

Vérification Statique de Programmes Répartis

Fabien Dagnat

Lundi 28 mai 2001

Contexte

- Typage des acteurs
 - Modélisation par un calcul de processus (Cap)
 - Systèmes de type pour Cap
 - Problème :
 - Cap centré sur communication
 - Langages d'acteurs doivent contenir du calcul
- ⇒ Étendre ces systèmes pour typer de *vrais* langages

Approche suivie

- Intégration typage fonctionnel et concurrent pour analyser des programmes mixant ces deux aspects

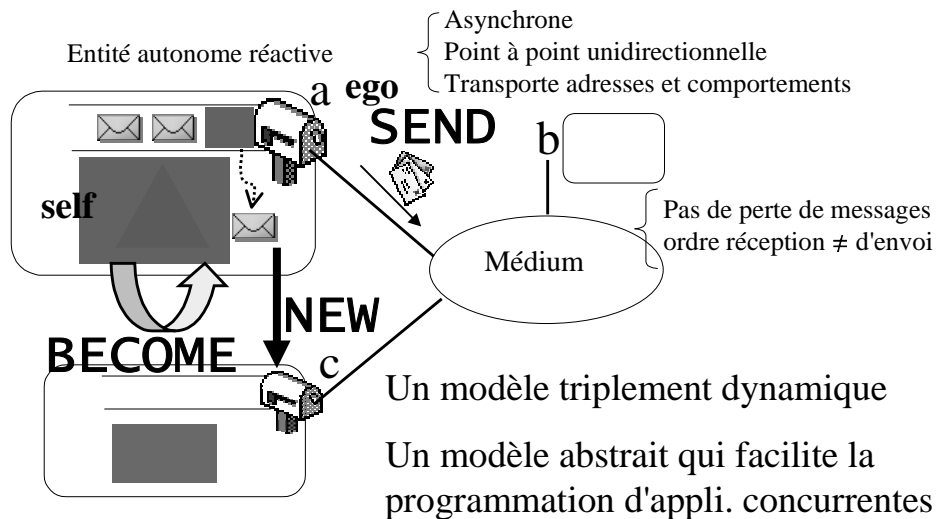
```
let rec broadcast m = fonction
  | [] -> ()
  | h::t -> send m to h; broadcast m t
```

→ 'a -> (@'a) list -> unit

Plan

- Le modèle des acteurs
- Notre modélisation des acteurs (les configurations)
- Les erreurs de communication
- Le typage
- Application à ML-Act et à Erlang

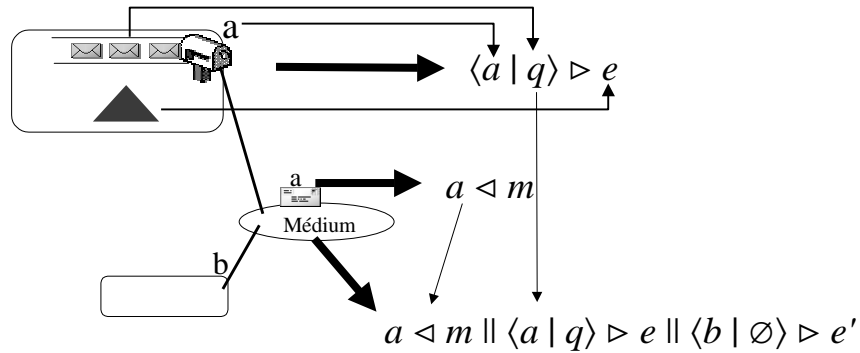
Les Acteurs



Les configurations

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

La Syntaxe des configurations



ϵ et $va(\dots)$ et $\star \triangleright e$

Première application : $Func_0$

- Un noyau minimal :

$$e ::= x \mid a \mid nop \mid e \ e \mid \lambda x. e \mid ! \text{etrec } x = e \text{ in } e \mid$$

$$m(e) \mid \chi[m(x).e, \dots, m(x).e] \mid$$

$$new \mid send(e,e) \mid become(e,e) \mid init(e,e)$$

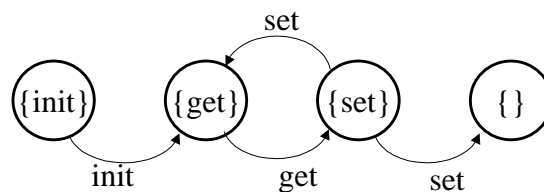
- Sémantique :

- partie fonctionnelle usuelle avec appel par valeur
- accès à la boîte aux lettres FIFO utilisant l'étiquette

Les erreurs

- Objets : « message not understood »
⇒ calcul statique de l'ensemble des méthodes
- Acteurs : même problème « message orphelin »
mais plus complexe car chgt dynamique d'interface

Exemple d'erreurs

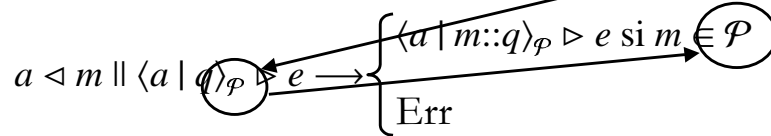


- **init** : si plus d'une fois erreur ⇒ orphelin potentiel
- **get, set** : toujours traités comme messages finis (si équilibrés)
potentiels
- **put** : toujours erreur ⇒ orphelin trivial
- **put** : toujours erreur ⇒ orphelin trivial

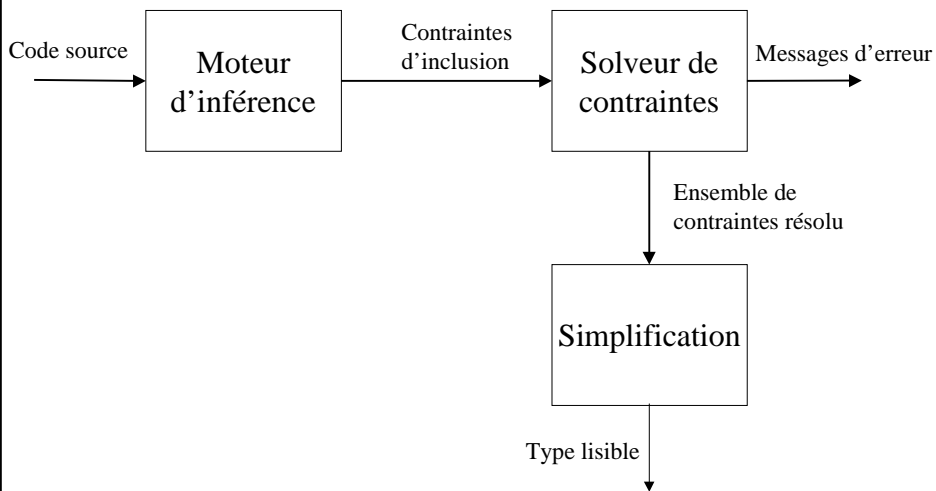
Intégration à la sémantique

- Caractérisation fortement dynamique
- Restriction pour une analyse statique
orphelins triviaux ($m \in Q(a, w_n) \Rightarrow \forall k \geq n \ m \in Q(a, w_k)$)
 \Rightarrow potentiel (collecte des interfaces locales)

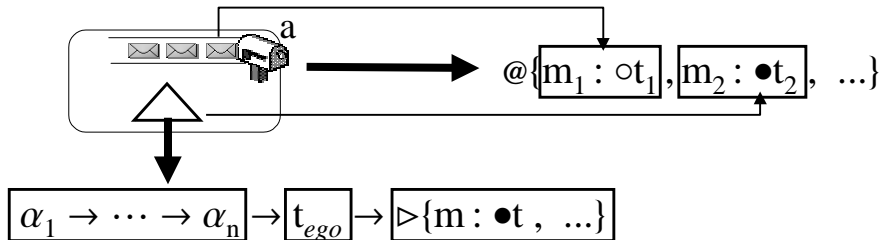
Initialisation de a avec le comportement $e : \langle a \mid \emptyset \rangle_{\mathcal{P}(e)} \triangleright e$



Le typage - Architecture



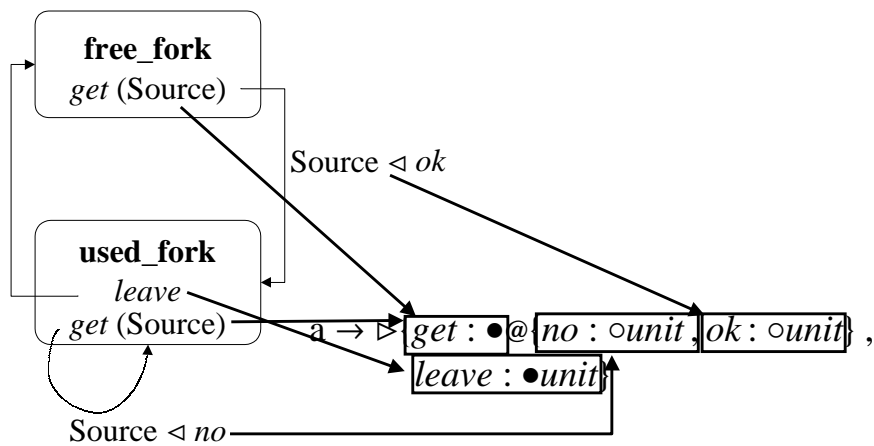
Le typage - L'approximation



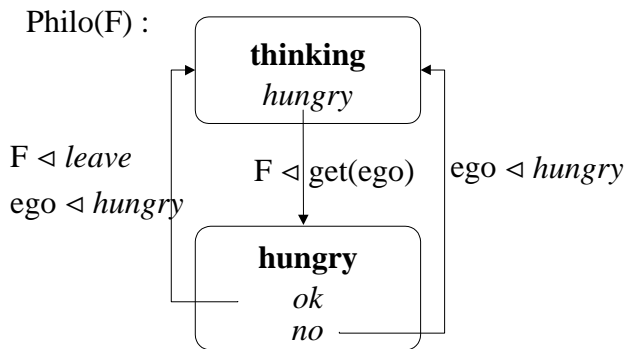
Sous-typage :

$$\begin{aligned}
 @t \sqsubseteq @t' &\Leftrightarrow t' \sqsubseteq t & ot \sqsubseteq ot' &\Leftrightarrow t \sqsubseteq t' & \bullet t \sqsubseteq \bullet t' &\Leftrightarrow t' \sqsubseteq t \\
 & & ot \sqsubseteq \bullet t' &\Leftrightarrow t \sqsubseteq t' & &
 \end{aligned}$$

Une Fourchette



Un Philosophe



$@\{get : \circ b, leave : \circ unit\} \emptyset b \emptyset \triangleright \{hungry : \bullet unit, ok : \bullet unit, no : \bullet unit\}$
avec $b \sqsubseteq @\{hungry : \circ unit\}$

Le Repas

Donc, si on crée une fourchette F et un philosophe P(F) :

$F : t_F$ et $t_F = @\{get : \bullet @\{no : \circ unit, ok : \circ unit\}, leave : \bullet unit\}$

$P : t_P$ et $t_P = @\{hungry : \bullet unit, ok : \bullet unit, no : \bullet unit\}$

Avec les contraintes :

$C = \{t_F \sqsubseteq @\{get : \circ t_P, leave : \circ unit\} ; t_P \sqsubseteq @\{hungry : \circ unit\}\}$

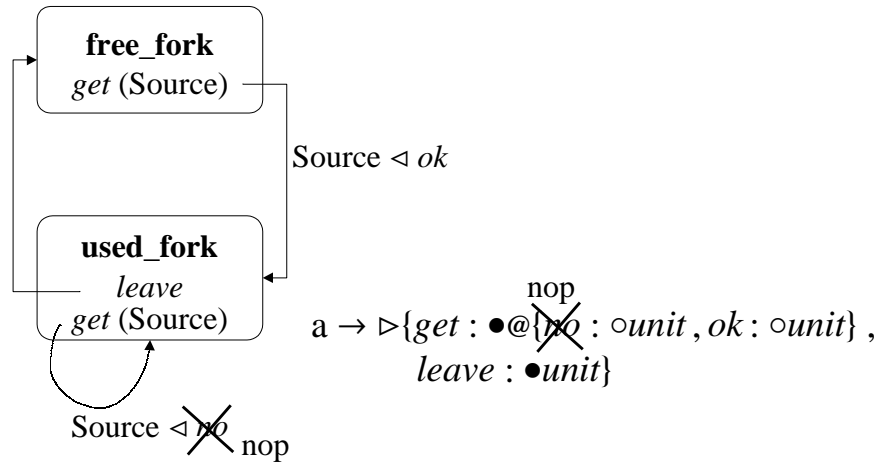
Soit, par simplification :

$C \Leftrightarrow \{t_P \sqsubseteq @\{no : \circ unit, ok : \circ unit\} ; unit \sqsubseteq unit\}$

$C \Leftrightarrow \{unit \sqsubseteq unit\}$

$C \Leftrightarrow \{\}$

Une Fourchette



Le Repas

Donc, si on crée une fourchette F et un philosophe P(F) :

$F : t_F \text{ et } t_F = @ \{ \text{get} : \bullet @ \{ \cancel{/ok} : \circ \text{unit}, \text{ok} : \circ \text{unit} \}, \text{leave} : \bullet \text{unit} \}$

$P : t_P \text{ et } t_P = @ \{ \text{hungry} : \bullet \text{unit}, \text{ok} : \bullet \text{unit}, \text{no} : \bullet \text{unit} \}$

Avec les contraintes :

$C = \{ t_F \sqsubseteq @ \{ \text{get} : \circ t_P, \text{leave} : \circ \text{unit} \}; t_P \sqsubseteq @ \{ \text{hungry} : \circ \text{unit} \} \}$

Soit, par simplification :

$C \Leftrightarrow \{ t_P \sqsubseteq @ \{ \cancel{/ok} : \circ \text{unit}, \text{ok} : \circ \text{unit} \}; \text{unit} \sqsubseteq \text{unit} \}$

$C \Leftrightarrow \{ \cancel{\text{unit} \sqsubseteq \text{unit}} \quad \text{nop} \in \{ \text{hungry ok no} \}$

$C \Leftrightarrow \{ \} \quad \text{Erreur}$

Le typage - Le principe

$\{m_1 : \circ t_1, \dots, m_n : \circ t_n\} \sqsubseteq \varphi_a$ Messages envoyés à a

$\{m'_1 : \bullet t'_1, \dots, m'_n : \bullet t'_n\} \sqsubseteq \varphi_a$ Comp. initial de a

$\{m''_1 : \circ t''_1, \dots, m''_n : \circ t''_n\} \sqsubseteq \varphi_{ego} \sqsubseteq \varphi_a$ Envois à ego

$\{m'''_1 : \bullet t'''_1, \dots, m'''_n : \bullet t'''_n\} \sqsubseteq \varphi_{ego} \sqsubseteq \varphi_a$ Become

$$\begin{aligned} \Rightarrow \left\{ \begin{array}{l} \{m : \circ t_1\} \sqsubseteq \varphi_a \\ \{m : \bullet t_2\} \sqsubseteq \varphi_a \end{array} \right. & \Rightarrow \varphi_a = \{m : \alpha, i\} \wedge \left\{ \begin{array}{l} \circ t_1 \sqsubseteq \alpha \\ \bullet t_2 \sqsubseteq \alpha \end{array} \right. \\ & \Rightarrow \alpha = \bullet \beta \wedge t_1 \sqsubseteq \beta \sqsubseteq t_2 \end{aligned}$$

Le typage - La résolution

- Graphe de contraintes clos + Valuation
- Processus incrémental
- Simplification des contraintes ($@t \sqsubseteq @t' \Rightarrow t' \sqsubseteq t$)
- Réintroduction d'unification sur les données fonctionnelles ($@t \sqsubseteq v \Rightarrow v = @v'$ et $v' \sqsubseteq t$)
- Seuls les termes de présence ont des bornes ($\circ t \sqsubseteq \alpha$)

ML-Act et Erlang

- ML-Act = ML + label + ego + self + behavior + new + send + become (+ suicide)
- Erlang : langage
 - fonctionnel concurrent et réparti conçu par ERICSSON
 - industriel utilisé pour des équipements de télécom.
 - qui hérite de :
 - Prolog : atomes+ filtrage hétérogène, non-linéaire et dynamique
 - LISP : typage dynamique + liaison partiellement dynamique
 - ML : assignation unique + gestion automatique de la mémoire
 - contient également exceptions, modules, temps réel, chargement dynamique de code

Application 2 : ML-Act

- Traduction dans un lambda calcul : $Func_1$
- Ajoute à $Func_0$: constantes, constructeurs, filtrage (fusion abstraction et interface)
- Ajout au typage : types de bases, constructeurs de type et typage des filtres
- new, init, become, send \Rightarrow fonctions prédéfinies
- Résolution similaire à $Func_0$

Application 2 : Erlang - Modifications sémantiques

- Liaisons très différentes :
 - fonctions globales implicitement récursives
 - liaison partiellement dynamique
- Filtrage gardé
- Identité d'une fonction = nom + arité (pas toujours calculable)
- Boîte aux lettres différente
- Messages sans étiquette \Rightarrow restriction des programmes

Application 3 : Erlang - Modifications du typage

- \Rightarrow Forte modification de la résolution
- Accès synchrone à la boîte aux lettres
 - \Rightarrow calcul d'effets ($A \xrightarrow{E} B$)
 - Filtrage dynamique
 - \Rightarrow types étoilés ($\alpha^* \rightarrow \chi$)
 - \Rightarrow renommage / duplication de contraintes
($C + \{\alpha^* \sqsubseteq T\} = C + [\beta/\alpha]C + \{\beta = T\}$)

Application 3 : Erlang - Modifications du typage

- Filtrage hétérogène
 - ⇒ contraintes conditionnelles
 $\alpha \sqsubseteq int \Rightarrow unit \sqsubseteq \beta$
 - ⇒ différence de types
 $(1 \rightarrow \alpha) \sqcup (int \setminus 1 \rightarrow \beta)$
 - ⇒ abandon total de l'unification
 - ⇒ disjonction d'ensemble de contraintes
 $C + \{\alpha \sqsubseteq int \Rightarrow unit \sqsubseteq \beta\} =$
 $(C + \{\alpha \sqsubseteq int; unit \sqsubseteq \beta\}) \vee (C + \{\alpha \sqsubseteq T \setminus int\})$

Bilan

- Conception d'un cadre de construction de sémantique et de système de type pour les langages d'acteurs
- Abstraction validée dans le cadre de deux langages à la sémantique radicalement différente
- Système de type plus général que sur Cap (transmission comportement)

Prototypes

- Un compilateur pour ML-Act
- Un analyseur pour Erlang
- Analyses qui demandent beaucoup de calculs
⇒ plutôt outils de vérification que compilateur
+ utilisation des types d'un typeur fonctionnel pour les parties purement fonctionnelles

Perspectives

- Finalisation d'un système de type comptant les messages
 - Extension du système de type d'Erlang (exceptions et chargement dynamique)
 - Évaluation de ces systèmes dans le cadre du développement d'applications
- ⇒ Intégration dans un atelier de développement d'applications réparties
- ⇒ Ajout de la modélisation de la distribution (sites + pannes)