

Structured Interactional Exceptions in Session Types (Full Version)

Marco Carbone¹, Kohei Honda¹, and Nobuko Yoshida²

¹ Department of Computer Science, Queen Mary, University of London

² Department of Computing, Imperial College London

Abstract. We propose an interactional generalisation of structured exceptions based on the session type discipline. Interactional exceptions allow communicating peers to asynchronously and collaboratively escape from the middle of a dialogue to reach another in a coordinated fashion, under an arbitrary nesting of exceptions. New exception types guarantee communication safety, as well as offering precise representations and type-abstraction of advanced conversation patterns found in practice. Protocols for coordinating normal and exceptional exit among asynchronously running sessions are introduced. The liveness property established under these protocols guarantees consistency of coordinated exception handling among communicating peers.

Contents

1	Introduction	1
2	Session Calculus with Interactional Exceptions	4
2.1	Syntax	4
2.2	Two Realistic Examples	6
2.3	Semantics	7
2.4	Properties of Reduction and Meta Reduction.	11
2.5	Examples of Reduction	11
3	Typing Interactional Exception	13
3.1	Type Syntax	13
3.2	Typing System	13
4	Type Safety	18
4.1	Basic Properties.	18
4.2	Subject Meta Reduction.	20
4.3	Type Reduction	22
4.4	Subject Congruence and Subject Reduction.	23
5	Liveness and Termination	31
5.1	A Termination Protocol	31
5.2	Normal and Exceptional Exits.	32
5.3	Liveness	36
6	Related Work and Conclusion	39
6.1	Further Results	39

6.2	Related Work	39
6.3	Further Topics	40

1 Introduction

Structured exceptions in modern programming languages such as Java and C# allow a thread of control in a block (often designated as “try block”) to get transferred to another block (exception handler, “catch block”), when a system or user raises an event called *exception*. Their central merit is to enable a dynamic escape from a block of code to another (like `goto`), but in a controlled and structured way (unlike `goto`). They are useful not only for error-handling but more generally for a flexible control flow while preserving well-structured description and type-safety.

This paper studies the new notion of structured exceptions for distributed, concurrent, asynchronously communicating programs based on session types [10, 18], motivated by collaboration with industry partners in web services [20] and financial protocols [14]. These two application domains contain a wealth of structured conversation patterns arising from practical needs [11], and many of these patterns crucially rely on dynamic escape: a conversation is interrupted by a special communication action, after which all peers move to a different stage of conversation. Realising such conversation patterns requires a consistent propagation of exception messages among concurrently communicating peers; an exception affects not only a sequential thread but also a collection of parallel processes; and an escape needs to move into another dialogue in a concerted manner. The distinguishing feature of these exceptions in comparison with their traditional counterpart is that they demand not only local but also coordinated actions among communicating peers. We call such exceptions, *interactional exceptions*.

As a simple example of interactional exceptions, we present the following scenario, coming from widely used financial protocols. Assume Seller wishes to sell a product to Buyer. Henceforth we assume a message passing in a session is asynchronous, i.e. the completion of a sending action does not need a handshake by its receiver, a standard assumption in financial messaging [1].

1. Seller repeats sending quotes of its products, where each repetition is done without waiting for Buyer’s acknowledgement;
2. When Buyer replies with his interest in one of the quotes, the loop terminates and Seller and Buyer move to another stage, for e.g. completion of the transaction.

This simple conversation pattern contains an asynchronous escape from one part of a conversation to another. After one party aborts, the same thing should happen to the other, both moving together to another part of the conversation.

As a second example, we continue the above scenario, extending to the situation where Buyer and Seller negotiate the price of the property through Broker.

1. Buyer initiates a conversation (session) with Broker, in order to buy a property.
2. As a result, Broker initiates a conversation with Seller, and start brokering between Buyer and Seller, to reach successful transactions.
3. If there is an exceptional circumstance (e.g. a legal issue arises) after 1 but before 2, Buyer or Broker will abort and they together move to an exception dialogue to abort the transactions formally.
4. On the other hand, if there is an exceptional circumstance during 2, then there is an exception dialogue involving all of Broker, Seller and Buyer.

In this scenario, an exception handling at Broker side is *nested*, whose later, or inner, exception handling (4, involving all three parties) supersedes the earlier, or outer, one (3, involving only Broker and Seller). As a conversation evolves, more communication peers may naturally be involved, making it necessary to coordinate more parties when an exception is raised.

To maintain the virtues of traditional structured exceptions, as well as those of the existing session type discipline, we may as well demand the following three properties for this generalised form of exceptions.

- *flexibility*: it should allow asynchronous escape at any desired point of a conversation, including nested exceptions;
- *consistency*: even under asynchrony, messages in a “default” conversation should not get mixed up with those in an “exception” conversation, under arbitrary nesting;
- *type safety*: communications inside a session always take place linearly and without communication mismatch, carrying out fundamental properties of foregoing session types disciplines.

We address these requirements by an extension to the standard session types, with the following key features.

1. Asynchronous exceptions where nested scopes are consistently handled by a meta-level reduction and a stack discipline. A simple machinery, based on exception levels, prevents mix-up of messages in normal and exception conversations.
2. An operational structure for coordinating exceptions including the protocols to propagate exceptions, to handle normal and exceptional exit from a conversation, and to coordinate entry into exception-handling conversations.
3. A type structure for interactional exceptions which minimally extends that of the existing session types. They ensure communication safety and liveness, which together guarantee the consistency of the introduced operational structures.

The stipulated formal semantics of interactional exceptions is intended to suggest a possible framework of implementation, as discussed in § 6. As far as we know, this work is the first to present a consistent extension of session type disciplines to interactional exceptions, backed up by its key formal properties.

In the remainder, Section 2 introduces the syntax and semantics. Section 3 proposes the type discipline and establishes type soundness. Section 5 establishes liveness. Section 6 concludes with related work and further topics.

2 Session Calculus with Interactional Exceptions

2.1 Syntax

Hereby, we introduce the syntax of processes using the original π -calculus with session primitives [10]. We use the *Service Channel Principle* (SCP) [4, 17] and asynchronous session communication [3, 6, 8, 12], making the calculus close to real-world settings.

Definition 2.1 (Service Channel Principle (SCP)). *Invocation channels are always available. Therefore, they can be shared and invoked repeatedly.*

Let a, b range over *service channels*; s, r, t over *session channels*; x, y, z over *variables*; and X, X', \dots over *term variables*. A session channel s^p is a *polarised channel* [7] where variable p ranges over polarities $\{+, -\}$. We define the dual of a polarised channel s^p as $s^{+} = s^{-}$ and $s^{-} = s^{+}$. The syntax of *programs* (processes written by the user and denoted by P, Q, R, \dots) is reported in Table 1.

Table 1 Syntax of Session Calculus with Interactional Exceptions

$P ::= *c(\lambda)[P, Q]$	(service)
$ \bar{c}(\lambda)[\bar{\kappa}, P, Q]$	(request)
$ \kappa?(x).P$	(input)
$ \kappa!\langle e \rangle.P$	(output)
$ \kappa \triangleright \{l_i : P_i\}_{i \in I}$	(branch)
$ \kappa \triangleleft l.P$	(select)
$ P Q$	(par)
 if e then P else P	(cond)
 $\mathbf{0}$	(inact)
$ (va)P$	(resServ)
$ X$	(termVar)
$ \mu X.P$	(recursion)
 throw	(throw)
$e ::= a tt ff e \text{ and } e \neg e \dots$	
$c ::= a x$	
$\kappa, \lambda ::= s^p$	

A service $*a(\lambda)[P, Q]$, named a , is a replicated process where P is the *default process* and Q is the *exception handler*. Note that, because of (SCP), a service can never occur (never be nested) in a default process or in an exception handler. As a consequence, the sudden termination of a process can never terminate a service, hence preserving (SCP).

When a session is established via a request to a shared name a at a service, a fresh session channel is established, through which a series of communication actions are performed. In a request $\bar{c}(\lambda)[\tilde{\kappa}, P, Q]$, the handler Q can be associated with already established sessions $\tilde{\kappa}$, thus allowing nesting of exceptions. These channels are used for exception propagation. We call $\bar{c}(\lambda)[\tilde{\kappa}, P, Q]$ a *refinement* of κ_i for $\kappa_i \in \{\tilde{\kappa}\}$, if $\kappa_i \neq \lambda$. For allowing consistent propagation of exceptions, we make the following assumptions³:

Assumption 1 For each $\bar{c}(\lambda')[\tilde{\kappa}', P', Q']$ occurring in P , we have $\kappa_i \in \tilde{\kappa}'$ if and only if $\tilde{\kappa} \subseteq \tilde{\kappa}'$.

Assumption 2 In (service) and (request), session channels κ_i are never refined in the exception handler.

The first assumption ensures that two channels coupled together by a refinement are never split in subsequent refinements avoiding propagation inconsistency due to asynchrony. The second assumption is more obvious: it avoids “trying” again once an exception is thrown. As an example, consider the program

$$\bar{a}(\lambda)[\lambda, \bar{b}(\kappa)[(\kappa, \lambda), \bar{c}(\kappa')[(\kappa', \lambda), P, Q], Q'], Q'']$$

Above, when service b is called, λ is refined and coupled with the new session channel κ . On the other hand, when service c is called, κ' is refined and coupled only with λ leaving κ out. Note that the two service requests (on b and c) contradict each other as the first call asks for exception propagation between λ and κ while the second only between λ and κ' .

The constant process **throw** denotes a process willing to throw an exception. We assume each **throw** only occurs in the default process of a service or a request as it would not make sense elsewhere. All other constructs are identical with those in [4, 10].

We assume recursion is *guarded*, i.e. in $\mu X. P$, P is prefixed by one of the input/output/branch/select/cond. We also require that term variable X never appears free in the default process of a request or a service. Free/bound (term) variables/channels and α -equivalence are standard. $\text{fsc}(P)$, $\text{fn}(P)$, $\text{fv}(P)$ and $\text{ftv}(P)$ respectively denote the sets of free session channels, service channels, variables and term variables in P . Hereafter we consider programs are closed (i.e. $\text{fsc}(P) = \text{fn}(P) = \text{fv}(P) = \text{ftv}(P) = \emptyset$). We often omit the tailing $\mathbf{0}$.

³ These assumptions could also be included in the type system reported in the next section. For the sake of simplicity and clarity of other sections, we have included them here.

2.2 Two Realistic Examples

Example 2.2 (Asynchronous Escape). We can write the first example in § 1 as:

Buyer	Broker
$\overline{\text{chSeller}}(s^+)[s^+,$ $\quad \mu X. s^+(y). \text{if ok}(y) \text{ then throw else } X,$ $\quad s^+!\langle \text{card} \rangle. s^+(z)$ $]$	$*\text{chSeller}(s^-)[$ $\quad \mu X. s^-!\langle \text{quote} \rangle. X,$ $\quad s^-?(y_2). s^-!\langle \text{time} \rangle$ $]$

Buyer keeps on reading messages on s^+ until condition $\text{ok}(y)$ is met and then it throws an exception. Seller, instead, is in an infinite loop where it persistently sends a quote over channel s^- (we assume quote changes over time). When the exception is raised the handlers are run: Buyer will send the credit card details card and Seller will acknowledge on channel s^- with the current time.

Example 2.3 (Nested Escapes). The second example given in the introduction, can be represented in our calculus as (Seller is unchanged from Example 2.2):

Buyer	Broker
$\overline{\text{chBroker}}(s^+)[s^+,$ $\quad s^+!\langle \text{id} \rangle.$ $\quad \mu X. s^+(y).$ $\quad \text{if ok}(y) \text{ then throw else } X,$ $\quad s^+ \triangleright \{ l_1 : s^+!\langle \text{card} \rangle. s^+(z),$ $\quad \quad l_2 : P_{\text{abort}} \}$ $]$	$*\text{chBroker}(s^-)[s^-,$ $\quad s^-?(x).$ $\quad \text{if bad}(x) \text{ then throw else}$ $\quad \overline{\text{chSeller}}(t^+)[(t^+, s^-),$ $\quad \quad \mu X. t^+(x). s^-!\langle x + 10\% \rangle. X,$ $\quad \quad s^- \triangleleft l_1. s^-?(y_2). t^+!\langle y_2 \rangle.$ $\quad \quad t^+(y_3). s^-!\langle y_3 \rangle \quad ,$ $\quad]$ $\quad s^- \triangleleft l_2. R_{\text{abort}}$ $]$

Buyer first sends its identification id and then Broker throws an exception or proceeds by invoking Seller based on the evaluation of $\text{bad}(\text{id})$. In the first case, process $s^- \triangleleft l_2. R_{\text{abort}}$ in the outermost handler selects the l_2 branch on Buyer's handler and proceeds with abortion (conversation between P_{abort} and R_{abort}). In the other case, Seller is invoked by refining session channel s^- and the protocol proceeds as in Example 2.2 with Broker forwarding messages and

increasing the quote by 10%. When Buyer decides to accept a quote, the innermost handler is run by Broker which selects the l_1 conversation in Buyer's handler and forwards the exception to Seller. Then Broker forwards messages between Buyer and Seller, successfully completing the transaction.

2.3 Semantics

We augment the semantics of asynchronous sessions [3, 8, 12], with protocols for interactional exception, which include, among others:

1. *Exception propagation*: once a local exception is thrown, the other peer needs to be informed, and if that peer is engaged in another session, the third party should also be notified. Such a protocol should take care of the situation when two or more parties are asynchronously throwing exceptions.
2. *Exception handling*: when a site throws an exception or is notified of one, it should consistently shut down its default process which may contain multiple concurrent activities, as well as propagating the exception.
3. *Communication at matching levels*: due to asynchrony, when a default process of one peer sends a message, the receiving peer may throw an exception before the message arrives, so that we should make sure each message sent at some level is necessarily received by the receiver at the same level, even under nested try-catch inside an exception handler.

In the following we substantiate these ideas as formal operational semantics. In brief, the first of the above three points is realised by propagation of a special message (written \dagger); the second by a meta level reduction (written \Downarrow); and the third by the levels annotating queues (in effect annotating messages in transit). Following [3, 6, 8, 12], we use *runtime processes* to define the operational semantics. The extension of the grammar of processes is given in Table 2.

Table 2 Extension of Syntax of Session Calculus with Interactional Exceptions

$P ::= \dots (vs) P$	(resSess)
$\quad \kappa \hookrightarrow_{\phi} \bar{\kappa} : L$	(queue)
$\quad \text{try}\{P\} \text{ catch } \{\bar{\kappa} : Q\}$	(try-catch)
$\quad \bar{\kappa}\{P\}$	(wrap)
$L ::= \epsilon h :: L$	
$h ::= l a \text{tt} \text{ff} \dagger$	

Free variables and channels are extended to run-time processes. Session restriction $(vs) P$ is standard. For formalising order-preserving asynchronous message passing, we use a directed message queue $\kappa \hookrightarrow_{\phi} \bar{\kappa} : L$ [3, 8], where κ (source)

and $\bar{\kappa}$ (target) are two dual endpoint session channels. ϕ ranges over natural numbers, describing the level of the nesting of exceptions at which messages in the queue are to be sent and received. We often write $\kappa \hookrightarrow \bar{\kappa} : L$ for $\kappa \hookrightarrow_0 \bar{\kappa} : L$. The list $L :: h$ is obtained by extending L with an extra tail element h . The *try-catch block* $\mathbf{try}\{P\} \mathbf{catch}\{\bar{\kappa} : \tilde{Q}\}$ is the runtime presentation of a default process and a handler: the default process P in the *try-block* is running during which an exception on channels $\bar{\kappa}$ can be thrown, which terminates P and launches the handler Q in the *catch-block*. When this Q is launched, it becomes a *wrapped process* or a *wrap*, $\tilde{\kappa}\{\{Q\}\}$, making Q immune to an exception notification at the same or upper levels (note such notifications can come due to asynchrony). The transition from a try-catch to the wrap is realised by the meta reduction.

Meta Reduction. The *meta reduction* performs the following three functions:

- Erase the remaining activity of the default process in the try-block.
- Propagate exceptions to the try-blocks inside the try-block.
- Leave wrapped processes as they are.

In traditional structured exceptions as found in Java or C++, an exception completely erases the try-block and lets the handler run in the same state. In processes, it is natural to have concurrently running threads inside a try-block, having conversations (sessions) with other agents. We cannot erase these processes since that would make the state of conversations inconsistent. Thus we have an exception thrown in each of them.⁴

Table 3 Meta Reduction rules

$$\begin{array}{l}
(\text{MTRY}) \quad P \Downarrow (P', S) \Rightarrow \mathbf{try}\{P\} \mathbf{catch}\{\bar{\kappa} : Q\} \Downarrow \begin{cases} (P', S) & \text{if } \bar{\kappa} \subseteq S \\ \tilde{\kappa}\{\{Q\}\} \mid P', S \cup \bar{\kappa} & \text{otherwise} \end{cases} \\
(\text{MWRAP}) \quad \tilde{\kappa}\{\{Q\}\} \Downarrow (\tilde{\kappa}\{\{Q\}\}, \emptyset) \\
(\text{MPAR}) \quad P \Downarrow (P', S_1) \text{ and } Q \Downarrow (Q', S_2) \Rightarrow P \mid Q \Downarrow (P' \mid Q', S_1 \cup S_2) \\
(\text{MNIL}) \quad R \Downarrow (\mathbf{0}, \emptyset) \quad \text{if } R \in \left\{ \begin{array}{l} (\text{zero}), (\text{request}), (\text{input}), (\text{output}), (\text{branch}), \\ (\text{select}), (\text{cond}), (\text{recursion}), (\text{throw}) \end{array} \right\}
\end{array}$$

The meta reduction is written $P \Downarrow (P', S)$, where the initial process P is transformed into process P' , the result of erasing and wrapping; and S denotes session channels via which we should communicate that the exception takes place including the ones of nested try-catch blocks. The rules are given in Table 3.

(MTRY) is the key rule, the one which propagates the exception to a nested try-catch block. If the try-block meta reduces to some P' with some set S then

⁴ An alternative mechanism may leave these try-catch blocks intact which can however be simulated by the present mechanism with a minor change in operational semantics.

Table 4 Reduction Semantics

$$\begin{array}{l}
\text{(INIT)} \quad *a(s^-)[P, Q] \mid C[\bar{a}(s^+)[\bar{\kappa}, P', Q']] \longrightarrow \\
\quad \quad \quad *a(s^-)[P, Q] \mid (vs) \left(\mathbf{try}\{P\} \mathbf{catch} \{s^- : Q\} \mid s^- \hookrightarrow_0 s^+ : \epsilon \mid \right. \\
\quad \quad \quad \left. C[\mathbf{try}\{P'\} \mathbf{catch} \{\bar{\kappa} : Q'\}] \mid s^+ \hookrightarrow_0 s^- : \epsilon \right) \\
\text{(OUT)} \quad \kappa!(e), P \mid \kappa \hookrightarrow_\phi \bar{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_\phi \bar{\kappa} : (v :: L) \quad (e \downarrow v) \\
\text{(IN)} \quad \kappa?(x), P \mid \bar{\kappa} \hookrightarrow_0 \kappa : (L :: v) \longrightarrow P\{v/x\} \mid \bar{\kappa} \hookrightarrow_0 \kappa : L \\
\text{(SEL)} \quad \kappa \triangleleft L, P \mid \kappa \hookrightarrow_\phi \bar{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_\phi \bar{\kappa} : (l :: L) \\
\text{(BRA)} \quad \kappa \triangleright \{l_i : P_i\}_{i \in I} \mid \bar{\kappa} \hookrightarrow_0 \kappa : (L :: l_j) \longrightarrow P_j \mid \bar{\kappa} \hookrightarrow_0 \kappa : L \quad (j \in I) \\
\text{(CON)} \quad P \longrightarrow Q \Rightarrow C[P] \longrightarrow C[Q] \\
\text{(IF)} \quad \mathbf{if} \ e \ \mathbf{then} \ P \ \mathbf{else} \ Q \longrightarrow P \quad (e \downarrow \text{tt}) \quad \mathbf{if} \ e \ \mathbf{then} \ P \ \mathbf{else} \ Q \longrightarrow Q \quad (e \downarrow \text{ff}) \\
\text{(STR)} \quad P \equiv P' \ \text{and} \ P' \longrightarrow Q' \ \text{and} \ Q' \equiv Q \Rightarrow P \longrightarrow Q \\
\text{(THR)} \quad \mathbf{try}\{P\} \mathbf{catch} \{\bar{\kappa} : Q\} \Downarrow (R, S) \Rightarrow \\
\quad \mathbf{try}\{\mathbf{throw} \mid P\} \mathbf{catch} \{\bar{\kappa} : Q\} \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : L_\kappa \longrightarrow R \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : (\dagger :: L_\kappa) \\
\text{(RTHR)} \quad \mathbf{try}\{P\} \mathbf{catch} \{\bar{\kappa} : Q\} \Downarrow (R, S) \Rightarrow \\
\quad \mathbf{try}\{P\} \mathbf{catch} \{\bar{\kappa} : Q\} \mid \bar{\kappa}_j \hookrightarrow_0 \kappa_j : (L :: \dagger) \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : L_\kappa \\
\quad \quad \quad \longrightarrow R \mid \bar{\kappa}_j \hookrightarrow_1 \kappa_j : L \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : (\dagger :: L_\kappa) \\
\text{(WVAL)} \quad \bar{\kappa}\{[Q]\} \mid \bar{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: v) \longrightarrow \bar{\kappa}\{[Q]\} \mid \bar{\kappa}_i \hookrightarrow_0 \kappa_i : L \\
\text{(WTHR)} \quad \bar{\kappa}\{[Q]\} \mid \bar{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: \dagger) \longrightarrow \bar{\kappa}\{[Q]\} \mid \bar{\kappa}_i \hookrightarrow_1 \kappa_i : L \\
\text{(CLEAN)} \quad \mathbf{try}\{P \mid \lambda \hookrightarrow_\phi \bar{\lambda} : L\} \mathbf{catch} \{\bar{\kappa} : \bar{Q}\} \longrightarrow P \mid \lambda \hookrightarrow_\phi \bar{\lambda} : L \quad (\lambda \in \bar{\kappa}, \dagger \in L)
\end{array}$$

$\mathbf{try}\{P\} \mathbf{catch} \{\bar{\kappa} : Q\}$ will reduce either to (i) the parallel composition of P' and $\bar{\kappa}\{[Q]\}$ or to (ii) P' itself with the new set $S \cup \bar{\kappa}$ ensuring that also channels $\bar{\kappa}$ will be notified with an exception. Case (i) happens when there is no refinement of $\bar{\kappa}$ in P while (ii) discards handler Q when another handler for $\bar{\kappa}$ is already in P . The mechanism is sound because of the assumption that κ_i are always refined together (cf. syntax). Note that, when there is only a single thread in the try-block, the meta reduction mechanism is identical with that of the standard exception handling.

Reduction. We now introduce the main reduction rules. Due to the nesting of wraps and try-catch blocks, the reduction is defined using the following reduction context:

$$C ::= \mathbf{try}\{C\} \mathbf{catch} \{\bar{\kappa} : Q\} \mid P \mid C \mid \bar{\kappa}\{[C]\} \mid (vs) C \mid (va) C \mid -$$

The reduction \longrightarrow is the smallest relation generated by the rules in Table 4.

We illustrate the most relevant rules. (INIT) gives the semantics of session initiation, generating two fresh dual session channels, the associated two empty queues (ϵ denotes the empty string) and the two try-catch blocks $\mathbf{try}\{P\} \mathbf{catch} \{s^- : Q\}$ and $\mathbf{try}\{P'\} \mathbf{catch} \{\bar{\kappa} : Q'\}$. Note that $*a(s^-)[P, Q]$ is not in a context. This

is because we have assumed that *services never appear nested in a try- or a catch-block* as we do not want them to be terminated (following SCP).

(OUT) and (SEL) enqueue, respectively, a value and a label at the tail of the queue for κ . Symmetrically (IN) and (BRA) dequeue, from the head of the queue, a value and a label. The exception level in the latter two rules is 0, indicating the level of an actual receiver. The exception level of a queue ensures that a message is sent and received at the same level, guaranteeing consistency of communication. This depends on the invariance that the sum of the exception level of the queue and the number of \dagger 's in the queue before a specific message, determines the depth (counted as the number of wraps) at which the message enqueueing is performed.

In (IF), $e \downarrow v$ says that expression e evaluates to values v . (CON,STR) are standard.

(THR) and (RTHR) represent the firing of an exception. (THR) is when **throw** appears top-level in the try-block, i.e. exception is thrown locally; while (RTHR) is when a remote exception is received (as \dagger in the queue). Note both rules use meta reduction. The same mechanism is at work in (WTHR). Eventually all communicating peers will be notified of the exception by sending \dagger via channels in S generated from P as well as $\tilde{\kappa}$.

We conclude with (WVAL) which describes the case when messages are drained into a sink (i.e. get dequeued but ignored) when messages at exception level meet a wrapped process. In (WTHR), \dagger meets a wrap and the exception level of the queue is incremented, allowing the queue to enter the wrap.

This last step is formally defined by the structural congruence \equiv which plays a key role in treating exceptions and, in particular, moving queues while maintaining their exception levels. \equiv is the least equivalence relation on processes such that $(P, |)$ is a commutative monoid and includes the standard rules for the restriction (such as scope extrusion) and the recursion. In addition, it has the following rules:

- a) $\mathbf{try}\{P \mid \lambda \hookrightarrow_{\phi} \bar{\lambda} : L\} \mathbf{catch} \{\tilde{\kappa} : Q\} \equiv \mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \mid \lambda \hookrightarrow_{\phi} \bar{\lambda} : L \quad (\lambda \in \tilde{\kappa} \Rightarrow \dagger \notin L)$
- b) $\tilde{\kappa}\{P \mid \bar{\lambda} \hookrightarrow_{\phi} \lambda : L\} \equiv \tilde{\kappa}\{P\} \mid \bar{\lambda} \hookrightarrow_{\phi} \lambda : L \quad (\lambda \notin \tilde{\kappa})$
- c) $\tilde{\kappa}\{P\} \mid \bar{\kappa}_i \hookrightarrow_{\phi} \kappa_i : L \equiv \tilde{\kappa}\{P \mid \bar{\kappa}_i \hookrightarrow_{\phi-1} \kappa_i : L\}$
- d) $\mathbf{try}\{(va) P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \equiv (va) \mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \quad (a \notin \text{fn}(Q))$
- e) $\tilde{\kappa}\{(va) P\} \equiv (va) \tilde{\kappa}\{P\}$

The first and second rules respectively allow a queue to move into a try-catch block and a wrap. The third rule says when the receiving side of the queue is in $\tilde{\kappa}$: when entering the wrap, ϕ is decreased so that the process inside the wrap can read the value if the level after the decrement is 0, cf. (IN,BRA) in Table 4. The last two rules open the scope.

2.4 Properties of Reduction and Meta Reduction.

We formally define the reflexive and transitive closure of the reduction semantics.

Definition 2.4 (\rightarrow^*). \rightarrow^* is defined as the reflexive and transitive closure of \rightarrow .

If we always start from programs, we have:

Lemma 2.5. *Let P be a program and $P \rightarrow^* P'$. Then any queue, wrap or session restriction occurring in P' is not prefixed.*

Proof. The proof follows from the semantics of the calculus. □

The following is a straightforward property of meta reduction.

Proposition 2.6. *For each P which does not contain session restrictions by (*resSess*) or queues and has no free recursion variables there exist P', S such that $P \Downarrow (P', S)$.*

Proof. It follows from the definition of meta reduction. □

Proposition 2.7. *If $P \Downarrow (R, S)$ then $R \equiv \prod_i \tilde{\kappa}_i \{Q_i\}$.*

Proof. By induction on the rules for meta reduction. □

In the following, let $\dagger(L)$ be the number of \dagger in the list L and $\text{wrap}(P, \kappa)$ be the number of nested wraps on κ ⁵. The following result states that, given a queue $\kappa \hookrightarrow_\phi \bar{\kappa} : L$, the number of nested wraps over κ is always equal to ϕ plus the number of \dagger present in L .

Theorem 2.8. *Let P be a program such that $P \rightarrow^* C[R \mid \kappa \hookrightarrow_\phi \bar{\kappa} : L]$. Then, we have $\text{wrap}(R, \kappa) = \phi + \dagger(L)$.*

Proof. By induction on the reduction rules. □

2.5 Examples of Reduction

Example 2.9. As an example of reduction, consider the following process with a queue:

$$\mathbf{try}\{s^+!(\text{hello}).P \mid \mathbf{throw}\} \mathbf{catch}\{s^+ : Q_1\} \quad | \quad s^+ \hookrightarrow s^- : \epsilon \quad (1)$$

which communicates with another process:

$$\mathbf{try}\{s^-?(x).R \mathbf{catch}\{s^- : Q_2\} \quad | \quad s^- \hookrightarrow s^+ : \epsilon \quad (2)$$

⁵ In the present calculus, the number of wraps can only be either 0 or 1, but this can be generalised to many levels. See the last section of this document for further details.

(1) is now sending `hello` but it is also throwing an exception. In one possible reduction sequence of (1), `hello` will be placed in the queue and then an exception is thrown.

$$\begin{aligned}
(1) &\equiv \mathbf{try}\{s^+!\langle\mathbf{hello}\rangle.P \mid \mathbf{throw} \mid s^+ \hookrightarrow s^- : \epsilon\} \mathbf{catch} \{s^+ : Q_1\} \\
&\rightarrow \mathbf{try}\{P \mid \mathbf{throw} \mid s^+ \hookrightarrow s^- : \mathbf{hello}\} \mathbf{catch} \{s^+ : Q_1\} \\
&\equiv \mathbf{try}\{P \mid \mathbf{throw}\} \mathbf{catch} \{s^+ : Q_1\} \mid s^+ \hookrightarrow s^- : \mathbf{hello} \\
&\rightarrow s^+\{\{Q_1\}\} \mid s^+ \hookrightarrow s^- : (\dagger :: \mathbf{hello})
\end{aligned} \tag{3}$$

Messages `hello` and then \dagger are now delivered to (2) as follows, omitting the use of \equiv :

$$\begin{aligned}
(3) \mid (2) &\rightarrow s^+\{\{Q_1\}\} \mid s^+ \hookrightarrow s^- : \dagger \mid \mathbf{try}\{R\{\mathbf{hello}/x\}\} \mathbf{catch} \{s^- : Q_2\} \mid s^- \hookrightarrow s^+ : \epsilon \\
&\rightarrow s^+\{\{Q_1\}\} \mid s^+ \hookrightarrow_1 s^- : \epsilon \mid s^-\{\{Q_2\}\} \mid s^- \hookrightarrow_0 s^+ : \dagger \\
&\rightarrow s^+\{\{Q_1\}\} \mid s^+ \hookrightarrow_1 s^- : \epsilon \mid s^-\{\{Q_2\}\} \mid s^- \hookrightarrow_1 s^+ : \epsilon
\end{aligned}$$

Observe the increment of the exception levels of the two queues: this allows Q_1 and Q_2 to converse at the same level. \square

Example 2.10. We now show how the semantics manages the presence of refinement. Consider for instance the following process:

$$\mathbf{try}\{\mathbf{try}\{\mathbf{throw}\} \mathbf{catch} \{(\kappa, \lambda) : Q_1\}\} \mathbf{catch} \{\kappa : Q_2\} \mid \bar{\kappa} \hookrightarrow_0 \kappa : \dagger \mid \kappa \hookrightarrow_0 \bar{\kappa} : L \tag{4}$$

The process above has two different reductions: either by (THR) throwing an exception in the inner try-catch block or by remote exception applying (RTHR). In the first case, applying (THR), (CLEAN) and (WTHR), we have (we omit some queues):

$$\begin{aligned}
(4) &\equiv \rightarrow \mathbf{try}\{(\kappa, \lambda)\{\{Q_1\}\} \mid \kappa \hookrightarrow_0 \bar{\kappa} : \dagger :: L\} \mathbf{catch} \{\kappa : Q_2\} \mid \bar{\kappa} \hookrightarrow_0 \kappa : \dagger \rightarrow \\
&(\kappa, \lambda)\{\{Q_1\}\} \mid \kappa \hookrightarrow_0 \bar{\kappa} : \dagger :: L \mid \bar{\kappa} \hookrightarrow_0 \kappa : \dagger \rightarrow (\kappa, \lambda)\{\{Q_1\}\} \mid \kappa \hookrightarrow_0 \bar{\kappa} : \dagger :: L \mid \bar{\kappa} \hookrightarrow_1 \kappa : \epsilon
\end{aligned}$$

While in the second case, by (RTHR), we reduce in one step to the same process above. \square

3 Typing Interactional Exception

This section introduces a type discipline for sessions with interactional exceptions. In comparison with the standard session types, the central difference is the shape of a type itself, which now consists of the abstraction of the default behaviour (the “try” part) and that of the handler behaviour (the “catch” part). This simple extension, combined with the use of levels, allows us to establish subject reduction, guaranteeing that messages are always delivered at proper levels at proper timings in the presence of nested asynchronous escapes, testifying consistency of the operational semantics introduced in §2. The typing rules for programs induce an efficient type checking algorithm.

3.1 Type Syntax

The grammar of types extends the standard session types:

$$\begin{aligned} \alpha &::= \downarrow(\theta).\alpha \mid \uparrow(\theta).\alpha \mid \oplus\{l_i : \alpha_i\}_{i \in I} \mid \&\{l_i : \alpha_i\}_{i \in I} \mid \alpha\{\beta\} \mid \mathbf{end} \mid \mu\mathbf{t}.\alpha \mid \mathbf{t} \\ \theta &::= \langle \alpha\{\beta\} \rangle \mid \mathbf{bool} \mid \dots \end{aligned}$$

α and θ are respectively called *session types* and *service types*. The grammar follows the standard session types [10, 18], except for *try-catch type* $\alpha\{\beta\}$, the abstraction of a try-catch block: in $\alpha\{\beta\}$, α denotes the type of the try-block and β the catch block. A session type α is *plain* if it does not use a try-catch type (except in a service type it carries). From now on in $\alpha\{\beta\}$, we stipulate α is plain.

We define the *dual* [10] of α , written $\bar{\alpha}$, as:

$$\begin{array}{ll} \overline{\downarrow(\theta).\alpha} = \uparrow(\theta).\bar{\alpha} & \overline{\uparrow(\theta).\alpha} = \downarrow(\theta).\bar{\alpha} \\ \overline{\oplus\{l_i : \alpha_i\}_{i \in I}} = \&\{l_i : \bar{\alpha}_i\}_{i \in I} & \overline{\&\{l_i : \alpha_i\}_{i \in I}} = \oplus\{l_i : \bar{\alpha}_i\}_{i \in I} \\ \overline{\alpha\{\beta\}} = \bar{\alpha}\{\bar{\beta}\} & \overline{\mu\mathbf{t}.\alpha} = \mu\mathbf{t}.\bar{\alpha} \\ \overline{\mathbf{t}} = \mathbf{t} & \overline{\mathbf{end}} = \mathbf{end} \end{array}$$

For example, by exchanging input and output, the dual of $\downarrow(\mathbf{string}).\mathbf{end}\{\{\uparrow(\mathbf{bool}).\mathbf{end}\}\}$ is $\uparrow(\mathbf{string}).\mathbf{end}\{\{\downarrow(\mathbf{bool}).\mathbf{end}\}\}$.

3.2 Typing System

Environments. *Typing judgements* of process and expression have the forms:

$$\Gamma \vdash P \triangleright \Delta \qquad \Gamma \vdash e : \theta$$

where Γ is a *service typing*, which typically maps service channels to service type and Δ is a *session typing* which typically maps session channels to session

types. The grammars of typings are given below.

$$\begin{aligned}
(\text{Service Typing}) \quad \Gamma &::= \emptyset \quad | \quad \Gamma, c : \langle \alpha \{\{\beta\}\} \rangle \quad | \quad c : \text{bool} \quad | \quad \Gamma, X : \Delta \\
(\text{Session Typing}) \quad \Delta &::= \emptyset \quad | \quad \Delta, \kappa :_n \alpha \quad | \quad \Delta, (\kappa, \bar{\kappa}) : \alpha[\cdot] \quad | \quad \Delta, (\kappa, \bar{\kappa}) : \perp
\end{aligned}$$

In the service typing, c either has type $\alpha\{\{\beta\}\}$ (a service using a session channel with default behaviour of type α and with a handler of type β) or an atomic type such as `bool`. $X : \Delta$ is used for recursion as in [4].

In session typings, $\kappa :_n \alpha$ says that: *at a polarised session channel κ , there is a session of type α* . The natural number n reveals the presence of a try-catch or a wrap: $n = 0$ means none, $n = 1$ means a try-catch block on s occurs, while $n \geq 2$ means $n - 1$ wraps on s occur. This is needed in the merging with a queue type (whether $n \geq 2$ or not) as well as in the try-catch and wrap typing (whether $n = 0$ or not). When not relevant, n will be omitted. We shall call a session channel with respect to its type *unprotected* if $n = 0$ and *protected* for $n \geq 1$.

$(\kappa, \bar{\kappa}) : \alpha[\cdot]$ and $(\kappa, \bar{\kappa}) : \perp$ are used for typing a queue from κ to $\bar{\kappa}$ (the type of a queue is composed with the type of a process in which case the queue's type becomes \perp). The term $\alpha[\cdot]$ is a *type context* i.e. a type with a hole [12]:

$$\alpha[\cdot] ::= \uparrow(\theta). \alpha[\cdot] \quad | \quad \oplus \{l_i : \alpha_i\}_{i \in I} \cup \alpha[\cdot] \quad | \quad [\cdot]$$

As in a queue there are only output messages, a type context contains only output types. Intuitively, the type of a queue is an incomplete type that needs to be merged with the type of the sending message.

Typing System for Programs. We show the typing system for checking whether a program is error free, in particular w.r.t. its exception usage. The typing rules are given in Table 5. Other than the typing rules for the exception constructs, all rules are identical to [22], augmented with annotation of exception levels. The typing rules directly induce a type checking algorithm for programs.

(TREQ) types a request on service channel c whose type, according to Γ , is $\alpha_j\{\{\beta_j\}\}$. Condition $s^+ = \kappa_j$ makes sure that the fresh name s^+ will also be in the try-catch after reduction. Session s^+ has type $\bar{\alpha}_j\{\{\bar{\beta}_j\}\}$, the dual of c 's type. This rule checks that each κ_i in Q (exception handler) has type $\bar{\beta}_i$ whereas in P it has type $\bar{\alpha}_i\{\{\bar{\beta}_i\}\}$ where each $\bar{\beta}_i$ may come from a refinement of κ_i in P . Finally, Γ' is a subset of Γ without free variables for service channels (otherwise the queue stores open terms at run-time). In (TSERV), because of the service channel principle, services should never be prefixed therefore the only visible (free) session in P and Q should be s^- .

The rules for communication are standard. For (TOUT), in $\uparrow(\theta). \alpha$, the prefixing of a type is read as $(\uparrow(\theta). \alpha')\{\{\beta\}\}$ whenever α has the form $\alpha'\{\{\beta\}\}$. Throwing an exception interrupts any conversation, therefore (TTHR) allows to type **throw**

Table 5 Typing System for Expressions and Processes

(NAME) $\Gamma, a : \langle \alpha \rangle \vdash a : \langle \alpha \rangle$	(BOOL) $\Gamma \vdash \text{tt}, \text{ff} : \text{bool}$	(OR) $\frac{\Gamma \vdash e_i : \text{bool}}{\Gamma \vdash e_1 \text{ or } e_2 : \text{bool}}$
$\frac{\Gamma \vdash P \triangleright \prod_i \kappa_i :_0 \bar{\alpha}_i \{\{\bar{\beta}_i\}\} \quad \Gamma \vdash Q \triangleright \prod_i \kappa_i :_0 \bar{\beta}_i \quad \text{fv}(\Gamma') = \emptyset \quad \Gamma' \subseteq \Gamma \quad \Gamma \vdash c : \langle \alpha_j \{\{\beta_j\}\} \rangle \quad s^+ = \kappa_j}{\Gamma \vdash \bar{c}(s^+)[\bar{\kappa}, P, Q] \triangleright \prod_{i \neq j} \kappa_i :_0 \bar{\alpha}_i \{\{\bar{\beta}_i\}\}}$	$\frac{\Gamma \vdash P \triangleright s^- :_0 \alpha \{\{\beta\}\} \quad \text{fv}(\Gamma) = \emptyset \quad \Gamma \vdash Q \triangleright s^- :_0 \beta}{\Gamma, a : \langle \alpha \{\{\beta\}\} \rangle \vdash *a(s^-)[P, Q] \triangleright \emptyset}$	
(TOUT) $\frac{\Gamma \vdash e : \theta \quad \Gamma \vdash P \triangleright \Delta \cdot \kappa :_0 \alpha}{\Gamma \vdash \kappa! \langle e \rangle, P \triangleright \Delta \cdot \kappa :_0 \uparrow(\theta), \alpha}$	(TIN) $\frac{\Gamma, x : \theta \vdash P \triangleright \Delta \cdot \kappa :_0 \alpha}{\Gamma \vdash \kappa?(x), P \triangleright \Delta \cdot \kappa :_0 \downarrow(\theta), \alpha}$	
(TSEL) $\frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa :_0 \alpha_j}{\Gamma \vdash \kappa \triangleleft l_j, P \triangleright \Delta \cdot \kappa :_0 \oplus \{l_i : \alpha_i\}_{i \in I}}$	(TRES) $\frac{\Gamma, a : \langle \alpha \{\{\beta\}\} \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a) P \triangleright \Delta}$	
(TBRA) $\frac{\Gamma \vdash P_i \triangleright \Delta \cdot \kappa :_0 \alpha_i \quad \forall i \in I}{\Gamma \vdash \kappa \triangleright \{l_i : P_i\}_{i \in I} \triangleright \Delta \cdot \kappa :_0 \& \{l_i : \alpha_i\}_{i \in I}}$	(TIF) $\frac{\Gamma \vdash e : \text{bool} \quad \Delta \text{ unprotected} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \text{if } e \text{ then } P \text{ else } Q \triangleright \Delta}$	
(TINACT) $\frac{\text{fv}(\Gamma) = \emptyset \quad \alpha_i \in \{\text{end}, \text{end}\{\{\beta_i\}\}\}}{\Gamma \vdash \mathbf{0} \triangleright \prod_i \kappa_i :_0 \alpha_i}$	(TREC) $\frac{\Gamma, X : \Delta \vdash P \triangleright \Delta}{\Gamma \vdash \mu X. P \triangleright \Delta}$	(TVAR) $\frac{}{\Gamma, X : \Delta \vdash X \triangleright \Delta}$
(TPAR) $\frac{\Gamma \vdash P_i \triangleright \Delta_i \quad (i = 1, 2) \quad \Delta_1 \times \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \odot \Delta_2}$	(TTHR) $\frac{\text{fv}(\Gamma) = \emptyset}{\Gamma \vdash \text{throw} \triangleright \prod_i \kappa_i :_0 \alpha_i}$	

with any $\kappa : \alpha$ (unprotected). (TINACT) allows to start from $\text{end}\{\{\beta\}\}$ if we are typing in a try-block, while we may want to start from end in a catch-block.

Note that as programs have no try-catch blocks or wraps, any session κ in the typing rules above has $n = 0$ in $\kappa :_n \alpha$.

(TPAR) requires the coherence relation \times and the partial operator \odot based on duality following [10, 18]. When typing programs, the operator becomes just a set union. We shall give its formal definition when typing run-time processes.

Typing System for Run-Time Processes. The rules for typing run-time processes, which are necessary for type soundness, are defined in Table 6. (TEXCEPT) associates the type $\alpha\{\{\beta\}\}$ to the try-catch block, ensuring each κ_i is used as $\alpha_i\{\{\beta_i\}\}$ in P (β_i may come from a refinement) and as β_i in Q . The condition $n > 0$ ensures that each λ_j is used in a try-catch block or a wrap in P . Without this condition we may end up with unprotected code that could be brutally removed by an exception violating the duality of types. As an example, in the process $\text{try}\{\lambda! \langle v \rangle. R\} \text{catch}\{\kappa : Q\} \mid \lambda?(v). R \mid P$, if an exception is thrown by P over κ , the output $\lambda! \langle v \rangle. R$ would be lost leaving the input $\lambda?(v). R$ alone violat-

Table 6 Typing System Extension to Runtime Processes

$$(\text{T}_{\text{EXCEPT}}) \frac{\Gamma \vdash P \triangleright \Delta \cdot \prod_j \lambda_j :_{n_j} \alpha'_j \cdot \prod_i \kappa_i :_{m_i} \alpha_i \{\{\beta_i\}\} \quad \text{queue}(\Delta) \quad \Gamma' \subseteq \Gamma \quad \text{fv}(\Gamma') = \emptyset \quad \Gamma' \vdash Q \triangleright \prod_i \kappa_i :_0 \beta_i \quad n_j > 0}{\Gamma \vdash \text{try}\{P\} \text{ catch } \{\bar{\kappa} : Q\} \triangleright \Delta \cdot \prod_j \lambda_j :_{n_j} \alpha'_j \cdot \prod_i \kappa_i :_1 \alpha_i \{\{\beta_i\}\}} \quad m_i \in \{0, 1\}$$

$$(\text{T}_{\text{SRES}}) \frac{\Gamma \vdash P \triangleright \Delta \cdot s^+ : \alpha \{\{\beta\}\} \cdot s^- : \bar{\alpha} \{\{\bar{\beta}\}\} \cdot (s^+, s^-) : \perp \cdot (s^-, s^+) : \perp}{\Gamma \vdash (vs) P \triangleright \Delta}$$

$$(\text{T}_{\text{WRAP}}) \frac{\Gamma \vdash Q \triangleright \Delta \cdot \prod_i \lambda_i :_{n_i} \alpha'_i \cdot \prod_i \kappa_i :_0 \beta_i \quad \text{queue}(\Delta) \quad n_i > 0}{\Gamma \vdash \bar{\kappa} \{\{Q\}\} \triangleright \Delta \cdot \prod_i \lambda_i :_{n_i} \alpha'_i \cdot \prod_i \kappa_i :_2 \alpha_i \{\{\beta_i\}\}}$$

$$(\text{T}_{\text{QUEUE}}) \frac{\Gamma \vdash L \triangleright \alpha[\cdot] \quad \alpha_i \in \{\text{end}, \text{end}\{\{\beta_i\}\}\}}{\Gamma \vdash \kappa \hookrightarrow_{\phi} \bar{\kappa} : L \triangleright (\kappa, \bar{\kappa}) : \alpha[\cdot] \odot \prod_{i \geq 0} \kappa_i :_0 \alpha_i}$$

ing duality. The other condition $m_i \in \{0, 1\}$ guarantees that κ_i is either refined in another try-catch block or simply used for communication but it is not inside a wrap (this would not make sense).

The predicate $\text{queue}(\Delta)$ checks that Δ contains queue types only. Note there is no need to remove variables from Γ when typing Q because, for terms derived from programs, this is guaranteed by (T_{REQ}) and (T_{SERV}). Finally, in order to record that now there is a try-catch block on κ_i , we update its type with $n = 1$ in $\kappa_i :_n \alpha_i \{\{\beta_i\}\}$.

(T_{WRAP}) types a wrap over a process Q . All the κ_i that have type β_i will have new type $\alpha_i \{\{\beta_i\}\}$ so to form the correct dual for the other side of the session in the case the exception has not been yet received there. Note that in the new type $\kappa_i :_{n_i} \alpha_i \{\{\beta_i\}\}$ of each κ_i , the number n_i is increased to the value 2 in order to remember that there is a wrap. As queues contain only output (and select) messages, queue types will only have output types (and select) [12].

(T_{QUEUE}) types queues and uses a judgement on lists for typing queue elements:

$$\Gamma \vdash L \triangleright \alpha[\cdot]$$

Above, it reads “list L contains messages of type $\alpha[\cdot]$ under the environment Γ ”. The type of the list is then merged with the type end or $\text{end}\{\{\beta_i\}\}$ of any κ_i : this is used to type a try-catch block or a wrap whenever its only nested process is a queue⁶. The rules for typing lists are given in table 7. Most of the rules are standard [12] apart from (Q_{THR}) where the type of any message sent before the throw \dagger is ignored.

⁶ A possible alternative would be to have a weakening rule which uses \odot to weaken any type

Table 7 Typing System for Queue Lists

$$\begin{array}{c}
 \text{(QEMPTY)} \frac{\text{fv}(\Gamma) = \emptyset}{\Gamma \vdash \epsilon \triangleright \cdot} \qquad \text{(QTHR)} \frac{\Gamma \vdash L \triangleright \alpha[\cdot]}{\Gamma \vdash (\dagger :: L) \triangleright \cdot} \\
 \\
 \text{(QVAL)} \frac{\Gamma, v : \zeta \vdash L \triangleright \alpha[\cdot]}{\Gamma, v : \zeta \vdash (v :: L) \triangleright \alpha[\uparrow(\zeta), \cdot]} \qquad \text{(QSEL)} \frac{\Gamma \vdash L \triangleright \alpha[\cdot]}{\Gamma \vdash (l :: L) \triangleright \alpha[\oplus\{l_i : \alpha_i\}_{i \in I} \cup \{l : \cdot\}]}
 \end{array}$$

We conclude by giving the definition of the operators \asymp and \odot using the treatment of queue types from [12]. First, we define an operation for merging queue types with ordinary session types:

Definition 3.1 (Merge).

$$\text{merge}(\alpha[\cdot], \beta, n) = \begin{cases} \alpha[\beta']\{\{\beta''\}\} & \text{if } \beta = \beta'\{\{\beta''\}\} \text{ and } n = 0 \text{ or } n = 1 \\ \beta'\{\{\alpha[\beta'']\}\} & \text{if } \beta = \beta'\{\{\beta''\}\} \text{ and } n = 2 \\ \alpha[\beta] & \text{otherwise } (\beta \text{ not exception type}) \end{cases}$$

When there is a try-catch type, the operator above merges a type context with the correct part (default or exception type) depending on the value of n . We are now able to define the two operators used in (TPAR).

Definition 3.2 (Operator \asymp). We say Δ_1 and Δ_2 are compatible, written $\Delta_1 \asymp \Delta_2$, if and only if

- $\text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset$;
- $\kappa : \alpha, (\kappa, \bar{\kappa}) : \perp, \bar{\kappa} : \beta, (\bar{\kappa}, \kappa) : \perp \in \Delta_1 \cup \Delta_2$ implies $\alpha = \bar{\beta}$;
- $\kappa :_n \alpha, (\kappa, \bar{\kappa}) : \alpha'[\cdot], \bar{\kappa} :_m \beta, (\bar{\kappa}, \kappa) : \beta'[\cdot] \in \Delta_1 \cup \Delta_2$ implies $\text{merge}(\alpha', \alpha, n) = \overline{\text{merge}(\beta', \beta, m)}$;
- $\kappa :_n \alpha, (\kappa, \bar{\kappa}) : \perp, \bar{\kappa} :_m \beta, (\bar{\kappa}, \kappa) : \beta'[\cdot] \in \Delta_1 \cup \Delta_2$ implies $\alpha = \overline{\text{merge}(\beta', \beta, m)}$;

We can finally define \odot whose definition is based on \asymp .

Definition 3.3 (Operator \odot). Given two session environments Δ_1 and Δ_2 , we have that $\Delta_1 \odot \Delta_2$ is defined whenever $\Delta_1 \asymp \Delta_2$ and is such that:

- if $\kappa :_n \beta \in \Delta_i$ and $(\kappa, \bar{\kappa}) : \alpha[\cdot] \in \Delta_i \text{ mod } 2+1$ then $\kappa :_n \text{merge}(\alpha[\cdot], \beta, n), (\kappa, \bar{\kappa}) : \perp \in \Delta_1 \odot \Delta_2$.
- otherwise if $\kappa :_n \beta \in \Delta_i$ then $\kappa :_n \beta \in \Delta_1 \odot \Delta_2$, if $(\kappa, \bar{\kappa}) : \perp \in \Delta_i$ then $(\kappa, \bar{\kappa}) : \perp \in \Delta_1 \odot \Delta_2$ and if $(\kappa, \bar{\kappa}) : \alpha[\cdot] \in \Delta_i$ then $(\kappa, \bar{\kappa}) : \alpha[\cdot] \in \Delta_1 \odot \Delta_2$.

The operation $\Delta_1 \odot \Delta_2$ (defined if $\Delta_1 \asymp \Delta_2$) is such that if $\kappa : \beta$ and $(\kappa, \bar{\kappa}) : \alpha$ are in $\Delta_1 \cup \Delta_2$ then it returns $(\kappa, \bar{\kappa}) : \perp$ as a type for the queue (so we can keep track that the corresponding queue exists) and a new type for κ . The other elements in the

union are the same as $\Delta_1 \odot \Delta_2$. When we create the new type for κ , if $\kappa :_n \alpha \{\{\beta\}\}$ is recorded in Δ_i , we need to merge it with the type of the queue (which contains outputs only) w.r.t. the level of the exception n i.e. if $n = 0$ or $n = 1$ then we merge with α while for $n = 2$ we merge with β .

Example 3.4. In Examples 2.2 and 2.3 all agents can be typed. The type for channel `chSeller` in both examples is $\mu t. \uparrow (\text{int}). \tau \{\{\downarrow (\text{int}). \uparrow (\text{time})\}\}$. In Example 2.3 we also need to type service `chBroker` whose type is $(\downarrow (\text{int}). \mu t. \uparrow (\text{int}). \tau) \{\{\oplus \{l_1 : \downarrow (\text{int}). \uparrow (\text{time}), l_2 : \alpha\}\}\}$ where α is the type of s^- in R_{abort} .

4 Type Safety

This section is about the fundamental behavioural properties of typed processes. In the sequel, we establish some basic properties of well-typed processes such as weakening or strengthening, subjected meta reduction, subject congruence and reduction.

4.1 Basic Properties.

Lemma 4.1 (Weakening). *Let $\Gamma \vdash P \triangleright \Delta$. Then*

- if $X \notin \text{dom}(\Gamma)$ then $\Gamma, X : \Delta' \vdash P \triangleright \Delta$
- if $c \notin \text{dom}(\Gamma)$ then $\Gamma, c : \langle \alpha \rangle \vdash P \triangleright \Delta$
- if $\kappa \notin \text{dom}(\Delta)$ then $\Gamma \vdash P \triangleright \Delta \cdot \kappa :_0 \text{end}$ or $\Gamma \vdash P \triangleright \Delta \cdot \kappa :_0 \text{end}\{\{\beta\}\}$

Proof. Standard, by induction on the derivation tree. □

Lemma 4.2 (Strengthening). *If $\kappa \notin \text{fsc}(P)$ and $\Gamma \vdash P \triangleright \Delta$ then $\Gamma \vdash P \triangleright \Delta \setminus \kappa$.*

Proof. Standard, by induction on the derivation tree. □

Lemma 4.3 (Substitution). *Let $\Gamma \vdash P \triangleright \Delta \cdot \kappa : \alpha$ and let κ' be fresh. Then $\Gamma \vdash P \triangleright \Delta \cdot \kappa' : \alpha$.*

Proof. Standard. □

Lemma 4.4. *Let P be a process such that $\Gamma \vdash P \triangleright \Delta$. Then κ free in P implies κ is in Δ .*

Proof. Straightforward from the definition of free session channel and typing rules. □

Lemma 4.5 (Context). *Let R be a such that:*

- $R \longrightarrow^* R'$
- $R' \equiv (\nu \tilde{r}) (R'' \mid C[\bar{a}(s^+)[\tilde{k}, P, Q]])$
- $\Gamma \vdash C[\bar{a}(s^+)[\tilde{k}, P, Q]] \triangleright \Delta$

where $s^+ \notin \text{fsc}(C)$ and such that $C[-]$ is restriction free. Then $\Gamma \vdash C[\mathbf{try}\{P\} \mathbf{catch} \{\tilde{k} : Q\}] \triangleright \Delta' \cdot \prod_i \kappa_i :_1 \alpha_\kappa$ and $s^+ = \kappa_j$ for some j .

Proof. In order to prove this Lemma, we observe that, by definition of context and by the assumption that $C[-]$ is restriction-free, $C[-]$ may only contain parallel processes, try-catch blocks and wraps.

As a consequence, each κ_i , if different from s^+ , is free in $C[\bar{a}(s^+)[\tilde{k}, P, Q]]$. Hence, by previous Lemma, we have that $\Delta = \Delta' \cdot \prod_{\kappa_i \neq s^+} \kappa_i :_{n_i} \alpha_i$. Therefore, we shall prove that, if $s^+ \notin \text{fsc}(C)$ and $C[-]$ is restriction free then

$$\Gamma \vdash C[\bar{a}(s^+)[\tilde{k}, P, Q]] \triangleright \Delta' \cdot \prod_{\kappa_i \neq s^+} \kappa_i :_{n_i} \alpha_i$$

implies

$$\Gamma \vdash C[\mathbf{try}\{P\} \mathbf{catch} \{\tilde{k} : Q\}] \triangleright \Delta' \cdot \prod_i \kappa_i :_1 \alpha_\kappa$$

and $s^+ = \kappa_j$ for some j . The proof proceeds by induction on the context C . We analyse the key cases:

- $C = -$. In this case, the context is empty. Then by (TR_{EQ}) we get:

$$\Gamma \vdash P \triangleright \prod_i \kappa_i :_0 \bar{\alpha}_i \{\{\bar{\beta}_i\}\} \quad (5)$$

and

$$\Gamma' \vdash Q \triangleright \prod_i \kappa_i :_0 \bar{\beta}_i$$

with $\text{fv}(\Gamma') = \emptyset$, $\Gamma' \subseteq \Gamma$, $\alpha_j = \alpha$, $\beta_j = \beta$ and $s^+ = \kappa_j$. As sessions in (5) are all unprotected, we can then apply (T_{EXCEPT}) and finally obtain

$$\Gamma \vdash \mathbf{try}\{P\} \mathbf{catch} \{\tilde{k} : Q\} \triangleright \prod_i \kappa_i :_1 \bar{\alpha}_i \{\{\bar{\beta}_i\}\}$$

- $C = \mathbf{try}\{C'[-]\} \mathbf{catch} \{\tilde{\lambda} : Q'\}$. By (T_{EXCEPT}), we have that (for $\text{queue}(\Delta_q)$, $n_j > 0$ and $m_i \in \{0, 1\}$):
 - $\Gamma \vdash C'[\bar{a}(s^+)[\tilde{k}, P, Q]] \triangleright \Delta_q \cdot \prod_j \lambda'_j :_{n_j} \alpha'_j \cdot \prod_i \lambda_i :_{m_i} \alpha_i \{\{\beta_i\}\}$
 - $\Gamma' \vdash Q \triangleright \prod_i \kappa_i :_0 \beta_i$

with $\text{fv}(\Gamma') = \emptyset$ and $\Gamma' \subseteq \Gamma$. Note that we assume that s^+ is not in $\tilde{\lambda}'$. By induction hypothesis, as s^+ is not in C' (which is part of C), we have that $\Gamma \vdash C'[\mathbf{try}\{P\} \mathbf{catch}\{\tilde{\kappa} : Q\}] \triangleright \Delta_q \cdot \prod_j \lambda'_j :_{n_j} \alpha'_j \cdot \prod_i \lambda_i :_{m_i} \alpha_i \{\{\beta_i\}\} \cdot s^+ :_1 \bar{\alpha} \{\{\bar{\beta}\}\}$ where if $\lambda_i = \kappa_j$ then $m_i = 1$. Now, by Assumption 1 (cf. Syntax) we know that if $\lambda_i = \kappa_j$ then it must be that $\tilde{\lambda} \subseteq \tilde{\kappa}$ therefore $m_i = 1$ for all i . We can now apply (TEXCEPT), which concludes this case.

- $C = P' | C'[-]$. By (TPAR) we have $\Gamma \vdash P' \triangleright \Delta_1$ and $\Gamma \vdash C'[\bar{\alpha}(s^+)[\tilde{\kappa}, P, Q]] \triangleright \Delta_2$ with $\Delta = \Delta_1 \odot \Delta_2$. By induction hypothesis, the thesis follows.
- $C = \tilde{\lambda} \{C'[-]\}$. As $C[\bar{\alpha}(s^+)[\tilde{\kappa}, P, Q]]$ is part of a process derived from a program, by Assumption 2 (cf. Syntax), we observe that $\tilde{\lambda} \cap \tilde{\kappa} = \emptyset$. The rest of the proof for this case follows by induction hypothesis.

□

Lemma 4.6 (Queue).

1. $\Gamma \vdash (L :: v) \triangleright \beta[\cdot]$ with $\Gamma \vdash v : \zeta$ iff $\Gamma \vdash L \triangleright \alpha[\cdot]$ where $\beta = \uparrow(\zeta). \alpha[\cdot]$.
2. $\Gamma \vdash (L :: l) \triangleright \beta[\cdot]$ iff $\Gamma \vdash L \triangleright \alpha[\cdot]$ where $\beta = \oplus\{l_i : \alpha_i\}_{i \in I} \cup \{l : \alpha[\cdot]\}$.
3. $\Gamma \vdash (L :: \dagger) \triangleright \beta[\cdot]$ iff $\Gamma \vdash L \triangleright \beta[\cdot]$.

Proof. It directly follows from the typing rules (see [12]).

□

4.2 Subject Meta Reduction.

Lemma 4.7. *Let $C[\cdot]$ be a context. Then $C[\mathbf{throw} | Q] \Downarrow (P', S)$ if and only if $C[Q] \Downarrow (P', S)$.*

Proof. Directly from the meta reduction definition.

Lemma 4.8 (Meta Reduction). *Let P be a process derived from a well-typed program and such that*

$$\Gamma \vdash P \triangleright \prod_j \lambda_j :_{n_j} \alpha'_j \{\{\beta'_j\}\} \cdot \prod_i \kappa_i :_0 \alpha_i$$

where $n_j > 0$. If $P \Downarrow (P', S)$ then

$$\Gamma \vdash P' \triangleright \prod_{\lambda_j \in \mathcal{S}} \lambda_j :_2 \beta_j \{\{\beta'_j\}\} \cdot \prod_{\lambda_j \notin \mathcal{S}} \lambda_j :_{n_j} \alpha'_j \{\{\beta'_j\}\}$$

any β_j .

Proof.

The proof proceeds by induction on the meta reduction rules. In the following, we fix $\Delta = \prod_j \lambda_j :_{n_j} \alpha'_j \{\{\beta'_j\}\} \cdot \prod_i \kappa_i :_0 \alpha_i$.

– (MTRY)

$$R \Downarrow (R', T) \Rightarrow \mathbf{try}\{R\} \mathbf{catch} \{\tilde{\kappa}' : Q\} \Downarrow \begin{cases} (R', T) & \text{if } \tilde{\kappa}' \subseteq T \\ (\tilde{\kappa}' \llbracket Q \rrbracket \mid R', T \cup \tilde{\kappa}') & \text{otherwise} \end{cases}$$

From $\Gamma \vdash \mathbf{try}\{R\} \mathbf{catch} \{\tilde{\kappa}' : Q\} \triangleright \Delta$, by rule (TEXCEPT), we have:

$$\Delta = \prod_j \lambda'_j :_{n'_j} \alpha''''_j \llbracket \beta''''_j \rrbracket \cdot \prod_i \kappa'_i :_1 \alpha''_i \llbracket \beta''_i \rrbracket$$

where $n'_j > 0$ (the fact that $n'_j > 0$ ensures that the type of each λ'_j is a try-catch type). Note that we have removed the queue types as the applicability of meta reduction ensures the absence of queues. Also, as $\Delta = \prod_j \lambda_j :_{n_j} \alpha'_j \llbracket \beta'_j \rrbracket \cdot \prod_i \kappa_i :_0 \alpha_i$, we have that $\tilde{\kappa}$ is empty and $\tilde{\lambda}'\tilde{\kappa}' = \tilde{\lambda}$. Therefore, in this case, $\Delta = \prod_j \lambda_j :_{n_j} \alpha'_j \llbracket \beta'_j \rrbracket$ for $n_j > 0$.

Now, by (TEXCEPT), we also have

$$(a) \Gamma \vdash R \triangleright \prod_j \lambda'_j :_{n'_j} \alpha''''_j \llbracket \beta''''_j \rrbracket \cdot \prod_i \kappa'_i :_{m_i} \alpha''_i \llbracket \beta''_i \rrbracket$$

$$(b) \Gamma \vdash Q \triangleright \prod_i \kappa'_i :_0 \beta''_i$$

where $n'_j > 0$, $m_i \in \{0, 1\}$. Note that Q should be typed by removing any free variable in Γ : by weakening, we also have (b).

We can distinguish two cases depending on $\tilde{\kappa}' \subseteq T$:

- $\tilde{\kappa}' \subseteq T$. By applying induction hypothesis to (a), we get

$$\Gamma \vdash R' \triangleright \prod_{\lambda'_j \in T \setminus \tilde{\kappa}'} \lambda'_j :_2 \beta_j \llbracket \beta''_j \rrbracket \cdot \prod_{\lambda'_j \notin T \setminus \tilde{\kappa}'} \lambda'_j :_{n'_j} \alpha''''_j \llbracket \beta''''_j \rrbracket \cdot \prod_i \kappa'_i :_2 \beta''_i \llbracket \beta''_i \rrbracket$$

for any β_j and β''_i . This concludes this case.

- $\tilde{\kappa}' \not\subseteq T$ (note that it cannot be the case that only some of κ'_i are in T because of our assumption on refinement - cf. Syntax, Assumption 1 - and the rules for meta reduction - cf. Semantics). By point (b), we know that $\Gamma \vdash Q \triangleright \prod_i \kappa'_i :_0 \beta''_i$. By (TWRAP), we have

$$\Gamma \vdash \tilde{\kappa}' \llbracket Q \rrbracket \triangleright \prod_i \kappa'_i :_2 \beta''_i \llbracket \beta''_i \rrbracket \quad (6)$$

for any β''_i .

Now, as $\tilde{\kappa}' \not\subseteq T$, we know that κ' does not necessarily have to be in the type of R' . Moreover, by induction hypothesis we get:

$$\Gamma \vdash R' \triangleright \prod_{\lambda'_j \in T} \lambda'_j :_2 \beta_j \llbracket \beta''_j \rrbracket \cdot \prod_{\lambda'_j \notin T} \lambda'_j :_{n'_j} \alpha''''_j \llbracket \beta''''_j \rrbracket$$

Finally, by (TPAR) applied to the above and (6) we can conclude this case.

– (MWRAP)

$$\tilde{\kappa}\{\{Q\}\} \Downarrow (\tilde{\kappa}\{\{Q\}\}, \emptyset)$$

It follows from rule (TWRAP).

– (MNL)

$$R \Downarrow (\mathbf{0}, \emptyset) \quad \text{if } R \in \left\{ \begin{array}{l} (\text{request}), (\text{input}), (\text{output}), (\text{branch}), \\ (\text{select}), (\text{cond}), (\text{recursion}), (\text{throw}) \end{array} \right\}$$

The typing rules for the operators above have always all terms unprotected ($n = 0$).

– (MPAR)

$$P \Downarrow (P', S_1) \text{ and } Q \Downarrow (Q', S_2) \quad \Rightarrow \quad P \mid Q \Downarrow (P' \mid Q', S_1 \cup S_2)$$

This case is simple as we do not have any queue type around, therefore anything on the right hand side cannot appear on the left hand side and viceversa because of the standard linearity property of session types [10].

□

4.3 Type Reduction

Hereby, we introduce a notion of reduction for types also introduced in [12]. Type reduction is a type abstraction of the reduction on processes. First, we analyse a notion of well-formed type.

Definition 4.9 (Well-formedness). *Let Δ be a session environment. We say that Δ is well-formed if whenever κ and $(\kappa, \bar{\kappa})$ are in $\text{dom}(\Delta)$ it follows that $\Delta((\kappa, \bar{\kappa})) = \perp$.*

Lemma 4.10 (Well-formedness and \odot). *If $\Delta = \Delta_1 \odot \Delta_2$ then Δ is well-formed.*

Proof. By contradiction. Let us assume that Δ is not well-formed. This means that κ and $(\kappa, \bar{\kappa})$ are in Δ and $\Delta((\kappa, \bar{\kappa})) = \alpha[-]$ for some κ and $\alpha[-]$. Now, as κ and $(\kappa, \bar{\kappa})$ are in Δ , then it must be the case that they are also in $\Delta_1 \cup \Delta_2$ by definition of \odot . But then, again by definition of \odot , $(\kappa, \bar{\kappa}) : \perp$ has to be in Δ , leading to a contradiction. □

We are now able to define the reduction relation on types based on the notion of well-formedness.

Definition 4.11 (Reduction of Typings). \rightarrow *is the smallest relation on well-formed environments satisfying the rules in Table 8.*

Table 8 Reduction for Types

$$(TR-COM_1) \quad \kappa :_0 \downarrow (\theta). \alpha \cdot (\bar{\kappa}, \kappa) : \uparrow (\theta). \beta[\cdot] \cdot \Delta \quad \rightarrow \quad \kappa :_0 \alpha \cdot (\bar{\kappa}, \kappa) : \beta[\cdot] \cdot \Delta$$

$$(TR-COM_2) \quad \kappa :_0 \downarrow (\theta). \alpha \cdot \bar{\kappa} :_0 \uparrow (\theta). \beta \cdot (\bar{\kappa}, \kappa) : \perp \cdot \Delta \\ \rightarrow \quad \kappa :_0 \alpha \cdot \bar{\kappa} :_0 \beta \cdot (\bar{\kappa}, \kappa) : \perp \cdot \Delta$$

$$(TR-SEL_1) \quad \kappa :_0 \&\{l_i : \alpha_i\}_{i \in I} \cdot (\bar{\kappa}, \kappa) : \oplus\{l_i : \beta_i\}_{i \in I} \cup l_j : \beta_j[\cdot] \cdot \Delta \\ \rightarrow \quad \kappa :_0 \alpha_j \cdot (\bar{\kappa}, \kappa) : \beta_j[\cdot] \cdot \Delta$$

$$(TR-SEL_2) \quad \kappa :_0 \&\{l_i : \alpha_i\}_{i \in I} \cdot \bar{\kappa} :_0 \oplus\{l_i : \beta_i\}_{i \in I} \cdot (\bar{\kappa}, \kappa) : \perp \cdot \Delta \\ \rightarrow \quad \kappa :_0 \alpha_j \cdot \bar{\kappa} :_0 \beta_j \cdot (\bar{\kappa}, \kappa) : \perp \cdot \Delta$$

$$(TR-EXC_1) \quad \prod_i \kappa_i :_0 \alpha_i \cdot \Delta \quad \rightarrow \quad \prod_i \kappa_i :_1 \alpha_i \cdot \Delta$$

$$(TR-EXC_2) \quad \prod_i \kappa_i :_1 \alpha_i \cdot \Delta \quad \rightarrow \quad \prod_i \kappa_i :_2 \alpha_i \cdot \Delta$$

$$(TR-ISO) \quad \frac{\Delta_1 \approx \Delta_2 \quad \Delta_2 \rightarrow \Delta'_2 \quad \Delta'_2 \approx \Delta'_1}{\Delta_1 \rightarrow \Delta'_1}$$

Lemma 4.12 (Well-formedness Preservation). *Let Δ be well-formed and such that $\Delta \rightarrow \Delta'$. Then Δ' is well-formed.*

Proof. By induction on the typing rules. □

Lemma 4.13 (Consistency of Type Reduction with \odot). *Let Δ be well-formed and such that $\Delta = \Delta_1 \odot \Delta_2$ and $\Delta_1 \rightarrow \Delta'_1$. Then $\Delta \rightarrow \Delta'_1 \odot \Delta_2$.*

Proof. By induction on the type reduction rules in Table 8. □

4.4 Subject Congruence and Subject Reduction.

Theorem 4.14 (Subject Congruence).

If $\Gamma \vdash P \triangleright \Delta$ and $P \equiv Q$ then $\Gamma \vdash Q \triangleright \Delta$.

Proof. We proceed by cases:

- a) **try**{ $P \mid \lambda \hookrightarrow_\phi \bar{\lambda} : L$ } **catch** { $\tilde{\kappa} : Q$ } \equiv **try**{ P } **catch** { $\tilde{\kappa} : Q \mid \lambda \hookrightarrow_\phi \bar{\lambda} : L$ } where $\lambda \in \tilde{\kappa} \Rightarrow \dagger \notin L$. We will only consider the case when λ is indeed contained in $\tilde{\kappa}$. The other case is straightforward.

\Rightarrow . Applying rule (TEXCEPT) we have (for $\text{queue}(\Delta_q)$)

$$\Delta = \Delta_q \cdot (\lambda, \bar{\lambda}) : \perp \cdot \prod_j \lambda_j :_{n_j} \alpha'_j \cdot \prod_i \kappa_i :_1 \alpha_i \{\{\beta_i\}\} \quad (7)$$

with $n_j > 0$. Note that the queue type from λ to $\bar{\lambda}$ must be in Δ and it has type \perp because it is well-formed (Lemma 4.16). And, for $m_i \in \{0, 1\}$,

$$\Gamma \vdash P \mid \lambda \hookrightarrow_{\phi} \bar{\lambda} : L \triangleright \Delta_q \cdot (\lambda, \bar{\lambda}) : \perp \cdot \prod_j \lambda_j :_{n_j} \alpha'_j \cdot \prod_i \kappa_i :_{m_i} \alpha_i \{\{\beta_i\}\} \quad (8)$$

$$\Gamma' \vdash Q \triangleright \prod_i \kappa_i :_0 \beta_i \quad (9)$$

with $\Gamma' \subseteq \Gamma$ and $\text{fv}(\Gamma') = \emptyset$. Now, by rule (TPAR) and (8), we can deduce:

$$\Gamma \vdash \lambda \hookrightarrow_{\phi} \bar{\lambda} : L \triangleright (\lambda, \bar{\lambda}) : \alpha''[\cdot] \quad (10)$$

$$\Gamma \vdash P \triangleright \Delta_q \cdot \prod_j \lambda_j :_{n_j} \alpha'_j \cdot \prod_{i \neq j} \kappa_i :_{m_i} \alpha_i \{\{\beta_i\}\} \cdot \kappa_j :_{m_j} \alpha''' \{\{\beta_j\}\} \quad (11)$$

such that, for $\lambda = \kappa_j$, $\alpha_j \{\{\beta_j\}\} = \text{merge}(\alpha''[\cdot], \alpha''' \{\{\beta_j\}\}, m_j)$. From (11) and (9), by applying rule (TEXCEPT), we deduce

$$\Gamma \vdash \mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \triangleright \Delta_q \cdot \prod_j \lambda_j :_{n_j} \alpha'_j \cdot \prod_{i \neq j} \kappa_i :_1 \alpha_i \{\{\beta_i\}\} \cdot \kappa_j :_1 \alpha''' \{\{\beta_j\}\}$$

and finally, by (TPAR) and (10), for $R = \mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \mid \lambda \hookrightarrow_{\phi} \bar{\lambda} : L$,

$$\Gamma \vdash_1 R \triangleright \Delta_q \cdot (\lambda, \bar{\lambda}) : \perp \cdot \prod_j \lambda_j :_{n_j} \alpha'_j \cdot \prod_{i \neq j} \kappa_i :_1 \alpha_i \{\{\beta_i\}\}$$

\Leftarrow . We can inversely follow the steps above.

- b) $\tilde{\kappa}\{[P \mid \bar{\lambda} \hookrightarrow_{\phi} \lambda : L]\} \equiv \tilde{\kappa}\{[P]\} \mid \bar{\lambda} \hookrightarrow_{\phi} \lambda : L \quad (\lambda \notin \tilde{\kappa})$
Trivial, by rules (TWRAP) and (TPAR).
- c) $\tilde{\kappa}\{[P]\} \mid \bar{\kappa}_i \hookrightarrow_{\phi} \kappa_i : L \equiv \tilde{\kappa}\{[P \mid \bar{\kappa}_i \hookrightarrow_{\phi-1} \kappa_i : L]\}$
The proof is similar to the previous case.
- d) $\mathbf{try}\{(\nu a) P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \equiv (\nu a) \mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \quad (a \notin \text{fn}(Q))$
Standard.
- e) $\tilde{\kappa}\{[(\nu a) P]\} \equiv (\nu a) \tilde{\kappa}\{[P]\}$
Standard.

□

Lemma 4.15 (Well-formedness of the Typing System). *If $\Gamma \vdash P \triangleright \Delta$ then Δ is well-formed.*

Proof. It follows by induction on the typing rules. The most relevant cases are (TPAR) and (TQUEUE) where the previous result, Lemma 4.11, plays a key role. \square

Theorem 4.16 (Subject Reduction).

- If $\Gamma \vdash P \triangleright \Delta$ and $P \longrightarrow P'$ then $\Gamma \vdash P' \triangleright \Delta'$ where either $\Delta = \Delta'$ or $\Delta \twoheadrightarrow \Delta'$.
- If $\Gamma \vdash P \triangleright \emptyset$ and $P \rightarrow Q$ then $\Gamma \vdash Q \triangleright \emptyset$.

Proof. We shall prove only the first point as the second one is a straightforward consequence. The proof proceeds by induction on the reduction rules. Note that \twoheadrightarrow is always defined as, by Lemma 4.16, Δ and Δ' are always well-formed. We proceed by cases:

- *Rule (INIT)*

$$\begin{aligned} & *a(s^-)[P, Q] \mid C[\bar{a}(s^+)[\tilde{\kappa}, P', Q']] \longrightarrow \\ & *a(s^-)[P, Q] \mid (\nu s) \left(\begin{array}{l} \mathbf{try}\{P\} \mathbf{catch} \{s^- : Q\} \mid s^- \hookrightarrow_0 s^+ : \epsilon \mid \\ C[\mathbf{try}\{P'\} \mathbf{catch} \{\tilde{\kappa} : Q'\}] \mid s^+ \hookrightarrow_0 s^- : \epsilon \end{array} \right) \end{aligned}$$

We can assume that $C[\cdot]$ has no restrictions. In fact, using structural congruence, these could be taken out and then reduce this case to the restriction case (part of (CON)).

Now, after applying (TPAR), we notice that (TSERV) requires that $*a(s^-)[P, Q]$ has an empty session typing environment. Therefore, we have:

- a) $\Gamma \vdash *a(s^-)[P, Q] \triangleright \emptyset$
- b) $\Gamma \vdash C[\bar{a}(s^+)[\tilde{\kappa}, P', Q']] \triangleright \Delta$

As $C[-]$ is restriction-free and s^+ is not free in $C[-]$, by applying Lemma 4.5 to b), we get:

$$\Gamma \vdash C[\mathbf{try}\{P'\} \mathbf{catch} \{\tilde{\kappa} : Q'\}] \triangleright \Delta' \cdot \prod_i \kappa_i :_1 \bar{a}_i\{\bar{\beta}_i\} \quad (12)$$

Now, from point a) and (TSERV), for $\text{fv}(\Gamma) = \emptyset$, we get:

- i) $\Gamma \vdash P \triangleright s^- :_0 \alpha\{\beta\}$
- ii) $\Gamma \vdash Q \triangleright s^- :_0 \beta$

Now, from (i) and (ii), applying (TEXCEPT) we get

$$\Gamma \vdash \mathbf{try}\{P\} \mathbf{catch} \{s^- : Q\} \triangleright s^- :_1 \alpha\{\beta\} \quad (13)$$

Finally, by (TSRES), (TQUEUE), (12) and (13) we can conclude with

$$\begin{aligned}
& \Gamma \vdash_0 \\
& * a(s^-)[P, Q] \mid (\nu s) \\
& (\mathbf{try}\{P\} \mathbf{catch} \{s^- : Q\} \mid C[\mathbf{try}\{P'\} \mathbf{catch} \{\tilde{\kappa} : Q'\}]) \mid \\
& s^- \hookrightarrow_0 s^+ : \epsilon \mid s^+ \hookrightarrow_0 s^- : \epsilon) \\
& \triangleright \Delta' \cdot \prod_i \kappa_i :_1 \bar{\alpha}_i \{\{\bar{\beta}_i\}\}
\end{aligned}$$

and, as expected, $\Delta \rightarrow \Delta' \cdot \prod_i \kappa_i :_1 \bar{\alpha}_i \{\{\bar{\beta}_i\}\}$.

– *Rule (THR)*

$$\begin{aligned}
& \mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \Downarrow (R, S) \Rightarrow \\
& \mathbf{try}\{\mathbf{throw} \mid P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : L_\kappa \longrightarrow R \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : (\dagger :: L_\kappa)
\end{aligned}$$

Because of Lemma 4.8, we can rewrite the rule as

$$\begin{aligned}
& \mathbf{try}\{\mathbf{throw} \mid P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \Downarrow (R, S) \Rightarrow \\
& \mathbf{try}\{\mathbf{throw} \mid P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : L_\kappa \longrightarrow R \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : (\dagger :: L_\kappa)
\end{aligned}$$

Let us assume that $\Gamma \vdash \mathbf{try}\{\mathbf{throw} \mid P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : L_\kappa \triangleright \Delta$.

By (TPAR) and (TQUEUE), for some Δ_1 and Δ_2 such that $\Delta = \Delta_1 \odot \Delta_2$, we have:

$$\Gamma \vdash \mathbf{try}\{\mathbf{throw} \mid P\} \mathbf{catch} \{\tilde{\kappa} : \tilde{Q}\} \triangleright \Delta_1 \quad (14)$$

$$\Gamma \vdash \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : L_\kappa \triangleright \prod_{\kappa \in S} (\kappa, \bar{\kappa}) : \alpha_\kappa[\cdot] \quad (15)$$

where $\Delta_2 = \prod_{\kappa \in S} (\kappa, \bar{\kappa}) : \alpha_\kappa[\cdot]$. Note that the queue type of $(\kappa, \bar{\kappa})$ is not \perp for each $\kappa \in S$. If that was the case, then κ should also be there (not only as a queue type) and the linearity condition (\asymp) would be violated as κ is also in Δ_1 .

Now, by applying (TEXCEPT) to (14) we have $\Delta_1 = \Delta_q \cdot \prod_j \lambda_j :_{n_j} \alpha'_j \{\{\beta'_j\}\} \cdot \prod_i \kappa_i :_1 \alpha_i \{\{\beta_i\}\}$ with $\mathbf{queue}(\Delta_q)$ and $n_j > 0$ (as $n_j > 0$ then each λ_j must have a try-catch type). Moreover, as we can apply meta reduction, there are no queues in $\mathbf{try}\{\mathbf{throw} \mid P\} \mathbf{catch} \{\tilde{\kappa} : Q\}$, and therefore Δ_q is empty. Hence, $\Delta_1 = \prod_j \lambda_j :_{n_j} \alpha'_j \{\{\beta'_j\}\} \cdot \prod_i \kappa_i :_1 \alpha_i \{\{\beta_i\}\}$.

If we now apply the meta reduction Lemma to (14), noting that by (MTRY) $\tilde{\kappa} \in S$, we get, for any β''_j and β''''_i ,

$$\Gamma \vdash R \triangleright \prod_{\lambda_j \notin S \setminus \tilde{\kappa}} \lambda_j :_{n_j} \beta''_j \{\{\beta'_j\}\} \cdot \prod_{\lambda_j \in S \setminus \tilde{\kappa}} \lambda_j :_2 \alpha'_j \{\{\beta'_j\}\} \cdot \prod_i \kappa_i :_2 \beta''''_i \{\{\beta_i\}\} \quad (16)$$

We choose β_j''' and β_i'''' such that $\Delta(\lambda_j) = \beta_j'''\{\beta_j'\}$ and $\Delta(\kappa_i) = \beta_i''''\{\beta_i\}$ (notice that in Δ the catch types are still β_i and β_j' as there is no wrap on κ_i). Now, by (QTHR) applied to (15) through (TQUEUE), we have

$$\Gamma \vdash \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : \dagger :: L_\kappa \triangleright \prod_{\kappa \in S} (\kappa, \bar{\kappa}) : \cdot \quad (17)$$

Finally, applying (TPAR) to (16) and (17), we obtain:

$$\Gamma \vdash R \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : \dagger :: L_\kappa \triangleright \Delta'$$

where $\Delta' = \prod_{\lambda_j \notin S \setminus \bar{\kappa}} \lambda_j : m_j' \beta_j'''\{\beta_j'\} \cdot \prod_{\lambda_j \in S \setminus \bar{\kappa}} \lambda_j : 2 \alpha_j'\{\beta_j'\} \cdot \prod_i \kappa_i : 2 \beta_i''''\{\beta_i\} \cdot \prod_{\kappa \in S} (\kappa, \bar{\kappa}) : \perp$. Hence $\Delta \rightarrow \Delta'$.

– *Rule (RTHROW)*.

$$\mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \Downarrow (R, S) \Rightarrow$$

$$\begin{aligned} \mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \mid \bar{\kappa}_j \hookrightarrow_0 \kappa_j : (L :: \dagger) \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : L_\kappa \\ \longrightarrow R \mid \bar{\kappa}_j \hookrightarrow_1 \kappa_j : L \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : (\dagger :: L_\kappa) \end{aligned}$$

The proof proceeds similarly to the previous case. Let us assume that $\Gamma \vdash \mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \mid \bar{\kappa}_j \hookrightarrow_0 \kappa_j : (L :: \dagger) \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : L_\kappa \triangleright \Delta$. By applying (TPAR) we have for $\Delta = \Delta_1 \odot \Delta_2$:

$$\Gamma \vdash \mathbf{try}\{P\} \mathbf{catch} \{\tilde{\kappa} : Q\} \triangleright \Delta_1 \quad (18)$$

$$\Gamma \vdash \bar{\kappa}_j \hookrightarrow_0 \kappa_j : (L :: \dagger) \mid \prod_{\kappa \in S} \kappa \hookrightarrow_{\phi_\kappa} \bar{\kappa} : L_\kappa \triangleright \Delta_2 \quad (19)$$

Note that, similarly to the previous case, queue types in Δ_2 are not equal to \perp . If that was the case, then κ_i should be in Δ_2 (not only as a queue type) and the linearity condition (\asymp) would be violated as each κ_i is also in Δ_1 .

Now, by (TEXCEPT) applied to (18), noting there are no queue types in Δ_1 (similarly to the previous case, if we can apply meta reduction then there cannot be any queue), we have

$$\Delta_1 = \prod_j \lambda_j : n_j \alpha_j'\{\beta_j'\} \cdot \prod_i \kappa_i : 1 \alpha_i\{\beta_i\}$$

for $n_j > 0$. Note that, similarly to the previous case, we can assume that each λ_j has a try-catch type because $n_j > 0$. Now, by Meta Reduction Lemma, noting that by (MTRY) $\tilde{\kappa} \in S$, we have:

$$\Gamma \vdash R \triangleright \prod_{\lambda_j \in S \setminus \bar{\kappa}} \lambda_j : 2 \beta_j'''\{\beta_j'\} \cdot \prod_{\lambda_j \notin S \setminus \bar{\kappa}} \lambda_j : n_j \alpha_j'\{\beta_j'\} \cdot \prod_i \kappa_i : 2 \beta_i''''\{\beta_i\} \quad (20)$$

for some β'_j and β'''_i . Similarly to the previous case, we choose them such that $\Delta(\lambda_j) = \beta'_j \{\{\beta'_j\}\}$ and $\Delta(\kappa_i) = \beta'''_i \{\{\beta_i\}\}$.

Now, if we apply the Queue Lemma to (19), we can deduce that the type of the queue from $\bar{\kappa}_j$ to κ_j remains unchanged (by point 3. of the Queue Lemma, removing \dagger does not change the type) and the type of the other queues becomes the type context \cdot by (TQUEUE) and (QTHR).

Finally, applying (TPAR) to the type for the queues and (20) we obtain, similarly to the previous case, the new type Δ' such that $\Delta \rightarrow \Delta'$.

– *Rule (WVAL).*

$$\bar{\kappa}\{\{Q\}\} \mid \bar{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: v) \longrightarrow \bar{\kappa}\{\{Q\}\} \mid \bar{\kappa}_i \hookrightarrow_0 \kappa_i : L$$

This is a key case where we can see the correctness of the reduction mechanism. In fact, as $\phi = 0$, then, by Theorem 2.8, there must be at least one \dagger in L . Hence, the value v has no influence on the queue type, leaving the whole type unchanged.

– *Rule (WTHR).*

$$\bar{\kappa}\{\{Q\}\} \mid \bar{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: \dagger) \longrightarrow \bar{\kappa}\{\{Q\}\} \mid \bar{\kappa}_i \hookrightarrow_1 \kappa_i : L$$

Similar to the previous case.

– *Rule (OUT).*

$$\kappa!\langle e \rangle. P \mid \kappa \hookrightarrow_0 \bar{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_0 \bar{\kappa} : (v :: L) \quad (e \downarrow v)$$

By (TPAR) we have

$$\frac{\Gamma \vdash \kappa!\langle e \rangle. P \triangleright \Delta_1 \quad \Gamma \vdash \kappa \hookrightarrow_0 \bar{\kappa} : L \triangleright \Delta_2 \quad \Delta_1 \asymp \Delta_2}{\Gamma \vdash \kappa!\langle e \rangle. P \mid \kappa \hookrightarrow_0 \bar{\kappa} : L \triangleright \Delta_1 \odot \Delta_2}$$

And by (TOUT) it follows:

$$\Gamma \vdash \kappa!\langle e \rangle. P \triangleright \Delta' \cdot \kappa :_0 \uparrow(\zeta). \alpha$$

and

$$\Gamma \vdash P \triangleright \Delta' \cdot \kappa :_0 \alpha \tag{21}$$

where $\Gamma \vdash e : \zeta$ and $\Delta_1 = \Delta' \cdot \kappa :_0 \uparrow(\zeta). \alpha$. Moreover, by rule (TQUEUE), we have that

$$\frac{\Gamma \vdash L \triangleright \beta[\cdot]}{\Gamma \vdash \kappa \hookrightarrow_0 \bar{\kappa} : L \triangleright (\kappa, \bar{\kappa}) : \beta[\cdot]}$$

Now, by applying (TQUEUE) and (QVAL) to the derivation above we have (assuming $\Gamma \vdash e : \zeta$ implies $\Gamma \vdash v : \zeta$):

$$\frac{\Gamma \vdash L \triangleright \beta[\cdot]}{\Gamma \vdash (v :: L) \triangleright \beta[\uparrow(\zeta). \cdot]}$$

and by applying (TQUEUE) again:

$$\frac{\Gamma \vdash (v :: L) \triangleright \alpha[\uparrow(\zeta). \cdot]}{\Gamma \vdash \kappa \hookrightarrow_0 \bar{\kappa} : (v :: L) \triangleright (\kappa, \bar{\kappa}) : \beta[\uparrow(\zeta). \cdot]} \quad (22)$$

Finally, by (21) and (22) we have (applying (TPAR)):

$$\Gamma \vdash P \mid \kappa \hookrightarrow_0 \bar{\kappa} : (v :: L) \triangleright \Delta_1 \odot \Delta_2$$

– *Rule (IN)*

$$\kappa?(x). P \mid \bar{\kappa} \hookrightarrow_0 \kappa : (L :: v) \longrightarrow P\{v/x\} \mid \bar{\kappa} \hookrightarrow_0 \kappa : L$$

By (TPAR) we have

$$\frac{\Gamma \vdash \kappa?(x). P \triangleright \Delta_1 \quad \Gamma \vdash \bar{\kappa} \hookrightarrow_0 \kappa : (L :: v) \triangleright \Delta_2 \quad \Delta_1 \asymp \Delta_2}{\Gamma \vdash \kappa?(x). P \mid \bar{\kappa} \hookrightarrow_0 \kappa : (L :: v) \triangleright \Delta_1 \odot \Delta_2}$$

And by (TIN) it follows:

$$\Gamma \vdash \kappa?(x). P \triangleright \Delta' \cdot \kappa : \downarrow(\theta). \alpha$$

and

$$\Gamma, x : \zeta \vdash P \triangleright \Delta' \cdot \kappa : \alpha \quad (23)$$

where $\Delta_1 = \Delta' \cdot \kappa : \downarrow(\theta). \alpha$. Moreover, by rule (TQUEUE), we have that

$$\Gamma \vdash \kappa \hookrightarrow_0 \bar{\kappa} : (L :: v) \triangleright (\kappa, \bar{\kappa}) : \beta[\cdot]$$

Now, by Lemma 4.7, we know that $\Gamma \vdash L \triangleright \beta'[\cdot]$ (where $\Gamma \vdash v : \zeta$) and $\beta = \uparrow(\zeta). \beta'[\cdot]$. applying (TQUEUE) again:

$$\frac{\Gamma \vdash L \triangleright \beta'[\cdot]}{\Gamma \vdash \kappa \hookrightarrow_0 \bar{\kappa} : L \triangleright (\kappa, \bar{\kappa}) : \beta'[\cdot]} \quad (24)$$

Finally, by (23) and (24) we have (applying (TPAR)):

$$\frac{\Gamma, x : \zeta \vdash P \triangleright \Delta' \cdot \kappa : \alpha \quad \Gamma \vdash \kappa \hookrightarrow_0 \bar{\kappa} : L \triangleright (\kappa, \bar{\kappa}) : \beta'[\cdot]}{\Gamma \vdash P \mid \kappa \hookrightarrow_0 \bar{\kappa} : L \triangleright \Delta'_1 \odot \Delta'_2} \quad (25)$$

where $\Delta \rightarrow \Delta'_1 \odot \Delta'_2$.

– *Rule (SEL)*

$$\kappa \triangleleft l. P \mid \kappa \hookrightarrow_0 \bar{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_0 \bar{\kappa} : (l :: L)$$

This case proceeds similarly to the (OUT) case.

– *Rule (BRA)*

$$\kappa \triangleright \{l_i : P_i\}_{i \in I} \mid \bar{\kappa} \hookrightarrow_0 \kappa : (L :: l_j) \longrightarrow P_j \mid \bar{\kappa} \hookrightarrow_0 \kappa : L \quad (j \in I)$$

The proof proceeds similarly to the (IN) case.

– *Rule (CLEAN)*

$$\mathbf{try}\{P \mid \lambda \hookrightarrow_\phi \bar{\lambda} : L\} \mathbf{catch} \{\tilde{\kappa} : \tilde{Q}\} \longrightarrow P \mid \lambda \hookrightarrow_\phi \bar{\lambda} : L \quad (\lambda \in \tilde{\kappa}, \dagger \in L)$$

It follows directly from rule (TEXCEPT).

– (CON)

$$P \longrightarrow Q \quad \Rightarrow \quad C[P] \longrightarrow C[Q]$$

By cases on the context $C[-]$. We analyse the most relevant cases only.

- $C[-] = R \mid C'[-]$. In this case, the reduction rule becomes:

$$P \longrightarrow Q \quad \Rightarrow \quad R \mid C'[P] \longrightarrow R \mid C'[Q]$$

If $\Gamma \vdash C[P] \triangleright \Delta$, by (TPAR), we have:

$$\Gamma \vdash R \triangleright \Delta_1 \quad \Gamma \vdash C'[P] \triangleright \Delta_2$$

where $\Delta = \Delta_1 \odot \Delta_2$. By induction hypothesis, we have that, for some Δ'_1 ,

$$\Gamma \vdash C'[Q] \triangleright \Delta'_2$$

with $\Delta_2 \rightarrow \Delta'_2$. Finally, by Lemma 4.14, we have

$$\Delta \rightarrow \Delta' = \Delta_1 \odot \Delta'_2$$

– (IF). Similar to branching.

– (STR). This case follows from the Subject Congruence Lemma.

5 Liveness and Termination

The previous section establishes a basic consistency of dynamics of typed processes via subject reduction (which immediately entails the standard communication safety [10]). This section strengthens this result with a liveness property, which intuitively says that all compensating session channels eventually get fired (except those which are “erased” by thrown exceptions in which case the compensating actions both disappear). Along the way we also show relative termination of individual protocols for exceptions and their partial confluence vis-a-vis ordinary communications, two key consistency criteria for the proposed operational mechanism.

5.1 A Termination Protocol

We first augment the operational semantics in §2 with yet another protocol, called *termination protocol*, which in fact is an intrinsic part of the dynamics of exceptions. As an example, suppose there are two (and only two) processes in a configuration, which are try-catch blocks and which are communicating in a session. If each party’s default process becomes the inaction, it is natural to reduce each try-catch block to the inaction, freeing up the resources for its handler. This “garbage collection” is essential when we consider integration of interactional exception into the standard imperative programming languages with a sequential composition operation, $P;Q$, since in this case launching Q depends on whether P reduces to the inaction or not.

We call a termination of a try-block in this form, *normal exit*. In interactional exceptions, a normal exit of a try-catch block demands an agreement of all peers: even if one try-block has terminated, if any of its communicating peers throws an exception, it also has to throw one, hence synchronising among all peers is essential for consistency. The termination protocol we introduce below makes the most of the tree structure associated with hierarchy of service invocation, leading to relatively efficient execution in terms of the number of messages. The protocol consists of two stages:

(Stage 1: Voting) Each try-catch block notifies its caller in the caller-callee relation of services by sending its vote of termination after itself terminating and receiving the same news from its callees.

(Stage 2: Decision) When the initial caller hears this, it in turn lets the news flow down to all of its callees, upon whose receipt the try-catch blocks can normally terminate.

Above, callees and callers refer to the service invocation mechanism. We formalise this protocol by extending the reduction relation. First, we augment polarities of channels in try-catch blocks with $\{\oplus, \ominus\}$, replacing $+, -$ for Stage 2 (indicating the casting of votes). We also use two special messages, $\{V, D\}$

standing for Voting and Decision. The new rules are reported in Table 9 where

Table 9 Rules for the Termination Protocol

(COLLECT)	$\mathbf{try}\{\star\} \mathbf{catch} \{s'^{-}; \tilde{r}^{\oplus}; s^{+}; \tilde{r}^{+} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : V \rightarrow$ $\mathbf{try}\{\star\} \mathbf{catch} \{s'^{-}; \tilde{r}^{\oplus}; s^{\oplus}; \tilde{r}^{+} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : \epsilon$
(VOTE)	$\mathbf{try}\{\star\} \mathbf{catch} \{s^{-}; \tilde{r}^{\oplus} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : \epsilon \rightarrow$ $\mathbf{try}\{\star\} \mathbf{catch} \{s^{\ominus}; \tilde{r}^{\oplus} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : V$
(ROOT)	$\mathbf{try}\{\star\} \mathbf{catch} \{\tilde{s}^{\oplus} : \tilde{Q}\} \mid \prod_i s_i^{+} \hookrightarrow_{\phi} s_i^{-} : \epsilon \rightarrow \star \mid \prod_i s_i^{+} \hookrightarrow_{\phi} s_i^{-} : D$
(DECISION)	$\mathbf{try}\{\star\} \mathbf{catch} \{s^{\ominus}; \tilde{r}^{\oplus} : \tilde{Q}\} \mid s^{+} \hookrightarrow_{\phi} s^{-} : D \mid \prod_i t_i^{+} \hookrightarrow_{\phi} t_i^{-} : \epsilon \rightarrow$ $\star \mid s^{+} \hookrightarrow_{\phi} s^{-} : \epsilon \mid \prod_i t_i^{+} \hookrightarrow_{\phi} t_i^{-} : D$

$\star ::= \mathbf{0} \mid \star \mid \tilde{k}[\star]$. Briefly, (COLLECT) collects the votes from the callees; (VOTE) sends a vote to the caller once all the callees have voted; (ROOT) is for the initial caller which terminates once all its callees have voted; and (DECISION) terminates once the caller has terminated. The rules only make sense for well-typed processes as we shall show later on this section. We write $P \rightarrow_{term} P'$ for a reduction generated from the above rules.

Example 5.1. As an example, consider the following processes:

$$\mathbf{try}\{\mathbf{0}\} \mathbf{catch} \{s^{+}; t^{+}; r^{+} : Q_1\} \mid \mathbf{try}\{\mathbf{0}\} \mathbf{catch} \{s^{-} : Q_2\} \quad (26)$$

$$\mathbf{try}\{\mathbf{0}\} \mathbf{catch} \{t^{-} : Q_3\} \mid \mathbf{try}\{\mathbf{0}\} \mathbf{catch} \{r^{-} : Q_4\} \quad (27)$$

By (VOTE) the two processes in (27) will put V in the queues with writing side r^{+} and t^{+} respectively. Applying (COLLECT) twice, the right-hand process in (26) will reach the state $\mathbf{try}\{\mathbf{0}\} \mathbf{catch} \{s^{-}; t^{\oplus}; r^{\oplus} : Q_2\}$. Again, by (VOTE), it will send V to the parent. Finally, by (ROOT) and then (DECISION) it will reduce to process $\mathbf{0}$.

Note that for the exit protocol to work, the caller-callee relation must have a tree structure where the root is the initial service invoker and leaves are a collection of interacting processes. We shall explore this topic further in the next subsection.

5.2 Normal and Exceptional Exits

We first show that the above protocol always leads to a normal exit. For this purpose, we need to identify the set of nodes that form the caller-callee relation in a given process.

Definition 5.2 (Node). R is a node process of P whenever $P \equiv (\nu \tilde{s}) (Q \mid R)$, R is restriction/queue/service-free, and does not have the form \star or $R' \mid R''$. If $\text{fsc}(R) \neq \emptyset$ then $\text{fsc}(R_i)$ is called a node of P .

A node process of P is a subprocess of P which is not the parallel composition of two other processes and contains no restriction, queue, service or is composed by zero or more wraps over process $\mathbf{0}$. Then a node is the set of free names (only if non-empty) of a node process. Using the processes (1) and (2) given above, the process (26) | (27) has four nodes: s^+ , (s^-, t^+, r^+) , t^- and r^- . Process $\mathbf{try}\{\kappa!\langle v \rangle. \mathbf{0}\} \mathbf{catch}\{\kappa : Q\}$ has a unique node κ while $\mathbf{try}\{\kappa!\langle v \rangle. \mathbf{0}\} \mathbf{catch}\{\kappa : Q\} | \lambda?(x)$ also has λ .

Lemma 5.3 (Structure of Program-Derived Terms). *Let P be a typable program and $P \rightarrow^* Q$. Then $Q \equiv (\nu \tilde{s}) (Q_{\text{extra}} | \prod_i R_i)$ where Q_{extra} contains queues, replicated services and \star processes, each R_i is a node process and if $s^p \in \text{fsc}(R_i)$ then $s^{\bar{p}} \notin \text{fsc}(R_i)$.*

Proof. We observe:

- by structural congruence we can move restrictions and queues in the position we wish because they are never prefixed;
- we can always group all services in Q_{extra} as, by assumption in Section 2.1, they are always top-level i.e. never nested in a try-catch block or in a wrap.
- by structural congruence, we can always put a process in the form $\prod_i R_i$ such that each R_i cannot be expressed as the parallel composition of its subprocesses.

Therefore, we only need to show that if $s^p \in \text{fsc}(R_i)$ then $s^{\bar{p}} \notin \text{fsc}(R_i)$. Note that this is not always the case for any typable process. For instance we could have $s^+?(x). s^-!\langle e \rangle$ which is typable but both s^+ and s^- are free in it. We can also observe that a process Q derived from programs is always such that $\Delta = \emptyset$ in $\Gamma \vdash Q \triangleright \Delta$. But this is still not enough. In fact, process $Q' = (\nu s) (s^+?(x). s^-!\langle e \rangle | s^+ \hookrightarrow_{s^-} \phi : \epsilon | s^- \hookrightarrow_{s^+} \phi : \epsilon)$ is typable as $\emptyset \vdash Q' \triangleright \emptyset$ but still has the same problem.

In order to prove it, we need to show that given a process Q in the form $(\nu \tilde{s}) (Q_{\text{extra}} | \prod_i R_i)$ such that if $s^p \in \text{fsc}(R_i)$ then $s^{\bar{p}} \notin \text{fsc}(R_i)$, then for any reduction rule we still get something in such a form. Note that the property already holds for programs (no free session channels). We just consider the most interesting cases:

- (INT). In order to apply this reduction we have that $R_j = C[\bar{a}(s^+)[\tilde{k}, P'', Q'']]$ for some j and $*a(s^-)[P', Q']$ is in Q_{serv} . Note that we are assuming that $C[-]$ has no parallel as this would just form other nodes that would not interfere with the reduction therefore not changing the structure. After reduction, s^+ is added to node $\text{fsc}(C[\mathbf{try}\{P''\} \mathbf{catch}\{\tilde{k} : Q''\}])$ and the new node $\{s^-\}$ is created.

- (THR) and (RTHR). In this case wrapped processes are created by these rules (and meta reduction). Therefore node processes do not change as each try-catch is substituted by a corresponding wrap containing exactly the same free names.
- (ROOT) and (DECISION) erase the node \tilde{s}^\oplus and $s^\ominus; \tilde{t}^\oplus$ respectively.

The caller-callee structure of a process is identified by the directed edges of the following graph.

Definition 5.4 (Invocation Graph). *Let G be its set of nodes. Then the invocation graph of process P is the directed graph $\mathcal{G} = (G, E)$ where $(\tilde{\kappa}, \tilde{\lambda}) \in E$ if and only if $\kappa_i \in \{s^+, s^\oplus\}$ and $\lambda_j \in \{s^-, s^\ominus\}$ for some s, i, j . An invocation tree in P is a maximal subtree in the invocation graph.*

The invocation graph of process (26) | (27) is a graph with one edge from s^+ to (s^-, t^+, r^+) , one from (s^-, t^+, r^+) to t^- and one from (s^-, t^+, r^+) to r^- . Recall the definition of programs given in § 2.1. If we reduce a typed program by zero or more steps then the invocation graph of the resulting process always forms a forest.

Lemma 5.5 (Evolution). *Let P be a typable program. If $P \rightarrow^* R$ then R 's invocation graph $\mathcal{G}(R)$ is a forest.*

Proof. The proof exploits the properties of typed programs. The service channel principle coupled with the requirement that services are top-level guarantees that sessions are always between node processes.

We shall prove that given a well-typed process P (derived from a program) whose invocation graph is a forest we have that if $P \rightarrow P'$ for some reduction rule then P' 's invocation graph is still a forest. The proof proceeds by cases on the reduction rules. We only report the interesting cases as the others do not modify the invocation graph.

- (INRT). By definition of node and by Lemma 5.3 above, we have that P has at least one node processes which is $C[\bar{a}(s^+)[\tilde{\kappa}, P'', Q'']]$. Similarly to the previous proof, we assume $C[-]$ has no parallel as this would just define other nodes which do not interfere with the one we are dealing with. The exact number of node processes will depend on the context $C[-]$ if it has any parallel composition. As s has to be fresh, we can assume that it does not appear in the context $C[-]$. After reduction, by definition of node process, we have that one node that is created in the graph is going to be s^- corresponding to **try** $\{P'\}$ **catch** $\{s^- : Q'\}$ (note that by typing P' and Q' have no free names apart from s^-). Then, if $\{s^+\} \subset \tilde{\kappa}$, we are just adding an edge directed from the node associated to the node process containing $\bar{a}(s^+)[\tilde{\kappa}, P'', Q'']$ (now

become **try**{ P' } **catch** { $\tilde{\kappa} : Q'$ } into s^- . In the case $\{s^+\} = \tilde{\kappa}$ then a new node will be created with an edge going into s^- resulting in a new tree in the forest.

- (THR) and (RTHR). In this case wrapped processes are created by these rules (and meta reduction). The forest does not change as each try-catch is substituted by a corresponding wrap containing exactly the same free names.
- (COLLECT) and (VOTE). These rules just change $-$ into \ominus and $+$ into \oplus . By definition of node and invocation graph the topology of the graph does not change resulting into a new isomorphic graph.
- (ROOT) and (DECISION) erase the node \tilde{s}^\oplus and $s^\ominus; \tilde{t}^\oplus$ respectively.

□

From this result, we can prove the following non-interference property of \rightarrow_{term} .

Lemma 5.6 (Partial Confluence of \rightarrow_{term}). *Let P_0 be a typable program and $P_0 \rightarrow^* P$. Then whenever $P \rightarrow_{term} P_1$ and $P \rightarrow P_2$ we have either (1) $P_1 \equiv P_2$ or (2) $P_1 \rightarrow R$ and $P_2 \rightarrow_{term} R$ for some R .*

An invocation tree in P is in the *pretermination state* if each of its try-blocks have the form \star . The following theorem is a corollary of Lemma 5.6 and says that once an invocation tree reaches a pretermination state then all of its nodes will eventually vanish.

Theorem 5.7 (Normal Exit). *Let P_0 be typable program and $P_0 \rightarrow^* P$ such that P has an invocation tree \mathcal{T} in the pretermination state. Then whenever $P \rightarrow^* P'$ there is Q such that $P' \rightarrow_{term}^* Q$ where Q does not contain any active nodes from \mathcal{T} .*

The result above can actually be made stronger. For instance, in **try**{**try**{ $\mathbf{0}$ } **catch** { $\lambda : Q_1$ }} **catch** { $\kappa : Q_2$ }, we do not have a pretermination state as the outer try-block contains a try-catch block. Nevertheless, the normal exit is guaranteed as when the inner block and the subtree connected to λ terminate, the outer catch block can proceed with the termination protocol.

A try-catch block can also exit by an exception. Note this form of exit is also propagated through invocation graphs. The propagation also terminates up to the termination of other processes. We write $P \rightarrow_{ex} P'$ if this is generated from (RTHR)-reduction rule (which propagates exception) and we say that κ is in *preexception* if it is the exception channel for a try-block which moreover contains an active **throw**.

Theorem 5.8 (Exception Exit). *Let P_0 be a typable program and $P_0 \rightarrow^* P$ such that P has an invocation tree \mathcal{T} . Suppose $\tilde{\kappa}$ is in a node of \mathcal{T} which is in preexception for some κ_i . Then $P \rightarrow^* P'$ implies $P' \rightarrow_{ex}^* R$ for some R .*

5.3 Liveness

We can use the Evolution Lemma to obtain a strong form of liveness for well-typed processes in the presence of asynchronous exceptions. We first define:

Definition 5.9. *We say P is stable if P is the parallel composition of zero or more \star processes and zero or more empty queues, possibly under v -restrictions. We say P has all resources if $a \in \text{fn}(P)$, then $P \equiv (v\bar{u}) (*a(\lambda)[Q_1, Q_2] \mid R)$, i.e. all output channels are compensated by replicated inputs.*

We can finally state that a well-typed program either continues to reduce for ever or reaches a state which does not contain active prefixes (except replicated services), try-catch blocks, throws nor messages in transit. This theorem implies that any stuck of a typed process is not due to session communications.

Theorem 5.10 (Liveness). *Suppose P is a typable program and has all resources. Then $P \rightarrow^* Q$ implies either Q is stable or $Q \rightarrow Q'$ for some Q' .*

Proof. The proof exploits all results shown before including Subject Reduction. The tree structure of the invocation graph plays a key role: this tree form ensures that we never have the a deadlock such as $s^{-?}(x), t^{+!}(x) \mid t^{-?}(v), s^{+!}(y)$. Since all runtime processes are non-blocking apart from the input, the proof reduces to showing that deadlock does not happen. As a consequence of types, at least one node in the invocation graph (forest) must be performing an output. Since outputs are not blocking by operational semantics, we conclude the proof.

By Lemma 5.3, we know that $Q \equiv (v\bar{s}) (Q_{\text{extra}} \mid \prod_i R_i)$ where Q_{extra} contains message queues, services and \star processes only and each R_i is a node process such that if $s^p \in \text{fsc}(R_i)$ then $s^{\bar{p}} \notin \text{fsc}(R_i)$.

In the sequel, we say a process Q' is active in R_i whenever $R_i \equiv C[Q']$ for some reduction context $C[-]$ and Q' is not (try-catch) or (wrap). Obviously, given the shape of each R_i , the context $C[-]$ will only be a nesting of try-catch blocks and wraps and processes have a unique active process i.e. if Q' and Q'' are two active processes of R_i then $Q' = Q''$.

Let us now assume, by contraddiction, that $Q \not\rightarrow$ and Q is not stable. If this is the case, then we observe that each R_i cannot have an active process which is:

- (output) or (select). An output (or a select) is never blocking as queues are never prefixed and can always receive messages;
- (cond). A conditional is never blocking as far as terms contain no free variable and this is the case with programs;
- (throw). Throws must be inside a try-block (by assumption in Section 2.1) and they can always ignite an exception by reduction rule (THR);

- (request). We are assuming that service requests are compensated by replicated inputs (services) and replicated inputs cannot be in a process node (by definition) and never nested (assumption in Section 2.1) therefore always available in Q_{extra} allowing for reduction.

Therefore, we have that:

R_i 's active process must be (input) or (inact).

By Evolution Lemma, the set of node processes R_i must be such that their induced graph is a forest. Note that by typing (as $\Gamma \vdash Q \triangleright \emptyset$) each R_i generates a node (has free names) as its active process must be either an input or the inactive process inside a try-catch block (with some possible wraps).

From the reasoning above we have that for some κ_i , R_i 's active process is either an (inact) or has the form:

$$\kappa_i?(x_i). R'_i$$

Note that some R_i could also have a branch rather than an input, but the proof is similar therefore we assume there are only normal inputs instead.

Now, by Evolution Lemma, the invocation graph of Q is a tree (without loss of generality we assume there is one single tree as the proof trivially extends to the forest case). Let us now pick a node, say $\text{fsc}(R_j)$, which is a leaf. We then have to reason by cases:

- $R_j = C[\mathbf{0}]$. As $\text{fsc}(R_j)$ is a leaf then $\text{fsc}(R_j) = \{s^-\}$ for some s . Therefore, by typing, we have that $R_j = \mathbf{try}\{\mathbf{0}\} \mathbf{catch} \{s^- : R'_j\}$ for some R'_j . But if this is the case, then we can use rule (VOTE) having a reduction. As it contradicts our assumption of deadlock, this cannot be the case.
- $R_j = C[s^-?(x_j). R'_j]$. As we are assuming there is no reduction, then it must be that the queue from s^+ to s^- is empty. Now, by typing, in another node process, there must be an output at s^+ that matches this input. Moreover, as there is deadlock, this must be prefixed by an input (there could be other inputs and outputs in between). Let us assume that this is node process $R_k = \kappa_k?(x_k). R'_k$. Now, as there is deadlock, the queue from $\bar{\kappa}_k$ to κ_k must be empty. Now, we have two cases depending on the polarity of κ_k .
 - If the polarity is positive then node $\text{fsc}(R_k)$ must have another son containing $t^- = \bar{\kappa}_k$. Now, by typing, the associated node process must have an output at t^- which is blocked i.e. it must be prefixed (there could be other inputs and outputs in between) by an input at some other channel (with positive polarity because we are in a tree). Inductively, by this reasoning, we must reach a leaf where there must be an output which

is not blocked contradicting our assumption. Therefore, this cannot be the case if we have deadlock.

- If the polarity is negative then node $\text{fsc}(R_k)$ must have a father with an outgoing edge from t^+ . By typing, the associated node process, must have an output at channel t^+ . As we are assuming deadlock, then it must be prefixed by an input at some other channel (there could be other inputs and output in between). If the polarity of such channel is positive then we go back to the previous case. If the polarity is negative then, by typing there must be the same channel with negative polarity on another node performing an output. Inductively, we must reach the root of the tree where we shall find an output on a negative channel which is not prefixed. Therefore, this cannot be the case and as it exhausts all cases, deadlock is not possible.

□

In particular, if there are two compensating actions in a session and both peers are not interrupted by an exception, then these two will eventually interact, attesting the consistency of exception protocols. Practically, observing that the service channel principle [4, 17] (the key reason the result holds) is widely found in e.g. services in world wide web, the result says that if a conversation ever gets stuck in such an environment, it may as well be for non-interactive reasons (such as deadlock over shared resources at the server), giving clear delineation of the involved engineering issues.

6 Related Work and Conclusion

Extending the session type discipline to asynchronous, interactional exceptions requires the development of the fine-grained operational structures and a refined type discipline, as well as their consistent integration. The standard branching/s-election types or traditional (local) exceptions are not sufficient for expressing interactional exceptions, as discussed through examples in the preceding sections. The presented framework allows arbitrarily nested exceptions to occur asynchronously at each endpoint, while ensuring session compatibility, even in the presence of simultaneous escape. We have shown that typable processes with interactional exceptions satisfy a strong liveness property in the presence of protocols for exceptions, whose analogue has not been proved in the previous typing systems for session types.

6.1 Further Results

For simplicity of presentation, we have restricted so far session types so that, in $\alpha\{\beta\}$, we do not allow β to contain try-catch types. Operationally this means that, in $*c(\lambda)[P, Q]$, the handler Q does not contain another try-catch at the same λ . In fact, allowing arbitrarily nested handlers with the same session channel demands only minor extensions in the syntax and typing. First we allow Q in $*c(\lambda)[P, Q]$ to be a try-catch block at λ , including in programs. Second, try-catch types are extended by the following induction: (a) $\alpha\{\beta\}$ with α and β plain is a try-catch type; and (b) $\alpha\{\beta\}$ with α plain and β a try-catch type, is a try-catch type. Third, n in $\kappa :_n \alpha$ becomes arbitrary natural numbers, generalising their composition with queue types. With essentially the same operational semantics, this generalised calculus satisfies the subject reduction and liveness we stated for the restricted case.

Further generalisation is possible with respect to various extensions of session types, including parallel session types [4], multiparty session types [12], and unordered asynchronous communications [9]. In the last case, instead of annotating queues with levels, we annotate each asynchronous message with its level, which is incremented as it goes out of its source and decremented as it goes into its target through a wrap.

6.2 Related Work

In concurrent programming of distributed objects, exception handling is investigated in [21] where an algorithm to resolve multiple kinds of exceptions (which form a linear order) among concurrently running objects is proposed. Asynchronous exceptions among concurrent threads and their interplay with states in Haskell is studied in [16]. Motivated by subtle race conditions for mutual states, they formalise and implement blocking constructs to postpone asynchronous

exceptions. The key idea is to relax the exception mask through the use of interruptible operations, to balance asynchrony and state consistency. [2] introduces a model for long-running transactions which treats failures by restoring the initial state and firing a compensation process. The calculus for web services called COWS [15] provides an operation to kill processes except those protected by wraps similar to our exception mechanism. [19] introduces a calculus for service orientations by extending the π -calculus, equipped with service and request primitives and local exceptions, without asynchronous queues. An interesting idea is *context*, a named tree-like structure where a process is located. They do not have an explicit notion of a session nor its type abstraction. Their exceptions are the traditional local exception, without supporting propagation of exceptions, coordinated transfer to a different part of a dialogue, nor the associated type abstraction, so that type checking protocols with exceptions, such as Examples 1/2 in §2, would be difficult.

The central focus of the present work is to have basic high-level typed abstractions for clear and flexible descriptions of conversation structures, separating abstraction concerns and their underlying mechanisms. Our exceptions are asynchronously raised by multiple communicating peers, for which the session-based abstraction and compatibility can guarantee type-safety in the presence of arbitrarily nested exceptions. These key aspects, backed up by safety and liveness properties relying on linearity of session-based communications, have not been investigated in the existing studies.

6.3 Further Topics

The key idea of the presented operational semantics is the use of exception levels in queues and their interplay with wrapped processes. In implementation, the queue level can be recorded in a header of each message which its receiver can check efficiently. The wrapping level can be part of a process state, recording its exception depth. Various optimisations are possible, for example dispensing with most coordination protocols when the handler type is trivial, obtaining essentially the same level of efficiency as local exception. In the near future, we plan to incorporate this exception mechanism to our on-going implementation of Java with session types [13].

For simplicity, we omit session delegations: we formulated this extension by storing frozen processes in queues. The type soundness holds by incorporating the present one with the typing rules in [10]. However a construction of the invocation graphs which can guarantee forest structures for the liveness property is left open.

Our liveness property, which involves the termination protocols, is similar to the property as found in e.g. [5]. Apart from presence of exceptions, the aims and

approaches of the two works are quite different: in the present work, the liveness is used for ensuring consistency of the proposed exception mechanisms, and the proof method is more operational, being applicable to any session-based calculus which has the service channel principle without delegation, without changing its typing system. On the other hand, in [5], an effect-based typing system is used for progress with delegation. It is an interesting further topic to incorporate our typing system with [5] for obtaining liveness with both exceptions and delegations.

A significant future topic is the treatment of multiple kinds of exception types in the present framework. Following [21], we may assume the ordering on exception types (such as the exception class hierarchy as found in Java) for coordinating exceptions among multiple peers. Using a similar technique developed in our termination protocol, we can exploit the tree-structure of an invocation graph for efficient resolution of exception types to handle the subtle interplay between multiple exception types and nested exceptions for a more refined exception propagation.

With session exceptions, many practical communication scenarios can be accurately described through session primitives, and type-checked by the session type theory. The syntax and type structures developed in this paper are being considered for the use in a Web Services language (WS-CDL [4, 20]) and a language of message schemes for financial communications (ISO 20022 UNIFI [14]), throughout our collaborations with the industry partners.

References

1. AMQP Advanced Message Queuing Protocol. <http://www.ionas.com/opensource/amqp/>.
2. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *FMOODS'03*, LNCS, pages 124–138. Springer, 2003.
3. E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In *TGC07*, LNCS. Springer, 2007.
4. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
5. M. Dezani-Ciancaglini, U. de'Liguoro, and N. Yoshida. On Progress for Structured Communications. In *TGC07*, LNCS. Springer, 2007.
6. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, , and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In W. Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.
7. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
8. S. Gay and V. T. Vasconcelos. Asynchronous functional session types. TR 2007–251, University of Glasgow, may 2007.
9. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, volume 512, pages 133–147, 1991.

10. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
11. K. Honda, N. Yoshida, and M. Carbone. Web Services, Mobile Processes and Types. *The Bulletin of the European Association for Theoretical Computer Science*, February(91):165–185, 2007.
12. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284, 2008.
13. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOOP'08*, LNCS. Springer, 2008. To appear.
14. International Organization for Standardization ISO 20022 UNiversal Financial Industry message scheme. http://www.iso20022.org/index.cfm?item_id=56664#interest.
15. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *ESOP '07*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
16. S. Marlow, S. L. P. Jones, A. Moran, and J. H. Reppy. Asynchronous exceptions in Haskell. In *PLDI*, pages 274–285. ACM, 2001.
17. D. Mostrous and N. Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA'07*, LNCS. Springer-Verlag, 2007.
18. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
19. H. Vieira, L. Caires, and J. Seco. The conversation calculus: A model of service oriented computation. In *ESOP'08*, LNCS. Springer, 2008.
20. Web Services Choreography Working Group. <http://www.w3.org/2002/ws/chor/>.
21. J. Xu, A. B. Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1019–1032, 2000.
22. N. Yoshida and V. T. Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In *SecRet'06*, ENTCS. Elsevier, 2007.