
Typing Concurrent Objects and Actors

Fabien Dagnat — Marc Pantel — Matthias Colin — Patrick Sallé

*ENSEEIH*T — Institut de Recherche en Informatique de Toulouse
LIMA, 2 rue Camichel
31071 Toulouse Cedex
{dagnat, pantel, colin, salle}@enseeiht.fr

ABSTRACT. As informal methods do little to help their use for concurrent and distributed programming, one of the most challenging current tasks is to build tools based on formal method in order to ease the development of such applications. In this context, we propose precise type systems for concurrent object and actor oriented programming languages. This paper analyzes the type systems that we have developed for realistic languages and presents their benefits and limits. First, it introduces the kernel of an actor language used to describe and validate the proposed technic. Then, the errors we wish to detect are precisely defined. Afterwards, the first type system is defined and its properties are presented. Finally, insights are given on a second more powerful type system, which is the basis of our future work.

RÉSUMÉ. Face à l'échec des méthodes informelles dans l'assistance à la programmation concurrente et distribuée, un des enjeux majeurs actuels consiste à fournir des outils d'aide basés sur des méthodes formelles. Dans ce contexte, nous proposons des systèmes de type précis et pertinents pour des langages à objets concurrents ou à acteurs. Cet article est un premier bilan des systèmes que nous avons mis au point pour des langages réalistes, il expose à la fois leurs avantages et leurs limites. Pour présenter ces systèmes de type, nous introduisons le noyau d'un langage d'acteurs qui sert de support pour valider les techniques proposées. Nous décrivons ensuite, de manière approfondie, les types d'erreurs que nous souhaitons détecter. Nous définissons un premier système de type et présentons ses propriétés. Enfin, nous terminons sur une brève introduction d'un second système de type plus puissant, base de nos futurs travaux.

KEYWORDS: Actors, Concurrent Objects, Static analysis, Typing, Inference

MOTS-CLÉS : Acteurs, Objets Concurrents, Analyse Statique, Typage, Inférence

1. Introduction

The development of the telecommunications industry and the generalization of network use bring concurrent and distributed programming into the limelight. In that context, programming is a hard task and, generally, the resulting applications contain many more *bugs* than usual centralized software. As sequential object oriented programming is commonly accepted as a *good* way to build software, concurrent object oriented programming seems to be well-suited for programming distributed systems. Since indeterminism resulting from the unreliability of networks makes it difficult to validate any distributed functionality using informal approaches, our work is focused on applying formal type systems to improve concurrent object oriented programming.

To obtain widely usable tools, we have chosen to use the actor model proposed by Hewitt in [HEW 73] and developed by Agha in [AGH 86]. This model is based on a network of autonomous and cooperative agents (called actors), which encapsulate data and programs, communicating using an asynchronous point to point protocol. An actor store each received message in a queue and when idle, it handles the first message it can treat in this queue. Besides those conventions (which are also true for concurrent object), an actor can dynamically change its interface. This property allows to increase or decrease the set of messages an actor may handle, yielding a more accurate programming model. This model, also known as concurrent objects with non uniform behavior (or interface), has been adopted by the telecommunications industry for the development of distributed and concurrent applications for the Open Distributed Computing framework (ITU X901-X904) and the Object Description Language (TINA-C extension of OMG IDL with multiple interfaces). Due to behavior change, it may happen that a message cannot be handled by its target in some execution path (a sequence of behaviors the actor can assume) and could be handled in some other path. Such messages, called orphan messages, may be of two kinds: safety ones or liveness ones. A safety orphan is a message that cannot be handled in **all** of its target future behaviors. A liveness orphan is a message that will not be treated because its recipient will not reach one of the behaviors which can handle it. Such messages correspond to deadlocks and are not the goal of this paper (apart from their semantic characterization), we concentrate on the static detection of safety orphans.

Type systems for concurrent objects and actors, with uniform or non-uniform behaviors, have been the subject of active research in the last years ([VAS 93], [NIE 95], [KOB 94], [KOB 95], [PUN 96], [NAJ 97], [RAV 97], [FOU 97], [DAL 97] and their more recent works). Two opposite approaches have been followed: explicit and implicit typing. Explicit types may provide more precise information but are sometimes very hard to write for the programmer (they might be much more complex than the program itself). We advocate the use of implicit typing, i.e. type inference, as it is simpler to use. To our knowledge little work had been done to develop inference system in presence of non-uniform behaviors and our type abstractions and constraint resolution algorithm provide more precise information than the works reported in the dedicated literature as will be shown later on in the following sections.

In a first approach, our type systems were defined using CAP, an actor calculus derived from asynchronous π -calculus and Cardelli's Calculus of Primitive Objects (see [COL 96]). Two type systems were developed: the former, used in this paper (see [COL 97b]), is based on usual object type abstractions and catches all functional and communication errors but only trivial safety orphans, the latter, mentioned at the end of the paper, detects all safety orphans but requires a much more complex type abstraction (see [COL 99a]).

In order to validate those theoretical systems practically, the need for an implementation on a programming language arose. To have a full and easy access to the internals of the compiler, we have chosen to develop our own language ML-ACT, (see [DAG 97] and [COL 98b]) integrating *à la ML* programming with actor primitives. The first type system for ML-ACT was inspired by Nielson's et al.'s works ([NIE 93], [NIE 96], [NIE 97]). It consisted in the joint synthesis of a concurrent effect (a CAP term) and a usual ML type for each expression. Then the concurrent effect was analyzed using CAP type systems. It appeared however that this strategy did not allow enough precision in concurrent types to detect orphans (or at the cost of forbidding effects in recursive functions and data structures). Indeed, functional and concurrent aspects are too mixed to be treated separately. So, as a second approach, we adapted CAP typing system to the full ML-ACT language.

This paper describes the results of this adaptation. First, it introduces the required syntax and semantics including the precise description of errors. Then, a typing system is formally presented and discussed. Finally, a second type system, currently being integrated in the compiler, is evoked and insights on our future work are given.

2. A simplified version of ML-Act

This section introduces ML-ACT's syntax, its illustration via an example, and its semantics. As typing is our main goal, the presentation of ML-ACT will be rather short.

2.1. Syntax

ML-ACT is a complete programming language, but for the sake of simplicity, only the required syntax is introduced in this paper (a full implementation has been developed, see [COL 98b]). It is an extension of CaML, its expressions keep their usual semantics and only specific constructions to manipulate actors have been introduced: actor's creation, suicide, change of behavior, sending of messages, and reference to ego (the actor address) and self (the actor current behavior). This added syntax is inspired by the Objective CaML syntax for classes and objects. ML-ACT programs are built from the EBNF grammar in Figure 1. Rules follow usual conventions. To make them easier to read, *non-terminals* are in italics, keywords and reserved symbols are in typewriter face. The only terminal contained in the grammar is IDENT (usual identifiers), but in the rest of the paper we use all classical terminals (such as integer, string ...) and

some basic operators (such as `+`, `print_int`, ...). Those data structures and operators are represented by the *Const* non-terminal (whose derivation is not given).

```

Prog ::= [Exp ; ; | behavior Beh [and Beh]*end ; ;]*
Beh  ::= [IDENT]+ = [message [IDENT]+ = Exp]+
Exp  ::= (Exp) | ego | self | Const | IDENT | Exp [Exp]+ | Exp [ ; Exp ]* |
         let [rec IDENT][IDENT]+ = Exp in Exp | if Exp then Exp else Exp |
         send IDENT [(Exp [, Exp]*)] to Exp | new Exp | become Exp | suicide

behavior empty =
  message init v = become (cell v)
and cell v =
  message get c = send prn(v) to c
  message set v' = become (cell v')
and screen =
  message prn v = print_int v
end ; ;

let a_cell = new empty in
let a_scr = new screen in
  send init(1) to a_cell;
  send get(a_scr) to a_cell;
  send set(2) to a_cell;
  send get(a_scr) to a_cell ; ;

```

Figure 1. ML-ACT grammar and a linear cell example

The expressions derived directly from *Prog* are called top-level expressions. They are the only expressions which are not part of an actor behavior, therefore, they cannot refer to `ego` or `self`.

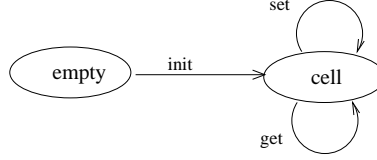
The actor approach is a kind of generalization of concurrent objects without inheritance. We preferred this model because it allows to type actor programs, concurrent short lifetime object programs (only containing `suicide`) as well as usual objects (with uniform behavior, i.e. containing no `become` or `suicide`).

In order to illustrate this grammar, let us examine a small example before studying ML-ACT semantics. The cell (Fig 1) consists in the definition of three behaviors:

- `empty`, which waits for an `init` message to start stocking a value.
- `cell`, which can send its value to a continuation `c` when receiving a `get` message, and which can be restarted with a new value with a `set` message.
- `screen`, which outputs the value it gets from a `print` message (we assume that `print_int` is predefined).

Then, two actors are created using the `new` operator: one for the cell with initial behavior `empty` and one for the screen with initial behavior `screen`. And finally four messages are sent to the cell actor.

ML-ACT semantics has not been formally exposed yet, but the execution of this program consists intuitively in creating two processes and asynchronously sending four messages to them. The set of behaviors assumed by a given actor can be represented by a regular automaton whose nodes are behaviors, transitions are tagged by accepted messages and the initial state is the actor initial behavior. The following automaton describes all possible behaviors of `a_cell`.



As emission order may not be preserved by the medium, many executions are possible depending on the order in which the `a_cell` actor receives `get` and `set` messages. Indeed, the `init` is always treated first because it is the unique transition leaving the initial state. All executions of the system consist in printing on the screen 1 then 2, 2 then 2 or 1 then 1.

2.2. Semantics

ML-ACT has a reduction semantic, but before giving its formal definition, we need to introduce some machinery.

2.2.1. Conventions

First, let's introduce some necessary sets which we assume to be disjoint:

- \mathbb{M} and \mathbb{V} , respectively set of message labels and set of variables. Those two sets are constructed by collecting identifiers from the source code. $m, m_1 \dots$ range over \mathbb{M} and $x, y \dots$ range over \mathbb{V} .
- \mathbb{A} , set of actor addresses. Each execution of a `new` instruction generates a new address. This strategy guarantees the unicity of actors' names in a configuration. $a, a' \dots$ range over \mathbb{A} . In a first phase, we tag all `new` instructions in a program with a fresh name (new_a). This process also produces a mapping γ associating a name to a type. The type $\gamma(a)$ (detailed in 4.2) is used for every actor created by new_a .
- \mathbb{I} , set of interfaces, made of all possible behavior interfaces of a program. An interface \mathcal{R} is a partial mapping from \mathbb{M} to $\mathbb{V}^n \times \text{Exp}$, binding a message label to the reaction of the actor (a ML-ACT expression) with its arguments. The domain of an interface is therefore the set of messages it can handle.
- \mathbb{V} , set of semantic values, consists in all possible results of expressions. A semantic value is either an address (from \mathbb{A}), a constant (data from *Const*), an interface (from \mathbb{I}), a message (from $\mathbb{M} \times \mathbb{V}^n$), a behavior closure (from $\mathbb{V}^n \times \mathbb{I}$), a functional closure (from $\mathbb{V}^n \times \text{Exp}$) or an error (*Err*, *Saf*, *Liv*). The closures are functions taking n arguments and resulting in either an expression or an interface. Their set is \mathbb{C} and they are denoted $\langle \vec{x}; X \rangle$. $v, v_1 \dots$ range over \mathbb{V} .

We assume that an initial parsing of the program constructs all these sets. Moreover, to simplify the description of semantic rules, we build a global environment (denoted \mathcal{E}) which contains behavior definitions and verify that those behavior closures are well-formed (i.e. reaction to a given label is unique). This collecting phase may contain

some renaming to prevent behavior name clashes. The cell example presented in Figure 1, gives:

$$\begin{cases} \mathbb{M} = \{\text{init}, \text{get}, \text{set}, \text{prn}\} \\ \mathbb{V} = \{\text{empty}, \text{cell}, \text{screen}, v, v', c, \text{print_int}, \text{a_cell}, \text{a_scr}\} \\ \mathcal{R}_1 = \{\text{init} \mapsto \langle v; \text{become}(\text{cell } v) \rangle\} \\ \mathcal{R}_2 = \{\text{get} \mapsto \langle c; \text{send prn}(v) \text{ to } c \rangle; \text{set} \mapsto \langle v'; \text{become}(\text{cell } v') \rangle\} \\ \mathcal{R}_3 = \{\text{prn} \mapsto \langle v; \text{print_int } v \rangle\} \\ \mathbb{I} = \{\mathcal{R}_1; \mathcal{R}_2; \mathcal{R}_3\} \\ \mathcal{E} = \{\text{empty} \mapsto \mathcal{R}_1; \text{cell} \mapsto \langle v; \mathcal{R}_2 \rangle; \text{screen} \mapsto \mathcal{R}_3\} \end{cases}$$

Finally, the code contains only top-level expressions and its evaluation consists in their parallel evaluation (because there is no dependency between them). The evaluation of ML-ACT expressions is modeled by two different reductions: the functional reduction (the usual application of functions which occurs in independant concurrent processes) and the communication reduction (the exchange of messages between processes).

Functional reduction uses the usual notion of evaluation context to model the fact that an actor instruction executed inside a functional expression (the context of the instruction) may produce an effect on the communication medium (for example, it may create a new actor). Such an evaluation context consists in an expression with a unique *hole* $C[]$, it is built by the grammar in Figure 2. The hole indicates where the evaluation process acts. When we fill in this hole ($C[e]$), e corresponds to the expression being currently calculated. Priority and associativity are the same as in the ML language. The evaluation order is constrained by the context to be *leftmost/outermost*, i.e. the usual *call by value*. In particular, the message arguments are evaluated in leftmost order (the corresponding rule will not be given). This specific order was chosen because it is usual among the ML family of languages.

$$\begin{aligned} C ::= [] \mid C; Exp \mid \text{if } C \text{ then } Exp \text{ else } Exp \mid C Exp \mid v C \mid \text{send } m(F) \text{ to } Exp \mid \\ \text{send } m(v_1, \dots, v_n) \text{ to } C \mid \text{become } C \mid \text{new } C \mid \text{let } [\text{rec } x] x_1 \dots x_n = C \text{ in } Exp \\ F ::= [] \mid F, Exp \mid v, F \end{aligned}$$

Figure 2. Context grammar

The communication medium (a group of actors being concurrently executed and messages on their way to their destination) is modeled by the notion of configuration. As configurations express the concurrent parts of the reduction of a program, their definition is inspired by asynchronous π -calculus (more precisely, by the Calculus of Primitive Actors used for the initial definition of our type systems [COL 96]). Configurations could have been encoded using pure π -calculus. In this case, actor addresses and message labels would both be encoded using channel names. The type system for the resulting terms would therefore be unable to distinguish an address type from a message type which is necessary for our kind of analyzes. The set of configurations is built from the grammar of Figure 3. The two last configurations are called functional configurations. It is their reduction that uses evaluation context notion to simulate

functional execution. In the actor reaction configuration, the subscript gives the actor address (its `ego`) and its interface (its `self`). Furthermore, in the rest of the paper, configurations are used modulo the congruence \equiv (see Fig 3) which expresses their properties:

- The parallel operator is commutative, associative and ϵ is its neutral element.
- Any error occurring is propagated and causes the end of execution.

$w ::= \epsilon$		<i>do nothing</i> configuration
<code>Err</code>		execution error
$w \parallel w$		parallel execution
$a \triangleleft m(\vec{v})$	message m with arguments \vec{v} in transit toward a	
$a \triangleright \mathcal{R}$		actor a waiting for a <i>valid</i> message
$\llbracket e \rrbracket_{a \triangleright \mathcal{R}}$	reaction of a to a received message, executing the expression e	
$\llbracket e \rrbracket$		execution of a top-level expression e
$w \parallel \epsilon \equiv w$	$w \parallel \text{Err} \equiv \text{Err}$	$(w_1 \parallel w_2) \parallel w_3 \equiv w_1 \parallel (w_2 \parallel w_3)$
$w_2 \parallel w_1 \equiv w_1 \parallel w_2$		$w_1 \equiv w_2$ if w_1 is α -convertible in w_2

Figure 3. Configuration grammar and congruence on configurations

The result of the application of these translations to the cell example is the configuration: $\llbracket \text{let } a_cell = \text{new}_a \text{ empty in...} \rrbracket$ that will be reduced using the rules given in next subsection.

2.2.2. Reduction rules

Reduction rules given in Figures 4 and 5 are separated into two families. The first one concerns concurrent operations on configurations (i.e. communications and behavior change) and the second one is composed of rules about *functional* configurations. In some rules, a functional configuration appear with a “_” subscript meaning that it may be a top-level execution or an execution occurring in an actor. This notation will be used when information about current *actor* is not required in order to execute a reduction step.

The (PAR) rule specifies that disjoint sub-configurations can be reduced concurrently. References to `ego`, resp. `self`, can only appear in an actor reaction resulting in his own address (EGO), resp. behavior (SELF), otherwise an error is raised (EGOE, SELFE). When an actor receive a message, the corresponding reaction must have the same arity or it leads to an error (see REAE). If this is true, (REA) starts functional computations. Notice that, if the message label is not in the interface domain, no error occurs and the configuration is unchanged. Therefore, an actor waits until it receives a message that it can handle, and a message stays in the communication medium until its target assumes a behavior that can handle it. Such messages may be orphans if their target never assumes such a behavior, their formal definition is stated in section 3. To send a message, the recipient must be valid (SENE) and it add a new term (the message) to

$$\begin{array}{l}
\text{PAR: } \frac{w_1 \rightarrow w_2}{w \parallel w_1 \rightarrow w \parallel w_2} \\
\text{REA: } \frac{\mathcal{R}(m) = \langle \vec{x}; e \rangle \quad \phi = [\vec{v}/\vec{x}]}{a \triangleright \mathcal{R} \parallel a \triangleleft m(\vec{v}) \rightarrow \llbracket \phi(e) \rrbracket_{a \triangleright \mathcal{R}}} \\
\text{SELF: } \frac{}{\llbracket \mathbf{C}[\mathbf{self}] \rrbracket_{a \triangleright \mathcal{R}} \rightarrow \llbracket \mathbf{C}[\mathcal{R}] \rrbracket_{a \triangleright \mathcal{R}}} \\
\text{SEND: } \frac{}{\llbracket \mathbf{C}[\mathbf{send } m(\vec{v}) \text{ to } a] \rrbracket_{_} \rightarrow \llbracket \mathbf{C}[_] \rrbracket_{_} \parallel a \triangleleft m(\vec{v})} \\
\text{ACT: } \frac{b \text{ fresh} \quad \gamma \leftarrow \gamma :: \{b \mapsto \gamma(a)\}}{\llbracket \mathbf{C}[\mathbf{new}_a \mathcal{R}] \rrbracket_{_} \rightarrow \llbracket \mathbf{C}[b] \rrbracket_{_} \parallel b \triangleright \mathcal{R}} \\
\text{SUIE: } \frac{}{\llbracket \mathbf{C}[\mathbf{suicide}] \rrbracket \rightarrow \text{Err}} \quad \text{BEC1: } \frac{}{\llbracket \mathbf{C}[\mathbf{become } e] \rrbracket \rightarrow \text{Err}} \quad \text{BEC2: } \frac{v \notin \mathbb{I}}{\llbracket \mathbf{C}[\mathbf{become } v] \rrbracket_{_} \rightarrow \text{Err}} \\
\text{SUI: } \frac{}{\llbracket \mathbf{C}[\mathbf{suicide}] \rrbracket_{a \triangleright \mathcal{R}} \rightarrow \epsilon} \quad \text{END: } \frac{}{\llbracket v \rrbracket_{a \triangleright \mathcal{R}} \rightarrow a \triangleright \mathcal{R}} \quad \text{BEC: } \frac{}{\llbracket \mathbf{C}[\mathbf{become } \mathcal{R}_1] \rrbracket_{a \triangleright \mathcal{R}} \rightarrow a \triangleright \mathcal{R}_1} \\
\text{REAE: } \frac{\mathcal{R}(m) = \langle \vec{x}; e \rangle \quad lg(\vec{x}) \neq lg(\vec{v})}{a \triangleright \mathcal{R} \parallel a \triangleleft m(\vec{v}) \rightarrow \text{Err}} \\
\text{REFE: } \frac{x \in \{\mathbf{ego}, \mathbf{self}\}}{\llbracket \mathbf{C}[x] \rrbracket \rightarrow \text{Err}} \\
\text{EGO: } \frac{}{\llbracket \mathbf{C}[\mathbf{ego}] \rrbracket_{a \triangleright \mathcal{R}} \rightarrow \llbracket \mathbf{C}[a] \rrbracket_{a \triangleright \mathcal{R}}} \\
\text{SENE: } \frac{w \notin \mathbb{A}}{\llbracket \mathbf{C}[\mathbf{send } m(\vec{v}) \text{ to } w] \rrbracket_{_} \rightarrow \text{Err}} \\
\text{ACTE: } \frac{v \notin \mathbb{I}}{\llbracket \mathbf{C}[\mathbf{new}_a v] \rrbracket_{_} \rightarrow \text{Err}}
\end{array}$$

Figure 4. Semantic rules, concurrent aspects

the current configuration (SEND). The next two rules concern actor creations, the given expression must evaluate to a behavior (ACTE). In this case (ACT), we create a new name for the actor, and associate it with the type of the \mathbf{new}_a 's tag in γ . This name is used as the result of the \mathbf{new} expression and the newly created actor is added to the configuration.

The following three rules (SUIE, BECE1 and BECE2) describe errors in behavior change (illegal reference to current actor or changing to something which is not a behavior). Functional evaluation may end in three ways: a suicide (SUI), a behavior change (BEC) or the end of computation (END). In the first alternative, the current actor dies, in the second, it assumes its new behavior stopping the current calculation. Finally, the end of functional computation consists in keeping the same behavior for the current actor.

Notice that we do not exactly follow the actor's tradition. Usually, the code following a behavior change is computed by an anonymous actor that dies just after this execution (our choice was made to have similar semantics for all *behavior change* — suicide, change and implicit recursion). There is also another difference: by default, one of our actors keeps its behavior at the end of its reaction (as does a concurrent object), while generally in other models, it dies. These choices have no consequence on the typing of our system also applies to the usual actor model. In this hypothesis, suicide would become useless and recursion would be explicit.

The use of context to present (relatively) simple reduction rules arise in the rules SEND, ACT, SUI and BEC because they modify the communication medium from deep inside a functional context (by adding, erasing or modifying the current configuration).

$$\begin{array}{c}
\text{IFE: } \frac{v \notin \{\text{true}, \text{false}\}}{\llbracket \text{C}[\text{if } v \text{ then } e_1 \text{ else } e_2] \rrbracket_- \rightarrow \text{Err}} \qquad \text{IFT: } \frac{}{\llbracket \text{C}[\text{if true then } e_1 \text{ else } e_2] \rrbracket_- \rightarrow \llbracket \text{C}[e_1] \rrbracket_-} \\
\\
\text{SEQ: } \frac{}{\llbracket \text{C}[v; e] \rrbracket_- \rightarrow \llbracket \text{C}[e] \rrbracket_-} \qquad \text{IF: } \frac{}{\llbracket \text{C}[\text{if false then } e_1 \text{ else } e_2] \rrbracket_- \rightarrow \llbracket \text{C}[e_2] \rrbracket_-} \\
\text{APP1: } \frac{\phi = [\vec{v}/\vec{x}]}{\llbracket \text{C}[\langle \vec{x}; e \rangle \vec{v}] \rrbracket_- \rightarrow \llbracket \text{C}[\phi(e)] \rrbracket_-} \qquad \text{APP2: } \frac{m = \text{lg}(\vec{v}) < \text{lg}(\vec{x}) \quad \phi = [\vec{v}/\vec{x}]}{\llbracket \text{C}[\langle \vec{x}; e \rangle \vec{v}] \rrbracket_- \rightarrow \llbracket \text{C}[\langle x_{m+1}, \dots, x_n; \phi(e) \rangle] \rrbracket_-} \\
\text{APPE: } \frac{v \notin \mathcal{C}}{\llbracket \text{C}[v \ \bar{e}] \rrbracket_- \rightarrow \text{Err}} \qquad \text{LETR: } \frac{\phi = [\langle \vec{x}; \text{let rec } f \ \vec{x} = e_1 \text{ in } e_1 \rangle / f]}{\llbracket \text{C}[\text{let rec } f \ \vec{x} = e_1 \text{ in } e_2] \rrbracket_- \rightarrow \llbracket \text{C}[\phi(e_2)] \rrbracket_-} \\
\text{LETV: } \frac{}{\llbracket \text{C}[\text{let } x = v \text{ in } e] \rrbracket_- \rightarrow \llbracket \text{C}[[v/x]e] \rrbracket_-} \qquad \text{LETF: } \frac{\phi = [\langle \vec{x}; e_1 \rangle / f]}{\llbracket \text{C}[\text{let } f \ \vec{x} = e_1 \text{ in } e_2] \rrbracket_- \rightarrow \llbracket \text{C}[\phi(e_2)] \rrbracket_-}
\end{array}$$

Figure 5. Semantic rules, functional aspects

Finally Figure 5 describes usual functional evaluation (*à la ML* semantics).

Some reduction steps from a possible reduction of our basic example (Fig 1) illustrate this semantics. It uses relation \rightarrow^* that stands for multiple reduction steps:

$$\begin{aligned}
& \llbracket \text{let } a_cell = \text{new}_a \text{ empty in } \dots \rrbracket \rightarrow \\
& \llbracket \text{let } a_cell = \text{new}_a \mathcal{R}_1 \text{ in } \dots \rrbracket \rightarrow \\
& a_1 \triangleright \mathcal{R}_1 \parallel \llbracket \text{let } a_cell = a_1 \text{ in } \dots \rrbracket \rightarrow^* \\
& a_1 \triangleright \mathcal{R}_1 \parallel \llbracket [a_1/a_cell] \text{let } a_scr = \text{new}_b \mathcal{R}_3 \text{ in } \dots \rrbracket \rightarrow^* \\
& a_1 \triangleright \mathcal{R}_1 \parallel b_1 \triangleright \mathcal{R}_3 \parallel \llbracket [b_1/a_scr] \text{send init}(1) \text{ to } a_1 ; \dots \rrbracket \rightarrow^* \\
& a_1 \triangleright \mathcal{R}_1 \parallel a_1 \triangleleft \text{init}(1) \parallel \dots \rightarrow \dots
\end{aligned}$$

3. Orphan messages

Several kinds of errors may occur in ML-ACT programs. The previous reduction has described ill-formed expression errors (applying a value which is not a function, sending a value which is not a message ...), and a first kind of *concurrent* errors related to communication: arity mismatch between a message and its target behavior. These errors, denoted **Err**, correspond to the usual ones in concurrent calculi (see [MIL 91], [BOU 97] and [FOU 96]), in concurrent objects ([VAS 93]) or in actors ([KOB 94]). The second kind of concurrent errors is related to the absence of communications: orphan messages. Such messages will not be treated by their target and will stay indefinitely in the communication medium. In the context of objects, orphan messages correspond to the usual *message not understood* error. In the context of actors, the matter gets complicated by the possible behavior changes: a message, which cannot be taken into account by the current behavior of its target, may be handled by one of its future behaviors. Several strategies have been proposed in the literature:

– either this situation is considered as an error, although the message could be handled in the future (see [AGH 86] and [VAS 93]),

– or the message stays in the medium until it can be handled by a future behavior, as proposed by the previous semantic relation (see also [KOB 94] and [BOU 97]).

We advocate that the first approach is too restrictive because it requires excessive explicit synchronization in the program (for example, you must verify that a device has been initialized before sending input/output requests, instead of sending requests which are postponed by the system until the device is ready). The second usual proposal is too permissive, since it leaves the message in the communication medium without signaling an error. Even if this message can never be handled because either the target is deadlocked, or the effective sequence of behaviors taken by the target will not include a behavior which can accept this message.

One of the novelties of this work is a compromise between both approaches: we advocate that a system should be as permissive as possible to ease the programmers' work, but that a static type system should reject any incorrect programs (users need not check that devices have been initialized before sending requests, however errors will be signaled if requests are sent to devices that will never be initialized). In our previous work, we only gave an intuitive and partial characterization of orphans. The following part will provide a formal account of orphans using an extension of the previous semantics.

This extension will be defined in two steps: first, we will give a static characterization of orphans at the end of a finite computation, and then we will extend this characterization during reductions (including infinite ones).

3.1. *Static orphan detection*

In order to characterize orphans, we will first consider remaining messages at the end of the evaluation of a program. The end of a computation is characterized by using silent configuration. A configuration is silent when no more reduction rules, either concurrent or functional, can be applied to it. Notice that congruence rules can be still applied to silent configurations but are not considered as reductions. The silent configuration set is built from the following grammar:

$$s = \epsilon \mid \mathbf{Err} \mid s \parallel s \mid a \triangleleft m(\vec{v}) \mid a \triangleright \mathcal{R}$$

Silent configurations represent the result of the computation of an ML-ACT program. The following well-formedness condition ensures that no more communication can take place, which means that all remaining sent messages cannot be handled and are therefore orphans ($a \triangleleft m(\vec{v}) \in s \wedge a \triangleright \mathcal{R} \in s \Rightarrow m \notin \text{dom}(\mathcal{R})$). Their occurrence can thus be considered as a semantic error which will be written **Orph**. The following rule applicable on silent configurations should be added to the semantics:

$$\text{SORPHE:} \frac{}{a \triangleleft m(\vec{v}) \parallel s \rightarrow \mathbf{Orph}}$$

Notice that $a \triangleright \mathcal{R}$ may appear in s but, as s is well-formed, $m \notin \text{dom}(\mathcal{R})$.

The well-formedness condition relies only on the actor current behavior, whereas it is possible that future behaviors may be able to handle m . This remark introduces two kind of orphan messages:

- **Safety orphan:** *None of the future behaviors can handle this message.*
- **Liveness orphan:** *At least one of the future behaviors can handle m , but as the configuration is silent, no reduction will take place and, therefore, the actor will not assume any of its future behaviors.*

A special kind of safety orphans can be detected easily, messages which will never be understood by any of the possible behaviors of an actor. This kind of message are called **trivial safety orphans**.

Orphan messages can be characterized using the notion of actor message potential.

3.2. Actor message potential

An actor message potential is an upper approximation of the set of messages that can be handled by a given actor, i.e. the messages that can be handled by its current behavior and by its potential future behaviors. When an actor changes its behavior, it may assume a new behavior which has been sent to it by another actor as a message parameter. In a closed computing environment, the potential can be computed by taking into account all behavior-carrying messages sent to the actor (this is done in the following type systems). In an open computing environment, the potential cannot be computed and all message labels will therefore be in the potential (it is called an *open potential* and its value is \mathbb{M}).

Figure 6 defines the notion of actor message potential. \mathcal{P} is applied to an actor and a configuration. It calls \mathcal{P}_r which approximates a reaction \mathcal{R} using a set \mathcal{I} to store already abstracted behavior (a kind of occur check). Finally, we define two operators V and E , respectively computing the value set and the effect set of an expression (B is used instead of V and E when the definition of both operators are equals). The value set contains the message labels that the expression understands (the result of such an expression will be a reaction). The effect set computes the possible changes of behavior the expression contains. The `become` instruction connects the two sets. An actor's potential must be re-evaluated each time the actor changes its behavior.

Using this notion of potential, orphans in a silent configuration can now be defined more precisely:

- **Safety orphan:** *An orphan message is a safety orphan, denoted saf , if and only if it is not in its target message potential ($SSAFE$).*
- **Liveness orphan:** *An orphan message is a liveness orphan, denoted liv , if and only if it is in its target potential ($SLIVE$).*

$$\begin{aligned}
\mathcal{P}(a, \epsilon) &= \mathcal{P}(a, \llbracket e \rrbracket) = \mathcal{P}(a, x \triangleleft m(\vec{v})) = \emptyset & \mathcal{P}(a, w_1 \parallel w_2) &= \mathcal{P}(a, w_1) \cup \mathcal{P}(a, w_2) \\
\mathcal{P}(a, [e]_{\mathcal{R}}) &= \mathcal{P}(a, x \triangleright \mathcal{R}) = \begin{cases} \mathcal{P}_r(\{\mathcal{R}\}, \mathcal{R}) & \text{if } x = a \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{P}_r(\mathcal{I}, \mathcal{R}) &= \text{dom}(\mathcal{R}) \cup \bigcup_{m \in \text{dom}(\mathcal{R})} \{E(\mathcal{I}, [\vec{M}/\vec{x}][\emptyset/\text{ego}]e) \mid \mathcal{R}(m) = \langle \vec{x}; e \rangle\} \\
B(\mathcal{I}, x) &= B(\mathcal{I}, \mathcal{E}(x)) & B(\mathcal{I}, \text{let } x = e_1 \text{ in } e_2) &= B(\mathcal{I}, [B(\mathcal{I}, e_1)/x]e_2) \\
B(\mathcal{I}, \text{suicide}) &= \emptyset & B(\mathcal{I}, \text{let } x \vec{x} = e_1 \text{ in } e_2) &= B(\mathcal{I}, [[\vec{x}; e_1]/x]e_2) \\
B(\mathcal{I}, \text{let rec } x \vec{x} = e_1 \text{ in } e_2) &= B(\mathcal{I}, [\langle \vec{x}; \text{let rec } x \vec{x} = e_1 \text{ in } e_1 \rangle/x]e_2) \\
V(\mathcal{I}, \text{send } e_1 \text{ to } e_2) &= V(\mathcal{I}, \text{new } e) = V(\mathcal{I}, \text{become } e) = V(\mathcal{I}, m(\vec{e})) = \emptyset \\
V(\mathcal{I}, v) &= \begin{cases} v & \text{if } v \in \mathcal{C} \cup \{\mathbb{M}\} \\ \emptyset & \text{otherwise} \end{cases} & V(\mathcal{I}, e \vec{e}) &= \begin{cases} V(\mathcal{I}, [V(\mathcal{I}, \vec{e})/\vec{x}]e') & \text{if } V(\mathcal{I}, e) = \langle \vec{x}; e' \rangle \\ \mathbb{M} & \text{otherwise} \end{cases} \\
V(\mathcal{I}, e_1; \dots; e_n) &= V(\mathcal{I}, e_n) & V(\mathcal{I}, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) &= V(\mathcal{I}, e_2) \cup V(\mathcal{I}, e_3) \\
E(\mathcal{I}, v) &= \begin{cases} \mathbb{M} & \text{if } v = \mathbb{M} \\ \emptyset & \text{otherwise} \end{cases} & E(\mathcal{I}, \mathcal{R}) &= \begin{cases} \emptyset & \text{if } \mathcal{R} \in \mathcal{I} \\ \mathcal{P}_r(\{\mathcal{R}\} \cup \mathcal{I}, \mathcal{R}) & \text{otherwise} \end{cases} \\
E(\mathcal{I}, \text{new } e) &= E(\mathcal{I}, e) & E(\mathcal{I}, \text{send } e_1 \text{ to } e_2) &= E(\mathcal{I}, e_1) \cup E(\mathcal{I}, e_2) \\
E(\mathcal{I}, m(\vec{e})) &= E(\mathcal{I}, \vec{e}) & E(\mathcal{I}, \text{become } e) &= E(\mathcal{I}, e) \cup V(\mathcal{I}, e) \\
E(\mathcal{I}, \mathcal{R}) &= \emptyset & E(\mathcal{I}, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) &= E(\mathcal{I}, e_1) \cup E(\mathcal{I}, e_2) \cup E(\mathcal{I}, e_3) \\
E(\mathcal{I}, e \vec{e}) &= E(\mathcal{I}, e) \cup E(\mathcal{I}, \vec{e}) & E(\mathcal{I}, e_1; \dots; e_n) &= E(\mathcal{I}, e_1) \cup \dots \cup E(\mathcal{I}, e_n)
\end{aligned}$$

Figure 6. Computation of an actor message potential

Therefore, a safety orphan will not ever be taken into account by its target and a liveness one could be treated but will never be, because its target is deadlocked and no other actor can unlock it.

$$\text{SSAFE: } \frac{m \notin \mathcal{P}(a, s)}{a \triangleleft m(\vec{v}) \parallel s \rightarrow \text{Saf}} \quad \text{SLIVE: } \frac{m \in \mathcal{P}(a, s)}{a \triangleleft m(\vec{v}) \parallel s \rightarrow \text{Liv}}$$

Notice that if an actor potential is open, all orphan messages sent to this actor are liveness orphans because the computation must progress in order to provide the actor with new behaviors, which may be able to handle the orphans. In this open world, a new message may reach later the medium and trigger some behavior changes which will promote a liveness orphan to the status of safety orphan.

3.3. Dynamic orphan detection

As usual in concurrent programming, an ML-ACT program must not necessarily reach a silent state (reactive programs). However, liveness and safety orphans can still occur even if computation progresses indefinitely. Therefore, new semantic rules must be introduced in order to detect orphans occurring dynamically during a computation.

$$\text{DSAFE: } \frac{m \notin \mathcal{P}(a, w)}{a \triangleleft m(\vec{v}) \parallel w \rightarrow \text{Saf}} \quad \text{DLIVE: } \frac{m \in \mathcal{P}(a, s) \quad a \notin \mathcal{FN}(w)}{a \triangleleft m(\vec{v}) \parallel s \parallel w \rightarrow \text{Liv}}$$

3.4. Related work

Sangiorgi defined in [SAN 97] the notion of *uniform receptiveness* and an associated type system which detects orphans in the context of π -calculi. The main technical difference with our work is that we introduce the notion of message in order to distinguish the data communicated over a channel. The application of its approach in our context seems to allow the detection of all safety and liveness orphans at the cost of rejecting some correct programs. However, to our knowledge, it requires that actors have an infinite life-time and that they always accept the same set of messages, which are precisely the two requirements that we wish to reject as they constrain actors to behave as usual objects and reject the non-uniform behaviors.

Ravara introduced in [RAV 97] the notion of persistent bad redex which correspond to our orphan messages (both safety and liveness). However, to our knowledge, his semantics does not explicitly compute errors as we proposed in this part.

The purpose of the following type systems is to detect ML-ACT expressions that may reduce to **Err** or **Saf**. The first one detects all common errors (**Err**) and the trivial safety orphan messages (**saf**) and therefore accepts many incorrect programs reducing to **Saf** or **Liv**. The second one detects all safety orphans (**Saf**) at the cost of rejecting some correct programs and still accepting the programs producing liveness orphans (**Liv**). A third system being currently investigated tackles the problem of liveness orphans (**Liv**) providing a necessary condition for the progression of computations in a ML-ACT program, which therefore, still accepts some programs producing liveness orphans.

4. Checking for communication errors (**Err**)

4.1. Types

Types used to verify ML-ACT programs are built from the grammar of Figure 7. In order to make types and typing rules easier to write, an ordered list of types (a sequence), denoted \vec{T} , is used. Sequences can be empty, and all useful operators between types are extended to sequences of types. To obtain simpler rules, we assume that $\{\vec{x} : \vec{t}\}$ is empty if \vec{x} is empty, and that $\vec{t} \rightarrow t'$ is t' , if \vec{t} is empty. c denotes usual constant type such as *unit*, *bool*... Finally, to ease the reading of constraints we use two denotations for interface variables (i) whether they are input variables, i.e. messages sent to an actor (ψ) or capacity variables, i.e. messages accepted by an actor (χ).

Let's look more closely at name type and behavior type. In the case of name type, the interface abstracts the *set* of messages sent to the name (the input interface ψ). For behavior type, the first interface (capacity) corresponds to the reaction *set* of its current

$T ::= t$	type variable		
c	constant type	$t_1 \rightarrow t_2 \sqsubseteq t'_1 \rightarrow t'_2$	$\Leftrightarrow t'_1 \sqsubseteq t_1 \wedge t_2 \sqsubseteq t'_2$
$T \rightarrow T$	function type	$@\psi_1 \sqsubseteq @\psi_2$	$\Leftrightarrow \psi_2 \sqsubseteq \psi_1$
$@I$	name type	$(\chi_1, \psi_1) \triangleright \sqsubseteq (\chi_2, \psi_2) \triangleright$	$\Leftrightarrow \chi_1 \sqsubseteq_b \chi_2 \wedge \psi_1 \sqsubseteq \psi_2$
$(I, I) \triangleright$	behavior type	$\langle m_j(\vec{t}_j) \rangle^{j \in J} \sqsubseteq_b \langle m_k(\vec{t}'_k) \rangle^{k \in K}$	$\Leftrightarrow J \subseteq K \wedge \forall l \in J \vec{t}_l \sqsubseteq \vec{t}'_l$
$I ::= i$	interface variable	$\langle m_j(\vec{t}_j) \rangle^{j \in J} \sqsubseteq \langle m_k(\vec{t}'_k) \rangle^{k \in K}$	$\Leftrightarrow J \subseteq K \wedge \forall l \in J \vec{t}_l \sqsubseteq \vec{t}'_l$
$\langle m_j(\vec{T}_j) \rangle^{j \in J}$	interface value		

Figure 7. Types and subtyping

and all its possible future behaviors. The second interface part contains all messages that could be sent to its *ego*. Therefore a behavior type will be denoted $(\chi, \psi) \triangleright$. For example, the behavior b has type $(\langle m(@\langle r \rangle) \ p \rangle, \langle m(t) \rangle) \triangleright$, which means that an actor a with behavior b can treat p messages and $m(c)$ messages where c must know how to cope with r messages, and its *ego* receives $m(d)$ messages (where d has the type t). Notice that if we type non-closed programs (such as modules) those two kinds of types may be partial. Indeed, our types abstract the *entire* life cycle of an actor.

When typing concurrent objects and actors, a natural subtyping notion arises. Indeed, an actor a_1 can always be replaced by another one a_2 , if a_2 can receive at least the messages sent to a_1 . This subtyping notion is denoted $t_1 \sqsubseteq t_2$, which means that a value of type t_1 is able to replace a value of type t_2 . This inclusion relation on types is defined in Figure 7. The first inclusion shows the usual contravariant behavior of arguments in function types. The inclusion on names behaves as the subtyping notion using interface inclusion (\sqsubseteq). But, the inclusion on behaviors is more complex, since it uses the flattening inclusion (\sqsubseteq_b — issued from the flattening operator \cup^b that we have defined with Colaço in [COL 97b]). The flattening operator allows to merge all the behaviors of an actor in a safe way. It produces a *flat* view of an actor, composed of all reactions from all its possible behaviors, combining all the constraints on the reaction arguments.

4.2. Type system

The type inference system proceeds in two steps. First, it parses the source code to collect all type inclusion constraints and, then, solves them to give types to all *elements* (actors, behaviors, functions, ...). This section is only concerned with the collecting phase and the resolution techniques will not be discussed in this article (see [COL 99b] for their description). The global constraint set is made implicit in typing rules and all inclusions added to it are presented on the right side of the rules. Moreover, we suppose that all unbound type variables are fresh.

Following Aiken and Fähndrich in [FAH 97], the resolution mixes inclusion constraint resolution and unification. Indeed, we only need full precision for concurrent parts of programs and therefore functional parts use unification in the usual *à la ML* way. The type system could only use inclusion constraints to produce types, but at a

much higher calculation cost. Our tradeoff allows to keep reasonable calculation time and enough precision to detect communication errors and some safety orphans (the typing cost for a pure functional program is similar to the one in an ML compiler).

$$\begin{array}{c}
 \text{EXP: } \frac{\mathcal{E} \vdash_e e : t \quad \mathcal{E} \vdash p}{\mathcal{E} \vdash e ; ; p} \quad \text{BEH: } \frac{\forall j \mathcal{E} + \{\vec{x} : \vec{t}\} + \{\vec{x} \vec{p}^j : \vec{t} \vec{p}^j\} \vdash_b b_j : tr_j \quad \mathcal{E} + \{\vec{x} : \vec{t}\} \vdash p \left(\frac{\longrightarrow}{\vec{t} \vec{p} \rightarrow tr \sqsubseteq \vec{t}} \right)}{\mathcal{E} \vdash \text{behavior } x_1 \vec{x} \vec{p}^1 = b_1 \dots \text{and } x_n \vec{x} \vec{p}^n = b_n ; ; p} \\
 \\
 \text{EOF: } \frac{}{\mathcal{E} \vdash \emptyset} \quad \text{REA: } \frac{\forall j \mathcal{E} + \{\text{ego} : @\psi, \text{self} : (\chi, \psi) \triangleright\} + \{\vec{x}^j : \vec{t}^j\} \vdash_e e_j : tr_j \left(\begin{array}{l} \vec{t} \vec{r} = \text{unit} \\ \langle m_j(\vec{t}^j) \rangle^{j \in 1 \dots n} \sqsubseteq_b \chi \end{array} \right)}{\mathcal{E} \vdash_b \text{message } m_1 \vec{x}^1 = e_1 \dots \\ \text{message } m_n \vec{x}^n = e_n : (\chi, \psi) \triangleright} \\
 \\
 \text{PAR: } \frac{\mathcal{E} \vdash_e e : t}{\mathcal{E} \vdash_e (e) : t} \quad \text{CST: } \frac{}{\mathcal{E} \vdash_e C : \text{Typeof}(C)} \quad \text{SEND: } \frac{\mathcal{E} \vdash_e e : t \quad \mathcal{E} \vdash_e \vec{e} : \vec{t}}{\mathcal{E} \vdash_e \text{send } m(\vec{e}) \text{ to } e : \text{unit}} \left(t \sqsubseteq @ (m(\vec{t})) \right) \\
 \\
 \text{APP: } \frac{\mathcal{E} \vdash_e e : t \quad \mathcal{E} \vdash_e \vec{e} : \vec{t} \left(\begin{array}{l} \vec{t} \sqsubseteq \vec{t} \vec{p} \\ t \sqsubseteq \vec{t} \vec{p} \rightarrow tr \end{array} \right)}{\mathcal{E} \vdash_e e \vec{e} : tr} \quad \text{IF: } \frac{\mathcal{E} \vdash_e e_1 : t_1 \quad \mathcal{E} \vdash_e e_2 : t_2 \quad \mathcal{E} \vdash_e e_3 : t_3 \left(\begin{array}{l} t_1 = \text{bool} \\ t_2 \sqsubseteq t \\ t_3 \sqsubseteq t \end{array} \right)}{\mathcal{E} \vdash_e \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t} \\
 \\
 \text{SEQ: } \frac{\forall j \mathcal{E} \vdash_e e_j : t_j}{\mathcal{E} \vdash_e e_1 ; \dots ; e_n : t_n} \quad \text{LETREC: } \frac{\mathcal{E} + \{\vec{x} : \vec{t}; f : tf\} \vdash_e e : t \quad \mathcal{E} + \{f : \vec{t} \rightarrow t\} \vdash_e e' : t' \left(\vec{t} \rightarrow t \sqsubseteq tf \right)}{\mathcal{E} \vdash_e \text{let rec } f \vec{x} = e \text{ in } e' : t'} \\
 \\
 \text{ID: } \frac{x \in \text{dom}(\mathcal{E})}{\mathcal{E} \vdash_e x : \mathcal{E}(x)} \quad \text{SUI: } \frac{\text{ego} \in \text{dom}(\mathcal{E})}{\mathcal{E} \vdash_e \text{suicide} : \text{unit}} \quad \text{LET: } \frac{\mathcal{E} + \{\vec{x} : \vec{t}\} \vdash_e e : t \quad \mathcal{E} + \{f : \vec{t} \rightarrow t\} \vdash_e e' : t'}{\mathcal{E} \vdash_e \text{let } f \vec{x} = e \text{ in } e' : t'} \\
 \\
 \text{BEC: } \frac{\mathcal{E} \vdash_e e : t}{\mathcal{E} \vdash_e \text{become } e : \text{unit}} \left(t \sqsubseteq \mathcal{E}(\text{self}) \right) \quad \text{ACT: } \frac{\gamma(a) = (\chi_a, \psi_a) \quad \mathcal{E} \vdash_e e : t \left(\begin{array}{l} t \sqsubseteq (\chi_a, \psi_a) \triangleright \\ \diamond (\chi_a, \psi_a) \end{array} \right)}{\mathcal{E} \vdash_e \text{new}_a e : @\psi_a}
 \end{array}$$

Figure 8. Typing rules

Typing a program consists in typing all the expressions and behavior declarations, in sequence (EOF), (EXP). The behavior declaration rule (BEH) specifies that we add all behavior names and arguments to the environment before typing each mutually recursive behavior. The generated constraints are usual subtyping relations between the computed types and the corresponding type variables (from the environment). The message body e_j is typed in an environment where `ego`, `self` and all the arguments have been added, and is constrained to have the type `unit`. The type variable generated for the interface of the current behavior must contain all m_i messages.

Functional typing is classic. Subtyping is used because functional expressions can manipulate concurrent entities and therefore impose constraints on them. Whether to use inclusion constraints or unifications is decided during the resolution process.

Sending a message to a is collected in the type of a . When `suicide`, `ego` or `become` arise, the presence of `ego` in the typing environment is used to check whether we are in an actor or in a top-level expression (which leads to an error). This strategy is possible because the only rule defining `ego` is the behavior body typing rule (`ego` cannot be defined by a `let!`). Changing behavior (BEC) requires that the given behavior is a subtype of `self` type. Thus, we collect the new behavior interface and the messages potentially sent to its `ego`.

For the creation of actors (ACT), we use an environment γ constructed during the tagging phase of the `new` instructions (see section 2.2.1). Each `newa` causes the introduction of $\{a : (\chi_a, \psi_a)\}$ in γ where χ_a and ψ_a are fresh variables representing the inputs and the capacity of a . A `newa` evaluates to a fresh name a_1 (with $\gamma(a_1) = \gamma(a)$) and therefore `newa` also has the type $@\psi_a$. We also check the validity of this behavior by the constraint: $\diamond(\chi_a, \psi_a)$. This constraint is used to compare inputs and capacity on an actor address and is defined by: $\diamond(\chi, \psi) \iff \psi \subseteq \chi$. A \diamond -constraint collects the contributions of the arguments of messages (such as new behaviors) by enforcing subtyping on their types. It is also used to detect trivial orphan messages.

The resulting type system is sound, that is to say it detects all ML-ACT programs which could be reduced to `Err`. As a side effect, it also detects all trivial safety orphans.

4.3. Proof of soundness

As the complete proof is quite long with many trivial cases, we will only state the lemma and theorems and present some key cases of the demonstration.

First, we need to extend ML-ACT typing rules to take into account configurations and semantic values (Fig. 9). And then we state four lemmas required for the two soundness theorems.

$$\begin{array}{c}
\text{PAR: } \frac{\vdash_w w_1 : \wp \quad \vdash_w w_2 : \wp}{\vdash_w w_1 \parallel w_2 : \wp} \qquad \text{ACT: } \frac{\gamma(a) = (\chi_a, \psi_a) \quad \mathcal{R} : t}{\vdash_w a \triangleright \mathcal{R} : \wp} \left(t \sqsubseteq (\chi_a, \psi_a) \triangleright \right) \\
\text{EXP: } \frac{\{\} \vdash_e e : \text{unit}}{\vdash_w \llbracket e \rrbracket : \wp} \qquad \text{EXPA: } \frac{\{\} \vdash_e e : t}{\vdash_w \llbracket e \rrbracket_{a \triangleright \mathcal{R}} : \wp} \qquad \text{MESS: } \frac{\gamma(a) = (\chi_a, \psi_a) \quad \vec{v} : \vec{t}}{\vdash_w a \triangleleft m(\vec{v}) : \wp} \left(\langle m(\vec{t}) \rangle \subseteq \psi_a \right) \\
\text{CLOS: } \frac{\{\vec{x} : \vec{t}\} \vdash_e e : t}{\langle \vec{x}; e \rangle : \vec{t} \rightarrow t} \qquad \text{NAME: } \frac{\gamma(a) = (\chi_a, \psi_a)}{a : @\psi_a}
\end{array}$$

Figure 9. Typing rules, configurations and semantic values

Lemma 1 (Sub-expression typing) *If $\mathcal{E} \vdash_e C[e] : t$ then $\exists \mathcal{E}', \mathcal{E}' \vdash_e e : t'$.*

Lemma 2 (Context typing) *If $\mathcal{E}_1 \vdash_e C[e] : t_1$, $\mathcal{E}_2 \vdash_e e : t_2$, $\mathcal{E}_2 \vdash_e e' : t_3$ and $t_3 \sqsubseteq t_2$ then $\mathcal{E}_1 \vdash_e C[e'] : t_4$ and $t_4 \sqsubseteq t_1$.*

Lemma 3 (Substitution typing) *If $\mathcal{E} + \{x : t_x\} \vdash_e e : t$, $\mathcal{E} \vdash_e e' : t'$, $t' \sqsubseteq t_x$ and $\phi = [e'/x]$ then $\mathcal{E} \vdash_e \phi(e) : t_\phi$ and $t_\phi \sqsubseteq t$.*

Lemma 4 (Congruence typing) *If $\vdash_w w : \wp$ and $w \equiv w'$ then $\vdash_w w' : \wp$.*

The proof of the two first lemmas is made by induction on the context structure. The third lemma is proved by induction on the expression structure using usual substitution rules. And finally, the demonstration of the fourth lemma is detailed in [COL 97a].

Theorem 1 (Subject Reduction) *If $\vdash_w w : \wp$ and $w \rightarrow w'$ then $\vdash_w w' : \wp$.*

The proof of this theorem is made by induction on the configuration structure matchable by semantic rules. Let's give some key points:

$$\text{-- Message treatment: } \frac{\mathcal{R}(m) = \langle \vec{x}; e \rangle}{a \triangleright \mathcal{R} \parallel a \triangleleft m(\vec{v}) \rightarrow \llbracket [\vec{v}/\vec{x}]e \rrbracket_{a \triangleright \mathcal{R}}}$$

Let's suppose that we have the following deduction:

$$\frac{\text{ACT: } \frac{\text{BEH: } \frac{\dots}{\mathcal{R} : (\chi, \psi) \triangleright (3)}}{\vdash_w a \triangleright \mathcal{R} : \wp} (2) \quad \text{MESS: } \frac{\vec{v} : t\vec{v}}{\vdash_w a \triangleleft m(\vec{v}) : \wp} (1)}{\text{PAR: } \vdash_w a \triangleright \mathcal{R} \parallel a \triangleleft m(\vec{v}) : \wp}$$

with the valid constraints:

$$\begin{array}{ll} (1) & \langle m(t\vec{v}) \rangle \subseteq \psi_a \\ (2) & (\chi, \psi) \triangleright \subseteq (\chi_a, \psi_a) \triangleright \\ (3) & \langle m_j(\vec{T}_j) \rangle^{j \in J} \subseteq_b \chi_a \text{ and } \exists j_0 \in J, m_{j_0} = m \\ (4) & \diamond(\chi_a, \psi_a) \text{ because } a \in \gamma \end{array}$$

From (2+3), we can deduce $\langle m_j(\vec{T}_j) \rangle^{j \in J} \subseteq_b \chi_a$, which means $\chi_a = \langle m_k(\vec{T}_k) \rangle^{k \in K}$ with $J \subseteq K$ and $\forall j \in J, \vec{T}_j \subseteq \vec{T}_j$ (def. of \subseteq_b).

As well, from (1) we deduce that $\psi_a = @ \langle m(t\vec{v}') \dots \rangle$ and $t\vec{v} \subseteq t\vec{v}'$ (def. of \subseteq).

Finally, as a is a name, the constraint (4) provides the link between formal and real parameters of the message m : $t\vec{v}' \subseteq \vec{T}_{j_0}^i$. This leads by transitivity to $t\vec{v} \subseteq \vec{T}_{j_0}^i$.

Moreover, the behavior \mathcal{R} is well typed so $\{\vec{x} : t_{j_0}^i\} \vdash_e e : \text{unit}$. And by applying the lemma 3: $\{\} \vdash_e [\vec{v}/\vec{x}]e : t$ with $t \subseteq \text{unit}$, we finally conclude that $\vdash_w \llbracket [\vec{v}/\vec{x}]e \rrbracket_{a \triangleright \mathcal{R}} : \wp$.

$$\text{-- Actor creation: } \frac{b \text{ fresh } \quad \gamma \leftarrow \gamma :: \{b \mapsto \gamma(a)\}}{\llbracket \text{C}[\text{new}_a \mathcal{R}_1] \rrbracket_- \rightarrow b \triangleright \mathcal{R}_1 \parallel \llbracket \text{C}[b] \rrbracket_-}$$

Supposing that $\text{C}[\text{new}_a \mathcal{R}_1]$ has the type t in an empty environment and applying

$$\text{lemma 1, we get: } \frac{\gamma(a) = (\chi_a, \psi_a) \quad \mathcal{R}_1 : t_1 \quad (t_1 \subseteq (\chi_a, \psi_a) \triangleright (1))}{\mathcal{E} \vdash_e \text{new}_a \mathcal{R}_1 : @\psi_a}$$

As the name b and $\text{new}_a \mathcal{R}_1$ have the same type, we can apply the lemma 2. Which leads to $\{\} \vdash_e \text{C}[b] : t'$ with $t' \subseteq t$ and back to configuration to $\vdash_w \llbracket \text{C}[b] \rrbracket_- : \wp$. Furthermore, from constraint (1), we can deduce that $\vdash_w b \triangleright \mathcal{R}_1 : \wp$. And finally, the whole result is well typed as the parallel configuration is well typed and the restriction too.

$$\text{-- Message sending: } \frac{}{\llbracket \text{C}[\text{send } m(\vec{v}) \text{ to } a] \rrbracket_- \rightarrow \llbracket \text{C}[\langle \rangle] \rrbracket_- \parallel a \triangleleft m(\vec{v})}$$

Using the same idea as the previous case, we have: $\{\} \vdash_e C[\text{send } m(\vec{v}) \text{ to } a] : t$ and

lemma 1 gives: $\text{TSEND: } \frac{\gamma(a) = (\chi_a, \psi_a) \quad \vec{v} : \vec{t}}{\mathcal{E} \vdash_e \text{send } m(\vec{v}) \text{ to } a : \text{unit}} \left(@\psi_a \sqsubseteq @\langle m(\vec{t}) \rangle \text{ (1)} \right)$

As $\mathcal{E} \vdash_e () : \text{unit}$ and $\text{unit} \sqsubseteq \text{unit}$, applying lemma 2, we get $\{\} \vdash_e C[()] : t'$ with $t' \sqsubseteq t$, showing that $\vdash_w \llbracket C[()] \rrbracket_- : \wp$. And constraint (1) allows us to write $\vdash_w a \triangleleft m(\vec{v}) : \wp$, proving the necessary parallel composition.

Applying the previous theorem, a well typed program is evaluated to a well typed program and as **Err** is not well typed, no programs can lead to an error.

Theorem 2 (Soundness) *The evaluation of a well typed ML-ACT program cannot produce the Err semantic value.*

4.4. Examples

The typing of the cell example (Fig. 1) produces the following types. The \diamond -constraints are then checked to ensure that there are no trivial safety orphans.

$$\begin{aligned} - \text{a_cell} : @\psi_{mc} \text{ with } & \begin{cases} t_c \sqsubseteq @\langle \text{prn}(int) \rangle \\ \langle \text{init}(int) \text{ get}(t_c) \text{ set}(int) \rangle \sqsubseteq_b \chi_{mc} \\ \langle \text{init}(int) \text{ set}(int) \text{ get}(@\psi_{ms}) \rangle \sqsubseteq \psi_{mc} \\ \diamond(\chi_{mc}, \psi_{mc}) \end{cases} \\ - \text{a_scr} : @\psi_{ms} \text{ with } & \begin{cases} \langle \text{prn}(int) \rangle \sqsubseteq_b \chi_{ms} \\ \langle \text{prn}(int) \rangle \sqsubseteq \psi_{ms} \\ \diamond(\chi_{ms}, \psi_{ms}) \end{cases} \end{aligned}$$

4.5. Related work

The type system proposed in this section is mainly derived from usual type systems for sequential objects. Its main novelties are the use of subtyping instead of kind-based unification and the definition of the flattening operation \cup^b introduced to combine all the possible behaviors of an actor in a subtype-safe way. Our proposal can be related to the work of Vasconcelos and Tokoro ([VAS 93]) for concurrent objects and of Kobayashi and Yonezawa for actors ([KOB 94]). These two approaches use the same type structure and constraints (equalities resolved by using kind-based unification). We use the same type abstraction, but we use subtyping and set constraint resolution instead of kind based unification. Our subtype-based approach is strictly more expressive as will be shown by the concurrent objects' example in Figure 10.

Kind-based unification allows the extension of interfaces. But it requires that the message parameters have the same types in all definitions of a given reaction (in all the behaviors of an actor). Furthermore, all sendings of such a message must also have the same type. In this example, the message `quest` is sent twice to the same actor. The first time, its parameter has type `[rep, mess]` and the second time, it has

```

behavior b1 = message quest(c) = send reply to c
and b2 = message reply = ()
           message mess = ()
and b3 = message reply = ()
end;;
let a = new b1 in
let b = new b2 in
let c = new b3 in
  send quest(b) to a; send quest(c) to a;;

```

Figure 10. *Subtyping versus Kinds*

type [rep]. Therefore, both Vasconcelos' and Kobayashi's type systems reject this example as ill-typed.

Subtype-based constraints also allow the extension of interfaces, but their use is less restrictive on the message parameter types. The example is well-typed using our system. This advocates that kind-based unification is not powerful enough in the context of concurrent objects and actors. Thus subtyping should be used instead.

To our knowledge, subtype-based constraints had only been used in the more restrictive context of objects. The novel part of our work is the flattening relation \cup^b , which combines the types of an actor's various behaviors in a type safe way.

This first type system was initially defined in [COL 97b] for the Calculus of Primitive Actors, a process calculus merging asynchronous π -calculus and Cardelli's Calculus of Primitive Objects. In order to apply it to ML-ACT, it was initially extended without much rewriting to λ CAP (CAP merged with application and abstraction) in [COL 99b]. In this context, the interleaving between the concurrent and the functional world is very finely grained. In ML-ACT, this interleaving is much coarser, thus the application of the type system required a major rewriting and, in particular, a new soundness proof more technical and complicate.

5. Toward the detection of safety orphans (saf)

The previous type system has been implemented in the ML-ACT compiler. It detects the major part of common errors, however it is still too coarse for the detection of sophisticated orphans. Indeed, it only catch trivial safety orphans. To improve the detection, a second type abstraction for the CAP calculus has been developed (see [COL 99a]) and is currently being integrated in the compiler. This type system will not be described precisely but will be introduced with an informal and intuitive description of the type abstraction.

5.1. Type abstraction

An actor's execution path is a branch in its possible behaviors regular tree. Therefore a safe abstraction of the program consists in the intersection of the abstractions of every branches. Each branch is abstracted as in the previous system, the only difference is that we take into account the number of times a message can be treated when flattening all behaviors. In fact, we use multi-operators which are an extension of usual operators to multi-set. An interface is slightly changed by adding a multiplicity to each message label $\langle m_i^{\mu_i}; (\vec{t}_i) \rangle^{i \in I}$. *Multiplicity* is an integer or ω in the case it is unknown or infinite. When interfaces are used for capacities, it collects an upper bound of the quantity of messages that may be sent to a given target. When interfaces are used to describe inputs, *multiplicity* represents a lower bound of the number of messages (with this label) that can be handled by the behavior. To give a better insight of this approximation, let us consider a sequence of behaviors as a finite state deterministic automaton. In this automaton, the initial state is the actor's initial behavior and transitions are tagged by accepted messages without their parameters (a kind of trace semantics). For example, an actor and its associated graph are given in Figure 11.

```
behavior b1 = message m1 = become b2
              message m2 = become b4
and b2 = message m1 = become b3
           message m3 = ()
and b3 = message m4 = ()
and b4 = message m1 = ()
end;;
let a = new b1;;
```

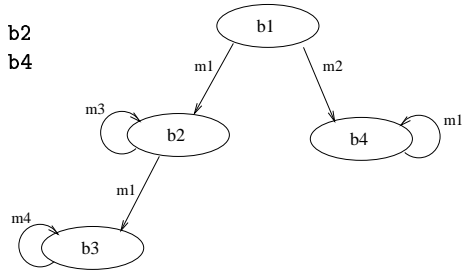


Figure 11. An example

Our aim is to abstract this input by the multi-set of transitions that are common to all paths in the automaton, including infinite ones. Each state of the automaton is considered as the initial state of a sub-automaton and is abstracted by the abstraction of this sub-automaton. Furthermore, we abstract an automaton by the set of transitions it can make, not taking into account the order in which they are fired (in fact, this information is only required for liveness orphan detection). For example, b_3 is abstracted by $\langle m_4^\omega \rangle$, b_4 by $\langle m_1^\omega \rangle$, b_2 by $\langle m_1, m_4^\omega \rangle$ and b_1 by $\langle m_1^2 \rangle$. Notice that our abstraction uses intersection, it explains the fact that the type of b_2 does not contain any m_3 . Indeed, as $\{m_1, m_4^\omega\}$ is accepted by the automaton, no m_3 can be in the type for it.

To give a sight of the types calculus for this exemple, we use \uplus and \sqcap operators that won't be defined formally (refer to [COL 97a] and [COL 98a]). \uplus adds multiplicities during the flattening of interfaces and \sqcap takes the minimal multiplicity during their intersection. The actor a of Figure 11 has type t and the computation of t follows (as messages take no parameters, we omit the $()$):

$$\left\{ \begin{array}{l} t = \langle m_1 \rangle \uplus^b t_1 \sqcap \langle m_2 \rangle \uplus^b t_2 \\ t_1 = \langle m_1 \rangle \uplus^b t_3 \sqcap \langle m_3 \rangle \uplus^b t_1 \\ t_2 = \langle m_1 \rangle \uplus^b t_2 \\ t_3 = \langle m_4 \rangle \uplus^b t_3 \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} t = \langle m_1 \rangle \uplus^b t_1 \sqcap \langle m_2 m_1^\omega \rangle \\ t_1 = \langle m_1 m_4^\omega \rangle \sqcap \langle m_3 \rangle \uplus^b t_1 \\ t_2 = \langle m_1^\omega \rangle \\ t_3 = \langle m_4^\omega \rangle \end{array} \right.$$

Therefore, m_3 is not in the left part of the intersection, this allows to calculate t_1 value: $\langle m_1 m_4^\omega \rangle \sqcap t_1 = \langle m_1 m_4^\omega \rangle$, finally giving: $t = \langle m_1^2 m_4^\omega \rangle \sqcap \langle m_1^\omega m_2 \rangle = \langle m_1^2 \rangle$. As the main principle of our analysis consists in computing for each name an input type and a capacity type and in comparing them in a *multi-set sense*. The conclusion is that any program containing the code in Figure 11 and more than two sendings of m_1 is rejected. Moreover, the typing of the cell example from Figure 1 show there are no safety orphans at all by producing:

$$\begin{array}{l} - \text{a_cell} : @\psi_{mc} \text{ with } \left\{ \begin{array}{l} t_c \sqsubseteq @(\text{prn}(int)) \\ \langle \text{init}(int) \text{ get}^\omega(t_c) \text{ set}^\omega(int) \rangle \sqsubseteq_b \chi_{mc} \\ \langle \text{init}(int) \text{ set}(int) \text{ get}^2(@\psi_{ms}) \rangle \sqsubseteq \psi_{mc} \\ \diamond(\chi_{mc}, \psi_{mc}) \end{array} \right. \\ - \text{a_scr} : @\psi_{ms} \text{ with } \left\{ \begin{array}{l} \langle \text{prn}^\omega(int) \rangle \sqsubseteq_b \chi_{ms} \\ \langle \text{prn}^2(int) \rangle \sqsubseteq \psi_{ms} \\ \diamond(\chi_{ms}, \psi_{ms}) \end{array} \right. \end{array}$$

This type system was shown to be sound for the Calculus of Primitive Actors (pure actor ML-ACT programs) by Colaço in [COL 97a]. We are currently extending it to ML-ACT following the work of Hughes et al. ([HUG 96]) to build a safe upper approximation of the number of time a function is executed depending on its parameters.

As it must eliminate all programs where a safety orphan may occur, our type system is strict. This strictness enforces the following programming discipline, every branch starting from a given behavior must contain at least all the messages which are sent to the actor when it assumes this behavior. Therefore, a branch should be an interleaving of the same message sets. However, this restriction can be relieved using a more sophisticated extension of Aiken's and Wimmers' resolution algorithm based on conditional types ([AIK 94]) that we used in a previous work on sequential objects ([PAN 94]). This extension has a much higher computational cost and will only be integrated in our works on Erlang ([ARM 96]) because it requires more precise type.

5.2. Related work

Many different studies are currently related to the static analysis of non-uniform service availability.

Following lines proposed by Nierstrasz [NIE 95], Puntigam ([PUN 96]), Ravara and Vasconcelos ([RAV 97, RAV 99]), and Najm and Nimour [NAJ 97, NAJ 99], propose the use of more sophisticated type abstractions (in fact, the structure of their types is a process calculus). Their work is very promising as their abstractions preserve more causal relations. However, their type systems rely on explicit typing and the feasibility

of type inference is still a conjecture. Given the complexities of the type abstraction used for concurrent objects and actors, we advocate that users should not be required to write so complex types (frequently more complex than the program itself), and that type inference should be used. In this purpose, a joint work between our team and Ravara has been initiated using a more sophisticated constraint resolution algorithm than the one evocated in this paper.

The use of multiplicities in the type abstraction of message sending was derived from the effect system proposed by Kobayashi et al. [KOB 95]. The main difference is that they must consider finite sets of possible values determined by an integer M ($0, \dots, M - 1, \omega$) which represents the accuracy of the analysis in order to have a terminating algorithm. Our algorithm terminates without choosing a maximal value.

6. Conclusion and perspectives

We have given in this paper a formal description of a sound type system for a realistic actor based language which features non-uniform behaviors. The formal definition of various kinds of errors such as functional, communication, safety and liveness orphans have been introduced. This type system detects all functional and communication errors and trivial safety orphans. We have evocated an extension of this type system which detects all safety orphans and is currently being integrated in our compiler.

The remaining task is to introduce some flexibility in the second type system. It is clear that the cost of this flexibility will be that some incorrect programs will be well-typed. One way being studied is to reject programs that always lead to orphan messages and to warn the programmer when orphan messages may arise. To get this behavior, the approach will combine both systems presented in this article.

The second important future goal is the liveness orphan detection. To reach it, we will follow two different tracks. First, we will use the precise types produced by the second type system in order to give a necessary condition of liveness orphan freedom. Then, some causality information will be introduced in types. Indeed, following the work of Kobayashi in [KOB 97], actor creation and message sending could be decorated with a time-stamp and ordered by the imbrication of term structure. Potential deadlocks then correspond to cyclic time-stamp chains.

Apart from improving type abstraction and the development of ML-ACT, we are currently investigating the application of our analyzes to the Erlang language. This application should give us access to a huge quantity of industrial concurrent and distributed code we could confront with our type systems.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments and Antonio Ravara and the other members of the team for the proofreading of this paper.

7. References

- [AGH 86] AGHA G., *Actors: A model of concurrent computation in distributed systems*, MIT Press, Cambridge, Mass., 1986.
- [AIK 94] AIKEN A., WIMMERS E., LAKSHMAN T. K., “Soft typing with conditional types”, *Proc. of PoPL*, Jan. 1994.
- [ARM 96] ARMSTRONG J., VIRDING R., WIKSTRÖM C., WILLIAMS M., *Concurrent programming in ERLANG*, Prentice Hall, Hemel Hempstead, 1996, see also www.erlang.org.
- [BOU 97] BOUDOL G., “The pi-calculus in direct style”, *Proc. of PoPL*, Jan. 1997.
- [COL 96] COLAÇO J.-L., PANTEL M., SALLÉ P., “CAP: An Actor dedicated process calculus”, *Prof. of Proof Theory of Concurrent Object-Oriented Programming*, May 1996.
- [COL 97a] COLAÇO J.-L., “Analyses statiques par typage de langages d’Acteurs”, Thèse de doctorat, INPT, Oct. 1997.
- [COL 97b] COLAÇO J.-L., PANTEL M., SALLÉ P., “A set-constraint-based analysis of Actors”, *Proc. of FMOODS*, Jul. 1997.
- [COL 98a] COLAÇO J.-L., PANTEL M., SALLÉ P., “From set based to multiset based analysis: a practical approach”, *Set constraints and Constraint based program analysis*, Oct. 1998.
- [COL 98b] COLIN M., “Intégration des Typages Fonctionnel et Concurrent d’un Langage Fonctionnel d’Acteurs”, DEA IFP, INPT, Sep. 1998.
- [COL 99a] COLAÇO J.-L., PANTEL M., DAGNAT F., SALLÉ P., “Static safety analysis for non-uniform service availability in Actors”, *Proc. of FMOODS*, Feb. 1999.
- [COL 99b] COLIN M., PANTEL M., DAGNAT F., SALLÉ P., “Intégration des Typages Fonctionnel et Concurrent d’un Langage Fonctionnel d’Acteurs”, *Actes des Journées Franco-phones des Langages Applicatifs*, Feb. 1999.
- [DAG 97] DAGNAT F., “Conception d’un Langage Fonctionnel d’Acteur et Réalisation de son Compilateur”, DEA IFP, INPT, Sep. 1997.
- [DAL 97] DAL-ZILIO S., “Implicit Polymorphic Type System for the Blue Calculus”, report num. 3244, Sep. 1997, INRIA.
- [FAH 97] FÄHNDRICH M., AIKEN A., “Program Analysis Using Mixed Term and Set Constraints”, *Proc. of the Static Analysis Symposium*, Sep. 1997.
- [FOU 96] FOURNET C., GONTHIER G., LÉVY J.-J., MARANGET L., RÉMY D., “A calculus of mobile agents”, *Proc. of the Int. Conf. on Concurrency Theory*, 1996.
- [FOU 97] FOURNET C., LANEVE C., MARANGET L., RÉMY D., “Implicit typing à la ML for the join-calculus”, *Proc. of CONCUR*, LNCS 1283, Springer-Verlag, 1997, p. 196-212.
- [HEW 73] HEWITT C., BISHOP P., STEIGER R., “A Universal Modular ACTOR Formalism for Artificial Intelligence”, *Proc. of Int. Joint Conference on Artificial Intelligence*, 1973.
- [HUG 96] HUGHES J., PARETO L., SABRY A., “Proving the Correctness of Reactive Systems using Sized Types”, *Proc. of PoPL*, vol. 23, ACM, 1996.
- [KOB 94] KOBAYASHI N., YONEZAWA A., “Type-Theoretic Foundations for Concurrent Object-Oriented Programming”, *Proc. of OOPSLA*, 1994, p. 31-45.
- [KOB 95] KOBAYASHI N., NAKADE M., YONEZAWA A., “Static Analysis of Communication for Asynchronous Concurrent Programming Languages”, *Proc. of Static Analysis Symposium*, LNCS 983, Springer-Verlag, 1995, p. 225-242.

- [KOB 97] KOBAYASHI N., "A partially deadlock-free typed process calculus", *Proc. of the conf. Logic In Computer Science*, 1997.
- [MIL 91] MILNER R., "The Polyadic π -Calculus: a Tutorial", report num. ECS-LFCS-91-180, 1991, LFCS.
- [NAJ 97] NAJM E., NIMOUR A., "A Calculus of Object Binding", *Proc. of FMOODS*, Chapman & Hall, Jul. 1997, p. 5-20.
- [NAJ 99] NAJM E., NIMOUR A., STEFANI J.-B., "Infinite types for distributed object interfaces", *Proc. of FMOODS*, Feb. 1999.
- [NIE 93] NIELSON F., NIELSON H. R., "From CML to process algebras", *Proc. of the Int. Conf. on Concurrency Theory*, LNCS 715, 1993, p. 493-508.
- [NIE 95] NIERSTRASZ O., "Regular Types for Active Objects", *In Object-Oriented Software Composition, ACM SIGPLAN Notices*, Prentice Hall, Oct. 1995, p. 99-121.
- [NIE 96] NIELSON H. R., NIELSON F., AMTOFT T., "Polymorphic Subtyping for Effect Analysis: the Integration, the Semantics, the Algorithm", report num. PB-501, PB-502, PB-503, April 1996, DAIMI, University of Aarhus, Denmark.
- [NIE 97] NIELSON H. R., NIELSON F., "*Communication Analysis for Concurrent ML*", Chapitre 7, p. 185-235, Monographs in Computer Science, Springer-Verlag, 1997.
- [PAN 94] PANTEL M., SALLÉ P., "Typage souple pour le langage FOL", *Actes des Journées Francophones des Langages Applicatifs*, Jan. 1994.
- [PUN 96] PUNTIGAM F., "Type for Active Objects based on Trace Semantics", *Proc. of FMOODS*, March 1996, p. 5-20.
- [RAV 97] RAVARA A., VASCONCELOS V. T., "Behavioural Types for a Calculus of Concurrent Objects", *Proc. of EuroPar'97*, LNCS 1300, Springer-Verlag, 1997.
- [RAV 99] RAVARA A., VASCONCELOS V., "Typing non-uniform concurrent objects", report num. 1049-001, 1999, Sect. Computer Science, Dept. Mathematics, Inst. Sup. Técnico.
- [SAN 97] SANGIORGI D., "The name discipline of uniform receptiveness", *Proc. of ICALP*, LNCS 221, Springer-Verlag, 1997.
- [VAS 93] VASCONCELOS V. T., TOKORO M., "A typing system for a calculus of objects", *Proc. of ISOTAS*, LNCS 742, Springer-Verlag, Nov. 1993, p. 460-474.

Fabien Dagnat est doctorant au sein de l'équipe Véstale. Il poursuit des recherches dans le domaine des systèmes de type pour les langages concurrents, et s'intéresse plus particulièrement à l'adaptation des techniques développées par l'équipe au langage Erlang.

Marc Pantel est docteur en Informatique de l'INPT en 1994, maître de conférences en Informatique à l'ENSEEIHHT depuis 1994. Il poursuit des recherches dans le domaine de l'analyse statique de programmes.

Matthias Colin est doctorant au sein de l'équipe Véstale. Il poursuit des recherches dans le domaine des systèmes de type pour les langages concurrents.

Patrick Sallé est docteur en Informatique, docteur d'état en Informatique de l'INPT en 1980, professeur en Informatique à l'ENSEEIHHT depuis 1981. Il poursuit des recherches dans le domaine de la programmation par Acteurs et de l'analyse statique de programmes.