN}(w) \qquad \nu a_1.\nu a_2.w \equiv \nu a_2.\nu a_1.w$$

$$\nu a.w_1 \parallel w_2 \equiv \nu a.(w_1 \parallel w_2) \text{ si } a \notin \mathcal{FN}(w_2) \qquad \langle a \,|\, \varnothing \rangle \triangleright \lambda[] \equiv \epsilon \qquad w \parallel \epsilon \equiv w$$

$$w \parallel \mathbf{Err} \equiv \mathbf{Err} \qquad w_1 \parallel w_2 \equiv w_2 \parallel w_1 \qquad (w_1 \parallel w_2) \parallel w_3 \equiv w_1 \parallel (w_2 \parallel w_3) \qquad \star \triangleright v \equiv \epsilon$$

Substitution de variables et filtrage

Ensuite, nous définissons la notion de variable libre d'un motif :

$$\mathcal{FV}(\_) = \mathcal{FV}(c) = \{\} \qquad \mathcal{FV}(x) = \{x\} \qquad \mathcal{FV}(C(p_1, \ldots, p_n)) = \bigcup_{i=1}^{n} \mathcal{FV}(p_i)$$

Puis nous en déduisons la substitution d'une variable par une valeur sémantique sur une expression de $\mathcal{F}unc_1$:

$$[v/x]a = a \qquad [v/x]c = c \qquad [v/x]x' = \begin{cases} v & \text{si } x' = x \\ x' & \text{sinon} \end{cases} \qquad [v/x]f = f$$

$$[v/x](e') = ([v/x]e') \qquad [v/x]C(e_1, \ldots, e_n) = C([v/x]e_1, \ldots, [v/x]e_n)$$

$$[v/x]e_1\ e_2 = [v/x]e_1\ [v/x]e_2$$

$$[v/x]\lambda[p_i.e_i]^{i \in I} = \lambda[p_i.e'_i]^{i \in I} \text{ avec } e'_i = \begin{cases} e_i & \text{si } x \in \mathcal{FV}(p_i) \\ [v/x]e_i & \text{sinon} \end{cases}$$

$$[v/x](\texttt{letrec}\ p = e_1\ \texttt{in}\ e_2) = \texttt{letrec}\ p = e'_1\ \texttt{in}\ e'_2 \text{ avec } e'_i = \begin{cases} e_i & \text{si } x \in \mathcal{FV}(p) \\ [v/x]e_i & \text{sinon} \end{cases}$$

Nous terminons alors cette sous-section par les règles de déduction exprimant la sémantique du filtrage dans $\mathcal{F}unc_1$:

$$\frac{}{v/\_ \Rightarrow []} \qquad \frac{}{v/x \Rightarrow [v/x]} \qquad \frac{}{c/c \Rightarrow []} \qquad \frac{v \neq c}{v/c \Rightarrow \mathit{failed}}$$

$$\frac{\forall k\ \forall i\ \forall v_i \quad v \neq C(v_1, \ldots, v_k)}{v/C(p_1, \ldots, p_n) \Rightarrow \mathit{failed}} \qquad \frac{k \neq n}{C(v_1, \ldots, v_k)/C(p_1, \ldots, p_n) \Rightarrow \mathbf{Err}}$$

$$\frac{\forall i \quad v_i/p_i \Rightarrow \sigma_i}{C(v_1, \ldots, v_n)/C(p_1, \ldots, p_n) \Rightarrow \sigma_1 \circ \cdots \circ \sigma_n}$$

Potentiel

On définit, alors, le calcul du potentiel d'un acteur par le calcul du potentiel de l'expression de corps :

$$\textsc{Potentiel:}\ \frac{\mathcal{P}(e) = V,\ E}{\mathcal{P}_e(e) = setof(V) \cup E}$$

$$\textsc{Val1:}\ \frac{v \in \mathbb{A} \cup \mathbb{C} \cup \mathcal{F}_c \cup \{nop,\ nil\}}{\mathcal{P}(v) = \{\},\ \{\}} \qquad \textsc{Val2:}\ \frac{V \in \mathcal{V}al}{\mathcal{P}(V) = V,\ \{\}} \qquad \textsc{Var:}\ \frac{}{\mathcal{P}(x) = \{\},\ \{\}}$$

$$\textsc{Mess:}\ \frac{\mathcal{P}(e_i) = V_i,\ E_i}{\mathcal{P}(m(\vec{e})) = \{\},\ \bigcup_i E_i} \qquad \textsc{Const:}\ \frac{C \in \{cons,\ tuple2, \ldots,\ tuplen\} \quad \mathcal{P}(e_i) = V_i,\ E_i}{\mathcal{P}(C(\vec{e})) = \bigcup_i V_i,\ \bigcup_i E_i}$$

$$\textsc{Int:}\ \frac{\vec{x}_i = \mathcal{FV}(p_i) \quad \mathcal{P}([\mathbb{M}/\vec{x}_i]e_i) = V_i,\ E_i}{\mathcal{P}(\lambda[m_i(p_i).e_i]^{i \in I}) = \bigcup_i \{m_i\},\ \bigcup_i E_i} \qquad \textsc{Bec:}\ \frac{\mathcal{P}(e) = V,\ E}{\mathcal{P}(become(a,\ e)) = \{\},\ setof(V) \cup E}$$

$$\textsc{App:}\ \frac{\mathcal{P}(e) = \bigcup_j V_j,\ E \quad \mathcal{P}(e') = V',\ E' \quad App(V_j, V') = V'_j,\ E_j}{\mathcal{P}(e\ e') = \bigcup_j V'_j,\ \bigcup_j E_j \cup E \cup E'}$$

$$\textsc{LRec1:}\ \frac{\mathcal{P}(e) = V,\ E \quad \mathcal{P}([V/x]e') = V',\ E'}{\mathcal{P}(\texttt{letrec}\ x = e\ \texttt{in}\ e') = V',\ E \cup E'}$$

$$\textsc{LRecn:}\ \frac{\mathcal{P}(\texttt{letrec}\ \vec{x}|_i = \vec{e}|_i\ \texttt{in}\ e_i) = V_i,\ E_i \quad \mathcal{P}([V_1/x_1] \ldots [V_n/x_n]e) = V,\ E}{\mathcal{P}(\texttt{letrec}\ \vec{x} = \vec{e}\ \texttt{in}\ e) = V,\ \bigcup_i E_i \cup E}$$

$$\textsc{App1:}\ \frac{\vec{x}_i = \mathcal{FV}(p_i) \quad \mathcal{P}([V/\vec{x}_i]e_i) = V_i,\ E_i}{App(\lambda[p_i.e_i]^{i \in I}, V) = \bigcup_i V_i,\ \bigcup_i E_i} \qquad \textsc{App2:}\ \frac{}{App(V, V') = \{\},\ \{\}}$$

Réduction

Enfin, la section rappellant la sémantique de ML-Act se termine par toutes les règles de déduction exprimant sa sémantique:

$$\textsc{BalE:}\ \frac{\exists j \quad (\forall i < j\ \ m/p_i \Rightarrow failed) \qquad m/p_j \Rightarrow \mathbf{Err}}{\lambda[p_i.e_i]^{i \in I} \prec m :: q \Longrightarrow q,\ \mathbf{Err},\ []}$$

$$\textsc{Bal1:}\ \frac{\exists j \quad (\forall i < j\ \ m/p_i \Rightarrow failed) \qquad m/p_j \Rightarrow \sigma}{\lambda[p_i.e_i]^{i \in I} \prec m :: q \Longrightarrow q,\ e_j,\ \sigma}$$

$$\textsc{Bal2:}\ \frac{(\forall i \in I\ \ m/p_i \Rightarrow failed) \qquad \lambda[p_i.e_i]^{i \in I} \prec q \Longrightarrow q',\ e,\ \sigma}{\lambda[p_i.e_i]^{i \in I} \prec m :: q \Longrightarrow m :: q',\ e,\ \sigma}$$

$$\textsc{Cong}: \frac{w_1 \equiv w_1' \quad w_1' \longrightarrow w_2' \quad w_2' \equiv w_2}{w_1 \longrightarrow w_2} \qquad \textsc{Par}: \frac{w_1 \longrightarrow w_2}{w||w_1 \longrightarrow w||w_2}$$

$$\textsc{Res}: \frac{w_1 \longrightarrow w_2}{\nu a.w_1 \longrightarrow \nu a.w_2} \qquad \textsc{Exp}: \frac{e_1 \longrightarrow_e e_2}{\alpha \triangleright e_1 \longrightarrow \alpha \triangleright e_2} \qquad \textsc{ExpE}: \frac{e \longrightarrow_e \mathbf{Err}}{\alpha \triangleright e \longrightarrow \mathbf{Err}}$$

$$\textsc{Rea}: \frac{\mathcal{R} \prec q \Longrightarrow q',\ e,\ \sigma}{\langle a \,|\, q\rangle \triangleright \mathcal{R} \longrightarrow \langle a \,|\, q'\rangle \triangleright \sigma(e)} \qquad \textsc{Rcv}: \frac{m \in \mathcal{P}}{\langle a \,|\, q\rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m \longrightarrow \langle a \,|\, q :: m\rangle_{\mathcal{P}} \triangleright e}$$

$$\textsc{RcvE}: \frac{m \notin \mathcal{P}}{\langle a \,|\, q\rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m \longrightarrow \mathbf{Err}} \qquad \textsc{Send}: \frac{}{\alpha \triangleright \mathrm{C}[send(m,a)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel a \triangleleft m}$$

$$\textsc{SendE}: \frac{v_1 \notin \mathcal{M}ess \vee v_2 \notin \mathbb{A}}{\alpha \triangleright \mathrm{C}[send\ (v_1, v_2)] \longrightarrow \mathbf{Err}} \qquad \textsc{New}: \frac{a \notin \mathcal{FN}(\alpha \triangleright \mathrm{C}[new])}{\alpha \triangleright \mathrm{C}[new] \longrightarrow \nu a.(\alpha \triangleright \mathrm{C}[a])}$$

$$\textsc{Init}: \frac{}{\alpha \triangleright \mathrm{C}[init(a,v)] \longrightarrow \alpha \triangleright \mathrm{C}[nop] \parallel \langle a \,|\, \varnothing\rangle_{\mathcal{P}(v\ a)} \triangleright v\ a}$$

$$\textsc{InitE}: \frac{v_1 \notin \mathbb{A} \vee v_2 \notin \mathbb{A} \rightarrow \mathbb{I}}{\alpha \triangleright \mathrm{C}[init\ (v_1, v_2)] \longrightarrow \mathbf{Err}} \qquad \textsc{Bec}: \frac{}{\alpha \triangleright \mathrm{C}[become(a, \mathcal{R})] \longrightarrow \alpha \triangleright \mathcal{R} \parallel \star \triangleright \mathrm{C}[nop]}$$

$$\textsc{BecE}: \frac{v_1 \notin \mathbb{A} \vee v_2 \notin \mathbb{I}}{\alpha \triangleright \mathrm{C}[become(v_1, v_2)] \longrightarrow \mathbf{Err}}$$

$$\textsc{Cont}: \frac{e \longrightarrow_e e'}{\mathrm{C}[e] \longrightarrow_e \mathrm{C}[e']} \qquad \textsc{ContE}: \frac{e \longrightarrow_e \mathbf{Err}}{\mathrm{C}[e] \longrightarrow_e \mathbf{Err}} \qquad \textsc{VarE}: \frac{}{x \longrightarrow_e \mathbf{Err}}$$

$$\textsc{AppE1}: \frac{v \notin \mathcal{F} \cup \mathcal{F}_c}{v\ e \longrightarrow_e \mathbf{Err}} \qquad \textsc{AppE2}: \frac{}{\lambda[]\ e \longrightarrow_e \mathbf{Fail}} \qquad \textsc{AppE3}: \frac{v/p_1 \Rightarrow \mathbf{Err}}{\lambda[p_i.e_i]^{i\in 1..n}\ v \longrightarrow_e \mathbf{Err}}$$

$$\textsc{AppF}: \frac{v/p_1 \Rightarrow failed}{\lambda[p_i.e_i]^{i\in 1..n}\ v \longrightarrow_e \lambda[p_i.e_i]^{i\in 2..n}\ v} \qquad \textsc{App}: \frac{v/p_1 \Rightarrow \sigma}{\lambda[p_i.e_i]^{i\in 1..n}\ v \longrightarrow_e \sigma(e_1)}$$

$$\textsc{LRec}: \frac{\sigma_i = [\texttt{letrec}\ \vec{x} = \vec{v}\ \texttt{in}\ v_i/x_i]}{\texttt{letrec}\ \vec{x} = \vec{v}\ \texttt{in}\ e \longrightarrow_e \sigma_1 \circ \cdots \circ \sigma_n(e)}$$

B.2 ERLANG

La sémantique de μErlang est également décomposée en quatre parties : la définition des configurations et de leur congruence, celle du filtrage dans $\mathcal{F}unc_2$, la notion de potentiel sur μErlang et enfin toutes les règles de réductions (sur les configurations aussi bien que sur $\mathcal{F}unc_2$). Avant de l'introduire, rappelons la syntaxe des configurations et celle de $\mathcal{F}unc_2$

:

$$
\begin{array}{rcl}
w & ::= & \epsilon \mid \mathbf{Err} \mid \nu a.w \mid w \parallel w \mid a \triangleleft v \mid \star \triangleright e \mid \langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e \\
q & ::= & \varnothing \mid v :: q \\
e & ::= & x \mid a \mid c \mid f \mid C(e, \ldots, e) \mid (e) \mid e\,;e \mid e\ e \mid \lambda[p, e.e, \ldots, p, e.e] \mid \texttt{let}\ p = e \\
p & ::= & \_ \mid x \mid c \mid C(p, \ldots, p) \\
v & ::= & a \mid c \mid f \mid C(v, \ldots, v) \mid \lambda[p, e.e, \ldots, p, e.e]
\end{array}
$$

Avec :

- $a \in \mathbb{A}$: ensemble des adresses ;
- $x \in \mathbb{V}$: ensemble des variables ;
- $c \in \mathbb{C} \triangleq \mathbb{N} \cup \mathbb{D} \cup \mathbb{A}_t \cup \mathit{Char} \cup \mathit{Char}^n \cup \{\mathbf{Err}\}$: ensemble des constantes ;
- $f \in \mathcal{F}_c \triangleq \{\mathit{send},\ \mathit{new},\ \mathit{init},\ \mathit{receive},\ \texttt{++},\ \texttt{==},\ \texttt{=<}, \ldots,\ \texttt{or}, \ldots,\ \texttt{+},\ \texttt{-}, \ldots,\ \texttt{atom},\ \texttt{constant},\ \texttt{float},\ \texttt{integer},\ \texttt{list},\ \texttt{number},\ \texttt{pid},\ \texttt{tuple},\ \texttt{hd},\ \texttt{element}, \ldots\}$: ensemble des fonctions prédéfinies ;
- $C \in \mathcal{C}\{\mathit{nop},\ \mathit{nil},\ \mathit{cons},\ \mathit{tuple2},\ \ldots,\ \mathit{tuplen}\}$: ensemble des constructeurs.

Noms libres, substitution de noms et congruence

On peut alors définir la notion de nom libre dans une configuration de μErlang :

$$\mathcal{FN}(\epsilon) = \mathcal{FN}(\mathbf{Err}) = \{\} \qquad \mathcal{FN}(a \triangleleft v) = \{a\} \cup \mathcal{FN}(v) \qquad \mathcal{FN}(\nu a.w) = \mathcal{FN}(w) \setminus \{a\}$$

$$\mathcal{FN}(w_1 \parallel w_2) = \mathcal{FN}(w_1) \cup \mathcal{FN}(w_2) \qquad \mathcal{FN}(\star \triangleright e) = \mathcal{FN}(e)$$

$$\mathcal{FN}(\langle a \,|\, q \rangle_{\mathcal{P}} \triangleright e) = \{a\} \cup \mathcal{FN}(q) \cup \mathcal{FN}(e)$$

$$\mathcal{FN}(\varnothing) = \{\} \qquad \mathcal{FN}(v :: q) = \mathcal{FN}(v) \cup \mathcal{FN}(q)$$

$$\mathcal{FN}(a) = \{a\} \qquad \mathcal{FN}(c) = \mathcal{FN}(x) = \mathcal{FN}(f) = \{\} \qquad \mathcal{FN}((e)) = \mathcal{FN}(e)$$

$$\mathcal{FN}(C(e_1, \ldots, e_n)) = \bigcup_{i=1}^{n} \mathcal{FN}(e_i) \qquad \mathcal{FN}(e_1; e_2) = \mathcal{FN}(e_1\ e_2) = \mathcal{FN}(e_1) \cup \mathcal{FN}(e_2)$$

$$\mathcal{FN}(\texttt{let}\ p = e) = \mathcal{FN}(p) \cup \mathcal{FN}(e)$$

$$\mathcal{FN}(\lambda[p_i, g_i.e_i]^{i \in I}) = \bigcup_{i=1}^{n} (\mathcal{FN}(p_i) \cup \mathcal{FN}(g_i) \cup \mathcal{FN}(e_i))$$

On construit alors la notion de substitution de noms :

$$[a'/a]\epsilon = \epsilon \qquad [a'/a]\mathbf{Err} = \mathbf{Err} \qquad [a'/a](b \triangleleft v) = [a'/a]b \triangleleft [a'/a]v \qquad [a'/a]\nu a.w = \nu a.w$$

$$[a'/a]\nu b.w = \nu b.[a'/a]w \qquad [a'/a](w_1 \parallel w_2) = [a'/a]w_1 \parallel [a'/a]w_2$$

$$[a'/a] \star \triangleright e = \star \triangleright [a'/a]e \qquad [a'/a](\langle b \,|\, q\rangle_{\mathcal{P}} \triangleright e) = \langle [a'/a]b \,|\, [a'/a]q\rangle_{\mathcal{P}} \triangleright [a'/a]e$$

$$[a'/a]\varnothing = \varnothing \qquad [a'/a]v :: q = [a'/a]v :: [a'/a]q$$

$$[a'/a]a = a' \qquad [a'/a]e = e \text{ si } e \in \mathbb{C} \cup \mathbb{V} \cup \mathcal{F}_c \cup \{\_\} \qquad [a'/a](e) = ([a'/a]e)$$

$$[a'/a]C(e_1, \ldots, e_n) = C([a'/a]e_1, \ldots, [a'/a]e_n) \qquad [a'/a]e_1 \,;\, e_2 = [a'/a]e_1 \,;[a'/a]e_2$$

$$[a'/a]e_1\ e_2 = [a'/a]e_1\ [a'/a]e_2 \qquad [a'/a](\texttt{let}\ p = e) = \texttt{let}\ [a'/a]p = [a'/a]e$$

$$[a'/a]\lambda[p_i, g_i.e_i]^{i\in I} = \lambda[[a'/a]p_i, [a'/a]g_i.[a'/a]e_i]^{i\in I}$$

On définit alors la relation de congruence $\equiv$ sur les configurations :

$$\nu a.w \equiv \nu b.[b/a]w \text{ si } b \notin \mathcal{FN}(w) \qquad \nu a.w \equiv w \text{ si } a \notin \mathcal{FN}(w) \qquad \nu a_1.\nu a_2.w \equiv \nu a_2.\nu a_1.w$$

$$\nu a.w_1 \parallel w_2 \equiv \nu a.(w_1 \parallel w_2) \text{ si } a \notin \mathcal{FN}(w_2) \qquad \langle a \,|\, \varnothing\rangle \triangleright \lambda[] \equiv \epsilon \qquad w \parallel \epsilon \equiv w$$

$$w \parallel \mathbf{Err} \equiv \mathbf{Err} \qquad w_1 \parallel w_2 \equiv w_2 \parallel w_1 \qquad (w_1 \parallel w_2) \parallel w_3 \equiv w_1 \parallel (w_2 \parallel w_3) \qquad \star \triangleright v \equiv \epsilon$$

Substitution de variables et filtrage

Ensuite, nous définissons la notion de substitution d'une variable par une valeur sémantique sur une expression de $\mathcal{F}unc_2$:

$$[v/x]e = e \text{ si } e \in \mathbb{A} \cup \mathbb{C} \cup \mathcal{F}_c \cup (\mathbb{V} \setminus \{x\}) \cup \{\_\} \qquad [v/x]x = v \qquad [v/x](e') = ([v/x]e')$$

$$[v/x]C(e_1, \ldots, e_n) = C([v/x]e_1, \ldots, [v/x]e_n) \qquad [v/x]e_1 \,;\, e_2 = [v/x]e_1 \,;[v/x]e_2$$

$$[v/x]e_1\ e_2 = [v/x]e_1\ [v/x]e_2 \qquad [v/x]\lambda[p_i, g_i.e_i]^{i\in I} = \lambda[[v/x]p_i, [v/x]g_i.[v/x]e_i]^{i\in I}$$

$$[v/x](\texttt{let}\ p = e) = \texttt{let}\ [v/x]p = [v/x]e \qquad [v/x]Fun(e) = \begin{cases} Fun_v([v/x]e) & \text{si } x = ego \\ Fun([v/x]e) & \text{sinon} \end{cases}$$

Nous terminons alors cette sous-section par les règles de déduction exprimant la sémantique du filtrage dans $\mathcal{F}unc_2$:

$$\frac{}{v/\_ \Rightarrow []} \qquad \frac{}{v/x \Rightarrow [v/x]} \qquad \frac{v \in \mathbb{A} \cup \mathbb{C} \cup \mathcal{F} \cup \mathcal{F}_c}{v/v \Rightarrow []} \qquad \frac{v_1 \in \mathbb{A} \cup \mathbb{C} \cup \mathcal{F} \cup \mathcal{F}_c \quad v_1 \neq v_2}{v_1/v_2 \Rightarrow \mathit{failed}}$$

$$\frac{\forall k\ \forall i\ \forall v_i \ \ v \neq C(v_1, \ldots, v_k)}{v/C(p_1, \ldots, p_n) \Rightarrow \mathit{failed}} \qquad \frac{\forall i \ \ v_i/p_i \Rightarrow \sigma_i}{C(v_1, \ldots, v_n)/C(p_1, \ldots, p_n) \Rightarrow \sigma_1 \oplus \cdots \oplus \sigma_n}$$

Potentiel

On définit, alors, le calcul du potentiel d'un acteur par le calcul du potentiel de l'expression de corps :

$$\textsc{Potentiel:}\ \frac{\mathcal{P}(\bot, \{\}, e) = V,\ E,\ \mathcal{E}}{\mathcal{P}_e(e) = E}$$

$$\textsc{Val:}\ \frac{v \in \mathbb{A} \cup \mathbb{C} \cup \mathcal{F}_c \cup \{nop,\ nil\} \cup \{\bot\}}{\mathcal{P}(\mathcal{E}, \mathcal{F}, v) = \bot,\ \{\},\ \mathcal{E}} \qquad \textsc{Var1:}\ \frac{x \in dom(\mathcal{E})}{\mathcal{P}(\mathcal{E}, \mathcal{F}, x) = \mathcal{P}(\mathcal{E}, \mathcal{F}, \mathcal{E}(x))}$$

$$\textsc{Var2:}\ \frac{x \notin dom(\mathcal{E})}{\mathcal{P}(\mathcal{E}, \mathcal{F}, x) = \bot,\ \{\},\ \mathcal{E}} \qquad \textsc{Const:}\ \frac{C \in \{cons,\ tuple2, \ldots\} \quad \mathcal{P}(\mathcal{E}, \mathcal{F}, e_i) = V_i,\ E_i,\ \mathcal{E}_i}{\mathcal{P}(\mathcal{E}, \mathcal{F}, C(\vec{e})) = \bigcup_i V_i,\ \bigcup_i E_i,\ \bigcup_i \mathcal{E}_i}$$

$$\textsc{Seq:}\ \frac{\mathcal{P}(\mathcal{E}, \mathcal{F}, e_1) = V_1,\ E_1,\ \mathcal{E}_1 \quad \mathcal{P}(\mathcal{E}_1, \mathcal{F}, e_2) = V_2,\ E_2,\ \mathcal{E}_2}{\mathcal{P}(\mathcal{E}, \mathcal{F}, e_1; e_2) = V_2,\ E_1 \cup E_2,\ \mathcal{E}_2}$$

$$\textsc{Fun1:}\ \frac{s \in dom(\mathcal{F})}{\mathcal{P}(\mathcal{E}, \mathcal{F}, Fun(s)) = \bot,\ \{\},\ \mathcal{E}} \qquad \textsc{Fun2:}\ \frac{s \notin dom(\mathcal{F})}{\mathcal{P}(\mathcal{E}, \mathcal{F}, Fun(s)) = Fun(s),\ \{\},\ \mathcal{E}}$$

$$\textsc{Abs:}\ \frac{}{\mathcal{P}(\mathcal{E}, \mathcal{F}, \lambda[p_i, g_i.e_i]^{i \in I}) = \lambda[p_i, g_i.e_i]^{i \in I},\ \{\},\ \mathcal{E}}$$

$$\textsc{Rcv:}\ \frac{\vec{x_i} = \mathcal{FV}(p_i) \qquad \mathcal{P}(\mathcal{E}{+}\{\vec{x_i} \mapsto \overrightarrow{\top}\}, \mathcal{F}, e_i) = V_i,\ E_i,\ \mathcal{E}_i}{\mathcal{P}(\mathcal{E}, \mathcal{F}, receive(\lambda[p_i, g_i.e_i]^{i \in I})) = \bigcup_i V_i,\ \bigcup_i (label(p_i) \cup E_i),\ \bigcup_i \mathcal{E}_i}$$

$$\textsc{App:}\ \frac{\mathcal{P}(\mathcal{E}, \mathcal{F}, e) = \bigcup_j V_j, E, \mathcal{E}_1 \quad \mathcal{P}(\mathcal{E}, \mathcal{F}, e') = \bigcup_k V'_k, E', \mathcal{E}_2 \quad App(\mathcal{E}_1 \cup \mathcal{E}_2, \mathcal{F}, V_k, V'_j) = V^j_k, E^j_k, \mathcal{E}^j_k}{\mathcal{P}(\mathcal{E}, \mathcal{F}, e\ e') = \bigcup_{j,k} V^j_k,\ \bigcup_{j,k} E^j_k \cup E \cup E',\ \bigcup_{k,j} \mathcal{E}^j_k}$$

$$\textsc{Let:}\ \frac{\vec{x} = \mathcal{FV}(p) \qquad \mathcal{P}(\mathcal{E}, \mathcal{F}, e) = V,\ E,\ \mathcal{E}_1}{\mathcal{P}(\mathcal{E}, \mathcal{F}, \texttt{let}\ p = e) = V,\ E,\ \mathcal{E}_1{+}\{\vec{x} \mapsto \vec{V}\}}$$

$$\frac{\mathcal{P}(\mathcal{E}, \mathcal{F} \cup \{s\}, \mathcal{E}_f(s)\ V) = V_1,\ E,\ \mathcal{E}_1}{App(\mathcal{E}, \mathcal{F}, Fun(s), V) = V_1,\ E,\ \mathcal{E}} \qquad \frac{}{App(\mathcal{E}, \mathcal{F}, \bot, V') = \bot,\ \{\}}$$

$$\frac{\vec{x_i} = \mathcal{FV}(p_i) \quad \mathcal{P}(\mathcal{E}{+}\{\vec{x_i} \mapsto \vec{V}\}, \mathcal{F}, e_i) = V_i,\ E_i,\ \mathcal{E}_i}{App(\mathcal{E}, \mathcal{F}, \lambda[p_i, g_i.e_i]^{i \in I}, V) = \bigcup_i V_i,\ \bigcup_i E_i,\ \bigcup_i \mathcal{E}_i} \qquad \frac{}{App(\mathcal{E}, \mathcal{F}, \top, V') = \top,\ \mathbb{M}}$$

Réduction

Enfin, la section rappellant la sémantique de μErlang se termine par toutes les règles de déduction exprimant sa sémantique. En commençant par la sémantique de la boîte aux lettres et celle du filtrage avec garde :

$$\textsc{Bal1}: \frac{\exists j \quad (\forall i < j \;\; m_i/p_1, g_1 \Rightarrow \mathit{failed}) \quad m_j/p_1, g_1 \Rightarrow \sigma}{\lambda[p_i, g_i.e_i]^{i\in 1..n} \prec (m_i)^{i\in J} \Longrightarrow (m_i)^{i\in J\setminus\{j\}},\; e_1,\; \sigma}$$

$$\textsc{Bal2}: \frac{(\forall i \in J \;\; m_i/p_1, g_1 \Rightarrow \mathit{failed}) \quad \lambda[p_i, g_i.e_i]^{i\in 2..n} \prec (m_i)^{i\in J} \Longrightarrow q,\; e,\; \sigma}{\lambda[p_i, g_i.e_i]^{i\in 1..n} \prec (m_i)^{i\in J} \Longrightarrow q,\; e,\; \sigma}$$

$$\frac{v/p \Rightarrow \mathit{failed}}{v/p, g \Rightarrow \mathit{failed}} \qquad \frac{v/p \Rightarrow \sigma \quad \sigma(g) \longrightarrow_e \mathit{false}}{v/p, g \Rightarrow \mathit{failed}} \qquad \frac{v/p \Rightarrow \sigma \quad \sigma(g) \longrightarrow_e \mathit{true}}{v/p, g \Rightarrow \sigma}$$

$$\textsc{Cong}: \frac{w_1 \equiv w_1' \quad w_1' \longrightarrow w_2' \quad w_2' \equiv w_2}{w_1 \longrightarrow w_2} \qquad \textsc{Par}: \frac{w_1 \longrightarrow w_2}{w||w_1 \longrightarrow w||w_2}$$

$$\textsc{Res}: \frac{w_1 \longrightarrow w_2}{\nu a.w_1 \longrightarrow \nu a.w_2} \qquad \textsc{Exp}: \frac{e_1 \longrightarrow_e e_2}{\alpha \triangleright e_1 \longrightarrow \alpha \triangleright e_2} \qquad \textsc{ExpE}: \frac{e \longrightarrow_e \mathbf{Err}}{\alpha \triangleright e \longrightarrow \mathbf{Err}}$$

$$\textsc{Rcv}: \frac{m \in \mathcal{P}}{\langle a \,|\, q\rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m \longrightarrow \langle a \,|\, q :: m\rangle_{\mathcal{P}} \triangleright e} \qquad \textsc{RcvE}: \frac{m \notin \mathcal{P}}{\langle a \,|\, q\rangle_{\mathcal{P}} \triangleright e \parallel a \triangleleft m \longrightarrow \mathbf{Err}}$$

$$\textsc{Send}: \frac{}{\alpha \triangleright \mathrm{C}[\mathit{send}(m, a)] \longrightarrow \alpha \triangleright \mathrm{C}[\mathit{nop}] \parallel a \triangleleft m} \qquad \textsc{SendE}: \frac{v_2 \notin \mathbb{A}}{\alpha \triangleright \mathit{send}\ (v_1, v_2) \longrightarrow \mathbf{Err}}$$

$$\textsc{New}: \frac{a \notin \mathcal{FN}(\alpha \triangleright \mathrm{C}[\mathit{new}])}{\alpha \triangleright \mathrm{C}[\mathit{new}] \longrightarrow \nu a.(\alpha \triangleright \mathrm{C}[a])}$$

$$\textsc{Init}: \frac{}{\alpha \triangleright \mathrm{C}[\mathit{init}(a, v)] \longrightarrow \alpha \triangleright \mathrm{C}[\mathit{nop}] \parallel \langle a \,|\, \varnothing\rangle_{\mathcal{P}(v\ a)} \triangleright v\ a}$$

$$\textsc{InitE}: \frac{v_1 \notin \mathbb{A}}{\alpha \triangleright \mathit{init}\ (v_1, v_2) \longrightarrow \mathbf{Err}} \qquad \textsc{Rec}: \frac{\mathcal{R} \prec q \Longrightarrow q',\; e,\; \sigma}{\langle a \,|\, q\rangle \triangleright \mathrm{C}[\mathit{receive}(\mathcal{R})] \longrightarrow \langle a \,|\, q'\rangle \triangleright \sigma(\mathrm{C}[e])}$$

$$\textsc{VarE}: \frac{}{\mathrm{C}[x] \longrightarrow_e \mathbf{Err}} \qquad \textsc{AppE1}: \frac{v \notin \mathcal{F} \cup \mathcal{F}_c}{\mathrm{C}[v\ e] \longrightarrow_e \mathbf{Err}} \qquad \textsc{App}: \frac{v/p_1, g_1 \Rightarrow \sigma}{\mathrm{C}[\lambda[p_i, g_i.e_i]^{i\in 1..n}\ v] \longrightarrow_e \sigma(\mathrm{C}[e_1])}$$

$$\textsc{AppE2}: \frac{}{\mathrm{C}[\lambda[]\ v] \longrightarrow_e \mathbf{Err}} \qquad \textsc{AppF}: \frac{v/p_1, g_1 \Rightarrow \mathit{failed}}{\mathrm{C}[\lambda[p_i, g_1.e_i]^{i\in 1..n}\ v] \longrightarrow_e \mathrm{C}[\lambda[p_i, g_i.e_i]^{i\in 2..n}\ v]}$$

$$\textsc{Seq}: \frac{}{\mathrm{C}[v\,; e] \longrightarrow_e \mathrm{C}[e]} \qquad \textsc{Let}: \frac{v/p \Rightarrow \sigma}{\mathrm{C}[\mathtt{let}\ p = v] \longrightarrow_e \sigma(\mathrm{C}[v])} \qquad \textsc{LetE}: \frac{v/p \Rightarrow \mathit{failed}}{\mathrm{C}[\mathtt{let}\ p = v] \longrightarrow_e \mathbf{Err}}$$

$$\textsc{FunEgo}: \frac{f \in \mathit{dom}(\mathcal{E})}{\mathrm{C}[\mathit{Fun}_v(f)] \longrightarrow_e \mathrm{C}[[v/\mathit{ego}]\mathcal{E}(f)]} \qquad \textsc{Fun}: \frac{f \in \mathit{dom}(\mathcal{E})}{\mathrm{C}[\mathit{Fun}(f)] \longrightarrow_e \mathrm{C}[\mathcal{E}(f)]}$$

$$\textsc{FunE1}: \frac{v \notin \mathit{dom}(\mathcal{E})}{\mathrm{C}[\mathit{Fun}_{v'}(v)] \longrightarrow_e \mathbf{Err}} \qquad \textsc{FunE2}: \frac{v \notin \mathit{dom}(\mathcal{E})}{\mathrm{C}[\mathit{Fun}(v)] \longrightarrow_e \mathbf{Err}}$$

Bibliographie

[AC94] Martín Abadi and Luca Cardelli. « A Theory of Primitive Objects — Untyped and First-Order Systems ». In M. Hagiya and John C. Mitchell, editors, *proceedings of the International Symposium on Theoretical Aspects of Computer Software TACS '94, Sendai, Japan, April 19–22, 1994*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320, New York, NY, USA, 1994. Springer-Verlag.

[ACM99] ACM. *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, January 20–22, 1999.

[ACM01] ACM. *Conference Record of POPL'01: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, London, United Kingdom, January 17–19, 2001.

[AG97a] Martín Abadi and Andrew D. Gordon. « A Calculus for Cryptographic Protocols: The Spi Calculus ». In *Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zürich, April 1997*. ACM Press, 1997. Long version as Technical Report 414, University of Cambridge.

[AG97b] Matin Abadi and Andrew D. Gordon. « Reasoning about Cryptographic Protocols in the Spi Calculus ». In Mazurkiewicz and Winkowski [MW97b], pages 59–73.

[Agh84] Gul Agha. « Semantic Considerations in the Actor Paradigm of Concurrent Computation ». In Steve Brookes, A. Roscoe, and G. Vowinkel, editors, *Proceedings of the NSF/SERC Seminar on Concurrency*, pages 151–179, Berlin, DE, 1984. Springer-Verlag.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. The MIT Press, Cambridge, MA, USA, 1986.

[Agh90] Gul Agha. The Structure and Semantics of Actor Languages. In Jacobus W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 1–59. Springer-Verlag, Berlin, DE, 1990.

[AH87] Gul Agha and Carl E. Hewitt. « Concurrent Programming Using Actors ». In Yonezawa and Tokoro [YT87], pages 37–53.

[AH92] Gul Agha and C. Houck. « HAL: A High-level Actor Language and Its Distributed Implementation ». In *21st InternationalConference on Parallel Processing (ICPP'92), vol II*, pages 158–165, August 1992.

[AKS+88] Gul Agha, Wooyoung Kim, Hans-Jörg Schek, Vineet Singh, Chris Tomlinson, and R. Will. « Rosette: An Object-Based Concurrent Systems Architecture ». Technical Report ACA-ST-254-88(Q), Microelectronics and Computer Consortium MCC/ACA, August 1988.

[AM91] Alexander S. Aiken and Brian R. Murphy. « Static Type Inference in a Dynamically Typed Language ». In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Orlando, Florida, January 21–23, 1991. ACM SIGACT-SIGPLAN, ACM Press.

[Ama97] Roberto Amadio. « Failure and Process Mobility ». In D. Garlan and Daniel Le Metayer, editors, *Proceedings of COORDINATION '97*, volume 1282 of *Lecture Notes in Computer Science*, pages 374–391. Springer-Verlag, 1997.

[AMST92] Gul Agha, I. A. Mason, S. F. Smith, and Carolyn L. Talcott. « Towards a Theory of Actor Computation ». In W. Rance Cleaveland, editor, *Proceedings of CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 565–579. Springer-Verlag, 1992.

[AMST97] Gul Agha, I. A. Mason, S. F. Smith, and Carolyn L. Talcott. « A Foundation for Actor Computation ». *Journal of Functional Programming*, 7:1–72, 1997.

[Arm96] Joe Armstrong. « Erlang - A Survey of the Language and its Industrial Applications ». In *INAP'96. The 9'th Exhibitions and Symposium on Industrial Applications of Prolog. Hino, Tokyo, Japan*, October 1996.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, second edition, 1996. ISBN 0-13-285792-8.

[AW93] Alexander S. Aiken and Edward L. Wimmers. « Type Inclusion Constraints and Type Inference ». In *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 31–41. ACM Press, June 1993.

[AWL94] Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. « Soft Typing with Conditional Types ». In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 163–173, Portland, Oregon, January 1994. ACM, ACM Press.

[BB92] Gérard Berry and Gérard Boudol. « The chemical abstract machine ». *Theoretical Computer Science*, 96:217–248, 1992.

[Ber00] Gérard Berry. « *The Foundations of Esterel* ». Foundations of Computing. MIT Press, 2000.

[Bot82] Christine Botella. « *Sémantique dénotationnelle du langage Plasma : Définition et Implémentation* ». Thèse de doctorat, Université Paul Sabatier, Toulouse, France, 1982.

[Bou92] Gérard Boudol. « Asynchrony and the π-Calculus(Note) ». Rapport de Recherche 1702, INRIA-Sophia Antipolis, May 1992.

[Bou97a] Gérard Boudol. « The π-Calculus in Direct Style ». In *Proceedings of POPL '97*, pages 228–241. ACM, January 1997.

[Bou97b] Gérard Boudol. « Typing the Use of Resources in a Concurrent Calculus ». In R. K. Shyamasundar and K. Ueda, editors, *Proceedings of ASIAN '97*, volume 1345 of *Lecture Notes in Computer Science*, pages 239–253. Springer-Verlag, 1997.

[BR98] Staffan Blau and Jan Rooth. « AXD 301 - A new generation ATM switching system ». *Ericsson Review*, (01), 1998.

[Bri88] Jean-Pierre Briot. « From Objects to Actors: Study of a Limited Symbiosis ». Research Report RXF-LITP 88-58, Equipe Mixte LITP — RankXeroxFrance, Paris, FR, 1988.

[Bri89] Jean-Pierre Briot. « Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment ». In *Proceedings of ECOOP '89*, pages 109–129, Cambridge, MA, USA, July 1989. Cambridge University Press.

[BV99] Jonas Barklund and Robert Virding. « ERLANG *4.7.3 Reference Manual* », February 1999. Draft 0.7.

[Car85] Luca Cardelli. « Semantics of Multiple Inheritance ». *Information and Computation*, 76, 1985.

[Car99] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer-Verlag, 1999.

[CBR96] D. Caromel, F. Belloncle, and Y. Roudier. The C++// System. pages 257–296. The MIT Press, 1996. ISBN 0-262-73118-5.

[CC77] Patrick Cousot and Radhia Cousot. « Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints ». In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 17–19, 1977. ACM SIGACT-SIGPLAN.

[CC95] P. Cousot and R. Cousot. « Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation ». In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, 25–28 June 1995. ACM Press, New York, NY.

[CF91] R. Cartwright and M. Fagan. « Soft typing ». In *PLDI91*, volume 26, pages 278–292, Toronto, Ontario, Canada, June 1991.

[CG99] Luca Cardelli and Andrew D. Gordon. « Types for mobile ambients ». In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* [ACM99], pages 79–92.

[CGG99] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. « Mobility Types for Mobile Ambients ». In Jiří Wiederman, Peter van Emde Boas, and Mogens Nielsen, editors, *Proceedings of ICALP '99*, volume 1644 of *Lecture Notes in Computer Science*, pages 230–239. Springer-Verlag, July 1999.

[Che00] David Chemouil. « Architecture répartie pour un langage fonctionnel d'acteurs ». DEA IFP, Institut National Polytechnique de Toulouse, September 2000.

[CLF99] Sylvain Conchon and Fabrice Le Fessant. « JoCaml: mobile agents for Objective-Caml ». In *Proceedings of the Joint Symposium ASA/MA '99. First International Symposium on Agent Systems and Applications (ASA '99). Third International Symposium on Mobile Agents (MA '99)*. IEEE-Computer Society, October 1999.

[Cli81] W. D. Clinger. « *Foundations of Actor Semantics* ». PhD thesis, MathematicsDepartment, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[CMP00] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano, editors. *Développement d'applications avec Objective Caml*. Éditions O'Reilly, Paris, FR, April 2000.

[Col97] Jean-Louis Colaço. « *Analyses statiques par typage de langages d'Acteurs* ». Thèse de doctorat, Institut National Polytechnique de Toulouse, October 1997.

[Col98] Matthias Colin. « Intégration des Typages Fonctionnel et Concurrent d'un Langage Fonctionnel d'Acteurs ». DEA IFP, Institut National Polytechnique de Toulouse, September 1998.

[CPDS99] Jean-Louis Colaço, Marc Pantel, Fabien Dagnat, and Patrick Sallé. « Static Safety Analysis for Non-uniform Service Availability in Actors ». In *Proceedings of FMOODS '99*, February 1999.

[CPS96] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. « CAP: An Actor Dedicated Process Calculus ». In *Proceedings of Proof Theory of Concurrent Object-Oriented Programming*, May 1996.

[CR95] D. Caromel and Y. Roudier. « Reactive programming in Eiffel// ». In Jean-Pierre Briot, J.-M Geib, and Akinori Yonezawa, editors, *Object-based parallel and distributed computation: France-Japan Workshop, OBPDC '95, Tokyo, Japan, June 21–23, 1995: selected papers*, volume 1107 of *Lecture Notes in Computer Science*, pages 125–147. Springer-Verlag, June 1995.

[Cri95] Régis Cridlig. « Semantic Analysis of Shared-Memory Concurrent Languages using Abstract Model-Checking ». In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 214–225, La Jolla, California, 21-23 June 1995.

[Cri96] R. Cridlig. « Semantic Analysis of Concurrent ML by Abstract Model-Checking ». In *Proceedings of the LOMAPS Workshop*, 1996.

[CV98] D. Caromel and J. Vayssiere. « A Java Framework for Seamless Sequential, Multi-threaded, and Distributed Programming ». pages 141–150, February 1998.

[CW85] L. Cardelli and P. Wegner. « On Understanding Types, Data Abstraction, and Polymorphism ». *ACM Comp.Surv*, 17, 1985.

[Dag97] Fabien Dagnat. « Conception d'un Langage Fonctionnel d'Acteur et Réalisation de son Compilateur ». DEA IFP, Institut National Polytechnique de Toulouse, September 1997.

[Dam96] Mads Dam. « Model Checking Mobile Processes ». *Journal of Information and Computation*, 129(1):35–51, 1996. Also available as Research Report R94:01, SICS, Sweden. A summary appeared in *Proceedings of CONCUR '93*, LNCS 715.

[DG94] J. Darlington and Yike K. Guo. « Formalising Actors in Linear Logic ». In *Proceedings of the International Conference on Object-Oriented Information Systems (OOIS'94)*, 1994.

[DPCS00] Fabien Dagnat, Marc Pantel, Matthias Colin, and Patrick Sallé. « Typing Concurrent Objects and Actors ». *L'Objet – Méthodes formelles pour les objets*, Volume 6(n° 1/2000):pages 83–106, May 2000.

[DPS98] Fabien Dagnat, Marc Pantel, and Patrick Sallé. « ML-Act, un langage fonctionnel d'Acteurs ». In *Actes des Journées Francophones des Langages Applicatifs*, February 1998.

[Dur81] J-L. Durieux. « *Sémantique des liaisons Nom-Valeur : Application à l'implémentation des Lambda-Langages* ». Thèse de doctorat, Université Paul Sabatier, Toulouse, France, 1981.

[DZ99] Silvano Dal-Zilio. « *Calcul bleu: types et objets* ». PhD thesis, Université de Nice - Sophia-Antipolis, 1999.

[DZ00] Silvano Dal-Zilio. « Mobile Processes: a Commented Bibliography ». In *In Proc. of MOVEP'2k - Summer School on Modelling and Verification of Parallel Processes*, 2000. to be published by Springer in the series Lecture Notes in Computer Science.

[DåF98] Mads Dam and Lars åke Fredlund. « On the Verification of Open Distributed Systems ». In *Proc. of the ACM Symposium on Applied Computing*, volume 28, pages 532–540. ACM, June 1998.

[DåFG98] Mads Dam, Lars åke Fredlund, and Dilian Gurov. « *Compositionality: The Significant Difference* », volume 1536 of *Lecture Notes in Computer Science*, Chapitre Toward Parametric Verification of Open Distributed Systems, pages 150–185. Springer-Verlag, 1998.

[FA97] Manuel Fähndrich and Alexander S. Aiken. « Program Analysis using Mixed Term and Set Constraints ». In P. van Hentenryck, editor, *Static analysis: 4th International Symposium, SAS'97, Paris, France, September 1997: proceedings*, volume 1302 of *Lecture Notes in Computer Science*, pages 114–?? Springer-Verlag, 1997.

[Fel87] Matthias Felleisen. « *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages* ». PhD thesis, Indiana University, 1987.

[FFKD87] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. « A Syntactic Theory of Sequential Control ». *Theoretical Computer Science*, 52:205–237, 1987.

[FG96] Cédric Fournet and Georges Gonthier. « The Reflexive Chemical Abstract Machine and the Join-Calculus ». In Steele [Ste96], pages 372–385.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. « A Calculus of Mobile Agents ». In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996.

[FGM+98] Gianluigi Ferrari, Stefania Gnesi, Ugo Montanari, Marco Pistore, and Gioia Ristori. « Verifying Mobile Processes in the HAL Environment ». In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of CAV '98*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998. Tool Poster.

[FLMR97] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. « Implicit Typing à la ML for the join-calculus ». In Mazurkiewicz and Winkowski [MW97b], pages 196–212.

[Fäh99] Manuel Fähndrich. « *BANE: A library for Scalable Constraint-Based Program Analysis* ». PhD thesis, University of California at Berkley, 1999.

[Gir72] Jean-Yves Girard. « *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur* ». Thèse de doctorat d'état, Université Paris VII, Paris, France, June 1972.

[GMM+78] M. Gordon, Robin Milner, L. Morris, M. Newey, and C. Wadsworth. « A Metalanguage for Interactive Proof in LCF ». In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, Tucson, Arizona, January 23–25, 1978. ACM SIGACT-SIGPLAN.

[Hal93] N. Halbwachs. « A tutorial of Lustre ». 1993.

[HBS73] Carl E. Hewitt, P. Bishop, and R. Steiger. « A Universal Modular ACTOR Formalism for Artificial Intelligence ». In *Proceedings of Int. Joint Conference on Artificial Intelligence*, 1973.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. « The Synchronous Data-flow Programming Language LUSTRE ». *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Hei92] Nevin Heintze. « *Set-Based Program Analysis* ». PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, October 1992.

[Hei93] Nevin Heintze. « Set Constraints in Program Analysis ». Technical Report, Carnegie-Mellon University, July 1993. (Unpublished).

[Hew77] Carl E. Hewitt. « Viewing Controll Structures as Patterns of Passing Messages ». *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. « Programming and Verifying Real-time Systems by Means of the Synchronous Data-flow Programming Language LUSTRE ». *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[Hoa85] C.A.R. Hoare. *Communications Sequential Processes*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. « Proving the Correctness of Reactive Systems Using Sized Types ». In Steele [Ste96], pages 410–423.

[HT91] Kohei Honda and Mario Tokoro. « An Object Calculus for Asynchronous Communication ». In Pierre America, editor, *Proceedings of ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, July 1991.

[Huc99] Frank Huch. « Verification of Erlang Programs using Abstract Interpretation and Model Checking ». *ACM SIGPLAN Notices*, 34(9):261–272, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

[KA92] Wooyoung Kim and Gul Agha. « Compilation of a Highly Parallel Actor-Based Language ». In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of 5th International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1992.

[Kim97] Wooyoung Kim. « *ThAL: An Actor System for Efficient and Scalable Concurrent Computing* ». PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1997.

[Kob97] Naoki Kobayashi. A Partially Deadlock-Free Typed Process Calculus. In *Proceedings of LICS '97*, pages 128–139. Computer Society Press, July 1997. Full version as as Technical Report 97-02, University of Tokyo.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. « Linearity and the π-Calculus ». In Steele [Ste96], pages 358–371.

[KY94a] Naoki Kobayashi and Akinori Yonezawa. « Asynchronous Communication Model Based on Linear Logic ». *Formal Aspects of Computing*, 1994.

[KY94b] Naoki Kobayashi and Akinori Yonezawa. « Type-theoretic Foundations for Concurrent Object-Oriented Programming ». In Norman Meyrowitz, editor, *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, volume 29 of *Sigplan Notices*, pages 31–45, Portland, Oregon, USA, October 1994. ACM.

[Kön99] Barbara König. « *Description and Verification of Mobile Processes with Graph Rewriting Techniques* ». PhD thesis, Technische Universität München, 1999.

[LBB91] Loïc Lescaudron, Jean-Pierre Briot, and Malik Bouabsa. « Prototyping Programming Environments for Object-Oriented Concurrent Languages: A

Smalltalk-Based Experience ». In *Proceedings of 5th Conference on the Technology of Object-Oriented Languages and Systems (Tools Usa'91)*, pages 449–462. Prentice Hall, 1991.

[Ler90] Xavier Leroy. « The ZINC experiment: an economical implementation of the ML language ». Technical report 117, INRIA, 1990.

[Ler92] Xavier Leroy. « *Typage polymorphe d'un langage algorithmique* ». Thèse de doctorat, Université Paris 7, 1992.

[LG88] John M. Lucassen and David K. Gifford. « Polymorphic Effect Systems ». In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, January 13–15, 1988. ACM SIGACT-SIGPLAN, ACM Press.

[Lin96] Anders Lindgren. « A Prototype of a Soft Type System for Erlang ». Master's thesis, Computer ScienceDepartment, Uppsala University, 1996.

[LTP86] W. R. Lalonde, D. A. Thomas, and J. R. Pugh. « Actors in a Smalltalk Multiprocessor: A Case for Limited Parallelism ». Technical Report SCS-TR-91, School of Computer Science, Carelton University, Ottawa, CA, 1986.

[Man87] C. R. Manning. « Acore: The Design of a Core Actor Language and its Compiler ». Revised master thesis, Computer ScienceDepartmentMassachusetts Institute of Technology, Cambridge, MA, USA, 1987.

[Mil80] Robin Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil93] R. Milner. « Action Calculi, or Syntactic Action Structures ». In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Proceedings of MFCS'93*, volume 711 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Mil96] Robin Milner. « Calculi for Interaction ». *Acta Informatica*, 33, 1996.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[MMS88] Annie Marcoux, C. Maurel, and Patrick Sallé. « A Language for Distributed Applications ». In *IEEE Workshop on Future Trends of Distributed Systems in the 90's*, 1988.

[MPS86] D. MacQueen, Gordon Plotkin, and Ravi Sethi. « An Ideal Model for Recursive Polymorphic Types ». *Information and Control*, 71(1/2):95–130, 1986.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. « A Calculus of Mobile Processes, Part I/II ». *Journal of Information and Computation*, 100:1–77, September 1992.

[MW97a] Simon Marlow and Philip Wadler. « A Practical Subtyping System for Erlang ». In *Proceedings of ICFP '97*, volume 32(8) of *Sigplan Notices*. ACM, August 1997.

[MW97b] Antoni Mazurkiewicz and Józef Winkowski, editors. *CONCUR '97: Concurrency Theory (8th International Conference, Warsaw, Poland, July 1997)*, volume 1243 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Nim99] Abdelkrim Nimour. « *Systèmes de typage non-uniforme pour objets repartis* ». PhD thesis, ENST, Paris, 1999.

[NN93] Flemming Nielson and Hanne Riis Nielson. « From CML to Process Algebras ». In Eike Best, editor, *Proceedings of CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[NN99] Elie Najm and Abdelkrim Nimour. « Explicit Behavioral Typing for Object Interfaces ». In Hans Hüttel, Josva Kleist, Uwe Nestmann, and António Ravara, editors, *Semantics of Objects As Processes SOAP '99 (Lisbon, Portugal, June 15, 1999)*, volume NS-99-2 of *BRICS Notes Series*, pages 47–64. BRICS, May 1999.

[NNA96] Flemming Nielson, Hanne Riis Nielson, and Torben Amtoft. « Polymorphic Subtyping for Effect Analysis: the Integration, the Semantics, the Algorithm ». Technical Report PB-501, PB-502, PB-503, DAIMI, University of Aarhus, Denmark, April 1996.

[NNS99a] Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. « Guaranteeing Liveness in an Object Calculus Through Behavioral Typing ». In *Proc. FORTE/PSTV Beijing, China*. Kluwer, October 1999.

[NNS99b] Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. « Infinite Types for Distributed Objects Interfaces ». In *Proceedings of third IFIP Conference on Formal Methods for Open Object-based Distributed Systems*. International Federation for Information Processing, Kluwer, February 1999.

[Pan94] Marc Pantel. « *Représentation et Transformation : Un modèle de la réutilisabilité dans les langages fonctionnels à objets* ». Thèse de doctorat, Institut National Polytechnique de Toulouse, February 1994.

[Par01] Lars Pareto. « *Types for Crash Prevention* ». PhD thesis, Göteborg University, jan 2001.

[Pes99] François Pessaux. « *Détection statique d'exception non rattrappées en Objective Caml* ». PhD thesis, Université Paris 6, December 1999.

[Plo77] Gordon Plotkin. « LCF as a Programming Language ». *Theoretical Computer Science*, 5, 1977.

[Pom80] C. Pomian. « *Contribution à la définition de l'implémentation d'un interprète du langage Plasma* ». Thèse de doctorat, Université Paul Sabatier, Toulouse, France, 1980.

[Pot98] François Pottier. « *Synthèse de types en présence de sous-typage: de la théorie à la pratique* ». PhD thesis, Université Paris 7, July 1998.

[PRT93] Benjamin C. Pierce, Didier Rémy, and David N. Turner. « A Typed Higher-Order Programming Language Based on the π-Calculus ». In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993.

[PS96] Benjamin C. Pierce and Davide Sangiorgi. « Typing and Subtyping for Mobile Processes ». *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[PT95] Benjamin C. Pierce and David N. Turner. « Concurrent Objects in a Process Calculus ». In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings of TPPP '94*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer-Verlag, 1995.

[PT97] Benjamin C. Pierce and David N. Turner. « Pict: A Programming Language Based on the π-calculus ». Technical Report, Computer Science Department, Indiana University, 1997.

[Pun00] Franz Puntigam. « *Concurrent Object-Oriented Programming with Process Types* ». Habilitationsschrift, Der Andere Verlag, Osnabrück, Germany, 2000.

[PV98] Joachim Parrow and Björn Victor. « The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes ». In *Proceedings of LICS '98*, pages 176–185. IEEE, Computer Society Press, July 1998.

[Rav00] António Ravara. « *Typing Non-Uniform Concurrent Objects* ». PhD thesis, IST, Universidade Técnica de Lisboa, 2000.

[Rep91] J. H. Reppy. « CML: a Higher-order Concurrent Language ». In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 293–305, Toronto, Ontario, Canada, June 1991.

[RH98] James Riely and Matthew Hennessy. « A Typed Language for Distributed Mobile Processes ». In *Proceedings of POPL '98*. ACM, January 1998.

[RH99] James Riely and Matthew Hennessy. « Trust and Partial Typing in Open Systems of Mobile Agents ». In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* [ACM99], pages 93–104. Full version as CogSci Report 4/98, University of Sussex, Brighton.

[RV98] Didier Rémy and Jérôme Vouillon. « Objective ML: An Effective Object-oriented Extension to ML ». *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. « Automatic Testing of Reactive Systems ». In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[Rém94] Didier Rémy. Type Inference for Records in a Natural Extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, April 1994.

[San92] Davide Sangiorgi. « *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms* ». PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[Sch95] Susanne Schacht. Proving Properties of Actor Programs Using Temporal Logic. In Gul Agha and Fiorella De Cindio, editors, *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency at*

the 16th InternationalConference on Application and Theory of Petri Nets. Torino, Italy, 26–30 Jun 1995. Università degli Studi di Milano, Milan, IT, June 1995.

[Sch96] Susanne Schacht. « Formal Reasoning about Actor Programs using Temporal Logic ». In *Proceedings of FOOL 3*, 1996.

[Sew98a] Peter Sewell. « Global/Local Subtyping and Capability Inference for a Distributed π-calculus ». In *Proceedings of ICALP '98: International Colloquium on Automata, Languages and Programming (Aalborg). LNCS 1443*, pages 695–706. Springer-Verlag, July 1998. A longer version is available [**?**].

[Sew98b] Peter Sewell. « Global/Local Subtyping and Capability Inference for a Distributed π-calculus ». In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of ICALP '98*, volume 1443 of *Lecture Notes in Computer Science*, pages 695–706. Springer-Verlag, July 1998. Full version as Technical Report 435, Computer Laboratory, University of Cambridge.

[Sew00] Peter Sewell. « Applied π – A Brief Tutorial ». Technical Report 498, Computer Laboratory, University of Cambridge, August 2000.

[Sew01] Peter Sewell. « Modules, Abstract Types, and Distributed Versioning ». In *Conference Record of POPL'01: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* [ACM01], pages 236–247.

[Shi01] Mark Shields. « *Static Types for Dynamic Documents* ». PhD thesis, Oregon Graduate Institute of Science and Technology, feb 2001.

[SM01] Mark Shields and Erik Meijer. « Type-indexed Rows ». In *Conference Record of POPL'01: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* [ACM01], pages 261 – 275.

[SS98] D. Schmidt and B. Steffen. « Program Analysis as Model Checking of Abstract Interpretations ». *Lecture Notes in Computer Science*, 1503:351–??, 1998.

[Ste96] Guy Steele, Jr, editor. *23rd Principles of Programming Languages (St. Petersburg Beach, Florida)*. ACM, January 1996.

[The82] Daniel G. Theriault. « A Primer for the Act-1 Language ». AI Memo 672, AI Lab., Massachusetts Institute of Technology, Cambridge, MA, USA, 1982.

[The83] Daniel G. Theriault. « Issues in the Design and Implementation of Act2 ». Technical Report 728, AI Lab., Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[Thi94] Xavier Thirioux. « Inférence de type par résolution de contraintes ensemblistes pour un Langage Fonctionnel à Objets (FOL) ». DEA IFP, Institut National Polytechnique de Toulouse, September 1994.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. « The Type and Effect Discipline ». *Information and Computation*, 111(2):245–296, June 1994.

[TS89] Chris Tomlinson and Vineet Singh. « Inheritance and Synchronization with Enabled Sets ». In Norman Meyrowitz, editor, *Proceedings of OOPSLA '89*, volume 24 of *Sigplan Notices*, pages 103–112. ACM, October 1989.

[Vas94] Vasco T. Vasconcelos. « *A Process-calculus Approach to Typed Concurrent Objects* ». PhD thesis, Keio University, 1994.

[Vas99] Vasco T. Vasconcelos. « Processes, Functions, Datatypes ». *Theory and Practice of Object Sytems*, 5(2):97–110, 1999.

[VC99] Jan Vitek and Giuseppe Castagna. « Seal: A framework for secure mobile computations ». In *Internet Programming Languages*, 1999.

[Ven97] Arnaud Venet. « Abstract Interpretation of the π-Calculus ». In Mads Dam, editor, *Proceedings of the Fifth LOMAPS Workshop*, volume 1192 of *Lecture Notes in Computer Science*, pages 51–75. Springer-Verlag, 1997.

[Ven98] A. Venet. « Automatic Determination of Communication Topologies in Mobile Systems ». *Lecture Notes in Computer Science*, 1503:152–??, 1998.

[Vic98] Björn Victor. « *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes* ». PhD thesis, Dept. of Computer Systems, Uppsala University, Sweden, June 1998.

[VM94] Björn Victor and Faron Moller. « The Mobility Workbench — A Tool for the π-Calculus ». In David Dill, editor, *Proceedings of CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[VT93] Vasco Thudichum Vasconcelos and Mario Tokoro. « A Typing System for a Calculus of Objects ». In Shojiro Nishio and Akinori Yonezawa, editors, *Object technologies for advanced software: first JSSST international symposium, Kanazawa, Japan, November 4–6, 1993: proceedings*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474, New York, NY, USA, 1993. Springer-Verlag.

[WA96] Mike Williams and Joe Armstrong. « *Program Development Using Erlang - Programming Rules and Conventions* ». ERICSSON, mar 1996. Doc. EPK/NP 95:035.

[Wan87] Mitchell Wand. « Complete Type Inference for Simple Objects ». In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, (LICS'87), Ithaca, New York*, pages 37–44. IEEE Computer Society Press, June 1987. Corrigendum in LICS'88, p. 132.

[WB99] Manfred Widera and Christoph Beierle. « Combining Strict and Soft Typing in Functional Programming ». In Wilhelm Schäfer Kurt Beiersdörfer, Gregor Engels, editor, *Informatik '99 - Informatik überwindet Grenzen, 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 5.-9. October 1999*, pages 350–359. Springer-Verlag, oct 199.

[Wri92] Andrew K. Wright. « Typing References by Effect Inference ». In B. Krieg-Bruckner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, February 26–28, 1992: proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491, New York, NY, USA, 1992. Springer-Verlag.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. « Object-Oriented Concurrent Programming in ABCL/1 ». In Norman Meyrowitz,

editor, *Proceedings of OOPSLA '86*, volume 21 of *Sigplan Notices*, pages 258–268. ACM, November 1986.

[YH99] Nobuko Yoshida and Matthew Hennessy. « Subtyping and Locality in Distributed Higher Order Processes ». In Jos C. M. Baeten and Sjouke Mauw, editors, *Proceedings of CONCUR '99*, volume 1664 of *Lecture Notes in Computer Science*, pages 557–572. Springer-Verlag, August 1999.

[Yon90] Akinori Yonezawa, editor. *ABCL: An object-oriented concurrent system*. Computer Systems Series. The MIT Press, Cambridge, MA, USA, 1990.

[Yos96] Nobuko Yoshida. « Graph Types for Monadic Mobile Processes ». In V. Chandru and V. Vinay, editors, *Proceedings of FSTTCS '96*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer-Verlag, 1996. Full version as Technical Report ECS-LFCS-96-350, University of Edinburgh.

[YSTH87] Akinori Yonezawa, Etsuya Shibayama, T. Takada, and Yasuaki Honda. Modelling and Programming in ABCL/1. In Yonezawa and Tokoro [YT87], pages 55–89.

[YT87] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming*. Computer Systems Series. The MIT Press, Cambridge, MA, USA, 1987.

[Zim00] Pascal Zimmer. « Subtyping and Typing Algorithms for Mobile Ambients ». In Jerzy Tiuryn, editor, *Proceedings of FoSSaCS 2000*, volume 1784 of *Lecture Notes in Computer Science*, pages 375–390. Springer-Verlag, 2000.

Résumé

Dans les programmes acteurs ou objets concurrents, et plus généralement dans les logiciels clients/serveurs, certaines requêtes ne pourront pas être traitées par leur cible. Une telle requête est appelée message orphelin, elle peut être : soit un orphelin de sûreté (son destinataire ne pourra jamais la traiter), soit un orphelin de vivacité (son destinataire pourrait éventuellement la traiter, mais il n'atteindra pas l'état nécessaire à ce traitement).

Dans le cadre de l'équipe *Vestale* qui m'a accueilli, des systèmes de type ont été conçus pour détecter les orphelins de sûreté pour un calcul de processus modélisant les acteurs. Dans cette thèse, nous adaptons ces analyses statiques pour détecter certains problèmes de communication au sein de *vrais* langages de programmation. Le premier, ML-Act, une extension de ML avec des primitives du modèle d'acteurs conçue au sein de l'équipe *Vestale*. Le second, Erlang, est un langage fonctionnel concurrent et réparti conçu par ERICSSON pour construire des applications de télécommunication.

Pour détecter les orphelins, nos systèmes sont basés sur un processus d'inférence qui calcule le type des différentes entités du programme. Les types qui approximent les acteurs sont inspirés des types utilisés usuellement pour les enregistrements ou les objets. Les systèmes de type sont très sophistiqués, ils contiennent une notion de sous-typage et reposent sur des algorithmes d'inférence qui collectent des contraintes à partir du code source, puis les résolvent. Leur correction est démontrée en utilisant une sémantique opérationnelle du modèle d'acteurs reposant sur un entrelacement de deux réductions (une sur les expressions fonctionnelles et une sur les acteurs). Le formalisme modélisant les acteurs, appelé configuration, est général et commun aux deux langages (qui ne se distinguent donc que par le calcul fonctionnel).

En conclusion, nous faisons un bilan des évolutions des théories et techniques qui ont été nécessaires pour adapter des systèmes construits sur un calcul de processus à des langages de programmation complexes.

Mots-clés: Acteurs, Typage, Analyses statiques, Inférence, Erlang, Concurrence.