arXiv:1206.1687v1 [cs.PL] 8 Jun 2012

Behavioural Types for Actor Systems

Silvia Crafa

Dipartimento di Matematica - Università di Padova

Abstract. Recent mainstream programming languages such as Erlang or Scala have renewed the interest on the Actor model of concurrency. However, the literature on the static analysis of actor systems is still lacking of mature formal methods. In this paper we present a minimal actor calculus that takes as primitive the basic constructs of Scala's Actors API. More precisely, actors can send asynchronous messages, process received messages according to a pattern matching mechanism, and dynamically create new actors, whose scope can be extruded by passing actor names as message parameters. Drawing inspiration from the linear types and session type theories developed for process calculi, we put forward a behavioural type system that addresses the key issues of an actor calculus. We then study a safety property dealing with the determinism of finite actor communication. More precisely, we show that well typed and *balanced* actor systems are (*i*) deadlock-free and (*ii*) any message will eventually be handled by the target actor, and dually no actor will indefinitely wait for an expected message.

1 Introduction

Recent mainstream programming languages such as Erlang or Scala have renewed the interest on the Actor model ([10,3]) of concurrent and distributed systems. In the Actor model a program is an ensemble of autonomous computing entities communicating through asynchronous message passing. Compared to shared-state concurrent processes, the Actor model more easily avoids concurrency hazards such as data races and deadlocks, possibly at the cost of augmenting the communication overhead. On the other hand, compared to the channel-based communication of process calculi such as the π-calculus or the Join-calculus, the actor abstraction better fits the object oriented paradigm found in mainstream programming languages.

Actors can send asynchronous messages, process received messages according to a pattern matching mechanism and dynamically create new actors, whose scope can be extruded by passing actor names as message parameters. The Actor model and the asynchronous process calculi then share similarities such as (bound) name passing, but they have also many differences: actors have an identity (a name), they are single threaded and they communicate by sending messages to the mailbox of other actors rather than using channels. Despite their similarities, while a rich literature on type-based formal methods has been developed for the static analysis of process calculi, few works deal with the Actor model (see Section 4 for a discussion of the related work). In this paper we study a minimal actor calculus, AC, with the aim of bringing in the context of Actors the successful techniques developed for process calculi.

More precisely, drawing inspiration from the linear types and session type theories developed for process calculi ([16,11,13,8]), we put forward a type system that addresses the key issues of an actor calculus. Programming an actor system entails the design of a communication protocol that involves a (dynamic) set of actors; we then study a behavioural type system for AC where actor types encode the intended communication protocol, and the type checking phase statically guarantees that runtime computation correctly implements that protocol. Moreover, we study a safety property dealing with the determinism of actor communication, so that in well typed and *balanced* actor systems any message will eventually be processed by the target actor, and viceversa, no actor will indefinitely wait for an expected message. Dealing only with finite computation, we devise a simple technique to let types also prevent deadlocks.

Even if AC only considers actors with finite computation, which is clearly a strong limitation, proving that a finite system complies with the intended communication protocol is not trivial, since nondeterminism, fresh actor name passing and the asynchronous semantics of the underlying model complicate the picture. As we said, our behavioural type system is reminiscent of the type discipline of linear and session types. However, even if the actor calculus shares with session types the idea of conceiving the computation as the implementation of a specified communication protocol, there are a number of key differences between the two models (see Section 4). In [14] session types are added to a Erlang-style core actor calculus. However, in that paper session types appears an orthogonal feature of the actor language, while we aim at showing in this paper that the reasoning underlying session types is in some sense inherent to the Actor model. As a general comment that can guide the reader through the technical part of the paper, one can think that session types describe the flow of communications withing a single conversation session. Instead, an actor's behavioural type takes the point of view of an entity that might concurrently participate to different (interleaved) conversations with different parties.

2 The Actor Calculus

We assume a countable set of actor names and a countable set of variables, ranged over by a,b,c and x,y respectively. Identifiers, denoted with u, rage over names and variables. We reserve the letter m to range over a distinct set of message labels. The syntax of the Actor calculus AC comes from the basic constructs of Scala's Actors API [15,9]:

$$\textit{Expressions}\;\; e ::= \mathtt{0} \mid u\,!\,m(\tilde{u});e \mid \mathsf{react}\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I} \mid \mathsf{val}\,a = \mathsf{actor}\{e\};e$$

The expression $u_1\,!\,m(\tilde{u}_2);e$ sends to the actor u_1 the message m with the tuple of actual parameters $\tilde{u}_2$, and then continues as e. According to the Actor model, sending a message is an asynchronous action that just adds the message $m(\tilde{u}_2)$ into the mailbox associated to the actor u_1. Message handling is carried over by the react expression, that suspends the execution of the actor until it receives a message $m_j \in \{m_i\}_{i\in I}$. When a matching message is found in the actor mailbox, the execution is resumed and the corresponding continuation is activated.

New actors are dynamically created with expression $\mathsf{val}\,a = \mathsf{actor}\{e_1\};e_2$, that corresponds to the inline Scala primitive for actor creation. It defines and starts a new actor with name a and body e_1 and then continues as e_2. The actor definition introduces a new, bound, name a whose scope is both the new actor's code e_1 and the continuation

e_2. In order to have a uniform semantics, we assume that a *program* is a top level sequence of actor definitions, while input and output expressions can only occur inside an actor body. This is not restrictive since it would be sufficient to assume an implicit main actor containing the top level sequence of expressions. Anyway, observe that besides the top level definitions, new actors can still be dynamically spawned by other actors anytime during the computation.

The execution of a program spawns a bunch of concurrent actors that interact by message passing. Therefore programs are represented runtime by configurations:

$$\textit{Configurations}\;\; F ::= \mathbb{0} \mid [a \mapsto M]\, a\{e\} \mid F|F \mid (\nu a)F \mid e$$

$(\nu a)F$ is a configuration where a is a private actor name. While each actor is single threaded, a configuration is a parallel composition of a number of active actors and an expression e containing the residual sequence of top level actor definitions. An *active actor* a is represented runtime by $[a \mapsto M]\, a\{e\}$, where e is the residual body of the actor and $[a \mapsto M]$ is its associated mailbox. Mailboxes are lists of received messages of the form $[a \mapsto m_1(\tilde{b}_1) \cdot \ldots \cdot m_k(\tilde{b}_k)]$. Message parameters are values (i.e. actor names) since, according to Scala semantics, message parameters are called by value as they are implemented as parameters of an Actor object's method invocation ([15]).

Definition 1 (Free Names and Well Formed Configurations). *In input expressions formal parameters are bound variables, and actor definitions act as name binders. We work with well formed configurations where any bound name and variable is assumed to be distinct (Barendregt's convention), and where in any input branching* $\mathsf{react}\{m_i(x_i) \Rightarrow e_i\}_{i \in I}$ *the labels m_i are pairwise distinct.*

The operational semantics is given in Figure 1. Most of the rules come directly from the π-calculus. The rule (ENDED) states that a terminated actor a with no pending message in its mailbox can be garbage collected. The rules (TOP SPAWN) and (SPAWN) are used to spawn a new actor respectively from the top level main thread and from another actor. In both cases the new actor is activated by extending the configuration with a new empty mailbox and an additional thread running the body of the new actor. The rules (SEND) and (RECEIVE) implement the Actor communication model: an output expression adds a message to the mailbox of the target actor, while an input expression scans the mailbox for a matching message. Notice that the mailbox is not handled as an ordered queue of messages, hence for instance, the configuration (where we omit message parameters)

$$\begin{aligned}[b \mapsto \varnothing]\; a\{\, b\,!\,m_1; b\,!\,m_2; \mathbb{0}\} \mid \mid [b \mapsto \varnothing]\; b\{\, \mathsf{react}\; \{&m_1 \Rightarrow \mathsf{react}\{m_2 \Rightarrow e\},\\ &m_2 \Rightarrow \mathsf{react}\{m_1 \Rightarrow e'\}\}\}\end{aligned}$$

nondeterministically reduces either to $[b \mapsto \varnothing]b\{e\}$ or to $[b \mapsto \varnothing]b\{e'\}$. In other words, besides being asynchronous, in the Actor model the ordering of outputs is not guaranteed to be mirrored by the ordering of input handlers, which is instead the case of, e.g., asynchronous session types with buffered channels [8,6].

Example 2. The following program defines two actors that meet in a three-way handshake. The actor b starts by sending a *ping* message to a, then waits for a *pong* message

$$\frac{F_1 \longrightarrow F_1'}{F_1 \mid F_2 \longrightarrow F_1' \mid F_2}\;(\textsc{Par}) \qquad \frac{F \longrightarrow F'}{(\nu a)F \longrightarrow (\nu a)F'}\;(\textsc{Res}) \qquad \frac{}{[a \mapsto \varnothing]\, a\{\mathbb{0}\} \longrightarrow \mathbb{0}}\;(\textsc{Ended})$$

$$\frac{F \equiv F' \quad \longrightarrow F'' \quad \equiv F'''}{F \longrightarrow F'''}\;(\textsc{Struct}) \qquad \begin{array}{c} (\nu a)(\nu b)F \equiv (\nu b)(\nu a)F \\ (\nu a)(F \mid F') \equiv F \mid (\nu a)F' \quad a \notin \mathrm{fn}(F) \\ F \mid \mathbb{0} \equiv F \quad F \mid F' \equiv F' \mid F \\ (F_1 \mid F_2) \mid F_3 \equiv F_1 \mid (F_2 \mid F_3) \quad (\nu a)\mathbb{0} \equiv \mathbb{0} \end{array}$$

$$\frac{}{\mathsf{val}\, a = \mathsf{actor}\{e\}; e' \longrightarrow (\nu a)([a \mapsto \varnothing]\, a\{e\} \mid e')}\;(\textsc{top spawn})$$

$$\frac{}{[b \mapsto M]\, b\{\mathsf{val}\, a = \mathsf{actor}\{e\}; e'\} \longrightarrow (\nu a)([b \mapsto M]\, b\{e'\} \mid [a \mapsto \varnothing]\, a\{e\})}\;a \notin \mathrm{fn}(M)\;(\textsc{spawn})$$

$$\frac{}{[a \mapsto M]\, a\{e\} \mid [b \mapsto M']\, b\{a\,!\,m(\tilde{c}); e'\} \longrightarrow [a \mapsto M \cdot m(\tilde{c})]\, a\{e\} \mid [b \mapsto M']\, b\{e'\}}\;(\textsc{Send})$$

$$\frac{}{[a \mapsto M{\cdot}m_j(\tilde{c}){\cdot}M']\; a\{\mathsf{react}\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I}\} \longrightarrow [a \mapsto M{\cdot}M']\; a\{e_j\{^{\tilde{c}}/_{\tilde{x}_j}\}\}}\;j \in I\;(\textsc{Receive})$$

Fig. 1. Operational semantics

that carries the name of the actor to which it sends the final *pang* message. The actor *a* performs the dual sequence of actions.

$$\begin{aligned} Pr = \mathsf{val}\, a = \mathsf{actor}\{\mathsf{react}\{ping(x) \Rightarrow x\,!\,pong(a); \mathsf{react}\{pang() \Rightarrow \mathbb{0}\}\}\}\;; \\ \mathsf{val}\, b = \mathsf{actor}\{a\,!\,ping(b); \mathsf{react}\{pong(y) \Rightarrow y\,!\,pang(); \mathbb{0}\}\}\;;\; \mathbb{0} \end{aligned}$$

Now consider the case where the actor *Alice* starts two sessions of this protocol to interact both with *Bob* and *Carl* (Figure 2). In order to prevent interferences between the two sessions, a couple of private sub-actors are established for each protocol session. This is similar to private sessions in the π-calculus.

$$\begin{aligned} &Alice\;\{\;\mathsf{val}\, ab = \mathsf{actor}\{\mathsf{react}\{dest(y) \Rightarrow P(y)\}\}\,; Bob\,!\,new(ab)\,; \\ &\qquad \mathsf{val}\, ac = \mathsf{actor}\{\mathsf{react}\{dest(y) \Rightarrow P(y)\}\}\,; Carl\,!\,new(ac)\,; \mathbb{0}\;\}\;\mid \\ &Bob\;\{\;\mathsf{react}\{new(z) \Rightarrow \mathsf{val}\, ba = \mathsf{actor}\{Q(z)\}; z\,!\,dest(ba); \mathbb{0}\}\;\}\;\mid \\ &Carl\;\{\;\mathsf{react}\{new(z) \Rightarrow \mathsf{val}\, ca = \mathsf{actor}\{Q(z)\}; z\,!\,dest(ca); \mathbb{0}\}\;\} \end{aligned}$$

where $P(y) = y\,!\,ping.\mathsf{react}\{pong \Rightarrow y\,!\,pang; \mathbb{0}\}$ and
$Q(z) = \mathsf{react}\{ping \Rightarrow z\,!\,pong.\mathsf{react}\{pang \Rightarrow \mathbb{0}\}\}$.

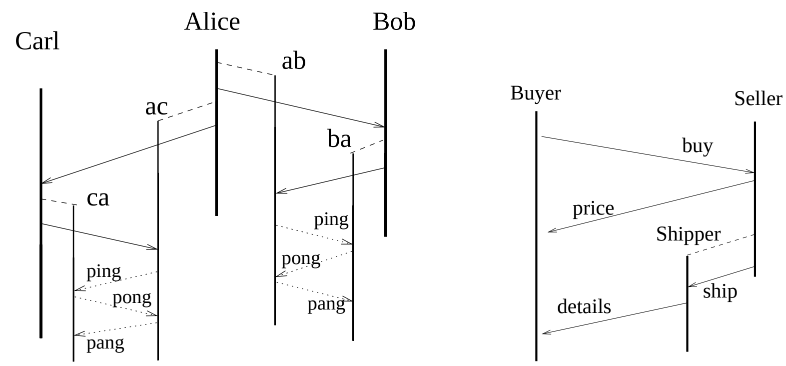

Fig. 2. Examples

Example 3. We can rephrase in the actor calculus a simple example of multiparty communication protocol that captures the interactions in a purchase system (Figure 2):

$$\begin{aligned}
&\mathit{Buyer}\{\,\mathit{Seller}\,!\,\mathit{buy}(\mathit{Buyer},\mathit{item});\,\mathsf{react}\{\mathit{price}(z) \Rightarrow \mathsf{react}\{\mathit{details}(w) \Rightarrow \ldots\}\}\,\}\ \mid\\
&\mathit{Seller}\{\,\mathsf{react}\{\,\mathit{buy}(x,y) \Rightarrow x\,!\,\mathit{price}(f(y));\\
&\qquad\qquad\qquad\qquad \mathsf{val}\,\mathit{Shipper} = \mathsf{actor}\{\mathsf{react}\{\mathit{ship}(x,y) \Rightarrow x\,!\,\mathit{details}(f'(y));\ldots\}\};\\
&\qquad\qquad\qquad\qquad \mathit{Shipper}\,!\,\mathit{ship}(x,y);\ldots\};
\end{aligned}$$

A *Buyer* actor sends to the *Seller* actor its name together with the item he wants to buy, and waits for the price and the shipping details. Dually, the *Seller* handles the *buy* message by sending to the *Buyer* the price $f(\mathit{item})$ of the selected item and spawns a new *Shipper* actor that directly interacts with the *Buyer* to finalize the shipping. Observe that the *Buyer* actor needs not to be aware that he is actually interacting not only with the *Seller* but also with a (restricted) *Shipper*. This is a further difference with the case of multiparty session types, where each interacting party is identified by its endpoint of the session channel.

3 The Type System

We assign behavioural types to actor names so that a type describes the sequence of inputs and outputs performed by the actor body. Moreover, inputs are handled as linear resources, so that to guarantee that for each expected input there is exactly one matching output. We use the following syntax for the types associated to actor names, where $\mathsf{NoMark}(S)$ means that S does not contain any marking:

$$\begin{aligned}
\textit{Types}\ \ T ::= [S] \qquad S ::= &\ \mathsf{end} \mid\ !m(\tilde{T}).S \mid \&_{i\in I}\{?m_i(\tilde{T}_i).S_i\}\\
&\mid \&^{\bullet}_{i\in I}\{{}^{\bullet}?m(\tilde{T}).S,\ ?m_i(\tilde{T}_i).S_i\}\ \text{ with } \mathsf{NoMark}(S_i)\ \forall i \in I
\end{aligned}$$

Types T are finite sequences of input and output actions. Output action $!m(\tilde{T})$ is the type of an output expression that sends the message m with a tuple of parameters of type

$\tilde{T}$. Dually, the input action $\&_{i\in I}\{?m_i(\tilde{T}_i).S_i\}$ offers the choice of receiving one of the messages m_i and continuing with the sequence S_i. Differently from session types, we do not consider output choices. Indeed our aim is not to provide an expressive calculus for protocol specification, but to put forward a type based technique to statically verify the protocol conformance of actors. On the other hand, it would not be difficult to extend the type system with output selection $\oplus_{i\in I}\{\,!\,m_i(\tilde{T}_i).S_i\}$ along the lines of input branches.

The type system then makes use of linear type assumptions to guarantee that each input is eventually matched by exactly one output in the system. Linear type assumptions are handled by means of markings. The marked action $\&^{\bullet}_{i\in I}\{{}^{\bullet}?m(\tilde{T}).S,\ ?m_i(\tilde{T}_i).S_i\}$ pinpoints an input that is "consumed" by one output expression. To illustrate, the actor $a\{b\,!\,m(c)\}$ is well typed assuming $a{:}[\,!\,m(T).S_a]$, $b{:}[{}^{\bullet}?m(T).S_b]$ since a consumes b's input. On the other hand the actor $b\{\mathsf{react}\{m(x)\Rightarrow e\}\}$ is well typed assuming the non marked type $b : [?m(T).S_b]$, since b offers an input without consuming it. Moreover, to deal with branching inputs we have to ensure that all the messages eventually received by an actor belong to the same branch of computation. For instance, consider the actor $a : [\&\{?m_1.?m_2,\ ?m_3.?m_4\}]$ (where we omit message parameters), then the actor $b\{a\,!\,m_1; a\,!\,m_4\}$ is incorrect since it sends to a two messages belonging to alternative execution paths. Indeed, the typing of b would require for a the type assumption $a : [\&^{\bullet}\{{}^{\bullet}?m_1.?m_2,\ ?m_3.{}^{\bullet}?m_4\}]$, which is prohibited by our syntax of types since it contains two markings in two different branches.

Another key point is the parallel composition of type assumptions, that must be defined so that to ensure the linear usage of marked inputs. To illustrate, given the type of the actor a above, the parallel composition $b\{a\,!\,m_1\} \mid c\{a\,!\,m_3\}$ must be prohibited since only one of the two messages will be handled by a while the other one will stay pending in a's mailbox. In other terms, the two outputs compete for the same "input resource". Notice that the typing of b, resp. c, would require the assumption $a : T^1_a = [\&^{\bullet}\{{}^{\bullet}?m_1.?m_2,\ ?m_3.?m_4\}]$, resp. $a : T^2_a = [\&^{\bullet}\{?m_1.?m_2,\ {}^{\bullet}?m_3.?m_4\}]$. The fact that the same input choice is marked both in T^1_a and T^2_a indicates that b and c consume the same input choice, hence they cannot be composed in parallel.

More formally, we define a *merge-mark* function that is used to linearly compose type assumptions. More precisely, parallel threads must assume the same type assumptions but with disjoint markings.

Definition 4 (Merge-Mark). *Let S, S' be two sequences that are equal but for the markings. Then $[S] \uplus [S'] = [S \uplus S']$ where the partial function $\uplus$ is defined by*

$$\begin{aligned}
&\mathsf{end} \uplus \mathsf{end} = \mathsf{end} \qquad\qquad !\,m(\tilde{T}).S \uplus !\,m(\tilde{T}).S' = !\,m(\tilde{T}).(S \uplus S')\\
&\&_{i\in I}\{?m_i(\tilde{T}_i).S_i\} \uplus \&_{i\in I}\{?m_i(\tilde{T}_i).S'_i\} = \&_{i\in I}\{?m_i(\tilde{T}_i).(S_i \uplus S'_i)\}\\
&\&_{i\in I}\{?m(\tilde{T}).S, ?m_i(\tilde{T}_i).S_i\} \uplus \&^{\bullet}_{i\in I}\{{}^{\bullet}?m(\tilde{T}).S', ?m_i(\tilde{T}_i).S_i\} =\\
&\qquad\qquad = \&^{\bullet}_{i\in I}\{{}^{\bullet}?m(\tilde{T}).(S \uplus S'), ?m_i(\tilde{T}_i).S_i\}
\end{aligned}$$

The main clause is the last one, together with the symmetric one that we omit. A marked input can only be merged with a corresponding non marked input, and merging recursively applies only to the marked branch. In this way we ensure that the same input choice is consumed by exactly one output, and that further outputs only consume inputs belonging to the same branch of computation.

A type environment Γ is a partial function assigning types to names and variables. We use for Γ the list notation. Let be Γ_1, Γ_2 two type environments such that $Dom(\Gamma_1) = Dom(\Gamma_2)$, then we denote by $\Gamma_1 \uplus \Gamma_2$ the type environment obtained by merging the markings contained in the two environments, i.e., $\Gamma_1 \uplus \Gamma_2 = \{u : \Gamma_1(u) \uplus \Gamma_2(u) \mid u \in Dom(\Gamma_1) = Dom(\Gamma_2)\}$. We use the notation $\Gamma; a : T$ for environment update, that is $\Gamma \setminus \{a : \Gamma(a)\} \cup \{a : T\}$.

So far so good, however this is not enough since the scope extrusion mechanism obtained by passing fresh actor names as message parameters raises additional issues. Consider the actor $a\{\mathsf{react}\{foo(x) \Rightarrow x\,!\,m(a)\}\}$, a consumes the m input offered by some actor which will be dynamically substituted for the bound variable x. In order to statically collect the resources consumed by a, the typing of a must assume for x the marked type $[{}^{\bullet}?m(T).\mathsf{end}]$. A similar situation applies when new actors are spawned. For instance, consider the previous actor a in parallel with $c\{\mathsf{val}\, b = \mathsf{actor}\{\mathsf{react}\{m(y) \Rightarrow e\}\}; a\,!\,foo(b)\}$. The parameter x of the foo message is substituted with the fresh actor name b. Hence in order to check that every input in the system is consumed, besides the type of x, we must record the type of the fresh actor $b : [?m(T).\mathsf{end}]$ and devise a way of matching the input “consumed” in x with that “offered” by b.

We then rely on type judgements of the form $\Gamma \vdash F \triangleright \Delta$, where Γ collects the type assumptions about free names and variables of F, while Δ collects type assumptions on bound names and bound variables of F. Observe that working under Barendregt's convention on bound names/variables, we avoid name conflicts. We call Δ the *escape environment*, and we let it preserve the branching structure of the computation where alternative continuations can be activated when an input choice is resolved.

Definition 5 (Escape Envirnoment). *The escape environment Δ is a choice between alternative type environments defined by $\Delta ::= \&_{i\in I}\Gamma_i \mid \&_{i\in I}\Delta_i$, where $Dom(\Gamma_i) \cap Dom(\Gamma_j) = \varnothing$ and $Dom(\Delta_i) \cap Dom(\Delta_j) = \varnothing$ for $i, j \in I$.*

We use the following notation for escape environment extension: $(\&_{i\in I}\Delta_i), u : T \stackrel{\triangle}{=} \&_{i\in I}(\Delta_i, u{:}T)$, and $(\&_{j\in J}\Delta_j), (\&_{i\in I}\Delta_i) \stackrel{\triangle}{=} \&_{j\in J}\&_{i\in I}(\Delta_j, \Delta_i)$.

3.1 Typing Rules

The main judgement of the type system is $\Gamma \vdash F \triangleright \Delta$. It means that the actors in F execute the sequence of actions described by their type and the marked input actions in Γ and Δ are *exactly* those that are consumed by the actors in F. Moreover, $Dom(\Gamma) \cap Dom(\Delta) = \varnothing$ and $\mathrm{fn}(F) \subseteq Dom(\Gamma)$ while $\mathrm{bn}(F) \subseteq Dom(\Delta)$. We also use additional judgements: $\Gamma \vdash \diamond$ states that Γ is well formed (according to standard rules given in Figure 3), $\Gamma \vdash a : T$ states that the actor a has type T in Γ, and $\Gamma \vdash [a \mapsto M]$ states that the mailbox only contains messages that are well typed according to the type that Γ assigns to a. Finally, $\Gamma \vdash_a e \triangleright \Delta$ states that e is well typed as the body of the actor a.

Type rules for actors. The rule (TYPE SPAWN) applies when the actor a spawns a new actor b. The type assumptions are split between those used by the continuation of a's body e_2, and those used by the body of the new actor e_1. The same holds for escape

$$\frac{\Gamma, u:T \vdash \diamond}{\Gamma, u:T \vdash u:T}\ (\textsc{Type Identif}) \qquad \frac{}{\varnothing \vdash \diamond}\ (\textsc{Ctx Empty}) \qquad \frac{\Gamma \vdash \diamond \quad u \notin Dom(\Gamma)}{\Gamma, u:T \vdash \diamond}\ (\textsc{Ctx Identif})$$

$$\frac{\Gamma_1, b:[S_1] \vdash_b e_1 \triangleright \Delta_1 \quad \Gamma_2, b:[S_2] \vdash_a e_2 \triangleright \Delta_2 \quad b \notin Dom(\Delta_1, \Delta_2)}{\Gamma_1 \uplus \Gamma_2 \vdash_a \mathsf{val}\, b = \mathsf{actor}\{e_1\}; e_2 \triangleright \Delta_1, \Delta_2, \{b:[S_1 \uplus S_2]\}}\ (\textsc{Type Spawn})$$

$$\frac{\begin{array}{c}\Gamma \vdash a:[!\,m(\tilde{T}).S_a] \qquad \Gamma \vdash u:[S_u\,.\&^{\bullet}_{i\in I}\{{}^{\bullet}?m(\tilde{T}).S,\ ?m_i(\tilde{T}_i).S_i\}] \\ \tilde{T} \ll \Gamma(\tilde{u}')\lfloor_m \quad \Gamma; u:[S_u.\&_{i\in I}\{?m(\tilde{T}).S,\ ?m_i(\tilde{T}_i).S_i\}]; a:[S_a] \vdash_a e \triangleright \Delta\end{array}}{\Gamma \vdash_a u\,!\,m(\tilde{u}'); e \triangleright \Delta}\ (\textsc{Type Send})$$

$$\frac{\Gamma \vdash a:[\&_{i\in I}\{?m_i(\tilde{T}_i).S_i\}] \qquad \Gamma; a:[S_i], \tilde{x}_i:\tilde{T}_i \vdash_a e_i \triangleright \Delta_i \quad i \in I}{\Gamma \vdash_a \mathsf{react}\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I} \triangleright \&_{i\in I}(\Delta_i, \{\tilde{x}_i : T_i\})}\ (\textsc{Type Receive}) \qquad \frac{\Gamma \vdash a:[\mathsf{end}] \quad \mathsf{NoMark}(\Gamma)}{\Gamma \vdash_a \mathbb{0} \triangleright \varnothing}\ (\textsc{Type End})$$

Fig. 3. Type Rules for Actors

assumptions, which collect the resources offered and consumed by bound names and variables in e_1 and e_2. The name b of the new actor must be fresh, and a type for b must be guessed. Since the scope of b includes both e_1 and e_2, both expressions are typed under a suitable assumption for b. S_1 must correctly describe the sequences of actions performed by b's body e_1. Moreover, S_1 must contain marked input actions that correspond to the input offered by b and locally consumed by messages sent by b's body to b itself. On the other hand, the marked inputs in S_2 must correspond to the messages sent to b in e_2. In the conclusion of the rule the escape environment globally collects the type assumptions locally used for b, hence $S_1 \uplus S_2$ must be defined, that is S_1 and S_2 must be the same sequence of actions with disjoint markings.
Accordingly to (Type Send), when a sends the message $m(u')$ to the actor u:

1. the first action in the type of a is the output of m;
2. the type of u contains the input of m as a marked input. The matching input needs not to be the first action in the type of u. This allows for instance that even if u accepts a *foo* message before m, the actor a is free to first output $u\,!\,m$ and then $u\,!\,foo$, according to the semantics of AC.
3. The continuation e is typed in an updated environment where the marking of the matching input has disappeared from the type of u to record the fact that the resource has been already consumed. Moreover the type of a is updated to the continuation type $[S_a]$ to record that the output action has been already performed. Observe that this implies that the behavioural type assumed for an actor changes (decreases) as long as the actor advances in its computation.

As far as the typing of the message parameters are concerned, let first introduce some notation: given two sequences S and S', we write $S << S'$ when S is a suffix of S' independently of the markings (see Appendix A for the formal definition). Given the output $u\,!\,m(\tilde{u}')$, it might not be the case that the actual parameters $\tilde{u}'$ have the types of the formal parameters $\tilde{T}$. Since the type of an actor decreases as long as the actor computes, in the asynchronous semantics the type of an actual parameter u' at sending time can be different from (longer than) the type u' has when u processes the message. Then the type of a formal parameter T is in general a suffix of the type $\Gamma(u')$. Moreover, the marked inputs contained in $\Gamma(u')$ are those that are consumed by the body of a, while the markings contained in T correspond to the inputs offered by the formal parameter and consumed by the actor that receives the message, that in general is not a. Summing up, the rule (TYPE SEND) requires (each type in the tuple) $\tilde{T}$ to be a suffix of $\Gamma(\tilde{u}')$ (componentwise), independently of the marked actions. A stronger requirement is needed when a parameter of the message coincides with the sender a, resp. the receiver u. In these cases the type of that formal parameter must be a suffix of the residual type of a after the output, resp. the residual type of u after the input. The rule uses the following predicate (componentwise extended to tuples of types), where we call $\Gamma(u')\lfloor_m$ the type of u' "after m":

$$T << \Gamma(u')\lfloor_m \stackrel{\triangle}{=} \text{if } u' = a \text{ then } T << [S_a] \\ \text{else if } u' = u \text{ then } T << [S] \text{ else } T << \Gamma(u')$$

To type an input expression the rule (TYPE RECEIVE) requires the type of a to indicate that the next action is a non marked matching input action, and every continuation e_i to be well typed in the type environment where the type of a has advanced to $[S_i]$ and the formal parameters $\tilde{x}_i$ have been added. Observe that if the input action were marked in the type of a, it would mean that the input is consumed by the continuation e_i, which would result in a deadlock, as in, e.g., $a\{\mathsf{react}\{m \Rightarrow a\,!\,m\}\}$. Finally, the names and the types of the formal parameters are recorded in the escape environment of the conclusion of the rule, preserving the branching structure of the computation.

According to the rule (TYPE END), the expression $\mathbb{0}$ is well typed assuming that the type of a contains no more actions. Moreover, Γ must contain no marked input: a judgement like $\Gamma, b : [^{\bullet}?m(T).S] \vdash_a \mathbb{0} \triangleright \varnothing$ would mean that the typing of the body of a has assumed to consume the input of b, but it is not the case since the body is terminated but the action $?m$ of b is still marked.

Type rules for Configurations. Rule (TYPE RES CONF) shows that when a new name is introduced, a corresponding type must be guessed. The new name is local to the configuration F, but it is globally collected in the escape environment. The rule requires a to be fresh in Δ, but the derivability of the judgement in hypothesis implies that $\Gamma, a : T$ is well formed, hence $a \notin Dom(\Gamma)$.

In order to type an active actor, the rule (TYPE ACTOR) requires the (residual) actor body e to comply with the (residual) sequence of actions in $\Gamma(a)$. Moreover, the mailbox $[a \mapsto M]$ contains the list of messages M that have been received but not handled yet by the actor a. A message in M will be processed by the actor only if the type $\Gamma(a)$ contains a matching input action. The rule (TYPE MAILBOX) does not require that

$$\frac{\Gamma, a : T \vdash F \triangleright \Delta \quad a \notin Dom(\Delta)}{\Gamma \vdash (\nu a)F \triangleright \Delta, a : T}\ (\textsc{Type Res Conf}) \qquad \frac{\Gamma \vdash [a \mapsto M] \quad \Gamma \vdash_a e \triangleright \Delta}{\Gamma \vdash [a \mapsto M]\ a\{e\} \triangleright \Delta}\ (\textsc{Type Actor}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash [a \mapsto \varnothing]}\ (\textsc{Type NoMail})$$

$$\frac{\Gamma \vdash [a \mapsto M] \qquad \Gamma \vdash a : [S_a] \quad ?m(\tilde{T}).S \in Inputs([S_a]) \Rightarrow \tilde{T} << \Gamma(\tilde{b})\lfloor_m}{\Gamma \vdash [a \mapsto M \cdot m(\tilde{b})]}\ (\textsc{Type Mailbox})$$

$$\frac{\Gamma_1 \vdash F_1 \triangleright \Delta_1 \quad \Gamma_2 \vdash F_2 \triangleright \Delta_2 \quad \mathsf{actors}(F_1) \cap \mathsf{actors}(F_2) = \varnothing}{\Gamma_1, F_1 \odot \Gamma_2, F_2 \vdash F_1 \mid F_2 \triangleright \Delta_1, \Delta_2}\ (\textsc{Type Para}) \qquad \frac{\Gamma \vdash \diamond \quad \mathsf{NoMark}(\Gamma)}{\Gamma \vdash \mathbb{0} \triangleright \varnothing}\ (\textsc{Type Dead})$$

$$\frac{\Gamma_1, b : [S_1] \vdash_b e_1 \triangleright \Delta_1 \quad \Gamma_2, b : [S_2] \vdash e_2 \triangleright \Delta_2 \quad b \notin Dom(\Delta_1, \Delta_2)}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{val}\, b = \mathsf{actor}\{e_1\}; e_2 \triangleright \Delta_1, \Delta_2, b : [S_1 \uplus S_2]}\ (\textsc{Type Top Spawn})$$

Fig. 4. Type Rules for Configurations

every message has a corresponding handler in $\Gamma(a)$. However, we show in the following that in well typed systems mailboxes only contain messages that will eventually be handled by the receiving actor. Let $Inputs([S])$ be the set of top level input actions contained in S. Then the rule (TYPE MAILBOX) states that if a message in the mailbox corresponds to one of the receivable inputs, then the type of the formal parameter is a suffix of the type that Γ assigns to the actual parameter. The notation $T << \Gamma(v)\lfloor_m$ means that if $v = a$ then $T << S$ else $T << \Gamma(v)$, and it is extended to tuples of types as expected.

The rules (TYPE DEAD) and (TYPE TOP SPAWN) are similar to the corresponding rules for actors, hence we reserve a final discussion for the rule (TYPE PARA) for parallel composition. The rule (TYPE PARA) splits the type environment and the escape environment so that to ensure that the resources consumed by $F_1 \mid F_2$ are consumed either by F_1 or by F_2. To illustrate, consider

$$a : T_a, b : T_b, \ldots \vdash [a \mapsto M]\, a\{e\} \triangleright \Delta \qquad a : T'_a, b : T'_b, \ldots \vdash [b \mapsto M']\, b\{e'\} \triangleright \Delta'$$

In order to correctly compose the two actors in parallel, the marked actions in T_a, resp T_b, must be disjoint form those in T'_a, resp. T'_b. Moreover, since in the typing of an active actor the behavioural type of the actor can be a suffix of the initial type of that actor, we have that $T_a << T'_a$ and $T'_b << T_b$. Hence the merge-mark function $\uplus$ must be extended so to compose a sequence with a subsequence of actions. Let $S' \Cup S$ be a partial function defined as $S' \uplus S$ plus the following two cases, that apply when S' is a proper suffix of S:

$$S' \Cup\, !m(\tilde{T}).S = \ !m(\tilde{T}).(S' \Cup S) \qquad \text{if } S' << S$$

$$S' \Cup \&_{i \in I}\{?m_i(\tilde{T}_i).S_i\} = \&_{i \in I \setminus \{j\}}\{?m_i(\tilde{T}_i).S_i,\ ?m_j(\tilde{T}_j).(S' \Cup S_j)\} \quad \text{if } S' << S_j$$

In particular we let be undefined the case $S' \Cap \&^{\bullet}_{i\in I}\{{}^{\bullet}?m(\tilde{T}).S, ?m_i(\tilde{T}_i).S_i\}$. Indeed, if $T_a = [S']$ and $T'_a = [\&^{\bullet}_{i\in I}\{{}^{\bullet}?m(T).S, ?m_i(T_i).S_i\}]$, it means that the actor b sends the message m to a but the corresponding input handler is not in the body of a anymore. Hence, a type with a marked action must be composed with a type containing the same non-marked action, so that to ensure that the input "consumed" by a thread is actually "offered" by a parallel thread. The type environment composition is defined as follows:

$$(\Gamma_1, F_1 \odot \Gamma_2, F_2)(u) \triangleq \begin{cases} \Gamma_2(u) \uplus \Gamma_1(u) & \text{if } u \notin \mathsf{actors}(F_1) \cup \mathsf{actors}(F_2) \\ \Gamma_1(u) \Cap \Gamma_2(u) & \text{if } u \in \mathsf{actors}(F_1) \\ \Gamma_2(u) \Cap \Gamma_1(u) & \text{if } u \in \mathsf{actors}(F_2) \end{cases}$$

where $\mathsf{actors}(F)$ collects the free names of active actors in F (see Appendix A).

Example 6. Consider the program Pr in Example 2. We have that $\varnothing \vdash Pr \triangleright \{a : T^1_a \uplus T^2_a,\ b : T_b,\ x : T_x,\ y : T_y\}$, which comes from the following two judgements where e_a, resp. e_b, is the body of the actor a, resp. b:

$$\begin{array}{ll} a : T^1_a \vdash_a e_a \triangleright \{x : T_x\} & a : T^2_a,\ b : T_b \vdash_b e_b \triangleright \{y : T_y\} \\ T^1_a = [?ping(T_x).\,!\,pong(T_y).?pang.\mathsf{end}] & T_x = [{}^{\bullet}?pong(T_y).\,!\,pang.\mathsf{end}] \\ T^2_a = [{}^{\bullet}?ping(T_x).\,!\,pong(T_y).?pang.\mathsf{end}] & T_y = [{}^{\bullet}?pang.\mathsf{end}] \\ T_b = [\,!\,ping(T_x).?pong(T_y).\,!\,pang.\mathsf{end}] & \end{array}$$

Preventing deadlocks. The type system described so far is enough to prove that actor implementations comply with the prescribed protocol, however program execution may stuck in a deadlock state, as for the program $P = \mathsf{val}\, a = \mathsf{actor}\{\mathsf{val}\, b = \mathsf{actor}\{\mathsf{react}\{n \Rightarrow a\,!\,m\}\}\,; \mathsf{react}\{m \Rightarrow b\,!\,n\}\}$ which is so that

$$P \longrightarrow [a \mapsto \varnothing]\, a\{\mathsf{react}\{m \Rightarrow b\,!\,n\}\} \mid [b \mapsto \varnothing]\, b\{\mathsf{react}\{n \Rightarrow a\,!\,m\}\} \not\longrightarrow$$

In order to prevent deadlocks we propose a simple technique that nicely copes with finite actor computation. We add more structure to types: we modify the syntax of types so to have output actions of the form $T\,!\,m(\tilde{T}).S$, where the additional component T describes the sequence of actions performed by the target actor *after processing the message m*. For instance, $a : [[S_b]\,!\,m(T).\mathsf{end}]$ is the type of an actor a that sends the message m to an actor that eventually reads the message m and then continues as described by S_b. Let b be the target of such a message, and let be $b : [S.?m(T).S_b]$. In asynchronous communication, when a delivers the message to b, it cannot know when b will process such a message, but it can safely assume that after the input of m, b will continue as S_b. Adding such a piece of information into types is enough to disallow deadlocks. Indeed, the typing of the program P above requires (overlooking the markings) the assumptions $a : [?m.T'\,!\,n.\mathsf{end}], b : [?n.T''\,!\,m.\mathsf{end}]$, that are not well defined since T' and T'' can only be mutual recursively defined: $T' = [T''\,!\,m.\mathsf{end}]$ and $T'' = [T'\,!\,n.\mathsf{end}]$. In other terms, there are no (finite) types so that P is well typed.

It turns out that the refinement of types leaves unchanged most of the type rules presented above. We only have to do a couple of modifications. First, the type assumption for the actor a in the rule (TYPE SEND) must be $\Gamma \vdash a[[S']\,!\,m(\tilde{T}).S_a]$, with S' equal to S

but for the markings. That is we add to the output action the sequence S indicated in the type of the target actor u as the continuation after the input of m. Then we have to adapt the relations between types that we introduced: $\uplus$, resp. $\Subset$, are obtained by adapting the clauses dealing with output actions, i.e., $T\,!\,m(\tilde{T}).S \uplus T\,!\,m(\tilde{T}).S' = T\,!\,m(\tilde{T}).(S \uplus S')$, resp. $S' \Subset T\,!\,m(\tilde{T}).S = T\,!\,m(\tilde{T}).(S' \Subset S)$.

3.2 Properties of the Type System

We show that the type system respects the semantics of AC, i.e., well typed configurations reduce to well typed configurations. However, since actor types decrease as long as the computation proceeds, the subject reduction theorem relies on the following notion of environment consumption.

Definition 7 (Environment Consumption).

- *We write* $\Gamma' << \Gamma$ *when* $Dom(\Gamma) = Dom(\Gamma')$ *and* $\forall u \in Dom(\Gamma)$, $\Gamma'(u) = [S'], \Gamma(u) = [S]$ *such that* $S' << S$.
- $\Delta' << \Delta$ *when* $\Delta = \&_{i\in I}\Gamma_i$, $\Delta' = \&_{j\in J}\Gamma'_j$ *with* $J \subseteq I$, $\Gamma'_j \subseteq \Gamma_j$ *and* $\forall u \in Dom(\Gamma'_j), \forall j \in J.\ \Gamma'_j(u) << \Gamma_j(u)$.

The substitution lemma allows a name b to be substituted for a variable x. In the lemma the type of x is assumed to be a suffix of the type of b, and when x is unified with b, the markings assumed for b must be updated so that they also contain those assumed for x. With an abuse of notation, when $S' << S$ we let $S' \uplus S$ be defined as $S' \Subset S$ plus the clause $S' \uplus \&^{\bullet}_{i\in I}\{{}^{\bullet}?m(T).S, ?m_i(T_i).S_i\} = \&^{\bullet}_{i\in I}\{{}^{\bullet}?m(T).(S' \uplus S), ?m_i(T_i).S_i,\}$, that were forbidden in the composition of parallel threads. Such a clause here is not a problem since the substitution lemma applies within a single thread, i.e. within the body of an actor at the moment of receiving an input, where it is safe to merge local assumptions.

Lemma 8 (Substitution). *Let be* $\Gamma, x : T \vdash_a e \triangleright \Delta$

- *let* c *be an actor name s.t.* $a \neq c$ *and* $T << \Gamma(c)$, *then* $\Gamma; c : T \uplus \Gamma(c) \vdash_a e\{^c/_x\} \triangleright \Delta$;
- *let be* $?m(T).S \in Input(\Gamma(a))$ *such that* $T << [S]$, *then* $\Gamma; a : T \uplus \Gamma(a) \vdash_a e\{^a/_x\} \triangleright \Delta$.

Theorem 9 (Subject Reduction). *If* $\Gamma \vdash F \triangleright \Delta$ *and* $F \longrightarrow F'$, *then there exist* Γ' *such that* $\Gamma' \vdash F' \triangleright \Delta'$, *with* $\Gamma' << \Gamma$ *and* $\Delta' << \Delta$.

Let F be a well typed closed system, i.e. $\varnothing \vdash F \triangleright \Delta$. We have that any input action that is marked in Δ exactly corresponds to one output expression in F that eventually consumes that input. We say that a well typed actor system is *balanced* whenever *every* input in the system appears marked in the escape environment. As a consequence, in balanced systems every input has a matching output and viceversa. Then (finite) balanced systems eventually terminate in the empty configuration, correctly implementing the communication protocol defined by the typing.

The definition of balanced environment checks that every actor has a fully marked type, possibly with the contribution of the markings contained in the types of a number of variables. Indeed, since actor names are passed as parameters, the inputs offered by an actor a can be consumed by outputs directed to variables that are dynamically substituted with a, as in $b\{\mathsf{react}\{foo(x) \Rightarrow x\,!\,m\}\} \mid a\{b\,!\,foo(a);\mathsf{react}\{m \Rightarrow e\}\}$. Let be $\mathsf{fullmrk}([S]) = [S']$ where S' and S are the same sequence of actions, but in S' every top level input is marked.

Definition 10 (Balanced environment). *We write* $\mathsf{balanced}(\Delta)$ *when*

- *if* $\Delta = \{x_1 : T_1, \ldots, x_n : T_n\}$ *then* $\mathsf{NoMark}(T_1), \ldots \mathsf{NoMark}(T_n)$*;*
- *if* $\Delta = \{u_1 : T_1, \ldots, u_n : T_n\}$*, then for any name* $a \in Dom(\Delta)$
  1. $\exists\, x_1,\ldots,x_k \in Dom(\Delta)$ *such that* $\Delta(a) = T$*,* $\Delta(x_i) = T_i$ *with* $T_i << T$ *and* $((T \uplus T_1) \uplus \ldots \uplus T_k) = \mathsf{fullmrk}(T)$
  2. $\mathsf{balanced}(\Delta \setminus \{a : T, x_1 : T_1, \ldots, x_k : T_k\})$*;*
- *if* $\Delta = \&_{i\in I}\Gamma_i$*, then* $\mathsf{balanced}(\Gamma_i)$ *for any* $i \in I$*.*

Observe that the escape environment in Example 6 is balanced. Indeed we have that $\Delta = \{a{:}T_a^1 \uplus T_a^2,\ b{:}T_b,\ x{:}T_x,\ y{:}T_y\}$, $\Delta(y) << \Delta(a)$ and $\Delta(y) \uplus \Delta(a) = \mathsf{fullmrk}(\Delta(a))$. Similarly, $\Delta(x) << \Delta(b)$ and $\Delta(x) \uplus \Delta(b) = \mathsf{fullmrk}(\Delta(b))$.

Let $\longrightarrow^*$ be the transitive closure of $\longrightarrow$. A final lemma shows that during the computation of well typed actor systems mailboxes only contain messages that are eventually handled by the receiving actor.

Lemma 11. *If* $\varnothing \vdash Pr \triangleright \Delta$ *and* $Pr \longrightarrow^* (\nu a_1,..,a_k)([a \mapsto M]a\{e\} \mid F)$*, then there exists* Γ *such that* $\Gamma \vdash [a \mapsto M]a\{e\}$ *and for any* $m(v) \in M$*, there exists a matching input action* $?m(T).S$ *that belongs to* $Inputs(\Gamma(a))$*.*

Theorem 12 (Safety). *If* $\varnothing \vdash Pr \triangleright \Delta$ *with* $\mathsf{balanced}(\Delta)$*. If* $Pr \longrightarrow^* F$ *then either* $F = \mathbf{0}$ *or* $F \longrightarrow F'$ *for some* F'*.*

Example 13. Consider the program $\hat{Pr}$:

$$\mathsf{val}\, a = \mathsf{actor}\{\mathsf{react}\{ping(x) \Rightarrow x\,!\,pong(\mathsf{self});\mathsf{react}\{pang() \Rightarrow \mathbf{0}\}\}\}\ ;$$

$$\mathsf{val}\, b = \mathsf{actor}\{a\,!\,ping(\mathsf{self});\mathsf{react}\{pong(y) \Rightarrow \mathbf{0}\}\}\ ;\ \mathbf{0}$$

It is easy to see that $\hat{Pr} \longrightarrow^* [a \mapsto \varnothing]\ a\{\mathsf{react}\{pang() \Rightarrow \mathbf{0}\}\} \mid \mathbf{0} \not\longrightarrow$ since there is no actor sending the message that a is waiting for. Nevertheless, the program can be typed, but with an escape environment that is not balanced. Indeed, $\varnothing \vdash \hat{Pr} \triangleright\ \Delta$ is derivable with $\Delta = \{a : [^\bullet ?ping(T_x).T^*],\ b : [T^*\,!\,ping(T_x).?pong([?pang.\mathsf{end}]).\mathsf{end}],\ x : T_x {=} [^\bullet ?pong([?pang.\mathsf{end}]).\mathsf{end}],\ y : [?pang.\mathsf{end}]\ \}$ where $T^* = [[\mathsf{end}]\,!\,pong([?pang.\mathsf{end}]).?pang.\mathsf{end}]$. Then the communications are well typed but the fact that in Δ there is no mark for the input $?pang$ shows that the program does not consume that resource, as indeed the operational semantics above has shown.

Example 14. As a final example observe that the deadlock program discussed above, that is $P = \mathsf{val}\, a = \mathsf{actor}\{\mathsf{val}\, b = \mathsf{actor}\{\mathsf{react}\{n \Rightarrow a\,!\,m\}\};\mathsf{react}\{m \Rightarrow b\,!\,n\}\}$, is balanced since a's input is matched by b's output and viceversa. However, the program stucks in a deadlock and indeed it is not well typed, according to the safety theorem.

4 Conclusions and Related Work

We presented AC, a core actor calculus designed around the basic primitives of the Scala's Actors API, together with a behavioural type system and a safety property dealing with the determinism of finite actor communications. We think that this work sheds light on how formal methods developed in the context of linear types and session types for the π-calculus can be profitably reused for the analysis of actor systems.

As we pointed out in the Introduction, our type system draws inspiration from the formal methods developed in the contexts of linear types and session types for process algebras ([16,11,13,8]). Indeed, the Actor programming model shares with session types the idea of conceiving the computation as the implementation of a specified communication protocol. However, there are a number of key differences between the two models. First, in asynchronous session types ([8,6]) two sequential outputs are (asynchronously) processed according to the sending order, while if an actor a sends two messages to the actor b, i.e. $a\{b\,!\,m_1; b\,!\,m_2\}$, then b is free to read/process the second message before reading/processing the first one, i.e. $b\{\mathsf{react}\{m_2 \Rightarrow \mathsf{react}\{m_1 \Rightarrow \ldots\}\}\}$. This means that in the Actor model we have to deal with looser assumptions, in that we cannot look at the order of input, resp. output, actions to infer something about the order of the dual output, resp. input actions. More importantly, in multiparty session types ([12,7]) the set of interacting parties (or the set of interacting roles) in a given session is known from the beginning, while an actor system is a dynamic set of interacting parties. In particular, since new actors can be created and actor names can be passed as parameters, the communication capabilities of actors dynamically increase, in a way similar to the scope extrusion phenomenon of the π-calculus.

Intuitively, session types describe the flow of communications withing a single conversation session. An actor's behavioural type instead takes the point of view of an entity that concurrently participates to different (interleaved) conversations with different parties. In this sense the Actor model share some similarities with the Conversation Calculus [17,4] CC, where processes concurrently participate to multiparty conversations and conversation context identities can be passed around to allow participants to dynamically join conversations. The Conversation Calculus is designed to model service-oriented computation, and it is centered around the notion of conversation context, which is a medium where related interactions take place. The main difference with the Actor model is that in CC named entities are the conversation contexts, while named actors are the conversant parties. Hence the powerful type system in [4] associates types to conversations rather than to actors.

To he best of our knowledge, there are few works dealing with type systems for Actor calculi. In [2] the Actor model is encodes in a typed variant of the π-calculus, where types are used to ensure uniqueness of actor names and freshness of names of newly created actors. The work in [5] study a type system for a primitive actor calculus, called CAP, which is essentially a calculus of concurrent objects à la Abadi and Cardelli [1] where actors are objects that dynamically, that is in response to method invocation, change the set of available methods. Such a dynamic behaviour may lead to so called "orphan messages" which may not be handled by the the target actor in some execution path. In order to avoid such orphan messages, a type system is proposed so to provide a safe abstraction of the execution branches. Finally, In [14] a concurrent fragment of the

Erlang language is enriched with sessions and session types. The safety property guaranteed by the typing is that all within-session messages have a chance of being received and sending and receiving follows the patterns prescribed by types. In our work we followed a different approach: instead of adding sessions to an actor calculus, we reused session type techniques to deal with the communication model distinctive of Actors.

As for future work we plan to extend the AC calculus to deal with recursive actors. Infinite computation requires a different formulation of the safety property, since compliance with the intended protocol does not reduces anymore to the termination of all actors with empty mailboxes. Moreover, deadlock freedom requires more sophisticated techniques, such as those in [6,4] that are based on a proof system that identifies cyclic dependencies between actions.

Acknowledgements. The author is indebted to Mariangiola Dezani-Ciancaglini and Luca Padovani for insightful discussions about session type theories.

References

1. Abadi, M., Cardelli, L.: A theory of primitive objects - untyped and first-order systems. In: Proc. of TACS. LNCS, vol. 789, pp. 296–320. Springer (1994)
2. Agha, G., Thati, P.: An algebraic theory of actors and its application to a simple object-based language. In: Essays in Memory of Ole-Johan Dahl. LNCS, vol. 2635, pp. 26–57 (2004)
3. Agha, G.: ACTORS - a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence, MIT Press (1990)
4. Caires, L., Vieira, H.: Conversation types. TCS 411(51-52), 4399–4440 (2010)
5. Colaco, J., Pantel, M., Dagnat, F., Sallé, P.: Static safety analysis for non-uniform service availability in actors. In: Proc. of FMOODS. vol. 139. Kluwer (1999)
6. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N.: Asynchronous session types and progress for object oriented languages. In: Proc. of FMOODS. LNCS, vol. 4468, pp. 1–31. Springer (2007)
7. Deniélou, P., Yoshida, N.: Dynamic multirole session types. In: Proc. of POPL. pp. 435–446. ACM (2011)
8. Gay, S., Vasconcelos, V.: Linear type theory for asynchronous session types. J. Funct. Program. 20(1), 19–50 (2010)
9. Haller, P., Sommers, F.: Actors in Scala. Artima Series, Artima Inc. (2012)
10. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI. pp. 235–245 (1973)
11. Honda, K., Vasconcelos, V., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Proc. of ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998)
12. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proc. of POPL. pp. 273–284. ACM (2008)
13. Kobayashi, N., Pierce, B., Turner, D.: Linearity and the pi-calculus. ACM Trans. Program. Lang. Syst. 21, 914–947 (September 1999)
14. Mostrous, D., Vasconcelos, V.T.: Session typing for a featherweight erlang. In: Proc of COORDINATION. LNCS, vol. 6721, pp. 95–109. Springer (2011)
15. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Inc., USA (2008)
16. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Proc. of PARLE. LNCS, vol. 817, pp. 398–413. Springer (1994)
17. Vieira, H., Caires, L., Seco, J.: The conversation calculus: A model of service-oriented computation. In: Proc. of ESOP. LNCS, vol. 4960, pp. 269–283. Springer (2008)

A Notation and Useful Definitions

Let $\&^{\circ}_{i\in I}\{?m_i(T_i).S_i\}$ stands for an input action, either marked or not.

Definition 15 (Suffix). *Let S and S' be two sequences of actions, we write $S << S'$ when S is a suffix of S' according to the following rules:*

$$\frac{}{\mathsf{end} << S} \qquad \frac{}{S << S} \qquad \frac{S << S'}{S << T\,!\,m(T').S'} \qquad \frac{S << S_i \quad \exists i \in I}{S << \&^{\circ}_{i\in I}\{?m_i(T_i).S_i\}}$$

Definition 16 (Inputs). *The set of top level inputs contained in a type T, written $Inputs(T)$, is defined as follows:*

$$Inputs([T\,!\,m(T').S]) = Inputs([S])$$
$$Inputs([\&^{\circ}_{i\in I}\{^{\circ}?m_i(T_i).S_i\}]) = \bigcup_{i\in I}\{?m_i(T_i).S_i\} \cup Inputs([S_i])$$
$$Inputs([\mathsf{end}]) = \varnothing$$

Definition 17 (Free active actors). *The set* $\mathsf{actors}(F)$ *of free active actors in the configuration F is defined as follows:*

$$\mathsf{actors}(\mathtt{0}) = \mathsf{actors}(e) = \varnothing \qquad \mathsf{actors}([a \mapsto M]a\{e\}) = \{a\}$$
$$\mathsf{actors}(F_1 \mid F_2) = \mathsf{actors}(F_1) \cup \mathsf{actors}(F_2) \qquad \mathsf{actors}((\nu a)F) = \mathsf{actors}(F) \setminus \{a\}$$

B Proof Sketches

Substitution Lemma Let be $\Gamma, x : T \vdash_a e \triangleright \Delta$ with $x \neq a$,

- let c be an actor name s.t. $a \neq c$ and $T << \Gamma(c)$, then $\Gamma; c : T \uplus \Gamma(c) \vdash_a e\{^c/_x\} \triangleright \Delta$;
- let be $?m(T).S \in Input(\Gamma(a))$ such that $T << [S]$, then $\Gamma; a : T \uplus \Gamma(a) \vdash_a e\{^a/_x\} \triangleright \Delta$.

Proof. The proof is by induction on the derivation of the judgement $\Gamma, x : T \vdash_a e \triangleright \Delta$. The base case is when $e = \mathtt{0}$. In this case the hypothesis is $\Gamma, x : T \vdash_a \mathtt{0} \triangleright \varnothing$, $\Gamma, x : T \vdash a : [\mathsf{end}]$ and $\mathsf{NoMark}(\Gamma, x : T)$, and the thesis is $\Gamma; c : T \uplus \Gamma(c) \vdash_a \mathtt{0} \triangleright \varnothing$. Hence it is sufficient to show that $\Gamma; c : T \uplus \Gamma(c) \vdash a : [\mathsf{end}]$, which is immediate if $a \neq c$. On the other hand, if $a = c$ then from the first hypothesis we have $\Gamma(a) = [\mathsf{end}]$ and from the second one we have $T << [\mathsf{end}]$, hence $T = [\mathsf{end}] = T \uplus [\mathsf{end}]$ as desired. For the inductive cases we proceed by a case analysis on the last rule that has been used to derive $\Gamma, x : T \vdash_a e \triangleright \Delta$:

(Type Spawn**)** In this case the hypothesis are $\Gamma_1, x : T_1 \uplus \Gamma_2, x : T_2 \vdash_a \mathsf{val}\, b = \mathsf{actor}\{e_1\}; e_2 \triangleright \Delta_1, \Delta_2, \{b : [S_1 \uplus S_2]\}$ and $T = T_1 \uplus T_2 << (\Gamma_1 \uplus \Gamma_2)(c)$. The first judgement comes from $\Gamma_1, x : T_1, b : [S_1] \vdash_c e_1 \triangleright \Delta_1$ and $\Gamma_2, x : T_2, b : [S_2] \vdash_a e_2 \triangleright \Delta_2$. Observe that b is fresh, then $b \neq c$, then by induction we have $\Gamma_1; c : T_1 \uplus \Gamma_1(c), b : [S_1] \vdash_b e_1\{^c/_x\} \triangleright \Delta_1$ and $\Gamma_2; c : T_2 \uplus \Gamma_2(c), b : [S_2] \vdash_a e_2\{^c/_x\} \triangleright \Delta_2$. Then by (Type Spawn) we have $\Gamma' \vdash_a \mathsf{val}\, b = \mathsf{actor}\{e_1\{^c/_x\}\}; e_2\{^c/_x\} \triangleright \Delta_1, \Delta_2, \{b : [S_1 \uplus S_2]\}$, where $\Gamma' = \Gamma_1; c : T_1 \uplus \Gamma_1(c) \uplus \Gamma_2; c : T_2 \uplus \Gamma_2(c)$. Now observe that $\Gamma' = (\Gamma_1 \uplus \Gamma_2); c : (T_1 \uplus \Gamma_1(c)) \uplus (T_2 \uplus \Gamma_2(c)) = (\Gamma_1 \uplus \Gamma_2); c : (T \uplus (\Gamma_1 \uplus \Gamma_2)(c))$ as desired.

(TYPE SEND) For simplicity let assume a single message parameter. In this case the hypothesis is $\Gamma, x:T \vdash_a u\,!\,m(u'); e \triangleright \Delta$, which come from

1. $\Gamma, x:T \vdash a : [[S]\,!\,m(T').S_a]$;
2. $\Gamma, x:T \vdash u : [S_u\,.\,\&^{\bullet}_{i\in I}\{{}^{\bullet}?m(T').S,\ ?m_i(\tilde{T}_i).S_i\}]$;
3. $T' << (\Gamma, x:T)(u')\lfloor_m$, that is if $u' = a$ then $T' << [S_a]$ else if $u' = u$ then $T' << [S]$ else $T' << \Gamma, x:T(u')$ and
4. $\Gamma, x:T\,; u:[S_u.\&_{i\in I}\{?m(T').S,\ ?m_i(\tilde{T}_i).S_i\}]\,; a:[S_a] \vdash_a e \triangleright \Delta$. Notice that $x \neq a$, then we have two subcases:
   (a) $x \neq u$, then $\Gamma; u:[S_u.\&_{i\in I}\{?m(T').S,\ ?m_i(\tilde{T}_i).S_i\}]\,; a:[S_a], x:T \vdash_a e \triangleright \Delta$
   (b) $x = u$ and $u \neq a$, then $\Gamma; a:[S_a], x:[S_u.\&_{i\in I}\{?m(T').S,\ ?m_i(\tilde{T}_i).S_i\}] \vdash_a e \triangleright \Delta$.

Now, since $x \neq a$, but it might be the case that $x = u$ and/or $x = u'$. Then we have to prove that $\Gamma; c:T \uplus \Gamma(c) \vdash_a u\{^c/_x\}\,!\,m(u'\{^c/_x\}); e\{^c/_x\} \triangleright \Delta$, which comes from the following judgements, implied by the enumeration above:

- $\Gamma \vdash a : [[S]\,!\,m(T').S_a]$, since $x \neq a$, hence $\Gamma; c:T \uplus \Gamma(c) \vdash a : T_a$, where
  - if $a \neq c$ then $T_a = [[S]\,!\,m(T').S_a]$
  - if $a = c$, then $T_a = T \uplus [[S]\,!\,m(T').S_a]$ together with the hypothesis $T << [S]$
- $\Gamma; c:T \uplus \Gamma(c) \vdash u\{^c/_x\} : T_u$, where
  - if $u \neq c, x$, i.e. $u\{^c/_x\} = u$, then $T_u = [S_u\,.\,\&^{\bullet}_{i\in I}\{{}^{\bullet}?m(T').S,\ ?m_i(\tilde{T}_i).S_i\}]$
  - if $u = c$ or $u = x$, i.e., $u\{^c/_x\} = c$, then $T_u = T \uplus [S_u\,.\,\&^{\bullet}_{i\in I}\{{}^{\bullet}?m(T').S,\ ?m_i(\tilde{T}_i).S_i\}]$, together with the hypothesis $T << \Gamma(c)$ and $T << [S]$ for $c = a$;
- by induction we have
  - in the case 4.(a), i.e. $x \neq u, a$, $(\Gamma; u:T'_u; a:[S_a]); c:T \uplus (\Gamma; u:T'_u; a:[S_a])(c) \vdash_a e\{^c/_x\} \triangleright \Delta$ where $T'_u = [S_u.\&_{i\in I}\{?m(T').S,\ ?m_i(\tilde{T}_i).S_i\}]$.
  - in the case 4.(b), i.e. $x = u \neq a$, $(\Gamma; a:[S_a]); c:T \uplus (\Gamma; a:[S_a])(c) \vdash_a e\{^c/_x\} \triangleright \Delta$ with $T = [S_u.\&_{i\in I}\{?m(T').S,\ ?m_i(\tilde{T}_i).S_i\}]$

  In both cases the environment is equal to
  $(\Gamma; c:T \uplus \Gamma(c)); u:[S_u.\&_{i\in I}\{?m(T').S,\ ?m_i(\tilde{T}_i).S_i\}]; a:[S_a]$
- let prove that $T' << (\Gamma; c:T \uplus \Gamma(c))(u'\{^c/_x\})\lfloor_m$:
  - if $u'\{^c/_x\} = a$ then either $u' = a$, then $T' << [S_a]$ by the hypothesis 3. above, or $u' = x$ and $a = c$, and in this case the hypothesis 3. above gives $T' << \Gamma, x:T(u')$, that is $T' << T$. Moreover, from the hypothesis $T << [S]$ and the fact that $S << S_a$, we have $T' << [S_a]$.
  - if $u'\{^c/_x\} = u\{^c/_x\}$ then either $u' = u$, then $T' << [S]$ by the hypothesis 3. above, or $u'\{^c/_x\} = u\{^c/_x\} = c$ and as in the previous item $T' << T$ and by hypothesis $T << [S]$, hence $T' << [S]$.
  - otherwise we have to show that $T' << (\Gamma; c:T \uplus \Gamma(c))(u'\{^c/_x\})$, that is either $T' << (\Gamma; c:T \uplus \Gamma(c))(c)$ or $T' << (\Gamma; c:T \uplus \Gamma(c))(u')$. The first case come from the fact that by hypothesis 3. above we have $T' << T$, that together with $T << \Gamma(c)$ gives what desired. The sencond case come from the fact that by hypothesis 3. above we have $T' << \Gamma(u')$, which is what desired since in this case $u' \neq c$.

(TYPE RECEIVE) In this case the hypothesis is $\Gamma, x:T \vdash_a \mathsf{react}\{m_i(\tilde{y}_i) \Rightarrow e_i\}_{i\in I} \triangleright \&_{i\in I}(\Delta_i, \{\tilde{y}_i : T_i\})$, which comes from (i) $\Gamma, x:T \vdash a : [\&_{i\in I}\{?m_i(\tilde{T}_i).S_i\}]$ and (ii) $\Gamma, x:T\,; a:[S_i], \tilde{y}_i : \tilde{T}_i \vdash_a e_i \triangleright \Delta_i$ for $i \in I$. We can assume that $x \neq \tilde{y}_i$, hence from the last judgement we have $\Gamma; a : [S_i], \tilde{y}_i : \tilde{T}_i, x : T \vdash_a e_i \triangleright \Delta_i$, which gives by inductive hypothesis $\Gamma; a : [S_i], \tilde{y}_i : \tilde{T}_i; c : T \uplus (\Gamma; a : [S_i])(c) \vdash_a e_i\{^c/_x\} \triangleright \Delta_i$, that is $\Gamma; c : T \uplus (\Gamma)(c); a : [S'_i], \tilde{y}_i{:}\tilde{T}_i \vdash_a e_i\{^c/_x\} \triangleright \Delta_i$ where $S'_i = S_i \uplus T$ if $c = a$, otherwise $S'_i = S_i$. Now, from (i) and the hypothesis $T << [S_i]$ of the lemma, we have $\Gamma; c : T \uplus (\Gamma)(c) \vdash a : [\&_{i\in I}\{?m_i(\tilde{T}_i).S'_i\}]$,hence we conclude $\Gamma; c : T \uplus (\Gamma)(c) \vdash_a \mathsf{react}\{m_i(\tilde{y}_i) \Rightarrow e_i\{^c/_x\}\}_{i\in I} \triangleright \&_{i\in I}(\Delta_i, \{\tilde{y}_i : T_i\})$ by (TYPE RECEIVE).

Lemma 18. *Let be $\Gamma = \Gamma_1, F_1 \odot \Gamma_2, F_2 \vdash F_1 \mid F_2 \triangleright \Delta_1, \Delta_2$ a derivable judgement. Then*

- *For all $u \notin \mathsf{actors}(F_1) \cup \mathsf{actors}(F_2)$, $\Gamma(u) = \Gamma_1(u) \uplus \Gamma_2(u)$, hence $\Gamma_1(u) << \Gamma_2(u)$ and $\Gamma_1(u) << \Gamma_2(u)$, i.e. the type of u has the same length in Γ_1 and Γ_2*
- *For all $u \in \mathsf{actors}(F_1)$, it holds $\Gamma_1(u) << \Gamma_2(u)$*
- *For all $u \in \mathsf{actors}(F_2)$, it holds $\Gamma_2(u) << \Gamma_1(u)$*

Subject Reduction Theorem If $\Gamma \vdash F \triangleright \Delta$ and $F \longrightarrow F'$, then there exist Γ' such that $\Gamma' \vdash F' \triangleright \Delta'$, with $\Gamma' << \Gamma$ and $\Delta' << \Delta$.

Proof. The proof is by induction on the derivation of $F \longrightarrow F'$. We start with the base cases:

(ENDED) in this case the hypothesis are $\Gamma \vdash [a \mapsto \varnothing]\, a\{\underline{0}\} \triangleright \Delta$ and $[a \mapsto \varnothing]\, a\{\underline{0}\} \longrightarrow \underline{0}$. From the first hypothesis we have that $\Gamma \vdash [a \mapsto \varnothing]$ and $\Gamma \vdash_a \underline{0} \triangleright \Delta$, hence $\Delta = \varnothing$, $\Gamma \vdash a : [\mathsf{end}]$ and $\mathsf{NoMark}(\Gamma)$. Then $\Gamma \vdash \diamond$ is also derivable, and by (TYPE DEAD) we conclude $\Gamma \vdash \underline{0} \triangleright \Delta$ as desired.

(TOP SPAWN) In this case the hypothesis are $\mathsf{val}\, a = \mathsf{actor}\{e\}; e' \longrightarrow (\nu a)([a \mapsto \varnothing]\, a\{e\}\} \mid e'))$ and $\Gamma \vdash \mathsf{val}\, a = \mathsf{actor}\{e\}; e' \triangleright \Delta$, which comes form $\Gamma = \Gamma_1 \uplus \Gamma_2$, $\Delta = \Delta_1, \Delta_2, a : [S_1 \uplus S_2]$, $\Gamma_1, a : [S_1] \vdash_a e \triangleright \Delta_1$ and $\Gamma_2, a : [S_2] \vdash e' \triangleright \Delta_2$. Then we also have $\Gamma_1, a : [S_1] \vdash [a \mapsto \varnothing]\, a\{e\} \triangleright \Delta_1$, and by (TYPE PARA), $\Gamma' \vdash [a \mapsto \varnothing]\, a\{e\} \mid e' \triangleright \Delta_1, \Delta_2$ where $\Gamma' = \Gamma_1, a : [S_1] \odot \Gamma_2, a : [S_2] = \Gamma_1 \uplus \Gamma_2, a : [S_1 \uplus S_2]$. Then by (TYPE RES CONF) we conclude $\Gamma \vdash (\nu a)([a \mapsto \varnothing]\, a\{e\} \mid e') \triangleright \Delta$.

(SPAWN) in this case the hypothesis are $[b \mapsto M] b\{\mathsf{val}\, a = \mathsf{actor}\{e\}; e'\} \longrightarrow (\nu a)([b \mapsto M] b\{e'\} \mid [a \mapsto \varnothing]\, a\{e\})$ and $\Gamma \vdash [b \mapsto M] b\{\mathsf{val}\, a = \mathsf{actor}\{e\}; e'\} \triangleright \Delta$, which comes form $\Gamma \vdash [b \mapsto M]$ and $\Gamma \vdash_b \mathsf{val}\, a = \mathsf{actor}\{e\}; e' \triangleright \Delta$. Then the proof is similar to the previous case.

(SEND) In this case the hypothesis are $[a \mapsto M]\, a\{e\} \mid [b \mapsto M']\, b\{a\,!\,m(\tilde{c}); e'\} \longrightarrow [a \mapsto M \cdot m(\tilde{c})]\, a\{e\} \mid [b \mapsto M']\, b\{e'\}$ and $\Gamma \vdash [a \mapsto M]\, a\{e\} \mid [b \mapsto M']\, b\{a\,!\,m(\tilde{c}); e'\} \triangleright \Delta$, which comes from $\Gamma = \Gamma_1, F_1 \odot \Gamma_2, F_2$, $\Delta = \Delta_1, \Delta_2$, $\Gamma_1 \vdash [a \mapsto M]\, a\{e\} \triangleright \Delta_1$ and $\Gamma_2 \vdash [b \mapsto M']\, b\{a\,!\,m(\tilde{c}); e'\} \triangleright \Delta_2$. The last two judgements must have been derived from (i) $\Gamma_1 \vdash [a \mapsto M]$ and (ii) $\Gamma_1 \vdash_a e \triangleright \Delta_1$, resp. (iii) $\Gamma_2 \vdash [b \mapsto M']$ and (iv) $\Gamma_2 \vdash_b a\,!\,m(\tilde{c}); e' \triangleright \Delta_2$.
From (iv) he know that $\Gamma_2 \vdash b : [[S]\,!\,m(\tilde{T}).S_b]$, $\Gamma_2 \vdash a : [S_u\,.\,\&^{\bullet}_{i\in I}\{^{\bullet}?m(\tilde{T}).S,\ ?m_i(\tilde{T}_i).S_i\}]$, $(*)$ $\tilde{T} << \Gamma_2(\tilde{c})\lfloor_m$ and $\Gamma'_2 \vdash_b e' \triangleright \Delta_2$, then also $\Gamma'_2 \vdash b\{e'\} \triangleright \Delta_2$, where $\Gamma'_2 = \Gamma_2\,; a : [S_u.\&_{i\in I}\{?m(\tilde{T}).S,\ ?m_i(\tilde{T}_i).S_i\}]\,; b : [S_b]$. Notice that $\Gamma'_2 << \Gamma_2$. Moreover, from (iii) we also have $\Gamma'_2 \vdash [b \mapsto M']$ since $Inputs(\Gamma'_2(b)) = Inputs(\Gamma_2(b))$. Then by (TYPE ACTOR) we have $\Gamma'_2 \vdash [b \mapsto M']\, b\{e'\} \triangleright \Delta_2$.

Let show that from (i) we also have $\Gamma_1 \vdash [a \mapsto M \cdot m(\tilde{c})]$ by (TYPE MAILBOX). From $\Gamma_2 \vdash a : [S_u . \&^{\bullet}_{i\in I}\{{}^{\bullet}?m(\tilde{T}).S,\ ?m_i(\tilde{T}_i).S_i\}]$ we have $?m(\tilde{T}).S \in Inputs(\Gamma_2(a))$, then by Lemma 18 $?m(\tilde{T}).S \in Inputs(\Gamma_1(a))$. It is then sufficient to show that $\tilde{T} << \Gamma_1(\tilde{c})\lfloor_m$, that is if $c = a$ then $T << S$ else $T << \Gamma_1(c)$. From $(*)$ above we know that if $c = b$ then $T << \Gamma_2(b)$, if $c = a$ then $T << S$ else $T << \Gamma_2(c)$. By Lemma 18 $\Gamma_2(c) << \Gamma_1(c)$ and $\Gamma_2(b) << \Gamma_1(b)$ hence we have $\tilde{T} << \Gamma_1(\tilde{c})\lfloor_m$ as desired.
So we have $\Gamma_1 \vdash [a \mapsto M \cdot m(\tilde{c})]$, that together with (ii) gives $\Gamma_1 \vdash [a \mapsto M \cdot m(\tilde{c})]\, a\{e\} \triangleright \Delta_1$. Then by (TYPE PARA) we have $\Gamma' \vdash [a \mapsto M \cdot m(\tilde{c})]\, a\{e\} \mid [b \mapsto M']\, b\{e'\} \triangleright \Delta_1, \Delta_2$ where $\Gamma' = \Gamma_1 \odot \Gamma'_2$, i.e. $\Gamma' << \Gamma$ as desired.

(RECEIVE) In this case the hypothesis are $[a \mapsto M{\cdot}m_j(\tilde{c}){\cdot}M']\ a\{\mathsf{react}\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I}\} \longrightarrow [a \mapsto M{\cdot}M']\ a\{e_j\{^{\tilde{c}}/_{\tilde{x}_j}\}\}$ with $j \in I$ and $\Gamma \vdash [a \mapsto M{\cdot}m_j(\tilde{c}){\cdot}M']\ a\{\mathsf{react}\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I}\} \triangleright \Delta$, which comes form (i) $\Gamma \vdash [a \mapsto M{\cdot}m_j(\tilde{c}){\cdot}M']$ and (ii) $\Gamma \vdash_a \mathsf{react}\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I} \triangleright\ \Delta$, where $\Delta = \&_{i\in I}(\Delta_i, \{\tilde{x}_i : T_i\})$. From (ii) and $j \in I$ we have $\Gamma \vdash a : [\&_{i\in I}\{?m_i(\tilde{T}_i).S_i\}]$ and $\Gamma; a : [S_j], \tilde{x}_j : \tilde{T}_j \vdash_a e_j \triangleright \Delta_j$. Now, from (i) and $?m_j(\tilde{T}_j).S_j \in Input(\Gamma(a))$ we have $\tilde{T} << \Gamma(\tilde{c})\lfloor_m$, that is if $c = a$ then $T << S_j$ else $T << \Gamma(c)$. Let be $\Gamma' = \Gamma; a : [S_j]$, then we also have $\tilde{T} << \Gamma'(\tilde{c})\lfloor_m$. From $\Gamma', \tilde{x}_j : \tilde{T}_j \vdash_a e_j \triangleright \Delta_j$ and $\tilde{T} << \Gamma'(\tilde{c})\lfloor_m$, by Substitution Lemma we have $\Gamma'; \tilde{c} : \tilde{T}_j \uplus \Gamma'(\tilde{c}) \vdash_a e_j\{^{\tilde{c}}/_{\tilde{x}}\} \triangleright \Delta_j$. Let be $\Gamma^* = \Gamma'; \tilde{c} : \tilde{T}_j \uplus \Gamma'(\tilde{c})$. Note that $\Gamma^* << \Gamma$ and from (i) we also have $\Gamma^* \vdash [a \mapsto M \cdot M']$, than by (TYPE ACTOR) we conclude $\Gamma^* \vdash [a \mapsto M \cdot M'] \vdash a\{e_j\{^{\tilde{c}}/_{\tilde{x}}\}\} \triangleright \Delta_j$, with $\Delta_j << \Delta$.

For the inductive cases we proceed by a case analysis on the last rule tha has been applied:

(PAR) In this case the hypothesis is $F_1 \mid F_2 \longrightarrow F'_1 \mid F_2$ since $F_1 \longrightarrow F'_1$, and $\Gamma \vdash F_1 \mid F_2 \triangleright \Delta$. Hence $\Delta = \Delta_1, \Delta_2$, $\Gamma = \Gamma_1, F_1 \odot \Gamma_2, F_2$, $\Gamma_1 \vdash F_1 \triangleright\ \Delta_1$ and $\Gamma_2 \vdash F_2 \triangleright\ \Delta_2$. Then by inductive hypothesis we have $\Gamma'_1 \vdash F'_1 \triangleright\ \Delta'_1$ with $\Gamma'_1 << \Gamma_1$ and $\Delta'_1 << \Delta_1$. Then by (TYPE PARA) we have $\Gamma' \vdash F'_1 \mid F_2 \triangleright\ \Delta'$ with $\Delta' = \Delta'_1, \Delta_2$ and $\Gamma' = \Gamma'_1, F'_1 \odot \Gamma_2, F_2$ (note that we can guarantee that $\mathsf{actors}(F'_1) \cap \mathsf{actors}(F_2) = \varnothing$ since we assumed that dynamically spawned actors have fresh names). Then observe that $\Delta' << \Delta$, hence it is sufficient to show that $\Gamma' << \Gamma$, which comes oberving that $\Gamma'_1(u) = \Gamma_1(u)$ for all $u \notin \mathsf{actors}(F_1)$ while $\Gamma'_1(u) << \Gamma_1(u)$ for all $u \in \mathsf{actors}(F_1) \cap \mathsf{actors}(F'_1)$.

(RES) In this case the hypothesis is $(\nu a)F \longrightarrow (\nu a)F'$ since $F \longrightarrow F'$, and $\Gamma \vdash (\nu a)F \triangleright \Delta, a : T$. The last judgement comes from $\Gamma, a : T \vdash F \triangleright \Delta$, which gives, by inductive hypothesis, $\Gamma', a : T' \vdash F' \triangleright\ \Delta'$ with $\Gamma', a : T' << \Gamma, a : T$ and $\Delta' << \Delta$, and we conclude $\Gamma \vdash (\nu a)F' \triangleright \Delta, a : T'$ by (TYPE RES CONF).

(STRUCT) In this case the hypothesis is $F \longrightarrow F'$ since $F \equiv F'$, $F' \longrightarrow F''$ and $F'' \equiv F'''$. This case comes by the fact that structural congruence preservs the typing.

Lemma 11 If $\varnothing \vdash Pr \triangleright\ \Delta$ and $Pr \longrightarrow^* (\nu\tilde{a})([a \mapsto M]a\{e\} \mid F)$, then there exists Γ such that $\Gamma \vdash [a \mapsto M]a\{e\}$ and for any $m(v) \in M$, there exists a matching input action $?m(T).S$ that belongs to $Inputs(\Gamma(a))$.

Proof. (Sketch) Since actor initially have an empty mailbox, if $m(v) \in M$, then it must be $Pr \longrightarrow^* (\nu\tilde{a}')([b \mapsto N]\, b\{a\,!\,m(v)e_b\} \mid [a \mapsto M']a\{e'\}) \longrightarrow^* (\nu\tilde{a})([a \mapsto M]a\{e\} \mid F)$. By Subject Reduction we know that the actor $b\{a\,!\,m(v); e_b\}$ is well typed, that is $\Gamma_b \vdash a : [S_a.\&^{\bullet}\{?m(T).S, ..\}]$, hence $?m(T).S \in Inputs(\Gamma_b(a))$. Now, if $?m(T).S \notin Inputs(\Gamma(a))$, it means that the input handler in a has been consumed by another output that would also require the type $a : [S_a.\&^{\bullet}\{?m(T).S,..\}]$, which is not possible since marking is linear.

Lemma 19.
Let be $\varnothing \vdash Pr \triangleright \Delta$ *with* $\mathsf{balanced}(\Delta)$ *and* $Pr \longrightarrow^* (\nu\tilde{a})([a_1 \mapsto M_1]\, a_1\{e_1\} \mid \ldots \mid [a_k \mapsto M_k]\, a_k\{e_k\}) \not\longrightarrow$. *Then it is not possible that every actor body* e_i *is a (stuck) input expression.*

Proof. (Sketch) By Subject Reduction there exists Δ' such that
$\varnothing \vdash (\nu\tilde{a})([a_1 \mapsto M_1]\, a_1\{e_1\} \mid \ldots \mid [a_k \mapsto M_k]\, a_k\{e_k\}) \triangleright \Delta'$, hence there exist $\Gamma_1,...,\Gamma_k$ such that $\Gamma_1 \odot \ldots \odot \Gamma_k \subseteq \Delta'$ and $\Gamma_i \vdash [a_i \mapsto M_i]\, a_i\{e_i\} \triangleright \Delta_i$ for $i = 1,..,k$. By contradiction, assume that every actor body e_i is a (stuck) input expression. Then we have that for all $i = 1,..,k$, $\Gamma_i \vdash a_i : [\&\{?m^i_\ell(x^i_\ell).S^i_\ell\}]$. Moreover, since the initial system is balanced and since by hypothesis no matching message is already in the mailbox, for any actor a_i there must be an actor a_j whose body sends a matching message, i.e. e_j must contain the sub-expression $a_i\,!\,m^i_\ell(c)$ for some message m_ℓ. However, this is not possible since the typing of (the output action of) a_j depends on the (continuation of the input) type of a_i and so on yielding a cyclic dependence between a set of actors which would require recursive typing.

Safety Theorem Let be $\varnothing \vdash Pr \triangleright \Delta$ with $\mathsf{balanced}(\Delta)$. If $Pr \longrightarrow^* F$ then either $F = \mathbf{0}$ or $F \longrightarrow F'$ for some F'.

Proof. (Sketch) Let prove it by contradiction: assume that there exists a configuration $F^* \neq \mathbf{0}$ such that $F^* \not\longrightarrow$. We can assume $F^* \equiv (\nu\tilde{a})([a_1 \mapsto M_1]\, a_1\{e_1\} \mid \ldots \mid [a_k \mapsto M_k]\, a_k\{e_k\})$ with $\{a_1,...,a_k\} \subseteq \tilde{a}$. Then by Subject Reduction there exists Δ' such that $\varnothing \vdash F^* \triangleright \Delta'$, with $\Delta' << \Delta$ and $\mathsf{balanced}(\Delta')$. Then we also have that there exist $\Gamma_1,...,\Gamma_k$ such that $\Gamma_1 \odot \ldots \odot \Gamma_k \subseteq \Delta'$ and $\Gamma_i \vdash [a_i \mapsto M_i]\, a_i\{e_i\} \triangleright \Delta_i$ for $i = 1,..,k$. We have one of the following cases:

- for all $i \in \{1,..,k\}$, if $e_i = a_j\,!\,m(\tilde{c}); e$ then we have that $\Gamma_i \vdash a_j : [S_u.\&^{\bullet}\{?^{\bullet}m(\tilde{T}).S,..\}]$ and by definition of $\odot$ we have that $\Gamma_j(a) = [S'_u.\&\{?m(\tilde{T}).S,..\}]$ for some $S'_u << S_u$, that is the input handler of the message m is still in the body of the actor a_j, i.e. $e_j \neq \mathbf{0}$, that is $a_j \in \{a_1,...,a_k\}$ and $F^* \longrightarrow$ contradicting the assumption.
- for all $i \in \{1,..,k\}$, if $e_i = \mathbf{0}$ we have two cases: if $M_i = \varnothing$ then the (ENDED) reduction rule applies, giving a contradiction. On the other hand, let be $M_i \neq \varnothing$, from $e_i = \mathbf{0}$ and the typing we have $\Gamma_i \vdash a_i : [\mathsf{end}]$, but by Lemma 11 and $M_i \neq \varnothing$ we have $\Gamma \not\vdash a_i : [\mathsf{end}]$, giving the desired contradiction.
- If an actor body starts with the spawning of a new actor, then trivially $F^* \longrightarrow$, giving the contradiction.
- Finally, we have the case where every actor body e_i starts with an input expression, which is not possible by Lemma 19.