

Vers un ERLANG typé ?

Fabien Dagnat

Marc Pantel*

Résumé

ERLANG est un langage fonctionnel non typé concurrent et distribué conçu par ERICSSON pour la programmation de leurs équipements de télécommunication. Il hérite de nombreux concepts de Prolog et de LISP et intègre des mécanismes sophistiqués pour la programmation d'applications distribuées. L'absence de mécanisme de typage, c'est-à-dire de vérification statique de certaines formes de correction de programmes, complique la tâche des programmeurs. Nous présentons dans cet article les travaux que nous menons pour munir ERLANG d'un système de type. Celui-ci est dérivé de travaux précédents réalisés dans le cadre des acteurs. Ils ont permis la mise au point d'un prototype de système de type pour ERLANG. Celui-ci consiste en l'inférence d'un type pour chaque programme et en la vérification de la correction de ce programme par la résolution des contraintes collectées par le système d'inférence.

Mots-clé :

Programmation Distribuée, ERLANG, Typage

1 Préalables

Face au développement fulgurant des réseaux, les programmeurs 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.

En effet, la multiplication des contextes d'utilisations des machines informatiques provoque l'extension continue des logiciels par de nouvelles fonctionnalités, de nouvelles interfaces... Ceci conduit à une croissance exponentielle de la quantité de code à développer. A titre d'exemple, Windows 2000, le système d'exploitation de Microsoft a mobilisé 5 000 programmeurs pour mettre au point ses dix millions de lignes de code¹. De tels ordres de grandeur rendent 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, un logiciel doit savoir s'adapter à des supports informatiques très divers (Carte à puce, PC ...). Il se doit donc d'être souple et polyvalent 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.

*Equipe VESTALE, ENSEEIHT-IRIT-FERIA/SVF, 2 rue Camichel, 31071 TOULOUSE FRANCE, Marc.Pantel@enseeiht.fr

1. Les chiffres cités proviennent de communiqués publiés sur le site de Microsoft : <http://www.microsoft.com>.

De plus, vu le rôle critique généralement joué 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 événements courants qui ne peuvent donc pas être ignorées lors du développement d'une application répartie.

Enfin, 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.

Les arguments précédemment évoqués montrent que la certification *a priori* des logiciels est indispensable car elle ne peut plus reposer uniquement sur un développement avec des tests *a posteriori*. Toutes les phases du processus de construction d'une application devront être certifiées formellement de manière automatique ou assistée, c'est-à-dire en utilisant des logiciels pour vérifier que le résultat du développement en cours possède certaines propriétés². 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.

Pour contribuer à la construction d'outils de certification du développement, nous procédons à une approximation des programmes par des analyses statiques à base d'inférence de types. Historiquement, nous avons développé des techniques sophistiquées de typage sur un calcul de processus (CAP [CPS96]) puis sur un langage académique (ML-ACT [DPCS00]). Il nous a semblé alors intéressant de les adapter à ERLANG, un langage à la fois industriel et suffisamment abstrait (du point de vue de la répartition et de la communication) pour fournir un bon support d'expérimentation. Ainsi, l'importante quantité de code opérationnel existant fournit une base intéressante d'évaluation de nos systèmes.

Dans cet article, nous commençons par introduire ERLANG, à partir de quelques exemples de programmes. Puis, nous introduisons le cadre général que nous avons conçu pour la construction de systèmes de type pour les langages d'acteurs en précisant comment il s'adapte à ERLANG. Nous présentons ainsi une sémantique formelle pour une version simplifiée d'ERLANG. Ensuite, nous donnons un aperçu du fonctionnement du prototype d'analyseur réalisé sur ces bases. Enfin, nous concluons par une discussion sur ce que nos travaux permettent d'envisager et ainsi nous donnons une ébauche de réponse à la question posée par le titre de ce papier.

2 Erlang

Le langage ERLANG est un langage de programmation fonctionnel, concurrent et réparti conçu par ERICSSON dans le but de programmer 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.

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

La présentation que nous allons effectuer du langage sera rapide et incomplète. Nous renvoyons le lecteur souhaitant de plus amples informations sur ERLANG au livre de Armstrong et al. [AVWW96], ainsi qu'au manuel de référence du langage [BV99] et enfin au site internet de la version libre d'ERLANG : <http://www.erlang.org>.

Un langage fonctionnel moderne 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). Cet héritage ne l'empêche pas d'être un langage fonctionnel moderne qui a su intégrer les avancées récentes dans le domaine des langages de programmation. 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* (i.e. une fois définie, il n'est pas possible d'affecter une variable). Simultanément à l'exécution d'un programme, un *ramasse-miettes* libère la mémoire occupée par des variables qui ne serviront plus. L'introduction de variable est réalisée par un mécanisme de filtrage fortement inspiré de ML mais conserve de Prolog trois caractéristiques :

- il est hétérogène, les différents filtres peuvent être de formes totalement différentes (le typage est souple et dynamique en ERLANG).
- il 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.
- il peut être dynamique, si une variable figurant dans le motif est déjà définie dans le contexte courant, alors sa valeur est utilisée comme motif. Les occurrences de variables dans les motifs sont donc non liantes, ce qui permet de réaliser des filtres dynamiques (par exemple dans le corps d'une fonction).

L'identité d'une fonction se compose de son nom (un atome) et de son arité. Comme l'illustre le petit exemple de calculatrice ci-dessous, les fonctions sont définies par filtrage :

```
calculate(plus, {N, N}) -> 2 * N;
calculate(plus, {N1, N2}) -> N1 + N2;
calculate(moins, {N1, N2}) -> N1 - N2;
calculate(plus, []) -> 0;
calculate(plus, [N|L]) -> N + calculate(plus, L);
calculate(plus, N) -> N;
calculate(moins, N) -> -N.
```

La fonction `calculate` ainsi définie est d'arité 2, le premier argument est un atome qui permet de choisir l'opération réalisée (addition ou soustraction) et le second contient des opérandes pouvant prendre trois formes : une variable (opérations unaires), un couple (opérations binaires) ou une liste (opérations n-aires). Notons que la 1^{re} branche impose aux deux membres du couple d'être égaux et que la 5^e branche montre que la fonction peut être récursive. Lors d'un appel de fonction, le choix de la fonction se fait simultanément sur le nom et sur le nombre de paramètres. Ainsi, la fonction suivante peut être définie³ :

```
calculate() -> Op = get_integer("Quel opérateur ? (1 pour +, 2 pour -)"),
              Ope = case Op of 1 -> plus; 2 -> moins end,
              calculate(Ope, get_operand()).
```

3. On suppose que sont définies les fonctions `get_integer` qui affiche son argument et renvoie l'entier entrée en réponse et `get_operand` qui se charge de récupérer l'argument.

Notons que la deuxième variable (**Op**) définie ne peut pas s'appeler **Op**, sinon, comme le filtrage est dynamique, elle serait remplacée par la valeur effective de **Op** (1 ou 2) et l'opérateur = réalise le filtrage de cet entier par le résultat du **case** (**plus** ou **moins**) et produit donc une erreur signalée par une exception.

Un langage concurrent réparti ERLANG intègre la concurrence et la répartition, non pas pour augmenter ses capacités de calcul, mais dans le but de décrire explicitement le comportement concurrent et réparti naturel des applications de télécommunication.

La sémantique qui est associée aux processus Erlang est très proche de celle des acteurs (voir [Agh86]). ERLANG est donc un langage fonctionnel concurrent et réparti non typé qui peut être rapproché de *Plasma* et de ses extensions (voir [MMS88]). 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 point à point et est garantie. De plus, deux messages envoyés en séquence par un processus à un même processus arrivent dans cet ordre.

L'exemple ci-dessous définit une fonction **timeout** qui crée un processus (via **spawn**) jouant le rôle d'un *timer*. Cette fonction reçoit un temps et une alarme et renvoie l'adresse du processus timer. L'utilisateur peut alors appeler la fonction **cancel** en lui fournissant cette adresse pour arrêter le timer. Le corps de cette fonction envoie (opérateur !) alors un message **cancel** au timer. Notons que dans les deux fonctions précédentes utilisent une fonction intégrée à ERLANG (*built in function* ou *bif* en anglais) qui renvoie l'adresse du processus englobant (**self**). Enfin, le timer est bloqué en attente (via **receive**):

- de la réception d'un message d'annulation (**{cancel,Pid}**) qui provoque la fin du processus,
- de l'expiration du temps (**after Time**) auquel cas le message **Alarm** est envoyé au processus qui avait créé le timer.

```
timeout(Time, Alarm) -> spawn(timer, [self(),Time,Alarm]).
cancel(Timer) -> Timer ! {cancel,self()}.
timer(Pid, Time, Alarm) -> receive {cancel,Pid} -> true
                             after Time -> Pid ! Alarm
                             end.
```

Remarquons que le filtrage étant dynamique en ERLANG, dans le filtre du **receive**, **Pid** est remplacé par l'adresse du processus ayant initialisé le timer permettant ainsi de limiter l'annulation du timer au processus l'ayant créé.

Enfin, notons que pour obtenir une version répartie du timer, il suffit d'utiliser la version répartie de **spawn** qui compte un argument de plus, le nœud sur lequel le processus doit être lancé. Le support d'exécution se chargeant d'éventuellement transférer le code nécessaire sur le nœud cible et d'y faire suivre tous les messages à destination du processus.

Un langage industriel ERLANG est exploité pour programmer des équipements de l'industrie des télécommunications, la robustesse a donc été un objectif primordial dans sa conception. Pour cela, il possède un puissant *système de gestion des erreurs* composé d'un *support d'exécution* et d'une notion d'exceptions tous deux distribués. Ils permettent de bien maîtriser les erreurs et leurs propagations. De plus, 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

```

programme ::= [ATOM '(' [motifs] ') 'when' expr '->' expr
              [';' ATOM '(' [motifs] ') 'when' expr '->' expr]* '.' ]+
motifs     ::= motif [';' motif]*
motif      ::= '_' | VAR | ATOM | INT | '{' [motifs] '}' | '[' [motifs ['|' motif]] ']'
exprs      ::= expr [';' expr]*
expr       ::= VAR | ATOM | INT | '{' [exprs] '}' | '[' [exprs ['|' expr]] ']'
              | '(' expr ')' | expr ',' expr | expr '!' expr | expr '(' [exprs] ')'
              | 'case' expr 'of' filtrages 'end' | 'receive' filtrages 'end'
filtrages  ::= motif 'when' expr '->' expr [';' motif 'when' expr '->' expr]*

```

FIG. 1 – La grammaire de μ Erlang

d'utilisation d'ERLANG nécessite une gestion du temps rigoureuse. Ainsi, c'est un langage temps réel qui contient une notion interne de temps que nous n'aborderons pas dans le système de type.

Le langage, pour développer des applications complexes, utilise un *système de module* simple et permet une communication aisée avec des programmes écrits dans d'autres langages. En fait, une application *réelle* est généralement composée de trois parties : le bas niveau généralement temps réel codé en C, la partie logique et algorithmique de l'application écrite en ERLANG et 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, on peut citer le commutateur ATM AXD 301 (voir [BR98]) et une application pour PABX (le *mobility server*) dont le code totalise 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 fax ou même un téléphone portable), le serveur se chargeant de transmettre tout appel à l'équipement adéquat. Ce produit ainsi que deux autres de taille nettement moindre sont décrits dans [Arm96]. Au delà d'ERICSSON, de plus en plus de projets sont développés autour d'ERLANG. On peut par exemple citer le travail de BLUETAIL⁴ qui commercialise des logiciels dans le domaine des produits Internet sécurisés.

μ Erlang une version simplifiée d'ERLANG Pour exprimer la sémantique et les systèmes de type des langages, il est de coutume de les simplifier en supprimant toutes les constructions qui facilitent la vie du programmeur mais peuvent s'exprimer en termes d'autres constructions (le sucre syntaxique). De plus, certaines constructions du langage se typent de manière totalement orthogonale à notre travail et sont donc ignorées (par exemple, les exceptions ou les enregistrements). Enfin, la composante temps réel (la partie **after** du **receive**) qui pose des problèmes importants lors de la construction d'une sémantique pour ERLANG n'est pas prise en compte.

La grammaire de μ Erlang est fournie dans la figure 1. Les terminaux sont composés des atomes (ATOM), des variables (VAR) et des entiers (INT). Un programme est composé d'une suite de définitions de fonctions⁵, qui sont composées de clauses séparées par ';' et terminées par '.'. Les clauses doivent référencer le même nom de fonction et avoir la même arité. Notons que les gardes sont obligatoirement présentes (sachant qu'une garde valant **true** est triviale) et peuvent être constituées d'expressions quelconques (ce qui n'est pas le cas en ERLANG⁶). Un motif peut être un joker (il filtre tout), une variable (il filtre tout et définit la variable si elle n'existe pas ou filtre par le contenu de la variable sinon), une constante, un tuple ou une liste. Symétriquement, une expression peut être une variable, une constante, un tuple ou une liste. Les expressions

4. <http://www.bluetail.com>

5. Nous supposons que l'une d'elles porte le nom **main** et est lancée à l'exécution du programme.

6. Seulement des tests de type et des comparaisons simples sont autorisées.

peuvent être également groupées par des parenthèses. Les opérateurs de séquençement et d’envoi de message sont respectivement `,` et `!`. L’appel de fonction doit fournir tous les arguments, il n’existe pas d’application partielle en ERLANG. Le choix `case` filtre une expression en utilisant un ensemble de filtres qui sont composés d’un motif, d’une garde et d’une séquence d’expressions (qui sera évaluée si le filtrage réussit). La réception `receive` fonctionne sur le même principe mais utilise la boîte aux lettres du processus courant, au lieu de l’expression.

3 Sémantique

Nos travaux portent sur l’analyse statique de programmes, cependant pour démontrer la correction de notre système de type, il faut disposer 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 (ils ont également pour but la vérification statique) :

- La première sémantique présentée dans [DkF98] est conçue dans le but de démontrer la validité d’une analyse par *model checking* au moyen d’une logique temporelle. Il s’agit d’une sémantique opérationnelle définie par un système de transition étiqueté fortement inspirée de celle de CCS et du π -calcul. La sémantique obtenue est complexe et ne se prête pas bien à la preuve de correction d’un système de type. Notons également que la communication y est réalisée par synchronisation entre l’envoyeur et la queue du destinataire et que les règles de portée des identificateurs sont modifiées.
- La seconde sémantique définie dans [Huc99], est rendue finie par une interprétation abstraite et ainsi permet de vérifier un programme sur la base d’une logique LTL par *model checking*. La sémantique introduite est également un système de transition étiqueté qui même si elle est nettement plus simple que la précédente, introduit les mêmes modifications de la sémantique 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. Nous avons construit un cadre général, les *configurations*, qui permet d’exprimer la sémantique et le typage de langage suivant une sémantique concurrente proche du modèle des acteurs. Pour cela, nous définissons une syntaxe générale décrivant la partie concurrente d’une application. Celle-ci abstrait (au sens de prendre en argument) la partie fonctionnelle du langage étudié. Ainsi, la partie fonctionnelle réutilise des sémantiques et des techniques de typage usuelles dans le domaine des langages fonctionnels non concurrents. La sémantique d’ERLANG est alors obtenue par instantiation de ce cadre en fournissant une sémantique de son noyau fonctionnel. Notons que la confrontation avec les deux sémantiques précédentes a permis de valider notre sémantique des configurations.

Nous ne donnerons pas toutes les définitions ou toutes les justifications qui sont détaillées dans [Dag01]. Nous allons uniquement présenter rapidement les configurations et en déduire la sémantique d’ERLANG.

Configuration Une configuration est un terme qui représente un système concurrent à un instant donné. Sa définition est paramétrée par trois ensembles : l’ensemble des noms (notés $a \in \mathbb{A}$), celui des messages (notés $m \in \mathcal{Mess}$) et celui des expressions (notés $e \in \mathcal{Exp}$). Leur ensemble est noté \mathcal{W} et elles sont construites par la grammaire suivante :

$$w ::= \epsilon \mid \mathbf{Err} \mid \nu a.w \mid w \parallel w \mid a \triangleleft m \mid \alpha \triangleright e \quad \text{et} \quad \alpha ::= \star \mid \langle a \mid \tilde{m} \rangle$$

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 (**Err**), un message m en transit dans le

médium vers a ($a \triangleleft m$) ou un processus de nom α exécutant e ($\alpha \triangleright e$). L'identité d'un processus peut être \star , s'il est *anonyme*, ou $\langle a | \tilde{m} \rangle$, si son nom est a et sa boîte aux lettres est \tilde{m} . Un processus anonyme code une expression globale (*oplevel*) qui correspond au démarrage du programme (e ne pourra pas contenir de **receive** ou de **self**). Enfin, la création d'un nouveau nom (donc d'un nouveau processus) se fait par un opérateur de restriction⁷ de portée ($\nu a.w$).

Remarquons que les ensembles \mathbb{A} , \mathcal{Mess} et \mathcal{Exp} doivent vérifier $\mathbb{A} \subset \mathcal{Exp}$ et $\mathcal{Mess} \subset \mathcal{Exp}$. Dans le cadre de μErlang , \mathcal{Exp} correspond à la syntaxe présentée Fig 1, les adresses (\mathbb{A}) sont créées automatiquement par appel de la fonction prédéfinie **spawn** et les messages peuvent être n'importe quelle valeur sémantique de μErlang (soit des atomes, des entiers, des chaînes de caractères, des listes ou des tuples).

De nombreuses configurations ont le même sens, ce qui s'exprime par la définition d'une relation de congruence :

- (W, \parallel, ϵ) est un monoïde commutatif, l'ordre des sous-configurations n'est donc pas important et on peut supprimer toutes les occurrences de ϵ .
- $w \parallel \mathbf{Err} \equiv \mathbf{Err}$, les erreurs sont propagées jusqu'à l'arrêt de tous les calculs.
- $\nu a.w \equiv w$ si a n'est pas libre dans w , $\nu a.w \equiv \nu b.[b/a]w$ si b n'est pas libre dans w et $\nu a_1.\nu a_2.w \equiv \nu a_2.\nu a_1.w$; ces trois propriétés usuelles permettent, de supprimer la liaison de noms qui ne sont plus référencés, de changer un nom lié et de modifier l'ordre des restrictions.
- $\nu a.w_1 \parallel w_2 \equiv \nu a.(w_1 \parallel w_2)$ si a n'est pas libre dans w_2 , ce qui permet d'étendre la portée d'un nom. Combinée avec la règle précédente elle permet toujours (moyennant un renommage de a dans w_1) d'étendre la portée et simule donc la propagation de nom dans le médium.
- $\star \triangleright v \equiv \epsilon$ et $\nu a.(\langle a | \emptyset \rangle \triangleright v) \equiv \epsilon$ si v est une valeur sémantique (elle ne peut alors plus être réduite; ainsi, un calcul global (ou un acteur) qui aboutit à une valeur peut être détruit par le ramasse-miettes. Notons que l'acteur doit avoir une boîte aux lettres vide et ne plus être connu dans le médium (sinon il pourrait recevoir d'autres messages).

Remarquons qu'il est également possible d'ajouter une règle signalant une erreur si un acteur est bloqué en attente d'un message, s'il n'est plus connu et s'il ne comprend aucun des messages présent dans sa boîte aux lettres. Cependant, comme notre système de type ne pourra pas capturer tous ces messages, nous ne pourrions montrer sa correction si nous ajoutons cette règle.

Pour commencer, la réduction d'un terme des configurations est faite modulo la congruence précédemment définie et peut se dérouler sur les sous-termes d'une mise en parallèle ou d'une restriction :

$$\begin{array}{ccc}
\text{CONG :} & \text{PAR :} & \text{RES :} \\
\frac{w_1 \equiv w'_1 \quad w'_1 \longrightarrow w'_2 \quad w'_2 \equiv w_2}{w_1 \longrightarrow w_2} & \frac{w_1 \longrightarrow w_2}{w \parallel w_1 \longrightarrow w \parallel w_2} & \frac{w_1 \longrightarrow w_2}{\nu a.w_1 \longrightarrow \nu a.w_2}
\end{array}$$

La règle de réduction d'une communication suppose l'existence d'un prédicat $\mathcal{P}(m,e)$ qui détermine si un message m pourra éventuellement être traité par une expression e . Si ce prédicat est faux lors d'une réception, une erreur est levée. La sémantique est ainsi paramétrée par \mathcal{P} qui peut avoir une précision variable. Ainsi, dans les modèles initiaux d'acteurs ([Agh86]) ou dans *Plasma II*, ce prédicat vérifie si m est traité par le prochain **receive** de e . A l'opposé, **ERLANG** ou le join calculus ([FGL⁺96]) utilisent un \mathcal{P} trivial (toujours vrai). Nous pensons cependant que ces deux approches ne sont pas idéales car, la première est trop rigide (il faut quasiment

7. L'opérateur ν est donc un lieu vis-à-vis des adresses, ce qui permet de définir la notion de nom libre ainsi que la notion de substitution associée.

systématiquement se synchroniser avant l'envoi d'un message) et la seconde est trop lâche (elle permet l'engorgement des boîtes aux lettres par des messages qui ne seront jamais traités). Dans [Dag01], nous définissons une notion de potentiel qui est une abstraction du typage qui permet d'éliminer les messages qui seront *clairement* jamais traités. Par exemple, un timer (voir page 4) ne saura jamais traiter de message **restart**...

$$\begin{array}{c} \text{RCV:} \\ \frac{\mathcal{P}(m,e)}{\langle a | \tilde{m} \rangle \triangleright e \parallel a \triangleleft m \longrightarrow \langle a | m \tilde{m} \rangle \triangleright e} \end{array} \qquad \begin{array}{c} \text{RCvE:} \\ \frac{\text{not}(\mathcal{P}(m,e))}{\langle a | \tilde{m} \rangle \triangleright e \parallel a \triangleleft m \longrightarrow \mathbf{Err}} \end{array}$$

Notre sémantique générale inclue, enfin, les règles spécifiant l'interaction du calcul fonctionnel avec le calcul concurrent. Pour cela, nous supposons que la réduction fonctionnelle aura la forme :

$$N \vdash \alpha, e \xrightarrow{l} \alpha', e' \quad \text{or} \quad N \vdash \alpha, e \longrightarrow_e \mathbf{Err}$$

où N est un ensemble de nom, α et α' sont des identités (soit \star ou $\langle a | \tilde{m} \rangle$), l est une étiquette de réduction et e et e' sont des expressions. Les étiquettes (\mathbb{L}) sont construites par la grammaire suivante :

$$l ::= \epsilon \mid \text{PAR}(w) \mid \text{NEW}(a, e)$$

Les étiquettes vides (ϵ) signifie que la réduction fonctionnelle n'a pas d'influence concurrente (l'étiquette ne sera pas précisée). Une étiquette **PAR** indique que la réduction a pour effet l'ajout d'une configuration en parallèle avec l'activité englobante. Enfin, une étiquette **NEW** signifie que la réduction a pour effet la création d'un processus (qui doit être restreint).

$$\begin{array}{c} \text{EXPE:} \\ \frac{\mathcal{FN}(\alpha \triangleright e) \vdash \alpha, e \longrightarrow_e \mathbf{Err}}{\alpha \triangleright e \longrightarrow \mathbf{Err}} \end{array} \qquad \begin{array}{c} \text{EXP:} \\ \frac{\mathcal{FN}(\alpha \triangleright e) \vdash \alpha, e \longrightarrow_e \alpha', e'}{\alpha \triangleright e \longrightarrow \alpha' \triangleright e'} \end{array}$$

$$\begin{array}{c} \text{PAR:} \\ \frac{\mathcal{FN}(\alpha \triangleright e) \vdash \alpha, e \xrightarrow{\text{PAR}(w)} \alpha', e'}{\alpha \triangleright e \longrightarrow \alpha' \triangleright e' \parallel w} \end{array} \qquad \begin{array}{c} \text{NEW:} \\ \frac{\mathcal{FN}(\alpha \triangleright e) \vdash \alpha, e \xrightarrow{\text{NEW}(a, e'')} \alpha', e'}{\alpha \triangleright e \longrightarrow \nu a. (\alpha' \triangleright e' \parallel \langle a | \emptyset \rangle \triangleright e'')} \end{array}$$

Réduction fonctionnelle Un programme μ Erlang est un ensemble de définitions de fonctions et une fonction **main**. L'exécution d'un tel programme correspond à la réduction du corps de la fonction **main** dans un contexte où les fonctions déclarées sont définies. La première étape de la sémantique fonctionnelle consiste donc à construire un environnement (que nous notons \mathcal{F}) avec ces fonctions. Dans \mathcal{F} , un couple composé d'un atome (nom de la fonction) et de son arité permet de retrouver le filtrage corps de la fonction. L'expression corps de la fonction **main** est alors indexée par cet environnement : $e_{\mathcal{F}}$. Pour simplifier les règles de la sémantique fonctionnelle, nous ne ferons apparaître l'indice d'une expression que dans les règles décrivant l'appel de fonction.

La réduction fonctionnelle utilise la notion usuelle de contexte d'évaluation. Un contexte d'évaluation noté $C[]$ est une expression comportant un trou qui indique la sous-expression qui est en cours d'évaluation. Ainsi, la réduction de $C[e] \longrightarrow_e C[e']$ réduit l'expression e et la remplace dans le contexte par le résultat de cette réduction e' . Les contextes d'évaluation sont les termes construits par la grammaire suivante ne contenant qu'un seul trou :

$$\begin{array}{l} C ::= [] \mid (C) \mid \{A\} \mid [A]e \mid [v, \dots, v]C \mid C, e \mid C!e \mid e!C \mid C(e, \dots, e) \mid e(A) \\ \quad \mid \text{case } C \text{ of } f \text{ end} \mid \text{spawn}(C, e) \mid \text{spawn}(e, C) \\ A ::= [] \mid e, A \mid A, e \end{array}$$

Les variables, une fois définies, voient leurs valeurs propagées par une substitution. Cette substitution se comporte de manière habituelle et ne sera donc pas détaillée ici. La seule particularité concerne le filtrage dynamique qui impose la substitution également dans les motifs de filtrage. Ces substitutions seront notées σ . Le filtrage utilise l'opérateur $/$ défini à partir d'une fonction **filt** que nous ne détaillerons pas qui compare un motif et une valeur et construit la substitution des variables du motif par les composantes de la valeur auxquelles elles correspondent. Cette fonction renvoie donc une substitution ou échoue (**fail**).

$$\left\{ \begin{array}{ll} v/[] \triangleq \mathbf{Err} & \\ v/p \text{ when } g \rightarrow e :: f \triangleq v/f & \text{si } \mathbf{filt}(p, v) = \mathbf{fail} \\ v/p \text{ when } g \rightarrow e :: f \triangleq v/f & \text{si } \mathbf{filt}(p, v) = \sigma \text{ et } \sigma(g) \rightarrow_e \mathbf{false} \\ v/p \text{ when } g \rightarrow e :: f \triangleq \sigma, e & \text{si } \mathbf{filt}(p, v) = \sigma \text{ et } \sigma(g) \rightarrow_e \mathbf{true} \end{array} \right.$$

Notons que l'évaluation de la garde est notée de façon simplifiée ci-dessus mais suit le processus d'évaluation des autres expressions μ Erlang.

Présentons tout d'abord les sept règles décrivant la sémantique purement fonctionnelle :

$$\begin{array}{lll} \text{VARE:} & \text{SEQ:} & \text{APPE1:} \\ N \vdash \alpha, C[x] \rightarrow_e \mathbf{Err} & N \vdash \alpha, C[v, e] \rightarrow_e \alpha, C[e] & \frac{(v, n) \notin \text{dom}(\mathcal{F})}{N \vdash \alpha, C[v(v_1, \dots, v_n)]_{\mathcal{F}} \rightarrow_e \mathbf{Err}} \\ \\ \text{APPE2:} & & \text{APP:} \\ \frac{f = \mathcal{F}(v, n) \quad \{v_1, \dots, v_n\}/f = \mathbf{Err}}{N \vdash \alpha, C[v(v_1, \dots, v_n)]_{\mathcal{F}} \rightarrow_e \mathbf{Err}} & & \frac{f = \mathcal{F}(v, n) \quad \{v_1, \dots, v_n\}/f = \sigma, e}{N \vdash \alpha, C[v(v_1, \dots, v_n)]_{\mathcal{F}} \rightarrow_e \alpha, C[\sigma(e)]_{\mathcal{F}}} \\ \\ \text{CASEE:} & & \text{CASE:} \\ \frac{v/f = \mathbf{Err}}{N \vdash \alpha, C[\text{case } v \text{ of } f \text{ end}] \rightarrow_e \mathbf{Err}} & & \frac{v/f = \sigma, e}{N \vdash \alpha, C[\text{case } v \text{ of } f \text{ end}] \rightarrow_e \alpha, \sigma(C[e])} \end{array}$$

Comme toutes les variables définies sont substituées, aucune variable ne peut plus figurer dans une expression sans provoquer une erreur (VARE). Une valeur en séquence avec une expression disparaît (SEQ). Une application provoque une erreur si, soit la fonction appelée n'existe pas (APPE1), soit si aucun filtre du corps de la fonction ne correspond aux arguments (APPE2). Si le filtrage réussit, les variables liées lors du filtrage sont substituées dans le corps de la fonction. Remarquons que la liste des arguments peut être vide, alors n vaut 0. Notons que ces définitions ne se propagent pas dans le contexte englobant contrairement au filtrage du choix. Enfin, le choix suit une sémantique proche de l'application. Les deux différences sont : la présence du filtrage dans le contexte et donc la substitution des variables des filtres déjà définies dans le contexte, et, la propagation des variables définies par le filtrage au contexte. Notons que, cette sémantique n'est pas sûre puisque certaines branches peuvent introduire des variables qui ne le sont pas dans les autres branches, provoquant éventuellement une erreur à l'exécution. Notre système de type vérifiera que le programme respecte la contrainte imposée par le compilateur ERLANG : ne sont propagées que les variables qui sont définies dans toutes les branches.

Enfin, la sémantique des constructions concurrentes suit les 8 règles suivantes :

$$\begin{array}{c}
\text{SENDE :} \\
\frac{v_1 \notin \mathbb{A}}{N \vdash \alpha, C[v_1!v_2] \longrightarrow_e \mathbf{Err}} \\
\\
\text{SELF E :} \\
N \vdash \star, C[\mathbf{self}()] \longrightarrow_e \mathbf{Err} \\
\\
\text{SPAWNE :} \\
\frac{v' \text{ is not a list}}{N \vdash \alpha, C[\mathbf{spawn}(v,v')] \longrightarrow_e \mathbf{Err}} \\
\\
\text{RECEIVE E :} \\
N \vdash \star, C[\mathbf{receive } f \text{ end}] \longrightarrow_e \mathbf{Err}
\end{array}
\qquad
\begin{array}{c}
\text{SEND :} \\
\frac{v_1 \in \mathbb{A}}{N \vdash \alpha, C[v_1!v_2] \xrightarrow{\text{PAR}(v_1 \triangleleft v_2)}_e \alpha, C[v_2]} \\
\\
\text{SELF :} \\
N \vdash \langle a | \tilde{m} \rangle, C[\mathbf{self}()] \longrightarrow_e \langle a | \tilde{m} \rangle, C[a] \\
\\
\text{SPAWN :} \\
\frac{a \in \mathbb{A} \setminus N}{N \vdash \alpha, C[\mathbf{spawn}(v,[v_1,\dots,v_n])] \xrightarrow{\text{NEW}(a,v(v_1,\dots,v_n))}_e \alpha, C[a]} \\
\\
\text{RECEIVE :} \\
\frac{\tilde{m}/f = \tilde{m}', \sigma, e}{N \vdash \langle a | \tilde{m} \rangle, C[\mathbf{receive } f \text{ end}] \longrightarrow_e \langle a | \tilde{m}' \rangle, \sigma(C[e])}
\end{array}$$

Les quatre règles de droite décrivent les erreurs possibles : respectivement, l'envoi d'un message à une valeur qui n'est pas un nom ; **self** et **receive** ne peuvent pas apparaître dans les expressions globales et enfin, le deuxième argument d'un **spawn** doit être une liste. Si ces conditions sont vérifiées : respectivement un envoi résulte en la valeur envoyée ; une référence à **self** en l'adresse du processus courant et une création de processus renvoi l'adresse créée. Notons que lors d'un envoi ou d'une création, un terme concurrent (l'étiquette de la réduction) est ajouté à la configuration englobante. Enfin, la réception suit une sémantique proche du choix excepté le fait qu'elle modifie la boîte aux lettres.

La sémantique du traitement des messages dans la boîte aux lettres est originale en ERLANG puisqu'elle donne une priorité aux filtres définis en premier. Ainsi, elle tente de trouver le premier message de la boîte aux lettres qui est filtré par le premier filtre. Si elle n'en trouve pas, elle essaie avec le second filtre et ainsi de suite. Soit :

$$\frac{\exists j \quad (\forall i < j \quad m_i/f_1 = \mathbf{Err}) \quad m_j/f_1 = \sigma, e}{(m_i)_{i \in J}/f_1 :: \mathfrak{fl} = (m_i)_{i \in J \setminus \{j\}}, \sigma, e}
\qquad
\frac{(\forall i \in J \quad m_i/f_1 = \mathbf{Err}) \quad (m_i)_{i \in J}/\mathfrak{fl} = q, \sigma, e}{(m_i)_{i \in J}/f_1 :: \mathfrak{fl} = q, \sigma, e}$$

Remarquons que si aucun filtrage n'est possible ou si la boîte aux lettres est vide aucune réduction n'est possible ce qui bloquera le processus.

4 Typage

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

- les *erreurs fonctionnelles*, usuelles des langages séquentiels (qu'ils soient fonctionnels ou impératifs), une valeur est manipulée dans un contexte erroné.
- les *erreurs concurrentes*, la mauvaise utilisation d'une interface de communication d'un processus. C'est-à-dire, un message est envoyé à un processus qui ne saura jamais le traiter. Ce sont des erreurs de non respect des protocoles de communication entre entités (notion de *message non compris* (*message not understood*) des langages objets).

Dans le contexte d'ERLANG où un processus présente des interfaces multiples, le problème est beaucoup plus complexe que les message non compris. En effet, l'ensemble des messages compris évolue dans le temps et ne peut pas être construit statiquement de manière exacte. En fait, notre

système de type va approximer cette évolution. 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 pourra le traiter plus tard. Pour simplifier le problème, la plupart des systèmes actuels ont adopté l’une des deux politiques drastiques vis-à-vis de ces messages présentées lors de la discussion du potentiel page 7 (le rejet immédiat ou l’acceptation continue). Jugeant ces deux approches respectivement trop restrictive et trop permissive, 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 que nous appelons messages orphelins. Remarquons qu’il existe une sous-classe que nous appelons messages orphelins triviaux (messages jamais compris), l’ensemble des messages qui ne figure dans aucune interface (**receive**) que le processus pourrait exécuter. Ces messages sont considérés comme des erreurs de type et les programmes en contenant sont rejetés (le prédicat \mathcal{P} calcule un sous-ensemble de ces orphelins triviaux).

Un système de type pour ERLANG (comme pour tout langage) peut comporter plusieurs niveaux d’analyse. Deux prototypes de systèmes ont été mis au point pour ERLANG (voir [MW97] et [Lin96]), se concentrant sur la partie purement fonctionnelle du langage et en simplifiant fortement la sémantique (surtout pour le premier). Notre ambition est de construire un système réellement utilisable pour des programmes ERLANG et analysant également la partie concurrente des programmes. Comme, nous utilisons des techniques assez proches de collecte et de résolution de contraintes, nos travaux peuvent être considérés comme une extension de ces systèmes.

Les systèmes de type que nous utilisons sont basés sur une technique d’inférence qui permet de reconstruire les types des différentes entités du programme. Pour cela, un parcours du programme affecte des variables de type à chaque terme et collecte les contraintes qui relient ces différentes variables. En fin de collecte, un outil de résolution va déterminer si l’ensemble des contraintes collectées possède une solution. Si tel est le cas, le programme est déclaré bien typé. Remarquons que le système de type de ML peut s’exprimer comme une collecte de contraintes d’égalité entre types. Lorsque l’on combine cette technique d’inférence avec du sous-typage, les contraintes deviennent plus complexes. Leur résolution exploite alors de puissants algorithmes de parcours de graphes (voir [Pot98]).

Comme le typage d’ERLANG est très complexe (contrairement à sa sémantique), nous ne donnerons pas le système complet mais discuterons certains aspects remarquables (le lecteur intéressé pourra consulter [Dag01]).

Filtrage hétérogène Les filtrages peuvent être hétérogènes et la politique de filtrage usuelle adoptée par ML, d’unification du type des différents filtres $(\alpha_1 \rightarrow \alpha_2) \sqcup (\beta_1 \rightarrow \beta_2) = (\alpha_1 \sqcap \beta_1) \rightarrow (\alpha_2 \sqcup \beta_2)$ n’est pas acceptable. Pour cela, A. Aiken et al. utilisent un type dit *conditionnel* qui approxime le filtrage dans le langage des types et ainsi, réintroduit une forme d’analyse de flot de contrôle dans les types. Ce type conditionnel a été introduit dans [AWL94] sous l’écriture $t_1 ? t_2$ et définit par :

$$t_1 ? t_2 \triangleq \begin{cases} t_1 & \text{si } t_2 \neq \perp \\ \perp & \text{sinon} \end{cases}$$

Le choix **case** e **of** **true** \rightarrow **1**; **false** \rightarrow **autre** est alors approximé par $(\text{int}?(t_e \sqcap \text{true})) \sqcup (\text{autre}?(t_e \sqcap \text{false}))$ si t_e est le type de e . Nous n’utilisons pas un type conditionnel mais une contrainte conditionnelle $c_1 \Rightarrow c_2$ (si c_1 est vérifiée alors c_2 doit l’être) qui est générée lors d’un

filtrage. Elle permet de relier précisément les contraintes sur le résultat au type de la donnée filtrée: $t'_1 \sqsubseteq t_1 \Rightarrow t_2 \sqsubseteq t'_2$ 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 le filtrage est correct. Le typage du choix précédent produit donc $C = \{t_e \sqsubseteq \mathbf{true} \Rightarrow \mathit{int} \sqsubseteq t_r, t_e \sqsubseteq \mathbf{false} \Rightarrow \mathbf{autre} \sqsubseteq t_r, t_e \sqsubseteq \mathbf{true} \sqcup \mathbf{false}\}$. Si la forme de t_e est connue, le système peut être simplifié, sinon il se décompose en deux systèmes (car t_e est composé de deux types):

- Si $t_e \sqsubseteq \mathbf{true}$ alors $C = \{t_e \sqsubseteq \mathbf{true}, \mathit{int} \sqsubseteq t_r\}$ (la 2^e contrainte disparaît et la 3^e est triviale).
- Si $t_e \sqsubseteq \mathbf{false}$ alors $C = \{t_e \sqsubseteq \mathbf{false}, \mathbf{autre} \sqsubseteq t_r\}$ (même raisonnement).

Le filtrage d'une donnée de type inconnu doit être correct pour tous ses types possibles, les deux systèmes doivent donc être solubles. Notons qu'afin d'attribuer des types suffisamment précis aux filtrages, nous considérons que chaque constante possède son propre type. Ainsi, par exemple, 1 est de type 1 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. Soit la fonction \mathbf{g} définie par: $\mathbf{g}(X) \rightarrow \mathbf{case\ 1\ of\ } X \rightarrow \mathbf{ok}; _ \rightarrow \mathbf{no\ end..}$. Alors, $\{\mathbf{g}(1), \mathbf{g}(2)\}$ se réduit en $\{\mathbf{case\ 1\ of\ } 1 \rightarrow \mathbf{ok}; \dots, \mathbf{case\ 1\ of\ } 2 \rightarrow \mathbf{ok}; \dots\}$ puis en $\{\mathbf{ok}, \mathbf{no}\}$. Or, le typage usuel du couple d'application conduit au type $(\mathbf{ok} \sqcup \mathbf{no}) \times (\mathbf{ok} \sqcup \mathbf{no})$ car les contraintes d'application sont $1 \sqsubseteq \alpha$ et $2 \sqsubseteq \alpha$ et le typage du corps de la fonction \mathbf{g} produit le type $\alpha \rightarrow t$ et les contraintes $\{1 \sqsubseteq \alpha \Rightarrow \mathbf{ok} \sqsubseteq t; 1 \sqsubseteq (\top \setminus \alpha) \Rightarrow \mathbf{no} \sqsubseteq t\}$.

En effet, 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 qui vérifient par transitivité la compatibilité ($t' \sqsubseteq t$). Or, si un filtre contient un argument, il va générer une contrainte $t \sqsubseteq \alpha$ incomparable avec $t' \sqsubseteq \alpha$.

Le type obtenu n'est pas très précis mais surtout le problème devient plus critique si le joker ne figure pas dans le filtrage. Car alors, le programme conduit à une erreur (1 n'est pas filtré par 2) qui ne peut pas être détectée par les contraintes obtenues $\{1 \sqsubseteq \alpha; \mathbf{ok} \sqsubseteq t; 1 \sqsubseteq \alpha; 2 \sqsubseteq \alpha\}$ qui possèdent des solutions. Pour résoudre ce problème, une phase d'analyse préalable au typage marque les occurrences de liaison des variables qui apparaissent également dans d'autres filtres (X devient X^* dans le motif de définition de \mathbf{g}). Une variable ainsi étiquetée a alors un type marqué α^* .

Intuitivement, le procédé adopté construit, pour chaque appel d'une telle fonction, une nouvelle instance de son type. En effet, le corps de la fonction dépend des arguments et est 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 instantiation 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 . L'égalité est alors utilisée sur ces variables instanciées. Cette approche est voisine du polymorphisme par nom [Ler92] (que nous limitons aux variables étiquetées). Par exemple, dans le programme ci-dessus, l'appel $\mathbf{g}(1)$ instancie le corps de \mathbf{g} en $\mathbf{case\ 1\ of\ } 1 \rightarrow \mathbf{ok}; \dots$ et celui de $\mathbf{g}(2)$ en $\mathbf{case\ 1\ of\ } 2 \rightarrow \mathbf{ok}; \dots$. Donc le type de \mathbf{g} qui utilise la variable étoilée devient respectivement $1 \rightarrow \mathbf{ok}$ et $2 \rightarrow \mathbf{ok}$.

De plus, le résultat d'une fonction dont un des arguments est étoilé doit être étoilé. En effet, chaque appel introduit des contraintes sur le résultat qui ne sont valables que pour cet appel. Et, il ne faut pas créer de contraintes entre ses différents résultats. Par exemple, la première version de la fonction \mathbf{g} conduit à \mathbf{ok} et à \mathbf{no} .

Le typage du programme sans joker attribue le type t_g à \mathbf{g} avec les contraintes $C = \{1 \sqsubseteq$

α ; $\text{ok} \sqsubseteq t$; $\alpha^* \rightarrow t^* \sqsubseteq t_g$. Ainsi, la première application instancie α et t par α_1 et t_1 et donc ajoute à C les contraintes $\{1 \sqsubseteq \alpha_1; \alpha_1 \rightarrow t_1 \sqsubseteq 1 \rightarrow \beta_1\}$ où β_1 est le type du résultat. Puis, si β_2 est le type du second résultat $C = C \cup \{2 \sqsubseteq \alpha_2; \alpha_2 \rightarrow t_2 \sqsubseteq 2 \rightarrow \beta_2\}$.

Le sous-typage sur les variables de type étoilées est transformé en égalité et C devient donc :

$$\begin{aligned} C &= \{1 \sqsubseteq \alpha; \text{ok} \sqsubseteq t; \alpha^* \rightarrow t^* \sqsubseteq t_g; 1 \sqsubseteq \alpha_1; \text{ok} \sqsubseteq t_1; 1 = \alpha_1; \alpha_1^* \rightarrow t_1^* \sqsubseteq 1 \rightarrow \beta_1; 1 \sqsubseteq \alpha_2; \\ &\quad \text{ok} \sqsubseteq t_2; 2 = \alpha_2; \alpha_2^* \rightarrow t_2^* \sqsubseteq 2 \rightarrow \beta_2\} \\ &= \{1 \sqsubseteq \alpha; \text{ok} \sqsubseteq t; \text{ok} \sqsubseteq t_1; t_1^* \sqsubseteq \beta_1; \boxed{1 \sqsubseteq 2}; \text{ok} \sqsubseteq t_2; t_2^* \sqsubseteq \beta_2\} \end{aligned}$$

C contient alors clairement une contradiction (la contrainte entourée) et l'erreur est donc détectée.

Forme et type des messages Une analyse automatique du code du compilateur, des bibliothèques standards et de programmes disponibles sur Internet⁸, 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 dont le premier élément est un atome. Ces atomes jouent le rôle d'étiquettes de message pour le programmeur. Accessoirement, la règle 5.7 issues des *règles de bonne programmation* ([WA96]) conseille d'étiqueter tous les messages. De plus, la seule forme de filtrage de réception qui ne contient pas d'étiquette et qui est (peu) employée 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 un autre processus. Ces programmes (rares) peuvent être converti simplement pour utiliser des étiquettes. 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.

Ces étiquettes ne sont pas des valeurs manipulables dans le langage à l'image des *labels d'enregistrement* en Caml. Nous exploitons donc une méthode similaire à celle utilisée dans Caml pour approximer les interfaces et les processus suivant le modèle des *rangées*. Initialement conçues pour typer des objets simples, elles ont été utilisées pour le typage des enregistrements extensibles (voir par exemple [Rém94]) mais sont maintenant utilisées dans de nombreux systèmes de type, on peut par exemple citer la détection des exceptions qui ne sont pas rattrapées ([Pes99]) ou le typage de la partie objet d'Objective-Caml (voir [RV98]). Pour nous, une rangée est une fonction partielle qui associe à une étiquette m le type des paramètres du message correspondant 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 processus 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.

Les rangées vérifient un grand nombre de propriétés et nous renvoyons le lecteur intéressé à [Rém94]. Entre autres, on peut extraire d'une rangée se **terminant par une variable** une étiquette qui n'y figure pas initialement. Cette dernière possibilité est appelée *l'expansion à la demande* et permet de comparer deux rangées ne contenant pas les mêmes étiquettes. Par exemple, pour comparer $\{m_1 : p_1, m_2 : p_2, i_1\}$ et $\{m_2 : p'_2, i_2\}$, il faut *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.

Ainsi, nous utilisons la notion de rangée pour typer les processus et les interfaces (le corps des **receive**). L'accès d'un processus 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. Chaque expression peut donc contenir un accès à la boîte aux lettres. Nous utilisons un calcul d'effets (indirects) inspiré de [TJ94] pour collecter ces différentes interfaces d'accès à la boîte aux lettres, chaque expression cumule ses interfaces dans le type de self. Celui-ci

8. La quantité de code analysée représente environ 200 000 lignes de code.

est ensuite stocké dans les types des fonctions : le type $t_1 \xrightarrow{I} t_2$ décrit une fonction ayant pour domaine t_1 , pour codomaine t_2 et pour self I . Ainsi, lors de la création d'un processus, ces effets sont ajoutés au type du processus.

L'analyse du code du timer de la page 4 produit les types :

$$\begin{aligned} \text{timeout} &: [int, \alpha] \xrightarrow{\text{Mess}(\alpha)} @\{\text{cancel} : \bullet @\text{Mess}(\alpha)\} \\ \text{cancel} &: [@\{\text{cancel} : \circ @\phi\}] \xrightarrow{\phi} \text{cancel} \times @\phi \\ \text{timer} &: [@\text{Mess}(\alpha), int, \alpha] \xrightarrow{\{\text{cancel} : \bullet @\text{Mess}(\alpha)\}} \text{true} \sqcup \alpha \end{aligned}$$

La fonction sur les types **Mess** permet de convertir un tuple message en rangée. Elle transforme un atome s en $\{s : \circ unit\}$, un tuple $s \times T_1 \times \dots \times T_n$ en $\{s : \circ(T_1 \times \dots \times T_n)\}$, une union $\sqcup_i T_i$ en $\sqcup_i \text{Mess}(T_i)$ et préserve les variable en attendant de connaître leur valeur.

Ces types indiquent que :

- le paramètre **Alarm** de type α doit être un message (un tuple commençant par un atome);
- le processus qui appelle cette fonction recevra ce message (codé par l'effet de la fonction);
- le résultat est une adresse de processus comprenant les messages **cancel** transportant une adresse qui recevra l'alarme.
- un appel à **cancel** doit lui fournir en argument une adresse qui recevra un message **cancel** ayant comme contenu l'adresse du processus courant (l'effet de la fonction). Le résultat est alors un couple composé de **cancel** et de l'adresse du processus courant.
- enfin, la fonction décrivant le corps du timer prend en argument une adresse (qui va recevoir le troisième argument – l'alarme), un entier et une valeur quelconque (devant être un message). Le résultat est alors ou bien **true** ou bien le troisième argument (l'alarme). Le processus exécutant cette fonction comprend les messages **cancel** transportant des adresses (qui recevront l'alarme).

On construit ainsi, un système de type sophistiqué dont nous avons prouvé la correction (voir [Dag01]) et pour lequel nous avons construit un prototype.

5 Discussion

Dans cet article, nous avons donc rapidement décrit ERLANG un langage fonctionnel concurrent et réparti conçu par ERICSSON. Nous avons ensuite proposé une formalisation de sa sémantique par une réduction à deux niveaux : un premier, traitant uniquement des aspects concurrents, basé sur les configurations et un second, décrivant la sémantique fonctionnelle (et ses éventuels effets concurrents) dans un cadre plus classique. Enfin, nous avons évoqué un système de type pour ERLANG, en focalisant notre présentation sur les parties les plus originales de nos travaux. La formalisation complète de ce système de type a été réalisée mais sa description dépasse de loin le format de cet article.

La sémantique d'ERLANG exhibée n'est que partielle et écrire une sémantique formelle complète du langage est une tâche beaucoup plus complexe car il faudrait :

- **ajouter la notion de site**. Pour cela, les configurations doivent être augmentées par une construction de la forme $s\{w\}$ signifiant que w s'exécute sur le site s .
- **gérer le chargement dynamique de code**. Chaque site doit contenir l'environnement des fonctions accessibles et les définitions de ces fonctions peuvent changer : $s\{\mathcal{E} \mid w\}$.
- **permettre l'envoi de messages entre sites**. Le destinataire du message peut être local en gardant la même syntaxe ou situé sur un autre 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 l’attente au bout du temps précisé. Il faudrait donc intégrer une notion d’horloge logique à chaque 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 et représente un processus (cette valeur peut changer). Chaque site doit alors maintenir un tel dictionnaire (en fait un environnement) : $s\{\mathcal{E}_f \mid \mathcal{E}_n \mid w\}$.
- **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 un message d’un signal.

Certains travaux récents sur des calculs de processus distribués comme le $D\pi$ (voir [Sew98]) ou le join calculus (voir [FGL⁺96]) peuvent servir de base dans un projet de formalisation de la notion de site dans ERLANG.

Les points ci-dessus sont les principaux problèmes à résoudre mais ce 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, tentent de procéder de manière la plus systématique possible.

En ce qui concerne le typage d’ERLANG, l’approximation grossière effectuée par le système de type proposée ne permet pas de détecter tous les orphelins potentiels. Il est donc souhaitable d’améliorer la précision du système de type de manière à réduire le nombre de message non détecté.

Dans le cadre d’un calcul de processus pour les acteurs (CAP), J-L. Colaço et al. ont défini un système de type plus précis qui capture tous les orphelins potentiels du programme. Ce système repose sur un décompte des messages envoyés et installés au moyen de multiplicités (voir [CPDS99]). Le système de type d’ERLANG a été conçu de façon à permettre simplement cette augmentation de la précision de l’approximation des interfaces. Pour cela, il faut modifier :

- Les termes de présence pour qu’ils intègrent les multiplicités (cette intégration est aisée). Un terme de présence devient un couple qui mémorise le nombre de messages envoyés et installés. Par exemple, le type d’une cellule linéaire est :

$$@\{\mathbf{start} : (1,1)int, \mathbf{get} : (2,\omega)@\{\mathbf{reply} : (1,\omega)int\}, \mathbf{set} : (1,\omega)int\}$$

- Les règles de typage de l’envoi des messages et celle du **receive** doivent additionner les messages. Le cas du **receive** est assez simple (il suffit d’incrémenter le compteur à chaque apparition dans un **receive**) mais celui des messages envoyés est nettement plus complexe. En effet, il faut modifier le type fonctionnel car une fonction qui contient l’envoi d’un message m , le fait autant de fois qu’elle est appelée. Par exemple, la fonction **send1** ci-dessous peut être approximée par le type $[@\{m : (x,y)t\}, [t, \dots, t_n] \rightarrow @\{m : (x + n, y)t\}$.

```
send1(A, []) -> A;
send1(A, [H|T]) -> A ! {m,H}, send1(A,T).
```

Ce type permet de conserver le nombre de messages m envoyés à chaque appel (qui est égal à la taille des listes passées en paramètre). 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. Par exemple, si la fonction **send1** ne prend pas **a** en paramètre ou/et ne le renvoie pas, le type de la fonction

ne permet plus de mémoriser les messages envoyés à **a**. Nous explorons actuellement deux pistes pour résoudre ce problème :

- Les contraintes générées par une fonction sont stockées dans son type : $\alpha \xrightarrow{c} \beta$ et il faut modifier le typage des fonctions et de l'application pour respectivement stocker les contraintes dans le type ou les relâcher. Cependant, 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 théoriquement.
- 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.
- Le processus de résolution doit alors intégrer une résolution de contraintes en arithmétique entière pour calculer ces multiplicités. Ce point pose des difficultés techniques pour arriver à faire collaborer le solveur de contraintes ensemblistes avec un solveur de contraintes arithmétiques. En nous inspirant du système de type de [HPS96], nous effectuons actuellement des travaux dans ce but. Notons que pour des problèmes de décidabilité des inéquations arithmétiques, il faut limiter la forme des programmes typables.

Le système de type nécessite également quelques extensions pour devenir un outil d'analyse précis de programmes ERLANG «complets».

Premièrement, les messages d'ERLANG ne contenant pas d'étiquettes de messages, les types donnés aux processus et les effets de fonctions doit être modifiés. Pour cela, les travaux sur XML (un langage fonctionnel typé pour manipuler des documents XML) de [SM01] peuvent servir de base. En effet, pour typer correctement les choix de XML, ils construisent un λ -calcul typé manipulant des enregistrements sans étiquette. Par exemple, $(1) + (\text{"test"}) + (\lambda x. \text{if } x \text{ then } 1 \text{ else } 0)$ est typé par $\{int; string; bool \rightarrow int\}$. Cependant, cette adaptation ne semble pas évidente car le système de type de XML 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, 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 de [Pes99] serait donc importante pour obtenir un outil d'analyse statique pour 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, en ERLANG, un module est utilisé par des centaines de milliers de nœuds qui ne peuvent pas être stoppés et redémarrés. Une évolution de ce module utilise donc un changement au cours de l'exécution (le nouveau code doit pouvoir cohabiter avec l'ancienne version s'exécutant sur certains nœuds pendant une période transitoire). Un premier pas vers de tels systèmes pourrait être l'adaptation du système de type présenté dans [Sew01].

Références

- [Agh86] G. AGHA. *Actors: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. The MIT Press, Cambridge, MA, USA, 1986.

- [Arm96] J. ARMSTRONG. « Erlang - A Survey of the Language and its Industrial Applications ». Dans *INAP'96. Hino, Tokyo, Japan*, octobre 1996.
- [AVWW96] J. ARMSTRONG, R. VIRIDING, C. WIKSTRÖM, et M. WILLIAMS. *Concurrent programming in ERLANG*. Prentice Hall, second edition édition, 1996.
- [AWL94] A. S. AIKEN, Edward L. WIMMERS, et T. K. LAKSHMAN. « Soft Typing with Conditional Types ». Dans *Conference Record of POPL'94*, pages 163–173, Portland, Oregon, janvier 1994. ACM Press.
- [BR98] S. BLAU et J. Rooth. « AXD 301 - A new generation ATM switching system ». *Ericsson Review*, (01), 1998.
- [BV99] J. BARKLUND et R. VIRIDING. « ERLANG 4.7.3 Reference Manual », February 1999. http://www.erlang.org/download/erl_spec47.ps.gz.
- [CPDS99] J-L. COLAÇO, M. PANTEL, F. DAGNAT, et P. SALLÉ. « Static Safety Analysis for Non-uniform Service Availability in Actors ». Dans *Proc. 3rd IFIP Workshop on FMOODS*, pages 371–386, Florence, Italy, février 1999. Kluwer.
- [CPS96] J-L. COLAÇO, M. PANTEL, et P. SALLÉ. « CAP: An Actor Dedicated Process Calculus ». Dans *Proceedings of Proof Theory of Concurrent Object-Oriented Programming*, mai 1996.
- [Dag01] F. DAGNAT. « *Vérification statique de programmes répartis* ». Thèse de doctorat, Institut National Polytechnique de Toulouse, mai 2001.
- [DkF98] M. DAM et Lars åke FREDLUND. « On the Verification of Open Distributed Systems ». Dans *Proc. of the ACM Symposium on Applied Computing*, volume 28, pages 532–540. ACM, juin 1998.
- [DPCS00] F. DAGNAT, M. PANTEL, M. COLIN, et P. SALLÉ. « Typing Concurrent Objects and Actors ». *L'Objet – Méthodes formelles pour les objets*, Volume 6(1/2000):pages 83–106, mai 2000.
- [FGL⁺96] C. FOURNET, G. GONTHIER, J-J. LÉVY, L. MARANGET, et D. RÉMY. « A Calculus of Mobile Agents ». Dans *Proceedings of CONCUR '96, Pisa, Italy*, volume 1119 de *LNCS*, pages 406–421. Springer-Verlag, 1996.
- [HPS96] J. HUGHES, L. PARETO, et A. SABRY. « Proving the Correctness of Reactive Systems Using Sized Types ». Dans *Proceedings of POPL '96 (St. Petersburg Beach, Florida)*, pages 410–423. ACM, janvier 1996.
- [Huc99] F. HUCH. « Verification of Erlang Programs using Abstract Interpretation and Model Checking ». *Proceedings of ICFP '99*, 34(9):261–272, septembre 1999.
- [Ler92] X. LEROY. « *Typage polymorphe d'un langage algorithmique* ». Thèse de doctorat, Université Paris 7, 1992.
- [Lin96] A. LINDGREN. « A Prototype of a Soft Type System for Erlang ». Master's thesis, Computing Science Departement, Uppsala University, 1996.
- [MMS88] A. MARCOUX, C. MAUREL, et P. SALLÉ. « A Language for Distributed Applications ». Dans *Workshop on Future Trends of Distributed Systems in the 90's*, 1988.
- [MW97] S. MARLOW et P. WADLER. « A practical Subtyping System For Erlang ». Dans *Proc. of International Conference on Functionnal Programming*, June 1997.
- [Pes99] F. PESSAUX. « *Détection statique d'exception non rattrappées en Objective Caml* ». PhD thesis, Université Paris 6, décembre 1999.
- [Pot98] F. POTTIER. « *Synthèse de types en présence de sous-typage: de la théorie à la pratique* ». PhD thesis, Université Paris 7, juillet 1998.

- [Rém94] D. RÉMY. Type Inference for Records in a Natural Extension of ML. Dans *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, avril 1994.
- [RV98] D. RÉMY et J. VOILLON. « Objective ML: An Effective Object-oriented Extension to ML ». *Theory And Practice of Object Systems*, 4(1):27–50, 1998.
- [Sew98] P. SEWELL. « Global/Local Subtyping and Capability Inference for a Distributed π -calculus ». Dans *Proceedings of ICALP '98. LNCS 1443*, pages 695–706. Springer-Verlag, juillet 1998.
- [Sew01] P. SEWELL. « Modules, Abstract Types, and Distributed Versioning ». Dans *Conference Record of POPL'01*, pages 236–247, London, United Kingdom, janvier 2001.
- [SM01] M. SHIELDS et E. MEIJER. « Type-indexed Rows ». Dans *Conference Record of POPL'01*, pages 261 – 275, London, United Kingdom, janvier 2001.
- [TJ94] J-P. TALPIN et P. JOUVELOT. « The Type and Effect Discipline ». *Information and Computation*, 111(2):245–296, juin 1994.
- [WA96] M. WILLIAMS et J. ARMSTRONG. « *Program Development Using Erlang - Programming Rules and Conventions* ». ERICSSON, mar 1996. Doc. EPK/NP 95:035.