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 communications implemented by channels. In particular, they prescribe the sequence of communications, whether they are input or output actions, and the type of value exchanged. Crucial to any language with session types is the notion of linearity, which is essential to ensure that channels exhibit the behaviour prescribed by their type without interference in the presence of concurrency. In this work 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* [24]. *Session types* [16] 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 receive a value of type bool, and nothing more. This type can be materialised by the π-calculus [20] process $P_1 \doteq \overline{a}5.\overline{a}\text{“hello”}.a(x).\mathbf{0}$. The dual of the previous type is $?\mathsf{nat}.?\mathsf{string}.!\mathsf{bool}.\mathsf{end}$, and can be implemented by $P_2 \doteq b(x).b(y).\overline{b}(x+1 < 2).\mathbf{0}$. To compose two processes and enable them to communicate, we use a *double binder* [25]. For the above example, we can write $(\boldsymbol{\nu} ab)(P_1 \mid P_2)$, indicating that a and b are the two *endpoints* of the same channel. The double binder guides reduction, so that we have $(\boldsymbol{\nu} ab)(P_1 \mid P_2) \longrightarrow (\boldsymbol{\nu} ab)(\overline{a}\text{“hello”}.a(x).\mathbf{0}) \mid b(y).\overline{b}(5+1 < 2).\mathbf{0}$. In a well-typed term, the endpoints of a channel must have complementary (or *dual*) types, so that an input on one will match an output on the other, and vice versa. This is the case for a and b, above.

Beyond the basic input/output types, sessions typically provide constructors for alternative sub-protocols, which are very useful for structured interaction. For example, type

1998 ACM Subject Classification: F.3.2 Semantics of Programming Languages (Process Models), F.4.1 Mathematical Logic (Proof theory).

Key words and phrases: Session typing, Affine logic, pi-calculus.

An extended abstract of this paper appeared in COORDINATION 2014.

$\&\{\mathsf{go}\colon T_1, \mathsf{cancel}\colon T_2\}$ can be assigned to an (external) choice $a \triangleright \{\mathsf{go}.Q_1 \; [] \; \mathsf{cancel}.Q_2\}$, a process that offers the choice go and then Q_1 or cancel and then Q_2. The dual type, where $\overline{T}$ denotes T with an alternation of all constructors, is $\oplus\left\{\mathsf{go}\colon \overline{T_1}, \mathsf{cancel}\colon \overline{T_2}\right\}$, and corresponds to a process that will make a (internal) choice, either $\overline{b} \triangleleft \mathsf{go}.R_1$ or $\overline{b} \triangleleft \mathsf{cancel}.R_2$. In the first case the two processes will continue as Q_1 and R_1, respectively.

From Linearity to Affinity. To ensure that sequenced interactions take place in the prescribed order, session typing relies crucially on the notion of *linearity* [13]. However, instead of requiring each endpoint to appear exactly once in a term, which is the standard notion of linearity, session systems only require that an endpoint can interact once at any given moment. Both channel ends a and b in processes P_1 and P_2 are linear in this sense. To see why this condition is required, imagine that we write the first process as $P_1' \doteq \overline{a}5.\mathbf{0} \mid \overline{a}\text{“hello”}.a(x).\mathbf{0}$. Now, a does not appear linearly in P_1' since there are two possible outputs ready to fire. The net effect is that P_2 can receive a “hello” first, which would clearly be unsound and would most likely raise an error in any programming environment. We only relax this condition in one case: two outputs (of the same type) are allowed in parallel when the dual endpoint is a replicated input.

It is because of linearity, as explained above, that sessions can be used to structure protocols with sequences of inputs and outputs, without losing type safety. However, linearity is a rather rigid condition, because it demands that everything in the description of a session type *must* be implemented by an endpoint with that type. In 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 issue. In technical terms, we relax the condition of linearity to that of *affinity*, so that endpoints can perform less interactions than the ones prescribed by their session type. However, a naive introduction of affinity can leave programs in a stuck state: let us re-write P_1 into $P_1'' \doteq \overline{a}5.\overline{a}\text{“hello”}.\mathbf{0}$, i.e., without the final input $a(x)$; then, after two communications process $(\boldsymbol{\nu}ab)(P_1'' \mid P_2)$ will be stuck trying to perform the output $\overline{b}(5+1<2).\mathbf{0}$. We want to be able to perform only an initial part of a session, but we also want to ensure that processes do not get stuck waiting for communications that will never take place. Our solution is to introduce a new kind of communication action written $a\lightning$, which reads *cancel a*. This action is used to explicitly signal that a session has finished, so that communications on the other endpoint can also be cancelled and computation can proceed. For example, we can replace P_1 with $P_1^c \doteq \overline{a}5.\overline{a}\text{“hello”}.a\lightning$, and after two steps $(\boldsymbol{\nu}ab)(P_1^c \mid P_2)$ becomes $(\boldsymbol{\nu}ab)(a\lightning \mid \overline{b}(5+1<2).\mathbf{0})$, which reduces (modulo structural equivalence) to $\mathbf{0}$.

Our development is inspired by Affine Logic, the variation of Linear Logic with unrestricted weakening. The work by Asperti [2], which studies Proof Nets for Affine Logic, shows that weakening corresponds to an actual connective with specific behaviour. In particular, this connective performs the weakening step by step, progressing through the dependencies of a proof, and removing all that must be removed. This is exactly what $a\lightning$ represents.

We take the idea of affinity 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, $(\boldsymbol{\nu}ab)(\mathsf{do}\ \overline{a}(5+1<2).\mathbf{0}\ \mathsf{catch}\ P \mid b\lightning)$ will result in the *replacement* of $\overline{a}(5+1<2).\mathbf{0}$ with the exception handler P. Note that a do-catch is not the same as the try-catch commonly found

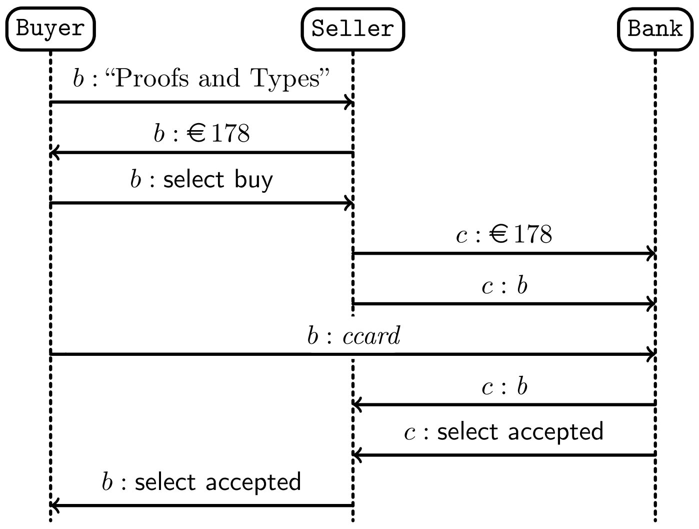

Figure 1: Sequence Diagram for Succesful Book Purchase

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, $(\boldsymbol{\nu} ab)(\mathsf{do}\ \overline{a}(5+1<2).\mathbf{0}\ \mathsf{catch}\ P \mid b(x).\mathbf{0})$ becomes $\mathbf{0}$, because the communication was successful.

The outline of the rest of the paper is as follows. The next section presents affine sessions in action. Section 3 introduces the calculus of affine sessions, Section 4 its typing system, and Section 5 the main properties. Section 6 discusses related works and future plans. The appendix contains the proof of the Subject Reduction theorem.

2. Affine Sessions by Example

We describe a simple interaction comprising three processes—Buyer, Seller, and Bank—that implements a book purchase. The buyer sends the title of a book, receives the price, and chooses either to buy or to cancel. If the buyer decides to buy the book, the credit card information is sent over the session, and the 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 session terminology [16]. Dually, an output that activates a replication is written $\mathsf{req}\,\overline{a}b.P$, and is called a "request." Channels are described by two distinct identifiers, denoting their two endpoints and introduced by $(\boldsymbol{\nu} ab)P$ [25].

We use some standard language constructs that can be easily encoded in π-calculus, such as $\overline{a}e$ for the output of the value obtained by evaluating the expression e, and $\mathsf{if}\,t\,\mathsf{then}\,P\,\mathsf{else}\,Q$ for a conditional expression. The latter is an abbreviation of a new session $(\boldsymbol{\nu} ab)(a \triangleright \{\mathsf{true}.P \;[]\; \mathsf{false}.Q\} \mid R)$ where R represents the test t and evaluates to $\overline{b} \triangleleft \mathsf{true}$ or $\overline{b} \triangleleft \mathsf{false}$. An implementation of the interaction in Figure 1 is:

$$(\boldsymbol{\nu} seller_1 seller_2, bank_1 bank_2)(\,\mathsf{Buyer} \mid \mathsf{Seller} \mid \mathsf{Bank}\,)$$

where:

$$\mathsf{Buyer} \doteq (\boldsymbol{\nu} bb')\left(\begin{array}{l}\mathsf{req}\ \overline{seller_1}\, b' \mid \overline{b}\,\text{“Proofs and Types”}.b(price).\mathsf{if}\ price < 200\\ \mathsf{then}\ \overline{b} \lhd \mathsf{buy}.\overline{b}\, ccard.b \rhd \{\mathsf{accepted}.P \mathbin{[\!]} \mathsf{rejected}.Q\}\ \ \mathsf{else}\ \overline{b} \lhd \mathsf{cancel}\end{array}\right)$$

$$\mathsf{Seller} \doteq \mathsf{acc}\ seller_2(b).\left(\begin{array}{l} b(prod).\overline{b}\, price(prod).\\ b \rhd \{\ \mathsf{buy}.(\boldsymbol{\nu} kk')(\mathsf{req}\ \overline{bank_1}\, k' \mid \overline{k}\, price(prod).\overline{k}b.k(b').\\ \qquad k \rhd \{\mathsf{accepted}.\overline{b}' \lhd \mathsf{accepted} \mathbin{[\!]} \mathsf{rejected}.\overline{b}' \lhd \mathsf{rejected}\})\\ \quad \mathbin{[\!]}\ \mathsf{cancel}.\mathbf{0}\}\end{array}\right)$$

$$\mathsf{Bank} \doteq \mathsf{acc}\ bank_2(k).\left(\begin{array}{l} k(amount).k(b).b(card).\overline{k}b.\\ \mathsf{if}\ charge(amount, card)\ \mathsf{then}\ \overline{k} \lhd \mathsf{accepted}\ \mathsf{else}\ \overline{k} \lhd \mathsf{rejected}\end{array}\right)$$

First we note how sessions are established. For example, in Buyer fresh channel end b' is sent to Seller via the request $\mathsf{req}\ \overline{seller_1}\, b'$, while the other end, b, is kept in the Buyer for further interaction. The identifiers b and b' are the two endpoints of a session, and it is easy to check that the interactions match perfectly. Another point is the *borrowing* of the session b from Seller to Bank, with subprocess $\overline{k}b.k(b')$ at the Seller process, and $k(b).b(card).\overline{k}b$ at the Bank, so that the credit card information 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. In this case, the seller would provide an alternative payment provider. Concretely, we can substitute $\mathsf{req}\ \overline{bank_1}\, k'$ in Seller with $\mathsf{do}\ \mathsf{req}\ \overline{bank_1}\, k'\ \mathsf{catch}\ \mathsf{req}\ \overline{paymate}\, k'$, so that a failure to use the bank service (triggered by $bank_2 ↯$) will activate $\mathsf{req}\ \overline{paymate}\, k'$ and the protocol has a chance to complete successfully.

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

$$\mathsf{BuyerMsg} \doteq (\boldsymbol{\nu} bb')\left(\begin{array}{l}\mathsf{req}\ \overline{seller_1}\, b' \mid \overline{b}\,\text{“Proofs and Types”}.b(price).\\ \mathsf{if}\ price < 200\ \mathsf{then}\ \overline{b} \lhd \mathsf{buy}.\overline{b}\,\mathsf{ccard}.\\ \quad \mathsf{do}\ b \rhd \{\mathsf{accepted}.P \mathbin{[\!]} \mathsf{rejected}.Q\}\\ \quad \mathsf{catch}\ \mathsf{req}\ \overline{print}\text{“An error occurred”}\\ \mathsf{else}\ \overline{b} \lhd \mathsf{cancel}\end{array}\right)$$

As mentioned in the Introduction, a do-catch on a given communication does not catch subsequent cancellations. For instance, if in the above example the do-catch was placed around $\overline{b}$“Proofs and Types”, then any $b' ↯$ generated *after* this output has been read would be uncaught, since $\mathsf{req}\ \overline{print}$“An error occurred” would have been already discarded. However, a do-catch does catch cancellations emitted *before* the point of definition, so it should be placed near the end of a protocol if we just want a single exception handler that catches everything. In general, our mechanism is very fine-grained, and a single session can have multiple, nested do-catch on crucial points of communication and with distinct alternative behaviours.

Note also that cancellation 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 obtain the price of a book and use it in R:

$$\mathsf{CheckPriceA} \doteq (\boldsymbol{\nu} bb')(\mathsf{req}\ \overline{seller_1}\, b' \mid \bar{b}\text{"Principia Mathematica"}.b(price).(\bar{b} \triangleleft \mathsf{cancel} \mid R))$$

$$\mathsf{CheckPriceB} \doteq (\boldsymbol{\nu} bb')(\mathsf{req}\ \overline{seller_1}\, b' \mid \bar{b}\text{"Introduction to Metamathematics"}.b(price).(b\lightning \mid R))$$

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 this is impossible because such behavior is not *predicted* by the session type; with cancellation it is extremely simple, as shown below.

$$\mathsf{BuyerCancel} \ \doteq\ (\boldsymbol{\nu} bb')(\mathsf{req}\ \overline{seller_1}\, b' \mid \bar{b}\text{"Tractatus Logico-Philosophicus"}.b(price).\bar{b} \triangleleft \mathsf{buy}.b\lightning)$$

3. The Process Calculus of Affine Sessions

This section introduces our language, its syntax and operational semantics.

Syntax. The language we work with, shown in Figure 2, is a small extension of standard π-calculus [20]. We rely on a denumerable set of *variables*, denoted by lower case roman letters. As for processes, instead of the standard restriction $(\boldsymbol{\nu} a)P$, we use double binders [25] in the form $(\boldsymbol{\nu} ab)P$, which are similar to *polarities* [12], and enable syntactically distinguishing the two endpoints of a session. For technical convenience we shall consider all indexing sets I to be non-empty, finite, and 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, written $\mathsf{do}\ \rho\ \mathsf{catch}\ P$.

Parentheses introduce the *bindings* in the language: variable x is bound in processes $a(x).P$ and $\mathsf{acc}\, a(x).P$; both variables x and y are bound in process $(\boldsymbol{\nu} xy)P$. The notions of free and bound variables as well as that of substitution (of x by a in P, notation $P\{^a/_x\}$) are defined accordingly. We follow Barendregt's variable convention, whereby all variables in binding occurrences in any mathematical context are pairwise distinct and distinct from the free variables.

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 (with unit $\mathbf{0}$), the usual laws for scope extrusion, and satisfies the axioms below. (For the complete set of axioms with double binders, see [25]).

$$(\boldsymbol{\nu} ab)P \equiv (\boldsymbol{\nu} ba)P \qquad a\lightning \mid a\lightning \equiv a\lightning \qquad (\boldsymbol{\nu} ab)(a\lightning \mid b\lightning) \equiv \mathbf{0} \qquad (\boldsymbol{\nu} ab)a\lightning \equiv \mathbf{0}$$

The first axiom is needed for reduction; the second is needed for soundness; the remaining two are not strictly necessary but they allow to throw away garbage processes, specifically sessions that are fully cancelled.

From now on, in all contexts (notably reduction, typing, proofs) we shall consider processes up to structural equivalence; this is especially useful in typing. Note that $\mathsf{acc} a(x).P \not\equiv P \mid \mathsf{acc}\, a(x).P$, i.e., we did not add the axiom for replication found in many presentations of π-calculus. We made this choice because adding the axiom would put to question the decidability of $\equiv$ [19], and consequently of typing.

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

Figure 2: Syntax

$$
\begin{array}{rcll}
(\boldsymbol{\nu} ab)(H_1[\,\overline{a}c.P\,] \mid H_2[\,b(x).Q\,]) & \longrightarrow & (\boldsymbol{\nu} ab)(P \mid Q\{^c/_x\}) & \textsf{(R-Com)} \\
(\boldsymbol{\nu} ab)(H_1[\,\overline{a} \triangleleft l_k.P\,] \mid H_2[\,b \triangleright \{l_i.Q_i\}_{i\in I}\,]) & \longrightarrow & (\boldsymbol{\nu} ab)(P \mid Q_k) \qquad (k \in I) & \textsf{(R-Bra)} \\
(\boldsymbol{\nu} ab)(H[\,\mathsf{req}\ \overline{a}c.P\,] \mid \mathsf{acc}\ b(x).Q \mid R) & \longrightarrow & (\boldsymbol{\nu} ab)(P \mid Q\{^c/_x\} \mid \mathsf{acc}\ b(x).Q \mid R) & \textsf{(R-Ses)} \\
P \longrightarrow Q & \Rightarrow & P \mid R \longrightarrow Q \mid R & \textsf{(R-Par)} \\
P \longrightarrow Q & \Rightarrow & (\boldsymbol{\nu} ab)P \longrightarrow (\boldsymbol{\nu} ab)Q & \textsf{(R-Res)}
\end{array}
$$

Figure 3: Standard Reductions

Reduction. Do-catch contexts allow for possible exception handling.

$$H ::= [\,] \mid \mathsf{do}\ [\,]\ \mathsf{catch}\ P$$

Notation $H[P]$ denotes the process obtained by filling the hole $[\,]$ in context H with process P, as usual.

Reduction is defined in two parts: the standard rules (Figure 3), and the cancellation rules (Figure 4). First, recall that we work up to structural equivalence, which means we do not explicitly state that $P \equiv P' \longrightarrow Q' \equiv Q \Rightarrow P \longrightarrow Q$, but of course it holds. In standard reductions, the only notable point is that we discard any do-catch handlers, since there is no cancellation, which explains why the H-contexts disappear. For example, $(\boldsymbol{\nu} ab)(\mathsf{do}\ \overline{a}c.P\ \mathsf{catch}\ Q \mid b(x).R) \longrightarrow (\boldsymbol{\nu} ab)(P \mid R\{^c/_x\})$. The type system ensures that it is sound to discard Q, since it implements the same sessions as P as well as the session on c. On the other hand, a cancellation activates a handler, which may provide some default values to a session, completing it or eventually re-throwing a cancellation. For example, notice how c appears in the handler when a is cancelled in $(\boldsymbol{\nu} ab)((\mathsf{do}\ \overline{a}c.P\ \mathsf{catch}\ \overline{c}5.c\lightning) \mid b\lightning) \longrightarrow (\boldsymbol{\nu} ab)(\overline{c}5.c\lightning \mid b\lightning) \equiv \overline{c}5.c\lightning$.

Our cancellation reductions are inspired by cut-elimination for weakening in Proof Nets for Affine Logic (see [2]). Specifically, $a\lightning$ behaves like a weakening (proof net) connective which consumes progressively everything it interacts with (in logic this happens with cut).

$$
\begin{array}{rcllr}
(\boldsymbol{\nu} ab)(\mathsf{acc}\, a(x).P \mid b\lightning \mid R) & \longrightarrow & (\boldsymbol{\nu} ab)(\mathsf{acc}\, a(x).P \mid R) & & \textsf{(C-Acc)}\\
(\boldsymbol{\nu} ab)(\mathsf{req}\, \overline{a}c.P \mid b\lightning \mid R) & \longrightarrow & (\boldsymbol{\nu} ab)(P \mid b\lightning \mid c\lightning \mid R) & & \textsf{(C-Req)}\\
(\boldsymbol{\nu} ab)(\overline{a}c.P \mid b\lightning) & \longrightarrow & (\boldsymbol{\nu} ab)(P \mid b\lightning \mid c\lightning) & & \textsf{(C-Out)}\\
(\boldsymbol{\nu} ab)(a(x).P \mid b\lightning) & \longrightarrow & (\boldsymbol{\nu} ab)(\boldsymbol{\nu} xy)(P \mid b\lightning \mid y\lightning) & & \textsf{(C-Inp)}\\
(\boldsymbol{\nu} ab)(\overline{a} \triangleleft l_k.P \mid b\lightning) & \longrightarrow & (\boldsymbol{\nu} ab)(P \mid b\lightning) & & \textsf{(C-Sel)}\\
(\boldsymbol{\nu} ab)(a \triangleright \{l_i.P_i\}_{i\in I} \mid b\lightning) & \longrightarrow & (\boldsymbol{\nu} ab)(P_k \mid b\lightning) & \mathsf{max}(I) = k & \textsf{(C-Bra)}\\
(\boldsymbol{\nu} ab)(\mathsf{do}\, \rho\, \mathsf{catch}\, P \mid b\lightning \mid R) & \longrightarrow & (\boldsymbol{\nu} ab)(P \mid b\lightning \mid R) & \mathsf{subject}(\rho) = a & \textsf{(C-Cat)}
\end{array}
$$

Figure 4: Cancellation Reductions

For example, using $(\mathsf{C{-}Inp})$ we can perform $(\boldsymbol{\nu} ab)(a(x).\overline{x}c \mid b\lightning) \longrightarrow (\boldsymbol{\nu} ab)(\boldsymbol{\nu} xy)(\overline{x}c \mid y\lightning \mid b\lightning)$ and then by $(\mathsf{C{-}Out})$ we obtain $(\boldsymbol{\nu} ab)(\boldsymbol{\nu} xy)(\overline{x}c \mid y\lightning \mid b\lightning) \longrightarrow (\boldsymbol{\nu} ab)(\boldsymbol{\nu} xy)(b\lightning \mid y\lightning \mid c\lightning) \equiv c\lightning$.

In the cancellation of branching, $(\mathsf{C{-}Bra})$, we choose the maximum index k which exists given our assumption that index sets are non-empty and totally ordered. This is a simple way to avoid non-determinism via cancellation, i.e., to ensure that cancellation does not break confluence.

In the rule $(\mathsf{C{-}Cat})$, 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 prefixes ρ. If ρ happens to be a request $\mathsf{req}\ \overline{a}c.Q$, then $b\lightning$ plays the role of an accept. This explains why $b\lightning$ remains in the result: like an accept, it must be replicated to deal with possibly multiple requests in P and R.

The rule $(\mathsf{C{-}Acc})$ is not strictly necessary for computation. It simply reinforces the fact that a request does not cancel an accept, a fact that may not be as obvious if we simply do not have a reduction for this case. Moreover, it is important to define how cancellation interacts with all constructors.

In the cancellation reductions $(\mathsf{C{-}Acc/Req/Cat})$, R represents the remaining scope of a and b, so in the general case we should have it also in $(\mathsf{C{-}Out/Inp/Sel/Bra})$. However, the typing system guarantees that in these cases both a and b are linear, and therefore cannot appear elsewhere, so we preferred to keep the rules simpler. The same reasoning applies to the R in the standard reductions; only $(\mathsf{R{-}Ses})$ needs it.

In rule $(\mathsf{C{-}Inp})$, variable y is not free in P, a fact that results from the variable convention.

4. Typing affine sessions

This section introduces our notion of types and the typing system. It motivates our choices and discusses the typing of the running example.

Types. The session types we use, shown in Figure 5, are based on the constructs of Honda et al. [16] with two exceptions. First, following Vasconcelos [25] we allow a linear type to evolve into a shared type. Second, following the concept of Caires and Pfenning [4] we decompose shared types into accept types $\mathsf{acc}\ T$ and request types $\mathsf{req}\ T$. Technically, $\mathsf{acc}\ T$ corresponds to $!T$ ("of course T") and $\mathsf{req}\ T$ to $?\overline{T}$ ("why not $\overline{T}$") from Linear Logic [13].

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

Figure 5: Session Types

Duality. The two ends of a session can be 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.[1]

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

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

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

Interfaces (or typing contexts). We use Γ, Δ, and Θ to range over interfaces, unordered lists of entries of the form $a\colon T$. We note that processes can have multiple uses of $a\colon \mathsf{req}\ T$, which corresponds to the logical principle of contraction; this is the only kind of entry that can appear multiple times in a context. In this way, formation of contexts requires that $T = U = \mathsf{req}\ V$ for contexts $\Gamma, a\colon T$ when $a\colon U \in \Gamma$. Henceforth, we assume all contexts are of this form, so that, whenever we write $\Gamma, a\colon T$, then it must be the case that if a is in Γ, then its type is a request type equal to T.

To simplify the presentation, we identify interfaces up to permutations, so we do not need to define a type rule for the exchange of entries. We also use a pair of abbreviations: $\mathsf{req}\,\Gamma$ stands for an interface of the shape $a_1\colon \mathsf{req}\ T_1, \ldots,\ a_n\colon \mathsf{req}\ T_n$, and similarly, $\mathsf{end}\,\Gamma$ stands for an interface $a_1\colon \mathsf{end}, \ldots, a_n\colon \mathsf{end}$.

Typing rules. Typing judgements take the form:

$$\Gamma \vdash P$$

meaning that process P has interface Γ.

The typing rules are presented in Figure 6. We focus on some key points, noting that a rule can only be applied if the interface of the conclusion is well-formed.

In (Out), an output $\overline{a}b.P$ records a conclusion $b\colon T_1$, so in fact it composes against the dual $b\colon \overline{T_1}$. Therefore $!T_1.T_2$ really means to send a name of type $\overline{T_1}$, which matches with the dual input. The same reasoning applies to requests; see (Req). In (Res) we split the process so that each part implements one of the ends of the session. This is inspired by Caires and

[1]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.

$$\textsf{(Out)}\ \frac{\Gamma, a\colon T_2 \vdash P}{\Gamma, a\colon !T_1.T_2, b\colon T_1 \vdash \overline{a}b.P} \qquad \textsf{(In)}\ \frac{\Gamma, x\colon T_1, a\colon T_2 \vdash P}{\Gamma, a\colon ?T_1.T_2 \vdash a(x).P}$$

$$\textsf{(Sel)}\ \frac{\Gamma, a\colon T_k \vdash P \qquad k \in I}{\Gamma, a\colon \oplus\{l_i\colon T_i\}_{i\in I} \vdash \overline{a} \lhd l_k.P} \qquad \textsf{(Bra)}\ \frac{\forall i \in I\,.\,\Gamma, a\colon T_i \vdash P_i \qquad I \neq \emptyset}{\Gamma, a\colon \&\{l_i\colon T_i\}_{i\in I} \vdash a \rhd \{l_i.P_i\}_{i\in I}}$$

$$\textsf{(Req)}\ \frac{\Gamma \vdash P}{\Gamma, a\colon \textsf{req}\ T, b\colon T \vdash \textsf{req}\ \overline{a}b.P} \qquad \textsf{(Acc)}\ \frac{\textsf{req}\ \Gamma, x\colon T \vdash P}{\textsf{req}\ \Gamma, a\colon \textsf{acc}\ T \vdash \textsf{acc}\ a(x).P}$$

$$\textsf{(Res)}\ \frac{\Gamma_1, a\colon T \vdash P \qquad \Gamma_2, b\colon \overline{T} \vdash Q}{\Gamma_1, \Gamma_2 \vdash (\boldsymbol{\nu} ab)(P \mid Q)}$$

$$\textsf{(Contraction)}\ \frac{\Gamma, a\colon \textsf{req}\ T, a\colon \textsf{req}\ T \vdash P}{\Gamma, a\colon \textsf{req}\ T \vdash P} \qquad \textsf{(Nil)}\ \frac{}{\emptyset \vdash \mathbf{0}} \qquad \textsf{(Weak)}\ \frac{\Gamma \vdash P}{\Gamma, \textsf{req}\ \Delta, \textsf{end}\ \Theta \vdash P}$$

$$\textsf{(Catch)}\ \frac{\Gamma, a\colon T \vdash \rho \qquad \Gamma \vdash P \qquad \textsf{subject}(\rho) = a}{\Gamma, a\colon T \vdash \textsf{do}\ \rho\ \textsf{catch}\ P} \qquad \textsf{(Cancel)}\ \frac{}{a\colon T \vdash a\lightning}$$

Figure 6: Affine Session Typing

Pfenning [4] which interprets sessions as propositions in a form of Intuitionistic Linear Logic; the notion of "cut as composition under name restriction" comes from Abramsky [1].

In (Cancel), $a\lightning$ can be given any type. A do-catch process is typed using rule (Catch), 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 cancelled. The rule is sound, since no session is left unfinished, irrespectively of which process we execute, ρ or P. Notice that, if $T = \textsf{req}\ U$, we may have more occurrences of $a\colon T$ in Γ, because of contraction. We made the choice to allow this, since it does not affect any property.[2]

Typing the book purchase example. It is easy to verify that the examples in Section 2 are well-typed. For the Buyer we obtain the following sequent

$$ccard\colon \textsf{string}, seller_1\colon \textsf{req}\ T_1 \vdash \textsf{Buyer}$$

where T_1 abbreviates type $?\textsf{string}.!\textsf{double}.\&\{\textsf{buy}\colon ?\textsf{string}.T_2, \textsf{cancel}\colon \textsf{end}\}$ and T_2 stands for $\oplus\{\textsf{accepted}\colon \textsf{end}, \textsf{rejected}\colon \textsf{end}\}$. The behaviour of b inside process Buyer is described by $\overline{T_1}$.

For the Bank we obtain the following sequent

$$bank_2\colon \textsf{acc}\ T_3 \vdash \textsf{Bank}$$

with $T_3 = ?\textsf{double}.?(?\textsf{string}.T_2).!T_2.T_2$.

[2] On the other hand, adding a premise $a \notin \textsf{dom}(\Gamma)$ in (Catch) would cause problems with subject reduction. For example $(\boldsymbol{\nu} cd)(\overline{c}a \mid d(x).\textsf{do}\ \textsf{req}\,\overline{a}y\ \textsf{catch}\ \textsf{req}\,\overline{x}z)$ would be typable, but it reduces to $\textsf{do}\ \textsf{req}\,\overline{a}y\ \textsf{catch}\ \textsf{req}\,\overline{a}z$ which is not typable.

Finally, for the Seller we have

$$\mathit{bank}_1 \colon \mathsf{req}\ T_3, \mathit{seller}_2 \colon \mathsf{acc}\ T_1 \vdash \mathsf{Seller}$$

Interestingly, no type structure is needed for the affine adaptations: cancellation is completely transparent. The variation of Seller with an added $\mathsf{do\text{-}catch}$:

$$\mathsf{do}\ \mathsf{req}\ \overline{\mathit{bank}_1}\, k'\ \mathsf{catch}\ \mathsf{req}\ \overline{\mathit{paymate}}\, k'$$

will simply need $\mathit{paymate} \colon \mathsf{req}\ T_3$ in its interface, i.e., with a type matching that of bank_1, but the original Seller can also be typed in the same way by using weakening to add an extra assumption to the context. Similarly, $\mathsf{BuyerMsg}$ has the same interface as Buyer, except that it must include $\mathit{print} \colon \mathsf{req\ string}$, and again the two processes can be assigned the same interface by weakening, if needed. Processes $\mathsf{CheckPriceA}$, $\mathsf{CheckPriceB}$, and $\mathsf{BuyerCancel}$ can typed under contexts containing $\mathit{seller}_1 \colon \mathsf{req}\ T_1$, exactly as in process Buyer.

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

Typing modulo structural equivalence. Since we consider processes up to $\equiv$, we have that $P \equiv P'$ and $\Gamma \vdash P'$ implies $\Gamma \vdash P$. This possibility is suggested by Milner [19] and used by Caires and Pfenning [4]. It is necessary because associativity of "$\mid$" does not preserve typability, for example $(\boldsymbol{\nu} ab)(P \mid (Q \mid R))$ may be untypable in the form $(\boldsymbol{\nu} ab)((P \mid Q) \mid R)$. This applies also to scope extrusion $(\boldsymbol{\nu} ab)(P \mid Q)$ and $(\boldsymbol{\nu} ab)P \mid Q$.

Implicit and explicit weakening. The rules (Weak) and (Cancel) implement a form of weakening. In the case of (Weak), this weakening is *implicit*, in the sense that we extend an interface without adding any behaviour at the process level. Specifically, to close a session (introducing $a \colon \mathsf{end}$) or to record that a service is invoked (with $a \colon \mathsf{req}\ T$) weakening is standard. On the other hand, (Cancel) introduces an *explicit* weakening, since we maintain the logical concept of a device to perform it, and this is why, as we shall see, sessions do not get stuck.

Deriving the Mix rule. A common situation is when we want to compose independent processes, i.e., processes that do not communicate. This corresponds to the introduction of Girard's "Mix" rule, which is derived in our system as follows:

$$\overset{(\mathsf{Mix})}{\dfrac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1, \Gamma_2 \vdash P \mid Q}} \quad \doteq \quad \dfrac{\dfrac{\Gamma_1 \vdash P}{\Gamma_1, a \colon \mathsf{end} \vdash P}\,(\mathsf{Weak}) \qquad \dfrac{\Gamma_2 \vdash Q}{\Gamma_2, b \colon \mathsf{end} \vdash Q}\,(\mathsf{Weak})}{\Gamma_1, \Gamma_2 \vdash P \mid Q}\,(\mathsf{Res})$$

Recall that whenever a, b are not free in P and Q, we have $P \mid Q \equiv (\boldsymbol{\nu} ab)(P \mid Q)$.

The need for double binders. In our system, the terms $(\boldsymbol{\nu} ab)(\mathsf{req}\ \overline{a}c.P \mid b\lightning \mid R)$ and $(\boldsymbol{\nu} ab)(\mathsf{req}\ \overline{a}c.P \mid a\lightning \mid R)$ behave differently. In particular, there is a reduction of the first one, using ($\mathsf{C{-}Req}$), but not of the second one where both the request and the cancellation are on the same endpoint, namely a. Both terms are typable, but with different derivations: for the first we need to use (Res) on $a\colon \mathsf{req}\ T$ and $b\colon \mathsf{acc}\ T$, while on the second we apply (Mix) and possibly ($\mathsf{Contraction}$) for $a\colon \mathsf{req}\ T$. However, in a system with the standard scope restriction from π-calculus there are no separate endpoints and therefore the two terms above are identified as $(\boldsymbol{\nu} a)(\mathsf{req}\ \overline{a}c.P \mid a\lightning \mid R\{^a/_b\})$.

Can we apply a cancellation reduction? The two terms above have different dynamics, so there is no solution once we identify them. From the perspective of the typing system, we would not know any more if the type of $a\lightning$ in the last term is $a\colon \mathsf{req}\ T$ or $a\colon \mathsf{acc}\ T$, since there can be different type derivations for each possibility. In the extended abstract of this work [22] we did not employ double-binders, and as a result we had to deal with this ambiguity. Our solution was to impose a rather severe condition on typing, roughly that $a\colon T \vdash a\lightning$ only if T does not contain a subexpression $\mathsf{req}\ T'$. In this way we excluded one of the two possible cases. The introduction of double binders resolves such ambiguities without any restriction on the shape of types.

5. Properties

This section discusses the tree main properties of our system: soundness, confluence, and progress.

Theorem 5.1 (Subject Reduction)**.** *If* $\Gamma \vdash P$ *and* $P \longrightarrow Q$ *then* $\Gamma \vdash Q$.

Proof. See Appendix A. □

Theorem 5.2 (Diamond property)**.** *If* $\Gamma \vdash P$ *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 easy to establish, since the only critical pairs would 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.

Possible critical pairs arise when two reductions overlap (they both use a common subprocess). Let us consider a process $(\boldsymbol{\nu} ab)(H_1[\mathsf{req}\ \overline{a}c_1.P_1] \mid H_2[\mathsf{req}\ \overline{a}c_2.P_2] \mid \mathsf{acc}\ b(x).P_3)$. There are two critical pairs, depending on which request reacts first, resulting in either the process $(\boldsymbol{\nu} ab)(P_1 \mid P_3\{^{c_1}/x\} \mid H_2[\mathsf{req}\ \overline{a}c_2.P_2] \mid \mathsf{acc}\ b(x).P_3)$ or alternatively to the process $(\boldsymbol{\nu} ab)(P_2 \mid P_3\{^{c_2}/x\} \mid H_1[\mathsf{req}\,\overline{a}c_1.P_1] \mid \mathsf{acc}\,b(x).P_3)$. But both reduce in one step to $(\boldsymbol{\nu} ab)(P_1 \mid P_3\{^{c_1}/x\} \mid P_2 \mid P_3\{^{c_2}/x\} \mid \mathsf{acc}\ b(x).P_3)$.

Another possibility is $(\boldsymbol{\nu} ab)(H_1[\mathsf{req}\,\overline{a}c_1.P_1] \mid H_2[\mathsf{req}\,\overline{a}c_2.P_2] \mid b\lightning)$. Again, the critical pair is trivial.

In both cases above, the same reasoning applies if there are more than two possible reductions. □

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 [14], but without any cancellation apparatus at the language level, it would be easy to type a process such as $(\boldsymbol{\nu} xz)((\boldsymbol{\nu} ab)\overline{a}x.P \mid z(y).Q)$ and clearly not only a (and everything in P) but also z (and everything in Q) would be stuck for ever. In this section we prove that this never happens to a well-typed process.

The next notion identifies the set of characteristic processes, conducting a session T on a channel a.

Definition 5.3 (Atomic Action)**.** *The atomic actions are defined by the grammar below.*

$$\alpha ::= H[\,\rho\,] \mid \mathsf{acc}\, a(x).P \mid a\lightning$$

Definition 5.4 (Active and Inactive Processes)**.** *We say that a process P is* inactive*, written* $\mathsf{inactive}(P)$*, if it belongs to the set*

$$\{P \mid \Gamma \vdash P \textit{ and } P \equiv (\boldsymbol{\nu}\vec{cc'}, a_1a_1', \cdots, a_na_n')(\mathsf{acc}\, a_1(x_1).P_1 \mid \cdots \mid \mathsf{acc}\, a_n(x_n).P_n\}$$

Otherwise P is called active*, written* $\mathsf{active}(P)$*.*

The reason for distinguishing inactive processes is that they are typable (and naturally emerge) because an accept can be used zero or more times. For example the process $(\boldsymbol{\nu} ab)(\mathsf{req}\overline{a}c.\mathbf{0} \mid \mathsf{req}\overline{a}r.\mathbf{0} \mid \mathsf{acc} b(x).\mathbf{0})$ is active and can be typed with interface $(c\colon \mathsf{end}, r\colon \mathsf{end})$, but after two steps it reduces to the inactive process $(\boldsymbol{\nu} ab)\mathsf{acc}\, b(x).\mathbf{0}$. To type the last process, we need to transform it to $(\boldsymbol{\nu} ab)(\mathsf{acc}\, b(x).\mathbf{0} \mid \mathbf{0})$ and apply $(\mathsf{Nil}, \mathsf{Weak})$ so as to obtain $a\colon \mathsf{req}\ \mathsf{end} \vdash \mathbf{0}$, then (Res) to obtain the result.

Definition 5.5 (Observation Predicate)**.** *We write $P{\downarrow_a}$ whenever $P \equiv (\boldsymbol{\nu}\vec{bb'})(\alpha \mid P')$ with* $\mathsf{subject}(\alpha) = a$ *and $a \notin \vec{b}, \vec{b'}$. We write $P{\Downarrow_a}$ whenever $P \longrightarrow^* P'$ and $P'{\downarrow_a}$, for some a.*

Below we write $P \not\longrightarrow$ if there does not exist Q such that $P \longrightarrow Q$.

Lemma 5.6. *If $\Gamma \vdash P$ and $P \not\longrightarrow$ and* $\mathsf{active}(P)$*, then $\Gamma = \Gamma', a\colon T$ and $P{\downarrow_a}$.*

Proof. By rule induction on the first premise. □

The assumption $P \not\longrightarrow$ is necessary, since there are active processes that reduce to inactive processes, such as $(\boldsymbol{\nu} cc')(\overline{c}a \mid c'(x))$ or the example under Definition 5.4. Our formulation of the Lemma 5.6 is close in principle to the approach of Dezani-Ciancaglini et al. [10] and is similar to Caires and Pfenning's Lemma 4.3 [23], except that we use the shape of the process (having a top-level action on a) instead of the ability of the process to perform a labelled transition $P \xrightarrow{a}$.

The next notion identifies the set of characteristic processes of the type of any given session on a, following the terminology and main idea of Coppo et al. [9].

Definition 5.7 (Characteristic Process)**.** *The set of processes $(\!|\, a\colon T \,|\!)$ is defined, for each channel end a, inductively on the type T. We take b, b' to be different from a.*

$$\begin{aligned}
(\!|\, a\colon \mathsf{end} \,|\!) &\doteq \{\mathbf{0}\} \\
(\!|\, a\colon !T_1.T_2 \,|\!) &\doteq \big\{(\boldsymbol{\nu} bb')(\overline{a}b'.P \mid Q) : P \in (\!|\, a\colon T_2 \,|\!), Q \in (\!|\, b\colon \overline{T_1} \,|\!)\big\} \\
(\!|\, a\colon ?T_1.T_2 \,|\!) &\doteq \{a(x).(P \mid Q) : P \in (\!|\, a\colon T_2 \,|\!), Q \in (\!|\, x\colon T_1 \,|\!)\} \\
(\!|\, a\colon \oplus\{l_i\colon T_i\}_{i\in I} \,|\!) &\doteq \cup_{i\in I}\,\{\overline{a} \triangleleft l_i.P : P \in (\!|\, a\colon T_i \,|\!)\} \quad (I \neq \emptyset) \\
(\!|\, a\colon \&\{l_i\colon T_i\}_{i\in I} \,|\!) &\doteq \big\{a \triangleright \{l_i.P_i\}_{i\in I} : P \in (\!|\, a\colon T_i \,|\!)\big\} \\
(\!|\, a\colon \mathsf{req}\; T_1 \,|\!) &\doteq \big\{(\boldsymbol{\nu} bb')(\mathsf{req}\, \overline{a}b' \mid P) : P \in (\!|\, b\colon \overline{T_1} \,|\!)\big\} \\
(\!|\, a\colon \mathsf{acc}\; T_1 \,|\!) &\doteq \{\mathsf{acc}\, a(x).P : P \in (\!|\, x\colon T_1 \,|\!)\}
\end{aligned}$$

The clauses $(\!|\, a\colon !T_1.T_2 \,|\!)$ and $(\!|\, a\colon ?T_1.T_2 \,|\!)$ identify sets of characteristic processes which are *non-interleaving*. This means that they do not mix different sessions within the same prefix sequence. Indeed, the reason is that P must continue its evolution on a independently from Q which instead works on b or x. Moreover, all characteristic processes are *complete*, meaning that they provide communication actions for the whole session as defined by the type. For example, any process in $(\!|\, a\colon !\mathsf{end}.?\mathsf{end}.\mathsf{end} \,|\!)$ will perform an output on a and then an input also on a. On the other hand, there are processes such as $\overline{a}b.\overline{c}a$ which can be assigned $\Gamma, a\colon !\mathsf{end}.?\mathsf{end}.\mathsf{end}$ but do not complete the session on a.

Proposition 5.8. *(a) For all a and T, $(\!|\, a\colon T \,|\!)$ is non-empty;*
(b) For all $P \in (\!|\, a\colon T \,|\!)$ we have $a\colon T \vdash P$;
(c) For all $P \in (\!|\, a\colon T \,|\!)$, either $T = \mathsf{end}$ or $P{\downarrow_a}$.

Proof. (a) Immediate from the definition. (b) Easy induction on T. (c) In all cases except for $T = \mathsf{end}$, the characteristic processes $P \in (\!|\, a\colon T \,|\!)$ are of the form α or $(\boldsymbol{\nu} bb')(\alpha \mid R)$ with $\mathsf{subject}(\alpha) = a$, and therefore we have $P{\downarrow_a}$. □

We can now claim a standard progress result, which guarantees that active processes can always perform some interaction.

Corollary 5.9 (Progress)**.** *If $\Gamma \vdash P$ and $P \not\longrightarrow$, then either $\mathsf{inactive}(P)$ or there exists Q, a, a', Δ, Θ with $\Delta \vdash Q$ and $Q \not\longrightarrow$, such that $\Theta \vdash (\boldsymbol{\nu} aa')(P \mid Q)$ and $(\boldsymbol{\nu} aa')(P \mid Q) \longrightarrow$.*

Proof. We focus on the interesting case, which is when the process P is active. From the assumptions and from Lemma 5.6 we know that $\Gamma, a\colon T \vdash P$ and $P{\downarrow_a}$; notice that $T \neq \mathsf{end}$, since this contradicts $P{\downarrow_a}$. But then we can compose with a characteristic process $Q \in (\!|\, a'\colon \overline{T} \,|\!)$ obtaining $\Gamma \vdash (\boldsymbol{\nu} aa')(P \mid Q)$, and clearly $(\boldsymbol{\nu} aa')(P \mid Q) \longrightarrow$ because both P and Q (see Proposition 5.8(c)) are ready to react on a and a', respectively, having actions of dual type. □

At first sight, Corollary 5.9 may seem to allow some deadlocks. For example, it might seem that we could have a deadlocked sub-process R in parallel to a request $\mathsf{req}\,\overline{a}_1 b.P'$, then we could always apply a parallel composition with a forwarder $\mathsf{acc}\, a'_i(x).\mathsf{req}\,\overline{a}_{i+1}x$ (as the Q in Corollary 5.9) and there would always be a reduction step. However, the sub-process R would need to be typed as part of the larger derivation, but then we arrive at a contradiction: by Corollary 5.9, R must be able to perform an action given a suitable context, which contradicts the assumption that it is deadlocked. To prove this formally, it suffices to show that applications of (Res) do not inhibit any action except the restricted one, which is easy to establish.

Discussion. Finally, it can be shown that typed processes are strongly normalising, which is not so surprising since we followed closely the logical principles of Affine Logic. This can be shown by a small adaptation of the standard method [13], first by giving an interpretation of types based on biorthogonals, then by strengthening the induction hypothesis using a notion of reducibility (a contextual test for normalisation), and finally by making use of Theorem 5.2 to obtain strong normalisation from weak normalisation.

Note that Progress (Corollary 5.9) is in a sense more important, for two reasons: first, a system without progress can still be strongly normalising, since blocked processes are by definition irreducible; second, practical systems typically allow recursion, 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 [11], while concentrating on those works more closely related to ours. Carbone et 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 [17]. 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 [26]. 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. In comparison to our work, no type abstraction is proposed since their language is untyped, moreover errors are caught in a context and do not follow and destroy sessions as in our work, and finally their exception mechanism does not guarantee progress.

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]. To avoid deadlocks, we follow the same approach and impose that any two processes can communicate over at most a single session; in our case this in ensured in (Res). Therefore, both systems reject processes of the shape $(\boldsymbol{\nu} aa', bb')(a(x).\bar{b}c \mid b'(y).\overline{a'}r)$ which are stuck. This restriction is founded on logical cut and was first introduced into a process algebra by Abramsky [1]. Caires and Pfenning [4] achieve a progress property similar to Corollary 5.9 in our work, but only in the more rigid setting of linearity. Adding affinity to their system, without a mechanism similar to our cancellation, would allow sessions to get stuck. Our formulation allows to type more processes than Linear Logic interpretations, such as BuyerCancel from Section 1 and the alternative form of choice shown in Section 4. Moreover, to our knowledge our work is the first logical account of exceptions in sessions, based on an original interpretation of weakening. Finally, Propositional Affine Logic is decidable, a result by Kopylov [18], so we face better prospects for type inference.

As part of future work, we would like to develop an algorithmic typing system, along the lines of [25]. We also believe it would be interesting to apply our technique to multiparty session types [15] based on Proof Nets [21]. Finally, we plan to study the Curry-Howard correspondence with Affine Logic in depth, and examine more primitives that become possible with our mechanism.

Acknowledgement

We would like to thank the anonymous reviewers of COORDINATION'14 and of LMCS, and also Nobuko Yoshida, Hugo Torres Vieira, Francisco Martins, and the members of the Gloss group at 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 Computational 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] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 760:1–65, 2015.

[10] 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.

[11] 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.

[12] Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[13] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[14] 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.

[15] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

[16] 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.

[17] 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.

[18] A.P Kopylov. Decidability of linear affine logic. *Information and Computation*, 164(1):173 – 198, 2001.

[19] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[20] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1), 1992.

[21] Dimitris Mostrous. Multiparty sessions based on proof nets. In *PLACES*, volume 155 of *EPTCS*, pages 1–8, 2014.

[22] Dimitris Mostrous and Vasco Thudichum Vasconcelos. Affine sessions. In Eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages: 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, pages 115–130, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[23] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.

[24] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[25] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

[26] 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.

[27] Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286. ACM, 2012.

Appendix A. Subject Reduction

Notation for type derivations. Let $\mathcal{D} :: \Gamma \vdash P$ stand for a typing derivation with conclusion P and interface Γ. To ease the notation, sometimes we ignore the process, writing $\mathcal{D} :: \Gamma$, or even the interface, writing simply $\mathcal{D}$.

We let $\mathcal{D}[\cdot_\Delta]$ stand for a derivation with a hole that contributes an interface Δ. We make a simplification and require the hole to be unguarded in $\mathcal{D}$, which means that it does not appear inside a sub-derivation $\mathcal{D}_1 :: \alpha$ of $\mathcal{D}$. To put simply, we do not consider holes in sub-derivations guarded by a prefix.

Definition A.1 (Derivation Composition). *We denote by $\mathcal{D}_1[\mathcal{D}_2] :: \Gamma, \Theta$ the derivation obtained from $\mathcal{D}_1[\cdot_{\Delta,\Lambda}] :: \Gamma, \Lambda$ by the placement of $\mathcal{D}_2 :: \Delta, \Theta$ in the hole, assuming the following conditions:*

(1) no element of Θ is bound in $\mathcal{D}_1$ (but elements of Δ can be bound);
(2) Γ, Θ is well-formed (as is Δ, Θ, by assumption).

The point of these conditions is to guarantee that compositions are sound, and specifically they constrain the type of derivation that can be plugged-in a hole allowing us to know the final interface; this is very useful in Subject Reduction. The part Λ is what passes from the inner context to the conclusion, which means it is unused in the derivation. The idea is that we can substitute it with some Θ that also passes to the conclusion, under suitable assumptions, namely that no name in Θ is bound in the derivation context. This allows to place in the hole a derivation with different (typically larger) interface, which is what happens during our Subject Reduction proof.

After we insert a derivation in place of the hole, we thus obtain $\mathcal{D}_1[\mathcal{D}_2] :: \Gamma, \Theta$.

Remark. If we wanted to provide a detailed definition of derivations with holes—which we think is not necessary— we would use an extended version of the typing system with the extra axiom $\overline{\Delta \vdash \cdot}$ and would require that it appears at most once in any derivation $\mathcal{D}$. (And when it does appear we obtain $\mathcal{D}[\cdot_\Delta]$, otherwise we have a normal derivation.)

Regarding the restriction to unguarded holes in a derivation, we do not need the more general form where the hole can be anywhere, since parts under prefix do not reduce and we never need to manipulate them. Also, our restriction simplifies the notion of placing a derivation with different interface in a hole, which we need to use extensively. Specifically, if the hole was allowed to appear inside a sub-derivation with restrictions on the resulting interface — such as in the premises of (Acc) or (Bra) — then an arbitrary interface added to them (Θ above) would not guarantee that we obtain a well-formed derivation, i.e., this plugging-in of a derivation could be unsound.

Proposition A.2. *If $\mathcal{D}_1[\cdot_{\Delta,\Lambda}] :: \Gamma, \Lambda$ and the conditions of instantiation detailed previously (see Def. A.1) are respected for some $\mathcal{D}_2 :: \Delta, \Theta$, then $\mathcal{D}_1[\mathcal{D}_2] :: \Gamma, \Theta$.*

Proof. This is easy to establish by induction on the structure of $\mathcal{D}_1$. □

Lemma A.3 (Atomic Action Sub-derivation). *If $\mathcal{D} :: \Gamma \vdash P$ and $P \equiv (\boldsymbol{\nu}\overrightarrow{bb'})(\alpha \mid Q)$ with $\mathsf{subject}(\alpha) = a$, then $\mathcal{D}$ is of the shape $\mathcal{D}_1[\mathcal{D}_2 :: \Delta, a\colon T \vdash \alpha]$.*

To put this into english, when a communication action α appears at the top level of a well-typed process P, then any derivation $\mathcal{D}$ of P will contain a sub-derivation with α

as its conclusion. Notice that we cannot guarantee that the last rule corresponds to the introduction of α. We give a simple example:

$$P = (\boldsymbol{\nu} cc')(\underbrace{\overline{a}b.\overline{c'}r}_{\alpha} \mid c(x).Q\,)$$

If the above is well-typed, the last rule is (Res) on endpoints c and c' and cannot be (Out) on channel a. This situation is obvious in Affine and Linear Logic, and in particular it is clear that without commutative reductions the instance of (Res) — which corresponds to a "cut" in Affine Logic — cannot go "up" in the derivation. In particular we would need $P \equiv \overline{a}b.(\boldsymbol{\nu} cc')(\,\overline{c'}r \mid c(x).Q\,)$.[3] This lemma is needed to identify and manipulate the derivations obtained during the proof of Subject Reduction. Specifically, when dealing with a redex inside a larger derivation, we can use this inversion-like lemma to obtain the sub-derivations for the two communications, compose them, use Lemma A.7 (Small Subject Reduction, defined shortly), and finally recompose the derivation for the contractum into the original derivation.

Proposition A.4 (Sub-interface Compositionality)**.** *If $\mathcal{D}_{1i}[\mathcal{D}_{2i} :: \Delta_i, a_i \colon T_i] :: \Gamma_i, a\colon T_i$ ($i \in \{1,2\}$) and Γ_1, Γ_2 is well-formed, then Δ_1, Δ_2 is well-formed.*

Proof. A simple explanation of this proposition is that, if two derivations can be composed using (Res), as is the case when $T_1 = \overline{T_2}$, then their sub-derivations can also be composed without conflicts.

In more detail, recall that well-formedness means that only elements of the shape $a\colon \mathsf{req}\ T$ can appear multiple times in an interface. In general, the Γ_i can have a larger interface than the Δ_i since other parts of the derivation can extend it, which is no problem. The only obstacle to the well-formedness of Δ_1, Δ_2 would be the case where the Δ_i have a larger domain than the corresponding Γ_i, since in that case the well-formedness of Γ_1, Γ_2 does not imply that of Δ_1, Δ_2. Now, let us consider the possible differences between Δ_i (the sub-derivation interface) and Γ_i. Because of weakening, Γ_i can have a larger domain (set of names). Because of contraction, Γ_i can have lesser copies of some $a\colon \mathsf{req}\ T$, but in this case the domain is the same. Session constructors change the types so that Δ_i can contain some $a\colon T$ which appears as $a\colon T'$ in Γ_i, so again the domain is the same. Finally, some part of Δ_i can be closed (by scope restriction) and in that case — only in that case — the domain of Δ_i can be larger than that of Γ_i. However, by the variable convention all bound names are distinct from each other and from all free names, and therefore there is no conflict when composing Δ_1, Δ_2. Therefore, the sub-derivations have interfaces that can be composed, i.e., the result is well-formed as required. □

Lemma A.5 (Free Variables)**.** *If a is free in P and $\Gamma \vdash P$ then $a\colon T \in \Gamma$.*

For the purpose of the Substitution Lemma, we denote by $\vec{x}\colon T$ the *non-empty* context $x\colon T, \ldots, x\colon T$. According to context formation, if x occurs twice in the context, then it must be the case that T is a request type.

Lemma A.6 (Substitution)**.** *If $\Gamma, \vec{x}\colon T \vdash P$ and $x \notin \Gamma$ then $\Gamma, b\colon T \vdash P\{^{b}/_{x}\}$.*

Proof. The proof is by rule induction on the first premise. Most cases feature two or three subcases depending on x being equal or different to names a and b in the typing rules. Extra

[3]This becomes unmanageable for more complex processes with multiple cuts, unless if type derivations are manipulated (see for example [27]).

subcases may also be needed depending on T being a request type or not. We illustrate one representative case: when derivation ends with the (Catch) rule.
Case $a = x$ and T is not a request type. We have $\Gamma, x\colon T \vdash \mathsf{do}\ \rho\ \mathsf{catch}\ P$ and $x \notin \Gamma$. By rule inversion we obtain $\Gamma, x\colon T \vdash \rho$ and $\Gamma \vdash P$ and $\mathsf{subject}(\rho) = x$. By induction, we get $\Gamma, b\colon T \vdash \rho\{^b/_x\}$. We are now in a position to apply rule (Catch) to obtain $\Gamma, b\colon T \vdash \mathsf{do}\ \rho\{^b/_x\}\ \mathsf{catch}\ P$. To complete the case we note that Lemma A.5 gives us that b is not free in P, hence $\mathsf{do}\ \rho\{^b/_x\}\ \mathsf{catch}\ P = (\mathsf{do}\ \rho\ \mathsf{catch}\ P)\{^b/_x\}$.
Case $a = x$ and T is a request type. We have $\Gamma, \vec{x}\colon T, x\colon T \vdash \mathsf{do}\ \rho\ \mathsf{catch}\ P$ and $x \notin \Gamma$. By rule inversion we obtain $\Gamma, \vec{x}\colon T, x\colon T \vdash \rho$ and $\Gamma, \vec{x}\colon T \vdash P$ and $\mathsf{subject}(\rho) = x$. By induction, we get $\Gamma, b\colon T \vdash \rho\{^b/_x\}$ and $\Gamma, b\colon T \vdash P\{^b/_x\}$. Applying the (Weak) rule as often as needed, we obtain $\Gamma, \vec{b}\colon T, b\colon T \vdash \rho\{^b/_x\}$ and $\Gamma, \vec{b}\colon T \vdash P\{^b/_x\}$. We can easily see that $\mathsf{subject}(\rho\{^b/_x\}) = b$. We are then in a position to apply rule (Catch) to obtain $\Gamma, \vec{b}\colon T \vdash \mathsf{do}\ \rho\{^b/_x\}\ \mathsf{catch}\ P\{^b/_x\}$, from which the result follows based the definition of substitution.
Case $a \neq x$. We have $\Gamma, a\colon T, \vec{x}\colon U \vdash \mathsf{do}\ \rho\ \mathsf{catch}\ P$. By rule inversion we obtain $\Gamma, a\colon T, \vec{x}\colon U \vdash \rho$ and $\Gamma, \vec{x}\colon U \vdash P$ and $\mathsf{subject}(\rho) = a$. Applying the induction hypothesis to both sequents, followed by the (Catch) rule we get $\Gamma, a\colon T, b\colon U \vdash \mathsf{do}\ \rho\{^b/_x\}\ \mathsf{catch}\ P\{^b/_x\}$. The result follows by applying the (Weak) rule as often as needed and the definition of substitution. □

Lemma A.7 (Small Subject Reduction)**.** *If* $\Gamma \vdash (\boldsymbol{\nu} ab)(\alpha_1 \mid \alpha_2)$ *and* $(\boldsymbol{\nu} ab)(\alpha_1 \mid \alpha_2) \longrightarrow (\boldsymbol{\nu} ab)\, Q$ *then* $\Gamma \vdash (\boldsymbol{\nu} ab)\, Q$.

Proof. From $\Gamma \vdash (\boldsymbol{\nu} ab)(\alpha_1 \mid \alpha_2)$, using (Contraction) and (Weak) (zero or more times) and (Res), we obtain $\Gamma_1, \mathsf{req}\ \Gamma_1', \mathsf{req}\ \Gamma_1', a\colon T \vdash \alpha_1$ and $\Gamma_2, \mathsf{req}\ \Gamma_2', \mathsf{req}\ \Gamma_2', b\colon \overline{T} \vdash \alpha_2$, where Γ is $\Gamma_1, \mathsf{req}\ \Gamma_1', \Gamma_2, \mathsf{req}\ \Gamma_2', \mathsf{req}\ \Theta, \mathsf{end}\,\Lambda$. We proceed by case analysis on the various reduction axioms, illustrating the most complex cases. Notice that $\mathsf{subject}(\alpha_1) = a$ and $\mathsf{subject}(\alpha_2) = b$, otherwise there is no reduction.
Case (R−Com). In this case T is $!T_1.T_2$ and $\Gamma_1, \mathsf{req}\ \Gamma_1', \mathsf{req}\ \Gamma_1'$ is $\Gamma_1'', c\colon T_1$. We have four cases to consider depending on the shape of H_1 and H_2. When both H_1 and H_2 are do-catch contexts, using rules (Contraction), (Res), (Catch), (Out) and (In), we continue with the only derivation scheme for the hypothesis, to conclude that $\Gamma_1'', a\colon T_2 \vdash P$ and $\Gamma_2, \mathsf{req}\ \Gamma_2', \mathsf{req}\ \Gamma_2', b\colon \overline{T_2}, x\colon T_1 \vdash Q$. The result follows from the substitution lemma, and rules (Contraction), (Weak) and (Res).
Case (C−Catch). Using rules (Contraction), (Res), (Cancel), and (Catch), we conclude: $\Gamma_1, \mathsf{req}\ \Gamma_1', \mathsf{req}\ \Gamma_1', a\colon T \vdash \rho$ and $\Gamma_1', \mathsf{req}\ \Gamma_1'', \mathsf{req}\ \Gamma_1'' \vdash P$ where Γ_2 is the empty context. We then distinguish two cases. When a is free in P, from Lemma A.5 we know that $a\colon U$ is in Γ, hence, by context formation, U is a request type. We conclude the proof using rules (Weak), (Contraction), (Cancel) and (Res). When a is not free in P, we conclude the proof from $\Gamma_1', \mathsf{req}\ \Gamma_1'', \mathsf{req}\ \Gamma_1'' \vdash P$, using contraction and the derived rule (Mix), recalling that $P \equiv (\boldsymbol{\nu} a'b')(P \mid \mathbf{0})$ where a', b' are fresh names. (We first obtain $(\boldsymbol{\nu} ab)(P \mid b\lightning) \equiv P$ and then introduce the new binders which are α-convertible to the desired result.)
Case (R−Ses). We obtain $\Gamma_1, \mathsf{req}\ \Gamma_1', \mathsf{req}\ \Gamma_1', a\colon \mathsf{acc}\ T \vdash \alpha_1$ and $\Gamma_2, \mathsf{req}\ \Gamma_2', \mathsf{req}\ \Gamma_2', b\colon \mathsf{req}\ T \vdash \alpha_2$. The part R of the scope in the rule is taken to be $\mathbf{0}$. We consider two cases, depending on the shape of the H-context; we analyse the case where H is empty since the exception handler disappears anyway. Rules (Contraction), (Acc), (Req) allow to conclude that $\Gamma_1, \mathsf{req}\ \Gamma_1', \mathsf{req}\ \Gamma_1' \vdash P$ and $\Gamma_2, \mathsf{req}\ \Gamma_2', \mathsf{req}\ \Gamma_2', x\colon U \vdash Q$. The result follows from the

application of the substitution lemma, and rules (Contraction), (Weak), (Mix), and (Res) if a appears in P. $\square$

Theorem 5.1 (Subject Reduction). If $\Gamma \vdash P$ and $P \longrightarrow Q$ then $\Gamma \vdash Q$.

Proof. We proceed by induction on the typing derivation. Since P must contain a redex, the possible last rules for any derivation of $\Gamma \vdash P$ are (Res, Contraction, Weak). From these, the cases for (Contraction, Weak) follow immediately by the induction hypothesis. Thus, we can focus on (Res).

Case (Res) We have $\mathcal{D} :: \Gamma \vdash P$ and the conclusion is of the shape $\Gamma_1, \Gamma_2 \vdash (\boldsymbol{\nu} ab)(P_1 \mid P_2)$ with premises $\mathcal{D}_{11} :: \Gamma_1, a\colon T \vdash P_1$ and $\mathcal{D}_{21} :: \Gamma_2, b\colon \overline{T} \vdash P_2$. (We indicate the derivations because we need to manipulate them.)

Now we consider the possible reductions of P. If $P_i \longrightarrow P_i'$ then the result easily follows from the induction hypothesis followed by an application of (Res). We therefore focus on the case where (part of) P_1 interacts with (part of) P_2. In this case there is only one possibility, that of a reduction on the endpoints a and b (possibly more than one when T or $\overline{T}$ is of the shape $\mathsf{req}\ T'$), which follows by the well-formedness of Γ_1, Γ_2. (Recall that the Γ_i can only have common elements of the shape $c\colon \mathsf{req}\ T'$, so it follows that the P_i can only communicate over the interface provided by $a\colon T$ and $b\colon \overline{T}$.)

We now consider the possible sub-cases for a redex on a/b. In order for a redex to be active, both components obviously need to be at the top level (i.e., unguarded), so we can determine that:

$$P_1 \equiv (\boldsymbol{\nu} \overrightarrow{cd})(\alpha_1 \mid P_1') \qquad P_2 \equiv (\boldsymbol{\nu} \overrightarrow{rs})(\alpha_2 \mid P_2')$$

where the α_i are dual actions on a and b, respectively.

We thus have $P \longrightarrow (\boldsymbol{\nu} ab)\, Q$ where Q is

$$(\boldsymbol{\nu} \overrightarrow{cd})(\boldsymbol{\nu} \overrightarrow{rs})(Q_1 \mid P_1' \mid P_2')$$

up to structural equivalence as usual.

Now, using Lemma A.3 we obtain:

$$\mathcal{D}_{11}[\mathcal{D}_{12} :: \Delta_1, a\colon T \vdash \alpha_1] :: \Gamma_1, a\colon T \vdash P_1 \qquad \mathcal{D}_{21}[\mathcal{D}_{22} :: \Delta_2, b\colon \overline{T} \vdash \alpha_2] :: \Gamma_2, b\colon \overline{T} \vdash P_2$$

Moreover, by Proposition A.4 we obtain that Δ_1, Δ_2 is a well-formed interface.

We now consider two cases, where the redices are linear and unrestricted, respectively. This is because each case requires a slightly different construction in order to obtain a derivation for the contractum. We start with the linear case, in which we know that the α_i are the only actions with subject a and b (for the other case we need to take into account the fact that multiple actions with interface $b\colon \mathsf{req}\ T'$ can appear).

Linear redex. We form the following derivation:

$$\mathcal{D}_3 \qquad = \qquad \frac{\mathcal{D}_{12} :: \Delta_1, a\colon T \vdash \alpha_1 \qquad \mathcal{D}_{22} :: \Delta_2, b\colon \overline{T} \vdash \alpha_2}{\Delta_1, \Delta_2 \vdash (\boldsymbol{\nu} ab)(\alpha_1 \mid \alpha_2)}\ (\mathsf{Res})$$

We have $(\boldsymbol{\nu} ab)(\alpha_1 \mid \alpha_2) \longrightarrow (\boldsymbol{\nu} ab)Q_1$ and by Lemma A.7 and $\mathcal{D}_3$ we obtain $\mathcal{D}_4 :: \Delta_1, \Delta_2 \vdash (\boldsymbol{\nu} ab)Q_1$. Now we can obtain the following derivation:

$$\mathcal{D}_{11}[\mathcal{D}_{21}[\mathcal{D}_4] :: \Gamma_2, \Delta_1] :: \Gamma_1, \Gamma_2 \vdash Q$$

It is easy to check that the conditions of Definition A.1 are respected (see also Proposition A.2), i.e., that the result is a valid derivation as required.

Unrestricted redex. Let us assume, without loss of generality (the other case is symmetric), that $T = \mathsf{req}\ T'$. In this case we must take into account that multiple compositions against $b\colon \mathsf{acc}\ \overline{T'}$ are possible within the derivation, so we perform a few more steps.
Case (R−Ses). We have the following derivations, assuming $\alpha_1 = \mathsf{req}\ \overline{a}c.R_1$ and $\alpha_2 = \mathsf{acc}\ b(x).R_2$, which are obtained by inversion for each prefix, taking into account weakening and contraction:

$$\mathcal{D}_{12} = \dfrac{\dfrac{\mathcal{D}_{13} :: \Delta_1', \mathsf{req}\ \Sigma_1, \mathsf{req}\ \Sigma_1 \vdash R_1}{\Delta_1', \mathsf{req}\ \Sigma_1, \mathsf{req}\ \Sigma_1, a\colon \mathsf{req}\ T', c\colon T' \vdash \mathsf{req}\ \overline{a}c.R_1}\ (\mathsf{Req})}{\Delta_1', \mathsf{req}\ \Sigma_1, \mathsf{req}\ \Theta_1, \mathsf{end}\ \Lambda_1, a\colon \mathsf{req}\ T', c\colon T' \vdash \mathsf{req}\ \overline{a}c.R_1}\ (\mathsf{Contraction}), (\mathsf{Weak})$$

$$\mathcal{D}_{22} = \dfrac{\dfrac{\mathcal{D}_{23} :: \Delta_2', \mathsf{req}\ \Sigma_2, \mathsf{req}\ \Sigma_2, x\colon T' \vdash R_2}{\Delta_2', \mathsf{req}\ \Sigma_2, \mathsf{req}\ \Sigma_2, b\colon \mathsf{acc}\ T' \vdash \mathsf{acc}\ b(x).R_2}\ (\mathsf{Acc})}{\Delta_2', \mathsf{req}\ \Sigma_2, \mathsf{req}\ \Theta_2, \mathsf{end}\ \Lambda_2, b\colon \mathsf{acc}\ T' \vdash \mathsf{acc}\ b(x).R_2}\ (\mathsf{Contraction}), (\mathsf{Weak})$$

with $\Delta_1 = \Delta_1', \mathsf{req}\ \Sigma_1, \mathsf{req}\ \Theta_1, \mathsf{end}\ \Lambda_1, c\colon T'$ and $\Delta_2 = \Delta_2', \mathsf{req}\ \Sigma_2, \mathsf{req}\ \Theta_2, \mathsf{end}\ \Lambda_2$ with $\Delta_2' = x_1\colon \mathsf{req}\ T_1, \ldots, x_n\colon \mathsf{req}\ T_n$.

From the substitution lemma we obtain $\mathcal{D}_{24} :: (\Delta_2', \mathsf{req}\ \Sigma_2, \mathsf{req}\ \Sigma_2, x\colon T')\{^c/_x\} \vdash R_2\{^c/_x\}$.

First we form:

$$\mathcal{D}_{14} = \dfrac{\mathcal{D}_{13} \qquad \mathcal{D}_{24}}{\Delta_1', \mathsf{req}\ \Sigma_1, \mathsf{req}\ \Sigma_1, (\Delta_2', \mathsf{req}\ \Sigma_2, \mathsf{req}\ \Sigma_2, x\colon T')\{^c/_x\} \vdash R_1 \mid R_2\{^c/_x\}}\ (\mathsf{Mix})$$

The above interface is clearly well-formed. First, by Proposition A.4 Δ_1', Δ_2' is well-formed, and also $c\colon T'$ is composable with Δ_1' by assumption.

Now, we form the following derivation, which is equal to the original except that we put $\mathcal{D}_{14}$ in the place of $\mathcal{D}_{12}$:

$$\mathcal{D}_3 = \dfrac{\mathcal{D}_{11}[\mathcal{D}_{14}] \qquad \mathcal{D}_{21}}{\Gamma_1, \Gamma_2] \vdash (\boldsymbol{\nu} ab)Q}\ (\mathsf{Res})$$

We omit some applications of (Contraction) on $\mathcal{D}_{11}[\mathcal{D}_{14}]$, as well as below (Res), which are needed to obtain the original interface, as required.
Case (C−Acc/Req/Cat). Similar to the above. □