8

A Set-Constraint-based analysis of Actors

J-L. Colaço, M. Pantel and P. Sallé
LIMA/ENSEEIHT/IRIT
2 rue Camichel 31071 TOULOUSE, FRANCE.
{colaco, pantel, salle}@enseeiht.fr
http://www.enseeiht.fr/Recherche/Info/Logiciel/vestale/vestale.html

Abstract

This paper presents a type inference system for a primitive actor calculus (CAP) based on set-constraints resolution. In contrast with concurrent objects, actors can change dynamically their interface (the set of messages they can handle). Therefore, the CAP calculus reduction rules can lead to orphan messages which will never be handled. The aim of the inference system is to detect statically many orphan messages and to produce information leading to the dynamic detection of the others. In this purpose, we define a flattening operation which abstracts the various behaviors of an actor. This static analysis is based on Aiken and Wimmers set-constraints resolution. It gives slightly better results than Vasconcelos or Yonezawa kinded types based analysis for concurrent objects.

Keywords

Actors, concurrent objects, type inference, set constraints.

1 INTRODUCTION

The actor model proposed by Hewitt and Agha (Agha 1986, Hewit, Bishop and Steiger 1973) generally leads to dynamically typed languages in the LISP tradition (Marcoux, Maurel and Sallé 1988, Yonezawa 1990). These languages involve many dynamic type checks. "Soft typing" has been introduced to reduce the number of dynamic type check (see (Cartwright and Fagan 1991, Aiken, Wimmers and Lakshman 1994)). Types for functional and object-oriented languages have been the subjects of active investigations, but many recent studies focus on typing concurrence (Kobayashi and Yonezawa 1994, Vasconcelos and Tokoro 1993, Pierce and Sangiorgi 1995, Nielson and Nielson 1993, Puntigam 1996). Our study addresses the problem of type inference for Actor based languages.

Actors are self-contained computational agents interacting via asynchronous message passing. An actor is composed of a *mail address* (that identifies the actor) and a *behavior*. When handling a message, an actor can *create* new actors; *send* messages to its acquaintances; and *modify* its behavior. These acquaintances can be stored in

the local state of the actor or given as arguments of a message. These possibilities lead to a dynamic topology of communication. In the Actor model, messages are guaranteed to be received but their order of arrival is unknown.

In contrast with concurrent objects, "Actor" means that the behavior of the entities can change during the computation and so, its interface (the set of messages that an actor can handle) can change dynamically. In this paper, we use set constraints to type actor programs. This static analysis cannot detect all the orphan messages (*messages that will never be handled*). But information to detect dynamically remaining orphans can be derived from inferred types; then this system can be classified as a "*soft typing system*".

This paper is organized as follows. Section 1 describes the formalism used to study actors, i.e. CAP * (Colaço, Pantel, Sallé and Senteni 1996), a kind of "actor-oriented process calculus". Section 2 presents the definition of types and the type system. The use of types information for run-time is outlined in section 3. Then in section 4 we compare our type system to Vasconcelos' one (Vasconcelos and Tokoro 1993). Finally, some insights in our future works are proposed.

2 A PRIMITIVE ACTOR CALCULUS (CAP)

The kernel of actor languages is based on asynchronous communication between actors. Existing actor languages also contain predefined data structures and sequential control structures. Communication is yet sufficient to express all possible computations and actors can represent data structures (see (Colaço, Pantel, Sallé and Senteni 1996)). We advocate that our calculus is primitive, because it only contains communication to express computation.

As in Milner's π-calculus (Milner 1991) and in Honda's ν-calculus (Honda and Tokoro 1991) the mail address of an actor is represented by a name. The actor behavior is represented by an interface containing methods and private fields. Methods can be accessed by communication and private fields are only reachable by the actor itself. Private fields can be seen as a private record associated to a behavior. This record contains data local to an actor.

CAP does not respect all the principles of Agha's actors, but it contains the notion of behavior and the notion of address as primitives that allow to express very easily actor programs. It also contains behavior communication (which is not in the original Actor model, but allows a simple but useful form of reflection (Colaço, Pantel and Sallé 1996, Colaço, Pantel, Sallé and Senteni 1996)) and the sharing of the same address by several different actors. A programming discipline in the use of CAP leads to classic actor programs; this discipline can be enforced by a linearity analysis described in (Colaço, Pantel and Sallé 1997). In (Colaço, Pantel, Sallé and Senteni 1996), we have shown how to translate a "classical" actor language in CAP.

Here is an example of a CAP expression :

*in french:*Calcul d'Acteurs Primitifs*

$$\nu a,b,d\ \big(a \rhd [read(c) = \zeta(e,s)(c \lhd rep(s.val)) \parallel e \rhd s)$$
$$write(v) = \zeta(e,s)e \rhd s.val \Leftarrow v$$
$$, val = b] \parallel a \lhd write(d) \parallel \cdots) \parallel \cdots$$

The ν operator creates three names (a,b,d) whose scope is represented by the outside parenthesis. The main part of the expression describes an actor whose mail address is a and whose behavior both accepts two different messages (or methods) : *read(c)* and *write(v)* and has a private field called *val*. This is the behavior of a buffer cell. A message *write(d)* is sent to a.

The $\zeta(e,s)$ (zeta) captures the *current address* (called *ego*) and the *current behavior* (called *self*) when the actor accepts a message. This operator is inspired by the ς (sigma) defined by Abadi and Cardelli to formalize self-substitution in objects. In our context, the capture of *self* and *ego* is used to formalize behavior changes.

2.1 Syntax

Programs are build from names and behaviors using the following constructors: messages, actors, concurrent composition, name creation (or scope restriction), inaction, field modification and field selection. Here are the main constructions:

- Message sending: $T_1 \lhd m(\tilde{T_2})$, messages are addressed to a single actor identified by his name (or mail address) (T_1), they carry a message label (m) and a tuple $(\tilde{T_2})$ representing the content of the message.
- Actors: $T_1 \rhd T_2$, actors have a single identifier (the mail address) (T_1) associated to a behavior (T_2).
- Behaviors: $[m_1(\tilde{x_1}) = \zeta(e_1,s_1)C_1 \cdots , p_1 = T_1 \cdots]$, behaviors encapsulate a finite set of message labels with the associated formal arguments, and a set of private fields without formal arguments (no substitution is possible during the field selection).

The complete syntax is given in the following definition.

Definition 21 (Syntax) *Let N be an infinite set of names, V an infinite set of variables and L a finite set of labels. N^* (resp. V^*) represents the set of finite sequences over N (resp. over V).*

Convention: in the following, $a,b,c,\ldots \in N$ - $v,x,e_i,s_i,\ldots \in V$ - $m_i,p_j \in L$ - $\tilde{x},\tilde{x_i},\ldots \in V^*$ - $C,D,C_i,\ldots$ are configurations (actors and messages composed with parallel

composition) - $T, T_i, \ldots$ are terms.
CAP *syntactic rules are:*

$$
\begin{array}{lcll}
Config & ::= & \phi & ; \textit{Empty (Inaction)} \\
 & | & \nu a\, Config & ; \textit{restriction} \\
 & | & Actor \parallel Config & ; \textit{parallel composition} \\
 & | & Message \parallel Config & ; \textit{parallel composition} \\
 & | & (\, Config\,) & ; \textit{parenthesis} \\
Message & ::= & Term \lhd m(\widetilde{Term}) & ; \textit{message construction} \\
Actor & ::= & Term \rhd Term & ; \textit{actor construction}
\end{array}
$$

$$
\begin{array}{lcll}
Term & ::= & [m_i(\tilde{x}_i) = \zeta(e_i, s_i) C_i^{\,i \in I}, p_j = T_j^{\,j \in J}] & ; \textit{behavior} \\
 & | & Term.p_j \Leftarrow Term & ; \textit{modification} \\
 & | & Term.p_j & ; \textit{selection} \\
 & | & a & ; \textit{name} \\
 & | & x & ; \textit{variable} \\
 & | & (Term) & ; \textit{parenthesis}
\end{array}
$$

2.2 Semantics

As we enable the communication of behaviors, we need to differentiate names and variables. Variables represent identifiers that can be substituted by *names* or *behaviors*; no substitution can be done on names.

Definition 22 (Free variables and free names) *This notion is defined in an usual way. "ν" is the only binder for names, "$\zeta(e,s)$" and "$m(\tilde{x}_i)$" are the two binding forms for variables. The set of free names (resp. free variables) of an expression is given by the function $\mathcal{FN}()$ (resp. $\mathcal{FV}()$).*

Definition 23 (Substitution) *Let $\sigma = \{T/X\}$ (where T is a term and X a variable) be a substitution. Let A be the expression on which σ is to be applied; it is supposed that every name and variable of A have been renamed in order to avoid capture of free names or free variables of T.*

$$
\begin{array}{rcl}
(\nu a\, C)_\sigma & = & \nu a\, C_\sigma \\
(C \parallel D)_\sigma & = & C_\sigma \parallel D_\sigma \\
(T \lhd m(\tilde{U}))_\sigma & = & T_\sigma \lhd m(\tilde{U_\sigma}) \\
(T \rhd U)_\sigma & = & T_\sigma \rhd U_\sigma \\
(T.p_j \Leftarrow U)_\sigma & = & T_\sigma.p_j \Leftarrow U_\sigma \\
(T.p_j)_\sigma & = & T_\sigma.p_j \\
[m_i(\tilde{x}_i) = \zeta(e_i, s_i) C_i^{\,i \in I}, p_j = T_j^{\,j \in J}]_\sigma & = & [m_i(\tilde{x}_i) = \zeta(e_i, s_i) C_{i\sigma}^{\,i \in I}, p_j = T_{j\sigma}^{\,j \in J}]
\end{array}
$$

$$a_\sigma = a$$
$$x_\sigma = \begin{cases} T & \text{if } x = X \\ x & \text{otherwise} \end{cases}$$

To define the reduction rules in a more concise way, we first introduce a congruence relation between the expressions of the calculus.

Definition 24 (Congruence) *"$\equiv$" is the smallest congruence over* CAP *expressions defined by the following rules:*

1. $C \equiv D$ *if* C *is* α*-convertible to* D *(*α*-conversion of names and variables).*
2. $C \parallel \phi \equiv C$
3. $C \parallel D \equiv D \parallel C$
4. $(C \parallel D) \parallel E \equiv C \parallel (D \parallel E)$
5. $T \rhd T_1 \equiv T \rhd T_2$ *if* $T_1 \equiv T_2$
6. $[m(\tilde{x}) = \zeta(e,s)C_m\ n(\tilde{y}) = \zeta(e,s)C_n,\ p = T_p\ q = T_q]$
   $\equiv [n(\tilde{y}) = \zeta(e,s)C_n\ m(\tilde{x}) = \zeta(e,s)C_m,\ p = T_p\ q = T_q]$
   $\equiv [m(\tilde{x}) = \zeta(e,s)C_m\ n(\tilde{y}) = \zeta(e,s)C_n,\ q = T_q\ p = T_p]$
7. $\nu a\, C \parallel D \equiv \nu a(C \parallel D)$ *if* $a \notin \mathcal{FN}(D)$

Definition 25 (Reduction rules) *The reduction denoted by "$\longrightarrow$" is the smallest relation generated by the rules:*

$$\textit{STRUCT}: \frac{D \equiv C \quad C \longrightarrow C' \quad C' \equiv D'}{D \longrightarrow D'}$$

$$\textit{SELECT-CONTEXT}: \frac{T \longrightarrow T'}{T.p_k \longrightarrow T'.pk} \qquad \textit{AFFECT-CONTEXT}: \frac{T_1 \longrightarrow T_1'}{T_1.p_k \Leftarrow T_2 \longrightarrow T_1'.pk \Leftarrow T_2}$$

$$\textit{MESS-CONTEXT}: \frac{T \longrightarrow T'}{a \lhd m(\ldots T \ldots) \longrightarrow a \lhd m(\ldots T' \ldots)}$$

$$\textit{PAR}: \frac{C \longrightarrow C'}{C \parallel D \longrightarrow C' \parallel D} \qquad \textit{RES}: \frac{C \longrightarrow C'}{\nu x C \longrightarrow \nu x C'}$$

$$\textit{MESS}: \frac{T \longrightarrow T'}{T \lhd m(\tilde{t}) \longrightarrow T' \lhd m(\tilde{t})} \qquad \textit{ACT}: \frac{T_1 \longrightarrow T_1' \quad T_2 \longrightarrow T_2'}{T_1 \rhd T_2 \longrightarrow T_1' \rhd T_2'}$$

$$\textit{COMM}: \begin{cases} \text{if } k \in I \text{ and } len(\tilde{v}) = len(\tilde{x_k}), \\ a \lhd m_k(\tilde{v}) \parallel a \rhd [m_i(\tilde{x_i}) = \zeta(e_i,s_i)C_i{}^{i\in I}, p_j = T_j{}^{j\in J}] \\ \longrightarrow C_k\{a/e_k\}\{[m_i = \cdots, p_j = \cdots]/s_k\}\{\tilde{v}/\tilde{x_k}\} \end{cases}$$

$$\textit{SELECT}: \quad \textit{if } k \in J,\ [m_i(\tilde{x}_i) = \zeta(e_i, s_i)C_i{}^{i\in I}, p_j = T_j{}^{j\in J}].p_k \longrightarrow T_k$$

$$\textit{REDEF}: \begin{cases} \textit{if } k \in J, & [m_i(\tilde{x}_i) = \zeta(e_i, s_i)C_i{}^{i\in I}, p_j = T_j{}^{j\in J}].p_k \Leftarrow T \\ & \longrightarrow [m_i(\tilde{x}_i) = \zeta(e_i, s_i)C_i{}^{i\in I}, p_j = T_j{}^{j\in J\setminus\{k\}} \ \ p_k = T] \end{cases}$$

These reduction rules allow to send a message that cannot be understood by the target actor. In this case, the message remains in the expression. A future behavior of the target actor will perhaps be able to handle it, otherwise this message is called *orphan.*

This notion of *orphan messages* breaks the fairness principle of Agha's model. However, we can define a weaker form of fairness adapted to our semantics. We require that: *if a message "$a \lhd m(\cdots)$" is present in the medium, then "a" can only handle a finite number of messages labeled "m" before handling this message.*

CAP syntax and reduction rules are not very restrictive and they allow some undesirable expressions to be written or to appear during computation. These *ill-formed expressions* are of the following forms:

- ill-formed Actors : $[m_i(\tilde{x}_i) = \cdots, p_j = \cdots] \rhd T,\ T \rhd a$
- ill-formed Message : $[m_i(\tilde{x}_i) = \cdots, p_j = \cdots] \lhd m(\tilde{T})$
- ill-formed Terms : $\begin{cases} [\cdots, p_j = T_j{}^{j\in J}].p_k \text{ with } k \notin J, \\ [\cdots, p_j = T_j{}^{j\in J}].p_k \Leftarrow T \text{ with } k \notin J. \end{cases}$

Expressions leading to ill-forms will be statically eliminated by the type system.

2.3 Examples

To illustrate the calculus, two short examples are given.

Example 21 (the one-place buffer) *This example shows a two-states actor with an* empty buffer *behavior and a* full buffer *behavior.*

$$OPbuff \stackrel{\text{def}}{=} [\ put(v) = \zeta(e_{empty}, s_{empty}) \\ (e_{empty} \rhd [get(c) = \zeta(e_{full}, s_{full})(c \lhd rep(v) \parallel e_{full} \rhd s_{empty})])\ ]$$

When installing this behavior on a mail address, the actor behaves as an empty buffer that can only accept the message "put". After receiving such a message, the actor only accepts a "get" message and then behaves as an empty buffer. The variable s_{empty} is associated to the empty-buffer behavior and the variable s_{full} is associated to the full-buffer behavior containing the stored value.

Example 22 (2-D point) *This second example presents the use of private fields:*

$$
\begin{array}{lll}
point(x,y) \stackrel{\text{def}}{=} & [& getx(c) = \zeta(e,s)(c \lhd rep(s.xc) \;\|\; e \rhd s) \\
 & & gety(c) = \zeta(e,s)(c \lhd rep(s.yc) \;\|\; e \rhd s) \\
 & & move(xx,yy) = \zeta(e,s)(e \rhd (s.xc \Leftarrow xx).yc \Leftarrow yy) \\
 & , & xc = x \\
 & & yc = y \,]
\end{array}
$$

Coordinates of the point are stored in private fields. When a point receives a "move" message, the actor updates its data.

More examples are given in (Colaço, Pantel, Sallé and Senteni 1996, Colaço, Pantel and Sallé 1996).

3 TYPES

The work presented in this paper is the first step in our studies on type inference for actors; this first proposition focuses on indeterminism and behavior changes. Therefore we only consider names communications. The type system will reject behaviors communications as not typable in this system.

We define three kinds of types representing *name types*, *behavior types* and *configuration type*. "$\wp$" is the type of all well-typed configurations. A name type represents all the messages a name (or a variable that will be substituted by a name) can potentially receive. "Potentially" means that during the computation an actor can change his behavior and therefore lose the ability to handle some messages that he could handle before. A behavior type contains two components: a name type that will be associated to the name on which it will be installed and a record type representing its private fields.

Definition 31 (Types syntax) *Let $\mathcal{V}_\tau$ be an infinite set of type variables, the set $\mathcal{T}$ of types is defined by the following grammar.*

$$
\begin{array}{lll}
\tau & ::= & \alpha \mid \beta \mid \wp \\
\alpha & ::= & t \mid \langle m_i(\tilde{\alpha}_i)^{i\in I}\rangle \qquad \textit{(where I and J are finite sets)} \\
\beta & ::= & \alpha \rhd [p_j : \tau_j{}^{j\in J}]
\end{array}
$$

$\mathcal{T}$ is the union of name types $\mathcal{T}_{name}$, behavior types $\mathcal{T}_{beh}$ *and* process type $\wp$: $\mathcal{T} = \mathcal{T}_{name} \cup \mathcal{T}_{beh} \cup \{\wp\}$. *In this paper, we use the following conventions:* $\tau, \tau', \tau_i, \ldots$ *range over* $\mathcal{T}$; $\alpha, \alpha', \alpha_i, \ldots$ *over* $\mathcal{T}_{name}$; $\beta, \beta', \beta_i, \ldots$ *over* $\mathcal{T}_{beh}$ *and* $t, \ldots$ *range over* $\mathcal{V}_\tau$. *Sequences of types are denoted with a tilde* ($\tilde{\tau}$).

Note that a type variable can only denote a name type.

A **name type** $\langle m_i(\tilde{\alpha}_i)^{i\in I}\rangle$ is interpreted as a set of m_i messages (with constraints $\tilde{\alpha}_i$ on arguments) that can be sent to an actor identified by this name. Our set operator definitions are based on this interpretation.

Name types can also be seen as sets of names identifying actors that can accept the same set of messages $\{m_i\}$ with constraints on arguments represented by $\tilde{\alpha}_i$.

3.1 Flattenings and actor types

The structure we use for name types is frequently used in the context of distributed objects (objects correspond to entities that do not change their interface like actors do). In these cases, types in methods are contravariant to ensure that no *"message not understood"* error occurs due to name passing. In our context, to prevent all orphan messages, one simple approach consists in ensuring that any message sent can be handled by every behavior the target actor can adopt (the intersection of all behaviors). This policy allows actors to change their behavior, but the types lack the parts of the interface which are not common to all the future interfaces. This solution is safe but too restrictive; for example, the type of the *"one-place buffer"* will be $\langle\rangle$, no message is common to all behavior, therefore the type system forbids every emission to any *"one-place buffer"* actor. Our purpose is to type actors with changing interface, so intersection is not an adequate solution. The union is neither the solution; as contravariance leads to an unsafe relaxation of constraints on parameters (by doing intersection instead of union). We define a new operation called *flattening* that behaves like a union for the set of labels and like an intersection at the parameters level. This choice relaxes the constraints on the interface of the actors, but not on the parameters.

In the context of actors and concurrent objects, there is a natural notion of *subsumption* meaning that an actor can always be replaced by an other one offering more services. Our aim is to express such a condition with an inclusion relation (which will be defined below).

3.2 Operations and relations on Types

Notation: operations with a tilde represent the sequence obtained by applying each component of one sequence to each component of the other; relations with a tilde represent the logical "and" between each component. Both require sequences of the same length.

Definition 32 (Name type inclusion "$\subseteq$") *The contravariant inclusion is defined by:*

$$\langle m_i(\tilde{\alpha}_i)^{i\in I}\rangle \subseteq \langle m_j(\tilde{\alpha'_j})^{j\in J}\rangle \iff I \subseteq J \wedge (\forall k \in I)\tilde{\alpha_k} \tilde{\supseteq} \tilde{\alpha'_k}$$

Definition 33 (Type equality) *With the previous inclusion, we can define an equality between two types:*

$$\begin{array}{rcl} \alpha = \alpha' & \iff & \alpha \subseteq \alpha' \wedge \alpha \supseteq \alpha' \\ \alpha \triangleright [p_j : \beta_j{}^j \in J] = \alpha' \triangleright [p_j : \beta'_j{}^j \in J] & \iff & \alpha = \alpha' \wedge (\forall j \in J,\ \beta_j = \beta'_j) \end{array}$$

Definition 34 (Union and intersection) *Because of the contravariance on message arguments, union and intersection have the following form:*

$$\langle m_i(\tilde{\alpha_i})^{i\in I}\rangle \cup \langle m_j(\tilde{\alpha'_j})^{j\in J}\rangle = \langle m_k(\tilde{\alpha_k}\tilde{\cap}\tilde{\alpha'_k})^{k\in I\cap J}\ m_i(\tilde{\alpha_i})^{i\in I\setminus J}\ m_j(\tilde{\alpha'_j})^{j\in J\setminus I}\rangle$$
$$\langle m_i(\tilde{\alpha_i})^{i\in I}\rangle \cap \langle m_j(\tilde{\alpha'_j})^{j\in J}\rangle = \langle m_k(\tilde{\alpha_k}\tilde{\cup}\tilde{\alpha'_k})^{k\in I\cap J}\rangle$$

Definition 35 (Flattening and flat inclusion) *The flattening operation is denoted by " $\uplus$ ":*

$$\langle m_i(\tilde{\alpha_i})^{i\in I}\rangle \uplus \langle m_j(\tilde{\alpha'_j})^{j\in J}\rangle = \langle m_k(\tilde{\alpha_k}\tilde{\cup}\tilde{\alpha'_k})^{k\in I\cap J}\ m_i(\tilde{\alpha_i})^{i\in I\setminus J)}\ m_j(\tilde{\alpha'_j})^{j\in J\setminus I}\rangle$$

we also define an order on flattenings denoted by " $\underline{\subseteq}$ "

$$\langle m_i(\tilde{\alpha_i})^{i\in I}\rangle \ \underline{\subseteq}\ \langle m_j(\tilde{\alpha'_j})^{j\in J}\rangle \Longleftrightarrow I\subseteq J \wedge (\forall k\in I,\ \tilde{\alpha_k}\tilde{\subseteq}\tilde{\alpha'_k})$$

The union "$\cup$" (resp. flattening " $\uplus$ ") is a least upper bound operator for the inclusion "$\subseteq$" (rep. flat order " $\underline{\subseteq}$ "); this result is immediate from previous definitions.

3.3 The type system

In this type system, judgments are of the form $E \vdash A : \tau$ where E is a type environment for names and variables, A is a CAP expression and τ a type. Some rules are labeled with a constraint between name types at the right side; if these constraints are satisfied, no reduction can lead to *ill-formed expressions*. Moreover, no arity problem may occur (in the COMM reduction rule it is no more needed to check if "$len(\tilde{v}) = len(\tilde{x_k})$"). In our approach, these constraints are collected and used to compute most general types which inform on the allowed use of actors.

$$\text{(Beh)}\ \frac{\begin{array}{c}(\text{let } \beta_e = \alpha_e \triangleright [p_j : \tau_j])\\ E \vdash T_j : \tau_j\ \ (\forall j \in J)\\ E, \tilde{x_i} : \tilde{\alpha_i}, s_i : \beta_e, e_i : \alpha_{e_i} \vdash C_i : \wp\ (\forall i \in I)\end{array}}{E \vdash [m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_i,\ p_j = T_j] : \beta_e}\ \left(\alpha_e \ \underline{\ni}\ \langle m_i(\tilde{\alpha_i})^{i\in I}\rangle \uplus \left(\biguplus_{i\in I} \alpha_{e_i}\right)\right)$$

$$\text{(Actor)}\ \frac{E \vdash T_1 : \alpha_1 \qquad E \vdash T_2 : \beta_2 \qquad (\text{where } \beta_2 = \alpha_2 \triangleright [\cdots])}{E \vdash T_1 \triangleright T_2 : \wp}\ \left(\alpha_1 = \alpha_2\right)$$

$$\text{(Message)}\ \frac{E \vdash T_1 : \alpha_1 \qquad E \vdash \tilde{T_2} : \tilde{\alpha_2}}{E \vdash T_1 \triangleleft m(\tilde{T_2}) : \wp}\ \left(\alpha_1 \supseteq \langle m(\tilde{\alpha_2})\rangle\right)$$

$$\text{(Empty)}\ \frac{}{E \vdash \phi : \wp} \qquad\qquad \text{(Restriction)}\ \frac{E, a : \alpha_a \vdash C : \wp}{E \vdash \nu a\, C : \wp}$$

$$\text{(Parallel)}\ \frac{E \vdash C : \wp \qquad E \vdash D : \wp}{E \vdash C \,||\, D : \wp} \qquad\qquad \text{(Select)}\ \frac{E \vdash T : \alpha \triangleright [p_j : \tau_j{}^{j \in J}] \qquad k \in J}{E \vdash T.p_k : \tau_k}$$

$$\text{(Affect)}\ \frac{E \vdash T : \alpha \triangleright [p_j : \tau_j{}^{j \in J}] \qquad E \vdash T' : \tau \qquad k \in J}{E \vdash T.p_k \Leftarrow T' : \alpha \triangleright [p_j : \tau_j{}^{j \in J}]}\ \left(\tau_k = \tau\right)$$

The main rule of this system is the Beh rule; because it shows how to build an object type for an actor by flattening present and future behaviors. Most of the time, the constraint on α_e given in this first rule is recursive; this comes from the fact that actors usually adopt several times the same behavior. This recursive equation is of the form: $\alpha \supseteq \alpha_1 \uplus \cdots \uplus \alpha_n \uplus \alpha$; there is a least solution in the sense of flat inclusion given by the least upper bound of $\alpha_1 \cdots \alpha_n$ i.e. $\alpha = \alpha_1 \uplus \cdots \uplus \alpha_n$. The types computed by solving the set of constraints are the least solution of such equations; they are the most precise. If this system was used to type-check expressions, the user could give other solutions of the recursive problem and the analysis would be less precise; but we do not consider type-checking, our system has been designed to infer types.

The other important rule is the Message rule expressing constraints on the target of the message and on effective arguments ($\vec{\alpha_2}$) whose types must be subsets of types required by the target actor.

In practice, the inference is done in two steps: *collecting constraints* and *solving constraints*.

3.4 Some properties of the type system

If an expression is well-typed within this system, then it will never reduce to an ill-formed expression.

Lemma 31 (Structural preserving of type assignment) *If $A \equiv B$ and $E \vdash A : \tau$ then $E \vdash B : \tau$.*

PROOF : By induction on the size of the derivation.

Theorem 31 (Subject reduction) *If $E \vdash A : \tau$ and $A \longrightarrow A'$ then $\exists E'$ such that $E' \vdash A' : \tau'$.*

PROOF : By induction on the definition of the reduction, using the previous lemma.

Lemma 32 *Ill-formed expressions and communication of behaviors are not typable.*

Theorem 32 (Soundness of the type system) *If A is typable, the reduction of an expression will never produce Ill-formed expressions.*

3.5 Constraints resolution

The inference process introduces type variables for all type names and generates three kinds of constraints on these variables: *equalities*, *flat inclusions* and *inclusions*. The first ones are used for type unification; the second ones express the construction of behavior types; the third ones are used to check if a message has some chance to be treated. We use a set constraint solver to compute the solution of the system. This solver is derived from the works of Aiken and Wimmers (Aiken and Wimmers 1993). It uses the operators definition and properties to decompose complex constraints and then it combines constraints on variables using transitivity. This scheme is applied until no more new simplified constraints may be introduced. Such an approach has also been used in (Pantel 1994) to type a functional object-oriented language. In order to compute the least solution of recursive problems, constraints of the form $t \supseteq \tau_1 \uplus t \uplus \tau_2$ are replaced by $t \supseteq \tau_1 \uplus \tau_2$. When the closure of the system is computed, as we are interested in the least solution, flat inclusions on type variables are replaced by equalities. At the end of the resolution, the solved system is composed of:

- equalities for the type variables of an actor name or an *ego* variable: $t = \langle m_i(\tilde{t}_i)^{i \in I} \rangle$
- intervals for the type variables attached to formal parameters: $\tau_1 \subseteq t \subseteq \tau_2$

Example 31 (The one-place buffer actor typing) *Using the behavior "OPbuff" previously defined, the typing of the actor "$a \triangleright$ OPbuff" leads to the following system:*

$$\{\ t_a = t_{e_{empty}} = t_{e_{full}} = \langle get(\langle rep(t_v) \rangle)\ \ put(t_v) \rangle\ \}$$

Example 32 (linear-cell) *This example presents a cell that can only be assigned once, and then only accepts to give its value, but not to modify it.*

$a \triangleright [init(v) = \zeta(e,s)(e \triangleright [get(c) = \zeta(e',s')(c \triangleleft rep(v) \ \|\ e' \triangleright s')]\ )]$

The final system for this actor is:

$$\left\{ \begin{array}{l} t_a = \langle init(t_v)\ \ get(t_c) \rangle \\ t_e = t_{e'} = \langle get(t_c) \rangle \\ t_c \supseteq \langle rep(t_v) \rangle \end{array} \right\}$$

The type t_a contains the whole initial potential of the actor; the type t_e represents the actor type after handling a message "init", this type is smaller in the sense of flat inclusion: $t_e \trianglelefteq t_a$. The remaining inclusion requires that the argument of get must be a name which is attached to an actor understanding the message "rep". The stored value v is not really used in this actor, it is only transmitted to c; in this context any value is correct.

3.6 Interpretation of the system after resolution

When the system is applied on a configuration, its answer is either "$\wp$" or "*Type Error*", which is not very informative. The approach proposed by Vasconcelos in (Vasconcelos and Tokoro 1993) is to give as result the environment containing the type of each free identifier in the configuration. This answer is interesting in the context of open systems; free names and variables correspond to ports reachable from the outside, these types inform us on how to use these ports.

We can do the same by giving the set of constraints on the types of free identifiers to the user. Although constraints give very rich information, they are still difficult to read. Inclusion constraints express an idea of polymorphism because there is a solution for every set of types satisfying the set of constraints. Moreover, the notion of subsumption can help us for the simplification of constraints set. Fuh and Mishra (Fuh and Mishra 1989) have proposed some solutions to the problem of types simplification in a functional context with subtyping. It is sometimes possible to saturate a constraint and then replace an inclusion by an equality and give an equivalent system. This possibility comes from the notion of subsumption: a name of type τ can be used every time a smaller type is required. Intuitively, in order to represent all the solutions of the system, we have to minimize the constraints on formal arguments in the behaviors. But due to the contravariance of the inclusion, minimizing a type can lead to maximize an other one. These two phenomena can appear simultaneously on the same type variable. In these cases, no saturation can be done on this variable and the constraint cannot be simplified.
For example, let us apply the saturation to the linear-cell. The types t_v and t_c have to be minimized, the minimization of t_c implies the maximization of t_v then t_c can be saturated and replaced by $\langle rep(t_v)\rangle$ and t_v remains in the system.

Example 33 (the linear-cell in a context) *In order to present the typing of an actor in a context, this example considers the previous linear-cell actor in parallel with a message and another actor:*

$$\begin{array}{l} \quad a \rhd [init(v) = \zeta(e,s)(e \rhd [get(c) = \zeta(e',s')(c \lhd rep(v) \ \| \ e' \rhd s')] \,)] \\ \| \quad a \lhd get(b) \ \| \ b \rhd [rep(w) = \zeta(e'',s'')(w \lhd mess() \ \| \ e'' \rhd s'')] \end{array}$$

The resulting system after saturation is:

$$\left\{ \begin{array}{l} t_a = \langle init(t_v) \ \ get(\langle rep(t_v)\rangle \,) \ \rangle \\ t_e = t_{e'} = \langle get(\langle rep(t_v)\rangle \,) \ \rangle \\ t_b = t_{e''} = \langle rep(\langle mess()\rangle \,) \ \rangle \\ t_v \supseteq \langle mess()\rangle \end{array} \right\}$$

In this system, we can see more intuitively why t_v cannot be saturated. Its lower bound depends on its use by the context. Without any context: t_v is not constrained

(or has the constraint "$t_v \supseteq \langle\rangle$"). In this specific context t_v is constrained by "$t_v \supseteq \langle mess() \rangle$". The lower bound of t_v is therefore context-dependent.

4 PRACTICAL USE OF TYPES DURING RUN-TIME

As our type system is not able to detect statically all the orphan messages, we propose to add some annotations (derived from inferred types) on the terms of the calculus, to allow dynamic detection of remaining orphan messages.

4.1 Annotated calculus

Definition 41 (Interface) *The interface is the set of message labels an actor can potentially accept. It can be calculated from name types using the function $I()$:*
$I(\langle m_i(\tilde{\alpha}_i)^{i\in I}\rangle) = \{m_i{}^{i\in I}\}$
In the following we use $\sigma, \sigma', \sigma_i$ to denote interfaces.

These decorations only appear in the actor constructor, they are computed from the type of the term at the actor name position. An actor is now denotated as $t \overset{\sigma}{\triangleright} t'$ where $\sigma = I(\tau)$ and τ is the type computed for the term t by the inference system.

Now that we have annotated terms, we can extend the semantics. There are, at least, two possibilities to deal with orphans when they are detected: *Run-time error* or *Memory deallocation.*

(a) Extension with run-time error

In this case the emergence of an orphan message is considered as something undesirable during the computation. In this semantics, we need to add a constant *Error* and to choose if errors are propagated or not in the rest of the expression. The following rules detects errors:

ORPH-ERROR : if $m \notin \sigma$ then $a \lhd m(\tilde{v}) \parallel a \overset{\sigma}{\triangleright} [m_i(\tilde{x}_i)\cdots] \longrightarrow Error$

(b) Extension with garbage collection

If we don't consider an orphan message as something dangerous or undesirable, the decoration can be used to free the memory space where such messages are stored.

ORPH-COLLECT : if $m \notin \sigma$ then $\begin{cases} a \lhd m(\tilde{v}) \parallel a \overset{\sigma}{\triangleright} [m_i(\tilde{x}_i)\cdots] \\ \longrightarrow a \overset{\sigma}{\triangleright} [m_i(\tilde{x}_i)\cdots] \end{cases}$

Example 41 (Annotated linear-cell) *Using the types computed for this behavior, the linear-cell actor can be annotated :*

$$a \overset{\{init,get\}}{\triangleright} [init(v) = \zeta(e,s)(e \overset{\{get\}}{\triangleright} [get(c) = \zeta(e',s')(c \lhd rep(v) \parallel e' \overset{\{get\}}{\triangleright} s')]\,)]$$

5 APPLYING THE SYSTEM TO AN OBJECT CALCULUS

This type inference system can be used in the more restrictive context of objects. In this section, we show how to encode Vasconcelos' calculus of objects (Vasconcelos and Tokoro 1993) and we compare the types obtained by both systems. We could also compare our approach with the works of Yonezawa (Kobayashi and Yonezawa 1994). But this work is quite similar with Vasconcelos' one, therefore our comparison also holds for this work.

5.1 Vasconcelos' calculus of concurrent objects

The syntax of this calculus is given by the following rule:

$$\begin{array}{rcl} P & ::= & a \lhd l(\tilde{v}) \mid a \rhd [l_1(\tilde{x_1}).P_1 \& \cdots \& l_n(\tilde{x_n}).P_n] \mid P_1, P_2 \\ & & \mid \nu x P \mid !a \rhd [l_1(\tilde{x_1}).P_1 \& \cdots \& l_n(\tilde{x_n}).P_n] \mid 0 \end{array}$$

The syntax and semantics are very close to CAP, except that no orphan message is allowed and an object can't change its interface. This object calculus can be encoded in CAP with the function $\mathcal{T}[\![]\!]$ defined by:

$$\begin{array}{l} \mathcal{T}[\![a \lhd l(\tilde{v})]\!] = a \lhd l(\tilde{v}) \\ \mathcal{T}[\![a \rhd [l_1(\tilde{x_1}).P_1 \& \cdots]]\!] = a \rhd [l_1(\tilde{x_1}) = \zeta(\_,\_)\mathcal{T}[\![P_1]\!] \cdots] \\ \mathcal{T}[\![!a \rhd [l_1(\tilde{x_1}).P_1 \& \cdots]]\!] = a \rhd [l_1(\tilde{x_1}) = \zeta(e_1, s_1)(e_1 \rhd s_1 \parallel \mathcal{T}[\![P_1]\!]) \cdots] \\ \mathcal{T}[\![P_1, P_2]\!] = \mathcal{T}[\![P_1]\!] \parallel \mathcal{T}[\![P_2]\!] \\ \mathcal{T}[\![\nu x P]\!] = \nu x(\mathcal{T}[\![P]\!]) \\ \mathcal{T}[\![0]\!] = \phi \end{array}$$

5.2 Comparison of the two systems

As the interface of objects does not change, the flattening operation is equivalent to an intersection. In this case, the type constraints resolution detects all the messages that cannot be understood. Our system is more powerful than Vasconcelos' one because it uses subsumption instead of kinded types. Kinds express constraints, but they are limited to the level of interface; for the arguments of messages type equality is required. The following example cannot be typed with Vasconcelos' type system:

$$\begin{array}{ll} & !b \rhd [quest(x).(x \lhd rep())] \\ , & !a \rhd [rep().0] \\ , & b \lhd quest(a) \\ , & !c \rhd [rep().0 \,\&\, mess().0] \\ , & b \lhd quest(c) \end{array}$$

The typing process fails because the two messages $b \lhd quest(a)$ and $b \lhd quest(c)$ contain arguments of different types. But this program can be executed without error because the object b only requires an argument that can, at least, accept the message $rep()$. The translation in CAP of this example can be typed because the system only requires that a and c accept $rep()$ and this requirement is satisfied.

What we want to show with this simple example is that subsumption seems better adapted for concurrent programming than kinded types.

CONCLUSION

In this paper we have presented a static analysis based on type reconstructions using constraints for our primitive actor calculus (CAP). The main problem was the mix of indeterminism and behavior changes; for this purpose, we have define a flat union that allows to deal with all the possible futures of a behavior. This system can statically detect some orphan messages, but some of them are still hidden for the types. We have shown how to use the computed types (an abstract version) to dynamically detect the remaining orphans. The advantages of this approach are: *soft and expressive types*, *no type declaration* and a *relatively low cost analysis*. A prototype of the analysis has been implemented in CaML-Light.

We have also proposed in (Colaço et al. 1997) an analysis of linearity inspired by (Kobayashi, Pierce and Turner 1996). This analysis checks that one name identifies at last one actor; which corresponds to the usual actor discipline in writing CAP expressions.

The complete detection of orphan message needs a more precise abstraction than the presented types. We are working on an extension of structure of types presented here that detect every orphan, but which introduces some limitations on the use of behavior changes.In order to solve this problem, others static analysis techniques like *abstract interpretation* or *effect systems* have to be explored; works on their use in the context of concurrent computation have already been developed (Andreoli, Pareschi and T.Castagnetti 1993, Nielson and Nielson 1993).

REFERENCES

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

Aiken, A. and Wimmers, E.: 1993, Type inclusion constraints and type inference, *Proc. of the ACM Symp. on FPCA*.

Aiken, A., Wimmers, E. and Lakshman, T.: 1994, Soft typing with conditional types, *Proc. of the 21st. ACM Symp. on PoPL*.

Andreoli, J.-M., Pareschi, R. and T.Castagnetti: 1993, Abstract interpretation of linear logic programming, *Proc. of the International Logic Programming Symposium*, pp. 315–334.

Cartwright, R. and Fagan, M.: 1991, Soft typing, *Proc. of the ACM Symp. on PLDI*.

Colaço, J.-L., Pantel, M. and Sallé, P.: 1996, CAP: An actor dedicated process calculus, *ECOOP'96 Workshop on Proof Theory of Concurrent Object-Oriented Programming*.

Colaço, J.-L., Pantel, M. and Sallé, P.: 1997, Analyse de linéarité par typage dans un calcul d'acteurs, *Actes des Journées Francophones des Langages Applicatifs*.

Colaço, J.-L., Pantel, M., Sallé, P. and Senteni, A.: 1996, Un calcul d'acteurs primitifs (CAP), *Actes des Journées Francophones des Langages Applicatifs*, pp. 25–43.

Fuh, Y.-C. and Mishra, P.: 1989, Polymorphic subtype inference, closing the theory-practice gap, *Proc. of the Symp. on TAPSOFT*.

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

Honda, K. and Tokoro, M.: 1991, An object calculus for asynchronous communication, *in* P. America (ed.), *Proceedings ECOOP '91*, LNCS 512, Springer-Verlag, Geneva, Switzerland, pp. 133–147.

Kobayashi, N., Pierce, B. C. and Turner, D. N.: 1996, Linearity and the pi-calculus, *Proceedings of the ACM Symposium on Principles of Programming Languages*.

Kobayashi, N. and Yonezawa, A.: 1994, Type-theoretic foundations for concurrent object-oriented programming, *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pp. 31–45.

Marcoux, A., Maurel, C. and Sallé, P.: 1988, A language for distributed applications, *IEEE Workshop on Future Trends of Distributed Systems in the 90's*.

Milner, R.: 1991, The polyadic π-calculus: a tutorial, *Technical Report ECS-LFCS-91-180*, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.

Nielson, F. and Nielson, H.: 1993, From cml to process algebras, *CONCUR'93*, LNCS 715, pp. 493–508.

Pantel, M.: 1994, *Représentation et Transformation : Un modèle de la réutilisabilité dans les langages fonctionnels à objets*, PhD thesis, Institut National Polytechnique de Toulouse.

Pierce, B. and Sangiorgi, D.: 1995, Typing and subtyping for mobile processes, *Mathematical Structures in Computer Science* .

Puntigam, F.: 1996, Type for active objects based on trace semantics, *Proc. of Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*, pp. 5–20.

Vasconcelos, V. T. and Tokoro, M.: 1993, A typing system for a calculus of objects, *Proceedings of the International Symposium on Object Technologies for Advanced Software*, LNCS 742, Springer-Verlag, pp. 460–474.

Yonezawa, A.: 1990, *ABCL: An Object-Oriented Concurrent System*, MIT Press.