Affine Sessions

Dimitris Mostrous, Vasco Vasconcelos

Affine Sessions

Dimitris Mostrous and Vasco Thudichum Vasconcelos

University of Lisbon, Faculty of Sciences and LaSIGE
Lisbon, Portugal

Abstract. Session types describe the structure of protocols from the point of view of each participating channel. In particular, the types describe the type of communicated values, and also the dynamic alternation of input and output actions on the same channel, by which a protocol can be statically verified. Crucial to any term language with session types is the notion of linearity, which guarantees that channels exhibit exactly the behaviour prescribed by their type. We relax the condition of linearity to that of affinity, by which channels exhibit at most the behaviour prescribed by their types. This more liberal setting allows us to incorporate an elegant error handling mechanism which simplifies and improves related works on exceptions. Moreover, our treatment does not affect the progress properties of the language: sessions never get stuck.

1 Introduction

A session is "*a semantically atomic chain of communication actions which can interleave with other such chains freely, for high-level abstraction of interaction-based computing*" [21]. *Session types* capture this intuition as a description of the structure of a protocol, in the simplest case between two programs (binary sessions). This description consists of types that indicate whether a communication channel will next perform an output or input action, the type of the value to send or receive, and what to do next, inductively. For example, $!\mathsf{nat}.!\mathsf{string}.?\mathsf{bool}.\mathsf{end}$ is the type of a channel that will first send a value of type nat, then one of type string, then wait for a value of type bool, and nothing more. This type can be materialised by the π-calculus [19] process $\overline{a}5.\overline{a}\text{“hello”}.a(x).\mathbf{0}$. To compose two processes that communicate over a channel, we require that each has a complementary (or *dual*) type, so that an input will match an output, and vice versa. The dual of the previous type is $?\mathsf{nat}.?\mathsf{string}.!\mathsf{bool}.\mathsf{end}$, and can be implemented by $a(x).a(y).\overline{a}(x+1<2).\mathbf{0}$. To ensure that the actions take place in the prescribed order, session typing relies crucially on the notion of *linearity* [12], which means that a causal chain can be assumed. To see why, imagine that we write the first process as $\overline{a}5.\mathbf{0} \mid \overline{a}\text{“hello”}.a(x).\mathbf{0}$. Now we cannot determine which output can react first, and the second process can receive a "hello" first, which would clearly be unsound and would most likely raise an error.

Beyond the basic input/output types, sessions typically provide constructors for alternative sub-protocols, which are very useful for structured interaction. For example, $\&\{\mathsf{go}\colon T_1, \mathsf{cancel}\colon T_2\}$ can be assigned to an (external) choice $a \triangleright$

$\{\mathsf{go}.P_1 \,[]\, \mathsf{cancel}.P_2\}$. The dual type, where $\overline{T}$ denotes T with an alternation of all constructors, is $\oplus\{\mathsf{go}\colon \overline{T_1}, \mathsf{cancel}\colon \overline{T_2}\}$, and corresponds to a process that will make a (internal) choice, either $\overline{a} \triangleleft \mathsf{go}.Q_1$ or $\overline{a} \triangleleft \mathsf{cancel}.Q_2$. In the first case the two processes will continue as P_1 and Q_1, respectively.

As can be seen, sessions are very suitable as a static verification mechanism for interacting programs. However, they are also quite rigid, since everything in the description of a session type *must* be implemented by a program with that type. Indeed, in many real world situations, interactions are structured but can be aborted at any time, for example an online store should be prepared for clients that get disconnected, that close their web browsers, or for general *errors* that abruptly severe the expected pattern of interaction.

In this work, we address the above issues. In technical terms, we relax the condition of linearity to that of *affinity*, inspired by Affine Logic (which is the variation of Linear Logic with unrestricted weakening; see [2] for an introduction), and this allows processes to terminate their sessions prematurely. However, a naive introduction of affinity can leave programs in a stuck state: let us re-write the first process into $\overline{a}5.\overline{a}$"hello".$\mathbf{0}$, i.e., without the final input; then, after two communications the dual process will be stuck trying to perform $\overline{a}(5+1<2).\mathbf{0}$. One of the basic tenets of sessions, *progress*, is now lost. Actually, the study of Proof Nets for Affine Logic [2] reveals that weakening is not, and should not be, invisible. In particular, there exists a device that will perform the weakening step by step, progressing through the dependencies of a proof, and removing all that must be removed. This is exactly what we need in order to handle an abrupt termination of an interaction in an explicit way, and we denote it by $a\lightning$, which reads *cancel* a. Now we can write $\overline{a}5.\overline{a}$"hello".$a\lightning$, and after two steps against the dual process we obtain $\overline{a}(5+1<2).\mathbf{0} \mid a\lightning$, which results in the cancellation of the output (and in general, of any subsequent actions on a). We take this idea a step further: if cancellation of a session is explicit, we can treat it as an *exception*, and for this we introduce a *do-catch* construct that can provide an alternative behaviour activated when a cancellation is encountered. For example, we can write $\mathsf{do}\,\overline{a}(5+1<2).\mathbf{0}\,\mathsf{catch}\,P$, and a composition with $a\lightning$ will *replace* $\overline{a}(5+1<2).\mathbf{0}$ with the exception handler P. A do-catch is not the same as the try-catch commonly found in sequential languages: it does not define a persistent scope that captures exceptions from the inside, but rather it applies to the first communication and is activated by exceptions from the outside (as in the previous example). Thus, $\mathsf{do}\,\overline{a}(5+1<2).\mathbf{0}\,\mathsf{catch}\,P$ in parallel to $a(x).\mathbf{0}$ becomes $\mathbf{0}$, because the communication was successful.

2 Affine Sessions by Example

We describe a simple interaction that implements a book purchase taking place between three processes, Buyer, Seller, and Bank. The buyer sends the title of a book, receives the price, and makes a choice to either buy it or cancel. If the buyer chooses to buy the book, the credit card is sent over the session, and the

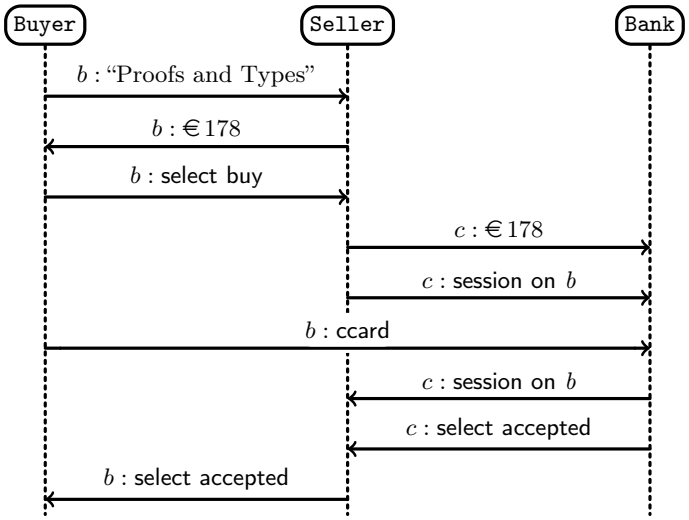

Fig. 1. Sequence Diagram for Succesful Book Purchase

buyer is informed whether or not the transaction was successful. The diagram in Figure 1 shows the interactions of a specific purchase.

We now show how this scenario can be implemented using sessions, and how our treatment of affinity can be used to enable a more concise and natural handling of exceptional outcomes. Our language is an almost standard π-calculus where replication is written $\mathsf{acc}\ a(x).P$ and plays the role of "accept" in sessions terminology [15]. Dually, an output that activates a replication is written $\mathsf{req}\,\overline{a}b.P$, and is called a "request". We will use some standard contructs that are encodable in π-calculus, like $\overline{a}e$ for an expression e. Also, we use $\mathsf{if}\,t\,\mathsf{then}\,P\,\mathsf{else}\,Q$ that can be implemented by a new session, specifically $a \triangleright \{\mathsf{true}.P \,[]\, \mathsf{false}.Q\}$ against some process representing the test t that communicates the result by a selection of one of the labels, $\overline{a} \triangleleft \mathsf{true}$ or $\overline{a} \triangleleft \mathsf{false}$.

An implementation of the protocol we described in Figure 1 is shown below:

$$\mathsf{Buyer} \doteq (\boldsymbol{\nu} b) \left(\begin{array}{l} \mathsf{req}\ \overline{\mathit{seller}}\, b \mid \overline{b}\text{"Proofs and Types"}.b(\mathit{price}).\mathsf{if}\ (\mathit{price} < 200) \\ \mathsf{then}\ \overline{b} \triangleleft \mathsf{buy}.\overline{b}\,\mathsf{ccard}.b \triangleright \{\mathsf{accepted}.P \,[]\, \mathsf{rejected}.Q\}\ \mathsf{else}\ \overline{b} \triangleleft \mathsf{cancel} \end{array} \right)$$

$$\mathsf{Seller} \doteq \mathsf{acc}\ \mathit{seller}(s).\left(\begin{array}{l} s(\mathit{prod}).\overline{s}\,\mathsf{price}(\mathit{prod}). \\ s \triangleright \{\mathsf{buy}.(\boldsymbol{\nu} c)(\mathsf{req}\ \overline{\mathit{bank}}\, c \mid \overline{c}\,\mathsf{price}(\mathit{prod}).\overline{c}s.c(r). \\ \qquad c \triangleright \{\mathsf{accepted}.\overline{r} \triangleleft \mathsf{accepted} \,[]\, \mathsf{rejected}.\overline{r} \triangleleft \mathsf{rejected}\}) \\ \qquad []\ \mathsf{cancel}.\mathbf{0}\} \end{array} \right)$$

$$\mathsf{Bank} \doteq \mathsf{acc}\ \mathit{bank}(k).\left(\begin{array}{l} k(\mathit{amount}).k(r).r(\mathit{card}).\overline{k}r. \\ \mathsf{if}\ \mathsf{charge}(\mathit{amount}, \mathit{card})\ \mathsf{then}\ \overline{k} \triangleleft \mathsf{accepted}\ \mathsf{else}\ \overline{k} \triangleleft \mathsf{rejected} \end{array} \right)$$

First we note how sessions are established. For example, in Buyer the fresh name b is sent to Seller via the request $\mathsf{req}\,\overline{seller}\,b$, where it will substitute s in a copy of the replication, and it appears also locally. These are the two endpoints of the session, and it is easy to check that the interactions match perfectly. Another point is the "borrowing" of the session b (which becomes identified with s) from Seller to Bank, with $\overline{c}s.c(r)$ and $k(r).r(card).\overline{k}r$ (again, c and k are identified), respectively, so that the credit card is received directly by Bank; see also Figure 1.

A more robust variation of Seller could utilise the do-catch mechanism to account for the possibility of the Bank not being available (or being crashed), by providing an alternative payment provider. This can be achieved if we substitute $\mathsf{req}\,\overline{bank}\,c$ with $\mathsf{do}\,\mathsf{req}\,\overline{bank}\,c\,\mathsf{catch}\,\mathsf{req}\,\overline{paymate}\,c$, so that a failure to use the bank service ($bank\lightning$) will activate $\mathsf{req}\,\overline{paymate}\,c$ (which must have the same type) and the protocol has a chance to complete successfully.

The Buyer might also benefit from our notion of exception handling. For example, we show an adaptation that catches a cancellation at the last communication of the buy branch and prints an informative message:

$$\mathsf{BuyerMsg} \;\doteq\; (\boldsymbol{\nu} b) \left(\begin{array}{l} \mathsf{req}\,\overline{seller}\,b \mid \overline{b}\text{“Proofs and Types”}.b(price). \\ \mathsf{if}\,(price < 200)\,\mathsf{then}\,\overline{b} \triangleleft \mathsf{buy}.\overline{b}\,\mathsf{ccard}. \\ \quad \mathsf{do}\,b \triangleright \{\mathsf{accepted}.P \,[]\, \mathsf{rejected}.Q\} \\ \quad \mathsf{catch}\,\mathsf{req}\,\overline{print}\text{“Error 42”} \\ \mathsf{else}\,\overline{b} \triangleleft \mathsf{cancel} \end{array} \right)$$

As mentioned in the Introduction, a do-catch on some communication does not catch subsequent cancellations. For instance, if in the above example the do-catch was placed on $\overline{b}$“Proofs and Types”, then any $b\lightning$ generated *after* this output was performed would be uncaught, since $\mathsf{req}\,\overline{print}$“Error 42” would have been already discarded. However, a do-catch does catch cancellations emitted *before* the point of definition, so placing it near the end of a protocol is very useful if we just want a single exception handler that catches everything.

Indeed, exception handlers that persist for the lifetime of the whole session are definable. Specifically, we can write $\mathsf{try}\,P\,\mathsf{catch}\,(b\colon Q)$ to mean that one endpoint of b is implemented in P and that Q should be activated if b is canceled at *any* point from the *outside* of P. This try-catch notation translates to a do-catch on the last prefix on b in P, in multiple branches as required, assuming that b is not delegated (i.e., sent over another channel), as is the case in BuyerMsg.[1] Then we have, for example, that $\mathsf{try}\,\overline{b}5.P'\,\mathsf{catch}\,(b\colon Q) \mid b(x).\mathbf{0}$ becomes $\mathsf{try}\,P'\,\mathsf{catch}\,(b\colon Q)$ and $\mathsf{try}\,\overline{b}5.P'\,\mathsf{catch}\,(b\colon Q) \mid b\lightning$ becomes $P'' \mid Q$ where P'' is P' with b and all its dependencies canceled. In general, however, our mechanism is very fine-grained, and a single session can have multiple, nested do-catch on crucial points of communication.

[1] We assume that in BuyerMsg P and Q do not use b. The restriction to non-delegated sessions is for simplicity: if the last action on b was to send it over some channel k, e.g., using $\overline{k}b$, the encoding would be more complicated, because we would not have access to the end of the session and anything after that output would not be caught.

Note also that $a\lightning$ can be very useful in itself, even without the do-catch mechanism. Here are two ways to implement a process that starts a protocol with Seller only to check the price of a book:

$$\mathsf{CheckPriceA} \doteq (\boldsymbol{\nu} b)\left(\mathsf{req}\ \overline{\mathit{seller}}\, b \mid \overline{b}\text{“The Prince”}.b(\mathit{price}).(\overline{b} \triangleleft \mathsf{cancel} \mid R)\right)$$
$$\mathsf{CheckPriceB} \doteq (\boldsymbol{\nu} b)\left(\mathsf{req}\ \overline{\mathit{seller}}\, b \mid \overline{b}\text{“Beyond Good and Evil”}.b(\mathit{price}).(b\lightning \mid R)\right)$$

Both the above processes can be typed. However, the first requires a knowledge of the protocol, which in that case includes an exit point (branch cancel), while the second is completely transparent. For example, imagine a buyer that selects buy by accident and then wishes to cancel the purchase: without cancellation it is impossible because it is not *predicted* by the session type; with cancellation it is extremely simple, as shown below.

$$\mathsf{BuyerCancel} \doteq (\boldsymbol{\nu} b)\left(\mathsf{req}\ \overline{\mathit{seller}}\, b \mid \overline{b}\text{“Gödel, Escher, Bach”}.b(\mathit{price}).\overline{b} \triangleleft \mathsf{buy}.b\lightning\right)$$

We now make a small digression to discuss how our affine sessions can be encoded in standard sessions. The purpose is to shed light on the complexity that is required, which motivates even more our development. First of all, it is possible that both endpoints of a session emit a cancellation, possibly at different moments. Therefore, if we are to encode this behaviour in a standard session system, we must allow a protocol to end at any point and by the request of either of the participants. This can be achieved by an exchange of a decision, to go or to cancel, by both endpoints, before all communications. We show the translation for output and input; the rest is similar.

$$[\![!T_1.T_2]\!] \doteq \oplus \left\{ \begin{array}{l} \mathsf{go} : \&\left\{\mathsf{go} : !T_1.[\![T_2]\!], \mathsf{cancel} : \mathsf{end}\right\}, \\ \mathsf{cancel} : \&\left\{\mathsf{go} : \mathsf{end}, \mathsf{cancel} : \mathsf{end}\right\} \end{array} \right\} \qquad \begin{array}{l} [\![?T_1.T_2]\!] \doteq \overline{[\![!T_1.\overline{T_2}]\!]} \\ [\![\mathsf{end}]\!] \doteq \mathsf{end} \end{array}$$

The notable point is that by an alternation of constructors we obtain a translation that preserves duality, and it is easy to check that it preserves soundness. Moreover, the only way to proceed is if both ends agree to go. The term-level translation follows the structure of the types, and $a\lightning$ becomes $\overline{a} \triangleleft \mathsf{cancel}$. All free sessions in the term must be canceled in the branches that do not result in normal execution so as to obtain the same typing environment, but this is always possible.

The above translation only handles cancellation. Our do-catch mechanism can also be encoded within the branches of the previous translation, but it becomes quite complicated due to the typing contraints that must be respected. In any case, we think it is obvious that the burden is heavy if one wishes to obtain a functionality as general as the one available in the affine system, and types would become completely illegible from the multiplication of constructors.

3 The Process Calculus of Affine Sessions

Syntax Our language is a small extension of standard π-calculus [19]. With respect to standard sessions systems [11,22], we avoid the need for *polarities*

and *double binders* by carefully introducing a logically-founded typing principle, detailed later. For technical convenience we shall consider all indexing sets I to be totally ordered, so that we can speak, e.g., of the maximum element. Also for technical convenience, we separate the prefixes denoted by ρ, i.e., all communication actions except for accept (replication). We only added two non-standard constructs: the *cancellation* $a\lightning$ and the *do-catch* construct that captures a cancellation, denoted by $\mathsf{do}\,\rho\,\mathsf{catch}\,P$. Notice that we restricted the action to a prefix in ρ, but this is not so limiting. In the case of replication, it does not make sense to catch an event that never occurs, since as we shall see we never explicitly cancel a persistent service. For parallel composition, it would be ambiguous to allow $\mathsf{do}\,(P \mid Q)\,\mathsf{catch}\,R$ since more than one action can be active in $(P \mid Q)$, and moreover we do not think it would really add any benefit since we can add a separate do-catch for each session. Similarly, $\mathsf{do}\,a\lightning\,\mathsf{catch}\,P$ would be strange: it would allow to trigger some behaviour when the other end is canceled, but while at the same time the protected session is canceled too. It can be added if a good use is discovered, but we preferred to keep the semantics simpler.

$$
\begin{array}{rll}
\rho ::= & a(x\colon T).P & \text{(input)} \\
\mid & \overline{a}b.P & \text{(output)} \\
\mid & a \triangleright \{l_i.P_i\}_{i\in I} & \text{(branching)} \\
\mid & \overline{a} \triangleleft l_k.P & \text{(selection)} \\
\mid & \mathsf{req}\,\overline{a}b.P & \text{(request)} \\
\\
P ::= & \rho & \text{(prefix)} \\
\mid & \mathsf{acc}\,a(x\colon T).P & \text{(replicated accept)} \\
\mid & \mathbf{0} & \text{(nil)} \\
\mid & P \mid Q & \text{(parallel)} \\
\mid & (\boldsymbol{\nu} a\colon T)P & \text{(restriction)} \\
\mid & a\lightning & \text{(cancel)} \\
\mid & \mathsf{do}\,\rho\,\mathsf{catch}\,P & \text{(catch)}
\end{array}
$$

Structural Congruence With $\equiv$ we denote the least congruence on processes that is an equivalence relation, equates processes up to α-conversion, satisfies the abelian monoid laws for parallel composition, the usual laws for scope extrusion, and satisfies the axiom:

$$(\boldsymbol{\nu} a\colon T)(a\lightning \mid \cdots \mid a\lightning) \equiv \mathbf{0}$$

We added this axiom mainly for the left-to-right direction which allows "leftover" cancellations to disappear; this is convenient for technical reasons.

Reduction We use two kinds of contexts, $C[\,]$ which are standard, and $H[\,]$ for (possible) exception handling, defined below.

$$\begin{array}{rl} \textit{Standard Contexts}: & C[\,\cdot\,] ::= \cdot \mid (\,C[\,\cdot\,] \mid P\,) \mid (\boldsymbol{\nu} a\colon T)\, C[\,\cdot\,] \\ \textit{Do-Catch Contexts}: & H[\,\cdot\,] ::= \cdot \mid \mathsf{do}\ \cdot\ \mathsf{catch}\ P \end{array}$$

Reduction is defined in two parts, first the standard rules, and then the cancellation rules. The standard reductions are defined below:

$$H_1[\,\overline{a}b.P\,] \mid H_2[\,a(x\colon T).Q\,] \longrightarrow P \mid Q\{^b/_x\} \qquad \text{(R-Com)}$$

$$H_1[\,\overline{a} \triangleleft l_k.P\,] \mid H_2[\,a \triangleright \{l_i.Q_i\}_{i\in I}\,] \longrightarrow P \mid Q_k \qquad (k \in I) \qquad \text{(R-Bra)}$$

$$H[\,\mathsf{req}\ \overline{a}b.P\,] \mid \mathsf{acc}\ a(x\colon T).Q \longrightarrow P \mid Q\{^b/_x\} \mid \mathsf{acc}\ a(x\colon T).Q \qquad \text{(R-Ses)}$$

$$P \equiv P' \longrightarrow Q' \equiv Q \;\Rightarrow\; P \longrightarrow Q \qquad \text{(R-Str)}$$

$$P \longrightarrow Q \;\Rightarrow\; C[\,P\,] \longrightarrow C[\,Q\,] \qquad \text{(R-Ctx)}$$

The only notable point is that we discard any do-catch handlers, since there is no cancellation, which explains why the H-contexts disappear.

The cancellation reductions follow:

$$\mathsf{req}\ \overline{a}b.P \mid a\lightning \longrightarrow a\lightning \mid b\lightning \mid P \qquad \text{(C-Req)}$$

$$\overline{a}b.P \mid a\lightning \longrightarrow a\lightning \mid b\lightning \mid P \qquad \text{(C-Out)}$$

$$a(x\colon T).P \mid a\lightning \longrightarrow (\boldsymbol{\nu} x\colon T)(a\lightning \mid x\lightning \mid P) \qquad \text{(C-Inp)}$$

$$\overline{a} \triangleleft l_k.P \mid a\lightning \longrightarrow P \mid a\lightning \qquad \text{(C-Sel)}$$

$$a \triangleright \{l_i.P_i\}_{i\in I} \mid a\lightning \longrightarrow P_k \mid a\lightning \qquad \mathsf{max}(I) = k \qquad \text{(C-Bra)}$$

$$\mathsf{do}\ \rho\ \mathsf{catch}\ P \mid a\lightning \longrightarrow P \mid a\lightning \qquad \mathsf{subject}(\rho) = a \qquad \text{(C-Cat)}$$

We discuss some notable points.

Only what is strictly needed will be deleted, in particular one might have expected $a(x\colon T).P \mid a\lightning$ to result in the annihilation of P, which can be done by generating $b\lightning$ for each b in the free names of P. However, this has several drawbacks: first, it is too absolute, since some interactions in P may not depend on a or x, and we prefer to preserve them; second, it is technically simpler, since in this setting we can use typing restrictions to avoid the creation of any $b\lightning$ for a replication $\mathsf{acc}\ b(x).Q$ inside P, which follows our decision to never delete services; finally, it is what happens in Proof Nets for Affine Logic (see [2]).

In the cancellation of branching, (C-Bra), we choose the maximum index k which exists by our assumption that index sets are totally ordered. This is a simple way to avoid non-determinism solely by cancellation.[2] Notice that it follows the pattern of activating a continuation, motivated above.

In the rule (C-Cat), we use a function $\mathsf{subject}(\rho)$ which returns the subject in the prefix of ρ. This is defined in the obvious way, e.g., $\mathsf{subject}(\overline{a}b.P) = \mathsf{subject}(\mathsf{req}\ \overline{a}b.P) = a$, and similarly for the other cases. The typing system will ensure that a does not appear in P, so it is ok to leave $a\lightning$ in the result; this is

[2] The language remains confluent, as expected in a logically founded system.

needed for canceled requests, where the $a\lightning$ should remain until it reacts with all of them.

We clarify some of the main points:

i) Consider $a(x).(\, Q \mid \mathsf{acc}\, b(z).R \,) \mid a\lightning$. The replication provided on b may or may not depend on x. A cancellation of a does not necessarily mean that b will be affected, but if x appears in R it is possible that subsequent sessions will be canceled.

ii) Consider $a(x).(\, \mathsf{acc}\, x(z).Q \mid \overline{b}x.R \,) \mid a\lightning$. As can be seen, the replicated channel x is delegated on b, and it should not be deleted just because a is canceled. Indeed, this situation is not allowed by the restrictions in our type system. In other words, some sessions *cannot* be canceled.

iii) Consider $a(x).(\, Q \mid \overline{b}e.R \,) \mid a\lightning$. The session output $\overline{b}e.R$ will not be canceled, but since it is possible that $e = x$ and in general e could appear in R, other cancellations may eventually be generated.

iv) A communication discards any handlers: $\mathsf{do}\, \overline{a}b.P\, \mathsf{catch}\, Q \mid a(x\colon T).R \longrightarrow P \mid R\{^{b}/_{x}\}$. The type system ensures that it is sound to discard Q, since it contains the same sessions as $\overline{a}b.P$, except for a.

v) A cancellation activates a handler, which may provide some default values to a session, completing it or eventually re-throwing a cancellation, as in: $\mathsf{do}\, \overline{a}b.P\, \mathsf{catch}\, (\overline{b}5.b\lightning\,) \mid a\lightning \longrightarrow \overline{b}5.b\lightning \mid a\lightning$.

4 Typing Affine Sessions

Types The session types we use are standard [15] with two exceptions. First, following [22] we allow a session type to evolve into a shared type. Second, we decompose shared types into accept types $\mathsf{acc}\ T$ and request types $\mathsf{req}\ T$, following the logical principles of Affine Logic. Technically, $\mathsf{acc}\ T$ corresponds to $!T$ ("of course T") and $\mathsf{req}\ T$ to $?\overline{T}$ ("why not $\overline{T}$") [12]. This has several technical advantages that simplify our development, for example $\mathsf{acc}\ T$ retains information on the persistence of a term with that type, since it must be replicated, which is useful for typing. Moreover, $\mathsf{req}\ T$ is the only type allowed in the context of a resource that can be used zero or more times.

$$
\begin{array}{rll}
T ::= & \mathsf{end} & \text{(nothing)} \\
\mid & !T.T & \text{(output)} \\
\mid & ?T.T & \text{(input)} \\
\mid & \oplus \{l_i\colon T_i\}_{i\in I} & \text{(selection)} \\
\mid & \& \{l_i\colon T_i\}_{i\in I} & \text{(branching)} \\
\mid & \mathsf{req}\ T & \text{(request)} \\
\mid & \mathsf{acc}\ T & \text{(accept)}
\end{array}
$$

Duality The two ends of a session are composed when their types are *dual*, which is defined as an involution over the type constructors, similarly to Linear Logic's

negation except that end is self-dual.[3]

$$\overline{!T_1.T_2} \doteq ?T_1.\overline{T_2} \qquad \overline{?T_1.T_2} \doteq !T_1.\overline{T_2}$$

$$\overline{\oplus\{l_i\colon T_i\}_{i\in I}} \doteq \&\{l_i\colon \overline{T_i}\}_{i\in I} \qquad \overline{\&\{l_i\colon T_i\}_{i\in I}} \doteq \oplus\{l_i\colon \overline{T_i}\}_{i\in I}$$

$$\overline{\mathsf{req}\ T} \doteq \mathsf{acc}\ T \qquad \overline{\mathsf{acc}\ T} \doteq \mathsf{req}\ T \qquad \overline{\mathsf{end}} \doteq \mathsf{end}$$

Typing Rules Typing judgements take the form:

$$P \triangleright \Gamma \qquad \text{where} \qquad \Gamma ::= \emptyset \mid \Gamma, a\colon T$$

meaning that term P has interface Γ. We shall also use Γ, Δ, Θ for interfaces.

We restrict replications to be unique, but allow multiple requests to take place against them. This means that processes can have multiple uses of $a\colon \mathsf{req}\ T$, which corresponds to the logical principle of contraction. For this, we make use of the splitting relation from [22]:

$$\frac{}{\emptyset = \emptyset \circ \emptyset} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, a\colon \mathsf{req}\ T = (\Gamma_1, a\colon \mathsf{req}\ T) \circ (\Gamma_2, a\colon \mathsf{req}\ T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, a\colon T = (\Gamma_1, a\colon T) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, a\colon T = \Gamma_1 \circ (\Gamma_2, a\colon T)}$$

In the typing rules, $\mathsf{req}\,\Gamma$ stands for an interface of the shape $a_1\colon \mathsf{req}\ T_1, \ldots, a_n\colon \mathsf{req}\ T_n$. Similarly, $\mathsf{end}\,\Gamma$ stands for an interface $a_1\colon \mathsf{end}, \ldots, a_n\colon \mathsf{end}$.

We also define a predicate $\mathsf{no\text{-}requests}(T)$, used to forbid any request type from appearing in the type of $a\lightning$, since this maps by duality to the deletion of a persistent accept on the other side, which we do not allow.[4]

$$\mathsf{no\text{-}requests}(\mathsf{req}\ T) = \mathsf{false}$$

$$\mathsf{no\text{-}requests}(\mathsf{acc}\ T) = \mathsf{no\text{-}requests}(T) \qquad \mathsf{no\text{-}requests}(\mathsf{end}) = \mathsf{true}$$

$$\mathsf{no\text{-}requests}(!T_1.T_2) = \mathsf{no\text{-}requests}(\overline{T_1}) \wedge \mathsf{no\text{-}requests}(T_2)$$

$$\mathsf{no\text{-}requests}(?T_1.T_2) = \mathsf{no\text{-}requests}(T_1) \wedge \mathsf{no\text{-}requests}(T_2)$$

$$\mathsf{no\text{-}requests}(\oplus\{l_i\colon S_i\}_{i\in I}) = \mathsf{no\text{-}requests}(\&\{l_i\colon S_i\}_{i\in I}) = \wedge_{i\in I}\mathsf{no\text{-}requests}(S_i)$$

The typing rules are presented in Figure 2. We focus on some key points. First, we type modulo structural equivalence, a possibility suggested by [18] and used in [4]. This is because associativity of " $\mid$ " does not preserve typability, i.e., a composition between P and $(Q \mid R)$ may be untypable as $(P \mid Q) \mid R$; see

[3] The expert might notice that logical negation suggests a dualisation of all components, e.g., $\overline{!T.T'} \doteq ?\overline{T}.\overline{T}'$ In fact the output type $!T.T'$ and the request $\mathsf{req}\ T$ hide a duality on T, effected by the type system, so everything is compatible.

[4] This method works fine until one adds a second-order fragment: then type substitutions must be carefully controlled, or some results will become slightly weaker.

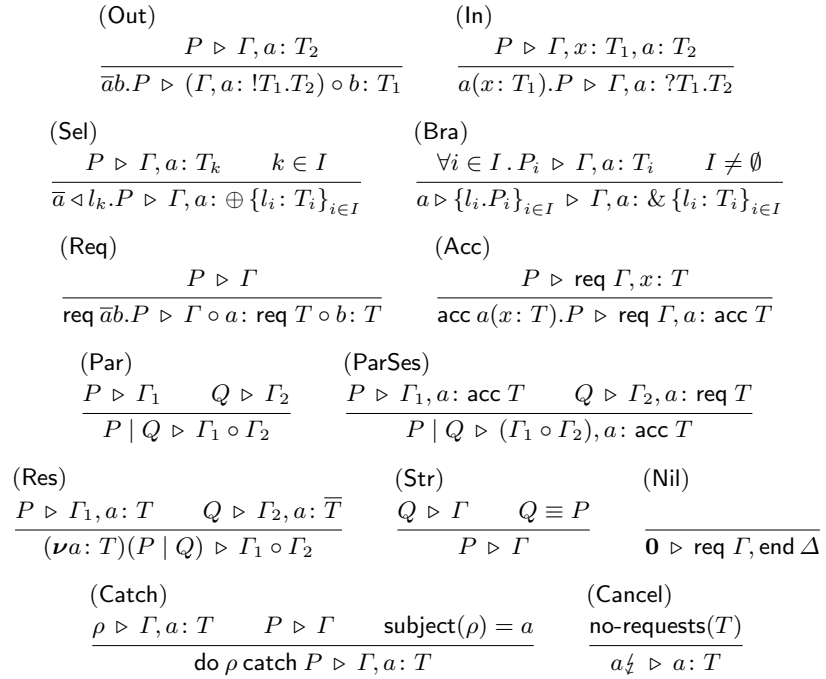

Fig. 2. Affine Session Typing

(ParSes), (Res). This applies also to $(\boldsymbol{\nu} a)$ that causes similar problems. In fact, the splitting of the terms in (Res) is inspired by the work [4] which interprets sessions as propositions in a form of Intuitionistic Linear Logic. It is because of this separation of terms, which applies also to (ParSes), that we can avoid channel polarities: the two ends of a session can never become causally dependent or intermixed. The purpose of (ParSes) is to type multiple requests against a persistent accept, which explains why $a\colon \mathsf{acc}\ T$ remains in the conclusion. An output $\overline{a}b.P$ records a conclusion $b\colon T_1$, so in fact it will compose against $b\colon \overline{T_1}$. Therefore $!T_1.T_2$ really means "send $\overline{T_1}$," which matches with the dual input. A cancellation $a\lightning$ can be given any type that does not contain a request, as explained previously.

A do-catch is typed as follows: if ρ is an action on a and has an interface $\Gamma, a\colon T$, then the handler P will implement Γ, i.e., all sessions of ρ *except* for $a\colon T$ which has been canceled. Of course, inside P these other sessions can be canceled anyway, which corresponds to "re-throwing" the cancellation, but they can also be implemented in whole or in part. The rule is sound, since no session is damaged, irrespectively of which term we execute, ρ or P.

Motivating the "no requests" restriction on the type of $a\lightning$ There are pragmatic motivations behind our decision to not allow cancellation of replicated terms,

namely that we do not wish a request to cancel a service possibly shared by many processes. However, there are also technical challenges, stemming from the fact that multiple actions of type $a\colon \mathsf{req}\ T$ can appear in a well-typed term, which as we explain below can create ambiguity in cancellation reductions.

Let us assume that the $\mathsf{no\text{-}requests}(T)$ restriction was lifted. Now, as an example consider the composition $\underline{\mathsf{req}\,\overline{a}b.P \mid a\lightning \mid \overline{c}a.Q} \mid c(x).\mathsf{acc}\ x(y).R$. First, let us look at the underlined term: it is impossible to know what is the type of $a\lightning$, as it could be either $\mathsf{acc}\ T$ or $\mathsf{req}\ T$. If the type is $\mathsf{acc}\ T$, which means that the (dual) accept is canceled, we should apply cancellation to $\mathsf{req}\ \overline{a}b.P$; if the type is $\mathsf{req}\ T$, i.e., if $a\lightning$ is in fact the cancellation of another request, then we should not touch $\mathsf{req}\,\overline{a}b.P$. In our example, it is easy to check that the replication will appear after a communication on c so the type of $a\lightning$ must be $\mathsf{req}\ T$, but in general it is not possible to determine this information (again, consider just the underlined term). Our restriction on the type of $a\lightning$ ensures that, in a case like the underlined term above, we can be sure that the type cannot be $\mathsf{req}\ T$, so it must be $\mathsf{acc}\ T$ and we can proceed to cancel $\mathsf{req}\,\overline{a}b.P$ using (C-Req). Indeed, the full composition is not typable since in that case the type of $a\lightning$ must be $\mathsf{req}\ T$, and $\mathsf{no\text{-}requests}(\mathsf{req}\ T)$ is not true. In fact, without our restriction, (C-Req) does not work any more, since it assumes $a\lightning$ to be of type $\mathsf{acc}\ T$, so we would need to replace it with:

$$(\boldsymbol{\nu} a\colon T)(\textstyle\prod_{i\in I} \mathsf{req}\,\overline{a}b_i.P_i \mid a\lightning) \longrightarrow (\boldsymbol{\nu} a\colon T)(\textstyle\prod_{i\in I}(b_i\lightning \mid P_i) \mid a\lightning) \qquad \text{(C-Req')}$$

This is the special case in which we know the type of $a\lightning$ must be $\mathsf{acc}\ T$, since it is bound and all other elements are requests. This variation is more complex, and we would also have to forego the ability to use do-catch on $\mathsf{req}\ \overline{a}b.P$, so we chose not to introduce it. Even if we did use this seemingly more liberal system, we would anyway not want replications to be deleted, so it would be of limited value. The only advantage of this alternative solution is that it does not put restrictions on the shape of types assigned to $a\lightning$, and therefore it works also with polymorphism (i.e., a second-order setting).

Typing the Book Purchase Example

It is easy to verify that the examples from Section 2 are well-typed. For the Buyer we obtain the following (for some $\mathsf{req}\ \Gamma_1$ and $\mathsf{end}\ \Delta_1$):

$$\mathsf{Buyer} \triangleright \mathsf{req}\ \Gamma_1, \mathsf{end}\ \Delta_1, \mathit{seller}\colon \mathsf{req}\ T_1$$

with $\overline{T_1} = !\mathsf{string}.?\mathsf{double}.\oplus \left\{ \begin{array}{l} \mathsf{buy}\colon !\mathsf{string}.\&\left\{\mathsf{accepted}\colon \mathsf{end}, \mathsf{rejected}\colon \mathsf{end}\right\}, \\ \mathsf{cancel}\colon \mathsf{end} \end{array} \right\}$.

The type $\overline{T_1}$ is the behaviour of b inside Buyer.

For the Seller we obtain:

$$\mathsf{Seller} \triangleright \mathsf{req}\ \Gamma_2, \mathit{bank}\colon \mathsf{req}\ T_2, \mathit{seller}\colon \mathsf{acc}\ T_1$$

with $T_2 = ?\mathsf{double}.?(?\mathsf{string}.T_3).!T_3.T_3)$ and $T_3 = \oplus\left\{\mathsf{accepted}\colon \mathsf{end}, \mathsf{rejected}\colon \mathsf{end}\right\}$.

For the Bank we obtain:

$$\mathsf{Bank} \triangleright \mathsf{req}\ \Gamma_3, \mathit{bank}\colon \mathsf{acc}\ T_2$$

Interestingly, no type structure is needed for the affine adaptations: cancellation is completely transparent. The variation of Seller with an added do-catch, $\mathsf{do}\,\mathsf{req}\,\overline{\mathit{bank}}\,c\,\mathsf{catch}\,\mathsf{req}\,\overline{\mathit{paymate}}\,c$, will simply need $\mathit{paymate}\colon \mathsf{req}\ T_2$ in its interface, i.e., with a type matching that of *bank*, but the original Seller can also be typed in the same way by weakening. Similarly, BuyerMsg has the same interface as Buyer, except that it must include $\mathit{print}\colon \mathsf{req}\ \mathsf{string}$, and again the two processes can be assigned the same interface by weakening, if needed. The processes CheckPriceA, CheckPriceB, and BuyerCancel can be assigned the same type for *seller*, namely $\mathsf{req}\ T_1$, exactly like Buyer.

Finally, as we shall see next affinity does not destroy any of the good properties we expect to obtain with session typing.

5 Properties

Typed terms enjoy the expected soundness properties. In particular we have:

Lemma 1 (Substitution). *If* $P \triangleright \Gamma$ *and* $a \notin \mathit{dom}(\Gamma)$ *then* $Q\{^a/_b\} \triangleright \Gamma\{^a/_b\}$.

Theorem 1 (Subject Reduction). *If* $P \triangleright \Gamma$ *and* $P \longrightarrow Q$ *then* $Q \triangleright \Gamma$.

Proof. The proof relies on several results including Lemma 1. The non-standard case is for cancellations and in particular for $\mathsf{do}\,\rho\,\mathsf{catch}\,P \mid Q$. However, it can be easily checked that the substitution of ρ by P is sound, because both terms offer the same interface except for the canceled session (and dually P can be thrown away in standard communication).

Theorem 2 (Diamond property). *If* $P \triangleright \Gamma$ *and* $Q_1 \longleftarrow P \longrightarrow Q_2$ *then either* $Q_1 \equiv Q_2$ *or* $Q_1 \longrightarrow R \longleftarrow Q_2$.

Proof. The result is actually easy to establish, since the only critical pairs arise from multiple requests to the same replication or to the same cancellation. However, even in that case the theorem holds because: a) replications are immediately available and functional (*uniform availability*); b) cancellations are persistent. The fact that $a\not\downarrow$ can never be assigned $\mathsf{req}\ T$ simplifies the proof.

The above strong confluence property indicates that our sessions are completely deterministic, even considering the possible orderings of requests.

Progress

Our contribution to the theory of session types is *well-behaved affinity*, in the sense that we can guarantee that any session that ends prematurely will not affect the quality of a program. Indeed, if we simply allowed unrestricted weakening,

for example by a type rule $\Gamma \vdash \mathbf{0}$ as done in [13], but without any apparatus at the language level, it would be easy to type a process such as $(\boldsymbol{\nu} a)\overline{a}b.P \mid b(x).Q$ and clearly not only a but also b would be stuck for ever. In this section we prove that this never happens to a well-typed term.

Let us write λ for a prefix ρ that is not a request, i.e., such that $\rho \neq \mathsf{req}\,\overline{a}b.P$.

Proposition 1. *If $P \triangleright \Gamma$ and $P \equiv (\boldsymbol{\nu}\widetilde{a})(H[\,\lambda\,] \mid Q)$ and $\mathsf{subject}(\lambda) = b$ and $b \in \mathsf{fn}(Q)$ then $b \in \widetilde{a}$.*

This is proved very easily by induction on typing derivations.

We now define a notion of permanently blocked process, which intuitively is a process that cannot proceed in *any* context, either because of deadlock or because of (restricted) sessions without a dual. We will use the fact that linear communications are always under the corresponding bound name, from Proposition 1. As usual $P \not\longrightarrow$ means that P cannot reduce.

Definition 1 (Blocked process). *A process P is* blocked *if $P \not\longrightarrow$ and:*

$$P \equiv (\boldsymbol{\nu}\widetilde{a}\colon \widetilde{T})\,(H_1[\,\rho_1\,] \mid \cdots \mid H_n[\,\rho_n\,]) \qquad n \geq 1 \quad \forall i \in \{1,\ldots,n\}\,.\,\mathsf{subject}(\rho_i) \in \{\widetilde{a}\}$$

We give some examples to clarify the definition:

i) $\mathsf{req}\,\overline{a}x.\mathsf{req}\,\overline{b}y.P \mid \mathsf{req}\,\overline{b}r.\mathsf{req}\,\overline{a}k.Q$ is not considered blocked since it can reduce properly if we add the appropriate replications.
ii) $(\boldsymbol{\nu} a)(\mathsf{req}\,\overline{a}x.\mathsf{req}\,\overline{b}y.P \mid \mathsf{req}\,\overline{b}r.\mathsf{req}\,\overline{a}k.Q)$ is not considered blocked because we can add a replication $\mathsf{acc}\,b(x).R$ and it will perform one step (before becoming blocked).
iii) both $(\boldsymbol{\nu} a)(\mathsf{req}\,\overline{a}x.\mathsf{req}\,\overline{b}y.P \mid \mathsf{req}\,\overline{a}k.\mathsf{req}\,\overline{b}r.Q)$ and $(\boldsymbol{\nu} a, b)(\mathsf{req}\,\overline{a}x.\mathsf{req}\,\overline{b}y.P \mid \mathsf{req}\,\overline{b}r.\mathsf{req}\,\overline{a}k.Q)$ are blocked, and indeed they have no chance of reducing.
iv) $a(x).\overline{b}y.P \mid b(z).\overline{a}r.Q$ is not considered blocked, even if it can never reduce, but from Proposition 1 we don't need to consider this case.
v) $(\boldsymbol{\nu} a, b, c)(a(x).\overline{b}y.P \mid b(z).\overline{c}r.Q \mid c(s).\overline{a}e.R)$ is blocked; there is a cycle spanning all three sub-processes.
vi) $(\boldsymbol{\nu} a)\overline{a}b.P$ and $(\boldsymbol{\nu} a, b)(\overline{a}b.P \mid b(x).Q)$ are blocked; this is "bad" affinity.
vii) The above examples can be extended with do-catch, which explains the H-contexts in the definition.

We can now present the main result:

Theorem 3 (Progress). *If $P \triangleright \Gamma$ then:*

a) for all Q, $C[\ ]$ such that $P \equiv C[\,Q\,]$, Q is not blocked.
b) if $P \not\longrightarrow$, then either $P \equiv \mathbf{0}$ or there exists Q, $\widetilde{a}$, Δ, Θ with $Q \triangleright \Delta$ and $Q \not\longrightarrow$, such that $(\boldsymbol{\nu}\widetilde{a})(P \mid Q) \triangleright \Theta$ and $(\boldsymbol{\nu}\widetilde{a})(P \mid Q) \longrightarrow$.

Part *a*) is shown by Theorem 1 and with the help of a lemma: if $P \triangleright \Gamma$ then P is not blocked. Part *b*), which is similar to the formulation in [9], is shown by induction on the type derivation.

The theorem is actually very strong, since it holds also for requests, contrary to related works such as [9] where terms enjoy progress only for the linear part

of sessions, i.e., where a request (that may never be activated) can permanently disable any sessions that depend on it.

Moreover, notice that b) in itself is not enough: we could have a blocked subterm in parallel to a request $\mathsf{req}\ \overline{a}_1 b.P'$, then we could iterate compositions with forwarders $\mathsf{acc}\ a_i(x).\mathsf{req}\ \overline{a}_{i+1}x$ and there would always be a reduction. In general, the existence of a "good" Q does not exclude a "bad" one that leads to deadlock. However, from a) we know that there is no subterm that is blocked.

Also, we have not considered circular dependencies for replications; it is easy to check that they cannot lead to deadlock, and actually they cannot be typed.

Finally, we have checked that typed processes are strongly normalising, which is not so surprising since we followed closely the logical principles of Affine Logic. We leave the complete proof of this result, which uses the technique of Reducibility Candidates [12] in conjunction with Theorem 2, to a longer version. Note that Progress (Theorem 3) is in a sense more important, for two reasons: first, a system without progress can still be strongly normalising, since blocked terms are by definition irreducible; second, practical systems typically allow divergence, and in that case the progress property (which we believe can be transferred without surprises to this setting) becomes much more relevant.

6 Related work and future plans

We divide our discussion on the related work in three parts: relaxing linearity in session types, dealing with exceptional behaviour, and logical foundations.

The study of language constructs for exceptional behavior (including exceptional handling and compensation handling) has received significant attention; we refer the reader to a recent overview [10], while concentrating on those works more closely related to ours. Carbone at al. are probably the first to introduce exceptional behaviour in session types [7]. They do so by extending the programming language (the π-calculus) to include a throw primitive and a try-catch process. The language of types is also extended with an abstraction for a try-catch block: essentially a pair of types describing the normal and the exceptional behaviour. The extensions allow communication peers to escape, in a *coordinated* manner, from a dialogue and reach another point from where the protocol may progress. Carbone [6] and Capecchi et al. [5] port these ideas to the multi-party setting. Hu et al. present an extension of multi-party session types that allow to specify conversations that may be interrupted [16]. Towards this end, an interruptible type constructor is added to the type language, requiring types that govern conversations to be designed with the possible interrupt points in mind. In contrast, we propose a model where programs with and without exceptional behaviour are governed by the same (conventional) types, as it is the norm in functional and object-oriented programming languages.

Caires et al. proposed the conversation calculus [23]. The model introduces the idea of conversation context, providing for a simple mechanism to locally handle exceptional conditions. The language supports error recovery via throw and try-catch primitives. No type abstraction is proposed.

Contracts take a different approach by using process-algebra languages [8] or labeled transition systems [3] for describing the communication behaviour of processes. In contrast to session types, where client-service compliance is given by a symmetric duality relation, contracts come equipped with an asymmetric notion of compliance usually requiring that a client and a service reach a successful state. In these works it is possible to end a session (usually on the client side only) prematurely, but there is no mechanism equivalent to our cancellation, no relationship with exception handling, and no clear logical foundations.

Caires and Pfenning gave a Curry-Howard correspondence relating Intuitionistic Linear Logic and session types in a synchronous π-calculus [4]. Although we do not use their types, there is a clear correspondence between $!T_1.T_2$ and $[\![\overline{T_1}]\!] \otimes [\![T_2]\!]$, and similarly for input. The splitting of the term when composing session endpoints, in our case with (Res), which is standard from [1], was never used in sessions before the work [4]. For output (and request) we followed a different but equivalent approach in which b in $\overline{a}b.P$ is free, when in [4] it would be restricted to appear strictly under P. In fact, we did not change anything compared to the usual output rule [15], which shows that a logical system can be obtained from a standard session system simply by an adaptation of (Res) so that it plays the role of a logical "cut."

Indeed, the system we presented can be mildly adapted to obtain an embedding of typed processes to proofs of Affine Logic. In any case, our formulation allows to type more processes than Linear Logic interpretations, and to our knowledge it is the first logical account of exceptions in sessions, based on an original interpretation of weakening. Moreover, Propositional Affine Logic is decidable, a result by Kopylov [17], so there are better prospects for type inference.

As part of future work, we would like to develop an algorithmic typing system, along the lines of [22]. We also believe it would be interesting to apply our technique to multiparty sessions [14] based on Proof Nets [20].

Acknowledgments This work was supported by FCT through funding of MULTICORE project, ref. PTDC/EIA-CCO/122547/2010, and LaSIGE Strategic Project, ref. PEst-OE/EEI/UI0408/2014. We would like to thank the anonymous reviewers and also Nobuko Yoshida, Hugo Torres Vieira, Francisco Martins, and the members of the Gloss group in the University of Lisbon, for their detailed and insightful comments.

References

1. Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.
2. Andrea Asperti and Luca Roversi. Intuitionistic light affine logic. *ACM Transactions on Compututational Logic*, 3(1), 2002.
3. Mario Bravetti and Gianluigi Zavattaro. Contract-based discovery and composition of web services. In *SFM*, volume 5569 of *LNCS*, pages 261–295. Springer, 2009.
4. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

5. Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. In *FSTTCS*, LIPIcs, pages 338–351. Schloss Dagstuhl, 2010.
6. Marco Carbone. Session-based choreography with exceptions. In *PLACES*, volume 241 of *ENTCS*, pages 35–55. Elsevier, 2009.
7. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
8. Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems*, 31(5):1–61, 2009.
9. Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On progress for structured communications. In *TGC'07*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.
10. Carla Ferreira, Ivan Lanese, António Ravara, Hugo Torres Vieira, and Gianluigi Zavattaro. *Results of the SENSORIA Project 2011*, volume 6582 of *LNCS*, chapter Advanced Mechanisms for Service Combination and Transactions, pages 302–325. Springer, 2011.
11. Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
12. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
13. Marco Giunti. Algorithmic type checking for a pi-calculus with name matching and session types. *The Journal of Logic and Algebraic Programming*, 82(8):263–281, 2013.
14. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.
15. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
16. Raymond Hu, Rumyana Neykova, Nobuko Yoshida, and Romain Demangeon. Practical interruptible conversations: Distributed dynamic verification with session types and Python. In *RV*, volume 8174 of *LNCS*, pages 148–130. Springer, 2013.
17. A.P Kopylov. Decidability of linear affine logic. *Information and Computation*, 164(1):173 – 198, 2001.
18. Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
19. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1), 1992.
20. Dimitris Mostrous. Multiparty sessions based on proof nets. In *Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, 2014.
21. Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE '94*, pages 398–413. Springer, 1994.
22. Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.
23. Hugo T. Vieira, Luís Caires, and João C. Seco. The conversation calculus: A model of service-oriented computation. In *ESOP*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.