

Vérification Statique de Programmes Répartis

Fabien Dagnat

`dagnat@enseeiht.fr`

Institut de Recherche en Informatique de Toulouse

France

Plan

- Contexte et problématique
- Les acteurs et leur modélisation
- Les erreurs
- Le typage et son fonctionnement
- L'application à Erlang
- Conclusion et perspectives

Contexte I

- Programmation concurrente et répartie de plus en plus courante.
- Mais domaine qui reste difficile (deadlock, data races, désynchronisation ...).
- Beaucoup de travaux de recherche en vue :
 - d'améliorer la qualité de telles applications,
 - de faciliter leur programmation.

Contexte II

- Amont :
 - méthodes et outils de conception (B, SDL, statecharts, ...)
 - preuve et vérification des specifications (obligation de preuve B, model checking, ...)
 - langages spécialisés (approche synchrone, Erlang, plan P, ...)
- Aval :
 - Vérification de programmes
 - Middleware transparent
 - Génération automatique de test et simulation

- Approche Langage + Vérification Statique
- Utilisation du modèle d'acteur, paradigme de haut niveau qui résoud certains problèmes (accès concurrents par exemple).
- Utilisation de techniques issues du typage pour pouvoir indiquer au programmeur des problèmes potentiels de communication.

Vérification Statique

Usuellement l'approche consacrée est :

- Réalisation d'une modélisation dans un calcul de processus (π -calcul, join calcul, ambient, **CAP**)
- construction d'une approximation d'un terme (interprétation abstraite, **typage**)
- vérification de cette approximation (model checking, **contraintes de sous-typage**)

Vérification Statique

Usuellement l'approche consacrée est :

- Réalisation d'une modélisation dans un calcul de processus (π -calcul, join calcul, ambient, CAP)
- construction d'une approximation d'un terme (interprétation abstraite, **typage**)
- vérification de cette approximation (model checking, **contraintes de sous-typage**)
- Adapter l'approximation à des langages de programmation.
- Adapter la vérification à des langages de programmation.

Approche Suivie

Intégration des typages fonctionnel et concurrent pour analyser des programmes mixant fortement ces deux aspects

```
let rec broadcast m = function
  | []    -> ()
  | h::t -> send m to h; broadcast m t
a pour type : 'a -> (@'a) list -> unit
```

Les Acteurs

- Entité réactive composée d'une adresse (appelée **ego**), d'une boîte aux lettres et d'un comportement (appelé **self**).
- Médium sans perte dans lequel aucun ordre n'est garanti.
- Trois opérations minimales :
 - **Send**, envoi de message selon un protocole asynchrone, point à point unidirectionnel et pouvant transporter des adresses et des comportements.
 - **New** qui crée un nouvel acteur.
 - **Become** qui permet à un acteur de modifier son self.

⇒ modèle abstrait qui facilite la prog. d'appli. concurrentes

MAIS triplement dynamique !

Les Configurations I

- But : construire un modèle minimal de la concurrence et de la communication.
- Abstraction de la notion de comportement :
 - la syntaxe et la sémantique fonctionnelle,
 - l'accès aux boîtes aux lettres,
 - la forme des messages
- Sémantique concurrente donnée par un système de transition étiqueté
- Sémantique finale par entrelacement réduction concurrente / fonctionnelle.

Les Configurations II

- Acteur : $\langle \text{adresse} \mid \text{boîte aux lettres} \rangle \triangleright \text{calcul}$
- Message en transit : $\text{adresse} \triangleleft \text{message}$
- $a \triangleleft m \parallel \langle a \mid q \rangle \triangleright e \parallel \langle b \mid \emptyset \rangle \triangleright e'$
- ϵ et $\nu a.(\dots)$ et $\star \triangleright e$
- Pour retrouver CAP et valider ce modèle, un noyau minimal :
 - récursivité, fonctions, variables, messages ($m(e)$), interfaces ($\chi[m(x).e, \dots]$), send, become et new
 - sémantique fonctionnelle usuelle avec appel par valeur
 - accès à la boîte aux lettres FIFO en utilisant l'étiquette

Les Erreurs

- Objets : « message not understood »
⇒ calcul statique de l'ensemble des méthodes (cumul des méthodes définies par l'objet et de celles de tous ces parents)
- Acteurs : même problème « message orphelin » mais plus **complexe** car chgt d'interface
- La calcul statique n'est plus possible, il faut faire une approximation !
- Notre originalité, c'est d'essayer de detecter ces messages ! Tous les autres, join, Erlang, ... ne s'intéresse pas au problème

Intégration à la Sémantique

- Caractérisation fortement dynamique
- Restriction pour une analyse statique, **orphelins triviaux**
- \Rightarrow potentiel (collecte des interfaces locales)
- Calcul à la création de l'acteur

$$\bullet a \triangleleft m \parallel \langle a|q \rangle_{\mathcal{P}} \triangleright e \longrightarrow \begin{cases} \langle a|m :: q \rangle_{\mathcal{P}} \triangleright e \text{ si } m \in \mathcal{P} \\ \text{ERREUR} \end{cases}$$

L'Approximation

- Acteur : $@\{m_1 : \circ t_1, m_2 : \bullet t_2\}$
- Comportement : $\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow t_{ego} \rightarrow \triangleleft\{m : \bullet t, \dots\}$
- Sous-typage :
 - $@t \sqsubseteq @t' \Rightarrow t' \sqsubseteq t$ $\circ t \sqsubseteq \circ t' \Rightarrow t \sqsubseteq t'$
 - $\bullet t \sqsubseteq \bullet t' \Rightarrow t' \sqsubseteq t$ $\circ t \sqsubseteq \bullet t' \Rightarrow t \sqsubseteq t'$
- Principe du typage d'un acteur a :
 - cumul des messages envoyés à a ,
 - de son interface initiale,
 - des messages envoyés aux ego de tous ses futurs comportements
 - de ceux qu'ils comprennent

La Résolution

- $\begin{cases} \{m : \circ t_1\} \sqsubseteq \phi_a \\ \{m : \bullet t_2\} \sqsubseteq \phi_a \end{cases} \Rightarrow \phi_a = \{m : \alpha, i\}$ et $\begin{cases} \circ t_1 \sqsubseteq \alpha \\ \bullet t_2 \sqsubseteq \alpha \end{cases}$
- $\Rightarrow \alpha = \bullet \beta$ et $t_1 \sqsubseteq \beta \sqsubseteq t_2$
- Utilisation d'un graphe de contraintes
- Processus incrémental qui maintient le graphe clos
- Simplification des contraintes
- Réintroduction d'unification sur les données fonctionnelles
- Au final, on vérifie que tous les acteurs créés ont un type qui ne contient plus de \circ

- ML-Act = ML + label + ego + self + behavior + new + send + become (+suicide)
- Extension « relativement » simple
- Intégration de techniques de typage fonctionnel usuelles
- Permet de maîtriser le langage

- langage fonctionnel concurrent et réparti conçu par ERICSSON
- langage industriel utilisé pour des équipements de télécommunication
- qui hérite de :
 - Prolog : atomes + filtrage hétérogène, non linéaire et dynamique
 - LISP : typage dynamique + liaison partiellement dynamique
 - ML : assignement unique + gestion automatique de la mémoire
- contient également exceptions, modules, temps réel, **chargement dynamique de code**

Sémantique d'Erlang

- Liaison des variables inhabituelle (fonctions globales implicitement récurives, liaison partiellement dynamique)
- Filtrage gardé
- Identité d'une fonction = nom (atome) + arité (pas toujours calculable)
- Boîte aux lettres différentes
- Messages sans étiquette (\Rightarrow restriction des programmes)

\Rightarrow Il faut construire un typage fonctionnel spécifique

Typage d'Erlang

Forte modification de la résolution des contraintes

- Accès synchrone à la boîte aux lettres et liaison dynamique de l'ego
⇒ calcul d'effet ($A \xrightarrow{E} B$)
- Filtrage dynamique
⇒ types étoilés (polymorphisme par nom) par renommage / duplication de contraintes
- Filtrage hétérogène
⇒ contraintes conditionnelles ($\alpha \sqsubseteq int \Rightarrow unit \sqsubseteq \beta$) par disjonction d'ensemble de contraintes

Bilan et Perspectives

- Conception d'un cadre général de construction de sémantique et de système de type pour les langages d'acteurs
- Abstraction validée dans le cadre de deux langages radicalement différents
- Un compilateur pour ML-ACT et un analyseur pour Erlang
- Compter les message pour s'attaquer à la vivacité
- Etudier le chargement dynamique de code
- Evaluer ces systèmes dans le cadre de développements