Session Typing for a Featherweight Erlang

Dimitris Mostrous and Vasco T. Vasconcelos

LaSIGE, Faculty of Sciences, University of Lisbon

Abstract. As software tends to be increasingly concurrent, the paradigm of message passing is becoming more prominent in computing. The language Erlang offers an intuitive and industry-tested implementation of process-oriented programming, combining pattern-matching with message mailboxes, resulting in concise, elegant programs. However, it lacks a successful static verification mechanism that ensures safety and determinism of communications with respect to well-defined specifications. We present a session typing system for a featherweight Erlang calculus that encompasses the main communication abilities of the language. In this system, structured types are used to govern the interaction of Erlang processes, ensuring that their behaviour is safe with respect to a defined protocol. The expected properties of subject reduction and type safety are established.

1 Introduction

In the age of web services, distributed systems and multicore processors, the paradigm of message passing is becoming increasingly prominent in computing. The functional-declarative language Erlang is widely used for process-oriented software, utilising pattern-matching to extract messages from mailboxes, and resulting in concise, elegant programs. However, it lacks a static verification mechanism that can ensure safety and determinism of communications with respect to well-defined protocol specifications. Such verification is highly useful but also very challenging, since the language is dynamically typed, and any type system has to work on top of the existing semantics of its communication primitives.

In this work we present the first typing system for the concurrent fragment of Erlang, based on session types, and distilled in a featherweight calculus. To overcome the uncontrolled nature of process identifiers, which address the unique mailbox owned by each process (thread), we make extensive use of the ability of the language to generate unique references (fresh names), created with the built-in function **make_ref()**. By carefully controlling the use of references, and by including them in messages where they play the role of uniquely identifying (correlating) conversations, we can guarantee properties about the fine-grained structure of communications between pairs of processes. For example we ensure that messages are always of the expected type and that sending and receiving follows a prescribed pattern respected by both sides.

The programming style required in our methodology may seem cumbersome for simple programs with protocols consisting of a single message exchange, but

without references it is difficult to *ensure* message correlation even for simple request–response: using just process identity (e.g., the unique mailbox of the sender) is not enough, as any process can "impersonate" another just by knowing its identity which it can attach to a message [2]. Thus, **make_ref()** seems to be the only means to "get concurrency right." Yet, an ad-hoc use of **make_ref()** may lead to applications that suffer from interference, race conditions, or even that fail from delivering the expected results. Our system provides for a methodology that governs its use, while statically guaranteeing that programs behave according to the plan. We have only addressed a tiny part of the language, a language that is untyped in nature. Scaling our proposal to a larger subset of Erlang constitutes an interesting challenge. Moreover, our contribution can be viewed also as a type system for an important pattern of concurrent behavior, a pattern that goes well beyond what conventional session types currently allow, while presenting ideas that may be incorporated in future message passing, buffered, concurrent languages where receivers may inspect a mailbox picking appropriate messages.

Core Erlang [4], a canonical format for Erlang programs, is used internally by the Erlang compiler, and also by many verification tools, most notably Dialyzer which is part of the Erlang distribution. Dialyzer detects errors by infering types based on Success Typings [9]. However, until now the type-based methods developed for Erlang focus entirely on the functional part of the language, and are therefore irrelevant in verifying the properties of concurrent message-passing programs.

More recently, in [5], an analysis method was implemented that can statically detect definite communication errors in Erlang programs, based on a topological synthesis of communication primitive usages. Such properties include the case where sent messages cannot be matched by a receive, however, it has a different approach than ours: it does not check programs against types, but rather analyses them against each other, detecting undesirable compositions of sending and receiving. On the other hand, this method is automated and has been implemented on top of the Dialyzer tool.

The rest of the paper is structured as follows. The next section presents our language via an example. Then, Section 3 formally introduces the syntax and reduction semantics of the language. Section 4 presents the type assignment system and its main results. Section 5 concludes the paper.

2 A motivating example

Consider the classical readers-writer problem. A given resource can be written by (exactly) one writer when no readers are reading; it can be simultaneously read by a bounded number or readers while no writer is writing. A controller protecting accesses to such a resource provides for two distinct operations (or services): *read* and *write*. Given the constraints enumerated above, each of these *services* is associated with a little *protocol*.

Upon invoking service *write*, writers receive one of two messages: *welcome* meaning that no reader is reading, or *reading* meaning there is at least one

reader reading. In the first case, the protocol terminates (the writer may try later, perhaps in a busy waiting manner); in the second case, the writer must *store* its data and the protocol terminates. Readers, on the other hand, invoke service *read*. Three things can happen: the reader is allowed in, there is one writer writing, or the bound on the number of readers was exceeded. In the first case, the reader receives a message *welcome*, after which it must *store* its data and the protocol terminates. In the two other cases, the protocol terminates after the reception of a *writing* or a *full* message.

The services and their associated protocols are captured by simple type abstractions. To a resource we associate a record type describing the two services:

{*write*: Write, *read*: Read}

Each service is described by a *session type*. Session type Write is of the form:

⊕**[***welcome*: &**[***load*: **end]**, *reading*: **end]**

where type operator ⊕ means that the resource sends one of the two messages *welcome* or *reading*, and operator & says that the resource accepts message *load*. Type constructor **end** denotes the conclusion of the session. The session type Read is similar, only that it starts with three options.

⊕**[***welcome*: &**[***store*: **end]**, *writing*: **end**, *full*: **end]**

We write the code for the resource monitor in an Erlang-like language. When idle the monitor accepts any of the service requests, answers *welcome* in both cases and proceeds appropriately. We could try writing our code as follows,

```
idle () = receive {write,Writer} →Writer!{welcome}, ...
                  {read,Reader} →Reader!{welcome}, ...
```

Messages are selected from the monitor's mailbox by a pattern matching mechanism. A pattern of the form {*write*,writer} matches an arbitrary message composed of a label (an *atom* in the Erlang jargon) *write* and any value (the process identifier—pid in short—of the writer) that becomes associated to variable Writer. Term Writer**!**{*welcome*} sends a message {*welcome*} to the Writer's mailbox.

Each interaction with the monitor is composed of a series (of two or three) messages; we call a *session* the sequence of messages that pertain to the same run of some protocol. When a monitor interacts with different clients, the client's pid is enough to distinguish to which session messages belong. For more elaborate scenarios, where the same client constitutes two or more readers or writers, we must resort to more complex protocols. A common method used to distinguish different sessions running simultaneously, is to use *correlation sets* [3, 10].

A correlation set is a set of identifiers (*references* in the Erlang jargon) that uniquely identifies a session. Clients create the required references and send them in the service invocation message. For each session we need two correlation references, one for the sending operations, the other for receiving. So here is the revised version of the monitor, noting that ',' denotes sequencing and ';' separates alternative receive clauses, with '.' marking the end:

```
idle () = receive {write,X,Y,Writer} →Writer!{welcome,Y}, write(X);
                  {read,X,Y,Reader} →Reader!{welcome,Y}, readOne(X).
```

In the first line the monitor receives a message with two references and uses the second, Y, for letting the writer know to which session does the *welcome* message belong to. The writer, in turn, uses the first reference, X, to 'sign' the subsequent messages in the session. In the write phase, the monitor may accept messages from the just initiated session (we omit the actual data to be stored at the resource).

```
write(X) = receive {store,X} →idle().
```

During this phase, readers invoking the *read* service would block waiting for the server to go back to the idle state. Our language allows for more than this: the server may as well answer immediately to clients (with a *writing* message), while waiting from the writer's *store* message. That is, our server is able to initiate new services while running other services.

```
write(X) = receive {store,X} →idle();
                   {read,_,Z,Reader} →Reader![writing,Z], write(X).
```

The code for the read phase should by now be easy to understand; for simplicity we allow two simultaneous readers, max. And we never leave a client without an answer.

```
readOne(X1) = receive {load,X1} →idle();
                      {write,_,Z,Writer} →Writer![reading,Z], readOne(X);
                      {read,X2,Y2,Reader2} →Reader2![welcome,Y2], readTwo(X1,X2).
readTwo(X1,X2) = receive {load,X1} →readOne(X2);
                         {load,X2} →readOne(X1);
                         {write,_,Z,Writer} →Writer![reading,Z], readTwo(X1,X2);
                         {read,_,Z,Reader} →Reader![full,Z]; readTwo(X1,X2).
```

In the readTwo phase we decided to honor all possible cases: continuing with the two open sessions with both readers, opening new sessions with new readers and writers. But that need not be the case, at any moment programmers may choose which sessions to continue and which new service requests to accept.

To complete our example we write the code for a reader that tries to store at the resource (and gives up if unable).

```
reader() = make_ref X,Y for self, Resource in
           Resource![read,X,Y,self],
           receive {welcome, Y} →Resource![load,X};
                   {writing, Y} →;
                   {full, Y} → .
```

For convenience, we create pairs of fresh references in one step with a **make_ref** operation. The thus created references, X and Y, must be bound to the pid of the processes that will engage in interaction. The monitor, with pid Resource, is going to use X for reading and Y for writing. Symmetrically, the current writer (with pid **self**) will use Y for reading and X for writing.

What guarantees do we obtain from our type system? To discuss this matter we must remember that, in Erlang, message sending is non-blocking and that messages may be retrieved from the mailbox in any order (as opposed to,

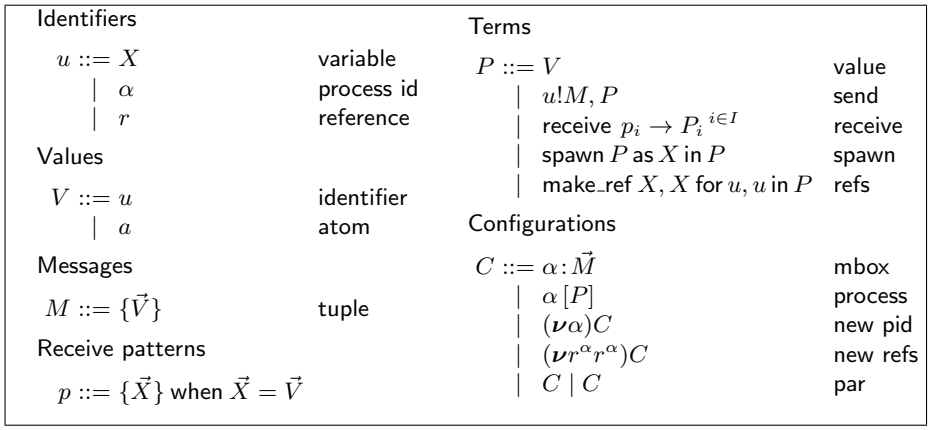

Fig. 1. Syntax

say, first-in first-out). The guarantee that Erlang processes engage in protocols as specified by the session types—commonly known as *session fidelity*—is captured in our setting by inspecting mailboxes at termination. In the case of the Reader above, the type system guarantees that the reader did not receive (during its short life) unexpected messages from the server that remain unseen in the mailbox. The same can be said of the monitor: at termination (if this ever happens) no unexpected message remains in the mailbox.

3 Featherweight Erlang

This section presents our language, its syntax and reduction semantics.

For the programmers' language we rely on a (countable) *set of variables*; we use upper-case letters X and Y to range over variables, following the Erlang conventions. A distinguished variable, self, plays a special role in the semantics. We also need a set of non-interpreted *atoms* (or labels), ranged over by lower-case letter a. The syntax of the language is defined in Figure 1. The *identifiers* in the programmer's syntax are variables only (the remaining two alternatives are described below). Values V of the programmers' language are simply variables or atoms. The messages exchanged by processes, M, are tuples of values.

A program is a (closed) *term* that uses for identifiers variables only. The constructors of terms include values as well as primitives to send and to receive messages, to spawn new processes and to create new unique references. A term of the form $u!M, P$ sends message M to the process named u and continues as P. A term of the form receive $p_i \to P_i\ ^{i \in I}$ attempts to pattern-match a message from the mailbox against the various patterns p_i and continues with the term P_j for which the matching succeeds (patterns and pattern matching are described below), blocking if no message matches. A term spawn P as X in Q creates a new process with running code P, binds the (newly created) process

identifier to variable X and continues with term Q. Finally, a term of the form $\mathsf{make\_ref}\ X, Y\ \mathsf{for}\ u, v\ \mathsf{in}\ P$ creates two unique references, binds them to variables X and Y, associates them to process identifiers u and v, and continues with term P. A simple form of terms allowing the description of unbounded behaviour, e.g. $\mathsf{def}\ A\vec{X} = P\ \mathsf{in}\ P$ and $A\vec{V}$, can be easily incorporated in our language, following, e.g., [7, 12]. For the sake of simplicity, and in order to concentrate on the novel aspects of our system, we decided not to include them.

For the *runtime language* we need two new classes of identifiers: *process identifiers* (pid's) denoted by α and *unique references* denoted by r. The syntax of terms remains unchanged, except for the extended category of identifiers. Terms do not engage in reduction per se. Instead they must be uploaded into a configuration. *Configurations* are built from five different constructors. A term of the form $\alpha : \vec{M}$ describes a *mailbox* for the process with pid α, containing a list of (unread) messages $\vec{M}$; a process $\alpha\,[P]$ is a term P located at pid α. Then we have scope restriction operators, $(\boldsymbol{\nu}\alpha)C$ for process identifiers, and $(\boldsymbol{\nu} r_1^{\alpha_1} r_2^{\alpha_2})C$ for pairs of references. Finally, configurations of the form $C_1 \mid C_2$ allow C_1 and C_2 to run in parallel.

We count with three *binders* for terms and two for configurations. They are: the variables $\vec{X}$ in a receive pattern $\{\vec{X}\}\ \mathsf{when}\ \vec{Y} = \vec{u}$, variable X in a spawn term $\mathsf{spawn}\ P\ \mathsf{as}\ X\ \mathsf{in}\ Q$, variables X_1 and X_2 (but not u_1 and u_2) in a reference creation term $\mathsf{make\_ref}\ X_1, X_2\ \mathsf{for}\ u_1, u_2\ \mathsf{in}\ P$, process identifier α in configuration $(\boldsymbol{\nu}\alpha)C$, and references r_1 and r_2 (but not α_1 and α_2) in configuration $(\boldsymbol{\nu} r_1^{\alpha_1} r_2^{\alpha_2})C$. In order to simplify the subsequent presentation we use letter n for any of the binders α or $r_1^{\alpha_1} r_2^{\alpha_2}$. The sets of free variables and bound variables are defined accordingly. We follow Barendregt's variable convention, requiring bound identifiers to be distinct from free identifiers in any mathematical context. A *substitution* is a map (finite, partial domain) from variables into values, written $\{\vec{V}/\vec{X}\}$ and ranged over by σ. The (capture free) operation of applying a substitution to term P, denoted $P\sigma$, is standard.

If P is a program (a closed term), we *upload* P at our machine by building a configuration of the form

$$(\boldsymbol{\nu}\alpha)(\alpha\,[P\{^{\alpha}/_{\mathsf{self}}\}] \mid \alpha : \varepsilon)$$

composed of program P located at process identifier α, and empty mailbox for the same pid (ε denotes the empty sequence). The distinguished nature of variable self is apparent in $P\{^{\alpha}/_{\mathsf{self}}\}$: process P may refer to its own pid via self, which at runtime is replaced by the actual value α.

Structural congruence is the smallest relation on processes including the rules in Figure 2. The first two rules say that parallel composition is commutative and associative. The rules in the second line deal with scope restriction. The first, scope extrusion, allows the scope of n to encompass C_2; due to the variable convention, n bound in $(\boldsymbol{\nu} n_2)C_1$, cannot be free in C_2. The other two rules allow exchanging the order of restrictions.[1]

[1] Notice that $(\boldsymbol{\nu} r_1^{\alpha_1} r_2^{\alpha_2})(\boldsymbol{\nu}\alpha_1)C \not\equiv (\boldsymbol{\nu}\alpha_1)(\boldsymbol{\nu} r_1^{\alpha_1} r_2^{\alpha_2})C$ due to the variable convention (the left-hand side configuration is not well formed).

$$C_1 \mid C_2 \equiv C_2 \mid C_1 \qquad (C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3)$$
$$(\boldsymbol{\nu} n) C_1 \mid C_2 \equiv (\boldsymbol{\nu} n)(C_1 \mid C_2) \qquad (\boldsymbol{\nu} n_1)(\boldsymbol{\nu} n_2) C_1 \equiv (\boldsymbol{\nu} n_2)(\boldsymbol{\nu} n_1) C_1$$

Fig. 2. Structure Congruence

$$\begin{aligned}
\mathsf{match}(\{\vec{X}\}\ \mathsf{when}\ \vec{Y} = \vec{U}, \{\vec{V}\}) &= \mathsf{match}_{\vec{Y}=\vec{U}}(\vec{X}, \vec{V}) \\
\mathsf{match}_{\vec{Y}=\vec{U}}(X\vec{X}, V\vec{V}) &= \{V/X\} \cup \mathsf{match}_{\vec{Y}=\vec{U}}(\vec{X}, \vec{V}) \qquad \text{if } X \notin \vec{Y} \\
\mathsf{match}_{\vec{Y}_1 X \vec{Y}_2 = \vec{U}_1 V \vec{U}_2}(X\vec{X}, V\vec{V}) &= \{V/X\} \cup \mathsf{match}_{\vec{Y}_1 X \vec{Y}_2 = \vec{U}_1 V \vec{U}_2}(\vec{X}, \vec{V}) \\
\mathsf{match}_{_=_}(\varepsilon, \varepsilon) &= \emptyset
\end{aligned}$$

Fig. 3. Pattern Matching

Messages are read from a mailbox via a *pattern matching* mechanism. In order to simplify the definitions (type system included), patterns $\{\vec{X}\}$ when $\vec{Y} = \vec{V}$ introduce as many variables $\vec{X}$ as the length of the tuple expected. The actual matching is then performed on the $\vec{Y} = \vec{V}$ part. The definition is in Figure 3. If defined, the output of the matching function may then be applied to a term. In examples we often elide the when clause, by using atoms as well as previously introduced variables in patterns. The code for write presented previously

```
write(X) = receive {store,X} →idle().
```

must be understood as

```
write(X) = receive {Y,Z} when Y,Z=store,X →idle().
```

Reduction is the smallest relation on processes that includes the rules in Figure 4. Rule send places message M in the mailbox of the target process α_2, while the sender continues as P. Syntactically splitting the process behavior $\alpha\,[P]$ from its mailbox $\alpha : \vec{M}$ as two separate resources allows a process to send to its own mailbox. That is the case when, in rule send, α_1 is equal to α_2. Rule recv reads from the mailbox the first message M that matches one of the patterns p_i in the receiving term. The matching function, if defined, yields a substitution σ which we apply to term P_j, corresponding to the selected pattern p_j. The message is removed from the mailbox. If no pattern matches M, then the configuration does not reduce. Rule mkref creates two fresh references r_1 and r_2 and replaces them by bound variables X_1 and X_2 in term P. Each reference becomes associated in the ν-binder to the correspondent process identifier, α_1 or α_2. Rule spawn creates a fresh pid α_2 for the spawned term P. Two new resources are created: process $\alpha_2\,[P\{^{\alpha_2}/_{\mathsf{self}}\}]$ where the self variable is replaced

$$\alpha_1\,[\alpha_2!M, P] \mid \alpha_2 : \vec{M} \longrightarrow \alpha_1\,[P] \mid \alpha_2 : \vec{M}M \qquad (\mathsf{send})$$

$$\frac{j \in I \quad \mathsf{match}(p_j, M) = \sigma \quad \mathsf{match}(p_i, M') \text{ undefined } \forall\, i \in I, \forall M' \in \vec{M}_1}{\alpha : \vec{M}_1 M \vec{M}_2 \mid \alpha\,[\mathsf{receive}\ p_i \to P_i\ ^{i \in I}] \longrightarrow \alpha : \vec{M}_1 \vec{M}_2 \mid \alpha\,[P_j\sigma]} \qquad (\mathsf{recv})$$

$$\alpha\,[\mathsf{make\_ref}\ X_1, X_2\ \mathsf{for}\ \alpha_1, \alpha_2\ \mathsf{in}\ P] \longrightarrow (\boldsymbol{\nu} r_1^{\alpha_1} r_2^{\alpha_2})\alpha\,[P\{^{r_1 r_2}/_{X_1 X_2}\}] \qquad (\mathsf{mkref})$$

$$\alpha_1\,[\mathsf{spawn}\ P\ \mathsf{as}\ X\ \mathsf{in}\ Q] \longrightarrow (\boldsymbol{\nu}\alpha_2)(\alpha_1\,[Q\{^{\alpha_2}/_X\}] \mid \alpha_2\,[P\{^{\alpha_2}/_{\mathsf{self}}\}] \mid \alpha_2 : \epsilon) \qquad (\mathsf{spawn})$$

$$\frac{C_1 \longrightarrow C_2}{C_1 \mid C_3 \longrightarrow C_2 \mid C_3} \qquad \frac{C_1 \longrightarrow C_2}{(\boldsymbol{\nu} n)C_1 \longrightarrow (\boldsymbol{\nu} n)C_2} \qquad \frac{C_1 \equiv C_2 \longrightarrow C_3 \equiv C_4}{C_1 \longrightarrow C_4} \qquad (\mathsf{par}, \mathsf{res}, \mathsf{str})$$

Fig. 4. Reduction

by α_2, and the (empty) queue $\alpha_2 : \epsilon$. The newly created pid is replaced in the continuation process Q, so that Q may then communicate with the new process.

What can go wrong with our machine? Looking at the operational semantics (Figure 4) nothing, really. Send always succeeds (for we admit mailbox buffers to be unbounded); receive may not succeed (for two reasons: no message in mailbox, no message in the mailbox matches the patterns) but that does not constitute an abnormal behaviour; finally, there is no reason why make_ref and spawn should not succeed.

The possible abnormal conditions have to do with our understanding of how sessions must happen. We identify two cases: a process terminates (reduces to a value) but leaves session messages in the mailbox; a process tries to receive a message with a given label within a given session but finds no such message in the mailbox. For the former case and given the asynchronous nature of our operational semantics, one may still find, at termination and in the mailbox, a session initiation message followed by session messages. This does constitute a malfunctioning since the session was never started on the server side. In the latter case, processes need not receive messages for all open sessions at all times, but if they decide to receive a message on a given session, then they must contain patterns for all possible messages in that session (otherwise one or both of the participants can get stuck by being unable to receive the next message).

We then say that a *configuration C constitutes an error* when C is structural congruent to $(\boldsymbol{\nu}\vec{n})(\alpha\,[P] \mid \alpha : \vec{M} \mid C')$ and

Incomplete session: term P is a value, buffer $\vec{M}$ is of the form $\vec{M}_1\{\_, X, \_\}\vec{M}_2$, and no message in $\vec{M}_1$ is of the form $\{\_, \_, X, \_\}$, or

Unmatched session message: term P is $\mathsf{receive}\,(\{X, Y, \_\}\,\mathsf{when}\,X, Y = a, r \to Q, \dots)$, there is one message in $\vec{M}$ of the form $\{\_, r, \_\}$ but no message of the form $\{a, r, \_\}$.

The type system in the next section filters out such abnormal cases.

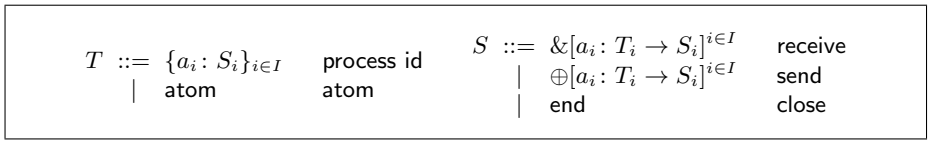

Fig. 5. Types

4 Typing

This section introduces our type system and presents its main result.

The syntax of *types* is in Figure 5. We distinguish types T for shared data and session types S. In the former category we have types for pids, $\{a_i\colon S_i\}_{i\in I}$, describing the set of sessions a process may engage in, and the type of atoms. For session types we distinguish a type $\&[a_i\colon T_i \to S_i]^{i\in I}$ describing patterns in a receive term labelled with a_i, receiving values of type T_i, and proceeding as prescribed by S_i; a type $\oplus[a_i\colon T_i \to S_i]^{i\in I}$ describing the various messages a client may send; and end, a type describing the completed session. A process may engage in different new sessions S_i, each labelled with a different label a_i. Receiving on a given session yields a type $\&[a_i\colon T_i \to S_i]^{i\in I}$; a client that sends on the same session has the *dual* type $\oplus[a_i\colon T_i \to \overline{S}_i]^{i\in I}$, where $\overline{S}$ denotes the type dual of S. Type end is dual of itself.

We use two sorts of typing environments: *shared environments*, Γ, containing entries of the form $p\colon T$, and *linear environments*, Δ, containing entries $(u_1, u_2 \triangleright p)\colon S$ and $(u_1^{p_1}, u_2^{p_2})\colon \mathsf{ref}$ (with p, p_1, p_2 variables or process identifiers, and u_1, u_2 variables or references). An entry of the form $(u_1, u_2 \triangleright p)\colon S$ describes a session running between the current process and p, using references u_1 and u_2, and at state S; an entry $(u_1^{p_1}, u_2^{p_2})\colon \mathsf{ref}$ describe a pair of references u_1, u_2 destined to be used in a session between processes with pids p_1 and p_2.

In typing rules we will freely compose Δ environments assuming that the result is defined (or the respective rule cannot be applied). The principle of composition is that when a pair of new references is added, the references do not already occur in the environment; also, when a session usage is added, the only allowed occurrence of the mentioned references is in a dual usage where they appear in reverse order. Formally, we have that $\Delta, (u_1^{p_1}, u_2^{p_2})\colon \mathsf{ref}$ is defined when $u_1 \neq u_2$ and, if $(u_3^{p_3}, u_4^{p_4})\colon \mathsf{ref} \in \Delta$ or $(u_3, u_4 \triangleright p_4)\colon S_2 \in \Delta$, then $u_{1,2} \notin \{u_3, u_4\}$. Similarly, $\Delta, (u_1, u_2 \triangleright p_1)\colon S_1$ is defined when $u_1 \neq u_2$ and, if $(u_3^{p_3}, u_4^{p_4})\colon \mathsf{ref} \in \Delta$ then $u_{1,2} \notin \{u_3, u_4\}$, and if $(u_3, u_4 \triangleright p_4)\colon S_2 \in \Delta$, then $u_{1,2} \neq u_{3,4}$ and $u_1 = u_4$ iff $u_2 = u_3$.

The type system for terms is in Figure 6. Sequents are of the form $\Gamma; \Delta \vdash_u P\colon T$, meaning that, under contexts Γ and Δ, term P with pid u has type T.

The rules for identifiers and atoms should be evident; we require 'completed' linear contexts at the leaves of typing derivations, as usual in session type systems. We then have two rules for message send, one to initiate a new session, the other to output on a running session. In the former case, we make sure that

$$\Gamma, u\colon T; \{(u_i, w_i \triangleright p_i)\colon \mathsf{end}\}_{i\in I} \vdash_{\_} u\colon T \qquad \Gamma; \{(u_i, w_i \triangleright p_i)\colon \mathsf{end}\}_{i\in I} \vdash_{\_} a\colon \mathsf{atom} \quad (\mathsf{identifier,atom})$$

$$\frac{\Gamma; \_ \vdash_{\_} p\colon \{a_i\colon S_i\}_{i\in I} \quad \Gamma; \Delta, (u_2, u_1 \triangleright p)\colon \overline{S_j} \vdash_u P\colon T \quad j \in I}{\Gamma; \Delta, (u_1^p, u_2^u)\colon \mathsf{ref} \vdash_u p!\{a_j, u_1, u_2, u\}, P\colon T} \quad (\mathsf{request})$$

$$\frac{\Gamma; \_ \vdash_{\_} V\colon T_j \quad \Gamma; \Delta, (u_1, u_2 \triangleright p)\colon S_j \vdash_u P\colon T \quad j \in I}{\Gamma; \Delta, (u_1, u_2 \triangleright p)\colon \oplus[a_i\colon T_i \to S_i]^{i\in I} \vdash_u p!\{a_j, u_2, V\}, P\colon T} \quad (\mathsf{out})$$

$$\frac{\Gamma; \Delta \vdash_u^{\mathsf{acc}} p_i \to P_i\colon T \quad \Gamma; \Delta \vdash_u^{\mathsf{in}} q_j \to Q_j\colon T \quad \forall i \in I, j \in J \quad \mathsf{consistent}(\Delta, (q_j)^{j\in J})}{\Gamma; \Delta \vdash_u \mathsf{receive}\,(p_i \to P_i)^{i\in I}, (q_j \to Q_j)^{j\in J}\colon T} \quad (\mathsf{receive})$$

$$\frac{\Gamma; \_ \vdash_{\_} u\colon \{a_i\colon S_i\}_{i\in I} \quad \Gamma; \Delta, (X_1, X_2 \triangleright p)\colon S_j \vdash_u P\{^{a_j}/_{X_a}\}\colon T \quad j \in I}{\Gamma; \Delta \vdash_u^{\mathsf{acc}} \{X_a, X_1, X_2, p\}\ \mathsf{when}\ X_a = a_j \to P\colon T} \quad (\mathsf{accept})$$

$$\frac{\Gamma, Y\colon T_j; \Delta, (u_1, u_2 \triangleright p)\colon S_j \vdash_u P\{^{a_j u_1}/_{X_a X}\}\colon T \quad j \in I}{\Gamma; \Delta, (u_1, u_2 \triangleright p)\colon \&[a_i\colon T_i \to S_i]^{i\in I} \vdash_u^{\mathsf{in}} \{X_a, X, Y\}\ \mathsf{when}\ X_a X = a_j u_1 \to P\colon T} \quad (\mathsf{in})$$

$$\frac{\Gamma, X\colon T; \Delta_1 \vdash_X P\{^X/_{\mathsf{self}}\}\colon \_ \quad \Gamma, X\colon T; \Delta_2 \vdash_u Q\colon T}{\Gamma; \Delta_1, \Delta_2 \vdash_u \mathsf{spawn}\, P\, \mathsf{as}\, X\, \mathsf{in}\, Q\colon T} \quad (\mathsf{spawn})$$

$$\frac{\Gamma; \Delta, (X^u, Y^v)\colon \mathsf{ref} \vdash_u P\colon T}{\Gamma; \Delta \vdash_u \mathsf{make\_ref}\, X, Y\, \mathsf{for}\, u, v\, \mathsf{in}\, P\colon T} \quad (\mathsf{mkref})$$

Fig. 6. Typing rules for terms

the process on p knows how to start an a_j session, read (and remove) the pair of references u_1, u_2 from Δ and add a new session-entry to Δ. The new entry records the two references, the pid of the target process and the dual (since we are on the client side) of the session type for session a_j. In the latter case we are within a session: we type check the continuation term P to obtain a type S_j for the session pertaining to u_2 (the write reference) and build a $\oplus$ type accordingly.

The rule for receive is the most complex one for there may be multiple branches, some trying to open new sessions, others trying to progress on already open sessions. We assume the branches partitioned in two sets: those opening new sessions and those engaged in open sessions. For the former we use rule accept which should be confronted with rule request. This time we use S_j because we are on the server side; we also propagate the effect of pattern matching on the continuation process P, via an appropriate substitution. For the latter we use rule in which should be confronted with rule out: we place an entry for message payload Y in the shared environment and propagate the substitution as in accept; for the type of the session, we use a & type, rather than a $\oplus$ type.

In the rule for receive all branches must have the same linear context Δ. But this is not enough, for in rule in we 'guess' from one label a_j the whole set of labels in a *receive* session type. We must then make sure that we do not declare in the type labels that are not in the receive pattern. Predicate consistent is used for the effect. We say that context Δ *is consistent with a set of patterns*

$(\{X_i, Y_i, \_\}$ when $X_i, Y_i = a_i, u_i)^{i \in I})$ when $\forall i \in I.(u_i, \_ \vdash \_)$: $\&[a\colon \_ \to \_, \dots] \in \Delta$ implies $\exists j \in I$ s.t. $a = a_j$ and $u_i = u_j$.

For spawn, we place an entry $X : T$ for the spawned process P in the typing environment and type check P by replacing self by X. The continuation term Q also knows X at type T. The shared environment is passed to both terms, whereas the linear one is split in two, one for each term. The rule for make_ref places a new ref-entry for the newly created pair of references in the linear context, and type checks the continuation process P.

At this point we can explain the reasons behind using two references per session instead of just one. Consider the following example:

```
clientAndServer () = make_ref X,Y for self, self in self ![connect,X,Y,self},
                       receive {connect,X,Y,Client} →
                         self ![hello,Y,_}, receive {hello,Y,_} →...
```

The above code, in which the request is made to self, is typable in our system, but if we had been using only one reference X, the presence of both ends of a session in a single term would (eventually, after some steps) produce a single typing for $(X \vdash \mathsf{self})$ which would include the actions of both participants (sending of {*hello*...} followed by receive of the same message) on one session type, due to the *aliasing* of the two intended uses of X in one place. This soundness problem with aliased endpoints is well-understood in the session types literature; see [12].

The type system in Figure 6 does not yield an obvious algorithm: it requires splitting linear context in rule spawn, as well guessing types in different rules. For the former problem there are well-known techniques associated with linear type systems that pass the whole context to one of the subterms, get back the unused part of the context and pass it to the second subterm; see e.g., [11]. The second problem occurs in rules spawn, out and in. In the first case, the common solution is to seek the help of programmers by requiring a type annotation for the pid of the spawned process P, providing the session types for its various services. This would avoid tedious annotation of every receive, in which new sessions are intermixed with existing ones that, moreover, can be partially satisfied. In rule in we need to guess the right &-type based on one of its branches. All these branches are then gathered together in rule receive where all types are checked for consistency via predicate consistent. The strategy here goes along the lines of preparing, in rule in, singleton branch types, and then merging them all together in rule receive. Finally, for rule out we record one only $\oplus$-branch in the type and add the remaining types to match the requirements in the remaining rules.

In order to prove subject-reduction we also have to type configurations. To facilitate typing in the presence of mailboxes, we introduce *types* τ *for messages in mailboxes*. A type $a(T)@r$ represents a session message with reference r carrying an atom a and a value of type T; type req is for new session requests.

The typing rules for configurations are in Figure 7. When typing with process, the actual process id α is propagated in the typing of the enclosed term, ensuring that it is understood as self. Rule par splits the linear context, and passes each part to a different sub-configuration (cf. rule spawn for terms in Figure 6). In rule newpid we introduce two usages for the subject pid: we add $\alpha : T$ in the

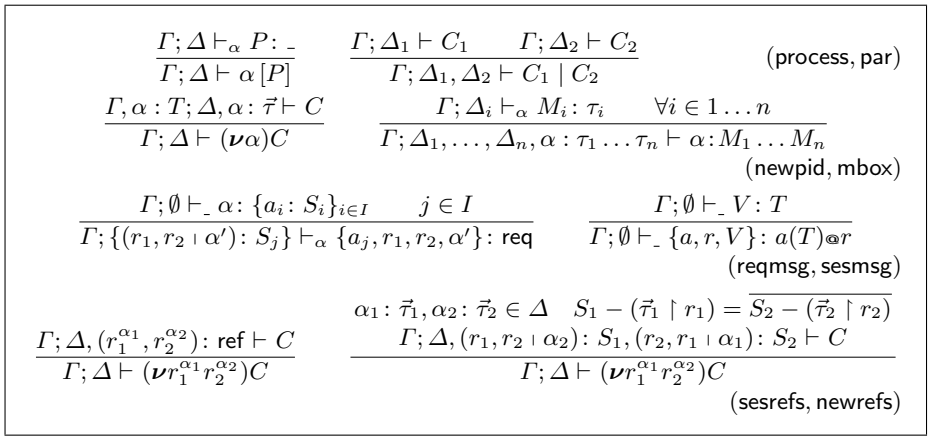

Fig. 7. Typing rules for configurations

shared environment, exposing a type for incoming requests, and we also expect in the linear environment some entry $\alpha\colon \vec{\tau}$ for the corresponding mailbox.

Rule mbox which types each message in the mailbox of α and composes the linear environments together with a sequence of message types for α. In turn, we can examine the message typing rules reqmsg and sesmsg. In reqmsg the request message introduces, in the linear environment, the usage that the process receiving the message would perform, which is needed to match the symmetric (dual) usage obtained with rule request of Figure 6. Observe that the given type req does not need to carry additional information. Then in sesmsg a session message is given a type $a(T)@r$; a sequence of such message types can inform about the messages of a session that are already in the mailbox, and is used to obtain the correct *remaining* usage (modulo these messages) per session.

Rule newrefs is for when a pair of references has been created, but a session request message has not been sent yet. It facilitates a subsequent use of rule request. Rule sesrefs ensures that sessions are *dual*. To obtain the actual session type that remains to be performed on each side of a session, we carefully advance the session types S_i of each session partner according to the types of messages already received. To achieve this, we utilise two auxiliary definitions. First, we want to extract from a mailbox the message type information that pertains to the specific reference r_i used for input; for this we use $(\tau_i \upharpoonright r_i)$ defined as:

$$\mathsf{req}\vec{\tau} \upharpoonright r = \vec{\tau} \upharpoonright r \qquad a(T)@r\vec{\tau} \upharpoonright r = a(T)(\vec{\tau} \upharpoonright r) \qquad a(T)@r'\vec{\tau} \upharpoonright r = \vec{\tau} \upharpoonright r \text{ if } r \neq r'$$

which generates a sequence (written $\vec{\rho}$) of message pre-types $a(T)$ stripped of reference information. Then, we advance each session type S_i by calculating the *session remainder* S_i' given from $S_i - \rho_i = S_i'$. The remainder is defined as:

$$S - \epsilon \;=\; S \qquad\qquad \&[a_i\colon T_i \to S_i]^{i\in I} - a_j(T_j)\vec{\rho} = S_j - \vec{\rho} \quad \text{if} \quad j \in I$$
$$\oplus[a_i\colon T_i \to S_i]^{i\in I} - \vec{\rho} \;=\; \oplus\,[a_i\colon T_i \to S_i]^{i\in I}$$

In the above definition, branch types advance according to received messages, but selections remain unchanged since they correspond to the messages that will be sent, and not to those that are received.

The basic tenet of sessions is that remaining communications always "match," captured by the notion of type duality. To this end, following the conditions of type rule sesrefs, we define *balanced* environments below.

Definition 1 (Balanced Δ). *Predicate* $\mathsf{balanced}(\Delta)$ *holds if* $(\underline{r_1, r_2 \vdash \alpha_2})\colon S_1$ *and* $(r_2, r_1 \vdash \alpha_1)\colon S_2, \alpha_1\colon \vec{\tau}_1, \alpha_2\colon \vec{\tau}_2$ *in* Δ *implies* $S_1 - (\tau_1 \restriction r_1) = \overline{S_2 - (\tau_2 \restriction r_2)}$.

Next, we define an ordering on linear environments that specifies the ways in which typings evolve with reduction.

Definition 2 (Δ Reduction). *We define* $\Delta \Rightarrow \Delta'$ *as follows:*

$$(r_2, r_1 \vdash \alpha_1)\colon S, (r_1, r_2 \vdash \alpha_2)\colon \&[a_i\colon T_i \to S_i]^{i\in I}, \alpha_1 : \vec{\tau}_1 a_j(T_j)@r_1\vec{\tau}_2 \Rightarrow (r_2, r_1 \vdash \alpha_1)\colon S, (r_1, r_2 \vdash \alpha_2)\colon S_j, \alpha_1 : \vec{\tau}_1\vec{\tau}_2 \quad \textit{if} \quad j \in I$$

$$(r_1, r_2 \vdash \alpha_2)\colon \oplus[a_i\colon T_i \to S_i]^{i\in I}, \alpha_2 : \vec{\tau}_2 \Rightarrow (r_1, r_2 \vdash \alpha_2)\colon S_j, \alpha_2 : \vec{\tau}_2 a_j(T_j)@r_2 \ \textit{if}\ j \in I$$

$$(r_1^{\alpha_1}, r_2^{\alpha_2})\colon \mathsf{ref} \Rightarrow (r_2, r_1 \vdash \alpha_1)\colon S, (r_1, r_2 \vdash \alpha_2)\colon \overline{S}$$

$$\Delta \Rightarrow \Delta \qquad\qquad \Delta_1, \Delta_2 \Rightarrow \Delta_1', \Delta_2 \quad \textit{if} \quad \Delta_1 \Rightarrow \Delta_1'$$

A property of the evolution of linear environments with $\Rightarrow$ is that it preserves balance, which in turn constitutes a measure of type soundness.

Lemma 1 (Balance Preservation). *If* $\mathsf{balanced}(\Delta)$ *and* $\Delta \Rightarrow \Delta'$ *then* $\mathsf{balanced}(\Delta')$.

Subject Reduction (type soundness) ensures that after reduction processes can be typed and that the resulting linear environment follows the above ordering. By Balance Preservation, this implies that the resulting environment is also balanced. The same can be easily shown for structural transformation.

Theorem 1 (Subject Reduction). *If* $\Gamma; \Delta \vdash C$ *with* $\mathsf{balanced}(\Delta)$ *and* $C \longrightarrow C'$, *then* $\Gamma; \Delta' \vdash C'$ *with* $\Delta \Rightarrow \Delta'$.

We can now state Type Safety which guarantees that configurations that are typed with balanced environments never reduce to an error configuration. Note also that environments are always balanced for user-level code in which no free references occur.

Theorem 2 (Type Safety). *If* $\Gamma; \Delta \vdash C$ *with* $\mathsf{balanced}(\Delta)$, *then* C *does not reduce to an error.*

Proof (Outline). Type Safety can be proved easily by contradiction: since we have Subject Reduction it is enough to show that error processes are not typable. In the case of an incomplete session with input reference r (where the corresponding request message has been consumed), the only possible typing

mentioning r in a terminated process $\alpha[V]$ will be end, and the mailbox will have a non-empty set of session messages on r not preceded by a corresponding request message (with input reference r); therefore the session remainder will be undefined. In the case of unmatched messages, we can show that a configuration in which a mailbox contains a message carrying r together with an atom that is not supported in the receiving process is untypable, since again the message remainder will be undefined. In both cases an application of sesrefs will fail.

There are other undesirable configurations, namely when the same reference appears in messages occurring in parallel threads (causing non-determinism in the receiving order), or when subsequent (or parallel) requests share some reference. However, such configurations are trivially untypable, since the linear environments composed in these cases are undefined.

5 Further work

Some Erlang programs consist of simple message exchanges and do not require provisions for sessions, in particular the use of references. We can easily adapt our system to handle these cases by extending pid types to $\{a_i : S_i, b_j : T_j\}_{i \in I, j \in J}$ allowing a process to receive simple messages such as $\{b, V\}$. Then, receive patterns of the shape $\{X, Y\}$ when $X = b$ can be typed using an extra rule in the style of accept, to be invoked from the receive rule in Figure 6.

Our type system guarantees that all within-session messages have a chance of being received. It would be desirable to also guarantee this property for session initiation messages, thus offering stronger behaviour guarantees. Intuitively, we need to ensure that at any state, terms can receive all possible session-initiation messages, either immediately or by reducing to a state that does so. A technique along the lines of non-uniform receptivity may prove helpful [1]. Moreover, since Erlang has general pattern matching, it would be useful to allow guards to impose constraints on the *values* received (e.g., receive only integer 5), and this can be achieved by using dependent types.

Delegation is the term used to describe the ability to pass a session identifier on a message. It allows, e.g., for a server to balance its load by sending some (open) sessions to other servers. The very nature of Erlang makes delegation a delicate matter, as opposed to the pi calculus where it is built in the language. Due to the nature of Erlang semantics, where communication is buffered, each process is co-located with its mailbox, and messages are addressed to pids, delegation requires a fairly complex protocol, and remains outside the scope of this work (if interesting at all in Erlang). A possible source of inspiration may come from the work on Session Java where a runtime API implements a delegation protocol for socket based session communication [8].

In order to concentrate on the novelty of our proposal, we deliberately excluded unbound behaviour. Such an extension should be easy to include via, e.g., recursive term definitions, as explained in Section 3. Realistic examples may require recursive types. This is, e.g., the case of our example in Section 2 if we allow

an unbounded number of store or load operations in a sequence. Fortunately, recursion in session types is well studied (see, e.g., [6, 12]) and its incorporation in the present setting should not present difficulties. In order to better convey our typing proposal, the typing system in this paper is not algorithmic. We are nevertheless confident that there is an equivalent algorithmic type system (see discussion in Section 4).

Acknowledgements. We are indebted to the anonymous reviewers and to Kostis Sagonas for their comments. This work was supported by FCT/MCTES via projects PTDC/EIA–CCO/105359/2008 and CMU–PT/NGN44–2009–12.

References

1. Roberto M. Amadio, Gérard Boudol, and Cédric Lhoussaine. On message deliverability and non-uniform receptivity. *Fundam. Inf.*, 53:105–129, May 2002.
2. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.
3. Business process execution language for web services. Available at `http://public.dhe.ibm.com/software/dw/specs/ws-bpel/ws-bpel.pdf`.
4. Richard Carlsson. An introduction to Core Erlang. In *PLI01 Erlang Workshop*, 2001.
5. Maria Christakis and Konstantinos Sagonas. Detection of asynchronous message passing errors using static analysis. In *Practical Aspects of Declarative Languages (PADL'2011)*, volume 6539 of *LNCS*, pages 5–18. Springer, 2011.
6. Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
7. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
8. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *Proceedings of ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
9. Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP'06, pages 167–178. ACM, 2006.
10. Mirko Viroli. Towards a formal foundation to orchestration languages. *Electronic Notes in Theoretical Computer Science*, 105:51 – 71, 2004. Proceedings of the First International Workshop on Web Services and Formal Methods (WSFM 2004).
11. David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2005.
12. Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *1st International Workshop on Security and Rewriting Techniques*, volume 171(4) of *ENTCS*, pages 73–93. Elsevier, 2007.