View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

Types for Progress in Actor Programs*

Minas Charalambides Karl Palmskog Gul Agha
University of Illinois at Urbana-Champaign, USA
{charala1,palmskog,agha}@illinois.edu

Properties in the actor model can be described in terms of the message-passing behavior of actors. In this paper, we address the problem of using a type system to capture liveness properties of actor programs. Specifically, we define a simple actor language in which demands for certain types of messages may be generated during execution, in a manner specified by the programmer. For example, we may want to require that each request to an actor eventually results in a reply. The difficulty lies in that such requests can be generated dynamically, alongside the associated requirements for replies. Such replies might be sent in response to intermediate messages that never arrive, but the property may also not hold for more trivial reasons; for instance, when the code of potential senders of the reply omit the required sending command in some branches of a conditional statement. We show that, for a restricted class of actor programs, a system that tracks typestates can statically guarantee that such dynamically generated requirements will eventually be satisfied.

1 Introduction

Liveness properties [4] state that a system will eventually produce an event of interest [24]. For example, a client may request exclusive use of a resource from a server, with the expectation that there will eventually be a reply indicating whether access has been granted or denied. Liveness properties are generally difficult to express and reason about; this is primarily because they are formulated over, and thus require reasoning on, sequences of runtime configurations.

Usually, type systems provide a straightforward way to capture *safety* properties of programs, i.e., properties which rule out executions that reach undesirable states. In contrast to liveness, safety can be established by analyzing single runtime transition steps. However, work in *session types* [35] has shown the feasibility of using a type system to establish certain notions of progress. These works apply type discipline to the use of communication channels in the π-calculus, and have viewed issues of progress under the prism of session fidelity: communication protocols, and hence the types that describe them, are designed so that adhering participants never get stuck. Session types usually constrain cyclic communication dependencies on the level of the protocol itself, so that well-typed processes communicate in a manner that always makes progress [5, 9, 19]. However, as Sumii and Kobayashi [34] remark, there is more to progress than breaking cyclic dependencies: *programmer intent* should be taken into account.

We are interested in guaranteeing that an implementation adheres to the programmer's intent on the delivery of certain messages. For example, the programmer may demand that whenever a client requests resource access from a server, it must eventually receive a reply. This reply does not necessarily need to come from the server process itself, but it does need to specify whether access to the resource has been granted or not. Certain bugs in the implementation can violate this requirement, for example, due to cyclic dependencies, or the omission of a message sending command by the programmer.

This paper presents a possible solution to the problem in the context of actors [1], which communicate via asynchronous message-passing. Our work is structured around a simple actor calculus that allows the

*An improved version of this paper is available [13], [10, chap. 4].

Presented at WAO 2017, Torino, Italy
http://costa.fdi.ucm.es/wao17/

© M. Charalambides, K. Palmskog & G. Agha
This work is licensed under the
Creative Commons Attribution License.

programmer to specify how messaging requirements may be generated at runtime. We regard *progress* to be a persistent property on the (runtime) program state such that a configuration C satisfies progress if every execution trace from C ends in a state where all (dynamically) generated messaging requirements have been satisfied. We propose a type system to guarantee this property for every runtime configuration resulting from the program. We use the notion of a *typestate* [33] on both the language level, and in the presented meta-theory, tagging actor names with the multi-set of message types that the corresponding actor needs to receive. We therefore regard progress as the question of whether an actor eventually receives all the messages included in their typestate. The type system enforces that, for every requirement that appears at runtime, suitable action is taken: it is either fulfilled in the current scope, or it is delegated to another actor. By recursive reasoning, the type system guarantees that such a postponed requirement will be satisfied by the delegate actor. In essence, we show that in all executions of well-typed programs, all requirements generated at runtime will eventually result in the corresponding messages being received.

The most important shortcoming of our system is that the typing does not terminate for programs with cyclic messaging patterns. Nevertheless, it is possible to overcome this limitation by extending the algorithm with a memoization technique, outlined in section 7. Moreover, in order to keep the presentation simple, the paper does not deal with any safety properties—those can be established with a separate type system that checks, for example, the number and types of handler arguments.

Paper outline. We first present, in section 2, two example actor programs that demonstrate the usefulness of explicit message requirements with regard to establishing progress. We then define our actor calculus formally in section 3, with its abstract syntax and operational semantics. The type system is defined in section 4, where we give example typings and discuss the system's limitations. In section 5, we show that executions of well-typed programs eventually satisfy stated requirements. Lastly, we cover related work in section 6, and conclude in section 7.

2 Motivating Examples

We motivate our approach by discussing two simple programs, given in pseudo-code syntax reminiscent of Scala [32] using the Akka toolkit [25] for actors. Later, in section 3, we will define a minimal calculus to make the underlying ideas precise. The first program, shown in listing 1, embodies a resource sharing scenario among multiple clients via a central server. The second program, shown in listing 2, implements a classic example from the session types literature, where two buyers coordinate to purchase a book from a seller.

2.1 Resource Sharing Program

In the resource sharing program of listing 1, there are three kinds of actors: servers, clients, and resources. Specifically, the program initially spawns two client actors and a server actor. The client actors attempt to acquire exclusive access to resources administered by the server actor, by sending it `request` messages. The server is responsible for responding to requests, by creating new resource actors and handing out their names, as permitted by system limits. The problem that we consider in this example is how to ensure that (a) clients eventually receive some reply after a request, whether positive or negative; (b) resources are properly allocated and de-allocated; and that (c) allocated resources are eventually put to work.

Actors are defined by their *behavior*, i.e., how they respond to messages, and their persistent *state*. For example, server actors have a `count` state variable that represents the number of available resources,

Listing 1: Example – *Resource Sharing.*

```
1  Server(count : Int) = {
2    request(c : Client) =
3      if count ≤ 0 then
4        c ! later()
5      else
6        let rs = new Resource() in
7        let rs_ = rs.add_req(kill) in
8          rs_ ! lock(c, self);
9          update(count - 1)
10
11   done(r : Resource) =
12     r ! kill();
13     update(count + 1)
14 }
15
16 Client() = {
17   start(s : Server) =
18     let self_ = self.add_req(ok + later) in
19       s ! request( self_ )
20
21   ok(r : Resource, s : Server) =
22     r ! work( self, s )
23
24   later() = ...
25
26   done(r : Resource, s : Server) =
27     s ! done(r)
28 }
29
30 Resource() = {
31   lock(c : Client, s : Server) =
32     let self_ = self.add_req(work) in
33       c ! ok( self_, s )
34
35   work(c : Client, s : Server) =
36     ...
37     c ! done(self, s);
38 }
39
40 let s = new Server(1) in
41 let c1 = new Client() in
42 let c2 = new Client() in
43   c1 ! start(s);
44   c2 ! start(s)
```

Listing 2: Example – *Two Buyer Protocol.*

```
45 Seller() = {
46   get_quote(title : String,
47             b1 : Buyer1,
48             b2 : Buyer2) =
49     let price = price_of(title) in
50     let self_ = self.add_req(yes + no) in
51       b1 ! quote(price, b2, self_)
52
53   yes() =
54     ...
55
56   no() =
57     ...
58 }
59
60 Buyer1(contr : Int) = {
61   start(s : Seller,
62         b2 : Buyer2) =
63     let self_ = self.add_req(quote) in
64       s ! get_quote("1984", self_, b2)
65
66   quote(price : Int,
67         b2 : Buyer2,
68         s : Seller) =
69     b2 ! ask(price, contr, s)
70 }
71
72 Buyer2(contr : Int) = {
73   ask(price : Int,
74       b1_contr : Int,
75       s : Seller) =
76     if price - b1_contr ≤ contr then
77       s ! yes()
78     else
79       s ! no()
80 }
81
82 let s = new Seller() in
83 let b1 = new Buyer1(11) in
84 let b2 = new Buyer2(5) in
85   b1 ! start(s, b2)
```

and handlers for request and done messages. Messages with the request label are sent to the server by client actors to ask for resource access. Upon receiving a request message, the server checks if count is zero or less (line 3), and if so, it sends a later message to the client. If count is positive, the server spawns a new resource actor to which it sends a lock message (line 8) with the client and itself as the payload; then, it decrements count. The resource reacts to lock by sending an ok message to the client (line 33), who replies with a work message (line 22).

The requirements (a), (b), and (c) from above are explicitly embedded in the code via the use of the construct add_req. On line 18, we express that the client actor currently executing this line is required to eventually receive either ok or later. On line 7, add_req expresses the requirement that the actor whose name is stored in rs (a resource actor) must eventually receive a kill message, representing de-allocation. Finally, on line 32, we express that the resource actor executing this line needs to eventually receive a work message.

As it turns out, the stated requirements will be satisfied in all executions of the program where message delivery and processing is not indefinitely postponed. With regard to the delivery of either ok or later to the clients, consider what happens when the server actor receives a request message. It is either $\mathsf{count} \leq 0$, in which case the server sends later to the client actor right away; or $\mathsf{count} > 0$, and the client (whose name was included in the message payload) eventually receives an ok message from the newly spawned resource actor.

Note that if we omit some message sending operation, requirements will be violated; for example, leaving out the statement c!done(**self**, s) from line 37 would have line 12 not get executed. This would violate the requirement set on line 7, resulting in failure to de-allocate the resource. Consequently, satisfaction of requirements captures a form of progress for actor programs, by ruling out that certain actors wait forever for some specific message.

2.2 Two Buyers Protocol

The program in listing 2 builds on the two buyer protocol—a classic example found, e.g., in the work of Honda et al. [20]. The general idea is that two buyers need to coordinate to buy a book from a seller. The first buyer sends a quote request to the seller, who replies with a price. When the first buyer receives this quote, it tells the second buyer how much it is willing to contribute, and the second buyer decides if the remaining amount is within their budget; then they let the seller know. In this example, the problem we consider is (a) whether the first buyer eventually gets a quote from the seller, and (b) whether the seller eventually gets a response to the quote.

There are three actor behaviors in the program (Seller, Buyer1, and Buyer2). Execution begins with the spawning of one actor for each, on lines 82–85. The protocol starts when the first buyer actor receives a start message with the names of the two other participants. The first buyer then sends a get_quote message to the seller actor with the title of a book. Requirement (a) is embedded through the use of add_req on line 63. In response to a get_quote message, the seller actor looks up the price of the title and sends it back in a quote message. Notice the assignment with add_req on line 50, which captures requirement (b) by demanding a yes or no message.

The first requirement is easily seen to be fulfilled by the seller actor, assuming the invocation of price_of terminates. Sending yes or no to the seller is ultimately fulfilled by the Buyer2 actor, in response to the ask message that Buyer1 sends on line 69. Once again, omitting send operations will result in requirement violations: for example, omitting s!yes() on line 77 would allow one of the branches of the conditional (line 76) to proceed without a response to the seller. As in the first example, it makes

Figure 1: Actor calculus syntax.

P	::=	$\overline{\mathtt{B}}\ \mathtt{S}$		
B	::=	**bdef** $b(\overline{x}) = \{\overline{\mathtt{H}}\}$	$b \in$ behavior names	
H	::=	**hdef** $h(\overline{x}) = \mathtt{S}$	$h \in$ handler names	
S	::=	$\mathtt{x}!h(\overline{e}).\mathtt{S}$		
	|	**add**$(\mathtt{x},\mathtt{R}).\mathtt{S}$		
	|	**if** e **then** $\mathtt{S}_1$ **else** $\mathtt{S}_2$	$e \in$ expressions (values, function calls, etc.)	
	|	$\nu x{:}b(\overline{e}).\mathtt{S}$	actor creation	
	|	**update**$(\overline{e})$	state update	
	|	**ready**		[runtime syntax]
x	::=	**self** | $x, y, z, \ldots$ | $\alpha, \beta, \ldots$	$x, y, z, \ldots \in$ variables $\alpha, \beta, \ldots \in$ runtime actor names	
R	::=	$(\mathtt{R}_1 + \cdots + \mathtt{R}_k)$	requirement disjunction	
	|	$(\mathtt{R}_1 \cdot \ldots \cdot \mathtt{R}_k)$	requirement conjunction	[runtime syntax]
	|	$(\mathtt{R}_1 \div \mathtt{R}_2)$	requirement satisfaction	[runtime syntax]
	|	h	simple message requirement	
	|	ε	empty requirement	[runtime syntax]

sense to demand that both branches of a conditional satisfy all stated requirements, perhaps via a different messaging path—in this example, via a yes or a no.

As both presented examples hint at, making messaging requirements explicit allows us to reason about the eventual delivery of certain messages—and to do so *statically*. Our approach is to reduce difficult parts of this reasoning to the checking of program conformance to a type system along the lines of process types [17, 30]. If a program passes the check, it is free of the discussed progress issues.

3 Actor Calculus

In this section, we define a minimal actor calculus to formalize the above ideas. Although the language follows standard actor semantics, its syntax does not adopt the λ-calculus extension of Agha et al. [1]; instead, to capture the examples above, we allow behaviors to include message handler definitions. The intention here is the following: consider an actor α with behavior b, where the definition of b includes a handler h with parameters $x_1 \ldots x_k$ and body S. Then, the receipt of a message $h(u_1 \ldots u_k)$ by actor α will invoke the code S, replacing the formal parameters $x_1 \ldots x_k$ with the values $u_1 \ldots u_k$, and **self** with α. The reserved name **self** refers to the actor in which it is evaluated. In what follows, we abbreviate sequences of the form $x_1 \ldots x_k$ with $\overline{x}$, sequences of the form $u_1 \ldots u_k$ with $\overline{u}$, et cetera. The calculus syntax is given in figure 1: programs P consist of a list of behavior definitions $\overline{\mathtt{B}}$ and an initial statement S. An actor behavior definition B includes a name b that identifies the behavior, variables $\overline{x}$ that store the assuming actor's state, and a list of message handler definitions $\overline{\mathtt{H}}$. In turn, a message handler definition H includes a name h that identifies the handler, a list of message parameters $\overline{x}$, and a statement S to be executed upon invocation of the handler.

Statements generally consist of single operations followed by another statement. For example, $\mathtt{x}!h(\overline{e}).\mathtt{S}$ sends a message for handler h of the actor x, with argument list $\overline{e}$, and then proceeds as S. The

Figure 2: Structural congruence on requirements.

$$\varepsilon + \mathtt{R} \equiv \varepsilon \qquad \varepsilon \cdot \mathtt{R} \equiv \mathtt{R}$$

$$\mathtt{R}_1 + \mathtt{R}_2 \equiv \mathtt{R}_2 + \mathtt{R}_1 \qquad \mathtt{R}_1 \cdot \mathtt{R}_2 \equiv \mathtt{R}_2 \cdot \mathtt{R}_1$$

$$\mathtt{R}_1 \cdot (\mathtt{R}_2 \cdot \mathtt{R}_3) \equiv (\mathtt{R}_1 \cdot \mathtt{R}_2) \cdot \mathtt{R}_3 \qquad \mathtt{R}_1 + (\mathtt{R}_2 + \mathtt{R}_3) \equiv (\mathtt{R}_1 + \mathtt{R}_2) + \mathtt{R}_3$$

$$\mathtt{R}_1 \cdot (\mathtt{R}_2 \div \mathtt{R}) \equiv (\mathtt{R}_1 \div \mathtt{R}) \cdot \mathtt{R}_2 \qquad (\mathtt{R}_1 + \mathtt{R}_2) \div \mathtt{R} \equiv (\mathtt{R}_1 \div \mathtt{R}) + (\mathtt{R}_2 \div \mathtt{R})$$

$$\mathtt{R} \div (\mathtt{R}_1 + \mathtt{R}_2) \equiv (\mathtt{R} \div \mathtt{R}_1) + (\mathtt{R} \div \mathtt{R}_2) \qquad \mathtt{R} \div (\mathtt{R}_1 \cdot \mathtt{R}_2) \equiv (\mathtt{R} \div \mathtt{R}_1) \div \mathtt{R}_2$$

Figure 3: Requirement reductions.

$$h \div h \longrightarrow \varepsilon \qquad \frac{h \neq h'}{h \div h' \longrightarrow h}$$

$$\frac{\mathtt{R}_1 \longrightarrow \mathtt{R}_1'}{\mathtt{R}_1 \cdot \mathtt{R}_2 \longrightarrow \mathtt{R}_1' \cdot \mathtt{R}_2} \qquad \frac{\mathtt{R}_1 \longrightarrow \mathtt{R}_1'}{\mathtt{R}_1 + \mathtt{R}_2 \longrightarrow \mathtt{R}_1' + \mathtt{R}_2}$$

$$\mathtt{R}_1 \cdot (\mathtt{R}_2 + \mathtt{R}_3) \longrightarrow \mathtt{R}_1 \cdot \mathtt{R}_2 + \mathtt{R}_1 \cdot \mathtt{R}_3$$

statement $\nu x{:}b(\overline{e}).\mathtt{S}$ creates a new actor (whose name is bound to $\mathtt{x}$ in $\mathtt{S}$) with behavior b and initial state variables set to the values of the expressions $\overline{e}$. The statement $\mathsf{update}(\overline{e})$ updates the values for actor state variables, and the **if** statement has the usual meaning of a conditional. The call $\mathsf{add}(\mathtt{x},\mathtt{R})$ adds the requirement $\mathtt{R}$ to the list of requirements already associated with the actor $\mathtt{x}$. Informally, to satisfy a disjunctive requirement $(h_1 + h_2)$ of $\mathtt{x}$, we have to send it a message for *either one* of h_1, h_2. Similarly, to satisfy a conjunctive requirement $\mathtt{R}_1 \cdot \mathtt{R}_2$, one has to satisfy *both* $\mathtt{R}_1$ and $\mathtt{R}_2$.

3.1 Operational Semantics

Our notation conventions include variants of $\longrightarrow$ for reduction relations, and $\equiv$ for congruences. For $k \geq 0$, we write $\longrightarrow^k$ to denote k-step reductions, and $\longrightarrow^*$ for the relation $\bigcup_{k=0}^{\infty} \longrightarrow^k$. For a reducible element X, we write $X \longrightarrow$ iff there exists X' such that $X \longrightarrow X'$. Additionally, we use $\rightsquigarrow$ for "complete" reductions, i.e., $X \rightsquigarrow X'$ such that $X \longrightarrow^* X'$ and $X' \not\longrightarrow X''$ for all X''. We write $\{x \mapsto \mathtt{R}\}$ for the function that maps x to $\mathtt{R}$, and $R[x \mapsto \mathtt{R}]$ for the result of altering the function R such that it maps x to $\mathtt{R}$. Assuming $dom(R_1) \cap dom(R_2) = \emptyset$, we write $R_1 \cup R_2$ for the mapping for which $(R_1 \cup R_2)(x) = R_i(x)$ when $x \in dom(R_i)$, $i \in \{1,2\}$. We reserve the symbol $\Longrightarrow$ for logical implications.

To formalize the program semantics, we first define an algebra on the extended syntax of requirements (including the empty, conjunction, and satisfaction rules of figure 1). The relation $\equiv$ on requirements is the least congruence relation that includes the rules of figure 2. The empty requirement ε is the zero element for $+$ (disjunction) and the unit element for $\cdot$ (conjunction). Reductions on requirements are defined in figure 3, and hold up to $\equiv$. An empty requirement ε is always considered satisfied, while we say that the messages $h_1, \ldots, h_k$ satisfy a non-empty requirement $\mathtt{R}$ iff $(\ldots(\mathtt{R} \div h_1) \div h_2) \div \cdots) \div h_k) \longrightarrow^* \varepsilon$.

Furthermore, we assume a reduction relation on expressions, such that the notation $e \rightsquigarrow_\Delta u$ means that expression e reduces to the value u, given static program information Δ. The latter is assumed to contain information extracted from the program, such as the parameters of message handlers. The transition relation for statements is defined in figure 4, where we write $\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}'$ to say that a statement $\mathtt{S}$ reduces

Figure 4: Labeled transition semantics for statements. Expressions e follow standard semantics, and Δ is static program information.

$$\frac{\overline{e} \rightsquigarrow_\Delta \overline{u} \qquad l = \alpha!h(\overline{u})}{\alpha!h(\overline{e}).\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}} \qquad \mathsf{add}(\alpha,\mathtt{R}).\mathtt{S} \xrightarrow{\mathsf{add}(\alpha,\mathtt{R})}_\Delta \mathtt{S}$$

$$\frac{\alpha \ \mathit{fresh} \qquad \overline{e} \rightsquigarrow_\Delta \overline{u} \qquad l = \alpha{:}b(\overline{u})}{\nu x{:}b(\overline{e}).\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}[\alpha/x]} \qquad \frac{\overline{e} \rightsquigarrow_\Delta \overline{u} \qquad l = \mathsf{update}(\overline{u})}{\mathsf{update}(\overline{e}) \xrightarrow{l}_\Delta \mathsf{ready}}$$

$$\frac{e \rightsquigarrow_\Delta \mathsf{true}}{\mathsf{if}\ e\ \mathsf{then}\ \mathtt{S}_1\ \mathsf{else}\ \mathtt{S}_2 \xrightarrow{\mathsf{if}}_\Delta \mathtt{S}_1} \qquad \frac{e \rightsquigarrow_\Delta \mathsf{false}}{\mathsf{if}\ e\ \mathsf{then}\ \mathtt{S}_1\ \mathsf{else}\ \mathtt{S}_2 \xrightarrow{\mathsf{if}}_\Delta \mathtt{S}_2}$$

Figure 5: Requirement algebra extended to requirement mappings.

$$(R_1 \cdot R_2)(x) = R_1(x) \cdot R_2(x)$$
$$(R_1 + R_2)(x) = R_1(x) + R_2(x)$$
$$(R_1 \div R_2)(x) = R_1(x) \div R_2(x)$$

$$\frac{\mathtt{R} \longrightarrow \mathtt{R}'}{R \cup \{\mathtt{x} \mapsto \mathtt{R}\} \longrightarrow R \cup \{\mathtt{x} \mapsto \mathtt{R}'\}} \qquad \frac{\mathtt{R} \equiv \mathtt{R}'}{R \cup \{\mathtt{x} \mapsto \mathtt{R}\} \equiv R \cup \{\mathtt{x} \mapsto \mathtt{R}'\}}$$

$$\frac{R \longrightarrow R'}{(\Delta, R, M, A) \longrightarrow (\Delta, R', M, A)} \qquad \frac{R \equiv R'}{(\Delta, R, M, A) \equiv (\Delta, R', M, A)}$$

to $\mathtt{S}'$ via l. The label l records the action being taken; for example, $\alpha!h(\overline{e}).\mathtt{S}$ reduces to $\mathtt{S}$, and $l = \alpha!h(\overline{u})$ records the sent message. The values $\overline{u}$ are computed from the expressions $\overline{e}$, that is, $\overline{e} \rightsquigarrow_\Delta \overline{u}$.

The transition relation $\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}'$ is referenced in the program-level rules of figure 6, which transform runtime configurations. A runtime configuration C is a tuple (Δ, R, M, A), where Δ records static program information; R is a map from actor names to requirements; M is the multi-set of pending (sent, but not received) messages, and A contains running actors. Elements of the set A have the form $\langle \mathtt{S} \rangle_\alpha^{b(\overline{w})}$ where α is the actor's name, b corresponds to its behavior, the values $\overline{w}$ constitute its state, and $\mathtt{S}$ is the statement the actor is currently executing.

We write $C \longrightarrow C'$ to say that the runtime configuration C reduces to C' via an application of some rule in figure 6. By extension, $C \longrightarrow C_1 \longrightarrow \cdots \longrightarrow \cdots$ denotes a possibly infinite sequence of configurations where each adjacent pair follows the transition rules of figure 6. Execution of a program $\mathtt{P} = \overline{\mathtt{B}}\,\mathtt{S}$ then consists of a sequence of transformations that starts from the program's initial configuration. Such an initial configuration is created via rule PROG in figure 6, and it records information Δ from the program, associates no requirements with any actor, has an empty message set, and includes a single initial actor executing $\mathtt{S}$. This actor has reserved name and behavior *in*, and no state variables. When $R = \emptyset$, we define $R(\mathtt{x}) = \varepsilon$ for all $\mathtt{x}$; i.e., by convention, $\emptyset$ maps no requirements to any actor. We assume a straightforward extension of the requirement algebra to runtime configurations, as shown in figure 5. As always, reductions hold up to $\equiv$.

Figure 6: Calculus runtime semantics.

$$\textsc{Send}\ \frac{R(\beta) \div h \rightsquigarrow \mathtt{R}' \qquad R' = R[\beta \mapsto \mathtt{R}'] \qquad \mathtt{S} \xrightarrow{\beta!h(\overline{u})}_{\Delta} \mathtt{S}'}{\Delta, R, M, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta, R', M \cup \{\beta!h(\overline{u})\}, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b(\overline{w})}\}}$$

$$\textsc{Receive}\ \frac{\mathtt{S} = \mathit{body}(\Delta, h) \qquad \overline{y} = \mathit{params}(\Delta, h) \qquad \overline{x} = \mathit{params}(\Delta, b)}{\Delta, R, M \cup \{\alpha!h(\overline{u})\}, A \cup \{\langle \mathsf{ready} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta, R, M, A \cup \{\langle \mathtt{S}[\alpha/\mathsf{self}][\overline{u}\,\overline{w}/\overline{y}\,\overline{x}] \rangle_{\alpha}^{b(\overline{w})}\}}$$

$$\textsc{Update}\ \frac{\mathtt{S} \xrightarrow{\mathsf{update}(\overline{u})}_{\Delta} \mathtt{S}'}{\Delta, R, M, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta, R, M, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b(\overline{u})}\}}$$

$$\textsc{New}\ \frac{\mathtt{S} \xrightarrow{\beta:b(\overline{u})}_{\Delta} \mathtt{S}'}{\Delta, R, M, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b_\alpha(\overline{w})}\} \longrightarrow \Delta, R, M, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b_\alpha(\overline{w})},\ \langle \mathsf{ready} \rangle_{\beta}^{b(\overline{u})}\}}$$

$$\textsc{AddReq}\ \frac{R(\beta) \cdot \mathtt{R} \rightsquigarrow \mathtt{R}' \qquad \mathtt{S} \xrightarrow{\mathsf{add}(\beta, \mathtt{R})}_{\Delta} \mathtt{S}' \qquad R' = R[\beta \mapsto \mathtt{R}']}{\Delta, R, M, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta, R', M, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b(\overline{w})}\}}$$

$$\textsc{If}\ \frac{\mathtt{S} \xrightarrow{\mathsf{if}}_{\Delta} \mathtt{S}'}{\Delta, R, M, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta, R, M, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b(\overline{w})}\}} \qquad \textsc{Prog}\ \frac{\Delta = \mathit{info}(\overline{\mathtt{B}})}{\overline{\mathtt{B}}\,\mathtt{S} \longrightarrow \Delta, \emptyset, \emptyset, \{\langle \mathtt{S} \rangle_{in}^{in()}\}}$$

Rule AddReq deals with calls of the form $\mathsf{add}(\beta, \mathtt{R})$ by appending $\mathtt{R}$ to $R(\beta)$, i.e., the requirements already associated with β. Note the use of $\div$ in Send: the rule adds the sent message to the multi-set of pending messages, and reduces the requirements associated with β by re-mapping it to the result of $R(\beta) \div h$. This corresponds to the fact that β will eventually receive h.

Only idle actors can receive messages [2]. Since statements take the form **ready** when completely reduced, Receive describes an idle actor α receiving a message to be processed by handler h. The statement $\mathtt{S}$ to execute is extracted from the program information Δ, and on it, the rule performs substitution of current values for handler and state variables. These values are taken from the message contents $\overline{u}$ and actor state $\overline{w}$. Handler and behavior (i.e., state) parameters are looked up via the auxiliary function *params*. Rule Update writes new values $\overline{u}$ to the state variables of α. Rule New creates a new actor $\langle \mathsf{ready} \rangle_{\beta}^{b(\overline{u})}$ with unique name β, initialized with the given behavior b and values $\overline{u}$ for state variables. Rule If has the usual effect of deciding a conditional.

4 Type System

The typing rules are given in figure 7. As before, Δ records static program information (such as the abstract syntax tree) which is used to retrieve, for example, the body of message handlers. R maps names to pending requirements, and $\mathtt{S}$ is the program statement being typed. Judgments have the form $R \vdash_{\Delta} \mathtt{S}$, read "under program information Δ and requirement map R, the statement $\mathtt{S}$ is well typed".

Rule T-Prog types programs, and we write $\vdash \mathtt{P}$ to state that the program $\mathtt{P}$ is well-typed. The rule prescribes that $\mathtt{P} = \overline{\mathtt{B}}\,\mathtt{S}$ is well-typed, when, using the information Δ extracted from the behavior definitions $\overline{\mathtt{B}}$, the statement $\mathtt{S}$ is well-typed under the empty requirement map. Rule T-New requires that

Figure 7: Static typing rules.

$$\text{T-PROG}\ \frac{\Delta = \mathit{info}(\overline{\mathtt{B}}) \qquad \emptyset \vdash_\Delta \mathtt{S}}{\vdash \overline{\mathtt{B}}\,\mathtt{S}}$$

$$\text{T-NEW}\ \frac{R \cup \{x' \mapsto \varepsilon\} \vdash_\Delta \mathtt{S}[x'/x] \qquad x' \mathit{\ fresh}}{R \vdash_\Delta \nu x{:}b(\overline{e}).\mathtt{S}} \qquad \text{T-ADD}\ \frac{\mathtt{R}_1 \cdot \mathtt{R} \rightsquigarrow \mathtt{R}' \qquad R \cup \{\mathtt{x} \mapsto \mathtt{R}'\} \vdash_\Delta \mathtt{S}}{R \cup \{\mathtt{x} \mapsto \mathtt{R}_1\} \vdash_\Delta \mathbf{add}(\mathtt{x},\mathtt{R}).\mathtt{S}}$$

$$\text{T-IF}\ \frac{R_1 \vdash_\Delta \mathtt{S}_1 \qquad R_2 \vdash_\Delta \mathtt{S}_2 \qquad R \equiv R_1 + R_2}{R \vdash_\Delta \mathbf{if}\ e\ \mathbf{then}\ \mathtt{S}_1\ \mathbf{else}\ \mathtt{S}_2} \qquad \text{T-UPDATE}\ \frac{\forall \mathtt{x}.(\mathtt{x} \in \mathit{dom}(R) \implies R(\mathtt{x}) \longrightarrow^* \varepsilon)}{R \vdash_\Delta \mathbf{update}(\overline{e})}$$

$$\text{T-SEND}\ \frac{\begin{array}{c}\overline{\mathtt{y}} = \mathit{actors}(\Delta, \overline{e}) \qquad \overline{z} = \mathit{actors}(\Delta, \mathit{params}(\Delta, h)) \\ \mathtt{S}_h = \mathit{body}(\Delta, h) \qquad \{\mathtt{x} \mapsto \mathtt{R}_1\,,\ \overline{\mathtt{y}} \mapsto \overline{\mathtt{R}}_2\} \vdash_\Delta \mathtt{S}_h[\mathtt{x}/\mathbf{self}][\overline{\mathtt{y}}/\overline{z}] \\ R \cup \{\mathtt{x} \mapsto (\mathtt{R}_\mathtt{x} \div (h \cdot \mathtt{R}_1)),\ \overline{\mathtt{y}} \mapsto (\overline{\mathtt{R}}_\mathtt{y} \div \overline{\mathtt{R}}_2)\} \vdash_\Delta \mathtt{S}\end{array}}{R \cup \{\mathtt{x} \mapsto \mathtt{R}_\mathtt{x},\ \overline{\mathtt{y}} \mapsto \overline{\mathtt{R}}_\mathtt{y}\} \vdash_\Delta \mathtt{x}!h(\overline{e}).\mathtt{S}}$$

the statement following the creation command be typed with no requirements associated with the new actor. Rule T-ADD demands that the statement following the $\mathbf{add}(\mathtt{x},\mathtt{R})$ command is well-typed under an environment which includes the new requirements $\mathtt{R}$ for $\mathtt{x}$. Per rule T-IF, typing conditionals requires that each of the two branches satisfies the known requirements. Consistent with the fact that statements end in a construct of the form $\mathbf{update}(\overline{e})$, rule T-UPDATE is the base case of the recursive typing algorithm: it demands that all requirements known in the current scope have been satisfied.

Typing the action of sending a message takes into account that the execution of the related handler may satisfy some requirements known in the current context. Thus, rule T-SEND demands that the statement $\mathtt{S}$ following the send command must be well-typed under a "reduced" requirement map, from which we have removed the sent message h, and the requirements satisfied by the body of h. These include some requirements $\mathtt{R}_1$ associated with $\mathtt{x}$, as well as some requirements $\mathtt{R}_2$ associated with (some of) the arguments $\overline{e}$.

To clarify the use of these rules, consider the example in figure 8 and the respective typing in figure 9. The program's main statement adds the requirement for a message m to y, but does not contain a $y!m()$ statement; rather, it sends $h(y)$ to x. When x receives that message, the requirement for m will be satisfied in the body of the handler, i.e., the statement $z!m().\mathbf{update}()$, with z bound to y. For this reason, when the typing reaches $x!h(y)$, it requires the typing of both the body of h, and the remaining commands—demonstrated by the application of rule T-SEND in figure 9.

Limitations. Our system does not consider a requirement fulfilled, if the necessary messaging happens via state variables. To clarify this limitation, consider the program on the left-hand side of figure 10. It includes two behavior definitions, b_1 and b_2, with one handler each: h_1 in b_1, and h_2 in b_2. Execution starts with the creation of actor x with behavior b_1, and actor y with behavior b_2. Actor x is created with $b_1()$, i.e., no state variables, and y is created with $b_2(x)$, i.e., storing x in the state variable z. The program proceeds to associate the requirement h_1 with x, then sends $h_2()$ to y. When y receives h_2, it will send $h_1()$ to x. However, the presented type system will reject the program, because the typing rules for message sending do not consult with the actor's state. Doing so requires the static tracking of dynamically changing actor state, and is the topic of future research.

Figure 8: Example of requirement delegation.

```
bdef b1() = {
  hdef h(z) = z!m().update()      [sends m to z and returns]
}
bdef b2() = {
  hdef m() = update()             [empty update, does nothing]
}
νx:b1()                           [creates x with behavior b1]
 .νy:b2()                         [creates y with behavior b2]
  .add(y,m)                       [associates requirement for m with y]
   .x!h(y)                        [sends h to x, with y in the payload]
    .update()                     [empty update, does nothing]
```

Figure 9: Typing the example in Figure 8. Δ is static program information per rule T-PROG on page 9.

$$\text{T-SEND}\ \dfrac{\text{T-SEND}\ \dfrac{\{x' \mapsto \varepsilon, y' \mapsto \varepsilon\} \vdash_\Delta \mathsf{update}() \quad \Downarrow\ body(\Delta,m) = \mathsf{update}() \quad \{x' \mapsto \varepsilon, y' \mapsto \varepsilon\} \vdash_\Delta body(\Delta,m)[y'/\mathsf{self}] \qquad \{x' \mapsto \varepsilon, y' \mapsto \varepsilon\} \vdash_\Delta \mathsf{update}() \quad \Downarrow\ (m \div m) \longrightarrow \varepsilon \quad \{x' \mapsto \varepsilon, y' \mapsto (m \div m)\} \vdash_\Delta \mathsf{update}()}{\{x' \mapsto \varepsilon, y' \mapsto m\} \vdash_\Delta y'!m() \,.\, \mathsf{update}()} \qquad \{x' \mapsto \varepsilon, y' \mapsto \varepsilon\} \vdash_\Delta \mathsf{update}() \quad \Downarrow\ (m \div m) \longrightarrow \varepsilon}{\text{T-SEND}\ \dfrac{\{x' \mapsto \varepsilon, y' \mapsto m\} \vdash_\Delta body(\Delta,h)[y'/z] \quad \Downarrow\ body(\Delta,h) = z!m() \,.\, \mathsf{update}() \qquad \{x' \mapsto \varepsilon \div h, y' \mapsto (m \div m)\} \vdash_\Delta \mathsf{update}()}{\text{T-NEW}\ \dfrac{\text{T-NEW}\ \dfrac{\text{T-ADD}\ \dfrac{\{x' \mapsto \varepsilon, y' \mapsto m\} \vdash_\Delta x'!h(y') \,.\, \mathsf{update}()}{\{x' \mapsto \varepsilon, y' \mapsto \varepsilon\} \vdash_\Delta \mathsf{add}(y', m) \,.\, x'!h(y') \,.\, \mathsf{update}()}}{\{x' \mapsto \varepsilon\} \vdash_\Delta \nu y{:}b_2 \,.\, \mathsf{add}(y, m) \,.\, x'!h(y) \,.\, \mathsf{update}()}}{\emptyset \vdash_\Delta \nu x{:}b_1() \,.\, \nu y{:}b_2() \,.\, \mathsf{add}(y, m) \,.\, x!h(y) \,.\, \mathsf{update}()}}$$

The example on the right-hand side of figure 10 shows two actors that exchange messages forever. Actor x sends h_2 to y, which replies with h_1, and so on. Let's see what happens when we attempt to type the body of h_1. First, the system encounters the call $\mathsf{add}(\mathsf{self}, h_1)$, which adds a requirement for h_1 to self, i.e., x. Because the remaining statement is $z!h_2(\mathsf{self}).\mathsf{update}()$, the typing will have to proceed via rule T-SEND. In accordance with the rule premises (page 9), we need to type the body of h_2. In doing so, we will eventually reach the statement $\mathsf{add}(\mathsf{self}, h_2)$, adding a requirement for h_2 to y. The remaining statement is $z!h_1(\mathsf{self}).\mathsf{update}()$, and T-SEND demands the typing of the body of h_1. Attempting to type h_1 restarts the process, entering an infinite sequence of rule applications.

Note that on an intuitive level, this program has the discussed progress property: every generated requirement is eventually satisfied—even though a new one takes its place immediately after. This can be made obvious from the program's reduction sequence, which is shown with minor simplifications in figure 11. Observe that the configurations marked with $(*)$ are identical, and that the part corresponding to the requirement mapping $\{x \mapsto h_1\}$ is tracked by the type system in a manner that mirrors execution—thus failing to terminate. A possible extension to the type system that matches requirement generation sites with future points of requirement satisfaction can mitigate the problem, and is discussed in the work of Charalambides [10].

Figure 10: Untypeable examples.

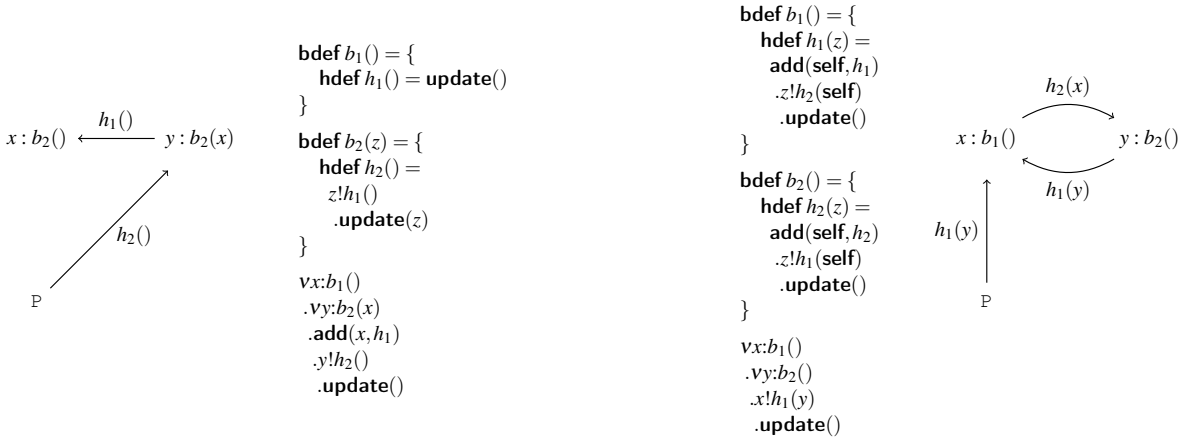

Figure 11: Simplified reduction sequence for the right-hand side program of figure 10.

	$\Delta,$	$\emptyset,$	$\emptyset,$	$\{\langle \nu x{:}b_1().\,\nu y{:}b_2.\,x!h_1(y).\,\mathsf{update}()\rangle_{in}^{in()}\}$	
$\longrightarrow^*$	$\Delta,$	$\emptyset,$	$\emptyset,$	$\{\langle \mathsf{ready}\rangle_x^{b_1()}, \langle \mathsf{ready}\rangle_y^{b_2()}, \langle x!h_1(y).\,\mathsf{update}()\rangle_{in}^{in()}\}$	
$\longrightarrow^*$	$\Delta,$	$\emptyset,$	$\{x!h_1(y)\},$	$\{\langle \mathsf{ready}\rangle_x^{b_1()}, \langle \mathsf{ready}\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	
$\longrightarrow$	$\Delta,$	$\emptyset,$	$\emptyset,$	$\{\langle \mathsf{add}(\mathsf{self},h_1).\,y!h_2(\mathsf{self}).\,\mathsf{update}()\rangle_x^{b_1()}, \langle \mathsf{ready}\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	
$\longrightarrow$	$\Delta,$	$\{x \mapsto h_1\},$	$\emptyset,$	$\{\langle y!h_2(\mathsf{self}).\,\mathsf{update}()\rangle_x^{b_1()}, \langle \mathsf{ready}\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	
$\longrightarrow^*$	$\Delta,$	$\{x \mapsto h_1\},$	$\{y!h_2(x)\},$	$\{\langle \mathsf{ready}\rangle_x^{b_1()}, \langle \mathsf{ready}\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	$(*)$
$\longrightarrow$	$\Delta,$	$\{x \mapsto h_1\},$	$\emptyset,$	$\{\langle \mathsf{ready}\rangle_x^{b_1()}, \langle \mathsf{add}(\mathsf{self},h_2).x!h_1(\mathsf{self}).\,\mathsf{update}()\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	
$\longrightarrow$	$\Delta,$	$\{x \mapsto h_1, y \mapsto h_2\},$	$\emptyset,$	$\{\langle \mathsf{ready}\rangle_x^{b_1()}, \langle x!h_1(\mathsf{self}).\,\mathsf{update}()\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	
$\longrightarrow^*$	$\Delta,$	$\{y \mapsto h_2\},$	$\{x!h_1(y)\},$	$\{\langle \mathsf{ready}\rangle_x^{b_1()}, \langle \mathsf{ready}\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	
$\longrightarrow$	$\Delta,$	$\{y \mapsto h_2\},$	$\emptyset,$	$\{\langle \mathsf{add}(\mathsf{self},h_1).\,y!h_2(\mathsf{self}).\,\mathsf{update}()\rangle_x^{b_1()}, \langle \mathsf{ready}\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	
$\longrightarrow$	$\Delta,$	$\{x \mapsto h_1, y \mapsto h_2\},$	$\emptyset,$	$\{\langle y!h_2(\mathsf{self}).\,\mathsf{update}()\rangle_x^{b_1()}, \langle \mathsf{ready}\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	
$\longrightarrow^*$	$\Delta,$	$\{x \mapsto h_1\},$	$\{y!h_2(x)\},$	$\{\langle \mathsf{ready}\rangle_x^{b_1()}, \langle \mathsf{ready}\rangle_y^{b_2()}, \langle \mathsf{ready}\rangle_{in}^{in()}\}$	$(*)$
				$\vdots$	

5 Calculus Meta-Theory

In order to establish our main result, we extend the typing relation to runtime configurations:

Definition 1 (Runtime Typing)**.** Let C be a runtime configuration. We say that C is well-typed, written $\vdash C$, iff C satisfies the rules shown in figure 12.

Figure 12: Runtime typing rules.

$$\text{R-TRANSITION}\ \frac{C \longrightarrow \quad \forall C'.(C \longrightarrow C' \implies \vdash C')}{\vdash C} \qquad \text{R-READY}\ \frac{\begin{array}{c}\forall \mathtt{x}.(\mathtt{x} \in dom(R) \implies R(\mathtt{x}) \longrightarrow^* \varepsilon) \\ \forall \mathtt{S}, b, \overline{w}, \alpha.(\langle \mathtt{S} \rangle_\alpha^{b(\overline{w})} \in A \implies \mathtt{S} = \mathsf{ready})\end{array}}{\vdash \Delta, R, \emptyset, A}$$

Note rule R-TRANSITION, which is defined with the intention of forcing the runtime typing to unfold program execution—facilitating the proofs of this section. For example, we can show that the typing holds up to structural congruence and requirement reductions (figure 5), captured by the next lemma.

Lemma 1.

$$\frac{C_1 \equiv C_2 \quad \vdash C_1}{\vdash C_2} \qquad \frac{R \longrightarrow R' \quad \vdash (\Delta, R, M, A)}{\vdash (\Delta, R', M, A)}$$

The proof is by induction on the structure of runtime typing derivations, and is omitted. The main result of this paper is that during executions of well-typed programs, all requirements generated dynamically are eventually satisfied; that is, runtime configurations satisfy the *progress* property:

Definition 2 (Progress)**.** Let $C = (\Delta, R, M, A)$ be a runtime configuration. We say that C satisfies the progress property, written $\mathbb{P}(C)$, iff for all executions $C \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_k$ that start from C, it is $C_k = (\Delta, R_k, M_k, A_k)$ with $R_k(\mathtt{x}) \longrightarrow^* \varepsilon$ for all $\mathtt{x} \in dom(R_k)$.

We remind the reader that the initial configuration of a program $\mathtt{P}$ is denoted with $init(\mathtt{P})$, and that $\longrightarrow^*$ is the transitive reflexive closure of the relation $\longrightarrow$. We can now state the main result:

Theorem 1. Let $\mathtt{P}$ be a program. Assuming statements $\mathtt{S}$ terminate, $\vdash \mathtt{P}$ and $init(\mathtt{P}) \longrightarrow^* C$ imply $\mathbb{P}(C)$.

The theorem states that all configurations reachable from the initial configuration of a well-typed program satisfy the progress property, notwithstanding the divergence of expressions (denoted with e in figure 1).

Proof outline. The main idea is to show that

(i) well-typed programs generate well-typed initial configurations, that is, $\vdash \mathtt{P}$ implies $\vdash init(\mathtt{P})$;

(ii) the reduction relation of figure 6 preserves typing, that is, $\vdash C$ and $C \longrightarrow^* C'$ imply $\vdash C'$; and

(iii) well-typed configurations satisfy the progress property, that is, $\vdash C$ implies $\mathbb{P}(C)$.

In other words, we show that the typing of configurations guarantees progress, and that reduction preserves the progress property. We proceed to prove the above items in sequence.

Recalling that satisfying $\mathtt{R}_1 \cdot \mathtt{R}_2$ requires the satisfaction of both $\mathtt{R}_1$ and $\mathtt{R}_2$, we state—without proof—an auxiliary lemma, which can be shown by induction on the structure of runtime typing derivations:

Lemma 2. If $\vdash (\Delta, R_1, M_1, A_1)$ and $\vdash (\Delta, R_2, M_2, A_2)$, then $\vdash (\Delta,\ R_1 \cdot R_2,\ M_1 \cup M_2,\ A_1 \cup A_2)$.

Moreover, the following is an immediate consequence of figures 2 and 3:

Lemma 3. $(\mathtt{R}_1 \div \mathtt{R}_2) \cdot \mathtt{R}_2 \longrightarrow^* \mathtt{R}_1$

The lemma that follows captures our intuition that the static typing of programs (per figure 7) implies that the respective runtime configurations are well-typed (per figure 12).

Lemma 4. Let $\mathtt{S}$ be a statement where **self** has been replaced by a runtime name α, and let A consist solely of actors executing **ready**. Then, for any static program information Δ, requirement map R, behavior instantiation $b(\overline{u})$, and variables $\overline{x}$ with $|\overline{x}| = |\overline{u}|$, we have that $R[\overline{u}/\overline{x}] \vdash_\Delta \mathtt{S}[\overline{u}/\overline{x}]$ implies $\vdash \big(\Delta, R[\overline{u}/\overline{x}], \emptyset, A \cup \{\langle \mathtt{S}[\overline{u}/\overline{x}]\rangle_\alpha^{b(\overline{u})}\}\big)$.

Proof. We proceed by induction on the syntax of statements.

Base case. From figure 1, the base case is that of the **update** call. Let Δ, R, A, α, $\overline{x}$ and $b(\overline{u})$ be as per the statement of the lemma. Moreover, fix values $\overline{w}$ with $|\overline{w}| = |\overline{u}|$. We need to show that

$$R[\overline{u}/\overline{x}] \vdash_\Delta \underbrace{\mathbf{update}(\overline{w})[\overline{u}/\overline{x}]}_{\mathbf{update}(\overline{w})} \text{ implies } \vdash (\Delta,\ R[\overline{u}/\overline{x}],\ \emptyset,\ A \cup \{\underbrace{\langle \mathbf{update}(\overline{w})[\overline{u}/\overline{x}]\rangle}_{\mathbf{update}(\overline{w})}{}_\alpha^{b(\overline{u})}\}).$$

Notice that the variables $\overline{x}$ do not appear in the values $\overline{w}$, and so the substitution $[\overline{u}/\overline{x}]$ leaves $\mathbf{update}(\overline{w})$ unchanged. Assume $R[\overline{u}/\overline{x}] \vdash_\Delta \mathbf{update}(\overline{w})$ per rule T-UPDATE on page 9, i.e.,

$$\forall \mathtt{y}.(\mathtt{y} \in dom(R[\overline{u}/\overline{x}]) \implies R[\overline{u}/\overline{x}](\mathtt{y}) \longrightarrow^* \varepsilon) \tag{1}$$

From (1) and rule R-READY in figure 12, we have that

$$\vdash (\Delta, R[\overline{u}/\overline{x}], \emptyset, A \cup \{\langle \mathbf{ready}\rangle_\alpha^{b(\overline{w})}\}) \tag{2}$$

which, by R-TRANSITION, implies

$$\vdash \big(\Delta, R[\overline{u}/\overline{x}], \emptyset, A \cup \{\langle \mathbf{update}(\overline{w})\rangle_\alpha^{b(\overline{u})}\}\big).$$

Inductive step – message sending. Let Δ, R, A, α, $\overline{x}$ and $b(\overline{u})$ be as per the statement of the lemma. Moreover, fix a message handler h, values $\overline{w}$, an actor name β, a statement $\mathtt{S}$, a behavior instantiation $b'(\overline{u}')$, and variables $\overline{x}'$ with $|\overline{x}'| = |\overline{u}'|$. Assume that $A = A_1 \cup A_2 \cup \{\langle \mathbf{ready}\rangle_\beta^{b'(\overline{u}')}\}$ for some A_1 and A_2 consisting solely of **ready** actors, and that $R = R'[\overline{u}/\overline{x}]$ for some R'. Also, assume that $\mathtt{S} = \mathtt{S}_0[\overline{u}/\overline{x}]$ for some $\mathtt{S}_0$ where **self** has been replaced with α. Further assumptions on $\overline{x}'$ and $\overline{u}'$ will become clear in the next few steps. We need to prove that

$$R \vdash_\Delta \beta!h(\overline{w}).\mathtt{S} \text{ implies } \vdash \big(\Delta, R, \emptyset, A \cup \{\langle \beta!h(\overline{w}).\mathtt{S}\rangle_\alpha^{b(\overline{u})}\}\big).$$

Assume $R \vdash_\Delta \beta!h(\overline{w}).\mathtt{S}$ was derived via an application of rule T-SEND, and thus

$$R \ = \ \underbrace{R_0}_{R_{01}[\overline{u}/\overline{x}]} \cup \ \underbrace{\{\beta \mapsto \mathtt{R}_\beta,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\}}_{R_{02}[\overline{u}/\overline{x}]} \tag{3}$$

for some mapping R_0, requirements $\mathtt{R}_\beta$ and $\overline{\mathtt{R}}_\gamma$, and $\overline{\gamma} = actors(\Delta, \overline{w})$. From T-SEND, it is

$$\underbrace{R_0}_{R_{01}[\overline{u}/\overline{x}]} \cup \ \underbrace{\{\beta \mapsto \mathtt{R}_\beta \div (h \cdot \mathtt{R}_1),\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma \div \overline{\mathtt{R}}_2\}}_{R_{03}[\overline{u}/\overline{x}]} \vdash_\Delta \mathtt{S} \tag{4}$$

$$\text{and} \quad \{\beta \mapsto \mathtt{R}_1,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_2\} \vdash_\Delta \underbrace{body(\Delta, h)[\beta/\mathbf{self}][\overline{\gamma}/\overline{z}]}_{\mathtt{S}'_h = \mathtt{S}_h[\overline{u}'/\overline{x}']} \tag{5}$$

where $\mathtt{R}_1$, $\overline{\mathtt{R}}_2$, $\overline{\gamma}$ and $\overline{z}$ are as in rule T-SEND. From the inductive hypothesis, (4) implies

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta \div (h \cdot \mathtt{R}_1),\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma \div \overline{\mathtt{R}}_2\},\ \emptyset,\ A_1 \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})}\}\big) \tag{6}$$

since A_1 consists solely of **ready** actors. Note that the mapping $\{\beta \mapsto \mathtt{R}_1, \overline{\gamma} \mapsto \overline{\mathtt{R}}_2\}$ in (5) subsumes the substitution $[\overline{u}'/\overline{x}']$. As a result, we can apply the inductive hypothesis on (5) to get

$$\vdash \big(\Delta,\ \{\beta \mapsto \mathtt{R}_1,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_2\},\ \emptyset,\ A_2 \cup \{\langle \mathtt{S}'_h \rangle_\beta^{b'(\overline{u}')}\}\big) \tag{7}$$

because A_2 consists solely of **ready** actors. Combining (6) and (7) per lemma 2, we get

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto (\mathtt{R}_\beta \div (h \cdot \mathtt{R}_1)) \cdot \mathtt{R}_1,\ \overline{\gamma} \mapsto (\overline{\mathtt{R}}_\gamma \div \overline{\mathtt{R}}_2) \cdot \overline{\mathtt{R}}_2\},\ \emptyset,\ A_1 \cup A_2 \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})}, \langle \mathtt{S}'_h \rangle_\beta^{b(\overline{u}')}\}\big) \tag{8}$$

We apply lemmas 1 and 3 to (8) to get

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta \div h,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\},\ \emptyset,\ A_1 \cup A_2 \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})}, \langle \mathtt{S}'_h \rangle_\beta^{b(\overline{u}')}\}\big) \tag{9}$$

We remind the reader that $\overline{u}'$ contains (among others) names in $\overline{w}$, and that $\mathtt{S}'_h$ is the body of handler h with the required substitutions. Thus, by rule R-TRANSITION, (9) implies

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta \div h,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\},\ \{\beta\,!h(\overline{w})\},\ A_1 \cup A_2 \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})}, \langle \mathbf{ready} \rangle_\beta^{b'(\overline{u}')}\}\big).$$

Since $A = A_1 \cup A_2 \cup \{\langle \mathbf{ready} \rangle_\beta^{b'(\overline{u}')}\}$, the above can be written

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta \div h,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\},\ \{\beta\,!h(\overline{w})\},\ A \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})}\}\big).$$

Applying R-TRANSITION again, the above implies

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\},\ \emptyset,\ A \cup \{\langle \beta\,!h(\overline{w}).\mathtt{S} \rangle_\alpha^{b(\overline{u})}\}\big).$$

From (3), the above is the same as

$$\vdash \big(\Delta,\ R,\ \emptyset,\ A \cup \{\langle \beta\,!h(\overline{w}).\mathtt{S} \rangle_\alpha^{b(\overline{u})}\}\big)$$

which completes the proof for message sending. The rest of the cases are simpler, and are thus omitted in the interest of space. □

Corollary 1 (Static Typing Implies Runtime Typing.)**.** $\vdash \mathtt{P}$ implies $\vdash \mathit{init}(\mathtt{P})$ for all programs $\mathtt{P}$.

Proof. Let $\mathtt{P} = \overline{\mathtt{B}}\,\mathtt{S}$ be a program, and assume **self** does not appear in $\mathtt{S}$ (**self** does not make sense in the context of the initial actor). We apply lemma 4 to $\emptyset \vdash_\Delta \mathtt{S}$ and $\vdash (\Delta, \emptyset, \emptyset, \{\langle \mathtt{S} \rangle_{in}^{in()}\})$ with $\Delta = \mathit{info}(\overline{\mathtt{B}})$. □

We now show that the reduction relation of figure 6 preserves typing:

Lemma 5 (Type Preservation)**.** Let C be a runtime configuration. Then $\vdash C$ and $C \longrightarrow^* C'$ imply $\vdash C'$.

Proof. Assume $\vdash C$, which means that one of the rules in figure 12 (page 12) applies. If there exists C' s.t. $C \longrightarrow C'$, then $\vdash C'$ from the definition of typing rule R-TRANSITION. If $C \not\longrightarrow$, i.e., $C \longrightarrow^* C$, the only possibility is that C is a quiescent state, i.e., there are no messages to be delivered, and all actor statements have been reduced to **ready**. Per rule R-READY, C is well-typed. □

Let C be a well-typed runtime configuration. Then a derivation of $\vdash C$ according to the rules of figure 12 forms a tree with root $\vdash C$, such that every path on this tree is a sequence of applications of R-TRANSITION that ends in a single application of R-READY. On each such sequence, we focus on the configurations on the rule conclusions, say $C, C_1 \ldots C_k$. We write $\mathit{Paths}(\vdash C)$ for the set of all such sequences of configurations. As it turns out, $\mathit{Paths}(\vdash C)$ includes all possible executions from configuration C:

Lemma 6 (Typing Unfolds Execution)**.** Let C_1 be a well-typed configuration, i.e., $\vdash C_1$ and $C_1 \longrightarrow \cdots \longrightarrow C_k$ an execution from C_1. Then $(C_1, \ldots, C_k) \in \mathit{Paths}(\vdash C)$.

Proof. Directly from lemma 5. □

Finally, item (iii) from the proof outline is captured by the statement below:

Lemma 7 (Runtime Typing Guarantees Progress)**.** Let C be a configuration. Then $\vdash C$ implies $\mathbb{P}(C)$.

Proof. A derivation of $\vdash C$ follows the rules of figure 12, and hence, such a derivation ends in an application of rule R-READY. Thus, every sequence in $\mathit{Paths}(\vdash C)$ ends in some configuration C_k for which $\vdash C_k$ is given by rule R-READY. From the definition of the rule, C_k must be a quiescent state with no requirements. By lemma 6 and the fact that $\vdash C$, every execution from C ends in such a state. □

We are now ready to prove the main result:

Proof (theorem 1). Direct consequence of

$\vdash \mathtt{P}$ implies $\vdash \mathit{init}(\mathtt{P})$	(corollary 1)
$\vdash C$ and $C \longrightarrow^* C'$ implies $\vdash C'$	(lemma 5)
$\vdash C$ implies $\mathbb{P}(C)$	(lemma 7).

□

6 Related Work

In mainstream programming languages, one expects that the arguments passed to a procedure are compatible with the way they are used in the procedure's body. Pierce and Sangiorgi [28] extended this idea to Milner's π-calculus [26], ensuring the proper use of communicable values. Their system guarantees that a value sent from one process to another can be safely used at the receiving end, including when the sent value is a channel name. Through a suitable subtyping relation, Pierce and Sangiorgi guarantee that after communicating the name of a channel, subsequent interactions through that channel recursively satisfy this same (safety) property.

Honda's early work [17] took a different approach, focusing on the typing of concurrent processes instead of channels. His system views types as descriptions of the sequencing of communication actions, and demands that composed processes have *dual* types. Duality corresponds to the sequencing of matching send and receive actions, ensuring safety and deadlock-freedom in two-party interactions. The idea was later applied to communication over π-calculus channels by Takeuchi et al. [35], an approach in turn refined by Honda et al. [18] and later by Yoshida and Vasconcelos [39].

These early works have been captured in practice by WSCDL, i.e., the Web Services Choreography Description Language [37]. In this language, one gives specifications of protocols involving multiple concurrent web services and clients, in XML form [36]. Its original purpose was to ensure standard safety

properties: clients only use services that exist, and communicated data is well-formed. Nevertheless, WSCDL has served as the starting point for the treatment of protocol types from a global perspective. For example, Carbone et al. [7] introduced the notion of a *projection*—the restriction of a globally specified protocol onto the individual participants, assigning a type to each concurrent entity in the system. Then, type-checking can be performed on a per process basis, rather than having to look at the protocol as a whole. This idea was extended by Honda et al. [19], who allowed sessions to include more than two participants. In that and other similar systems, a *global type* describes the session protocol, and a projection algorithm mechanically derives end-point types for the individual processes—describing how each process uses the channels known to them. Honda's system ensures progress on a per session basis, under the condition that communication within a session is not hindered by actions in a different session, an assumption also made by Dezani-Ciancaglini et al. [15].

Subtyping for Honda's system was considered by Gay and Hole [16], but without further progress guarantees. The assumption on the absence of inter-session hindrance was first lifted by Bettini et al. [5], who analyzed the flow of dependencies on the use of channels, tracking the sending of channel names and preventing cyclic dependencies. These systems have since been improved on to account for parameterized participant numbers, and to capture more complex protocols [11, 12, 14, 38]. A different direction was taken by Carbone and Montesi [8], who allowed the concurrent program itself to be written from a global perspective, while statically ensuring the absence of deadlocks via a suitable type system.

Much like the present paper, some authors observed that deadlock-freedom does not necessarily guarantee progress in the intuitive sense, and have considered more general notions [6]. For instance, Kobayashi [21] identifies three important classes of channel usage, and guarantees a notion of progress for programs that use such channels. In general, Kobayashi's typing associates a time tag with each channel, inferred from the relative order of actions in which the channel is involved. In order to disable cycles, the type system then enforces an ordering relation on these tags. The use of a partial order to break cyclic dependencies is also found in the work of Padovani [27].

Sumii and Kobayashi [34] take these ideas further with the explicit inclusion of programmer intent in the code, so that channels are annotated with capabilities and obligations. The resulting type system ensures that if a process has the capability of performing an I/O action on a channel, that action will eventually succeed; similarly, if a process has the obligation to perform an action on a channel, the action is eventually taken. This strategy for deadlock-freedom is improved on in Kobayashi's later work [22], where even more precise information is used: channels are additionally associated with the minimum number of reduction steps needed until capabilities are met, and also the maximum number of steps required until obligations are fulfilled. The automatic inference of similar type annotations has also been considered [23].

Related to our approach is the work of Puntigam and Peter [30, 31], who allow actors to produce and consume tokens as a reaction to message receipt. Their typing only allows an actor α to be sent a message m when the handler for m does not consume more tokens than α has. The type system keeps track of the tokens known in each scope; the effect is that, after a send command, actors update their knowledge of the tokens associated with the recipient. Puntigam and Peter distinguish among optional and *obligatory* tokens, such that if an actor is aware of the existence of obligatory tokens in another actor, they need to ensure that these tokens are consumed—by sending those actors suitable messages. Puntigam and Peter's system focuses on breaking cycles of requirements, and it does so by imposing a partial order on obligatory tokens. A salient drawback of their strategy is that well-typed programs still allow passing obligations around in circles, never sending the required messages. It is worth mentioning that our system does not suffer from this drawback, while in fact employing a simpler typing strategy.

7 Conclusions and Future Work

We presented a type-based approach to ensuring progress in actor systems, based on allowing the programmer to state requirements on messages that an actor must eventually receive at runtime. To demonstrate the practical usefulness of our approach, we showed that such requirements can be naturally expressed in the classic example scenarios of resource sharing and book selling. We formalized the idea as a type system for a simple language of stateful actors that communicate via asynchronous message passing, and proved that executions of well-typed actor programs will eventually fulfill all requirements that appear at runtime. Our approach permits some cases of indirect satisfaction of requirements, i.e., actors delegating obligations to one another, which is typical in actor systems.

We expect that similar type systems and constructs such as `add_req` can be straightforwardly implemented in practical actor languages. However, as is common of decidable type systems capturing powerful properties in full-featured languages, such implementations will necessarily be incomplete—in the sense that there will be programs that fulfill all requirements, but where type checking cannot attest to this fact. Our system is no different in this aspect, and two relevant examples were presented in section 4. The typing presented here works assuming no cyclic messaging patterns, which is the reason we do not require *fairness* [3]. Fair executions are those which disallow the indefinite postponement of basic operations such as the dispatching of messages; in absence of cyclic communication patterns, such indefinite postponement is not possible.

The restriction on cyclic communication patterns can be lifted to a great extent by considering each requirement generation site separately. This way, it is possible to remember each such site and avoid visiting it twice. To ensure our progress property, it is sufficient to match each requirement generation command with message sending commands guaranteed to execute later. Such an extension is proposed in the work of Charalambides [10], and it works under standard fairness assumptions.

Our type system is able to help programmers find simple but critical errors, such as the omission of a key message sending operation. However, in its present form, the type system does not deal with safety properties, even elementary ones such as ensuring that a message handler receives the correct number of arguments. We opted to not clutter the presentation with additional rules, to focus solely on issues of progress. It is nonetheless clear that such safety checks can be easily incorporated to this work as an additional type system, on top of ours. One can envision superimposing a number of more complex systems; one example is Puntigam's token system [29] which ensures that an actor has sufficient tokens to process the messages sent to it. Combining the two approaches would allow complicated coordination constraints to be statically expressed and enforced.

Acknowledgments. This work was supported in part by the National Science Foundation under grants NSF CCF 14-38982 and NSF CCF 16-17401.

References

[1] Gul Agha, Ian A. Mason, Scott F. Smith & Carolyn L. Talcott (1997): *A Foundation for Actor Computation*. *J. Funct. Program.* 7(1), pp. 1–72, doi:10.1017/S095679689700261X.

[2] Gul Agha & Prasanna Thati (2004): *An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language*. In Olaf Owe, Stein Krogdahl & Tom Lyche, editors: *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, *Lecture Notes in Computer Science* 2635, Springer, pp. 26–57, doi:10.1007/978-3-540-39993-3_4.

[3] Gul A. Agha (1990): *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence, MIT Press.

[4] Bowen Alpern & Fred B. Schneider (1985): *Defining Liveness*. *Inf. Process. Lett.* 21(4), pp. 181–185, doi:10.1016/0020-0190(85)90056-0.

[5] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2008): *Global Progress in Dynamically Interleaved Multiparty Sessions*. In Franck van Breugel & Marsha Chechik, editors: *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, *Lecture Notes in Computer Science* 5201, Springer, pp. 418–433, doi:10.1007/978-3-540-85361-9_33.

[6] Marco Carbone, Ornela Dardha & Fabrizio Montesi (2014): *Progress as Compositional Lock-Freedom*. In eva Kühn & Rosario Pugliese, editors: *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, *Lecture Notes in Computer Science* 8459, Springer, pp. 49–64, doi:10.1007/978-3-662-43376-8_4.

[7] Marco Carbone, Kohei Honda & Nobuko Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In Rocco De Nicola, editor: *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, *Lecture Notes in Computer Science* 4421, Springer, pp. 2–17, doi:10.1007/978-3-540-71316-6_2.

[8] Marco Carbone & Fabrizio Montesi (2013): *Deadlock-freedom-by-design: multiparty asynchronous global programming*. In Roberto Giacobazzi & Radhia Cousot, editors: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, ACM, pp. 263–274, doi:10.1145/2429069.2429101.

[9] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini & Luca Padovani (2012): *On Global Types and Multi-Party Session*. *Logical Methods in Computer Science* 8(1), doi:10.2168/LMCS-8(1:24)2012.

[10] Minas Charalambides (2018): *Actor Programming with Static Guarantees*. Doctoral dissertation, University of Illinois at Urbana-Champaign. Available at `http://hdl.handle.net/2142/101036`.

[11] Minas Charalambides, Peter Dinges & Gul Agha (2012): *Parameterized Concurrent Multi-Party Session Types*. In Natallia Kokash & António Ravara, editors: *Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012.*, *EPTCS* 91, pp. 16–30, doi:10.4204/EPTCS.91.2.

[12] Minas Charalambides, Peter Dinges & Gul A. Agha (2016): *Parameterized, concurrent session types for asynchronous multi-actor interactions*. *Science of Computer Programming* 115-116, pp. 100–126, doi:10.1016/j.scico.2015.10.006.

[13] Minas Charalambides, Karl Palmskog & Gul A. Agha (2019): *Types for Progress in Actor Programs*. *978-3-030-21484-5, De Nicola-Festschrift, LNCS* 11665.

[14] Pierre-Malo Deniélou & Nobuko Yoshida (2011): *Dynamic multirole session types*. In Thomas Ball & Mooly Sagiv, editors: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, ACM, pp. 435–446, doi:10.1145/1926385.1926435.

[15] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida & Sophia Drossopoulou (2006): *Session Types for Object-Oriented Languages*. In Dave Thomas, editor: *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, *Lecture Notes in Computer Science* 4067, Springer, pp. 328–352, doi:10.1007/11785477_20.

[16] Simon J. Gay & Malcolm Hole (2005): *Subtyping for session types in the pi calculus*. *Acta Inf.* 42(2-3), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[17] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, *Lecture Notes in Computer Science* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[18] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, *Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[19] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In George C. Necula & Philip Wadler, editors: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, pp. 273–284, doi:10.1145/1328438.1328472.

[20] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[21] Naoki Kobayashi (1998): *A Partially Deadlock-Free Typed Process Calculus*. *ACM Trans. Program. Lang. Syst.* 20(2), pp. 436–482, doi:10.1145/276393.278524.

[22] Naoki Kobayashi (2002): *A Type System for Lock-Free Processes*. *Information and Computation* 177(2), pp. 122–159, doi:10.1006/inco.2002.3171.

[23] Naoki Kobayashi, Shin Saito & Eijiro Sumii (2000): *An Implicitly-Typed Deadlock-Free Process Calculus*. In Catuscia Palamidessi, editor: *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, *Lecture Notes in Computer Science* 1877, Springer, pp. 489–503, doi:10.1007/3-540-44618-4_35.

[24] Leslie Lamport (1977): *Proving the Correctness of Multiprocess Programs*. *IEEE Trans. Software Eng.* 3(2), pp. 125–143, doi:10.1109/TSE.1977.229904.

[25] Lightbend: *Akka*. http://akka.io.

[26] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I*. *Inf. Comput.* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[27] Luca Padovani (2013): *From Lock Freedom to Progress Using Session Types*. In Nobuko Yoshida & Wim Vanderbauwhede, editors: *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013.*, *EPTCS* 137, pp. 3–19, doi:10.4204/EPTCS.137.2.

[28] Benjamin C. Pierce & Davide Sangiorgi (1996): *Typing and Subtyping for Mobile Processes*. *Mathematical Structures in Computer Science* 6(5), pp. 409–453.

[29] Franz Puntigam (1997): *Coordination Requirements Expressed in Types for Active Objects*. In Mehmet Aksit & Satoshi Matsuoka, editors: *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, *Lecture Notes in Computer Science* 1241, Springer, pp. 367–388, doi:10.1007/BFb0053387.

[30] Franz Puntigam (2000): *Concurrent Object-Oriented Programming with Process Types*. Habilitationsschrift. Der Andere Verlag, Osnabrück, Germany.

[31] Franz Puntigam & Christof Peter (2001): *Types for Active Objects with Static Deadlock Prevention*. *Fundam. Inform.* 48(4), pp. 315–341. Available at `http://content.iospress.com/articles/fundamenta-informaticae/fi48-4-02`.

[32] *The Scala Programming Language*. https://www.scala-lang.org/.

[33] Robert E. Strom & Shaula Yemini (1986): *Typestate: A Programming Language Concept for Enhancing Software Reliability*. *IEEE Trans. Software Eng.* 12(1), pp. 157–171, doi:10.1109/TSE.1986.6312929.

[34] Eijiro Sumii & Naoki Kobayashi (1998): *A Generalized Deadlock-Free Process Calculus*. *Electr. Notes Theor. Comput. Sci.* 16(3), pp. 225–247, doi:10.1016/S1571-0661(04)00144-6.

[35] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou & Sergios Theodoridis, editors: *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, *Lecture Notes in Computer Science* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[36] W3C: *Extensible Markup Language*. https://www.w3.org/XML/.

[37] W3C (2005): *The Web Services Choreography Description Language*. http://www.w3.org/TR/ws-cdl-10/.

[38] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri & Raymond Hu (2010): *Parameterised Multiparty Session Types*. In C.-H. Luke Ong, editor: *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, *Lecture Notes in Computer Science* 6014, Springer, pp. 128–145, doi:10.1007/978-3-642-12032-9_10.

[39] Nobuko Yoshida & Vasco Thudichum Vasconcelos (2007): *Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication*. *Electr. Notes Theor. Comput. Sci.* 171(4), pp. 73–93, doi:10.1016/j.entcs.2007.02.056.