Intégration des typages fonctionnel et concurrent d'un langage fonctionnel d'acteurs

Matthias Colin, Marc Pantel, Fabien Dagnat, Patrick Sallé

LIMA/IRIT/INPT
2, rue Charles Camichel
31071 Toulouse CEDEX, France
Matthias.Colin@enseeiht.fr
Marc.Pantel@enseeiht.fr
www.enseeiht.fr/fr/recherche/info/Logiciel/vestale/vestale.html

Résumé

La validation statique de programmes écrits dans des langages d'acteurs impose l'intégration de mécanismes de typage fonctionnel et concurrent. L'approche classique, proposée par Nielson et Nielson, repose sur un calcul d'effets conjoint au typage, suivi d'une analyse statique de l'effet (un terme d'une algèbre de processus) obtenu. Pour éviter les restrictions liées à l'unification des effets, nous proposons une approche intégrant en une seule phase les analyses statiques des éléments fonctionnels et concurrents des langages d'acteurs. L'exposé utilise une version allégée du langage d'acteurs ML-ACT et présente un système de types pour ce langage, dont les analyses sont basées à la fois sur les techniques de sous-typage et d'unification. Des exemples concrets illustrent l'algorithme de typage proposé.

1. Introduction.

Portée par la vague INTERNET, la programmation concurrente répartie connaît un essor sans précédent. Le modèle d'acteurs proposé par Hewitt [17] et développé par Agha [1], est l'un des premiers modèles de la programmation concurrente. Il a inspiré la plupart de ses successeurs mais reste l'un des plus simples à utiliser pour le programmeur. Il propose une structure à base d'agents (les acteurs) autonomes et réactifs qui encapsulent des données et des programmes. La coopération entre les différents acteurs d'une application suit un protocole (fixe) de communication asynchrone de type point à point. Un système peut donc être décrit (à un instant donné) comme un ensemble d'acteurs s'exécutant en parallèle et un ensemble de messages en transit. La création dynamique d'acteur et la communication des *adresses* par message rendent la topologie de communication dynamique.

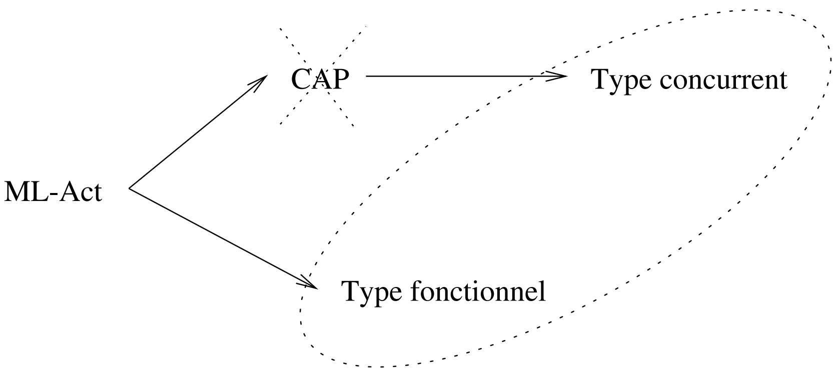
Chaque acteur présente une interface au monde extérieur et les messages qu'il recevra devront être compatibles avec celle-ci. Toutefois, ils diffèrent des objets concurrents classiques par le fait qu'ils peuvent changer leur interface dynamiquement. Ce phénomène appelé *changement de comportement* introduit une difficulté majeure au niveau du typage des programmes qui ne peut plus être réalisé par les techniques de typage usuelles des objets.

Dans le cadre de nos travaux, la validation statique d'un programme en langage d'acteurs consiste à détecter les messages orphelins, c'est-à-dire les messages qui sont envoyés à un acteur qui ne peut pas et ne pourra pas les prendre en compte. Deux catégories de messages orphelins peuvent apparaître : les orphelins de sûreté qui ne seront pas pris en compte car les comportements que prendra l'acteur ne savent pas traiter ces messages et les orphelins de vivacité qui ne seront pas pris en compte car l'acteur n'assumera pas les comportements (parmi ses comportements possibles) sachant traiter ces messages (cas d'interblocage). Dans une première étape, nos travaux se sont limités au cas des orphelins de sûreté. Dans le cadre du calcul de processus CAP (voir Colaço et al. [7, 8]), nous avons défini deux analyses statiques : la première [9], inspirée du typage des objets, ne détecte que les orphelins de sûreté triviaux (ceux qui ne peuvent être traité par aucun des comportements futurs de l'acteur), la seconde [12] détecte tous les orphelins de sûreté mais repose sur une abstraction de type plus complexe. Dans le cadre de cet article, nous nous limitons pour des raisons de simplicité à la première approche.

L'utilisation d'un calcul de processus permet de définir simplement les mécanismes d'abstraction et de résolution liés à l'analyse statique. Deux approches sont envisageables pour traiter le cas d'un langage complet : soit coder les constructions fonctionnelles en utilisant les constructions concurrentes mais les types obtenus pour les termes codés ne reflètent en général pas de manière satisfaisante la structure initiale de la partie fonctionnelle, soit étendre les mécanismes d'analyse statique de manière à traiter simultanément les deux aspects fonctionnels et concurrents. Dans le cadre de cette seconde approche, nous avons défini le langage ML-ACT (voir [14, 15]) qui étend le noyau du langage ML par la définition de comportement, la création d'acteurs et l'envoi de messages.

Dans une première expérience décrite dans [15] (voir figure 1.1), nous nous sommes inspiré des travaux de Nielson et Nielson [18, 19] consistant à calculer un effet concurrent (un terme CAP) conjointement

Figure 1.1 Evolution du typage de ML-ACT.



à l'inférence d'un type fonctionnel pour chaque programme ML-ACT, puis à appliquer les mécanismes d'analyse statique propre au calcul de processus CAP (voir [9, 10, 12]). Pour éviter les problèmes liés à l'unification des effets évoqués par Nielson et Nielson (une algèbre de processus n'est, en général, pas libre), nous avons choisi de construire un effet indépendamment du typage en effectuant des substitutions d'effets lors des appels de fonctions. Cette approche nous a permis de conserver l'expressivité du calcul CAP (Nielson et Nielson sont contraints à simplifier à outrance leur calcul de processus pour obtenir une algèbre libre sur laquelle il est possible de définir une procédure d'unification des termes [19]). Mais elle n'est pas complètement satisfaisante car elle ne nous permet pas de traiter le cas des fonctions récursives contenant des envois de messages ou des créations d'acteurs. Pour résoudre ce problème, nous proposons dans cet article de fusionner les deux analyses statiques (voir figure 1.1) pour effectuer les deux abstractions simultanément et éviter ainsi les restrictions (l'algèbre composée des types fonctionnels et des types concurrents est libre).

Nous présentons par la suite le langage mini-Act utilisé comme support pour la définition de l'analyse. Nous évoquons ensuite la structure des types utilisés, les règles de typage, l'algorithme de résolution de contraintes ainsi qu'une optimisation basée sur l'utilisation conjointe du sous-typage et de l'unification. Nous concluons par des exemples détaillés de typage et des perspectives sur la poursuite de nos travaux.

2. ML-Act.

Pour améliorer la clarté de la présentation du typage, nous utiliserons mini-Act qui simplifie la syntaxe de ML-ACT tout en conservant toutes les difficultés inhérentes au typage. Ainsi, nous n'utilisons pas les fonctions à plusieurs cas et les filtres, la conditionnelle suffisant à exprimer la notion de choix. Les analyses statiques exposés dans cet article s'étendent facilement à ces constructions (voir [13]).

Le langage mini-Act est composé d'expressions. La première partie de la grammaire est similaire au noyau fonctionnel de ML. Nous ne nous intéressons pas aux problèmes de parenthésage et de priorité ; nous supposons qu'aucun conflit de reconnaissance n'est possible durant l'évaluation d'une expression. Durant l'exposé de la syntaxe, nous utilisons la notation $\widetilde{x}$ pour représenter une suite quelconque de x. Les termes sont construits (en partie) par la grammaire suivante :

$$\boxed{\begin{array}{l} e ::= x \mid c \mid \text{fun } x \to e \mid op(\widetilde{e}) \mid e\ e \mid e;e \mid \text{let } x = e \text{ in } e \\ \quad\mid \text{letrec } x = e\ [\widetilde{\text{and } x = e}] \text{ in } e \mid \text{if } e \text{ then } e\ [\text{else } e] \end{array}}$$

On dispose d'un ensemble de variable $\mathbb{V}$, de constantes (les entiers, les flottants,..., () et [], les primitives +, fst, ::,...). Les primitives prennent un tuple comme unique argument. Les expressions ont la même signification qu'en ML.

Nous ajoutons à cette grammaire les règles concernant le noyau acteur, l'ensemble de nom $\mathbb{M}$ est utilisé pour les étiquettes de messages.

$$\boxed{\begin{array}{l} e ::= \text{send } m(\widetilde{e}) \text{ to } e \mid \{\text{mess } \widetilde{m(\tilde{e}) = e}\} \mid \text{become } e \\ \quad\mid \text{suicide} \mid \text{new } e \mid \text{ego} \end{array}}$$

L'envoi de messages est effectué par la primitive «send...». Ensuite figure le terme correspondant à la formation d'un comportement composé des messages acceptés et des réactions (ou programmes) associées. Le changement de comportement est réalisé par les instructions «become» et «suicide» (qui correspond à la disparition de l'acteur). La création d'acteurs se fait via la primitive «new». L'expression qui crée un acteur lui fournit alors son comportement initial. La variable «ego» permet à un acteur d'accéder à sa propre adresse.

La figure 2.1 présente l'exemple de deux acteurs `a` et `b` échangeant des messages. Ce programme comporte une erreur sémantique : l'acteur `b` reçoit un message `rep` sans paramètre alors qu'il sait traiter le même message avec un paramètre. Nous verrons comment détecter cette erreur en revenant sur cet exemple tout au long de la présentation du système de types.

Figure 2.1 Un exemple d'expression mini-Act

```
let a = new { mess m(c)= send rep() to c;
                         become { mess m(d)= () }
            mess p()= ()
          }
in let b = new { mess rep(f)=... }
   in send m(b) to a
```

3. Les types.

3.1. Définition des types.

La structure des types est dérivée du système de types proposé par J-L. Colaço et al. dans [11, 12]. Les types sont soit concurrents pour les *noms* et les *comportements*, soit fonctionnels pour les *fonctions* et les *constructeurs*.

La forme de ces types est spécifiée par la syntaxe donnée dans la définition suivante.

DÉFINITION 3.1 (SYNTAXE DES TYPES) *L'ensemble des types, parcouru par τ est défini par la grammaire suivante :*

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \textit{; variable de type} \\
 & | & \tau \rightarrow \tau & \textit{; fonction} \\
 & | & C(\tilde{\tau}) & \textit{; constructeur de type} \\
 & | & (\delta, \gamma) & \textit{; type de nom} \\
 & | & (\delta, \gamma)\triangleright & \textit{; type de comportement} \\
\delta & ::= & \chi & \textit{; variable d'interface} \\
 & | & [m_j(\tilde{\tau_j})^{j\in J}] & \textit{; type interface} \\
\gamma & ::= & \psi & \textit{; variable d'émission} \\
 & | & \langle m_j(\tilde{\tau_j})^{j\in J}\rangle & \textit{; type émission}
\end{array}
$$

Un type nom (δ, γ) est composé de deux parties, d'une part les messages acceptables par ce nom (la capacité δ) et d'autre part les messages qui sont émis vers ce nom (les messages ayant ce nom pour cible, γ). Le principe du typage consiste à s'assurer que δ est supérieur à γ, ce qui signifie informellement que l'ensemble des messages envoyés à un nom est inclus dans l'ensemble des messages que ce nom sait effectivement traiter.

Par exemple, le type $\big([m(([\,], \langle rep(\ )\rangle))\ ()],\quad \langle m(\tau_b)\rangle\ \big)$ associé à l'acteur d'adresse `a` de la figure 2.1 signifie que cet acteur peut traiter des messages `p()` sans argument et des messages `m(c)`, d'arité 1, avec comme contrainte sur l'argument `c` que celui-ci puisse traiter un message `rep()`. La deuxième partie du type indique que l'acteur reçoit un message `m(b)` où `b` a le type τ_b. Ces contraintes sont exprimées par la relation de sous-typage définie dans la section suivante.

Un type comportement est noté $(\delta, \gamma)\triangleright$. La partie δ du type représente l'ensemble des messages acceptables par le comportement (définis par le filtrage au niveau du corps) ainsi que ceux des «futurs» comportements (après un changement de comportement dans les réactions associées aux messages acceptés). La partie γ d'un type comportement représente les messages «latents» de ce comportement, c'est-à-dire les messages qui sont envoyés à l'*ego* dans le corps du comportement (lors de la prise en compte d'un message).

Une valeur du type *fonction* $\tau_1 \rightarrow \tau_2$ est une fonction dont le domaine a le type τ_1 et le co-domaine le type τ_2.

Le type constructeur $C(\tau_1, ..., \tau_n)$ correspond au produit cartésien nommé des types $\{\tau_i\}_{i \in [1..n]}$ qu'il regroupe. Les étiquettes de constructeurs appartiennent à un ensemble $\mathbb{C}$. Ils permettent de représenter les types de bases : entiers $Int()$, booléens $Bool()$, listes $List(\alpha)$... Pour cela nous disposerons d'un environnement où sont prédéfinis les constantes et opérateurs de base.

3.2. Le sous-typage.

Dans le contexte des objets concurrents et des acteurs, il existe une notion naturelle de sous-typage dans le sens où un acteur peut toujours être remplacé par un autre acteur offrant plus de méthodes (acceptant plus de messages) et imposant moins de contraintes sur les arguments. Cette notion est exprimée par une relation d'inclusion notée $\tau_1 \sqsubseteq \tau_2$ qui signifie qu'une valeur de type τ_1 peut remplacer une valeur de type τ_2. Nous utilisons le symbole $\sqsubseteq$ ainsi que les symboles $\sqcup$ et $\sqcap$ des unions et intersections correspondantes pour obtenir une notation uniforme au lieu de mélanger $\leqslant$, $\sqcup$, $\sqcap$.

Cette notion est exploitée pour exprimer la substitution d'un paramètre formel par un paramètre réel et ceci lors de l'appel de fonction, de la réception d'un message et de l'installation d'un comportement sur un nom.

L'inclusion de types $\sqsubseteq$ est définie par les règles suivantes. Le sous-typage des types nom et comportement est défini à partir des inclusions $\subseteq_i$ et $\subseteq_e$, portant sur les parties *interface* et *émission* d'un tel type.

DÉFINITION 3.2 (INCLUSION DE TYPES)

$$
\begin{array}{rcl}
\tau_1 \rightarrow \tau_2 \sqsubseteq \beta_1 \rightarrow \beta_2 & \Longleftrightarrow & \beta_1 \sqsubseteq \tau_1 \wedge \tau_2 \sqsubseteq \beta_2 \\
C(\tau_1, ..., \tau_n) \sqsubseteq C(\beta_1, ..., \beta_n) & \Longleftrightarrow & \forall i \in [1..n]\ \tau_i \sqsubseteq \beta_i \\
(\delta_1, \gamma_1) \sqsubseteq (\delta_2, \gamma_2) & \Longleftrightarrow & \delta_2 \subseteq_i \delta_1 \wedge \gamma_2 \subseteq_e \gamma_1 \\
(\delta_1, \gamma_1)\triangleright \sqsubseteq (\delta_2, \gamma_2)\triangleright & \Longleftrightarrow & \delta_1 \subseteq_i \delta_2 \wedge \gamma_1 \subseteq_e \gamma_2
\end{array}
$$

avec

$$
\begin{array}{rcl}
[m_j(\tilde{\alpha_j})^{j \in J}] \subseteq_i [m_k(\tilde{\alpha'_k})^{k \in K}] & \Longleftrightarrow & J \subseteq K \ \wedge \ (\forall l \in J)\ \tilde{\alpha_l} \tilde{\sqsupseteq} \tilde{\alpha'_l} \\
\langle m_j(\tilde{\alpha_j})^{j \in J} \rangle \subseteq_e \langle m_k(\tilde{\alpha'_k})^{k \in K} \rangle & \Longleftrightarrow & J \subseteq K \ \wedge \ (\forall l \in J)\ \tilde{\alpha_l} \tilde{\sqsubseteq} \tilde{\alpha'_l}
\end{array}
$$

où $\tilde{\sqsubseteq}$ représente l'inclusion terme à terme entre séquences de même longueur.

Les deux premières équivalences reprennent les définitions usuelles du sous-typage pour les fonctions et les constructeurs de données. Les deux équivalences suivantes, concernant les inclusions de noms et de comportements, sont détaillées ci-dessous.

Un type nom (δ_1, γ_1) remplace un autre type nom (δ_2, γ_2) si le premier acteur accepte au moins les mêmes messages que le deuxième ($\delta_2 \subseteq_i \delta_1$). De plus, les messages envoyés au deuxième acteur (le paramètre formel) doivent être pris en compte dans les envois au premier (le paramètre réel) ($\gamma_2 \subseteq_e \gamma_1$). En effet, dans le corps d'une fonction (ou d'un message), un paramètre formel peut être la cible d'un ou plusieurs messages. Lors de l'appel de cette fonction (ou de la réception de ce message), le type du paramètre réel doit prendre en compte ces envois de messages.

Dans l'exemple ci-dessous, le paramètre formel `x` a un type $(\delta_x, \langle m_2() \rangle)$ et le paramètre réel `a` un type $(\delta_a, \langle m_1()\, m_2() \rangle)$. Le type de `a` est bien un sous-type du type de `x`.

```
send m₁() to a;
(fun x → send m₂() to x)  a; ...
```

Lors d'un changement de comportement, le nouveau comportement, de type $(\delta_1, \gamma_1)\triangleright$, peut remplacer l'ancien, de type $(\delta_2, \gamma_2)\triangleright$, si le type *interface* du second inclue les méthodes du premier ($\delta_2 \supseteq_i \delta_1$) et si les messages envoyés au nouveau comportement sont pris en compte dans les émissions de l'ancien ($\gamma_2 \supseteq_e \gamma_1$). En effet, un type *comportement* reflète les messages définis dans le comportement lui-même ainsi que ceux des comportements futurs.

Prenons l'exemple d'un comportement :

```
{   mess m₁() = become {mess m₃() = send m₄() to ego }
    mess m₂() = ... }
```

Le type de ce comportement prendra en compte les messages acceptés `m1` et `m2` dans le présent ainsi que le message `m3` accepté par le possible comportement futur en cas d'acceptation du message $\mathtt{m}_1$. Le comportement global aura le type $([m_1()\, m_2()\, m_3()], \langle m_4()\rangle)\triangleright$ et le comportement «futur» le type $([m_3()], \langle m_4()\rangle)\triangleright$. Nous pouvons vérifier que le premier type est un sous-type du second. Le message $\mathtt{m}_4$ est pris en compte dans la partie émission des deux types. En effet, un acteur qui adopte ce comportement peut recevoir ce message dans le cas où il reçoit un message $\mathtt{m}_1$ puis $\mathtt{m}_3$.

L'utilisation de la contravariance de l'inclusion d'interface et de la covariance de l'inclusion d'émissions seront justifiées par la suite.

Les unions et intersections correspondant aux opérateurs de sous-typage et d'inclusions d'interfaces et d'émissions sont définies ci-dessous.

Une valeur de type $\tau_1 \sqcap \tau_2$ possède les caractéristiques communes aux types τ_1 et τ_2. Le type intersection indique qu'une même expression du langage peut jouer deux rôles différents. Cette notion a été initialement proposé par Coppo et al. dans [5].

Une valeur de type $\tau_1 \sqcup \tau_2$ possède l'ensemble des caractéristiques des types τ_1 et τ_2. Le type union permet par exemple d'exprimer le type du co-domaine d'une fonction définie par filtrage (cas de ML-Act), c'est-à-dire l'union des types des co-domaines de chaque abstraction intervenant dans cette définition.

Définition 3.3 (Union et intersection de types) *L'union et l'intersection sont des opérations associatives et commutatives définies par les égalités suivantes:*

$$\begin{array}{rcl}
\tau_1 \to \tau_2 \sqcup \beta_1 \to \beta_2 & = & (\tau_1 \sqcap \beta_1) \to (\tau_2 \sqcup \beta_2) \\
C(\tilde{\tau}) \sqcup C(\tilde{\tau'}) & = & C(\tilde{\tau} \tilde{\sqcup} \tilde{\tau'}) \\
(\delta_1, \gamma_1) \sqcup (\delta_2, \gamma_2) & = & (\delta_1 \cap_i \delta_2, \gamma_1 \cap_e \gamma_2) \\
(\delta_1, \gamma_1)\triangleright \sqcup (\delta_2, \gamma_2)\triangleright & = & (\delta_1 \cup_i \delta_2, \gamma_1 \cup_e \gamma_2)\triangleright
\end{array}$$

et

$$\begin{array}{rcl}
\tau_1 \to \tau_2 \sqcap \beta_1 \to \beta_2 & = & (\tau_1 \sqcup \beta_1) \to (\tau_2 \sqcap \beta_2) \\
C(\tilde{\tau}) \sqcap C(\tilde{\tau'}) & = & C(\tilde{\tau} \tilde{\sqcap} \tilde{\tau'}) \\
(\delta_1, \gamma_1) \sqcap (\delta_2, \gamma_2) & = & (\delta_1 \cup_i \delta_2, \gamma_1 \cup_e \gamma_2) \\
(\delta_1, \gamma_1)\triangleright \sqcap (\delta_2, \gamma_2)\triangleright & = & (\delta_1 \cap_i \delta_2, \gamma_1 \cap_e \gamma_2)\triangleright
\end{array}$$

et

$$\begin{array}{l}
\langle m_j(\tilde{\alpha_j})^{j\in J}\rangle \cup_e \langle m_k(\tilde{\alpha'_k})^{k\in K}\rangle = \\
\qquad \langle m_l(\tilde{\alpha_l} \tilde{\sqcup} \tilde{\alpha'_l})^{l\in J\cap K}\ m_j(\tilde{\alpha_j})^{j\in J\setminus K}\ m_k(\tilde{\alpha'_k})^{k\in K\setminus J}\rangle \\
{[m_j(\tilde{\alpha_j})^{j\in J}]} \cup_i [m_k(\tilde{\alpha'_k})^{k\in K}] = \\
\qquad [m_l(\tilde{\alpha_l} \tilde{\sqcap} \tilde{\alpha'_l})^{l\in J\cap K}\ m_j(\tilde{\alpha_j})^{j\in J\setminus K}\ m_k(\tilde{\alpha'_k})^{k\in K\setminus J}]
\end{array}$$

et

$$\begin{array}{rcl}
\langle m_j(\tilde{\alpha_j})^{j\in J}\rangle \cap_e \langle m_k(\tilde{\alpha'_k})^{k\in K}\rangle & = & \langle m_l(\tilde{\alpha_l} \tilde{\sqcap} \tilde{\alpha'_l})^{l\in J\cap K}\rangle \\
{[m_j(\tilde{\alpha_j})^{j\in J}]} \cap_i [m_k(\tilde{\alpha'_k})^{k\in K}] & = & [m_l(\tilde{\alpha_l} \tilde{\sqcup} \tilde{\alpha'_l})^{l\in J\cap K}]
\end{array}$$

Où $\tilde{\sqcup}$ et $\tilde{\sqcap}$ sont les opérateurs qui étendent $\sqcup$ et $\sqcap$ terme à terme pour les séquences de même longueur.

Les cas non décrits correspondent à des unions ou des intersections non définies, notamment l'union et l'intersection de deux constructeurs d'étiquettes différentes. Dans la pratique, ces cas non définis correspondent aux erreurs de typage dans l'algorithme de résolution des contraintes.

La contravariance et la covariance des opérateurs d'unions et intersections découlent des définitions des inclusions.

Les égalités concernant les fonctions sont en fait des approximations. En effet, le type $(\tau_1 \rightarrow \tau_2) \sqcap (\beta_1 \rightarrow \beta_2)$ apporte plus d'information que le type $(\tau_1 \sqcup \beta_1) \rightarrow (\tau_2 \sqcap \beta_2)$. Dans le premier cas, si l'argument est de type τ_1, le résultat sera de type τ_2 ; dans le deuxième cas le résultat est de type $\tau_2 \sqcap \beta_2$, partie commune à τ_2 et β_2.

Les opérations sur les types *interface* $\cup_i$ et $\subseteq_i$ correspondent aux opérations «plates» $\uplus$ et $\Subset$ proposées par J-L.Colaço et al. [9] pour représenter la composition des différents comportements que peut prendre un même acteur. L'opérateur $\cup_i$ permet de fournir une vue «plates» de l'acteur où tous les comportements, présents et futurs, sont pris en compte dans le type, mais où leur séquencement a disparu.

Reprenons l'exemple de la figure 2.1. Ne sachant pas à priori lequel des deux comportements de `a` traitera le message `m(b)`, le type de `a` doit donner des contraintes sur l'argument pour que le message soit correctement pris en compte dans les deux cas. Nous pouvons donc associer au nom `a` le type (δ_a, γ_a) avec δ_a égal à l'union $[m(\big([\,], \langle rep()\rangle\big))\ p()] \cup_i [m(\big([\,], \langle\,\rangle\big))]$ qui correspond aux deux possibilités de prise en compte du message `m` par l'acteur `a`. Le calcul de δ_a illustre la contravariance de l'opérateur $\cup_i$. En effet, δ_a est égal à $[m(\big([\,], \langle rep()\rangle\big) \sqcap \big([\,], \langle\,\rangle\big))\ p()]$ ce qui donne $[m(\big([\,], \langle rep()\rangle\big))\ p()]$. Sans la contravariance, nous aurions perdu l'information que l'argument du message `m` doit savoir traiter le message `rep`.

Les opérations sur les *émissions*, $\subseteq_e$ et $\cup_e$, restent classiques.

3.3. Les messages orphelins.

3.3.1. Les orphelins détectés.

Les types proposés permettent d'une part d'éviter les erreurs classiques : nombre et types des paramètres d'une fonction (ou d'un message) et de détecter certains messages orphelins de sûreté. En effet, un message orphelin de sûreté est un message envoyé à un acteur dont le comportement courant ainsi que ceux qu'il adoptera par le futur ne savent pas traiter le message en question. Avec l'opérateur de confrontation $\gtrapprox$ présenté ci-dessous, nous nous proposons d'éliminer ceux qui sont trivialement orphelins c'est à dire ceux qui ne seront pris en compte par aucun des comportements futurs possibles (réunis par l'union $\cup_i$ des toutes les interfaces).

DÉFINITION 3.4 (DÉTECTION DES MESSAGES ORPHELINS.)
L'opérateur de confrontation $\gtrapprox$ permet de comparer les parties interface et émission d'un type de nom :

$$[m_j(\tilde{\alpha_j})^{j\in J}] \gtrapprox \langle m_k(\tilde{\alpha'_k})^{k\in K}\rangle \iff K \subseteq J \ \wedge\ (\forall l \in K)\ \tilde{\alpha'_l} \tilde{\sqsubseteq} \tilde{\alpha}_l$$

Cet opérateur permet d'une part de s'assurer qu'il n'y a pas de messages «trivialement» orphelins, en vérifiant que l'ensemble des messages émis est inclus dans celui des messages acceptés, et d'autre part de contrôler la compatibilité (en nombre et en type) des paramètres des messages : le paramètre réel (celui du message envoyé) doit pouvoir remplacer le paramètre formel (celui du message accepté).

Sur l'exemple de la figure 2.1, la comparaison de l'interface de `a`, $[m(([\ ], \langle rep()\rangle))\ p()]$, avec ses émissions, $\langle m(\tau_b)\rangle$, permet de mettre en évidence que le nom `b` peut recevoir le message `rep()` (grâce à la contrainte $\tau_b \sqsubseteq ([\,], \langle rep()\rangle)$). Le type pour le nom `b` est donc $([rep(\tau_f)], \langle rep()\rangle)$ et, en comparant les deux parties de ce type, l'erreur sur le nombre de paramètres du message `rep` est détectée.

3.3.2. Les orphelins non détectés.

Les messages orphelins de sûreté que nous ne détecterons pas correspondent aux messages qui peuvent être pris en compte uniquement dans une branche qui ne sera pas choisie dynamiquement.

Prenons l'exemple d'un acteur qui suivant le premier message qu'il reçoit adopterait soit un comportement acceptant le message p soit un comportement n'acceptant pas ce message p. Si, le message p est envoyé à cet acteur, dans un cas sur deux, le message est orphelin et notre système ne le détectera pas, l'acteur ayant le message $p()$ dans son interface $[...p()...]$.

```
let  a = new   {  mess m() = become {mess p()=()}
                  mess n() = become {mess q()=()}
               }
 in  send n()  to  a;
     send p()  to  a
```

Les types proposés n'indiquent pas non plus le nombre de fois qu'un message peut etre accepté. Dans le cas où un acteur n'accepte qu'une seule fois un message et prend ensuite un comportement qui n'acceptera plus ce message, les types utilisés ne permettent pas de faire la différence entre l'envoi d'un seul message à cet acteur et l'envoi de n messages dont $n-1$ seront orphelins.

```
let cell =
   {mess init(v) = become {mess get(c) = send put(v) to c}}
in let a = new cell
   in send init(v1) a;
      send init(v2) to a
```

Nous reviendrons sur la détection de ces autres messages orphelins de sûreté dans la conclusion.

4. Le système de types.

Nous avons choisi d'effectuer un typage en deux étapes :

- la première consiste en la collecte par inférence de contraintes ensemblistes entre types,
- la seconde étant la résolution de ces contraintes.

Nous présentons dans une première étape les règles d'inférences. L'algorithme de résolution est présenté dans la partie suivante.

4.1. Les conventions

La description du système de type va suivre le principe de la sémantique naturelle. Le typage consistera donc en la construction d'un arbre de dérivation. Pour alléger les règles d'inférence, celle-ci ajouterons des contraintes dans un ensemble global de contraintes. Elle seront de la forme présentée ci-dessous où C est l'ensemble des contraintes issues de l'application de la règle.

$$\frac{E \vdash e_1 : \tau_1 \ldots E \vdash e_n : \tau_n}{E \vdash f(e_1, \ldots, e_n) : \tau}\{C\}$$

Les types de bases ML *integer, bool, char, string, 'a list* sont representés par les constructeurs $Int()$, $Bool()$, $Char()$, $String()$, $List(\alpha)$.

Les fonctions «*typeofcst*» et «*typeofop*» donnent les types des constantes et des opérateurs prédéfinis. Ces derniers ont des types polymorphes et les deux fonctions procèdent à l'instanciation du schéma de type à chaque utilisation d'une constante ou d'un opérateur. Ainsi, la liste vide «[]» a le type $List(\alpha)$, le constructeur de liste «::» le type $\alpha \to List(\alpha) \to List(\alpha)$, α étant remplacé par une nouvelle variable à chaque appel d'une des deux fonctions.

La notation $E \oplus \{x : \tau\}$ désigne l'ajout de l'association entre l'identificateur x et le type τ à l'environnement E. Cette liaison est prioritaire sur toute autre association concernant x dans E.

De plus, on définit le prédicat **CmpRec** sur la définition d'un comportement. Il détermine s'il existe une possibilité qu'un comportement soit récursif. En effet, un comportement boucle implicitement sur lui-même s'il n'exécute pas un *suicide* ou un *become*. Le calcul de ce prédicat se fait par inférence, en même temps que la collecte des contraintes. Il n'est pas présent dans les règles de typage pour ne pas alourdir celles-ci. Son algorithme est décrit ci-dessous.

- **CmpRec** sera implicitement vrai si un des corps de message, que le comportement accepte, est récursif.
- Un corps de message est implicitement récursif si dans la séquence d'instructions il n'y a ni suicide ni become.
- Le cas des conditionnelles fait que le prédicat est indécidable statiquement. Donc un *if then else* est assimilé à un changement de comportement si les deux branches exécutent un become ou suicide. Un *if then* ne sera jamais considéré comme un changement de comportement pour le calcul du prédicat. Ces deux approximations sont nécessaires pour obtenir la sûreté du système de types.

4.2. Les règles d'inférences

$$\text{EID} : \frac{x \in \mathrm{dom}(E)}{E \vdash x : E(x)} \qquad \text{ECST} : \frac{}{E \vdash c : \mathit{typeofcst}(c)}$$

Le typage d'une variable se déduit des hypothèses introduites par les règles (EFUNC) et (EBEH). Si cette variable n'apparaît pas dans les hypothèses, il s'agit d'une variable libre. Cette règle permet donc de mettre en évidence lors du typage les variables libres qui provoquent une erreur sémantique lors de l'évaluation.

$$\text{ELET} : \frac{E \vdash e_1 : \tau_1 \quad E \oplus \{x : \tau_1\} \vdash e_2 : \tau_2}{E \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

$$\text{ELETREC} : \frac{\alpha \text{ variable fraîche} \quad E \oplus \{x : \alpha\} \vdash e_1 : \tau_1 \quad E \oplus \{x : \tau_1\} \vdash e_2 : \tau_2}{E \vdash \text{letrec } x = e_1 \text{ in } e_2 : \tau_2} \; \left(\tau_1 \sqsubseteq \alpha\right)$$

La règle (ELET) construit l'environnement local, où x a le type

τ_1, dans lequel e_2 est typé. La seule différence entre les règles (ELETREC) et (ELET) est l'introduction de l'identificateur de fonction récursive, dans l'environnement de typage de la définition. La contrainte $\tau \sqsubseteq \alpha$ impose que le type associé à la variable introduite représente l'expression récursive, c'est à dire, qu'il soit au moins égal au type de l'expression. La règle (ELETEREC) est facilement étendue à une suite de définitions mutuellement récursives, en permettant l'accès à l'ensemble des identificateurs dans chaque définition.

$$\text{EFUNC}:\frac{E \oplus \{x : \alpha_x\} \vdash e : \tau_e}{E \vdash \text{fun } x \to e : \alpha_x \to \tau_e}$$

Le corps d'une fonction est typé dans un environnement où est défini son paramètre x à l'aide d'une variable fraîche α_x.

$$\text{EAPPLY}:\frac{\begin{array}{c}\alpha_3, \alpha_4 \text{ variables fraîches}\\ E \vdash ef : \tau_1 \quad E \vdash ep : \tau_2\end{array}}{E \vdash ef\ ep : \alpha_4} \begin{pmatrix} \tau_1 \sqsubseteq \alpha_3 \to \alpha_4 \\ \tau_2 \sqsubseteq \alpha_3 \end{pmatrix}$$

$$\text{EOP}:\frac{\begin{array}{c}E \vdash e_i : \tau_i\\ \tau_{op_1} \ldots \to \tau_{op_n} \to \tau_{op} = \mathit{typeofop}(\text{op})\end{array}}{E \vdash \text{op}(e_1, \ldots, e_n) : \tau_{op}} \left(\forall i \in [1..n], \tau_i \sqsubseteq \tau_{op_i}\right)$$

Les contraintes introduites lors du traitement de l'application expriment que la partie gauche d'une application doit avoir une structure fonctionnelle admettant au moins comme paramètre la partie droite de l'application.

L'application d'un opérateur est traitée comme l'application d'une fonction. La connaissance du nombre des paramètres permet de la simplifier.

$$\text{EIF\_1}:\frac{E \vdash e_1 : \tau_1 \quad E \vdash e_2 : \tau_2}{E \vdash \text{if } e_1 \text{ then } e_2 : \mathit{Unit}()} \begin{pmatrix} \tau_1 = \mathit{Bool}() \\ \tau_2 = \mathit{Unit}() \end{pmatrix}$$

$$\text{EIF\_2}:\frac{E \vdash e_1 : \tau_1 \quad E \vdash e_2 : \tau_2 \quad E \vdash e_3 : \tau_3}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_2 \sqcup \tau_3} \left(\tau_1 = \mathit{Bool}()\right)$$

$$\text{ESEQ}:\frac{E \vdash e_1 : \tau_1 \quad E \vdash e_2 : \tau_2}{E \vdash e_1;\, e_2 : \tau_2}$$

Pour la conditionnelle, la première expression doit être un booléen. Si le «else» est absent, on impose au «then» de ne rien renvoyer. Un «ifthenelse» possède l'ensemble des caractéristiques des types du «then» et du «else».

REMARQUE 1 *Etudions le cas de la conditionnelle :*

```
if c then acteur_1 else acteur_2
```

avec acteur_1 *de type* $([m1()m2()], \langle\rangle)$ *et* acteur_2 *de type* $([m2()m3()], \langle\rangle)$. *L'expression accepte soit les messages m1 et m2 soit les messages m2 et m3. Seul le message m2 est accepté dans tous les cas, la sûreté du typage impose donc de se limiter à accepter m2. La conditionnelle aura le type :* $([m1()\ m2()], \langle\rangle) \sqcup ([m2()\ m3()], \langle\rangle) = ([m2()], \langle\rangle)$ *d'après la définition 3.3.*

Le type d'une séquence (ESEQ) est celui de la dernière expression et nécessite le typage des deux expressions qui la composent.

$$\text{EBEH}: \frac{\begin{array}{c}\chi, \psi, \chi_e, \psi_e, \tilde{\alpha_j} \text{ variables fraîches}\\ E \oplus \{\tilde{x_j}{:}\tilde{\alpha_j}, \text{ego}{:}(\chi_e, \psi_e)\} \vdash e_j : \tau_j\end{array}}{E \vdash \{\text{mess } m_j(\tilde{x_j}) = e_j\}^{j \in J} : (\chi, \psi)\triangleright} \left(\begin{array}{c}\chi \supseteq_i [m_j(\tilde{\alpha_j})^{j\in J}]\\ \chi \supseteq_i \chi_e, \psi \supseteq_e \psi_e\\ \chi_e \gtrapprox \psi_e\\ \forall j \in J,\ \tau_j = Unit()\\ \text{si } \mathbf{CmpRec}\ () \text{ alors}\\ (\chi, \psi)\triangleright \sqsubseteq (\chi_e, \psi_e)\triangleright\end{array}\right)$$

Le corps e_j d'un message m_j est typé dans un environnement où sont prédefinis les paramètres $\tilde{x_j}$ du message ainsi que l'adresse de l'acteur (ego). Le contrôle de la présence de l'ego dans l'environnement permettra donc de vérifier l'utilisation correcte des primitives acteurs (suicide, become).

Les contraintes produites indiquent que l'acteur d'adresse (χ, ψ) accepte les messages $\mathbf{m}_j$ (représentant l'état «présent» de son interface) ainsi que ceux associés aux futurs comportements ($\chi \supseteq_i \chi_e$). La contrainte $\psi \supseteq_e \psi_e$ permet de cumuler les envois de messages faits sur l'«ego» des comportements courant et futurs.

Nous vérifions également la validité du type de l'ego, ce qui permet de détecter les messages «trivialement» orphelins.

Un acteur reprend implicitement le même comportement s'il n'exécute pas un *suicide* ou un *become*. Le prédicat **CmpRec** permet de le détecter et de générer la contrainte qui correspond à un acteur reprenant son comportement courant.

$$\text{ESEND}: \frac{E \vdash \tilde{e} : \tilde{\tau}_1 \quad E \vdash \mathrm{a} : \tau_2}{E \vdash \text{send } m(\tilde{e}) \text{ to } a : Unit()} \left(\tau_2 \sqsubseteq ([\,], \langle m(\tilde{\tau}_1)\rangle)\right)$$

$$\text{EBECOME}: \frac{E \vdash e : \tau \quad E(\mathrm{ego}) = (\chi_e, \psi_e)}{E \vdash \text{become e} : Unit()} \left(\tau \sqsubseteq (\chi_e, \psi_e) \triangleright\right)$$

$$\text{ESUICIDE}: \frac{\mathrm{ego} \in \mathrm{dom}(E)}{E \vdash \text{suicide} : Unit()}$$

$$\text{ENEW}: \frac{E \vdash e : \tau \quad \chi, \psi \text{ variables fraîches}}{E \vdash \text{new } e : (\chi, \psi)} \begin{pmatrix} \tau \sqsubseteq (\chi, \psi)\triangleright \\ \chi \gtrapprox \psi \end{pmatrix}$$

L'envoi d'un message (ESEND) est bien typé si le type de sa cible peut remplacer un type de nom dont la partie émission est formé par l'étiquette du message et le type de ses arguments.

Lors d'un changement de comportement (EBECOME), le type du nouveau comportement τ, installé à l'adresse *ego*, doit remplacer le comportement formel $(\chi_e, \psi_e)\triangleright$.

Pour que la création d'acteur (ENEW) soit bien typé, il faut que le comportement e de type τ, installé à la nouvelle adresse de type (χ, ψ), remplace son comportement formel de type $(\chi, \psi)\triangleright$. La relation $\chi \gtrapprox \psi$ permet d'assurer, tout au long de la résolution, la compatiblité entre paramètres réels et formels, ainsi que la détection, en fin de résolution des messages orphelins pour ce nom.

La règle (ESUICIDE) contrôle si le suicide est exécuté dans le corps d'un comportement et elle permet, avec la règle (EBECOME), de mettre à jour le prédicat **CmpRec**.

5. Résolution.

La collecte des contraintes, obtenues par inférence, nous donne cinq systèmes :

- le système des inclusions de sous-typage S_{st},
- le système des inclusions d'interfaces S_i,
- le système des inclusions d'émissions S_e,
- le système des égalités,
- l'ensemble des confrontations $\mathcal{E}_{\gtrapprox}$.

Dans un premier temps nous présentons un algorithme basé uniquement sur le sous-typage, puis nous l'étendrons simplement aux contraintes d'égalités.

5.1. L'algorithme de résolution.

Nous nous inspirons des algorithmes de résolution d'Aiken et Wimmers [3] et de Colaço [6] dont les propriétés de correction et de terminaison ont été montrées. Les paragraphes *calcul des interfaces et émissions* et *fermeture du système d'inclusions* décrivent les calculs effectués sur chacun des systèmes. L'algorithme de résolution des trois systèmes est présenté dans un troisième paragraphe. Il reprend les algorithmes présentés dans les deux premiers.

Calcul des interfaces et émissions.

On applique l'algorithme suivant à S_i :

1. Simplifier le système avec la règle :

$$S_i \cup \{\chi \supseteq_i I_1,\, \chi \supseteq_i I_2\} \longrightarrow S_i \cup \{\chi \supseteq_i I_1 \cup_i I_2\}$$

   où I_1 et I_2 sont des types interfaces effectifs (i.e. pas des variables).
2. Fermer le système par transitivité :

$$S_i \cup \{\delta_1 \supseteq_i \chi,\, \chi \supseteq_i \delta_2\} \;\longrightarrow\; S_i \cup \{\delta_1 \supseteq_i \chi,\, \chi \supseteq_i \delta_2,\, \delta_1 \supseteq_i \delta_2\}$$

3. Réeffectuer la simplification de l'étape 1.

Nous procédons de la même façon sur le système S_e : simplification, transitivité et à nouveau simplification.

Nous pouvons avoir une connaissance partielle de l'interface et de l'ensemble des émissions d'un nom de type (χ, ψ), à ce moment de la résolution :

- l'interface de l'acteur «est» $[m_j(\tilde{\tau}_j)^{j\in J}]$ si $\chi \supseteq_i [m_j(\tilde{\tau}_j)^{j\in J}] \in S_i$,
- l'ensemble des messages émis vers l'acteur «est» $\langle m_j(\tilde{\tau}_j)^{j\in J}\rangle$ si $\psi \supseteq_e \langle m_j(\tilde{\tau}_j)^{j\in J}\rangle \in S_e$.

L'étapes 3 de la résolution sur les systèmes S_i et S_e permet d'assurer l'unicité des ces interfaces et émissions «partielles». Nous nous servirons de ces calculs partiels dans la phase suivante de la résolution.

Fermeture du système d'inclusions.

La fermeture du système S_{st} est composée des étapes suivantes, appliquées jusqu'à obtenir un système stable :

1. Décomposition du système :

Les règles 2 à 5 (Fig. 5.1) de décomposition provenant de la définition 3.2 de l'inclusion de sous-typage, ainsi que la règle 6 de comparaison des interfaces et émissions d'un acteur sont appliquées à S_{st}.

La règle 6 compare les messages que reçoit «potentiellement» un acteur avec sa capacité de traitement, en utilisant les résultats du calcul partiel des interfaces et émissions d'un nom exposé au paragraphe précedent. Cette règle permet de vérifier la compatibilité des arguments de message ainsi que leur arité. Par contre, les messages «trivialement» orphelins, i.e. qu'aucun comportement ne pourra traiter, ne sont pas détectés à ce moment. En effet, le calcul des interfaces et des émissions effectué n'est que partiel. Par exemple, si le paramètre d'un message est un comportement, l'interface de l'acteur ne sera complétée qu'après la comparaison entre parametres réels et formels du message (phase que nous décrivons dans ce paragraphe). Pour cette raison, les messages orphelins seront détectés en fin de résolution.

Chaque inclusion disparaît du système lorsqu'elle est décomposée (en dehors de la règle 6).

2. Transitivité :

   Le système est fermé par la règle 1 (Fig. 5.1). Chaque nouvelle inclusion de sous-typage est à son tour décomposée.

Figure 5.1 Règles de fermeture du système S_{st}

$$
\begin{array}{ll}
1. & \tau \sqsubseteq \alpha \in S \wedge \alpha \sqsubseteq \tau' \in S \Rightarrow \tau \sqsubseteq \tau' \in S \\
2. & \tau_1 \rightarrow \tau_2 \sqsubseteq \tau_3 \rightarrow \tau_4 \in S \Rightarrow \{\tau_3 \sqsubseteq \tau_1, \tau_2 \sqsubseteq \tau_4\} \in S \\
3. & C(\tilde{\tau}) \sqsubseteq C(\tilde{\tau'}) \in S \Rightarrow \tilde{\tau} \tilde{\sqsubseteq} \tilde{\tau'} \in S \\
4. & (\delta_1, \gamma_1) \sqsubseteq (\delta_2, \gamma_2) \in S_{st} \Rightarrow \delta_2 \subseteq_i \delta_1 \in S_i \wedge \gamma_2 \subseteq_e \gamma_1 \in S_e \\
5. & (\delta_1, \gamma_1)\triangleright \sqsubseteq (\delta_2, \gamma_2)\triangleright \Rightarrow \delta_1 \subseteq_i \delta_2 \in S_i \wedge \gamma_1 \subseteq_e \gamma_2 \in S_e \\
6. & \left.\begin{array}{r} \chi \supseteq_i \langle m_i(\tilde{\tau}_i)^{i \in I} \rangle \in S_i \\ \psi \supseteq_e \langle m_j(\tilde{\tau'_j})^{j \in J} \rangle \in S_e \\ \chi \gtrapprox \psi \in \mathcal{E}_{\gtrapprox} \end{array}\right\} \Rightarrow \forall k \in (I \cap J)\ \tilde{\tau'_k} \tilde{\sqsubseteq} \tilde{\tau}_k \in S
\end{array}
$$

La figure 5.1 présente les règles utilisées dans les deux phases de la fermeture du système S_{st}. Toutes produisent de nouvelles inclusions de sous-typage (système S_{st}), entre interface (système S_i) ou entre émission (système S_e).

Algorithme de résolution des différents systèmes.

Pour résoudre les trois systèmes, on applique l'algorithme suivant reprenant les algorithmes exposés dans les paragraphes précédents.

1. Calcul des interfaces sur S_i.
2. Calcul des émissions sur S_e.
3. Fermeture du système S_{st}.
   Si de nouvelles inclusions ont été produites dans S_i ou S_e, les étapes 1, 2 et 3 sont de nouveau appliquées pour recalculer les interfaces et émissions des acteurs puis fermer le système S_{st}.
4. On détecte les messages «trivialement» orphelins tels que :

$$\left.\begin{array}{r} \chi \supseteq_i \langle m_i(\tilde{\tau}_i)^{i \in I} \rangle \in S_i \\ \psi \supseteq_e \langle m_j(\tilde{\tau}'_j)^{j \in J} \rangle \in S_e \\ \chi \gtrapprox \psi \in \mathcal{E}_{\gtrapprox} \end{array}\right\} \Rightarrow \forall k \in (J \setminus I)\ m_k \text{ message orphelin}$$

   En effet, un tel message m_k est «potentiellement» reçu par l'acteur de type (χ, ψ) (inclusion $\supseteq_e$), mais ce dernier ne le traite pas (inclusion $\supseteq_i$).

Cet algorithme a été développé pour traiter uniquement des contraintes de sous-typage. Aussi, pour rendre plus performant le typage, nous introduisons des contraintes d'égalités et procédons à quelques modifications exposées dans le paragraphe suivant.

5.2. Traitement des égalités.

Un programme ML-Act de «grande taille» peut générer un nombre important de contraintes lors de la résolution. Aussi nous cherchons à réduire ce nombre en procédant à des simplifications proposées par Aiken et Fähndrich dans [16]. Il s'agit de transformer un certain nombre d'inclusions de sous-typage en égalités, pour se rapprocher du typage à la ML pour l'aspect fonctionnel de ML-Act. En effet, nous souhaitons conserver les types précis que nous fournit le sous-typage pour les types concurrents et revenir à des types plus classiques pour les types fonctionnels. Les règles de simplification sont exposés dans la figure 5.2.

Ainsi une contrainte $\alpha \sqsubseteq Int()$ se transforme en $\alpha = Int()$. Pour les constructeurs avec paramètres : la contrainte $\alpha_1 \sqsubseteq List(\alpha_2)$ donnera $\alpha_1 = List(\alpha_3)$ et $\alpha_3 \sqsubseteq \alpha_2$.

De même, les unions et intersections de sous-typage se réécrivent en produisant des égalités (figure 5.3).

Figure 5.2 Simplification des inclusions.

$$
\begin{array}{rcl}
\alpha \sqsubseteq \tau_1 \rightarrow \tau_2 \in S_{st} & \Longrightarrow & \alpha = \alpha_1 \rightarrow \alpha_2 \wedge \\
 & & \{\tau_1 \sqsubseteq \alpha_1,\ \alpha_2 \sqsubseteq \tau_2\} \subseteq S_{st} \\
\tau_1 \rightarrow \tau_2 \sqsubseteq \alpha \in S_{st} & \Longrightarrow & \alpha = \alpha_1 \rightarrow \alpha_2 \wedge \\
 & & \{\alpha_1 \sqsubseteq \tau_1,\ \tau_2 \sqsubseteq \alpha_2\} \subseteq S_{st} \\
C(\tau_1, ..., \tau_n) \sqsubseteq \alpha \in S_{st} & \Longrightarrow & \alpha = C(\alpha_1, ..., \alpha_n) \wedge \\
 & & \{\tau_i \sqsubseteq \alpha_i\}_{\forall i \in [1..n]} \subseteq S_{st} \\
\alpha \sqsubseteq C(\tau_1, ..., \tau_n) \in S_{st} & \Longrightarrow & \alpha = C(\alpha_1, ..., \alpha_n) \wedge \\
 & & \{\alpha_i \sqsubseteq \tau_i\}_{\forall i \in [1..n]} \subseteq S_{st} \\
\alpha \sqsubseteq (\delta, \gamma) \in S_{st} & \Longrightarrow & \alpha = (\chi, \psi) \wedge \\
 & & \chi \supseteq_i \delta \in S_i \ \wedge \ \psi \supseteq_e \gamma \in S_e \\
(\delta, \gamma) \sqsubseteq \alpha \in S_{st} & \Longrightarrow & \alpha = (\chi, \psi) \wedge \\
 & & \delta \supseteq_i \chi \in S_i \ \wedge \ \gamma \supseteq_e \psi \in S_e \\
\alpha \sqsubseteq (\delta, \gamma)\triangleright \in S_{st} & \Longrightarrow & \alpha = (\chi, \psi) \triangleright \wedge \\
 & & \chi \subseteq_i \delta \in S_i \ \wedge \ \psi \subseteq_e \gamma \in S_e \\
(\delta, \gamma)\triangleright \sqsubseteq \alpha \in S_{st} & \Longrightarrow & \alpha = (\chi, \psi) \triangleright \wedge \\
 & & \delta \subseteq_i \chi \in S_i \ \wedge \ \gamma \subseteq_e \psi \in S_e
\end{array}
$$

où α_1, α_2, α_i, χ, ψ désignent des variables fraîches

Ces simplifications permettent de faire disparaître une grande partie des contraintes de sous-typage, celles sur la partie fonctionnelle d'un programme ML-ACT. Il ne reste alors plus que les contraintes sur les interfaces et les émissions concernant la partie acteur du programme.

Figure 5.3 Simplification des unions et intersections.

$$
\begin{array}{rcll}
\alpha \sqcup \tau_1 \rightarrow \tau_2 & = & (\alpha_1 \sqcap \tau_1) \rightarrow (\alpha_2 \sqcup \tau_2) & \text{avec} \quad \alpha = \alpha_1 \rightarrow \alpha_2 \\
\alpha \sqcap \tau_1 \rightarrow \tau_2 & = & (\alpha_1 \sqcup \tau_1) \rightarrow (\alpha_2 \sqcap \tau_2) & \text{avec} \quad \alpha = \alpha_1 \rightarrow \alpha_2 \\
C(\tilde{\tau}) \sqcap \alpha & = & C(\tilde{\tau} \tilde{\sqcap} \tilde{\alpha}) & \text{avec} \quad \alpha = C(\tilde{\alpha}) \\
C(\tilde{\tau}) \sqcup \alpha & = & C(\tilde{\tau} \tilde{\sqcup} \tilde{\alpha}) & \text{avec} \quad \alpha = C(\tilde{\alpha}) \\
(\delta, \gamma) \sqcap \alpha & = & (\delta \cup_i \chi, \gamma \cup_e \psi) & \text{avec} \quad \alpha = (\chi, \psi) \\
(\delta, \gamma) \sqcup \alpha & = & (\delta \cap_i \chi, \gamma \cap_e \psi) & \text{avec} \quad \alpha = (\chi, \psi) \\
(\delta, \gamma) \triangleright \sqcap \alpha & = & (\delta \cap_i \chi, \gamma \cap_e \psi)\triangleright & \text{avec} \quad \alpha = (\chi, \psi)\triangleright \\
(\delta, \gamma) \triangleright \sqcup \alpha & = & (\delta \cup_i \chi, \gamma \cup_e \psi)\triangleright & \text{avec} \quad \alpha = (\chi, \psi)\triangleright
\end{array}
$$

où α_1, α_2, $\tilde{\alpha}$, χ, ψ désignent des variables fraîches

Pour appliquer l'algorithme de résolution du paragraphe 5.1 en tenant compte des égalités, il suffit de mettre à jour les contraintes de sous-typage restantes tout au long de la résolution, en procédant aux substitutions des variables de type.

Nous allons étudier quelques exemples de résolution dans la section suivante.

5.3. Propriétés du système.

Le système de types est basé sur ceux de ML et de CAP tous les deux corrects. La correction du typage de ML-ACT découle de la composition de ces deux preuves (en fait, l'ajout des aspects fonctionnels à CAP correspond grossièrement à une transitivité supplémentaire dans le typage de CAP). Nous conjecturons donc cette correction dont nous construisons actuellement la preuve.

L'algorithme de résolution des contraintes est une adaptation directe des algorithmes de Aiken et Al. dont la correction et la terminaison ont été prouvées [2]. La preuve de correction du système de types visera à démontrer d'une part qu'un programme bien typé ne pourra pas s'évaluer en une erreur (conflit de type, arité d'un message) et d'autre part que chaque message orphelin «trivial» est bien détecté (ce qui n'est pas une erreur, le message étant stocké pour rien dans la boîte aux lettres). Par contre, tout message détecté comme orphelin par le système n'est pas forcément orphelin à l'exécution ; c'est le cas lors de l'utilisation d'une conditionnelle ou d'une structure de données comme les listes (voir l'exemple de la section précédente).

6. Exemples.

6.1. Un acteur recevant son comportement

Etudions l'exemple d'un acteur de comportement initial `a_definir` recevant son futur comportement par un message `take_beh`.

```
let a_definir =
   {  mess take_beh cmp =
         become cmp
   }
in let quelconque =
      {  mess m =
            ...
      }
   in let a = new a_definir
      in send take_beh(quelconque);
         send m() to a
```

Le comportement `a_definir` a pour type $(\chi, \psi)\triangleright$ avec les contraintes :

$$\begin{array}{llcl} (1) & \chi & \supseteq_i & [take\_beh(\alpha_{cmp})] \\ (2) & \chi & \supseteq_i & \chi_{ego} \\ (3) & \alpha_{cmp} & \sqsubseteq & (\chi_{ego}, \psi_{ego})\triangleright \end{array}$$

Les deux dernières contraintes permettent de compléter l'interface de l'acteur dans le cas où celui-ci recevrait un message `take_beh`. En effet, la dernière contrainte indique que le paramètre du message est un comportement : $\alpha_{cmp} = (\chi_{cmp}, \psi_{cmp})\triangleright$. Elle se décompose en

$$\begin{array}{llcl} (4) & \chi_{ego} & \supseteq_i & \chi_{cmp} \\ (5) & \psi & \supseteq_e & \psi_{cmp} \end{array}$$

Ainsi, la contrainte $\chi \supseteq_i \chi_{cmp}$ (6), obtenue par transitivité de (2) et (4), permet de compléter, en cas de réception de message, l'interface provisoire du comportement `a_definir`, donnée par la contrainte $\chi \supseteq_i [take\_beh((\chi_{cmp}, \psi_{cmp})\triangleright)]$ (1).

La création de l'acteur `a` et l'envoi du message `take_beh` donnent de plus :

$$\begin{array}{llcl} (7) & \chi_a & \supseteq_i & \chi \\ (8) & (\chi_a, \psi_a) & \sqsubseteq & ([\,], \langle take\_beh((\chi_2, \psi_2)\triangleright)\rangle) \end{array}$$

$(\chi_2, \psi_2)\triangleright$ représentant le type du comportement `quelconque`, dont l'interface est donnée par $\chi_2 \supseteq_i [m_2()]$ (9).

Ainsi, la comparaison de l'interface «provisoire» de l'acteur, $[take\_beh((\chi_{cmp}, \psi_{cmp})\triangleright)]$, avec le message envoyé , donné par $\psi_a \supseteq_e \langle take\_beh((\chi_2, \psi_2)\triangleright)\rangle$ (10) à partir de (8), permet de complèter l'interface de l'acteur `a`. Nous obtenons alors $(\chi_2, \psi_2)\triangleright \sqsubseteq (\chi_{cmp}, \psi_{cmp})\triangleright$ (11). La décomposition de (11) donne $\chi_{cmp} \supseteq_i \chi_2$ (12) puis $\chi_a \supseteq_i \chi \supseteq_i \chi_{cmp} \supseteq_i \chi_2 \supseteq_i [m_2()]$ (13) à partir de (7), (6), (12) et (9). L'interface de `a` devient $\chi_a \supseteq_i \langle m()\ take\_beh(\alpha_{cmp})\rangle$ en combinant (1) et (13).

Ainsi l'envoi d'un message `send m() to a` est typable.

6.2. Une fonction de diffusion

Etudions l'exemple d'un fonction qui envoit le message `m()` à tous les élements d'une liste reçue comme paramètre :

```
letrec broadcast =
    fun l ->
       if l=[] then ()
       else
          send m() to (hd l);
          broadcast (tl l)
in ...
```

Avec les hypothèses :

$$\begin{aligned} \texttt{broadcast} &: \alpha_{broad} \\ \texttt{l} &: \alpha_l \\ \texttt{[]} &: List(\alpha) \\ \texttt{=} &: \alpha_1 \to \alpha_1 \to Bool() \\ \texttt{hd} &: List(\alpha_2) \to \alpha_2 \\ \texttt{tl} &: List(\alpha_3) \to List(\alpha_3) \end{aligned}$$

Les contraintes inférées sont :

$$\begin{array}{rll} \left.\begin{array}{r} \alpha_l \sqsubseteq \alpha_1 \\ List(\alpha) \sqsubseteq \alpha_1 \end{array}\right\} & \texttt{(l=[])} & \begin{array}{r}(1)\\(2)\end{array} \\ \alpha_l \sqsubseteq List(\alpha_2) & \texttt{(hd l)} & (3) \\ \alpha_2 \sqsubseteq ([\,], \langle m() \rangle) & \texttt{(send m() to (hd l))} & (4) \\ \alpha_l \sqsubseteq List(\alpha_3) & \texttt{(tl l)} & (5) \\ \left.\begin{array}{r} \alpha_{broad} \sqsubseteq \alpha_4 \to \alpha_5 \\ List(\alpha_3) \sqsubseteq \alpha_4 \end{array}\right\} & \text{(appel récursif)} & \begin{array}{r}(6)\\(7)\end{array} \\ \alpha_l \to (Unit() \sqcup \alpha_5) \sqsubseteq \alpha_{broad} & \text{(récursivité)} & (8) \end{array}$$

La résolution débute par une suite de transformations d'inclusions de sous-typage en égalités.

L'union $Unit() \sqcup \alpha_5$ donne $\alpha_5 = Unit()$, les inclusions (6) et (8) sur le type α_{broad}, $\alpha_{broad} = \alpha_6 \to \alpha_7$. Donc (8) se décompose en

$$\begin{array}{ll} \alpha_6 \sqsubseteq \alpha_l & (9) \\ \alpha_7 \sqsubseteq Unit() & (10) \end{array}$$

De la même manière, la contrainte (3) se transforme en

$$\begin{array}{ll} \alpha_l = List(\alpha_8) & \\ List(\alpha_8) \sqsubseteq List(\alpha_2) & (11) \end{array}$$

et donc (9) devient

$$\begin{array}{ll} \alpha_6 = List(\alpha_9) & \\ List(\alpha_9) \sqsubseteq List(\alpha_8) & (12) \end{array}$$

A partir de (4), (11) et (12), on forme $\alpha_9 \sqsubseteq \alpha_8 \sqsubseteq \alpha_2 \sqsubseteq ([\,], \langle m() \rangle)$ (13) qui se décompose en $\alpha_9 = (\chi, \psi)$, $\chi \supseteq_i [\,]$ et $\psi \supseteq_e \langle m() \rangle$

Donc, le type obtenu pour la fonction `broadcast` est : $List(\,(\chi, \psi)\,) \to Unit()$ avec $\chi \supseteq_i [\,]$ et $\psi \supseteq_e \langle m() \rangle$

6.3. Polymorphisme et simplification.

Les système présenté ne fait pas intervenir de polymorphisme. Il produit un type ainsi qu'un ensemble de contraintes sur les variables apparaissant dans ce type. L'introduction de polymorphisme permet d'effectuer des résolutions partielles et de remplacer certaines contraintes par des égalités (voir les travaux d'Aiken et al. [4]). Ces simplifications produisent des types plus lisibles et améliorent nettement les performances de l'algorithme de résolution. Ainsi, le résultat obtenu pour la fonction `broadcast` serait $List(\,([\,], \langle m()\rangle)\,) \rightarrow Unit()$. L'exploitation du polymorphisme dans la partie fonctionnelle pure de ML-ACT ne pose pas de problème. Par contre, le changement de comportement et l'envoi de message sont des effets impurs imposant un traitement ad hoc du polymorphisme que nous étudions actuellement.

7. Conclusion et perspectives.

Le principal intérêt de l'approche adoptée a été d'analyser simultanément les aspects fonctionnels et concurrents du langage. Ce système n'impose plus les limitations contraignantes sur les programmes à typer du précedent système [15] qui séparait ces deux aspects. Il permet donc de conserver des types concurrents précis.

L'unification permet le typage d'exemples conséquents en limitant l'explosion du nombre des contraintes. Il reste à améliorer la résolution en procédant à des résolutions partielles, lors de l'introduction du polymorphisme, aux niveaux des définitions «let». Chaque résolution partielle sera suivie d'une saturation des types permettant d'éliminer les contraintes inutiles et d'associer à chaque entité un type polymorphe.

Le système proposé dans ce document ne permet pas de détecter tous les messages orphelins. J-L. Colaço et al. ont proposé dans [12] un système de types qui exploite une structure de type multi-ensembliste. L'étape suivante de nos travaux consiste à utiliser cette nouvelle structure. La difficulté majeure de cette extension résulte de la nécessité de pouvoir distinguer le cas de l'application de la fonction `broadcast` à une liste de n acteurs différents (qui recevront donc une fois le même message) du cas d'une liste contenant n fois le même acteur (qui recevra n fois le même message).

D'autre part, les analyses actuelles portent sur la programmation concurrente dans un environnement idéal sans panne. Cette hypothèse forte ne permet pas de décrire de manière satisfaisante le comportement de l'environnement de calcul réparti qu'offre le réseau INTERNET. L'ajout de la notion de site, de panne d'un site et de reprise sur panne dans ML-ACT permettra de combler cette lacune.

Le but est donc le développement d'un outil puissant permettant un diagnostique pertinent des communications dans un programme d'acteurs.

Références

[1] G. Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, Mass., 1986.

[2] A. Aiken and E. Wimmers *Solving Systems of Set Constraints"* LICS92, pages 329-340, june 1992,

[3] A. Aiken and E. Wimmers. Type Inclusion Constraints and Type Inference. FPCA93, Copenhagen, Denmark, pages 31-41, june 1993.

[4] A. Aiken and E. Wimmers. Optimal representations of polymorphic types with subtyping. In *Proc. of TACS'97, International Symposium on Theoretical Aspects of Computer Software*, pages 47-76. Springer-Verlag (LNCS 1281), Sendai, Japan, September 1997.

[5] M. Coppo and M. Dezani and P. Sallé. Functional characterization of some semantic equalities inside λ-calculus. 1979.

[6] J-L. Colaço. Analyses statiques d'un calcul d'Acteurs par typage. Thèse, Institut National Polytechnique de Toulouse, Octobre 1997.

[7] J-L. Colaço, M. Pantel, P. Sallé, and A. Senteni. Un calcul d'acteurs primitifs (CAP). Dans *Actes des JFLA*, Janvier 1996.

[8] J-L. Colaço, M. Pantel, and P. Sallé. CAP: An actor dedicated process calculus. In *ECOOP'96 Workshop on PTCOOP*, May 1996.

[9] J-L. Colaço and M. Pantel and P. Sallé. A set-constraint-based analysis of Actors. In *Proc. of the 1997 IFIP International conference on FMOODS*, Chapman & Hall, July 1997.

[10] J-L. Colaço and M. Pantel and P. Sallé. Analyse de linéarité par typage dans un calcul d'Acteurs. Dans *Actes des JFLA*, Janvier 1997.

[11] J-L. Colaço and M. Pantel and P. Sallé. Static Analysis of Behavior Changes in Actor languages. In *Proc. of the 2^{nd} France Japan Workshop on OBPDC*, October 1997.

[12] J-L. Colaço and M. Pantel and F. Dagnat and P. Sallé. Static safety analysis for non-uniform service availability in Actors In *Proc. of the 1999 IFIP International conference on FMOODS*, Chapman & Hall, February 1999.

[13] M. Colin. Intégration des Typages Fonctionnel et Concurrent d'un Langage Fonctionnel d'Acteurs. Rapport de DEA, ENSEEIHT, Septembre 1998.

[14] F. Dagnat Conception d'un langage Fonctionnel d'Acteurs et réalisation de son compilateur. Rapport de DEA, ENSEEIHT, Septembre 1997.

[15] F. Dagnat and M. Pantel and P. Sallé. ML-Act, un langage fonctionnel d'Acteurs. Dans *Actes des JFLA*, Février 1998.

[16] M. Fähndrich and A. Aiken. Program Analysis Using Mixed Term and Set Constraints. In *SAS*, 1997.

[17] C. Hewit, P. Bishop, and R. Steiger. An universal modular actor formalism for artificial intelligence. In *Proc. of the IJCAI'73*, 1973.

[18] F. Nielson and H.R. Nielson. From CML to process algebras. In *CONCUR'93*, LNCS 715, 1993.

[19] H.R. Nielson., F. Nielson and T. Amtoft. Polymorphic Subtyping for Effect Analysis : the Integration ; the Semantics ; the Algorithm. In *Proc. of the Fifth {LOMAPS} Workshop*, Springer-Verlag, 1997.